Mastering Ethereum

Andreas M. Antonopoulos, Gavin Wood

Table of Contents

LICENSE . . . 1
Předmluva . . . 3
- Jak používat tuto knihu . . . 3
- Zamýšlené publikum . . . 4
- Konvence použité v této knize . . . 4
- Ukázky zdrojových kódů . . . 5
- Použití příkladů kódu . . . 5
- Odkazy na společnosti a produkty . . . 6
- Ethereum adresy a transakce v této knize . . . 6
- Jak nás kontaktovat . . . 6
  - Kontaktování Andrease . . . 7
  - Kontaktování Gavina . . . 7
- Poděkování Andrease . . . 7
- Poděkování Gavina . . . 7
- Přispěvatelé . . . 8
- Edice otevřeného překladu . . . 14
- Zdroje . . . 16

Slovníček pojmů . . . 19
Co je Ethereum? . . . 33
- Srovnání s Bitcoinem . . . 33
- Součásti bločenky . . . 34
- Zrození Etherea . . . 35
- Čtyři etapy vývoje Etherea . . . 37
- Ethereum: bločenka pro univerzální účely . . . 38
- Součásti Etherea . . . 38
  - Další čtení . . . 40
- Ethereum a Turingovská úplnost . . . 40
  - Turingová úplnost jako „vylepšení“ . . . 41
  - Důsledky Turingovské úplnosti . . . 41
- Od univerzální bločenky po decentralizované aplikace (DApps) . . . 42
- Třetí věk internetu . . . 43
- Kultura vývoje Etherea . . . 44

Proč se učit Ethereum? . . . 45
Co vás tato kniha naučí . . . 45
Základy Etherea . . . 47
Měnové jednotky Etherea . . . 47
Výběr Ethereum peněženky . . . 48
Kontrola a odpovědnost . . . 49
Začínáme s MetaMaskem . . . 51
Vytvoření peněženky . . . 52
Přepínání sítí . . . 55
Získání testovacího éteru . . . 56
Odesíláme ether z MetaMasku . . . 58
Zkoumání transakční historie adresy . . . 60
Představujeme světový počítač . . . 63
Externě vlastněné účty (EOA) a kontrakty . . . 63
Jednoduchá smlouva: Testovací Ethereum kohoutek . . . 64
Kompilace kontraktu Kohoutek . . . 68
Vytvoření kontraktu na bločence . . . 70
Interakce s kontraktem . . . 73
Zobrazení adresy kontraktu v prohlížeči bloků . . . 73
Zaslání financí kontraktu . . . 75
Výběr financí z našeho kontraktu . . . 76
Závěry . . . 80
Ethereum klienti . . . 83
Sítě Etherea . . . 83
Mám provozovat úplný uzel? . . . 84
Výhody a nevýhody úplného uzlu . . . 85
Výhody a nevýhody veřejné testovací sítě . . . 86
Výhody a nevýhody místní simulace bločenky . . . 86
Provozování Ethereum klienta . . . 87
Hardwarové požadavky pro úplný uzel . . . 87
Softwarové požadavky na zprovoznění klienta (uzel) . . . 89
Parity . . . 90
Instalace Parity . . . 91
Go-Ethereum (Geth) . . . 93
Okopírování úložiště . . . 93

Sestavení Geth ze zdrojového kódu . . . 94
První synchronizace bločenek založených na Ethereu . . . 95
Spuštění Geth nebo Parity . . . 96
Rozhraní JSON-RPC . . . 96
Parity režim kompatibility s Geth . . . 99
Ethereum klienti pro vzdálený přístup . . . 99
Mobilní peněženky pro chytré telefony . . . 100
Webové peněženky . . . 101
MetaMask . . . 101
Jaxx . . . 102
MyEtherWallet (MEW) . . . 102
MyCrypto . . . 102
Mist (zastaralá) . . . 103
Závěry . . . 103
Kryptografie . . . 105
Klíče a adresy . . . 105
Kryptografie veřejného klíče a kryptoměna . . . 106
Soukromé klíče . . . 108
Tvorba soukromého klíče z náhodného čísla . . . 109
Veřejné klíče . . . 110
Vysvětlení kryptografie eliptických křivek . . . 111
Aritmetické operace na eliptické křivce . . . 117
Vypočtení veřejného klíče . . . 118
Knihovny eliptických křivek . . . 121
Kryptografické hashovací funkce . . . 121
Ethereum kryptografická hašovací funkce: Keccak-256 . . . 123
Kterou hašovací funkci používám? . . . 123
Ethereum adresy . . . 124
Formáty Ethereum adresy . . . 125
Mezibankovní protokol klientských účtů . . . 126
Hexadecimální kódování s kontrolním součtem pomocí velikostí písmen (EIP-55) . . . 128
Zjištění chyby v adrese zakódované EIP-55 . . . 130
Závěry . . . 131
Peněženky . . . 133
Přehled technologií peněženky . . . 133

Nedeterministické (náhodné) peněženky . . . 135
Deterministické (semínkové) peněženky . . . 137
Hierarchické deterministické peněženky (BIP-32 / BIP-44) . . . 137
Semínka a mnemotechnické kódy (BIP-39) . . . 139
Doporučené postupy pro práci s peněženkou . . . 140
Mnemotechnická kódová slova (BIP-39) . . . 140
Generování mnemotechnických slov . . . 141
Převod mnemotechnického kódu na semínko . . . 144
Volitelná přístupová fráze v BIP-39 . . . 147
Práce s mnemotechnickými kódy . . . 148
Vytváření HD peněženky ze semínka . . . 149
HD peněženky (BIP-32) a jejich vnitřní struktura (BIP-43/44) . . . 150
Rozšířené veřejné a soukromé klíče . . . 150
Odvození tvrzených dětských klíčů . . . 152
Číslování potomků pro normální a tvrzené odvození . . . 152
Cesta ke klíči v HD peněženkách . . . 153
Navigace ve stromové struktuře HD peněženky . . . 153
Závěry . . . 155
Transakce . . . 157
Struktura transakce . . . 157
Transakční nonce . . . 158
Sledování noncí . . . 160
Mezery v noncích, stejné nonce a potvrzení . . . 162
Souběh, vytvoření transakce a nonce . . . 163
Transakční plyn . . . 164
Příjemce transakce . . . 167
Transakční hodnota a data . . . 167
Přenos hodnoty ve prospěch EOA a kontratu . . . 170
Přenos užitečného datového zatížení EOA nebo kontraktu . . . 170
Zvláštní transakce: Vytvoření kontraktu . . . 172
Digitální podpisy . . . 177
Algoritmus digitálního podpisu pomocí eliptické křivky . . . 178
Jak fungují digitální podpisy . . . 178
Vytvoření digitálního podpisu . . . 178
Ověření podpisu . . . 180

Matematika ECDSA . . . 180
Podepisování transakcí v praxi . . . 182
Tvorba a podpisování surových transakcí . . . 183
Tvorba surové transakce dle EIP-155 . . . 185
Hodnota předpony podpisu (v) a obnova veřejného klíče . . . 186
Oddělení podpisu a přenosu (offline podpis) . . . 187
Propagace transakcí . . . 189
Zaznamenání na bločence . . . 190
Transakce s více podpisy (vícepodpisové) . . . 190
Závěry . . . 191
Chytré kontrakty a Solidity . . . 193
Co je to chytrý kontrakt? . . . 193
Životní cyklus chytrého kontraktu . . . 194
Úvod do vysokoúrovňových Ethereum jazyků . . . 195
Tvorba chytrého kontraktu pomocí Solidity . . . 197
Výběr verze Solidity . . . 198
Stáhnutí a instalace . . . 198
Vývojové prostředí . . . 199
Psaní jednoduchého programu v Solidity . . . 199
Kompilace pomocí Solidity kompilátoru (solc) . . . 200
ABI Ethereum kontraktu . . . 201
Výběr Solidity kompilátoru a verze jazyka . . . 202
Programování v Solidity . . . 204
Datové typy . . . 204
Předdefinované globální proměnné a funkce . . . 206
Kontext transakce / volání zprávy . . . 206
Kontext transakce . . . 207
Kontext bloku . . . 207
adresa objektu . . . 208
Vestavěné funkce . . . 209
Definice kontraktu . . . 209
Funkce . . . 210
Konstruktor kontraktu a samozničení . . . 212
Přidání konstruktoru a sebezničení do našeho příkladu kohoutku . . . 213
Modifikátory funkcí . . . 215

Dědičnost kontraktu. . . . 216
Zpracování chyb (assert, require, revert) . . . 218
Události . . . 220
Zachytávání událostí . . . 223
Volání dalších kontraktů (send, call, callcode, delegatecall) . . . 225
Vytvoření nové instance. . . . 225
Adresování existující instance . . . 227
Call, delegatecall. . . . 228
Úvahy o plynu . . . 234
Vyhněte se dynamické velikosti polí . . . 234
Vyhněte se volání jiných kontraktů . . . 234
Odhad nákladů na plyn. . . . 234
Závěry . . . 237
Chytré kontrakty a Vyper. . . . 239
Zranitelnosti a Vyper . . . 239
Srovnání se Solidity . . . 239
Modifiers. . . . 240
Dědičnost třídy . . . 242
Vložený asembler . . . 242
Přetěžování funkce . . . 242
Přetypování proměnných. . . . 243
Počáteční a koncové podmínky. . . . 245
Dekorátory . . . 245
Funkce a pořadí proměnných . . . 246
Kompilace . . . 248
Ochrana proti chybám přetečení na úrovni kompilátoru. . . . 249
Čtení a zápis dat. . . . 250
Závěry . . . 250
Bezpečnost chytrých kontraktů . . . 253
Doporučené postupy zabezpečení . . . 253
Bezpečnostní rizika a antivzory . . . 254
Opětovné zavolání . . . 255
Zranitelnost . . . 255
Peventivní techniky . . . 259
Skutečný příklad: The DAO . . . 260

Aritmetické podtečení / přetečení . . . 261
Zranitelnost . . . 261
Peventivní techniky . . . 264
Příklady ze života: PoWHC a přetečení dávkového přenosu (CVE-2018–10299) . . . 266
Neočekávaný ether . . . 266
Zranitelnost . . . 266
Peventivní techniky . . . 270
Další příklady . . . 272
DELEGATECALL . . . 272
Zranitelnost . . . 272
Peventivní techniky . . . 277
Příklad ze života: Parity vícepodpisová peněženka (druhé napadení) . . . 278
Výchozí viditelnost . . . 280
Zranitelnost . . . 280
Peventivní techniky . . . 281
Příklad ze života: Parity vícepodpisová peněženka (první napadení) . . . 281
Iluze entropie . . . 283
Zranitelnost . . . 283
Peventivní techniky . . . 283
Příklad ze života: PRNG kontrakt . . . 284
Externí odkazy na kontrakty . . . 284
Zranitelnost . . . 284
Peventivní techniky . . . 288
Příklad ze života: Hrnec medu s vícenásobným voláním . . . 289
Krátká adresa / útok na parametry . . . 291
Zranitelnost . . . 292
Peventivní techniky . . . 293
Nekontrolované návratové hodnoty CALL . . . 293
Zranitelnost . . . 294
Peventivní techniky . . . 294
Příklad ze života: Etherpot and King of the Ether . . . 295
Souběh / předbíhání . . . 297
Zranitelnost . . . 297
Peventivní techniky . . . 298
Příklady ze života: ERC20 a Bancor . . . 299

Odepření služby (DoS) . . . 300
Zranitelnost . . . 300
Peventivní techniky . . . 302
Příklady ze života: GovernMental . . . 303
Manipulace s časovou značkou bloku . . . 303
Zranitelnost . . . 304
Peventivní techniky . . . 305
Příklad ze života: GovernMental . . . 305
Konstruktory s péčí . . . 306
Zranitelnost . . . 306
Peventivní techniky . . . 307
Příklad ze života: Rubixi . . . 307
Neinicializované ukazatele úložiště . . . 307
Zranitelnost . . . 308
Peventivní techniky . . . 310
Příklad ze života: OpenAddressLottery a CryptoRoulette hrnce medu . . . 311
Plovoucí desetinná čárka a přesnost . . . 311
Zranitelnost . . . 311
Peventivní techniky . . . 312
Příklad ze života: Ethstick . . . 313
Tx.Origin ověřování . . . 313
Zranitelnost . . . 314
Peventivní techniky . . . 315
Knihovny kontraktů . . . 316
Závěry . . . 317
Tokeny . . . 319
Jak se používají tokeny . . . 319
Tokeny a zaměnitelnost . . . 321
Riziko protistrany . . . 321
Tokeny a vnitřnost . . . 322
Používání tokenů: užitek nebo majetkový podíl . . . 322
Je to kachna! . . . 323
Ethereum tokeny . . . 325
Standard tokenu ERC20 . . . 326
ERC20 požadované funkce a události . . . 326

Volitelné funkce ERC20 . . . 327
Rozhraní ERC20 definované v Solidity . . . 327
Struktura dat ERC20 . . . 328
Pracovní postupy ERC20: "transfer" a "approve & transferFrom" . . . 329
Implementace ERC20 . . . 331
Spuštění vlastního ERC20 tokenu . . . 331
Interakce s METoken pomocí příkazové řádky Truffle . . . 336
Odesílání ERC20 tokenů na adresy kontraktu . . . 340
Demonstrace pracovního postupu „Schválit a převést“ . . . 343
Problémy s ERC20 tokeny . . . 348
ERC223: Navrhovaný standard rozhraní tokenového kontraktu . . . 349
ERC777: Navržený standard rozhraní smluvních tokenů . . . 351
ERC777 háčky . . . 352
ERC721: Standard nezaměnitelného tokenu (vlastnická listina) . . . 354
Používání tokenového standardu . . . 356
Jaké jsou tokenové standardy? Jaký je jejich účel? . . . 356
Měli byste používat tyto standardy? . . . 357
Bezpečnost zralostí . . . 357
Rozšíření rozhraní tokenového standardu . . . 358
Tokeny a ICO . . . 359
Závěry . . . 360
Orákula . . . 361
Proč jsou potřeba orákula . . . 361
Případy užití orákula a příklady . . . 362
Návrhové vzory orákul . . . 363
Ověření dat . . . 366
Výpočetní orákula . . . 367
Decentralizovaná orákula . . . 369
Rozhraní orákulového klienta v Solidity . . . 370
Závěry . . . 376
Decentralizované aplikace (DApps) . . . 377
Co je to DApp? . . . 378
Backend (chytrý kontrakt) . . . 379
Frontend (webové uživatelské rozhraní) . . . 380
Úložiště dat . . . 380

IPFS . . . 380
Swarm . . . 381
Decentralizované komunikační protokoly zpráv . . . 381
Základní příklad DApp: Aukční DApp . . . 381
Aukční DApp: Backend chytrý kontrakt . . . 383
Správa DApp . . . 387
Aukční DApp: Frontend uživatelské rozhraní . . . 388
Další decentralizace aukčního DApp . . . 390
Uložení aukční DApp na Swarm . . . 391
Příprava Swarm . . . 391
Nahrávání souborů do Swarm . . . 393
Ethereum jmenná služba (ENS) . . . 396
Historie Ethereum jmenné služby . . . 396
Specifikace ENS . . . 397
Dolní vrstva: vlastníci jmen a překladače . . . 397
Algoritmus Namehash . . . 397
Jak vybrat platné jméno . . . 399
Vlastnictví kořenového uzlu . . . 399
Překladače . . . 400
Střední vrstva: .eth uzly . . . 400
Vickrey aukce . . . 400
Vrchní vrstva: vlastnické listiny . . . 401
Registrace jména . . . 402
Správa vašeho ENS jména . . . 407
Vytvoření subdomény ENS . . . 408
ENS překladače . . . 410
Překlad jména na Swarm haš (obsah) . . . 411
Od aplikace do DApp . . . 413
Závěry . . . 415
Ethereum virtuální stroj . . . 417
Co je EVM? . . . 417
Srovnání se stávající technologií . . . 419
Instrukční sada EVM (bajtkódové operace) . . . 419
Ethereum stav . . . 424
Kompilace Solidity EVM bajtkódu . . . 425

Kód nasazení kontraktu . . . 430
Rozklad bajtkódu na assembler . . . 431
Turingova úplnost a plyn . . . 437
Plyn . . . 438
Účtování plynu během provádění . . . 439
Úvahy o účtování plynu . . . 439
Náklady na plyn versus cena plynu . . . 440
Záporné náklady na plyn . . . 441
Blokový limit plynu . . . 441
Kdo rozhoduje, jaký je limit plynu v bloku? . . . 441
Závěry . . . 442
Konsensus . . . 443
Konsensus pomocí důkazu prací (PoW) . . . 444
Konsensus pomocí důkazu podílem (PoS) . . . 444
Ethash: Ethereum algoritmus pro důkaz prací . . . 445
Casper: Ethereum algoritmus důkazu podílem . . . 446
Principy konsensu . . . 447
Spor a soutěž . . . 447
Závěry . . . 448
Appendix A: Historie Ethereum rozštěpení . . . 449
Ethereum Classic (ETC) . . . 449
Decentralizovaná autonomní organizace (The DAO) . . . 449
Chyba vícenásobného volání . . . 450
Technické detaily . . . 450
Průběh útoku . . . 450
The DAO tvrdé rozštěpení . . . 451
Časová osa tvrdého rozštěpení DAO . . . 452
Ethereum and Ethereum Classic . . . 454
The EVM . . . 454
Vývoj základní sítě . . . 454
Další významná Ethereum rozštěpení . . . 454
Appendix B: Ethereum Standardy . . . 457
Návrhy na vylepšení Etherea (EIP) . . . 457
Nejvýznamnější EIP a ERC . . . 457
Appendix C: Instrukce Ethereum EVM a spotřeba plynu . . . 479

Appendix D: Vývojové nástroje, rámce a knihovny . . . 493
Rámce . . . 493
Truffle . . . 493
Instalace Truffle rámce . . . 493
Integrace předpřipraveného Truffle projektu (Truffle Box) . . . 494
Vytvoření adresáře projektu truffle . . . 495
Konfigurace truffle . . . 497
Použití truffle k nasazení kontraktu . . . 498
Truffle migrace a porozumění nasazovacím skriptům . . . 499
Používání Truffle příkazové řádky . . . 501
Embark . . . 505
OpenZeppelin . . . 506
ZeppelinOS . . . 510
Nástroje . . . 511
EthereumJS helpeth: nástroj příkazové řádky . . . 511
dapp.tools . . . 512
SputnikVM . . . 513
Knihovny . . . 513
web3.js . . . 513
web3.py . . . 514
EthereumJS . . . 514
web3j . . . 514
EtherJar . . . 514
Nethereum . . . 515
ethers.js . . . 515
Emerald Platform . . . 515
Testování chytrých kontraktů . . . 516
Testování na bločence . . . 517
Ganache: Místní testovací bločenka . . . 518
Appendix E: web3.js tutorial . . . 521
Popis . . . 521
Základní interakce web3.js kontraktu neblokovaným (Async) způsobem . . . 521
Provádění skriptu Node.js . . . 522
Prohlížení ukázkového skriptu . . . 523
Interakce kontraktu . . . 523

Asynchronní provoz s čekáním 526
Index 527

LICENSE

Předmluva

Tato kniha je spoluprací mezi Andreasem M. Antonopoulosem a Dr. Gavinem Woodem. Série šťastných náhod spojila tyto dva autory dohromady v úsilí, které povzbudilo stovky přispěvatelů k vytvoření této knihy, v nejlepším duchu otevřeného zdrojového kódů (open source) a kultury tvůrčího společenství (creative commons).

Gavin chtěl nějakou dobu psát knihu, která by rozšířila Žlutou knihu (jeho technický popis Ethereum protokolu), primárně proto, aby ji otevřel širšímu publiku. Původní dokument byl určen zejména odborné veřejnosti, obsahoval velké množství řeckých písmen a matematických výrazů.

Plány probíhaly - byl nalezen vydavatel - když Gavin mluvil s Andreasem, kterého znal od samého počátku svého působení v Ethereu, jako pozoruhodnou osobnost ve vesmíru.

Andreas nedávno vydal první vydání své knihy *Mastering Bitcoin* (O'Reilly), která se rychle stala autoritativní technickou příručkou pro Bitcoin a kryptoměny. Téměř jakmile byla kniha vydána, jeho čtenáři se ho začali ptát: „Kdy napíšete „Mastering Ethereum"? Andreas již zvažoval svůj další projekt a zjistil, že Ethereum je neodolatelná technická výzva.

Nakonec v květnu 2016 byli Gavin a Andreas shodou okolností ve stejném městě ve stejnou dobu. Setkali se na kávu, aby si spolu povídali o práci na knize. Andreas i Gavin byli oddaní paradigmatu otevřeného zdrojového kódu, a proto se oba zavázali k tomu, že se to bude snažit spolupracovat a bude vydáno na základě licence Creative Commons. Naštěstí vydavatel O'Reilly Media s potěšením souhlasil a oficiálně byl zahájen projekt *Mastering Ethereum*.

Jak používat tuto knihu

Kniha může sloužit jednak jako referenční příručka, ale lze ji číst i od začátku do konce a kompletně tak prozkoumat Ethereum. První dvě kapitoly nabízejí jemný úvod, vhodný pro začínající uživatele a příklady v těchto kapitolách si může vyzkoušet kdokoli s trochou technických dovedností. Tyto dvě kapitoly vám poskytnou dobrý přehled o základech a umožní vám použít základní Ethereum nástroje. < <ethereum_clients_chapter> > a dále jsou určeny hlavně pro programátory a zahrnují mnoho technických témat a programátorských příkladů.

Aby kniha mohla sloužit jednak jako referenční příručka, ale dala se i číst od začátku do konce, tak nevyhnutelně obsahuje určité duplicity. Některá témata, například *gas*, musí být představena

dostatečně brzy, aby ostatní témata dala smysl, ale jsou také podrobně prozkoumána ve svých vlastních oddílech.

Na konci knihy se nachází index, který umožňuje čtenářům snadno najít velmi specifická témata a příslušné oddíly knihy podle klíčových slov.

Zamýšlené publikum

Tato kniha je určena především pro programátory. Pokud umíte používat programovací jazyk, tato kniha vás naučí, jak fungují bločenky chytrých kontraktů, jak je používat a jak vyvíjet chytré kontrakty a decentralizované aplikace. Prvních několik kapitol je také vhodných jako podrobný úvod do Etherea pro neprogramátory.

Konvence použité v této knize

V této knize jsou použity následující typografické konvence:

kurzíva

Označuje nové termíny, URL adresy, e-mailové adresy, názvy souborů a přípony souborů.

+ Pevná šířka +

Používá se pro výpisy programů a také v odstavcích pro odkazování na prvky programu, jako jsou názvy proměnných nebo funkcí, databáze, datové typy, proměnné prostředí, příkazy a klíčová slova.

`Pevná ší¬ka tu¬n¬`

Zobrazuje příkazy nebo jiný text, který by měl uživatel napsat doslova.

_ Kurzíva s pevnou šířkou _

Zobrazuje text, který by měl být nahrazen uživatelsky zadanými hodnotami nebo hodnotami určenými kontextem.

Tato ikona označuje tip nebo návrh.

Tato ikona označuje obecnou poznámku.

Tato ikona označuje varování nebo upozornění.

Ukázky zdrojových kódů

Příklady jsou uvedeny v jazycích Solidity, Vyper a JavaScript a používají příkazové řádky operačního systému Unixového typu. Všechny ukázky kódu jsou k dispozici v úložišti GitHub v podadresáři *code*. Prohlédněte si kód knihy, vyzkoušejte příklady kódu nebo odešlete opravy prostřednictvím GitHub: https://github.com/ethereumbook/ethereumbook.

Všechny ukázky kódu lze replikovat na většině operačních systémů s minimální instalací kompilátorů, interpretrů a knihoven pro odpovídající jazyky. V případě potřeby poskytujeme základní pokyny k instalaci a podrobné příklady výstupu těchto pokynů.

Některé ukázky kódu a výstupy kódu byly přeformátovány pro tisk. Ve všech takových případech byly řádky rozděleny znakem zpětného lomítka (\) následovaným znakem nového řádku. Při přepisování příkladů odstraňte tyto dva znaky a znovu spojte řádky. Měli byste vidět stejné výsledky jako v příkladu.

Všechny ukázky kódu používají skutečné hodnoty a výpočty, pokud je to možné, takže můžete sestavovat příklad od příkladu a vidět stejné výsledky v jakémkoli kódu, který píšete, pro výpočet stejných hodnot. Například soukromé klíče a odpovídající veřejné klíče a adresy jsou všechny skutečné. Ukázkové transakce, kontrakty, bloky a odkazy do bločenky byly zavedeny do skutečné Ethereum bločenkya jsou součástí veřejné knihy, takže si je můžete prohlédnout.

Použití příkladů kódu

Tato kniha je zde, aby vám pomohla dokončit vaší práci. Obecně platí, že pokud je v této knize nabízen příklad kódu, můžete jej použít ve svých programech a dokumentaci. Nemusíte nás kontaktovat kvůli svolení, pokud reprodukujete podstatnou část kódu. Například zápis programu, který používá několik kousků kódu z této knihy, nevyžaduje povolení. Prodej nebo distribuce CD-ROM příkladů z knih O'Reilly vyžaduje svolení. Odpověď na otázku citováním této knihy a citací vzorového kódu nevyžaduje povolení. Začlenění značného množství vzorového kódu z této knihy do dokumentace k vašemu produktu vyžaduje povolení.

Oceňujeme, ale nevyžadujeme, citace. Uvedení zdroje obvykle zahrnuje název, autora, vydavatele, ISBN a autorská práva. Například: „*Mastering Ethereum* by Andreas M. Antonopoulos a Dr. Gavin

Odkazy na společnosti a produkty

Všechny odkazy na společnosti a produkty jsou určeny pro vzdělávací, demonstrační a referenční účely. Autoři neschvalují žádnou z uvedených společností nebo produktů. Netestovali jsme provoz ani zabezpečení žádného z produktů, projektů nebo segmentů kódů uvedených v této knize. Používejte je na vlastní nebezpečí!

Ethereum adresy a transakce v této knize

Adresy, transakce, klíče, QR kódy a data bločenky použité v této knize jsou z větší části skutečné. To znamená, že můžete procházet bločenku, podívat se na transakce nabízené jako příklady, načíst je pomocí vlastních skriptů nebo programů atd.

Nicméně uvádíme, že soukromé klíče použité k vytvoření adres vytištěných v této knize byly „spáleny“. To znamená, že pokud posíláte peníze na některou z těchto adres, budou peníze navždy ztraceny nebo (pravděpodobněji) přiděleny, protože kdokoli, kdo si knihu přečte, může tuto adresu převzít pomocí soukromých klíčů vytištěných v tomto dokumentu.

NEZASÍLEJTE PENÍZE NA ŽÁDNOU ADRESU UVEDENOU V TÉTO KNIHU. Vaše peníze si vezme jiný čtenář, nebo se ztratí navždy.

Jak nás kontaktovat

Informace o *Mastering Ethereum* jsou k dispozici na adrese https://ethereumbook.info/.

Kontaktování Andrease

Andrease M. Antonopoulos můžete kontaktovat na jeho osobním webu: https://antonopoulos.com/

Přihlásit se k odběru Andreasova kanálu na YouTube: https://www.youtube.com/aantonop

Oblíbit si Andreasovovu stránku na Facebooku: https://www.facebook.com/AndreasMAntonopoulos

Sledovat Andrease na Twitteru: https://twitter.com/aantonop

Spojit se s Andreasem na LinkedIn: https://linkedin.com/company/aantonop

Andreas by také rád poděkoval všem patronům, kteří podporují jeho práci měsíčními dary. Můžete podpořit Andrease na Patreonu https://patreon.com/aantonop.

Kontaktování Gavina

Dr. Gavin Wood může být kontaktován na jeho osobním webu: http://gavwood.com/

Sledujte Gavina na Twitteru: https://twitter.com/gavofyork

Gavin je obecně k zastižení na Polkadot Watercooler na Riot.im: http://bit.ly/2xciG68

Poděkování Andrease

Své Matce Tereze vděčím za svojí lásku ke slovům a knihám. Tereza mě vychovala v domě s knihami lemujícími každou zeď. Moje matka mi také koupila můj první počítač v roce 1982, přestože sama o sobě prohlašovala, že se bojí nových technologií. Můj otec, Menelaos, stavební inženýr, který vydal svou první knihu ve věku 80 let, byl ten, kdo mě učil logickému a analytickému myšlení a lásce k vědě a inženýrství.

Děkuji vám všem, že jste mě během této cesty podporoval.

Poděkování Gavina

Moje matka mi zajistila můj první počítač od souseda, když mi bylo 9 let, bez kterého by se můj technický pokrok nepochybně snížil. Dlužím jí také za můj dětský strach z elektřiny a musím přiznat

díky Trevorovi a mým prarodičům, kteří čas od času vykonávali vážnou povinnost „dívat se na mě, jak se připojují“, a bez nichž by uvedený počítač byl k ničemu. Musím také dát uznání různým pedagogům, na které jsem měl během svého života štěstí, od řečeného souseda Seana (který mě učil můj první počítačový program), až po pana Quinna, svého učitele základní školy, který mě vedl, abych více programoval a méně se věnoval historii, až po učitele středních škol, jako je Richard Furlong-Brown, kteří mě vedli, abych se zabýval více programováním a méně ragby.

Musím poděkovat matce mých dětí, Juttě, za její pokračující podporu, a mnoha lidem v mém životě, přátelům novým i starým, které mě udržují, zhruba řečeno, při zdravém rozumu. A konečně, obrovský dík patří Aeronu Buchananovi, bez kterého by se posledních pět let mého života nikdy nemohlo rozvinout tak, jak se stalo, a bez jehož času, podpory a vedení by tato kniha nebyla v tak dobré formě jak je.

Přispěvatelé

Mnoho přispěvatelů nabídlo komentáře, opravy a dodatky k návrhu k pracovním verzím na GitHubu.

Příspěvky na GitHub byly zpracovávány dvěma editory, kteří se dobrovolně rozhodli řídit projekty, kontrolovat, upravovat, slučovat a schvalovat návrhy na úpravy a problémy k řešení:

- Francisco Javier Rojas Garcia (fjrojasgarcia)
- Will Binns (wbnns)

Hlavní příspěvky byly poskytnuty na témata DApps, ENS, EVM, historie rozštěpení, plyn, orákula, bezpečnost chytých kontraktů a Vyper. Další příspěvky, které nebyly zahrnuty v tomto prvním vydání kvůli časovým a prostorovým omezením, lze nalézt ve složce *contrib* na úložišti GitHub. Tisíce menších příspěvků v celé knize zlepšily její kvalitu, čitelnost a přesnost. Upřímné díky všem, kteří přispěli!

Následuje abecedně seřazený seznam všech přispěvatelů GitHub, včetně jejich GitHub ID v závorkách:

- Abhishek Shandilya (abhishandy)
- Adam Zaremba (zaremba)
- Adrian Li (adrianmcli)

- Adrian Manning (agemanning)
- Alejandro Santander (ajsantander)
- Alejo Salles (fiiiu)
- Alessandro Coglio (acoglio)
- Alex Manuskin (amanusk)
- Alex Van de Sande (alexvandesande)
- Anthony Lusardi (pyskell)
- Anup Dhakal (anuphunt)
- Assaf Yossifoff (assafy)
- athio92 (athio92)
- Ben Kaufman (ben-kaufman)
- bgaughran (bgaughran)
- Bok Khoo (bokkypoobah)
- Brandon Arvanaghi (arvanaghi)
- Brian Ethier (dbe)
- Bryant Eisenbach (fubuloubu)
- Carl Park (4000D)
- Carlos Cebrecos (ccebrecos)
- Chanan Sack (chanan-sack)
- Charlie Leathers (cleathers)
- Chris Lin (ChrisLinn)
- Chris Remus (chris-remus)
- Christopher Gondek (christophergondek)
- Cornell Blockchain (CornellBlockchain)
    - Alex Frolov (sashafrolov)
    - Brian Guo (BrianGuo)

    - Brian Leffew (bleffew99)
    - Giancarlo Pacenza (GPacenza)
    - Lucas Switzer (LucasSwitz)
    - Ohad Koronyo (ohadh123)
    - Richard Sun (richardsfc)
- Cory Solovewicz (CorySolovewicz)
- crypto501 (crypto501)
- Dan Shields (NukeManDan)
- Danger Zhang (safetyth1rd)
- Daniel Jiang (WizardOfAus)
- Daniel McClure (danielmcclure)
- Daniel Peterson (danrpts)
- Dave Potter (dnpotter)
- David Lozano Jarque (davidlj95)
- David McFadzean (macterra)
- Denis Milicevic (D-Nice)
- Dennis Zasnicoff (zasnicoff)
- Diego H. Gurpegui (diegogurpegui)
- Dimitris Tsapakidis (dimitris-t)
- Dzmitry Bachko (dbachko)
- Edward Posnak (edposnak)
- Enrico Cambiaso (auino)
- Ersin Bayraktar (ersinbyrktr)
- Exhausted Mind (exhaustedmind)
- Flash Sheridan (FlashSheridan)
- Franco Abaroa (francoabaroa)

- Franco Daniel Berdun (fMercury)
- Griff Green (GriffGreen)
- Harry Moreno (morenoh149)
- Harshal Patil (ErHarshal)
- Håvard Anda Estensen (estensen)
- Hon Lau (masterlook)
- Hudson Jameson (Souptacular)
- Hyunbin Jeong (gusqls1603)
- Iuri Matias (iurimatias)
- Ivan Molto (ivanmolto)
- Jacques Dafflon (jacquesd)
- Jason Hill (denifednu)
- Javier Rojas (fjrojasgarcia)
- Jaycen Horton (jaycenhorton)
- JB Paul (yjb94)
- jeremyfny (jeremyfny)
- Joel Gugger (guggerjoel)
- John Woods (johnalanwoods)
- Jon Ramvi (ramvi)
- Jonathan Velando (rigzba21)
- jpopxfile (jpopxfile)
- Jules Lainé (fakje)
- Karolin Siebert (karolinkas)
- Kevin Carter (kcar1)
- Kevin Weaver (kevinweaver)
- Krzysztof Nowak (krzysztof)

- Lane Rettig (lrettig)
- Leo Arias (elopio)
- Liang Ma (liangma)
- Łukasz Gołębiewski (lukasz-golebiewski)
- Luke Schoen (ltfschoen)
- Marcelo Creimer (mcreimer)
- Martin Berger (drmartinberger)
- Masi Dawoud (mazewoods)
- Massimiliano Terzi (terzim)
- Matt Peskett (mattpeskett)
- Matthew Sedaghatfar (sedaghatfar)
- mehlawat (mehlawat)
- Michael Freeman (stefek99)
- Miguel Baizan (mbaiigl)
- Mike Pumphrey (bmmpxf)
- Milo Chen (milochen0418)
- Mobin Hosseini (iNDicat0r)
- Nagesh Subrahmanyam (chainhead)
- Nichanan Kesonpat (nichanank)
- Nicholas Maccharoli (Nirma)
- Nick Johnson (arachnid)
- Omar Boukli-Hacene (oboukli)
- Paulo Trezentos (paulotrezentos)
- Pet3rpan (pet3r-pan)
- Peter Kacherginsky (iphelix)
- Pierre-Jean Subervie (pjsub)

- Pong Cheecharern (Pongch)
- Qiao Wang (qiaowang26)
- Raul Andres Garcia (manilabay)
- rattle99 (rattle99)
- robFifth (robFifth)
- Robin Pan (robinpan1)
- Roger Häusermann (haurog)
- Robert Miller (bertmiller)
- Saxon Knight (knight7)
- Sebastian Falbesoner (theStack)
- Sejin Kim (sejjj120)
- Seong-il Lee (modolee)
- sgtn (shogochiai)
- Solomon Victorino (bitsol)
- stefdelec (stefdelec)
- Steve Klise (sklise)
- Sylvain Tissier (SylTi)
- Taylor Masterson (tjmasterson)
- Tim Nugent (timnugent)
- Timothy McCallum (tpmccallum)
- Tommy Cooksey (tcooksey1972)
- Tomoya Ishizaki (zaq1tomo)
- Ulrich Stark (ulrichstark)
- Vignesh Karthikeyan (meshugah)
- westerpants (westerpants)
- Will Binns (wbnns)

- Xavier Lavayssière (xalava)
- Yash Bhutwala (yashbhutwala)
- Yeramin Santana (ysfdev)
- Yukishige Nakajo (nakajo2011)
- Zhen Wang (zmxv)
- ztz (zt2)

Bez pomoci poskytnuté výše uvedenými by nebylo možné tuto knihu vytvořit. Vaše příspěvky prokazují sílu otevřených zdrojových kódů a otevřené kultury a my jsme věčně vděční za vaši pomoc. Děkujeme.

Edice otevřeného překladu

Otevřená edice *Mastering Ethereum* byla přeložena do jazyka ČEŠTINA těmito dobrovolníky:

RNDr. Jan Lánský, Ph.D. (zizelevak@gmail.com [mailto:zizelevak@gmail.com]), Vysoká škola finanční a správní, 2020; Nepřekládejte tento komentář Comment: Translate the word LANGUAGE above to the language you are translating. Then translate one of the names below replacing it with your name This will include your name in the acknowledgments. RNDr. Jan Lánský, Ph.D. (zizelevak@gmail.com [mailto:zizelevak@gmail.com]), Vysoká škola finanční a správní, 2020;

- Jan Lansky (zizelevak@gmail.com [mailto:zizelevak@gmail.com])
- Name2
- Name3
- Name4
- Name5
- Name6
- Name7
- Name8
- Name9
- Name10

- Name11
- Name12
- Name13
- Name14
- Name15
- Name16
- Name17
- Name18
- Name19
- Name20
- Name21
- Name22
- Name23
- Name24
- Name25
- Name26
- Name27
- Name28
- Name29
- Name30
- Name31
- Name32
- Name33
- Name34
- Name35
- Name36

- Name37
- Name38
- Name39
- Name40
- Name41
- Name42
- Name43
- Name44
- Name45
- Name46
- Name47
- Name48
- Name49
- Name50

Zdroje

Tato kniha uvádí různé veřejné a otevřené zdroje:

https://github.com/ethereum/vyper/blob/master/README.md

Licence MIT (MIT)

https://vyper.readthedocs.io/en/latest/

Licence MIT (MIT)

https://solidity.readthedocs.io/en/v0.4.21/common-patterns.html

Licence MIT (MIT)

https://arxiv.org/pdf/1802.06038.pdf

Arxiv Non-Exclusive-Distribution

https://github.com/ethereum/solidity/blob/release/docs/contracts.rst#inheritance

Licence MIT (MIT)

https://github.com/trailofbits/evm-opcodes

Apache 2.0

https://github.com/ethereum/EIPs/

Creative Commons CC0

https://blog.sigmaprime.io/solidity-security.html

Creative Commons CC BY 4.0

Slovníček pojmů

Slovníček pojmů obsahuje mnoho pojmů používaných ve spojitosti s Ethereem. Tyto pojmy se používají v celé knize, proto si tento slovníček uložte pro rychlý přehled.

Účet

Objekt obsahující adresu, zůstatek, nonci a nepovinné datové úložiště a bajtkód. Účet může být účet kontraktu nebo externě vlastněný účet (EOA).

Adresa

Obecně reprezentuje EOA nebo kontrakt. Adresa může přijímat (cílová adresa) nebo odesílat (zdrojová adresa) transakce do bločenky. Konkrétně jde o 160 bitů z výstupu hašovací funkce Keccak aplikované na vyřejný klíč vytvořený kryptografií eliptických křivek ECDSA.

Assert

V programovacím jazyku Solidity se assert (false) zkompiluje na 0xfe, neplatný operační kód, který spotřebuje veškerý zbývající plyn a vrátí všechny změny. Pokud je splněna podmínka v závorce příkazu assert (), znamená to, že došlo k něčemu velmi špatnému a neočekávanému a budete muset opravit kód. Měli byste použít assert () pro otestování podmínek, které by nikdy neměly nabýt pravdivé hodnoty.

Big-endian

Poziční reprezentace čísla, při které je nejvýznamnější číslice uvedena jako první. Odpovídá běžné používanému zápisu čísel lidmi. Opakem je little-endian, u kterého jsou pozice číslic obráceny, jako první je uvedena nejméně významná číslice.

BIPy

Návrhy na vylepšení Bitcoinu (Bitcoin Improvement Proposals). Soubor návrhů, které členové Bitcoinové komunity předložili ve snaze vylepšit Bitcoin. Například BIP-21 je návrh na zlepšení Bitcoinového jednotného označování zdrojů (uniform resource identifier; URI).

Blok

("block, defined"Uspořádaná posloupnost požadovaných informací (hlavička bloku) o transakcích, o které se rozšiřuje konsensus o stavu systému, a množina dalších hlaviček bloků známých jako potomci předka (ommers). Bloky jsou do Ethereum sítě přidávány těžaři.

Bločenka (blockchain)

V Ethereu, posloupnost bloků ověřená důkazem prací. Každý blok je spojen se svým předchůcem, postupně až k základnímu (genesis) bloku. Narozdíl od Bitcoinového protokolu zde není limit na velikost bloku, ale místo toho se používá limit na množství spotřebovaného plynu.

Bajtkód

Abstraktní instrukční sada určená pro efektivní provádění softwarovým interpretem nebo virtuálním strojem. Na rozdíl od lidsky čitelného zdrojového kódu je bajtkód vyjádřen v číselném formátu.

Byzantium rozštěpení

První ze dvou tvrdých rozštěpení (hard fork) pro vývojovou fázi Metropolis. Zahrnovalo EIP-649: odložení obtížnostní bomby a snížení odměny za vytěžení bloku. Ace Age (viz níže) byla odložena o 1 rok a odměna za vytěžení bloku byla snížena z 5 na 3 ethery.

Kompilace

Převod kódu zapsaného v programovacím jazyce vyšší úrovně (např. Solidity) do jazyka nižší úrovně (např. bajtkód EVM).

Konsenzus

Shoda na tom, které bloky jsou součástí bločenky. Pokud má mnoho uzlů (obvykle většina uzlů v síti) vzájemně stejné bloky ve svých lokálních bločenkách. Nesmí se zaměňovat s pravidly konsensu.

Pravidla konsensu

Pravidla ověřování správnosti bloků, která dodržují úplné uzly, aby zůstaly ve shodě s ostatními uzly. Nesmí se zaměňovat s konsensem.

Constantinople rozštěpení

Druhá část fáze Metropolis proběhla v únoru 2019. Očekávalo se, že mezi jinými změnami bude zahrnut přechod na hybridní algoritmus dosahování konsensu využívající důkaz práci a důkaz podílem, nicméně toto očekávání nebylo naplněno.

Účet kontraktu

Účet obsahující bajtkód, který se provede vždy, když obdrží transakci z jiného účtu (EOA nebo kontraktu).

Transakce vytvářející kontrakt

Speciální transakce, jejímž příjemce je „nulová adresa“, která se používá k vytvoření kontraktu a jeho zaznamenání do Ethereum bločenky (viz „nulová adresa“).

DAO

Decentralizovaná autonomní organizace. Společnost nebo jiná organizace, která funguje bez hierarchického řízení. Také může znamenat kontrakt s názvem „The DAO“, který byla spuštěn 30. dubna 2016, a který byl poté napaden v červnu 2016. Tato situace nakonec vedla k tvrdému rozštěpení (označovanému jako DAO) v bloku č. 1 192 000, které zneplatnilo transakce provedené v rámci kontraktu The DAO a způsobila rozdělení Etherea na dva konkurenční systémy: Ethereum Classic a Ethereum.

DApp

Decentralizovaná aplikace. Je to minimálně chytrý kontrakt a webové uživatelské rozhraní. Obecněji řečeno, DApp je webová aplikace, která je vytvořena na základě otevřených, decentralizovaných infrastrukturních služeb typu peer-to-peer. Mnoho DApps navíc zahrnuje decentralizované úložiště, komunikační protokol a platformu.

Vlastnická listina (deed)

Norma pro nezaměnitelné tokeny (Non-fungible token; NFT) zavedená návrhem ERC721. Na rozdíl od tokenů ERC20, vlastnické listiny prokazují vlastnictví jednoho konkrétního objektu a nejsou vzájemně zaměnitelné, ačkoli nejsou uznány jako právní dokumenty v žádné jurisdikci - alespoň ne v současnosti (viz také „NFT“).

Obtížnost těžby

Síťové nastavení, které udává, jaké množství výpočtů je třeba k vykonání důkazu prací.

Digitální podpis

Krátký řetězec dat, který uživatel vytváří pro daný dokument pomocí soukromého klíče, takže kdokoli s odpovídajícím veřejným klíčem, podpisem dokumentu a dokumentem může ověřit, že (1) dokument byl „podepsán“ vlastníkem konkrétního soukromého klíče, a že (2) dokument nebyl po podpisu změněn.

ECDSA

Algoritmus digitálního podpisu pomocí kryptografie eliptických křivek (Elliptic Curve Digital Signature Algorithm). Kryptografický algoritmus, který v Ethereu zajišťuje, že pokyn k provedení transakce smí vydat pouze vlastník účtu, ze kterého má být transakce odeslána.

EIP

Návrh na vylepšení Etherea. Návrhový dokument poskytující informace komunitě Etherea, popisující navrhovanou novou vlastnost, proces nebo prostředí. Další informace naleznete na adrese https://github.com/ethereum/EIPs (viz také „ERC“).

ENS

Ethereum jmenný systém (Ethereum Name Service). Další informace naleznete na adrese https://github.com/ethereum/ens/.

Entropie

V kryptografii znamená nedostatek předvídatelnosti nebo úroveň náhodnosti. Při generování tajných informací, jako jsou soukromé klíče, se algoritmy obvykle spoléhají na zdroj vysoké entropie, aby zajistily, že jejich výstup je nepředvídatelný.

EOA

externě vlastněný účet (Externally Owned Account). Účet vytvořený lidskými uživateli sítě Ethereum nebo pro ně.

ERC

Ethereum žádost o připomínky (Ethereum Request for Comments). Štítek přidělený některým EIP, které se pokoušejí definovat specifický standard použití Ethereem.

Ethash

Algoritmus důkazu prací pro Ethereum 1.0. Další informace naleznete na adrese https://github.com/ethereum/wiki/wiki/Ethash.

Ether

Nativní kryptoměna používaná v ekosystému Ethereum, pomocí které se platí náklady na plyn při provádění chytrých kontraktů. Jeho symbolem je Ξ, velké řecké písmeno Ksí.

Událost

Umožňuje použití logovacích zařízení EVM. DApps může poslouchat události a používat je ke spouštění zpětných volání JavaScriptu v uživatelském rozhraní. Další informace naleznete na adrese http://solidity.readthedocs.io/en/develop/contracts.html#events.

EVM

Virtuální stroj Ethereum (Ethereum Virtual Machine). Virtuální stroj založený na zásobníkové architektuře, který vykonává bytecode. V Ethereu prováděcí model specifikuje, jak se mění stav systému, pokud je mu zadána posloupnost instrukcí bajtkódu a uspořádaná n-tice dat prostředí. Toto je specifikováno formálním modelem virtuálního stavového stroje.

EVM jazyk symbolických instrukcí

Lidsky čitelná forma EVM bajtkódu.

Nouzová funkce

Výchozí (fallback) funkce, která je zavolána, pokud nejsou zadána data nebo není uvedeno jméno funkce.

Kohoutek (faucet)

Služba, která zdarma vydává finanční prostředky pro testovací účely. Takto získané ethery lze použít pouze na testovací síti.

Finney

Název jedné z dílčích jednotek etheru. 1 finney = 10^{15} wei, 10^{3} finney = 1 ether.

Rozštěpení (fork)

Změna protokolu způsobující vytvoření alternativního řetězce nebo dočasný nesoulad ve dvou potenciálních blokových cestách během těžby.

Frontier

Počáteční testovací fáze vývoje Etherea, která trvala od července 2015 do března 2016.

Ganache

Osobní Ethereum bločenka, kterou můžete použít ke spouštění testů, provádění příkazů a prohlížení stavu systému, přičemž můžete sledovat probíhající operace.

Plyn (Gas)

Virtuální palivo používané v Ethereu k provádění chytrých kontraktů. EVM používá účetní mechanismus k měření spotřeby plynu a omezení spotřeby výpočetních zdrojů (viz „Turingovská úplnost“).

Limit plynu

Maximální množství plynu, které transakce nebo blok může spotřebovat.

Gavin Wood

Britský programátor, který je spoluzakladatelem a bývalým technickým ředitelem Etherea. V srpnu 2014 navrhl programovací jazyk Solidity, který slouží pro psaní chytrých kontraktů.

Základní (genesis) blok

První blok v bločence, který se používá k inicializaci konkrétní sítě a její kryptoměny.

Geth

Go Ethereum. Jedna z nejvýznamnějších implementací protokolu Ethereum napsaná v programovacím jazyce Go.

Tvrdé rozštěpení (hard fork)

Trvalá neshoda v bločence; také známá jako tvrdě rozštěpující změna. Běžně se vyskytuje, když nedupgradované uzly nemohou ověřit bloky vytvořené upgradovanými uzly, které se řídí novějšími pravidly konsensu. Nesmí se zaměňovat s rozštěpením, měkkým rozštěpením, softwarovým rozštěpením nebo Git rozštěpením.

Haš

Otisk pevné délky bez ohledu na velikost vstupu, který je vytvořený hašovací funkcí.

HD peněženka

Peněženka vytvářející a uchovávající klíče hierarchicky deterministickým (HD) způsobem dle standardu BIP-32.

Semínko HD peněženky

Hodnota použitá k vygenerování hlavního soukromého klíče a kódu hlavního řetězce pro HD peněženku. Semínko peněženky může být reprezentováno mnemotechnickými slovy, což lidem usnadňuje kopírování, zálohování a obnovu soukromých klíčů.

Homestead

Druhá vývojová fáze Etherea, zahájená v březnu 2016 v bloku # 1 150 000.

ICAP

Výměnný protokol klientské adresy (Inter-exchange Client Address Protocol). Kódování Ethereum adresy , které je částečně kompatibilní s kódováním IBAN (International Bank Account Number) a nabízí univerzální, kontrolní součet a interoperabilní kódování pro adresy Ethereum. Adresy ICAP používají nový pseudo kód země IBAN: XE, zkratka pro „eXtended Ethereum“, jak se používá v nestátních měnách (např. XBT, XRP, XCP).

Ice Age

Tvrdé rozštěpení Etherea v bloku č. 200 000, ve kterém byl představen exponenciální nárůst obtížnosti (také známý jako obtížná bomba), motivující k přechodu k důkazu podílem.

IDE

Integrované vývojové prostředí. Uživatelské rozhraní, které obvykle kombinuje editor kódu, kompilátor, běhové prostředí a nástroje pro ladění.

Problém nezměnitelnosti nasazeného kódu

Once a contract's (or library's)Jakmile je kód kontraktu (nebo knihovny) nasazen, stane se nezměnitelným. Standardní postupy vývoje softwaru se spoléhají na to, že budou moci opravit možné chyby a přidat nové funkce, což představuje výzvu pro vývoj chytrých kontraktů.

Vnitřní transakce (také „zpráva“)

Transakce zaslaná chytrým kontraktem ve prospěch jiného chytrého kontrakt nebo EOA.

IPFS

Meziplanetární souborový systém (InterPlanetary File System). Protokol, síť a projekt s otevřeným zdrojovým kódem navržený k vytvoření obsahově adresovatelných, peer-to-peer metodou uložených a sdílených hypermedií v distribuovaném souborovém systému.

KDF

Funkce odvození klíče. Také se nazývá „algoritmus protahování hesla“. Používá se ve formátech pro ukládání klíčů k ochraně před útoky brutální sílou, slovníkem a duhovou tabulkou. Přístupové heslo je chráněno opakovanou aplikací hašovací funkce.

Keccak-256

Kryptografická hašovací funkce používaná v Ethereu. Keccak-256 byl standardizován jako SHA-3.

Soubor s uloženým klíčem

Soubor kódovaný JSON, který obsahuje jeden (náhodně vygenerovaný) soukromý klíč, šifrovaný heslem pro zvýšení bezpečnosti.

LevelDB

Otevřený software úložiště dvojic klíč - hodnota implementovaný jako odlehčená, jednoúčelová knihovna s vazbami na mnoho platforem. Má otevřený zdrojový kód.

Knihovna

Zvláštní typ kontraktu, který nemá žádné platby přijímající funkce, nouzovou funkci ani datové úložiště. Proto nemůže přijímat ani uchovávat ether ani ukládat data. Knihovna slouží jako dříve nasazený kód, který mohou ostatní kontrakty volat. Výpočet probíhá v režimu pouze pro čtení.

Odlehčený klient

Ethereum klient, který neukládá lokální kopii bločenky ani neověřuje bloky a transakce. Nabízí funkce peněženky a umí vytvářet a odesílat transakce.

Merkle Patricia strom

Datová struktura používaná v Ethereu k efektivnímu ukládání dvojic klíč - hodnota.

Zpráva

Interní transakce, která není nikdy ukládána do bločenky a je odesílána pouze v rámci EVM.

Volání zprávy

"message call") Akt předávání zprávy z jednoho účtu na druhý. Pokud je cílový účet spojen s kódem EVM, bude EVM spuštěn se stavem tohoto objektu a zpráva byla zpracována.

METoken

Mastering Ethereum Token. ERC20 token použitý v této knize pro demonstrační účely.

Metropolis

Třetí vývojová fáze Etherea, zahájená v říjnu 2017.

Těžař

"miners")Síťový uzel, který opakovaným výpočtem haše bloku se změněnou noncí najde platný důkaz prací pro nový blok: [hašování].

Mist

První prohlížeč s podporou Etherea, vytvořený Nadací Ethereum. Obsahuje prohlížečovou peněženku, která byla první implementací standardu tokenů ERC20 (Fabian Vogelsteller, autor ERC20, byl také hlavním vývojářem Mistu). Mist byla také první peněženka, která zavedla kontrolní součet camelCase (EIP-55; viz < <EIP55> >). Mist provozuje úplný uzel a nabízí úplný prohlížeč DApp s podporou úložiště Swarm a ENS adres.

Síť

Pokud hovoříme o síti Ethereum, síť typu peer-to-peer, která propaguje transakce a bloky každému Ethereum uzlu (účastník sítě).

NFT

Nezaměnitelný token (také známý jako „vlastnická listina“). Toto je standard tokenů zavedený návrhem ERC721. NFT lze sledovat a obchodovat, ale každý token je jedinečný a odlišný; nejsou zaměnitelné jako tokeny ERC20. NFT mohou představovat vlastnictví digitálních nebo fyzických aktiv.

Uzel

Softwarový klient, který je zapojen do sítě.

Nonce

V kryptografii je to hodnota, kterou lze použít pouze jednou. V Ethereum se používají dva typy noncí: nonce účtu je čítač transakcí na každém účtu, který se používá k zabránění útoku opakováním transakce; nonce důkazu prací je náhodná hodnota v bloku, která byly použita k dosažení důkazu prací.

Potomek předka (Ommer)

Blok, který je potomkem předka, ale sám není předkem. Když těžař najde platný blok, jiný těžař možná zveřejnil konkurenční blok, který se přidá na vrchol bločenky. Na rozdíl od

Bitcoinu mohou být osiřelé bloky v Ethereu zahrnuty novějšími bloky jako potomci předka a obdrží částečnou odměnu za vytvoření bloku. V anglickém originálu se používá termín „ommer“, který je preferovaným genderově neutrálním termínem pro sourozence rodičovského bloku, ale někdy je užíván výraz „strýc“.

Parity

Jedna z nejvýznamnějších interoperabilních implementací Ethereum klienta.

Tajný klíč (secret key)

Viz „soukromý klíč“.

Důkaz podílem (PoS)

Metoda, pomocí které protokol kryptoměnové bločenky distribuovaně dosahuje konsensu o svém stavu. PoS žádá uživatele, aby prokázali vlastnictví určitého množství kryptoměny (jejich „podíl“ v síti), aby se mohli podílet na ověřování transakcí.

Důkaz prací (PoW)

Poslounost dat (důkaz), k jejíž nalezení je potřeba provést významný výpočet. V Ethereu musí těžaři najít numerické řešení algoritmu Ethash, které splňuje cíl obtížnosti v celé síti.

Veřejný klíč

Číslo odvozené jednosměrnou funkcí ze soukromého klíče, které může být veřejně sdíleno a kdokoli ho smí použít k ověření digitálního podpisu vytvořeného odpovídajícím soukromým klíčem.

Účtenka

Data vrácená Ethereum klientem, která představují výsledek konkrétní transakce, včetně haše transakce, čísla jejího bloku, množství použitého plynu, a v případě nasazení chytrého kontraktu, adresa kontraktu.

Útok opětovného volání (Re-entrancy)

Útok, při kterém kontrakt útočníka volá kontrakt oběti takovým způsobem, že během vykonávání kontraktu oběti je opakovaně rekurzivně volán kontrakt útočníka. To může například vést k odcizení finančních prostředků kvůli přeskočení částí kontraktu oběti, které aktualizují zůstatky nebo počítají částky výběru.

Odměna

Množství etheru obsažené v každém novém bloku jako odměna sítě pro těžaře, který našel řešení důkazu prací.

RLP

Rekurzivní prefix délky (Recursive Length Prefix). Kódovací standard navržený vývojáři Ethereum pro kódování a serializaci objektů (datových struktur) libovolné složitosti a délky.

Satoshi Nakamoto

Jméno, které použila osoba nebo skupina osob, kteří navrhli Bitcoin, vytvořili jeho původní referenční implementaci a byli prvními, kteří vyřešili problém dvojitého utrácení digitální měny. Jejich skutečná identita zůstává neznámá.

Soukromý klíč (private key)

Tajné číslo, které umožňuje uživatelům Etherea prokázat vlastnictví účtu nebo kontraktu tím, že vytvoří digitální podpis (viz „veřejný klíč“, „adresa“, „ECDSA“).

Serenity

Čtvrtá a poslední fáze vývoje Etherea. Serenity dosud nemá naplánované datum nasazení.

Serpent

Procedurální (imperativní) programovací jazyk určený k tvorbě chytrých kontraktů. Jeho syntax je podobná Pythonu.

SHA

Bezpečný hašovací algoritmus (Secure Hash Algorithm). Rodina kryptografických hašovacích funkcí publikovaných Národním institutem pro standardy a technologie (NIST).

Jednoinstanční (singleton)

Termín z oblasti programování, který popisuje objekt, který může existovat pouze v jedné instanci.

Chytrý kontrakt

Program, který se provádí na výpočetní infrastruktuře Etherea.

Solidity

Procedurální (imperativní) programovací jazyk se syntaxí, která je podobná jazykům JavaScript, C ++ nebo Java. Nejoblíbenější a nejčastěji používaný jazyk pro Ethereum chytré kontrakty. Vytvořil ho Dr. Gavin Wood (spoluautor této knihy).

Solidity vkládaný assembler

jazyk symbolických instrukcí EVM v Solidity programu. Solidity podporuje vkládaný assembler, což usnadňuje psaní určitých operací.

Spurious Dragon

Tvrdé rozštěpení bločenky Etherea, ke kterému došlo v bloku č. 2 675 000, aby bylo možné omezit způsoby útoku odepřením služby a vyčistit stav (viz také „Tangerine Whistle“). Mechanismus ochrany proti opětovnému útoku.

Swarm

Decentralizovaná úložná síť (P2P), která se používá společně s Web3 a Whisper k vytváření DApps.

Szabo

Název jedné z dílčích jednotek etheru. 1 szabo = 10^{12} wei, 10^{6} szabo = 1 ether.

Tangerine Whistle

Tvrdé rozštěpení Ethereum bločenky, ke kterému došlo v bloku č. 2 463 000, byl změněn výpočet plynu pro určité operace náročné na práci s datovým úložištěm a byla vyčištěna data z datového úložiště stavu, která byla vytvořena při útoku odepření služby, který využil nízké náklady na plyn při těchto operacích.

Testnet

Zkratka pro "testovací síť." Síť používaná k simulaci chování hlavní Ethereum sítě.

Transakce

Data odevzdaná do Ethereum bločenky, podepsaná účtem odesilatele, směřující na určenou adresu. Transakce obsahuje metadata, například limit plynu pro tuto transakci.

Truffle

Jeden z nejčastěji používaných vývojových prostředí pro Ethereum.

Turingovská úplnost

Koncept pojmenovaný po anglickém matematiku a počítačovém vědci Alanovi Turingovi: systém pravidel pro manipulaci s daty (jako je počítačová instrukční sada, programovací jazyk nebo celulární automat) se označuje, že je „Turingovsky úplný“ nebo „výpočetně úplný“, pokud ho lze použít k simulaci jakéhokoli Turingova stroje.

Vitalik Buterin

Rusko-kanadský programátor a spisovatel primárně známý jako spoluzakladatel Etherea a časopisu Bitcoin Magazine.

Vyper

Vysokoúrovňový programovací jazyk podobný jazyku Serpent, se syntaxí podobnou jazyku Pythonu. Jeho cílem je přiblížit se čistě funkcionálnímu jazyku. Vytvořil ho Vitalik Buterin.

Peněženka

Software, který drží soukromé klíče. Slouží k přístupu a kontrole Ethereum účtů a interakci s chytrými kontrakty. Klíče pro zvýšení bezpečnosti nemusí být uloženy v peněžence a místo toho je možné je získat z offline úložiště (např. z paměťové karty nebo papíru). Navzdory jejich jménu peněženky nikdy neukládají skutečné mince nebo tokeny.

Web3

Třetí verze webu. Web3, který poprvé navrhl Dr. Gavin Wood, představuje novou vizi a zaměření na webové aplikace: od centrálně vlastněných a spravovaných aplikací po aplikace postavené na decentralizovaných protokolech.

Wei

Nejmenší z dílčích jednotek etheru. 10^{18} wei = 1 ether.

Whisper

Decentralizovaná (P2P) služba zasílání zpráv. Používá se spolu s Web3 a Swarm k vytváření DApps.

Nulová adresa

Speciální Ethereum adresa složená výhradně z nul, která je určena jako cílová adresa transakce vytvářející kontrakt.

Co je Ethereum?

Ethereum je často popisováno jako „světový počítač.“ Co to však znamená? Začněme popisem zaměřeným na informatiku a pak zkusíme uvést praktičtější analýzou schopností a charakteristik Etherea a porovnáme ho s Bitcoinem a dalšími decentralizovanými platformami pro výměnu informací (nebo zkráceně „bločenkami“).

Z pohledu informatiky je Ethereum deterministický, ale prakticky neomezený stavový stroj, skládající se z globálně přístupného a jednoinstančního (singleton) stavu a virtuálního stroje, který aplikuje změny na tento stav.

Z praktičtějšího hlediska je Ethereum software s otevřeným zdrojovým kódem, globálně decentralizovaná výpočetní infrastruktura, která provádí programy zvané *chytré kontrakty*. Používá bločenku k synchronizaci a ukládání změn stavu systému. Používá kryptoměnu zvanou *ether* k určování ceny transakcí a využívání systémových zdrojů.

Platforma Ethereum umožňuje vývojářům vytvářet výkonné decentralizované aplikace s vestavěnými ekonomickými funkcemi. Poskytuje vysokou dostupnost, kontrolovatelnost, transparentnost a neutralitu, ale také snižuje nebo vylučuje cenzuru a snižuje určitá rizika plynoucí z chování protistrany.

Srovnání s Bitcoinem

Mnoho se seznámí s Ethereem po nějaké předchozí zkušeností s kryptoměnami, konkrétně s Bitcoinem. Ethereum sdílí mnoho společných prvků s jinými otevřenými bločenkami: síť typu peer-to-peer spojující účastníky, byzantský konsenzuální algoritmus tolerantní vůči chybám pro synchronizaci aktualizací stavu (bločenka využívající důkaz prací), použití kryptografických primitiv, jako jsou digitální podpisy a haše a digitální měna (ether).

V mnoha ohledech se však účel i konstrukce Etherea nápadně liší od účelů a otevřených bločenek, které mu předcházely, včetně Bitcoinu.

Účelem Etherea není primárně být platební síť využívající digitální měnu. Zatímco digitální měna ether je nedílnou součástí a je nezbytná pro provozování Etherea, ether je zamýšlen jako *spotřební měna*, kterou se platí za použití platformy Ethereum jako světového počítače.

Na rozdíl od Bitcoinu, který má velmi omezený skriptovací jazyk, je Ethereum navrženo jako

univerzální programovatelná bločenka, který provozuje *virtuální stroj* schopný provádět kód libovolné a neomezené složitosti. Bitcoinový skriptovací jazyk úmyslně omezen na jednoduché vyhodnocení pravdivosti podmínky umožňující utrácení, zatímco Ethereum jazyk je *Turingovsky úplný*, což znamená, že Ethereum může přímo fungovat jako univerzální počítač.

Součásti bločenky

Součásti otevřené veřejné bločenky jsou (obvykle):

Síť typu peer-to-peer (P2P) spojující účastníky a propagující transakce a bloky ověřených transakcí, založená na standardizovaném algoritmu vlny: protokol * Zprávy, ve formě transakcí, využívané ke změně stavu * Sada pravidel konsensu, která určují, co představuje transakce a čím lze dosáhnout platné změny stavu * Stavový stroj, který zpracovává transakce podle pravidel konsensu Řetězec kryptograficky zabezpečených bloků, který funguje jako nezměnitelný záznam všech ověřených a přijatých stavových přechodů Algoritmus konsensu, který decentralizuje kontrolu nad bločenkou tím, že nutí účastníky ke spolupráci při prosazování pravidel konsensu * Z hlediska teorie her spolehlivá motivační schémata (např. náklady na důkaz prací versus odměna za vytvoření bloku), která ekonomicky zajistí stavový stroj otevřený prostředí * Jedna nebo více implementací softwaru s otevřeným zdrojovým kódem s funkčností uvedenou výše („klienti“)

Všechny nebo většina těchto komponent je obvykle včleněna do jediného softwarového klienta. Například Bitcoin má referenční implementaci s otevřenými zdrojovými kódy zvanou *Bitcoin Core*, která obsahuje i klienta zvaného *bitcoind*. Ethereum nemá referenční implementaci, ale místo má *referenční specifikaci*, matematický popis systému v Žluté knize (viz <<references> >). Existuje řada klientů, kteří jsou naprogramovány podle referenční specifikace.

V minulosti jsme termín „bločenka“ používali k reprezentaci všech právě vyjmenovaných složek, jako zkratka pro kombinaci technologií, které zahrnují všechny popsané vlastnosti. Dnes však existuje obrovské množství bločenek s různými vlastnostmi. Potřebujeme kvalifikátory, aby nám pomohli porozumět charakteristikám dotčené bločenky, jako je *otevřená, veřejná, globální, decentralizovaná, neutrální, _ a _odolná proti cenzuře*, abychom identifikovali důležité charakteristiky „bločenkového“ systému, který tyto komponenty obsahuje.

Ne všechny bločenky jsou vytvořeny stejným způsobem. Když vám někdo řekne, že je něco jako bločenka, neobdrželi jste odpověď; raději byste měli začít klást spoustu otázek, abyste si objasnili, co to znamená, když někdo používá slovo „bločenka“. Začněte tím, že požádáte o popis součástí v předchozím seznamu, a pak se zeptejte, zda tato „bločenka“ vykazuje vlastnosti být *otevřená, veřejná*

atd.

Zrození Etherea

Všechny velké inovace řeší skutečné problémy a Ethereum není výjimkou. Ethereum bylo vytvořeno v době, kdy lidé poznali sílu Bitcoinového modelu a snažili se nalézt další jeho využití než pouze jako kryptoměny. Vývojáři však čelili hlavolamu: buď mohli stavět na Bitcoinu, nebo mohli vytvořit novou bločenku. Stavět na Bitcoinu znamená žít v úmyslných omezeních sítě a snažit se najít řešení. Zdá se, že omezená sada typů transakcí, datových typů a velikostí ukládání dat omezuje druhy aplikací, které by mohly běžet přímo na Bitcoinu; všechno ostatní potřebovalo další vrstvy mimo bločenku, a to okamžitě popřelo mnoho výhod používání veřejné bločenky. Pro projekty, které vyžadovaly více svobody a flexibility při využívání veřejné bločenky, byla jedinou možností nová bločenka. To však znamenalo spoustu práce: vytvoření a zavádění všech prvků infrastruktury, vyčerpávající testování atd.

Na konci roku 2013 začal mladý programátor a Bitcoinový nadšenec Vitalik Buterin přemýšlet o dalším rozšiřování možností Bitcoinu a Mastercoinu (nadstavbový protokol, který rozšiřoval Bitcoin, aby nabízel základní chytré kontrakty). V říjnu téhož roku Vitalik navrhl obecnější přístup týmu Mastercoinu, který umožnil flexibilní a skriptovatelné (ale nikoli Turingovsky úplné) chytré kontrakty nahrazující specializovaný jazyk chytrých kontraktů, který používal Mastercoin. Přestože byl tým Mastercoinu ohromen, byl tento návrh příliš radikální změnou, než aby se vešly do jejich harmonogramu budoucích plánů.

V prosinci 2013 začal Vitalik sdílet technickou dokumentaci (whitepaper), která nastínila myšlenku Etherea: Turingovsky úplná, univerzální bločenka. Několik desítek lidí vidělo tento první návrh a nabídlo zpětnou vazbu, což Vitalikovi pomohlo tento návrh vyvinout.

Oba autoři této knihy obdrželi první návrhr technické dokumentace a komentovali jej. Andreas M. Antonopoulos byl nápadem zaujat a položil Vitalikovi mnoho otázek ohledně použití samostatné bločenky k prosazování konsensuálních pravidel pro provádění chytrých kontraktů a důsledků plynoucích z využitou Turingovsky úplného jazyka. Andreas nadále sledoval pokrok Etherea s velkým zájmem, ale byl v raných fázích psaní své knihy *Mastering Bitcoin* a přímo nezúčastnil raných prací na Ethereu. "Wood, Dr. Gavin","and birth of Ethereum")Dr. Gavin Wood byl však jedním z prvních lidí, který oslovil Vitalika a nabídl mu pomoc svými programovacími schopnostmi v jazyku C ++. Gavin se stal spoluzakladatelem Etherea, návrhářem zdrojového kódu a technickým ředitelem.

Jak Vitalik popisuje ve svém "Ethereum Prehistory" post [http://bit.ly/2T2t6zs]:

> To byl čas, kdy byl Ethereum protokol zcela mým vlastním výtvorem. Od této chvíle se však do této skupiny začali zapojovat noví účastníci. Zdaleka nejvýznamnějším na straně protokolu byl Gavin Wood ...
>
> Gavin má také velkou zásluhu na drobné změně vize jak nahlížet na Ethereum. Původní pohled byl platforma pro vytváření programovatelných peněz, s kontrakty uloženými v bločence, které mohou držet digitální aktiva a převádět je podle předem stanovených pravidel. Nový pohled je univerzální výpočetní platforma. Začalo to jemnými změnami v důrazu a terminologii a později se tento vliv zesílil rostoucím důrazem na spolupráci s „Web 3“, v němž bylo Ethereum považováno za jeden díl sady decentralizovaných technologií, další dva byly Whisper a Swarm.

Od prosince 2013 Vitalik a Gavin tuto myšlenku zdokonalovali a rozvíjeli a společně vytvořili vrstvu protokolu, která se stala Ethereem.

Zakladatelé Etherea přemýšleli o bločence bez konkrétního účelu, která by mohla podporovat *programování* široké škály aplikací. Myšlenkou bylo, že vývojář mohl s využitím univerzální bločenky, jako je Ethereum, naprogramovat svou konkrétní aplikaci, aniž by musel implementovat základní mechanismy sítí typu peer-to-peer, bločenek, algoritmů dosahování konsensu, atd. Platforma Ethereum byla navržena tak, aby abstraktovala od těchto podrobnosti a poskytovala deterministické a bezpečné programovací prostředí pro decentralizované bločenkové aplikace.

Stejně tak Satoshi, Vitalik a Gavin nevymysleli jen novou technologii; kombinovali nové vynálezy s existujícími technologiemi novým způsobem a dodali prototypový kód, aby dokázali světu své nápady.

Zakladatelé pracovali roky, budovali a vylepšovali vizi. A 30. července 2015 byl vytěžen první Ethereum blok. Světový počítač začal sloužit světu.

Článek Vitalika Buterina „A Prehistory of Ethereum“ vyšel v září 2017 a poskytuje fascinující pohled na první okamžiky Etherea z první osoby.

Můžete si ji přečíst na https://vitalik.ca/general/2017/09/14/prehistory.html.

Čtyři etapy vývoje Etherea

Vývoj Etherea byl naplánován do čtyřech různých fází, přičemž v každé fázi mělo dojít k zásadním změnám. Fáze může zahrnovat dílčí verze, známé jako „tvrdá rozštěpení," které mění funkčnost způsobem, který není zpětně kompatibilní.

Čtyři hlavní vývojové fáze jsou kódově označeny *Frontier*, *Homestead*, *Metropolis* a *Serenity*. Nadcházející tvrdá rozštěpení, která dosud proběhla (nebo se plánují), jsou označeny jako *Ice Age*, *DAO*, *Tangerine Whistle*, *Spurious Dragon*, *Byzantium* a *Constantinople*. Jak fáze vývoje, tak i nadcházející tvrdá rozštěpení jsou zobrazeny na následující časové ose, která je „datována" číslem bloku:

Blok #0

Frontier - Počáteční fáze projektu Ethereum, která trvá od 30. července 2015 do března 2016.

Blok #200 000

Ice Age - Tvrdě rozštěpení, které zavedlo exponenciální zvyšování obtížnosti těžby, což motivuje přechod k důkazu podílem, až bude připraven.

Blok #1 150 000

Homestead - Druhá etapa Etherea, zahájená v březnu 2016.

Blok #1,192,000

DAO - Tvrdé rozštěpení, které odškodnilo oběti útoku na kontrakt The DAO a způsobilo rozdělení Etherea na dva konkurenční systémy: Ethereum a Ethereum Classic.

Blok #2 463 000

Tangerine Whistle - Tvrdé rozštěpení, které změnilo výpočet plynu pro určité vstupně-výstupní operace a ze stavu systému vymazalo data vytvořená během útoku odepření služby (DoS), který využil nízké náklady na plyn při těchto operacích.

Blok #2 675 000

Spurious Dragon - Tvrdě rozštěpení zabraňující dalším způsobům DoS útoku a další vyčištění stavu systému. Také byl přidán Mechanismus ochrany proti útoku (replay attack), při kterém by mohly být v hlavní síti opakovány transakce, které předtím proběhly na testovací síti.

Blok #4,370,000

> *Metropolis Byzantium* - Metropolis je třetí etapa Etherea, aktuální v době psaní této knihy, spuštěná v říjnu 2017. Byzantium je první ze dvou tvrdých rozštěpení plánovaných pro Metropolis.

Po Byzantium je pro Metropolis naplánováno ještě jedno tvrdé rozštěpení Constantinople. Po Metropolis bude následovat závěrečná fáze nasazení Etherea, kódově pojmenovaná Serenity.

Ethereum: bločenka pro univerzální účely

Původní bločenka, konkrétně Bitcoinová bločenka, sleduje stav jednotek bitcoinů a jejich vlastnictví. Můžete o Bitcoinu přemýšlet jako o distribuovaném konsenzusu o *stavu stroje*, kde transakce způsobují globální *změnu stavu* a mění vlastnictví mincí. Přechody stavu jsou omezeny pravidly konsensu, které umožňují všem účastníkům (nakonec) konvergovat ke společnému (konsensuálnímu) stav systému poté, co bylo vytěženo několik bloků.

Ethereum je také distribuovaný stavový stroj. Ale namísto pouhého sledování stavu vlastnictví měny, Ethereum sleduje přechody stavů z obecného datového úložiště, tj. úložiště, ve kterém mohou být uložena libovolná data vyjádřitelná jako dvojice *klíč–hodnota*. Úložiště dvojic klíč-hodnota umožňuje k zadanému klíči nalézt k němu asociovanou hodnotu. Například klíčem může být „Název knihy“ a k němu asociovanou hodnotou „Mastering Ethereum“. V některých ohledech to slouží ke stejnému účelu jako model ukládání dat paměti *Random Access Memory* (RAM) používaný ve většině běžných počítačů. Ethereum má paměť, která ukládá kód i data, a používá bločenku Etherea ke sledování změn této paměti v průběhu času. Stejně jako běžný počítač s uloženým programem může Ethereum načíst kód do svého stavového stroje a *spustit* tento kód, přičemž výsledné změny stavu uloží do své bločenky. Dva z kritických rozdílů od většiny univerzálních počítačů spočívají v tom, že změny stavu Etherea se řídí pravidly konsensu a stav je distribuován globálně. Ethereum odpovídá na otázku: „Co kdybychom mohli sledovat libovolný stav a naprogramovat stavový stroj tak, aby vytvořil celosvětový počítač pracující na základě konsensu?“

Součásti Etherea

V Ethereu jsou následující součásti bločenkového systému, podrobněji jsou popsané v <<blockchain_components> >:

P2P síť

Ethereum běží na hlavní síti *Ethereum*, která je adresovatelná na TCP portu 30303, a provozuje protokol nazvaný *ĐΞVp2p*.

Pravidla konsensu

Pravidla Ethereum konsensu jsou definována v referenční specifikaci, Žluté knize (viz Další čtení).

Transakce

Ethereum transakce jsou síťové zprávy, které zahrnují (mimo jiné) odesílatele, příjemce, hodnotu a užitečná data.

Stavový stroj

Ethereum přechody stavu jsou zpracovávány virtuálním strojem *Ethereum Virtual Machine* (EVM), což je virtuální stroj založený na zásobníkové architektuře, který provádí *bajtkód* (instrukce strojového jazyka). Programy EVM, nazývané „chytré kontrakty“, jsou psány ve vysokoúrovňových jazycích na (např. Solidity) a kompilovány do bajtkódu pro provedení v EVM.

Datové struktury

Stav Etherea je uložen lokálně v každém uzlu jako *databáze* (obvykle Google LevelDB), která obsahuje transakce a stav systému v serializované hašované datové struktuře zvané *Merkle Patricia Strom*.

Algoritmus konsensu

Ethereum používá Bitcoinový konsenzuální model Nakamoto Consensus, který používá sekvenční bloky s jedním podpisem, vážené podle důležitosti důkazem prací (PoW) pro určení nejdelšího řetězce a tedy současného stavu. V blízké budoucnosti se však plánuje přechod na vážený hlasovací systém důkazu podílem (PoS), nazvaný *Casper*.

Ekonomické zabezpečení

Ethereum v současné době používá PoW pomocí algoritmu zvaného *Ethash*, ale to bude v budoucnosti nakonec zrušeno s přechodem na PoS.

Klienti

Ethereum má několik interoperabilních implementací klientského softwaru, z nichž nejvýznamnější jsou *Go-Ethereum* (*Geth*) a *Parity*.

Další čtení

Následující odkazy poskytují další informace o zde zmíněných technologiích:

- Ethereum Žlutá kniha (Yellow Paper): https://ethereum.github.io/yellowpaper/paper.pdf

Béžová kniha (Beige Paper), přepis žluté knihy pro širší publikum v méně formálním jazyce: https://github.com/chronaeon/beigepaper

- ĐΞVp2p síťový protokol: http://bit.ly/2quAlTE
- Ethereum virtuální stroj - seznam zdrojů: http://bit.ly/2PmtjiS
- Databáze LevelDB (nejčastěji používaná k ukládání lokální kopie bločenky): https://github.com/google/leveldb
- Merkle Patricia stromy: https://github.com/ethereum/wiki/wiki/Patricia-Tree
- Ethash PoW algoritmus: https://github.com/ethereum/wiki/wiki/Ethash
- Casper PoS v1 implementační příručka: http://bit.ly/2DyPr3l
- Go-Ethereum (Geth) klient: https://geth.ethereum.org/
- Ethereum Parity klient: https://parity.io/

Ethereum a Turingovská úplnost

Jakmile začnete číst o Ethereu, okamžitě se setkáte s termínem „Turingovská úplnost.“ Ethereum, říkají, na rozdíl od Bitcoinu, je Turingovsky kompletní. Co přesně to znamená?

Termín odkazuje na anglického matematika Alana Turinga, který je považován za otce informatiky. V roce 1936 vytvořil matematický model počítače skládající se ze stavového stroje, který manipuluje se symboly čtením a zápisem do sekvenční paměti (připomínající nekonečnou papírovou pásku). S tímto konstruktem Turing pokračoval, aby poskytl matematický základ k zodpovězení otázky ohledně *univerzální vyčíslitelnosti*, což znamená, zda jsou všechny problémy řešitelné. Dokázal, že existují třídy problémů, které nejsou vyčíslitelné. Konkrétně prokázal, že *problém zastavení* (zda je

možné, s ohledem na libovolný program a jeho vstup, určit, zda program nakonec přestane běžet) není řešitelný.

Alan Turing dále definoval že systém bude *Turingovsky úplný*, pokud může být použit k simulaci jakéhokoli Turingova stroje. Takový systém se nazývá *Univerzální Turingův stroj* (UTM).

Schopnost Etherea provádět uložený program ve stavovém stroji zvaném Ethereum Virtual Machine využívajícím čtení a zápis dat do paměti z něj činí Turingovsky úplný systém, tedy UTM. Ethereum umí vypočítat jakýkoli algoritmus, který lze vypočítat jakýmkoli Turingovým strojem, vzhledem k omezení konečné paměti.

Průkopnickou inovací Etherea je kombinace univerzální výpočetní architektury počítače s uloženým programem a decentralizovanou bločenkou, čímž se vytvoří distribuovaný počítač s jednoznačným stavem (singleton). Ethereum programy běží „všude“, ale přesto vytvářejí společný stav, který je zajištěn pravidly konsensu konsenzus.

Turingová úplnost jako „vylepšení“

Když uslyšíte, že Ethereum je Turingovsky úplné, můžete dojít k závěru, že toto je *vlastnost*, které nějakým způsobem chybí v systému, který je Turingovsky neúplný. Spíše je to naopak. Turingské úplnosti je velmi snadné dosáhnout; ve skutečnosti nejjednodušší známý Turingovsky úplný stavový stroj [http://bit.ly/2ABft33] má 4 stavy a používá 6 symbolů, definice přechodů mezi stavy je dlouhá pouze 22 instrukcí. Ve skutečnosti se někdy zjistí, že systémy jsou „nechtěně Turingovsky kompletní“. Zábavný odkaz na takové systémy lze nalézt na adrese http://bit.ly/2Og1VgX [].

Turingova úplnost je však velmi nebezpečná, zejména v systémech s otevřeným přístupem, jako jsou veřejné bločenky, kvůli problému zastavení, kterého jsme se dotkli dříve. Například moderní tiskárny jsou Turingovsky úplné a mohou jim být dány k tisku soubory, které je dovedou do zmrazeného stavu. Skutečnost, že Ethereum je Turingovsky úplné, znamená, že Ethereum může spočítat jakýkoli program jakékoli složitosti. Tato flexibilita však přináší některé komplikované problémy se zabezpečením a správou prostředků. Nereagující tiskárnu lze vypnout a znovu zapnout. To není možné s veřejnou bločenkou.

Důsledky Turingovské úplnosti

Turing dokázal, že nemůžete předpovědět, zda program skončí, pokud ho budeme simulovat na počítači. Zjednodušeně řečeno, nemůžeme předpovědět cestu programu bez jeho spuštění.

Turinovsky úplné systémy mohou běžet v „nekonečných smyčkách“, což je termín používaný (v zjednodušení) k popisu programu, který nikdy neskončí. Je triviální vytvořit program, který spouští smyčku, která nikdy nekončí. Neúmyslné nekonečné smyčky však mohou vzniknout bez varování kvůli složitým interakcím mezi počátečními podmínkami a kódem. V případě Etherea to představuje výzvu: každý zúčastněný uzel (klient) musí ověřit každou transakci a spustit jakékoli chytré kontrakty, které volá. Ale jak Turing dokázal, Ethereum nedokáže předvídat, zda chytrý kontrakt skončí, nebo jak dlouho to bude trvat, aniž by to vlastně nechal běžet (možná běží navždy). Ať už náhodou nebo úmyslně, chytrý kontrakt může být vytvořena tak, že běží navždy, když se ho uzel pokouší ověřit. To je skutečně útok DoS. A samozřejmě, mezi programem, který trvá milisekundu, a programem, který běží věčně, je nekonečná řada ošklivých, programů, které spotřebovávají paměť a další systémové prostředky, přehřívají procesor, jednoduše plýtvají prostředky. Ve světovém počítači program, který zneužívá zdroje, zneužívá světové zdroje. Jak Ethereum omezuje zdroje používané v chytrých kontraktech, pokud nemůže předvídat využití zdrojů předem?

Jako odpověď na tuto výzvu zavádí Ethereum měřící mechanismus nazvaný *plyn*. Když EVM vykonává chytrý kontrakt, pečlivě počítá každou instrukci (výpočet, přístup k datům atd.). Každá instrukce má předem stanovené náklady v jednotkách plynu. Když transakce vyvolá vykonání chytrého kontraktu, musí tato transakce obsahovat množství plynu, které stanoví horní limit toho, co může být spotřebováno při provádění chytrého kontraktu. EVM ukončí provádění, pokud množství plynu spotřebované výpočtem překročí množství plynu dostupného v transakci. Plyn je mechanismus, který Ethereum používá k umožnění Turingovsky úplného výpočtu při současném omezení zdrojů, které může kterýkoli program spotřebovat.

Další otázka zní: „Jak lze získat plyn, kterým se platí za výpočet na Ethereum světovém počítači?“ Na žádné burze nenajdete plyn. Lze jej zakoupit pouze jako součást transakce a lze jej za něj zaplatit pouze etherem. Ether musí být zaslán spolu s transakcí a musí být výslovně vyčleněn na nákup plynu spolu s přijatelnou cenou plynu. Stejně jako u čerpadla není cena plynu pevně stanovena. Pro transakci se nakupuje plyn, výpočet se provede a veškerý nepoužitý plyn se vrátí zpět odesílateli transakce.

Od univerzální bločenky po decentralizované aplikace (DApps)

Ethereum začalo jako způsob, jak vytvořit univerzální bločenku, kterou lze naprogramovat pro různé účely. Ale velmi rychle se vize Etherea rozšířila a stalo se platformou pro programování DApps. DApps představují širší perspektivu než chytré kontrakty. DApp je přinejmenším chytrý

kontrakt a webové uživatelské rozhraní. Obecněji řečeno, DApp je webová aplikace, která je vytvořena na základě otevřených, decentralizovaných infrastrukturních služeb typu peer-to-peer.

DApp se skládá alespoň z:

- Chytrého kontraktu na bločence
- Webového uživatelské rozhraní

Mnoho DApps navíc obsahuje další decentralizované komponenty, jako například:

- Decentralizovaný (P2P) protokol ukládání dat a platformu
- Decentralizovaný (P2P) komunikační protokol a platformu

Můžete vidět DApps hláskované jako *ĐApps*. Znak Đ je latinský znak zvaný „ETH“, který odkazuje na Ethereum. Chcete-li zobrazit tento znak, použijte kódový bod Unicode 0xD0, nebo případně znakovou entitu HTML eth (nebo desítkovou entitu #208).

Třetí věk internetu

V roce 2004 se dostal ve známost termín „web 2.0 “, který popisuje vývoj webu směrem k obsahu vytvářenému uživateli, responzivnímu rozhraní a interaktivitě. Web 2.0 není technická specifikace, ale spíše termín popisující nové zaměření webů aplikace [].

Koncept DApps přivádí World Wide Web do jeho další přirozené vývojové fáze a to zavedení decentralizace pomocí protokolů typu peer-to-peer do všech aspektů webové aplikace. Termín používaný k popisu tohoto vývoje je *web3*, což znamená třetí „verzi“ webu. Web3, který poprvé navrhl Dr. Gavin Wood, představuje novou vizi a zaměření na webové aplikace: od centrálně vlastněných a spravovaných aplikací po aplikace postavené na decentralizovaných protokolech.

V pozdějších kapitolách prozkoumáme JavaScript knihovnu Ethereum web3.js, která propojuje aplikace JavaScriptu spuštěné ve vašem prohlížeči s bločenkou Etherea. Knihovna web3.js také obsahuje rozhraní k úložné síti P2P zvané *Swarm* a službu zasílání zpráv P2P zvaná *Whisper*. Díky těmto třem komponentám zahrnutým do této JavaScript knihovny spuštěné ve webovém prohlížeči mají vývojáři kompletní sadu pro vývoj aplikací, která jim umožňuje vytvářet web3 DApps.

Kultura vývoje Etherea

Zatím jsme hovořili o tom, jak se cíle a technologie Ethereum liší od cílů a bločenek, které mu předcházely., jako Bitcoin. Ethereum má také velmi odlišnou kulturu svého vývoje.

V Bitcoinu se vývoj řídí konzervativními principy: všechny změny jsou pečlivě studovány, aby se zajistilo, že nebude narušen žádný ze stávajících systémů. Z větší části jsou změny implementovány pouze tehdy, jsou-li zpětně kompatibilní. Stávající klienti se mohou přihlásit, a dokonce mohou pokračovat v činnosti, i pokud se nerozhodnou upgradovat.

V Ethereum komunita se řídí odlišnou kulturou vývoje, zaměřuje se spíše na budoucnost než na minulost. (Ne zcela vážná) mantra je „pohybujte se rychle a ničte věci“. Pokud je nutná změna, je implementována, i když to znamená zneplatnění předchozích předpokladů, porušení kompatibility nebo nucení klientů k aktualizaci. Vývojová kultura Etherea se vyznačuje rychlými inovacemi, rychlým vývojem a ochotou zavést do budoucna vylepšení, i když je to na úkor určité zpětné kompatibility.

To pro vás jako vývojáře znamená, že musíte zůstat flexibilní a připraveni znovu vybudovat svou infrastrukturu, protože se změní některé základní předpoklady. Jednou z velkých výzev, kterým vývojáři v Ethereum čelí, je inherentní rozpor mezi nasazením kódu do neměnného systému a vývojovou platformou, která se stále vyvíjí. Nemůžete jednoduše upgradovat své chytré kontrakty. Musíte být připraveni nasadit nové, migrovat uživatele, aplikace a fondy a začít znovu.

Je ironií, že to také znamená, že cíl budování systémů s větší autonomií a méně centralizovanou kontrolou není stále plně realizován. Autonomie a decentralizace vyžadují na platformě o něco více stability, než se v Etereu pravděpodobně dostanete v příštích několika letech. Chcete-li platformu „vyvinout“, musíte být připraveni zrušit a restartovat vaše chytré kontrakty, což znamená, že si musíte nad nimi zachovat určitý stupeň kontroly.

Pozitivní je, že Ethereum postupuje velmi rychle. Je jen málo příležitostí pro „zabývání se nepodstatnými detaily,“ zastavit vývoj tím, že se bude dohadovat o drobných detailech, například o tom, jak postavit kůlnu na kola v zadní části jaderné elektrárny. Pokud začnete zabývat kůlnou na kola, najednou zjistíte, že zatímco jste byli rozptýleni, zbytek vývojového týmu změnil plán a odhodil kola ve prospěch autonomního vznášedla.

Nakonec se vývoj platformy Ethereum zpomalí a její rozhraní se zafixují. Mezitím je však hnací silou inovace. Měli byste raději držet krok, protože nikdo pro vás nezpomalí.

Proč se učit Ethereum?

Bločenky mají velmi strmou křivku učení, protože kombinují více disciplín do jedné domény : programování, zabezpečení informací, kryptografii, ekonomiku, distribuované systémy, sítě typu peer-to-peer atd. Ethereum dělá tuto křivku učení mnohem méně strmou, takže můžete začít rychle. Ale těsně pod povrchem klamně jednoduchého prostředí leží mnohem víc. Jak se učíte a začnete hlouběji hledat, je tu vždy další vrstva složitosti a údivu.

Ethereum je skvělá platforma pro učení o bločenkách a buduje masivní komunitu vývojářů rychleji než kterákoli jiná bločenková platforma. Ethereum je více než jakýkoli jiný nástroj pro vývojáře vytvořený vývojáři pro vývojáře. Vývojář, který je obeznámen s JavaScriptovými aplikacemi, se může dostat do Etherea velmi rychle a začít programovat funkční kód. Prvních několik let života Etherea bylo běžné vidět trička, jak oznamují, že můžete vytvořit token v pouhých pěti řádcích kódu. Jde samozřejmě o dvojsečný meč. Je snadné psát kód, ale je velmi těžké psát *dobrý* a *bezpečný* code.

Co vás tato kniha naučí

Tato kniha se ponořuje do Etherea a zkoumá všechny komponenty. Začnete jednoduchou transakcí, prozkoumáte, jak to funguje, vytvoříte jednoduchý chytrý kontrakt, vylepšíte ho a sledujete jeho cestu systémem Ethereum.

Dozvíte se nejen, jak používat Ethereum - jak to funguje -, ale také proč je navrženo tak, jak je. Budete schopni porozumět tomu, jak každá z jeho částí funguje a jak do sebe zapadají zapadají a proč.

Základy Etherea

V této kapitole začneme zkoumat Ethereum, naučíme se, jak používat peněženky, jak vytvářet transakce, a také jak provozovat jednoduchý chytrý kontrakt.

Měnové jednotky Etherea

Měnová jednotka Etherea se nazývá *ether*, označená také jako „ETH“ nebo se symboly Ξ (z řeckého písmene Ksí, které vypadá jako stylizované velké E) nebo, méně často, : například 1 ether, nebo 1 ETH, nebo Ξ1, nebo 1.

Použijte znak Unicode U+039E pro Ξ a U+2666 pro .

Ether jde dělit na menší jednotky, až na nejmenší možnou jednotku, která se jmenuje *wei*. Jeden ether je 1 trilion wei (1 * 10 ^ 18 ^ nebo 1 000 000 000 000 000 000). Občas lidé používají pojem „Ethereum“ i jako název jeho měnové jednotky, ale toto je běžná začátečnická chyba. Ethereum je systém, ether je měna.

Množství etheru je v Ethereu vždy interně reprezentována jako celé číslo bez znaménka vyjádřené v jednotkách wei. Když provedete transakci 1 etheru, transakce to převede na hodnotu 10000000000000000000 wei.

Různá označení dílčích měnových jednotek Etherea mají jednak vědecké jméno odvozené pomocí Mezinárodního systému jednotek (*SI*), tak i hovorové jméno, které vzdává hold mnoha velkým myslitelům v oblasti výpočetní techniky a kryptografie.

Dílčí měnové jednotky Etherea a jejich názvy ukazuje různé jednotky, jejich hovorové (běžné) názvy a jejich SI názvy. V souladu s interní reprezentací hodnoty tabulka zobrazuje všechny nominální hodnoty ve wei (první řádek), etherem je uvedeno na 7. řádku jako 10^{18} wei.

Table 1. Dílčí měnové jednotky Etherea a jejich názvy

Hodnota (ve wei)	Exponent	Běžný název	Označení dle SI
1	1	wei	Wei
1,000	10^3	Babbage	Kilowei nebo femtoether

Hodnota (ve wei)	Exponent	Běžný název	Označení dle SI
1,000,000	10^{6}	Lovelace	Megawei nebo picoether
1,000,000,000	10^{9}	Shannon	Gigawei nebo nanoether
1,000,000,000,000	10^{12}	Szabo	Microether nebo micro
1,000,000,000,000,000	10^{15}	Finney	Milliether nebo milli
1,000,000,000,000,000,000	*10^{18}*	*Ether*	*Ether*
1,000,000,000,000,000,000,000	10^{21}	Grand	Kiloether
1,000,000,000,000,000,000,000,000	10^{24}		Megaether

Výběr Ethereum peněženky

Pojem „peněženka“ může znamenat mnoho věcí, i když všechny jsou ve vzájemném vztahu při každodenním používání se scvrkávají na téměř stejnou věc. Termín „peněženka“ budeme používat pro softwarovou aplikaci, která vám pomůže spravovat váš Ethereum účet. V krátkosti, Ethereum peněženka je vaší bránou do systému Ethereum. Drží vaše klíče a může vytvářet a odesílat transakce vaším jménem. Výběr Ethereum peněženky může být obtížný, protože existuje mnoho různých možností s různými funkcemi a ovládáním. Některé jsou vhodnější pro začátečníky a jiné jsou vhodné pro odborníky. Samotná platforma Ethereum se stále zdokonaluje a „nejlepší“ peněženky se často přizpůsobují změnám, které přicházejí s upgradem platformy.

Ale nebojte se! Pokud si vyberete peněženku a nelíbí se vám, jak funguje - nebo pokud se vám líbí zpočátku, ale později chcete vyzkoušet něco jiného - můžete snadno měnit peněženky. Musíte pouze provést transakci, která odešle vaše peníze ze staré peněženky do nové peněženky, nebo exportovat své soukromé klíče ze staré peněženky a importovat je do nové peněženky.

Jako příklady v knize jsme vybrali tři různé typy peněženek: mobilní peněženku, peněženku pro stolní počítače a webovou peněženku. Vybrali jsme tyto tři peněženky, protože představují širokou škálu složitosti a funkcí. Výběr těchto peněženek však není potvrzením jejich kvality nebo bezpečnosti. Jsou prostě dobrým výchozím místem pro demonstrace a testování.

Pamatujte, že aby aplikace peněženky fungovala, musí mít přístup k vašim soukromým klíčům,

takže je nezbytné, abyste stahovali a používali pouze peněženkové aplikace ze zdrojů, kterým důvěřujete. Naštěstí obecně platí, že čím populárnější je peněženková aplikace, tím důvěryhodnější bude. Je však dobré se vyhnout „vložení všech vajec do jednoho košíku“ a rozložit své Ethereum účty na několik peněženek.

Následuje přehled několik dobrých peněženek pro začátečníky:

MetaMask

MetaMask je rozšíření prohlížeče, které běží v vašem prohlížeči (Chrome, Firefox, Opera nebo Brave). Je snadno použitelný a vhodný pro testování, protože je schopen se připojit k celé řadě Ethereum uzlů a testovacích bločenek. MetaMask je webová peněženka.

Jaxx

Jaxx je multiplatformní peněženka podporující mnoho měn, která běží na různých operačních systémech, včetně Android, iOS, Windows, macOS, a Linux. Je oblíben novými uživateli, protože je navržen pro jednoduché a snadné použití. Podle toho kde ho nainstalujete, může být Jaxx mobilní peněženkou nebo peněženkou pro stolní počítače.

MyEtherWallet (MEW)

MyEtherWallet je webová peněženka, která běží v libovolném prohlížeči. Má mnoho sofistikovaných funkcí, které prozkoumáme v mnoha našich příkladech. MyEtherWallet je webová peněženka.

Emerald Wallet

Emerald Wallet je navržen pro práci s bločenkou Etherea Classic, ale je kompatibilní s jinými bločenkami založenými na Ethereu. Je to peněženka s otevřeným zdrojovým kódem, určená pro stolní počítače, a funguje pod Windows, macOS a Linux. Emerald Wallet může provozovat úplný uzel nebo se připojit k veřejnému vzdálenému uzlu a pracovat ve „odlehčeném“ režimu. Má také doprovodný nástroj pro provádění všech operací z příkazového řádku.

Začneme instalací MetaMask na stolní počítač - nejprve si ale krátce vysvětlíme její ovládání a správu klíčů.

Kontrola a odpovědnost

Otevřené bločenky jako Ethereum jsou důležité, protože fungují jako *decentralizovaný* systém. To

znamená spoustu věcí, ale jedním zásadním aspektem je to, že každý uživatel Etherea může - a měl by - kontrolovat své soukromé klíče, což jsou věci, které řídí přístup k finančním prostředkům a chytrým kontraktům. Někdy nazýváme kombinaci přístupu k finančním prostředkům a chytrým kontraktům „účet“ nebo „peněženka“. Tyto výrazy mohou být ve docela složitou funkcionalitu, takže se na ně podrobněji podíváme později. Základní princip je však jednoduchý, jeden soukromý klíč se rovná jednomu účtu. Někteří uživatelé se rozhodnou vzdát kontroly nad svými soukromými klíči pomocí úschovny třetí strany, jako je například online burza. V této knize vás naučíme, jak ovládat a spravovat vlastní soukromé klíče.

S kontrolou přichází velká odpovědnost. Pokud ztratíte soukromé klíče, ztratíte přístup ke svým prostředkům a kontraktům. Nikdo vám nemůže pomoci znovu získat přístup - vaše prostředky budou navždy zamčené. Zde je několik tipů, které vám pomohou spravovat tuto odpovědnost:

- Neimprovizujte v zabezpečení. Používejte osvědčené standardní přístupy.
- Čím důležitější je účet (např. čím vyšší je hodnota kontrolovaných prostředků nebo čím významnější jsou kontrolované chytré kontrakty), tím vyšší by měla být přijatá bezpečnostní opatření.
- Nejvyšší míru zabezpečení dosahuje zařízení, které je fyzicky odděleno od sítě, ale tato úroveň není vyžadována pro každý účet.
- Nikdy neukládejte svůj soukromý klíč v nešifrované podobě, zejména digitálně. Naštěstí většina dnešních uživatelských rozhraní vám ani neumožní vidět nezašifrovaný soukromý klíč.
- Soukromé klíče lze uložit v zašifrované podobě jako digitální soubor „úložiště klíčů“. Protože jsou šifrováni, potřebují k jejich odemčení heslo. Až budete vyzváni, abyste si vybrali heslo, učiňte jej silným (tj. dlouhým a náhodným), zálohujte jej a nesdílejte. Pokud nemáte správce hesel, zapište si jej a uložte na bezpečném a tajném místě. Pro přístup k účtu potřebujete soubor úložiště klíčů i heslo.
- Neukládejte žádná hesla v digitálních dokumentech, digitálních fotografiích, screenshotech, online diskových jednotkách, šifrovaných PDF atd. Znova, neimprovizujte v oblasti zabezpečení. Použijte správce hesel nebo pero a papír.
- Když se zobrazí výzva k zálohování klíče jako mnemotechnické posloupnosti slov, pomocí pera a papíru vytvořte fyzickou zálohu. Nenechávejte tento úkol „na později“; zapomenete. Tyto zálohy lze použít k opětovnému vytvoření vašeho soukromého klíče v případě, že ztratíte všechna data uložená ve vašem systému, nebo pokud zapomenete nebo ztratíte své heslo.

Útočníci je však mohou také použít k získání vašich soukromých klíčů, takže je nikdy neukládejte digitálně a fyzickou kopii uložte bezpečně v uzamčené zásuvce nebo v bezpečí.

- Před převedením jakýchkoli velkých částek (zejména na nové adresy) nejprve proveďte malou testovací transakci (např. s hodnotou menší než jeden dolar) a počkejte na potvrzení přijetí.
- Při vytváření nového účtu začněte odesláním pouze malé testovací transakce na novou adresu. Jakmile obdržíte testovací transakci, zkuste odeslat znovu z tohoto účtu. Existuje spousta důvodů, proč se vytvoření účtu může pokazit, a pokud se to pokazilo, je lepší to zjistit s malou ztrátou. Pokud testy fungují, vše je v pořádku.
- Veřejní průzkumníci bloků jsou snadným způsobem, jak nezávisle zjistit, zda byla transakce přijata sítí. Toto pohodlí má však negativní dopad na vaše soukromí, protože své adresy odhalíte těmto průzkumníkům bloků, kteří vás mohou následně sledovat.
- Nikdy neposílejte peníze na žádnou z adres uvedených v této knize. Soukromé klíče jsou uvedeny v této knize a někdo si tyto peníze okamžitě vezme.

Nyní, když jsme se zabývali některými základními doporučenými postupy pro správu klíčů a zabezpečení, pojďme začít pracovat sí MetaMask!

Začínáme s MetaMaskem

("MetaMask","basics", id="ix_02intro-asciidoc3", range="startofrange"Otevřte prohlížeč Google Chrome a přejděte na https://chrome.google.com/webstore/category/extensions [].

Vyhledejte „MetaMask“ a klikněte na logo lišky. Měli byste vidět něco jako výsledek zobrazený v Detail stránky rozšíření MetaMask pro Chrome.

Figure 1. Detail stránky rozšíření MetaMask pro Chrome

Je důležité ověřit, že stahujete skutečné rozšíření MetaMask, protože někdy jsou lidé schopni propašovat škodlivá rozšíření skrz Google filtry. Skutečný MetaMask:

- Na řádce adresy zobrazuje ID nkbihfbeogaeaoehlefnkodbefgpgknn
- Je nabízen webem https://metamask.io
- Má více než 1 500 recenzí
- Má více než 1 000 000 uživatelů

Až se ujistíte, že se díváte na správné rozšíření, nainstalujte jej kliknutím na tlačítko „Add to Chrome“.

Vytvoření peněženky

Jakmile je MetaMask nainstalován, měla by se na panelu nástrojů prohlížeče zobrazit nová ikona (hlava lišky) . Začněte kliknutím na ní. Budete vyzváni k přijetí smluvních podmínek a poté k vytvoření nové Ethereum peněženky zadáním hesla (viz Stránka s heslem rozšíření MetaMask pro Chrome).

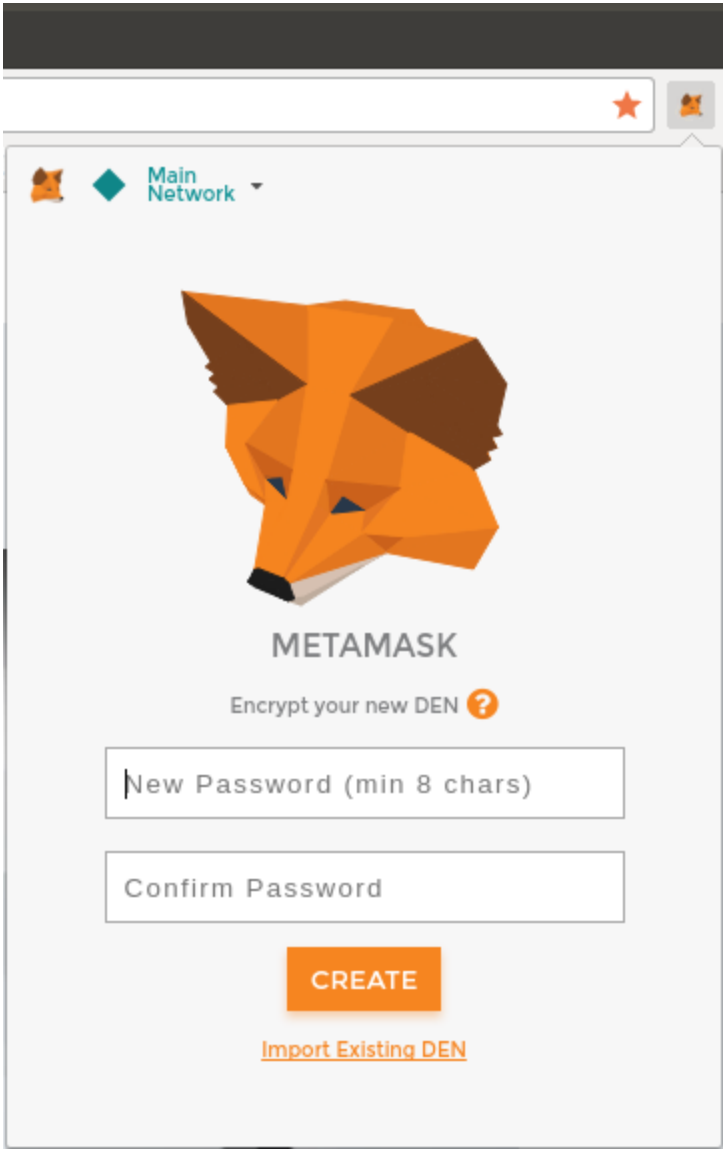

Figure 2. Stránka s heslem rozšíření MetaMask pro Chrome

Heslo ovládá přístup k MetaMasku, takže jej nemůže použít kdokoli s přístupem do vašeho prohlížeče.

Jakmile nastavíte heslo, MetaMask vygeneruje peněženku a zobrazí vám *mnemotechnickou zálohu* sestávající z 12 anglických slov (viz <<metamask_mnemonic> >). Tato slova lze použít v jakékoli kompatibilní peněžence k obnovení přístupu k vašim finančním prostředkům, pokud by se něco stalo MetaMaskem nebo s vaším počítačem. Pro toto obnovení nepotřebujete heslo; stačí 12 slov.

Zálohujte si 12 mnemotechnických slov na papír, dvakrát. Uložte obě papírové zálohy na dvou samostatných bezpečných místech, například v ohnivzdorném trezoru, v uzamčené zásuvce nebo v bezpečnostní schránce. Zacházejte s papírovými zálohami jako s penězi stejné hodnoty, jakou ukládáte do peněženky Ethereum. Každý, kdo má přístup k libovolné z těchto kopií 12 slov, může získat přístup a ukrást vaše peníze.

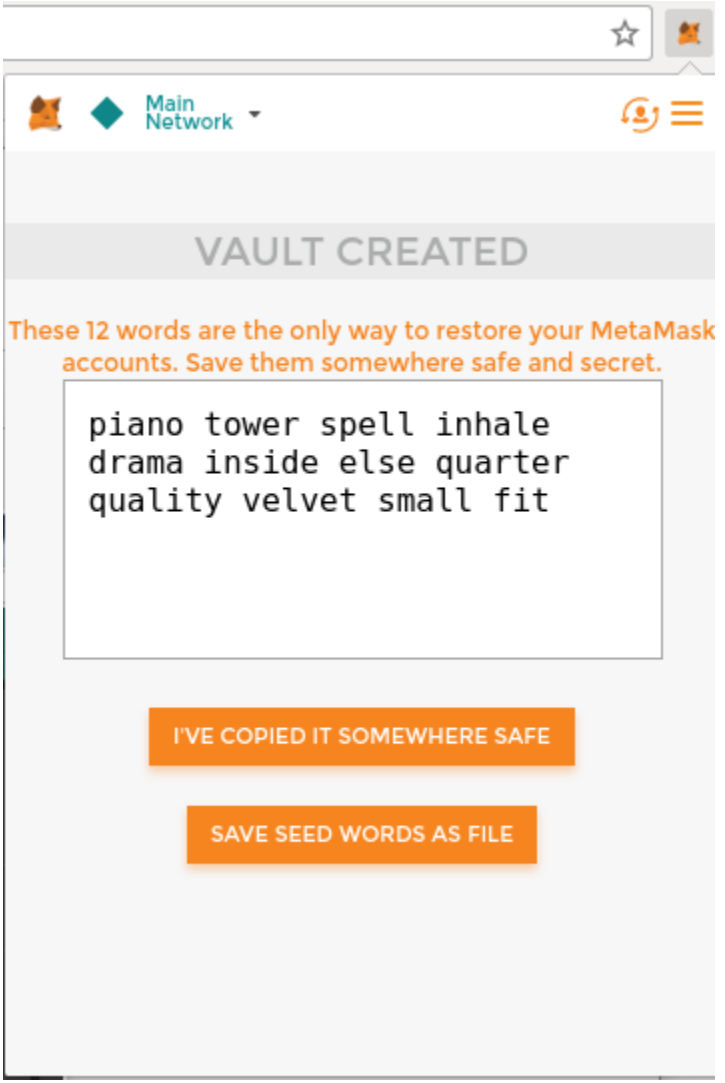

Figure 3. Mnemonická záloha vaší peněženky, vytvořená MetaMaskem

Jakmile potvrdíte, že jste mnemotechnickou zálohu bezpečně uložili, uvidíte podrobnosti vašeho

účtu Ethereum, jak je uvedeno v Váš účet Ethereum v MetaMasku.

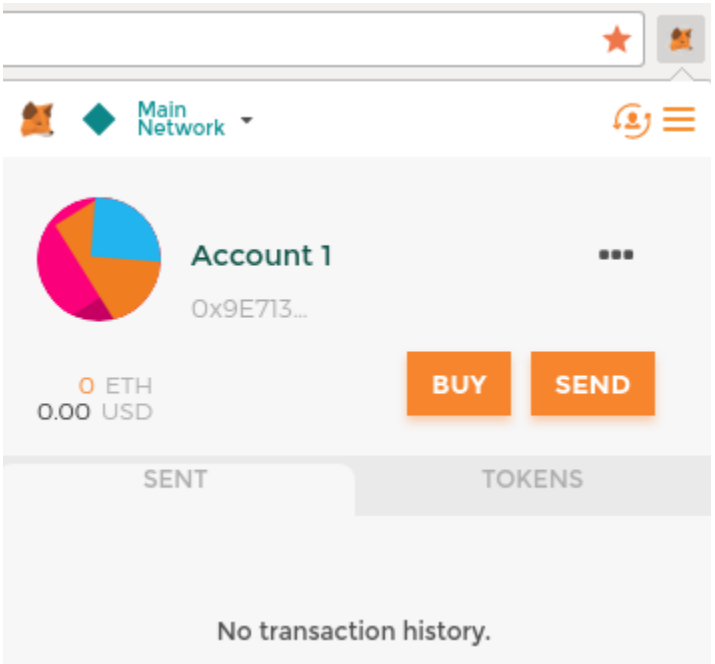

Figure 4. Váš účet Ethereum v MetaMasku

Na stránce vašeho účtu se zobrazuje název vašeho účtu (ve výchozím nastavení „Účet 1“), adresa Ethereum (v příkladu 0x9E713...) a barevná ikona, která vám pomůže vizuálně odlišit tento účet od ostatních účtů. V horní části stránky účtu můžete vidět, na které Ethereum síti aktuálně pracujete (v příkladu „Hlavní síť“).

Gratulujeme! Nastavili jste svojí první peněženku Ethereum.

Přepínání sítí

Jak můžete vidět na stránce účtu MetaMask, můžete si vybrat mezi několika Ethereum sítěmi. Ve výchozím nastavení se MetaMask pokusí připojit k hlavní síti. Dalšími možnostmi jsou veřejné testovací sítě, libovolný uzel Ethereum podle vašeho výběru nebo uzly provozující soukromé bločenky ve vašem počítači (localhost):

Hlavní Ethereum síť

Hlavní veřejná Ethereum bločenka. Skutečná ETH, skutečná hodnota a skutečné důsledky.

Ropsten testovací síť

Veřejná testovací bločenka Etherea. ETH v této síti nemá žádnou hodnotu.

Kovan testovací síť

Veřejná testovací bločenka Etherea a síť používající protokol dosahovaní konsensu typu důkaz autoritou (federované podpisy) zvaný Aura. ETH v této síti nemá žádnou hodnotu. Testovací síť Kovan je podporována pouze společností Parity. Ostatní Ethereum klienti používají protokol dosahovaní konsensu Clique, který byl navržen později, a který je také druhem důkazu autoritou.

Rinkeby testovací síť

Veřejná testovací Ethereum bločenka a síť, používající Clique konsensuální protokol typu důkaz autoritou (federované podepisování). ETH v této síti nemá žádnou hodnotu.

Localhost 8545

Připojuje se k uzlu spuštěnému na stejném počítači jako prohlížeč. Uzel může být součástí libovolné veřejné bločenky (hlavní nebo testovací) nebo soukromé testovací sítě.

Vlastní RPC

Umožňuje vám připojit MetaMask k libovolnému uzlu pomocí rozhraní RPC (vzdálené volání funkcí) kompatibilního s Geth. Uzel může být součástí libovolné nebo veřejné bločenky.

Vaše peněženka MetaMask používá stejný soukromý klíč a Ethereum adresu ve všech sítích, ke kterým se připojuje. Váš zůstatek Ethereum adres v každé Ethereum síti se však bude lišit. Vaše klíče mohou například ovládat ether a uzavírat kontrakty na Ropsten, ale ne na hlavní síti.

Získání testovacího éteru

Vaším prvním úkolem je získat nějaké finanční prostředky do vaší peněženky. Nebudete to dělat v hlavní síti, protože skutečný ether stojí peníze a manipulace s ním vyžaduje trochu více zkušeností. Prozatím naplníme peněženku pomocí testovacího etheru.

Přepněte MetaMask na *Ropsten Test Network*. Klikněte na „Deposit“ a poté na „Ropsten Test Faucet“. MetaMask otevře novou webovou stránku, jak ukazuje MetaMask Ropsten testovací kohoutek.

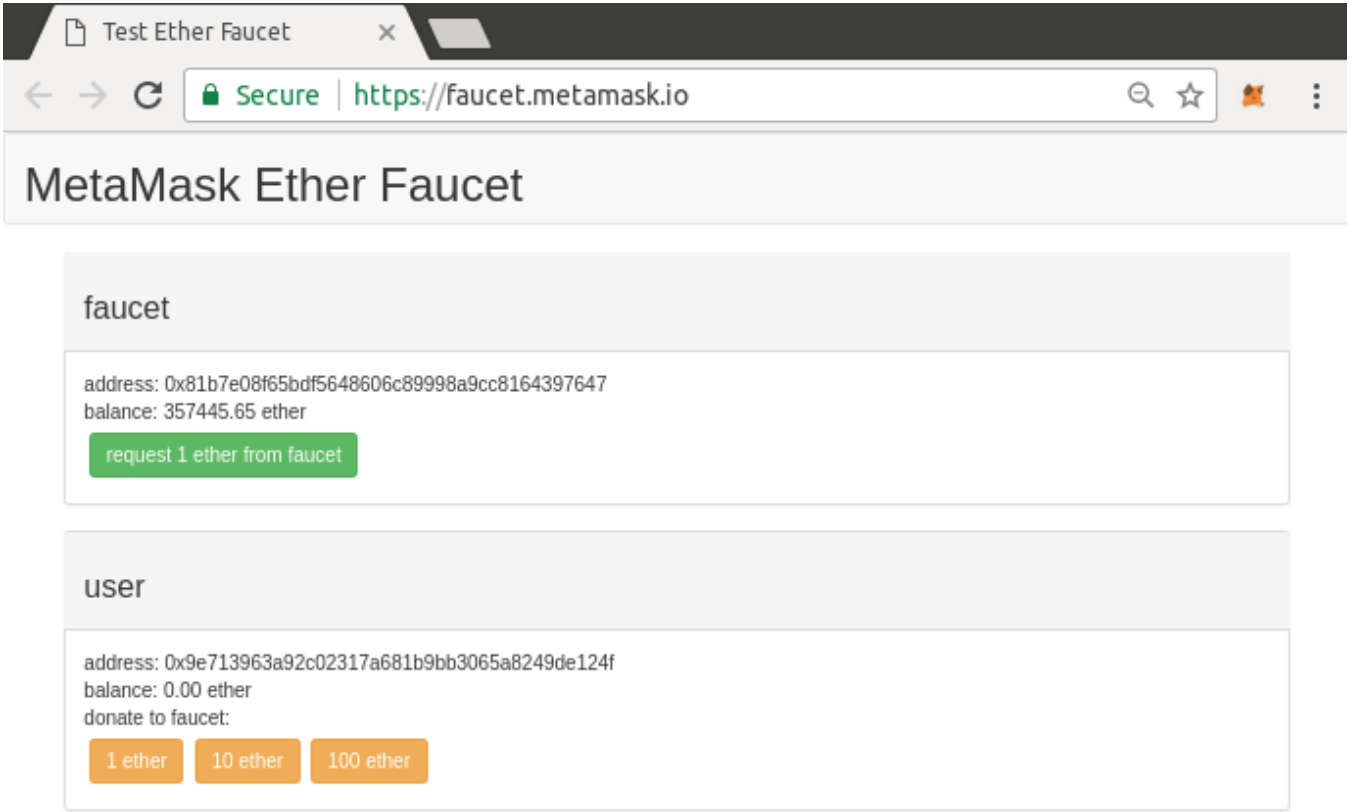

Figure 5. MetaMask Ropsten testovací kohoutek

Možná si všimnete, že webová stránka již obsahuje Ethereum adresu vaší peněženky MetaMask. MetaMask integruje webové stránky podporující Ethereum s vaší peněženkou MetaMask a může „vidět“ Ethereum adresy na webové stránce, což vám například umožňuje poslat platbu do internetového obchodu zobrazujícího adresu Ethereum. MetaMask může také vyplnit webovou stránku svou vlastní adresou peněženky jako adresou příjemce, pokud to webová stránka vyžaduje. Na této stránce aplikace kohoutek žádá MetaMask o adresu peněženky, kam má být odeslán testovací ether.

Klikněte na zelené tlačítko „request 1 ether from faucet“. Ve spodní části stránky se zobrazí ID transakce. Aplikace faucet vytvořila transakci - platba vám. ID transakce vypadá takto:

```
0x7c7ad5aaea6474adccf6f5c5d6abed11b70a350fbc6f9590109e099568090c57
```

Za pár vteřin bude těžaři Ropstenu vytěžena tato nová transakce a vaše peněženka MetaMask ukáže

zůstatek 1 ETH. Klikněte na ID transakce a váš prohlížeč vás přenese na *průzkumníka bloků*, což je web, který vám umožní vizualizovat a prozkoumat bloky, adresy a transakce. MetaMask používá Etherscan block explorer [https://etherscan.io/], jeden z nejpopulárnějších Ethereum průzkumníků bloků. Transakce obsahující platbu z kohoutku v testovací síti Ropsten je uvedena v Ethercan prohlížeč Ropsten bloků.

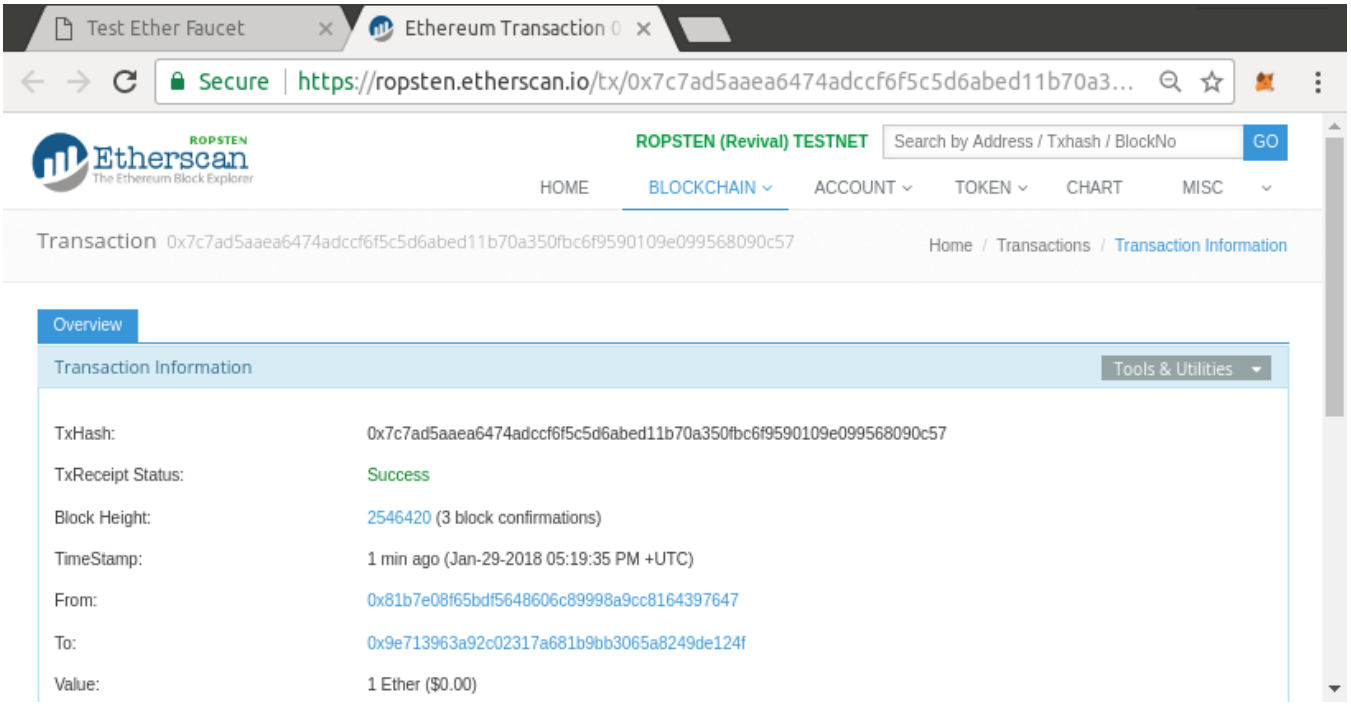

Figure 6. Ethercan prohlížeč Ropsten bloků

Transakce byla zaznamenána do Ropsten bločenkya může ji kdykoli zobrazit kdokoli, jednoduše vyhledáním ID transakce nebo návštěvou odkazu [http://bit.ly/2Q860Wk].

Zkuste tento odkaz navštívit nebo zadat haš transakce na web *ropsten.etherscan.io* a podívat se na výsledek své práce.

Odesíláme ether z MetaMasku

Jakmile obdržíte první testovací ether z Ropsten testovacího kohoutku, můžete experimentovat s odesíláním etheru tím, že se pokusíte poslat nějaký zpět do kohoutku. Jak můžete vidět na stránce Ropsten testovací kohoutek, existuje možnost „darovat“ 1 ETH do kohoutku. Tato možnost je k dispozici, takže jakmile budete hotovi s testováním, můžete vrátit zbytek testovacího éteru, aby ho

mohl použít někdo jiný. I když testovací ether nemá žádnou hodnotu, někteří lidé to shromažďují, což dělá používání testovací sítě pro všechny ostatní obtížnějším. Hromadit testovací ether není hezké!

Naštěstí, my ethery hromadit nebudeme. Kliknutím na oranžové tlačítko „1 ether“ řekněte MetaMasku, aby vytvořil transakci, která vrátí 1 ether zpátky do kohoutku. MetaMask připraví transakci a objeví se okno s potvrzením, jak je uvedeno v Odeslání 1 etheru do kohoutku.

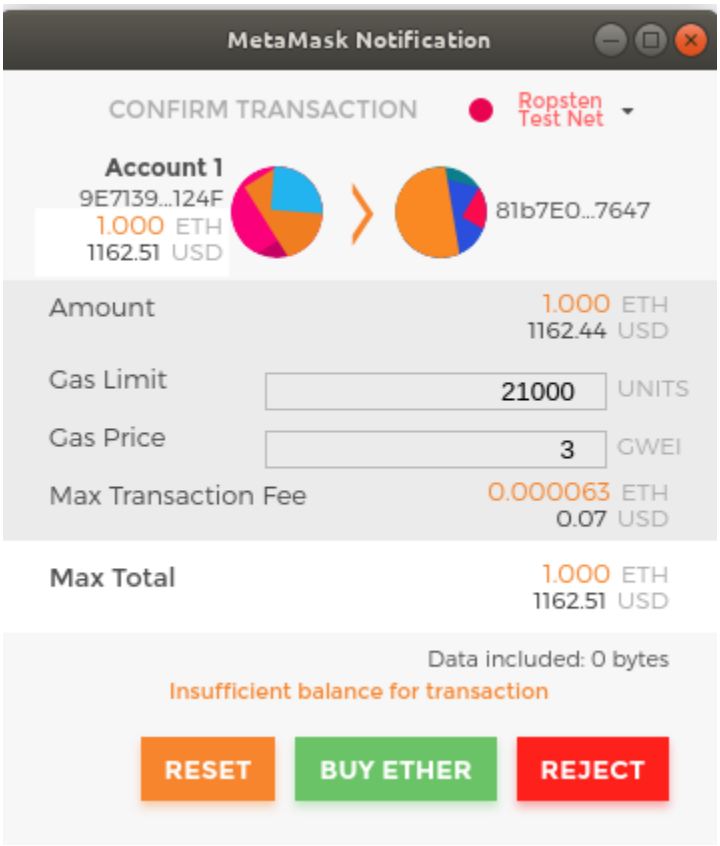

Figure 7. Odeslání 1 etheru do kohoutku

Jejda! Pravděpodobně jste si všimli, že transakci nemůžete dokončit - MetaMask říká, že nemáte dostatečný zůstatek. Na první pohled se to může zdát matoucí: máte 1 ETH, chcete poslat 1 ETH, tak proč MetaMask říká, že nemáte dostatek finančních prostředků?

Odpověď je kvůli nákladům na *plyn*. Každá Ethereum transakce vyžaduje zaplacení poplatku, který těžaři vybírají za ověření transakce. Poplatky v Ethereum jsou účtovány ve virtuální měně nazývané plyn. V rámci transakce platíte za plyn etherem.

Poplatky se vyžadují také ve zkušebních sítích. Bez poplatků by se zkušební síť chovala odlišně od hlavní sítě, což by z ní udělalo nevhodnou testovací platformu. Poplatky také chrání testovací sítě před útoky DoS a špatně vytvořenými chytrými kontrakty (např. nekonečné cykly), stejně jako chrání hlavní síť.

Když jste odeslali transakci, MetaMask vypočítal průměrnou cenu plynu za posledních úspěšných transakcí na 3 gwei, což znamená gigawei. Wei je nejmenší dílčí jednotka měny ether, jak jsme diskutovali v Měnové jednotky Etherea. Limit plynu je stanoven na náklady za odeslání základní transakce, což je 21 000 jednotek plynu. Proto maximální částka, kterou utratíte, je 3 * 21 000 gwei = 63 000 gwei = 0,000063 ETH. (Vezměte na vědomí, že průměrné ceny plynu mohou kolísat, protože jsou převážně určovány převážně těžaři. V další kapitole uvidíme, jak můžete zvýšit / snížit svůj limit plynu, abyste zajistili, že v případě potřeby bude mít transakce přednost.)

Shrneme si to: provedení transakce s 1 ETH nás príjde na 1.000063 ETH. MetaMask zobrazuje částky zaokrouhlené na několik platných desetinných míst. Uvidíme tedy zaokrouhlenou částku 1 ETH, ale skutečná částka, kterou potřebujete, je 1.000063 ETH a máte pouze 1 ETH. Klepnutím na Odmítnout zrušíte tuto transakci.

Pojďme získat další testovací ether! Znovu klikněte na zelené tlačítko „request 1 ether from the faucet“ a počkejte několik sekund. Nebojte se, kohoutek by měl mít dostatek éteru a dá vám více, pokud o to požádáte.

Jakmile budete mít zůstatek 2 ETH, můžete to zkusit znovu. Tentokrát, když kliknete na oranžové tlačítko darování „1 ether“, máte dostatečný zůstatek k dokončení transakce. Jakmile MetaMask zobrazí okno platby, klikněte na Odeslat. Po tom všem byste měli vidět zůstatek 0,999937 ETH, protože jste poslali 1 ETH do kohoutku s poplatkem 0,000063 ETH v plynu.

Zkoumání transakční historie adresy

Nyní jste se stali odborníkem na používání MetaMasku pro odesílání a přijímání testovacího etheru. Vaše peněženka obdržela alespoň dvě platby a poslala alespoň jednu. Všechny tyto transakce můžete zobrazit pomocí průzkumníka bloků *ropsten.etherscan.io*. Můžete buď zkopírovat svou adresu peněženky a vložit ji do vyhledávacího pole průzkumníka bloků, nebo nechat MetaMask stránku otevřít pro vás. Vedle ikony účtu v MetaMasku se zobrazí tlačítko se třemi tečkami. Kliknutím na ni zobrazíte nabídku možností souvisejících s účtem (viz <<metamask_account_context_menu> >).

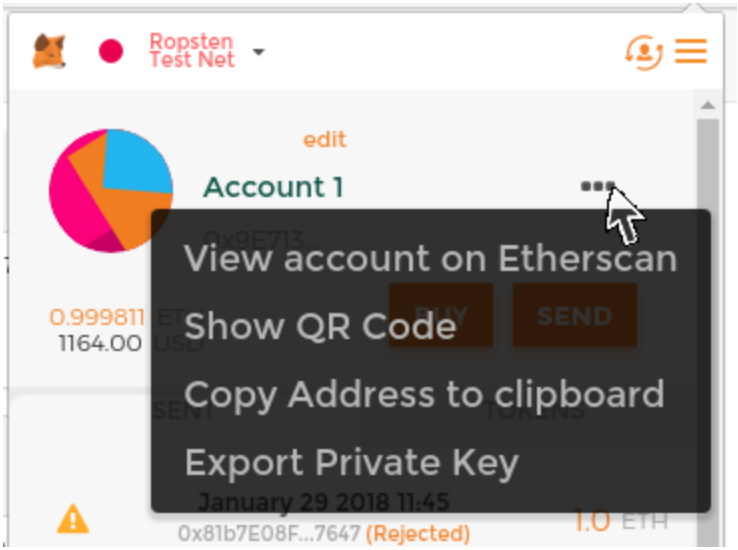

Figure 8. Kontextová nabídka účtu v MetaMasku

Výběrem možnosti „View account on Etherscan“ otevřete webovou stránku s prohlížečem bloků, který zobrazuje historii transakcí vašeho účtu, jak je uvedeno vTransakční historie adresy na Etherscan.

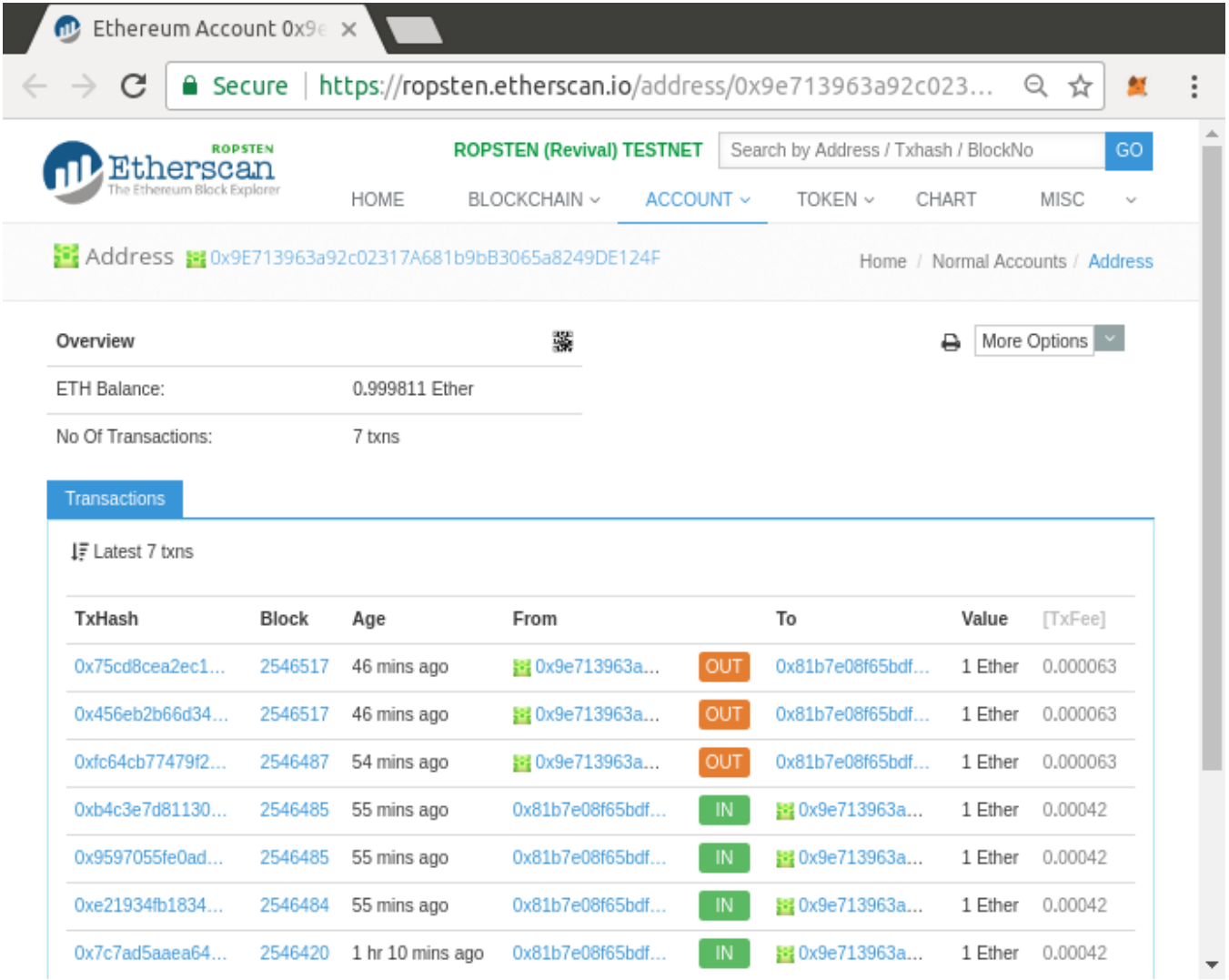

Figure 9. Transakční historie adresy na Etherscan

Zde můžete vidět celou transakční historii vaší Ethereum adresy. Zobrazují se všechny transakce zaznamenané na Ropsten bločence, u kterých je vaše adresa odesílatelem nebo příjemcem. Kliknutím na několik z těchto transakcí zobrazíte další podrobnosti.

Můžete prozkoumat historii transakcí jakékoli adresy. Podívejte se na transakční historii adresy Ropsten testovacího kohoutku (tip: jedná se o adresu odesílatele uvedenou v nejstarší platbě na vaši adresu). Můžete vidět veškerý testovací ether odeslaný z kohoutku na vaší adresu, ale i na jiné adresy. Každá transakce, kterou vidíte, může vést k více adresám a více transakcím. Brzy budete ztraceni v bludišti propojených dat. Veřejné bločenky obsahují obrovské množství informací, z nichž

všechny lze programově prozkoumat, jak uvidíme v budoucích příkladech .

Představujeme světový počítač

Nyní jste vytvořili peněženku a poslali a přijali ether. Zatím jsme s Ethereem zacházeli jako s kryptoměnou. Ale Ethereum je mnohem, mnohem víc. Ve skutečnosti je kryptoměnná funkce pouze jedna z mnoha funkcí Etherea jako decentralizovaného světového počítače.Ether je určen k platbě za provozování *chytrých kontraktů*, což jsou počítačové programy, které běží na emulovaném počítači zvaném *Ethereum Virtual Machine* (EVM).

EVM je globálně jednoinstanční (singleton), což znamená, že funguje, jako by šlo o globální počítač existující pouze v jedné instanci, který běží všude. Každý uzel v síti Ethereum provozuje lokální kopii EVM, aby ověřil provedení kontraktu, zatímco Ethereum bločenka zaznamenává měnící se *stav* tohoto světového počítače, když zpracovává transakce a chytré kontrakty. Budeme o tom diskutovat mnohem podrobněji v Ethereum virtuální stroj .

Externě vlastněné účty (EOA) a kontrakty

Typ účtu, který jste vytvořili v peněžence MetaMask, se nazývá externě vlastněny účet (EOA). Externě vlastněné účty jsou účty, které mají soukromý klíč; mít soukromý klíč znamená kontrolu přístupu k prostředkům nebo kontraktům. Nyní pravděpodobně hádáte, že existuje jiný typ účtu. Dalším typem účtu je *účet kontraktu*. Účet kontraktu má kód chytrého kontraktu, který jednoduchý EOA nemůže mít. Účet kontraktu navíc nemá soukromý klíč. Místo toho je vlastněn (a kontrolován) podle logiky svého kódu chytrého kontraktu: softwarového programu zaznamenaného do Ethereum bločenky při vytvoření účtu kontraktu a prováděného EVM.

Kontrakty mají adresy, stejně jako EOA. Kontrakty mohou také odesílat a přijímat ether, stejně jako EOA. Pokud je však cílem transakce adresa kontraktu, způsobí to, že tento kontrakt *bude spuštěn v EVM. Kontrakt jako svá vstupní data využije transakci a její vstupní data. Kromě etheru mohou transakce obsahovat _data* označující, která konkrétní funkce v kontraktu má být spuštěna a jaké parametry se mají předat této funkci. Tímto způsobem mohou transakce *volat* funkce v rámci kontraktů.

Protože účet kontraktu nemá soukromý klíč, nemůže transakci *iniciovat*, zahájit. Transakce mohou iniciovat pouze EOA, ale kontrakty mohou *reagovat* na transakce voláním jiných kontraktů a vytvářením složitých cest provádění. Typickým použitím je, že EOA odešle transakce s požadavkem

do vícepodpisové peněženky chytrého kontraktu, aby ten odeslal ETH na jinou adresu. Typickým vzorem programování DApp je mít kontrakt A, která volá kontrakt B, aby se zachoval sdílený stav mezi uživateli kontraktu A.

V následujících několika sekcích naprogramujeme náš první kontrakt. Poté se naučíte, jak vytvořit, financovat a používat tento kontrakt s vaší peněženkou MetaMask a testovacím etherem v testovací síti Ropsten.

Jednoduchá smlouva: Testovací Ethereum kohoutek

Ethereum má mnoho různých vysokoúrovňových jazyků, z nichž všechny lze použít k napsání kontraktu a vytvoření bajtkódu EVM. Několik nejvýznamnějších a nejzajímavějších jsme představili v Úvod do vysokoúrovňových Ethereum jazyků;. Jeden vysokoúrovňový jazyk se pro programování chytrých kontraktů zdaleka nejčastěji: Solidity. Solidity byl vytvořen Dr. Gavinem Woodem, spoluautorem této knihy, a stal se nejrozšířenějším jazykem Etherea (a dálších). Použijeme Solidity k napsání našeho prvního kontraktu.

Jako náš první příklad (Faucet.sol: Chytrý kontrakt Kohoutek v Solidity), napíšeme kontrakt, který řídí *kohoutek*. Už jste použili kohoutek k získání testovacího etheru v testovací síti Ropsten. Kohoutek je relativně jednoduchá věc: rozdává ether na jakoukoli adresu, která zažádá, a může být pravidelně doplňováň. Kohoutek můžete implementovat jako peněženku ovládanou člověkem nebo webovým serverem.

Example 1. Faucet.sol: Chytrý kontrakt Kohoutek v Solidity

```
// Version of Solidity compiler this program was written for
pragma solidity ^0.6.0;

// Our first contract is a faucet!
contract Faucet {
    // Accept any incoming amount
    receive () external payable {}

    // Give out ether to anyone who asks
    function withdraw(uint withdraw_amount) public {

        // Limit withdrawal amount
        require(withdraw_amount <= 100000000000000000);

        // Send the amount to the address that requested it
        msg.sender.transfer(withdraw_amount);
    }
}
```

Všechny ukázky kódu uvedené v této knize najdete v podadresáři *code* GitHub úložiště této knihy [https://github.com/ethereumbook/ethereumbook/]. Konkrétně náš kontrakt *Faucet.sol* je v:

```
code/Solidity/Faucet.sol
```

Toto je velmi jednoduchý kontrakt, tak jednoduchý, jak si jen lze představit. Je to také *vadný* kontrakt, který demonstruje řadu špatných praktik a slabých míst zabezpečení. Všechny jeho nedostatky prozkoumáme v pozdějších sekcích. Ale prozatím se podívejme, co tento kontrakt dělá a jak to funguje, řádek po řádku. Rychle si všimnete, že mnoho prvků Solidity je podobných existujícím programovacím jazykům, jako je JavaScript, Java nebo C++.

První řádek je komentář:

```
// Our first contract is a faucet!
```

Komentáře jsou určeny pro lidi a nejsou zahrnuty do spustitelného bajtkódu EVM. Obvykle je umístíme na řádek před kód, který se snažíme vysvětlit, nebo někdy na stejný řádek. Komentáře začínají dvěma lomítky: //. Všechno od prvního lomítka až do konce tohoto řádku je považováno za prázdnou řádku a je ignorováno.

Další řádek je místo, kde začíná naš skutečný kontrakt:

```
contract Faucet {
```

Tento řádek deklaruje kontrakt (contract), podobný deklaraci třídy (class) v jiných objektově orientovaných jazycích. Definice kontraktu zahrnuje všechny řádky mezi složenými složenými závorkami ({}), které definují jeho *rozsah*, podobně jako jsou složené složené závorky použity v mnoha jiných programovacích jazycích.

Dále deklarujeme první funkci kontraktu Faucet:

```
function withdraw(uint withdraw_amount) public {
```

Funkce se nazývá withdraw a vyžaduje jeden parametr typu celé číslo bez znaménka (uint) s názvem withdraw_amount. Je deklarována jako veřejná funkce, což znamená, že ji lze volat jinými kontrakty. Následuje definice funkce, ohraničená složenými závorkami. První část funkce withdraw stanoví maximální limit pro výběr:

```
require(withdraw_amount <= 100000000000000000);
```

Používá vestavěnou funkci Solidity require k testování předpokladů, že withdraw_amount je menší nebo rovno 100 000 000 000 000 000 wei, což je základní jednotka etheru (viz Dílčí měnové jednotky Etherea a jejich názvy) a odpovídá 0,1 etheru. Pokud je funkce withdraw volána s hodnotou parametru withdraw_amount větším než tato částka, funkce require způsobí zastavení provádění kontraktu a selže vyvoláním *výjimky*. Všimněte si, že příkazy v Solidity musí být zakončeny středníkem.

Tato část kontraktu je hlavní logikou našeho kohoutku. Řídí tok finančních prostředků z kontraktu maximálního limitu pro výběr. Je to velmi jednoduché ovládání, ale může vám demonstrovat sílu programovatelné bločenky: decentralizovaný software kontrolující peníze.

Dále následuje samotný výběr:

```
msg.sender.transfer(withdraw_amount);
```

Děje se zde několik zajímavých věcí. Objekt msg je jedním ze vstupů, ke kterým mají přístup všechny kontrakty. Představuje transakci, která vedla k provedení tohoto kontraktu. Objekt sender je adresa odesílatele transakce. Funkce transfer je vestavěná funkce, která přenáší ether z aktuálního kontraktu na adresu odesílatele. Při zpětném čtení to znamená převod (transfer) + k odesilateli (sender) zprávy (msg), který vyvolal spuštění tohoto kontraktu. Funkce transfer požaduje jediný parametr a to odesílanou částku. Předáme ji hodnotu parametru withdraw_amount, se kterou byla volána funkce withdraw o několik řádků výše.

Hnedka další řádek je uzavírací složená závorka, která označuje konec definice naší funkce withdraw.

Dále deklarujeme ještě jednu funkci:

```
function () external payable {}
```

Tato funkce se nazývá *výchozí*, *implicitní* nebo *fallback*. Funkce je volána, pokud transakce, která spustila kontrakt, neuvedla žádnou z deklarovaných funkcí v kontraktu nebo vůbec žádnou funkci nebo neobsahovala data. Kontrakty mohou mít jednu takovou výchozí funkci (bez názvu) a je to obvykle ta, která přijímá ether. Proto je definována jako externí a platby přijímající (payable) funkce, což znamená, že může do smlouvy přijmout ether. Nedělá nic jiného než, že přijímá ether, jak ukazuje prázdná definice v složených závorkách pass: [(<code>{}</code>)]. Pokud provedeme transakci, která pošle ether na adresu kontraktu, jako by to byla peněženka, tato funkce to zvládne.

Přímo pod naší výchozí funkcí je závěrečná uzavírací složená závorka, která uzavírá definici kontraktu Faucet. To je ono!

Kompilace kontraktu Kohoutek

Nyní, když máme náš první příklad kontrraktu, musíme použít překladač Solidity k převodu kódu Solidity na bajtkód EVM, aby jej mohl provést EVM na samotné bločence.

Kompilátor Solidity existuje jako samostatný spustitelný soubor, ale i jako součást různých rámců a integrovaných vývojových prostředí (IDE). Aby to bylo co nejjednodušší, použijeme jeden z populárnějších IDE, nazvaný *Remix*.

Pomocí prohlížeče Chrome (s peněženkou MetaMask, kterou jste nainstalovali dříve) přejděte na Remix IDE na adrese https://remix.ethereum.org [].

Když poprvé načtete Remix, zobrazí vzorový kontrakt nazvaný *ballot.sol*. Ten nepotřebujeme, takže jej zavřete kliknutím na x v rohu karty, jak je vidět v Zavření záložky výchozího příkladu.

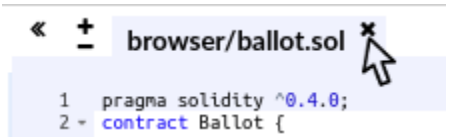

Figure 10. Zavření záložky výchozího příkladu

Nyní přidejte novou kartu kliknutím na kruhové znaménko plus na levém horním panelu nástrojů, jak je vidět v Kliknutí na znaménko plus pro otevření nového okna. Pojmenujte nový soubor *Faucet.sol*.

Figure 11. Kliknutí na znaménko plus pro otevření nového okna

Až budete mít novou kartu otevřenou, zkopírujte a vložte kód z našeho příkladu *Faucet.sol*, jak je vidět v Okopírování kódu Kohoutku do nového okna.

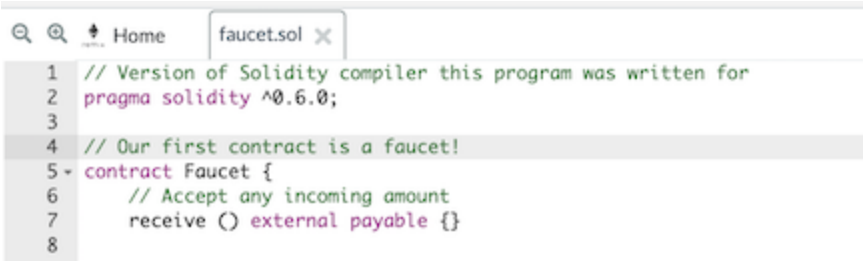

Figure 12. Okopírování kódu Kohoutku do nového okna

Po načtení kontraktu *Faucet.sol* do Remix IDE, IDE automaticky zkompiluje kód. Pokud vše půjde dobře, zobrazí se zelené pole s nápisem „Faucet“ vpravo na kartě Compile, který potvrdí úspěšné dokončení kompilace (viz Remix úspěšně zkompiluje kontrakt Faucet.sol).

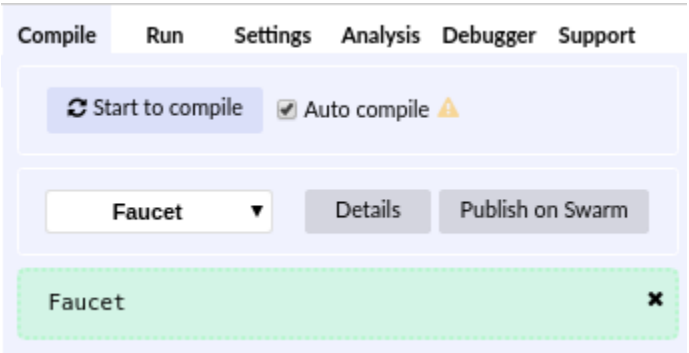

Figure 13. Remix úspěšně zkompiluje kontrakt Faucet.sol

Pokud se něco pokazí, nejpravděpodobnějším problémem je, že Remix IDE používá verzi kompilátoru Solidity, která se liší od 0.5.12. V takovém případě naše direktiva pragma zabrání kompilaci *Faucet.sol*. Chcete-li změnit verzi kompilátoru, přejděte na kartu Nastavení, nastavte verzi na 0,5.12 a akci opakujte.

Kompilátor Solidity nyní zkompiloval náš *Faucet.sol* do bajtkódu EVM. Pokud jste zvědaví, bajtkód vypadá takto:

```
PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE ISZERO PUSH2 0xF JUMPI PUSH1 0x0
DUP1
REVERT JUMPDEST PUSH1 0xE5 DUP1 PUSH2 0x1D PUSH1 0x0 CODECOPY PUSH1 0x0
RETURN
STOP PUSH1 0x60 PUSH1 0x40 MSTORE PUSH1 0x4 CALLDATASIZE LT PUSH1 0x3F
JUMPI
PUSH1 0x0 CALLDATALOAD PUSH29
0x100000000000000000000000000000000000000000000000000000000
SWAP1 DIV PUSH4 0xFFFFFFFF AND DUP1 PUSH4 0x2E1A7D4D EQ PUSH1 0x41 JUMPI
JUMPDEST STOP JUMPDEST CALLVALUE ISZERO PUSH1 0x4B JUMPI PUSH1 0x0 DUP1
REVERT
JUMPDEST PUSH1 0x5F PUSH1 0x4 DUP1 DUP1 CALLDATALOAD SWAP1 PUSH1 0x20 ADD
SWAP1
SWAP2 SWAP1 POP POP PUSH1 0x61 JUMP JUMPDEST STOP JUMPDEST PUSH8
0x16345785D8A0000 DUP2 GT ISZERO ISZERO ISZERO PUSH1 0x77 JUMPI PUSH1 0x0
DUP1
REVERT JUMPDEST CALLER PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
AND
PUSH2 0x8FC DUP3 SWAP1 DUP2 ISZERO MUL SWAP1 PUSH1 0x40 MLOAD PUSH1 0x0
PUSH1
0x40 MLOAD DUP1 DUP4 SUB DUP2 DUP6 DUP9 DUP9 CALL SWAP4 POP POP POP POP
ISZERO
ISZERO PUSH1 0xB6 JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST POP JUMP STOP LOG1
PUSH6
0x627A7A723058 KECCAK256 PUSH9 0x13D1EA839A4438EF75 GASLIMIT CALLVALUE
LOG4 0x5f
PUSH24 0x7541F409787592C988A079407FB28B4AD000290000000000
```

Nejste rádi, že používáte vysokoúrovňový programovací jazyk, jako je Solidity, namísto programování přímo v bajtkódu EVM? Já jsem rád také!

Vytvoření kontraktu na bločence

Máme tedy kontrakt. Zkompilovali jsme ji do bajtkódu. Nyní musíme „zaregistrovat“ kontrakt na Ethereum bločence. K otestování našeho kontraktu použijeme testovací síť Ropsten, na tuto bločenku chceme kontrakt zapsat.

Registrace kontraktuna bločenku zahrnuje vytvoření speciální transakce, jejímž cílem je adresa 0x0000000000000000000000000000000000000000, také známá jako *nulová adresa*. Nulová

adresa je speciální adresa, která říká Ethereum bločence, že chcete zaregistrovat kontrakt. Naštěstí to Remix IDE zvládne za vás a odešle transakci pomocí MetaMask.

Nejprve přejděte na kartu Run a v rozevíracím seznamu Environment vyberte Injected Web3. Tím se propojí Remix IDE s peněženkou MetaMask a přes MetaMask s testovací sítí Ropsten. Jakmile to uděláte, můžete vidět Ropsten v Environment. Ve výběrovém poli Account se také zobrazuje adresa vaší peněženky (viz Remix IDE záložka Run, s vybraným prostředím Injected Web3).

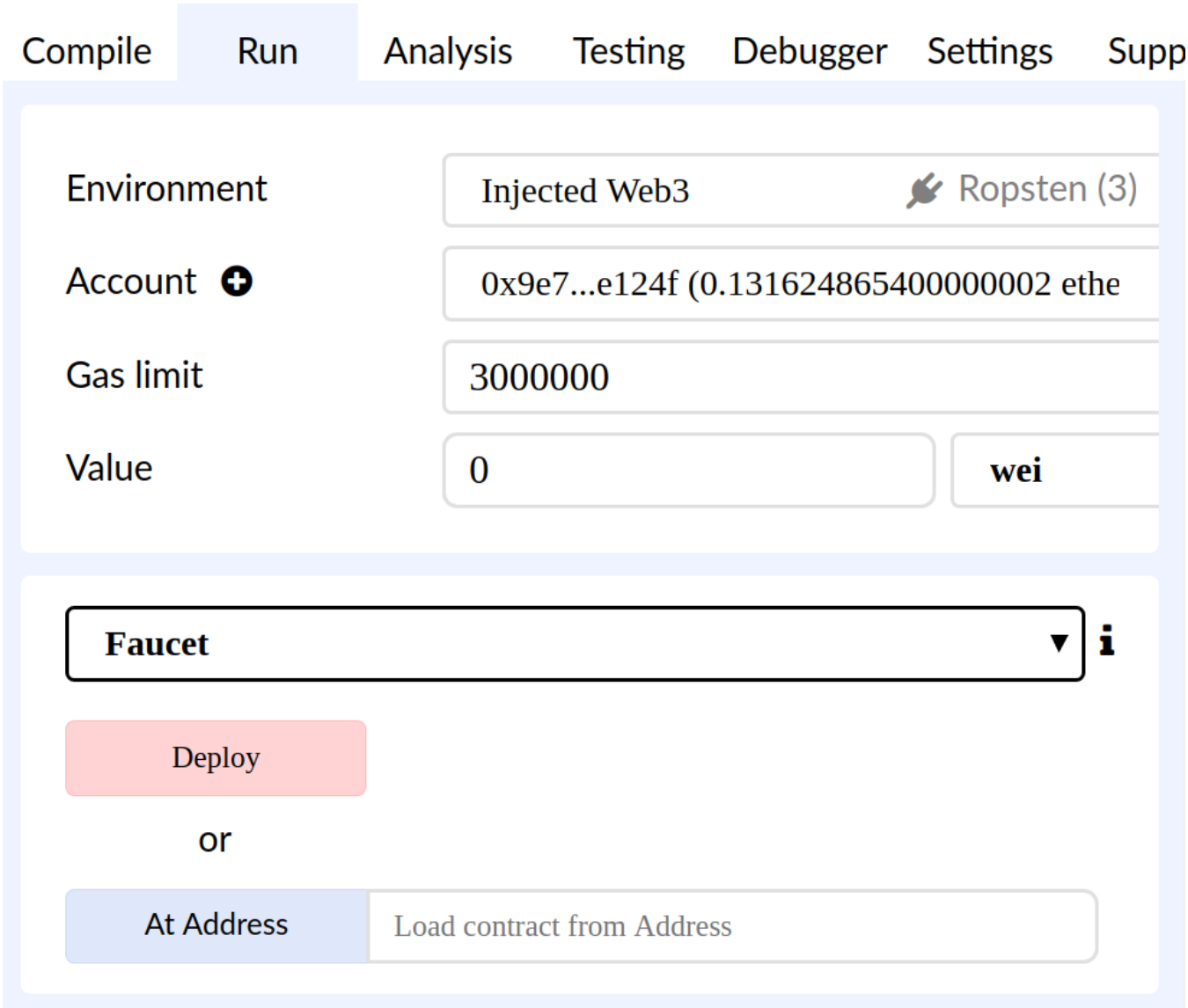

Figure 14. Remix IDE záložka Run, s vybraným prostředím Injected Web3

Přímo pod nastavením Run, které jste právě potvrdili, je kontrakt Faucet, připravený k vytvoření. Klikněte na tlačítko Deploy zobrazené v Remix IDE záložka Run, s vybraným prostředím Injected Web3.

Remix vytvoří speciální „vytvářející“ transakci a MetaMask vás požádá o schválení, jak ukazuje MetaMask ukazující transakci vytvářející kontrakt. Všimnete si, že transakce vytvoření kontraktu neobsahuje žádný ether, ale obsahuje 262 bajtů dat (zkompilovaný kontrakt) a spotřebuje 10 gwei v plynu. Kliknutím na Submit ji schválíte.

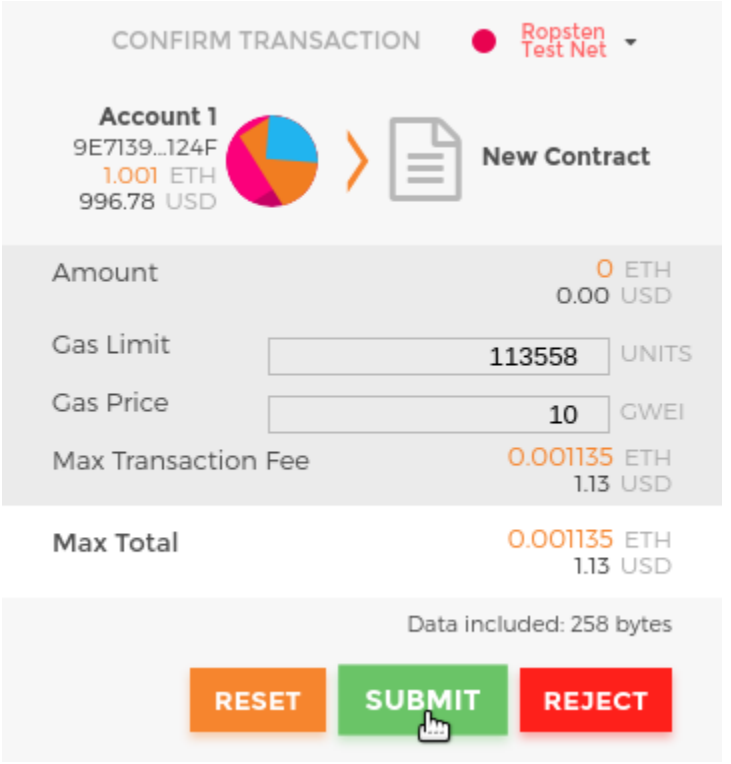

Figure 15. MetaMask ukazující transakci vytvářející kontrakt

Nyní musíte počkat. Bude trvat asi 15 až 30 sekund než transakce vytvářející kontrakt bude vytěžena na Ropstenu. Zdá se, že Remix nic nedělá, ale buďte trpěliví.

Jakmile je kontrakt vytvořen, objeví se ve spodní části záložky Run (viz Kontrakt Faucet ŽIJE!).

Figure 16. Kontrakt Faucet ŽIJE!

Všimněte si, že kontrakt Faucet má nyní svou vlastní adresu: Remix ji zobrazuje jako “Faucet at 0x72e…c7829” (ačkoli vaše adresa, náhodná písmena a čísla, se budou lišit). Malý symbol schránky vpravo umožňuje kopírovat adresu kontrakty do vaší schránky. Použijeme to v následující sekci.

Interakce s kontraktem

Zopakujme, co jsme se dosud dozvěděli: Ethereum kontrakty jsou programy, které ovládat peníze, které běží uvnitř virtuálního stroje zvaného EVM. Jsou vytvářeny speciální transakcí, která odešle jejich bajtkód, který se má zaznamenat do bločenky. Jakmile jsou vytvořeny na bločence, mají Ethereum adresu, stejně jako peněženky. Kdykoli někdo odešle transakci na adresu kontraktu, způsobí to, že kontrakt běží v EVM, s transakcí jako jejím vstupem. Transakce odesílané na adresu kontraktu mohou obsahovat ether nebo data nebo obojí. Pokud obsahují ether, je „uložen“ do zůstatku kontraktu. Pokud obsahují data, mohou data specifikovat pojmenovanou funkci v kontraktu a volat ji, předat funkci parametry.

Zobrazení adresy kontraktu v prohlížeči bloků

Nyní máme smlouvu zaznamenanou v bločence a vidíme, že má Ethereum adresu. Podívejme se na to v průzkumníku bloků *ropsten.etherscan.io* a uvidíme, jak vypadá kontrakt. V Remix IDE zkopírujte adresu kontraktu kliknutím na ikonu schránky vedle jejího názvu (viz Okopírování adresy kontraktu z Remixu).

Figure 17. Okopírování adresy kontraktu z Remixu

Udržujte Remix otevřený; k tomu se vrátíme později. Nyní přejděte v prohlížeči na adresu *ropsten.etherscan.io* a vložte adresu do vyhledávacího pole. Měli byste vidět historii Ethereum adresy kontraktu, jak je uvedeno v Zobrazení adresy kontraktu Faucet v průzkumníku bloku Etherscan.

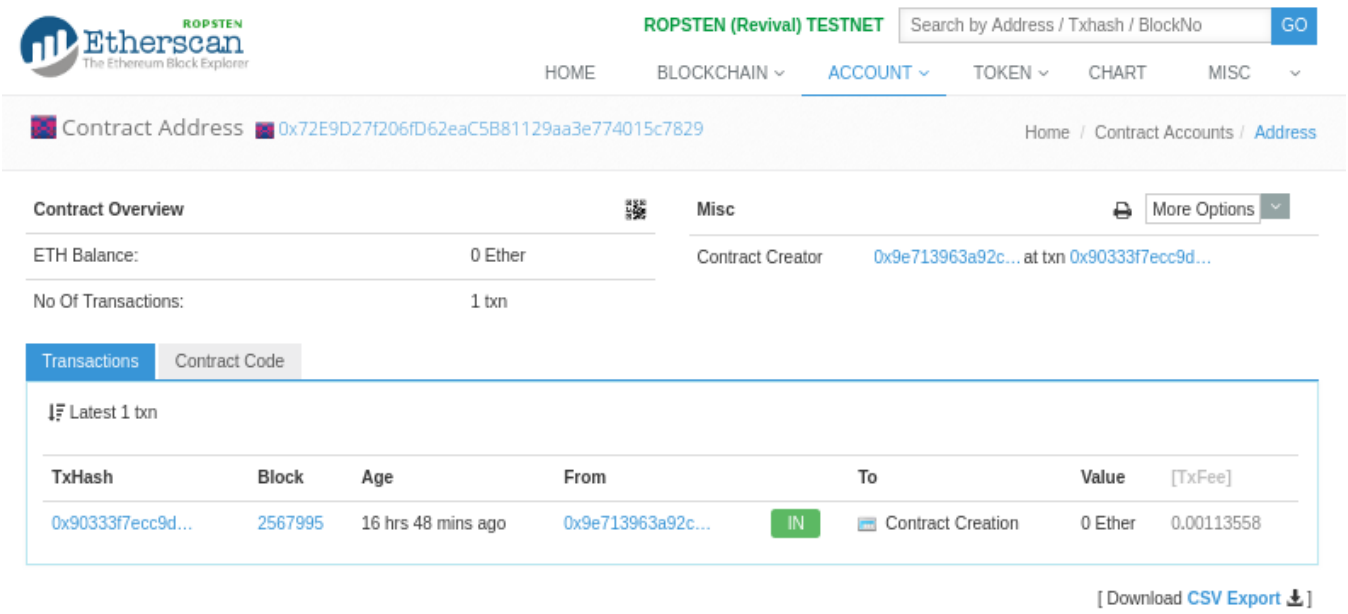

Figure 18. Zobrazení adresy kontraktu Faucet v průzkumníku bloku Etherscan

Zaslání financí kontraktu

Prozatím má kontrakt ve své historii pouze jednu transakci: transakce vytvoření kontraktu. Jak vidíte, kontrakt také neobsahuje ether (nulový zůstatek). Je to proto, že jsme kontrakt vytvářející transakci neposlali žádný ether, i když jsme mohli.

Náš kohoutek potřebuje finanční prostředky! Naším prvním úkolem bude použití MetaMasku k odeslání etheru ve prospěch kontraktu. Adresu kontraktu byste měli mít i nadále ve své schránce (pokud ne, zkopírujte ji znovu z Remixu). Otevřete MetaMask a pošlete kontraktu 1 ether, přesně jako na jakoukoli jinou adresu Ethereum (Poslání 1 etheru na adresu kontraktu).

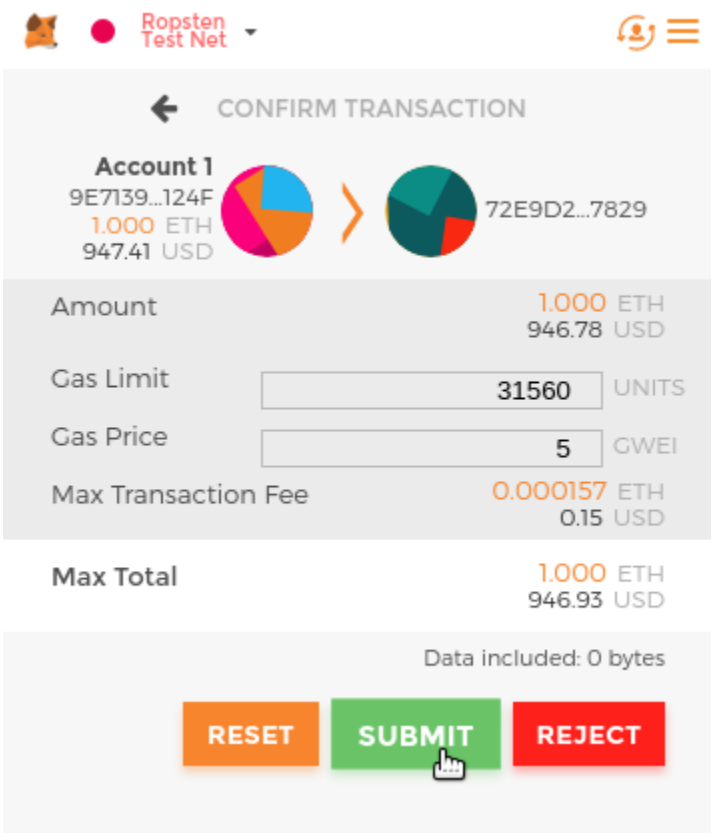

Figure 19. Poslání 1 etheru na adresu kontraktu

Pokud za minutu znovu načtete průzkumníka bloků Etherscan, ukáže další transakci na adrese kontraktu a aktualizovaný zůstatek 1 ether.

Pamatujete si nepojmenovanou, výchozí, externí, platby přijímající funkci v našem *Faucet.sol* kódu?

Vypadala takto:

```
function () external payable {}
```

Když jste odeslali transakci na adresu kontraktu, ale nespecifikovali jste žádná data, se kterými se má funkce volat, byla zavolána tato výchozí funkce. Protože jsme ji označili jako platby přijímající (payable), přijala a vložila 1 ether do zůstatku na účtu kontraktu. Vaše transakce způsobila spuštění kontraktu v EVM a aktualizaci jeho zůstatku. Financovali jste kohoutek!

Výběr financí z našeho kontraktu

Dále si vybereme některé finanční prostředky z kohoutku. Chcete-li vybírat, musíme sestavit transakci, která volá funkci+withdraw+ a předá jí parametr withdraw_amount. Aby to prozatím zůstalo jednoduché, Remix pro nás tuto transakci zkonstruuje a MetaMask ji předloží ke schválení.

Vraťte se na kartu Remix a podívejte se na kontrakt na kartě Run. Měli byste vidět oranžové pole označené withdraw se záznamem pole označeným uint256 withdraw_amount (viz Funkce výběru financí z kontraktu Faucet.sol v Remixu).

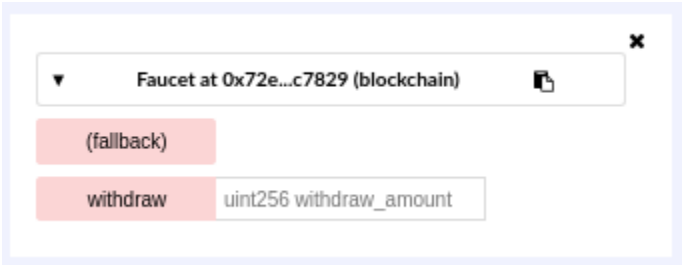

Figure 20. Funkce výběru financí z kontraktu Faucet.sol v Remixu

Toto je Remix rozhraní ke kontraktu. To nám umožňuje vytvářet transakce, které volají funkce definované v kontraktu. Zadáme withdraw_amount a kliknutím na tlačítko „withdraw“ vygenerujeme transakci.

Nejprve se podívejme na withdraw_amount. Chceme zkusit odebrat 0,1 etheru, což je maximální částka povolená naším kontraktem. Nezapomeňte, že všechny hodnoty měny v Ethereum jsou vnitřně uváděny ve wei a naše funkce withdraw očekává, že její parametr +withdraw_amount bude také zadán ve wei. Požadované množství je 0,1 etheru, což je 100 000 000 000 000 000 wei (1 následovaná 17 nulami).

Vzhledem k omezení v JavaScriptu nemůže Remix zpracovat číslo až 10^17. Místo toho ji uzavíráme do dvojitých uvozovek, abychom Remixu to umožnili přijmout jako řetězec a manipulovat s ním jako s velkým číslem (BigNumber). Pokud tento parametr neuzavřeme do uvozovek, Remix IDE jej nedokáže zpracovat a zobrazí se chybové hlášení „Error encoding arguments: Error: Assertion failed.“

Do pole withdraw_amount zadejte „100000000000000000“ (s uvozovkami) a klikněte na tlačítko„withdraw“ pro výběr (viz Vytvoření výběrové transakce kliknutím na tlačítko „withdraw“ v Remixu).

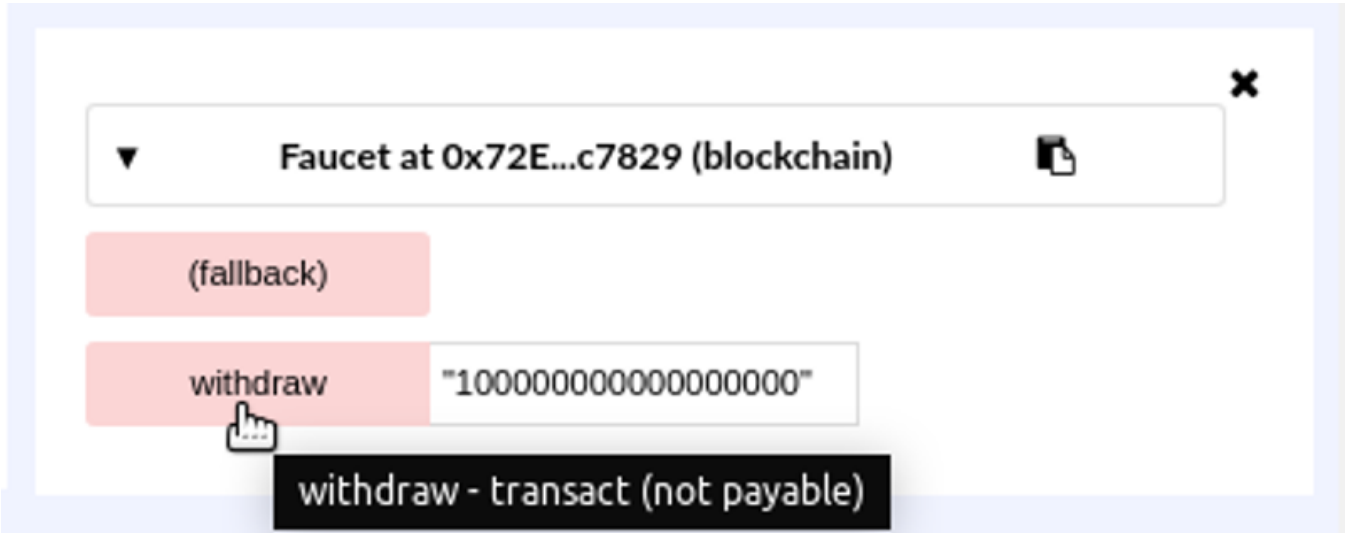

Figure 21. Vytvoření výběrové transakce kliknutím na tlačítko „withdraw“ v Remixu

MetaMask zobrazí transakční okno, ve kterém můžete souhlasit. Kliknutím na Confirm odešlete své volání funkce výběru našeho kontraktu (viz MetaMask transakce k vyvolání funkce výběru).

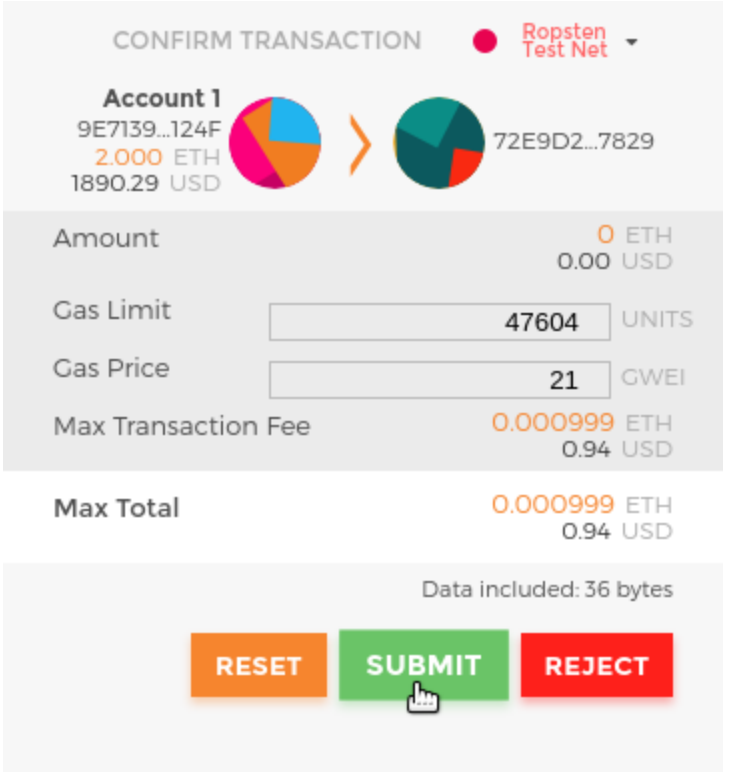

Figure 22. MetaMask transakce k vyvolání funkce výběru

Počkejte minutu a pak znovu načtěte průzkumníka bloků Etherscan, aby se transakce projevila v historii transakcí kontraktu Faucet (viz Ethercan zobrazuje transakci, která volá funkci výběru).

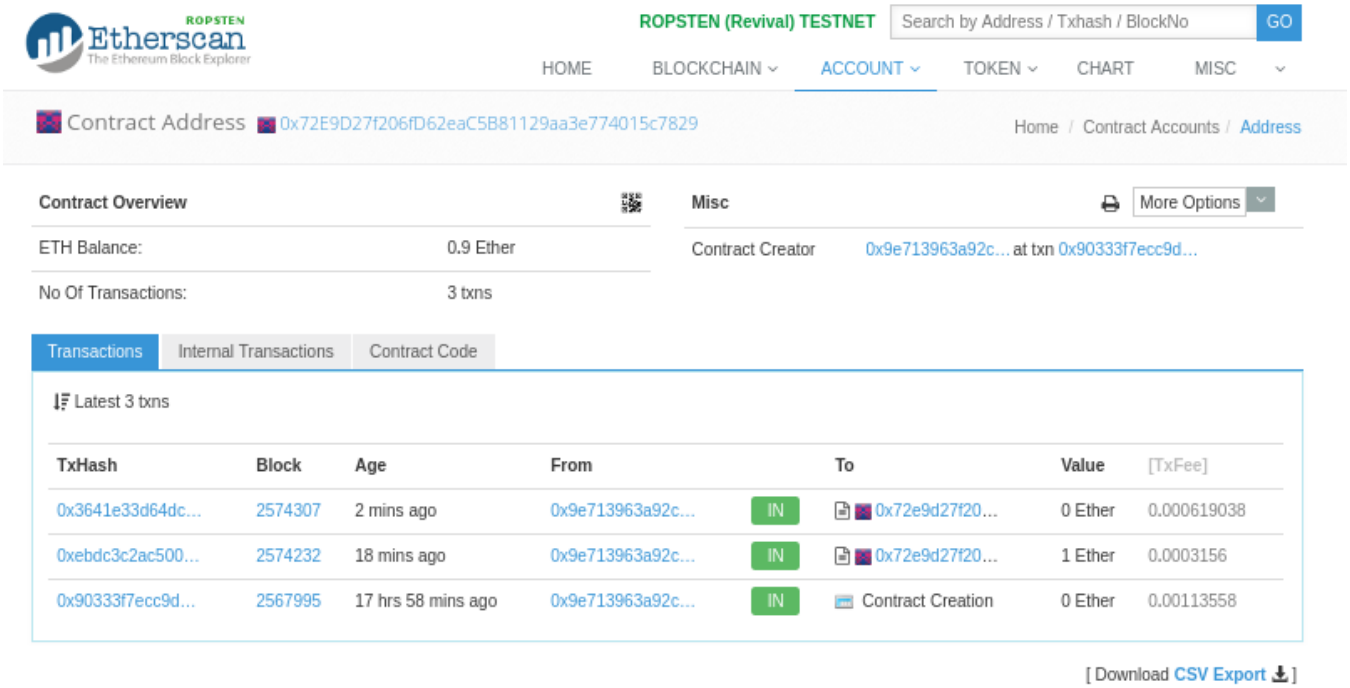

Figure 23. Ethercan zobrazuje transakci, která volá funkci výběru

Nyní vidíme novou transakci s adresou smlouvy jako cílem a hodnotou 0 ether. Zůstatek kontraktu se změnil a nyní je 0,9 etheru, protože nám poslal 0,1 etheru podle požadavku. Nevidíme však „odchozí“ transakci v *transakční historii kontraktu*.

Kde je odchozí výběr? Na stránce transakční historie adresy kontraktu se objevila nová karta s názvem vnitřní transakce (Internal Transactions). Protože přenos 0,1 etheru pochází z kódu kontraktu, jedná se o interní transakci (nazývanou také *zpráva*). Kliknutím na tuto kartu ji zobrazíte (viz Etherscan zobrazuje vnitřní transakci převádějící ether z kontraktu).

Tato „interní transakce“ byla zaslána kontraktem na tomto řádku kódu (z funkce pass: [<code>withdraw</code>] v *Faucet.sol*):

```
msg.sender.transfer(withdraw_amount);
```

Shrnutí: z vaší peněženky MetaMask jste odeslali transakci, která obsahovala data a instrukce k zavolání funkce withdraw s hodnotou parametru withdraw_amount 0,1 etheru. Tato transakce způsobila, že kontrakt byl spuštěn uvnitř EVM. Jakmile EVM spustil funkci withdraw kontraktu

withdraw, nejprve volal funkci require a ta potvrdila, že požadované množství bylo menší nebo rovno maximálnímu povolenému výběru 0,1 etheru. Pak zavolal funkci transfer, aby vám poslal ether. Spuštění funkce transfer vygenerovalo vnitřní transakci, která vložila 0,1 éteru do vaší peněženky ze zůstatku kontraktu. To je ten, který je zobrazen na kartě Internal Transactions v Etherscan. .

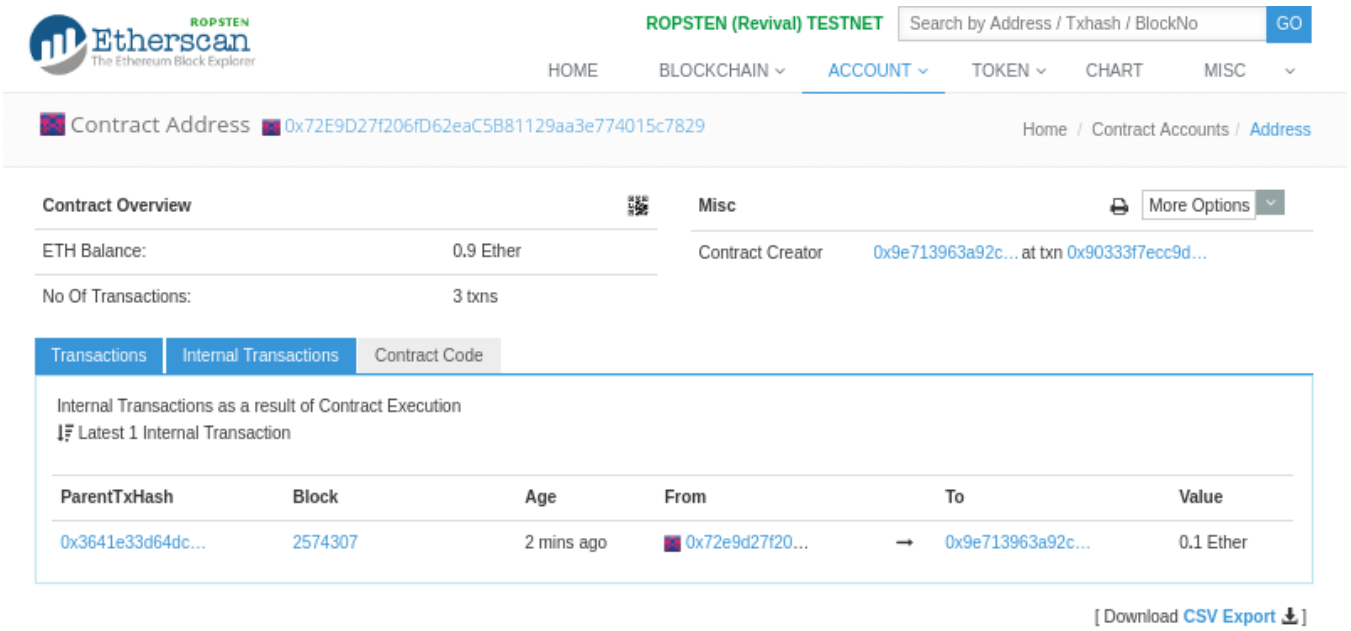

Figure 24. Etherscan zobrazuje vnitřní transakci převádějící ether z kontraktu

Závěry

V této kapitole jste založili peněženku pomocí MetaMasku a naplnili ji prostředky pomocí kohoutku v testovací síti Ropsten. Získali jste ether na Ethereum adresu vaší peněženky, pak jste poslali ether na Ethereum adresu kohoutku.

Dále jste v Solidity napsali kontrakt kohoutku. Použili jste Remix IDE k sestavení kontraktu do bajtkódu EVM, pak jste použili Remix k vytvoření transakce a vytvořili kontrakt Faucet na Ropsten bločence. Jakmile byl kontrakt Faucet vytvořen, získal Ethereum adresu a vy jste jí poslali nějaký ether. Nakonec jste vytvořili transakci pro volání funkce withdraw a úspěšně jste požádali o 0,1 etheru. Kontrakt zkontrolovala požadavek a zaslal vám 0,1 etheru pomocí vnitřní transakce.

Může to vypadat jako moc, ale právě jste úspěšně spolupracovali se softwarem, který řídí peníze v

decentralizovaném světovém počítači.

Naprogramujeme mnohem více chytrých kontraktů Chytré kontrakty a Solidity a seznámíme se s osvědčenými postupy a bezpečnostními hledisky v Bezpečnost chytrých kontraktů.

Ethereum klienti

Ethereum klient je softwarová aplikace, která implementuje specifikaci Etherea a komunikuje přes síť typu peer-to-peer s ostatními Ethereum klienty. Různí Ethereum klienti *spolupracují*, pokud splňují referenční specifikaci a standardizované komunikační protokoly. Zatímco tito různí klienti jsou implementováni různými týmy a v různých programovacích jazycích, všichni „mluví“ stejným protokolem a řídí se stejnými pravidly. Jako takové je lze všechny použít k provozu a interakci ve stejné sítí Ethereum.

Ethereum je projekt s otevřeným zdrojovým kódem a zdrojový kód pro všechny hlavní klienty je k dispozici na základě licencí pro otevřený zdrojový kód (např. LGPL v3.0), můžete ho zdarma stáhnout a použít pro jakýkoli účel. *Otevřený zdrojový kód* však znamená více než jen bezplatné použití. Znamená to také, že Ethereum je vyvíjeno otevřenou komunitou dobrovolníků a může jej měnit kdokoli. Více očí znamená důvěryhodnější kód.

Ethereum je definováno formální specifikací nazvanou „Žlutá kniha“ (viz Další čtení).

To je v kontrastu například s Bitcoinem, který není formálně definován. Kde Bitcoinová „specifikace“ je referenční implementací Bitcoin Core, specifikace Ethereum je dokumentována v článku, který kombinuje anglickou a matematickou (formální) specifikaci. Tato formální specifikace, spolu s různých návrhy na vylepšení Etherea, definuje standardní chování Etherea klient. Žlutá kniha je pravidelně aktualizována, protože jsou prováděny významné změny Ethereum.

V důsledku jasné formální specifikace Etherea existuje řada nezávisle vyvinutých, přesto schopných vzájemné spolupráce, softwarových implementací Ethereum klienta. Ethereum má větší rozmanitost implementací běžících v síti než jakékoli jiná bločenka, která je obecně považována za dobrou věc. Ve skutečnosti se tento přístup například osvědčil jako vynikající způsob obrany proti útokům v síti, protože nalezení chyby v implementaci jednoho konkrétního klienta umožňuje jeho vývojářům ji v klidu opravit, zatímco ostatní klienti udržují téměř bezproblémový běh sítě.

Sítě Etherea

Existuje celá řada sítí založených na Ethereu, které do značné míry odpovídají formálním specifikacím definovaným v Žluté knize Etherea, ale které mohou nebo nemusí vzájemně spolupracovat.

Mezi tyto sítě založené na Ethereu patří Ethereum, Ethereum Classic, Ella, Expanse, Ubiq, Musicoin a mnoho dalších. I když jsou tyto sítě většinou kompatibilní na úrovni protokolu, mají často funkce nebo atributy, které vyžadují, aby správci Ethereum klientského softwaru provedli malé změny, aby podporovali každou síť. Z tohoto důvodu ne každá verze Ethereum klientského softwaru umožňuje spouštět každou bločenku založenou na Ethereu.

V současné době existuje šest hlavních implementací protokolu Ethereum, psaných v šesti různých jazycích:

- Parity, napsaná v Rustu
- Geth, napsaná v Go
- cpp-ethereum, napsaná v C++
- pyethereum, napsaná v Pythonu
- Mantis, napsaná v Scale
- Harmony, napsaná v Javě

V této části se podíváme na dva nejběžnější klienty, Parity a Geth. Ukážeme, jak nastavit uzel pomocí každého klienta, a prozkoumat některé z možností jejich příkazového řádku a rozhraní pro programování aplikací (API).

Mám provozovat úplný uzel?

Zdraví, odolnost bločenky a její a schopnost vzdorovat cenzuře závisí na tom, že má mnoho nezávisle provozovaných a geograficky rozptýlených úplných uzlů. Každý úplný uzel může pomoci dalším novým uzlům získat data bloku k nastavení jejich provozu a také nabídnout jeho provozovateli autoritativní a nezávislé ověření všech transakcí a kontraktů.

Provoz celého uzlu však bude znamenat náklady na hardwarové prostředky a internetové připojení. Celý uzel musí stáhnout 80–300 GB dat (k lednu 2020, v závislosti na konfiguraci klienta) a uložit je na místní pevný disk. Toto množství dat se každým dnem zvyšuje, když jsou přidávány nové transakce a bloky. O tomto tématu diskutujeme podrobněji v Hardwarové požadavky pro úplný uzel.

Úplný uzel spuštěný v živé síti *mainnet* není pro vývoj v Ethereu nutný. S uzlem *testnet* (který vás spojí s jednou z menších veřejných testovacích bločenek), s místním soukromou bločenkou jako Ganache nebo s cloudovým klientem Ethereum nabízeným poskytovatelem služeb, jako je Infura,

můžete udělat téměř vše, co potřebujete.

Máte také možnost spustit vzdáleného klienta, který neukládá lokální kopii bločenky ani ověřuje bloky a transakce. Tito klienti nabízejí funkčnost peněženky a mohou vytvářet a odesílat transakce. Vzdálené klienty lze použít k připojení k existujícím sítím, jako je váš vlastní úplný uzel, veřejná bločenka, veřejná nebo autorizovaná (důkaz autoritou) testovací síť nebo soukromá místní bločenka. V praxi budete pravděpodobně používat vzdáleného klienta, jako je MetaMask, Emerald Wallet, MyEtherWallet nebo MyCrypto jako pohodlný způsob přepínání mezi všemi různými možnostmi uzlů.

Termíny „vzdálený klient“ a „peněženka“ se používají zaměnitelně, i když existují některé rozdíly. Vzdálený klient obvykle nabízí kromě transakčních funkcí peněženky navíc ještě API (například API web3.js).

Nepleťte si koncept vzdálené peněženky v Ethereum s konceptem *odlehčeného klienta* (který je analogický s Bitcoinovým klientem se zjednodušeným ověřováním plateb). Lehcí klienti ověřují záhlaví bloků a používají Merkle důkazy k ověření zahrnutí transakcí do bločenky a k určení jejich účinků, což jim poskytuje podobnou úroveň zabezpečení jako má úplný uzel. Naopak vzdálení Ethereum klienti neověřují záhlaví bloků nebo transakce. Plně důvěřují úplnému klientovi, že jim umožní přístup k bločence, a proto významně ztrácí záruky zabezpečení a anonymity. Tyto problémy můžete zmírnit pomocí úplného klienta, kterého sami provozujete.

Výhody a nevýhody úplného uzlu

Pokud zvolíte provozování úplného uzlu, pomůžete tím s provozováním sítí, ke kterým se připojujete, ale také vám vzniknou mírné až střední náklady. Pojďme se podívat na některé z výhod a nevýhod.

Výhody:

- Podporuje odolnost a nemožnost provádět cenzuru sítí založených na Ethereu
- Autoritativně ověřuje všechny transakce
- Může bez prostředníka komunikovat s jakýmkoliv kontraktem na veřejné bločence
- Může přímo bez prostředníka zavádět kontrakty do veřejné bločenky
- Může se v offline režimu dotazovat (pouze číst) na stav bločenky (účty, smlouvy, atd.)

- Může se dotazovat v bločence, aniž by třetí strana věděla informace, které čtete

Nevýhody:

*Vyžaduje značné a stále rostoucí hardwarové zdroje a internetové připojení * Plná synchronizace může vyžadovat několik dní po prvním spuštění * Musí být udržováno, upgradováno a být stále online, aby zůstalo synchronizované

Výhody a nevýhody veřejné testovací sítě

Bez ohledu na to, zda se rozhodnete či nerozhodnete provozovat úplný uzel, pravděpodobně budete chtít spustit veřejný testovací uzel. Pojďme se podívat na některé z výhod a nevýhod používání veřejné testovací sítě.

Výhody:

- Testovací uzel musí synchronizovat a ukládat mnohem méně dat - přibližně 45 GB v závislosti na síti.
- Testovací uzel lze plně synchronizovat během několika hodin.
- Zavádění kontraktů nebo provádění transakcí vyžaduje testovací ether, který nemá žádnou hodnotu a lze jej získat zdarma z několika „kohoutků“.
- Testovací sítě jsou veřejné bločenky s mnoha dalšími uživateli a kontrakty které běží „na živo“.

Nevýhody:

- Na testovací síti nelze použít „skutečné“ peníze; běží na testovacím etheru. V důsledku toho nemůžete otestovat bezpečnost proti skutečným protivníkům, protože v sázce není nic.
- Existuje několik aspektů veřejné bločenky, které nemůžete realisticky testovat na testovací síti. Například transakční poplatky, i když jsou nezbytné pro odesílání transakcí, nejsou brány v úvahu na testovací síti, protože plyn je zdarma. Dále testovací sítě nezažívají přetížení sítě, jako to někdy dělá hlavní veřejná síť.

Výhody a nevýhody místní simulace bločenky

Pro mnoho testovacích účelů je nejlepší volbou spustit izolovanou soukromou bločenku. Ganache (dříve pojmenovaný testrpc) je jednou z nejpopulárnějších simulací lokálního bločenky, se kterými

můžete komunikovat, bez dalších účastníků. Sdílí mnoho výhod a nevýhod veřejné testovací sítě, ale najdeme také určité rozdíly.

Výhody:

- Žádná synchronizace a téměř žádná data na disku; sami těžíte první blok
- Není třeba získat testovací ether; sami získáváte odměnu za těžbu bloků, kterou můžete použít pro testování
- Žádní další uživatelé, jen vy
- Žádné další kontrakty, pouze ty, které nasadíte po spuštění

Nevýhody:

- Protože nemá žádné další uživatele, znamená to, že se nechová stejně jako veřejná bločenka. Neexistuje žádná soutěž o transakční prostor nebo pořadí transakcí.
- Protože nejsou žádní další těžaři kromě vás, těžba je předvídatelnější; proto nemůžete otestovat některé scénáře, které se vyskytují na veřejné bločence.
- Protože nejsou žádné další kontrakty, musíte nasadit vše, co chcete testovat, včetně závislostí a knihoven kontraktů.
- Nemůžete znovu vytvořit některé veřejné kontrakty a jejich adresy, abyste otestovali některé scénáře (např. kontrakt The DAO).

Provozování Ethereum klienta

Pokud máte čas a prostředky, měli byste se pokusit provozovat úplný uzel, případně se alespoň teoreticky seznámit s tímto procesem . V této části se zabýváme tím, jak stahovat, kompilovat a provozovat Ethereum klienty Parity a Geth. To vyžaduje určitou znalost používání rozhraní příkazového řádku ve vašem operačním systému. Je vhodné nainstalovat tyto klienty, ať už se rozhodnete je provozovat jako úplné uzly, jako uzly testovací sítě nebo jako klienty do místní soukromé bločenky.

Hardwarové požadavky pro úplný uzel

Než začneme, měli byste mít počítač s dostatečnými systémovými prostředky pro spuštění úplného

uzlu Etherea. K uložení úplné kopie Ethereum bločenky budete potřebovat nejméně 300 GB místa na disku. Pokud chcete také spustit úplný uzel na Ethereum testovací síti, budete potřebovat alespoň dalších 45 GB. Stahování 345 GB dat bločenek může trvat dlouho, takže se doporučuje pracovat na rychlém připojení k internetu.

Synchronizace Ethereum bločenky je velmi náročná na vstupně-výstupní operace. Nejlepší je mít SSD diskovou jednotku. Pokud máte mechanickou jednotku pevného disku (HDD), budete potřebovat alespoň 8 GB paměti RAM, abyste ji mohli použít jako mezipaměť. V opačném případě můžete zjistit, že váš systém je příliš pomalý na to, aby zůstal plně synchronizován.

Minimální požadavky:

- CPU s 2+ jádry
- Nejméně 300 GB volného místa na disku
- Minimálně 4 GB RAM s SSD, 8 GB +, pokud máte HDD
- internetové připojení, minimálně 8 MBit/s

Toto jsou minimální požadavky na synchronizaci úplné (ale prořezané) kopie Ethereum bločenky.

V době psaní této knihy je klient Parity méně náročný na systémové zdroje, takže pokud máte k dispozici pouze omezený hardware, pravděpodobně dosáhnete lepších výsledků pomocí Parity.

Pokud chcete synchronizovat v přiměřeném množství času a uložit všechny vývojové nástroje, knihovny, klienty a bločenky, o nichž pojednáváme v této knize, budete potřebovat schopnější počítač počítač.

- Doporučené specifikace: *
- Rychlý procesor se 4 a více jádry
- 16 GB + RAM
- Rychlé SSD s nejméně 500 GB volného místa
- Interentové připojení 25+ MBit/s

Je obtížné předvídat, jak rychle se velikost bločenky zvětšovata kdy bude zapotřebí více místa na disku, takže před zahájením synchronizace doporučujeme zkontrolovat nejnovější velikost bločenky.

Zde uvedené požadavky na velikost disku předpokládají, že budete provozovat úplný s výchozím nastavením, kde je bločenka „ořezávána“ od starých stavových dat. Pokud místo toho spustíte úplný „archivační“ uzel, ve kterém je na disku zachován celý stav, bude pravděpodobně vyžadovat více než 1 TB místa na disku.

Tyto odkazy poskytují aktuální odhady velikosti bločenky:

- Ethereum [https://bitinfocharts.com/ethereum/]
- Ethereum Classic [https://bitinfocharts.com/ethereum%20classic/]

Softwarové požadavky na zprovoznění klienta (uzel)

Tato část se týká softwaru klientů Parity a Geth. Předpokládá také, že používáte prostředí příkazového řádku podobné Unixu. Příklady ukazují příkazy a výstupy, které se objevují v operačním systému Ubuntu GNU / Linux, na kterém běží bash shell (prostředí pro provádění příkazového řádku).

Obvykle bude mít každá bločenka svou vlastní verzi Geth, zatímco Parity poskytuje podporu současně více bločenkám založených na Ethereu (Ethereum, Ethereum Classic, pass: [Ellaism], Expanse, Musicoin) se stejným klientským stahováním.

V mnoha příkladech v této kapitole , budeme používat rozhraní příkazového řádku operačního systému (známé také jako „shell“), přístupné prostřednictvím „terminálové“ aplikace. Shell zobrazí výzvu; zadáte příkaz a shell odpoví nějakým textem a novou výzvou k dalšímu příkazu. Výzva se může ve vašem systému lišit, ale v následujících příkladech je označena symbolem $. V příkladech, když vidíte text za symbolem $, nezadávejte symbol $, ale zadejte příkaz bezprostředně za ním (zobrazen tučně) a poté stisknutím klávesy Enter proveďte příkaz. V příkladech jsou řádky pod každým příkazem reakce operačního systému na tento příkaz. Když uvidíte další předponu $, budete vědět, že se jedná o nový příkaz, a měli byste postup opakovat.

Než začneme, možná budete muset nainstalovat nějaký software. Pokud jste v počítači, který právě používáte, nikdy neprovedli žádný vývoj softwaru, budete pravděpodobně muset nainstalovat některé základní nástroje. V následujících příkladech budete muset nainstalovat systém správy zdrojového kódu git; golang, programovací jazyk Go a standardní knihovny; a Rust, systémový

programovací jazyk.

Git lze nainstalovat podle pokynů na adrese https://git-scm.com [].

Go lze nainstalovat podle pokynů na https://golang.org, nebo https://github.com/golang/go/wiki/Ubuntu, pokud používáte Ubuntu.

Požadavky Geth se liší, ale pokud se budete držet Go verze 1.10 nebo vyšší, měli byste být schopni sestavit libovolnou verzi Geth, kterou chcete. Samozřejmě byste se měli vždy podívat do dokumentace k vaší vybrané variantě Geth.

Verze golang, která je nainstalována ve vašem operačním systému nebo je k dispozici od správce balíčků vašeho systému, může být výrazně starší než 1.10. Pokud ano, odeberte ji a nainstalujte nejnovější verzi z https://golang.org/.

Rust lze nainstalovat podle pokynů na adrese https://www.rustup.rs/.

Parity vyžaduje verzi Rust 1.27 nebo vyšší.

Parity také vyžaduje některé softwarové knihovny, jako jsou OpenSSL a libudev. Chcete-li je nainstalovat do systému kompatibilního s Ubuntu nebo Debian GNU / Linux, použijte následující pass: [příkaz]:

```
<pre data-type="programlisting">
$ <strong>sudo apt-get install openssl libssl-dev libudev-dev cmake
clang</strong>
</pre>
```

U ostatních operačních systémů použijte správce balíčků vašeho operačního systému nebo podle Pokynů na Wiki [https://github.com/paritytech/parity/wiki/Setup] nainstalujte požadované knihovny.

Nyní, když máte nainstalované git, golang, Rust a potřebné knihovny, pusťme se do práce!

Parity

Parity je implementace Ethereum klienta s úplným uzlem a prohlížeče DApp. Bylo napsáno „od základu“ v jazyce Rust, systémovém programovacím jazyce, s cílem vybudovat modulárního,

bezpečného a škálovatelného Ethereum klienta. Parity vyvíjí britská společnost Parity Tech a je uvolněna pod licencí na bezplatný software GPLv3.

Upozornění: Jeden z autorů této knihy, Dr. Gavin Wood, je zakladatelem Parity Tech a napsal velkou část klienta Parity. Parity představuje asi 25% nainstalované klientské základny Ethereum.

Chcete-li nainstalovat Parity, můžete použít Rust správce balíčků cargo nebo si stáhnout zdrojový kód z GitHubu. Správce balíčků také stáhne zdrojový kód, takže mezi těmito dvěma možnostmi není velký rozdíl. V další části vám ukážeme, jak si Parity stáhnout a zkompilovat.

Instalace Parity

The Parity Wiki [https://wiki.parity.io/Setup] nabízí pokyny pro nainstalování Parity v různých prostředích. Ukážeme vám, jak nainstalovat Paritu ze zdroje. Předpokládá se, že jste již nainstalovali Rust pomocí rustup (viz Softwarové požadavky na zprovoznění klienta (uzel)).

Nejprve získejte zdrojový kód z GitHubu:

```
<pre data-type="programlisting">
$ <strong>git clone https://github.com/paritytech/parity</strong>
</pre>
```

Poté přejděte do adresáře *parity* a vytvořte spustitelný soubor pomocí cargo:

```
<pre data-type="programlisting">
$ <strong>cd parity</strong>
$ <strong>cargo install --path .</strong>
</pre>
```

Pokud vše půjde dobře, měli byste vidět něco jako:

```
<pre data-type="programlisting">
$ <strong>cargo install --path .</strong>
Installing parity-ethereum v2.7.0 (/root/parity)
Updating crates.io index
Updating git repository `https://github.com/paritytech/rust-ctrlc.git`
Updating git repository `https://github.com/paritytech/app-dirs-rs`
Updating git repository

 [...]

Compiling parity-ethereum v2.7.0 (/root/parity)
Finished release [optimized] target(s) in 10m 16s
Installing /root/.cargo/bin/parity
Installed package `parity-ethereum v2.7.0 (/root/parity)` (executable
`parity`)
$
</pre>
```

Vyzkoušejte spustit parity, abyste zjistili, zda je nainstalován, vyvoláním volby --version:

```
<pre data-type="programlisting">
$ <strong>parity --version</strong>
Parity Ethereum Client.
  version Parity-Ethereum/v2.7.0-unstable-b69a33b3a-20200124/x86_64-
unknown-linux-gnu/rustc1.40.0
Copyright 2015-2020 Parity Technologies (UK) Ltd.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

By Wood/Paronyan/Kotewicz/Drwi¬ga/Volf/Greeff
   Habermeier/Czaban/Gotchac/Redman/Nikolsky
   Schoedon/Tang/Adolfsson/Silva/Palm/Hirsz et al.
$
</pre>
```

Skvělý! Nyní, když je Parity nainstalovaná, můžete synchronizovat bločenku a začít s některými základními možnostmi příkazového řádku.

Go-Ethereum (Geth)

Geth je implementován v jazyku Go, je aktivně vyvíjen Nadací Etherea, takže se považuje za „oficiální“ implementaci Ethereum klienta. Obvykle bude mít každá bločenka na bázi Ethereum vlastní implementaci Geth. Pokud používáte Geth, budete se chtít ujistit, že si vyberete správnou verzi pro svojí bločenku pomocí jednoho z následujících odkazů na úložiště:

- Ethereum [https://github.com/ethereum/go-ethereum] (nebo https://geth.ethereum.org/)
- Ethereum Classic [https://github.com/etclabscore/go-ethereum]
- Ellaism [https://github.com/ellaism/go-ellaism]
- Expanse [https://github.com/expanse-org/go-expanse]
- Musicoin [https://github.com/Musicoin/go-musicoin]
- Ubiq [https://github.com/ubiq/go-ubiq]

Můžete také přeskočit tyto pokyny a nainstalovat předkompilovaný binární soubor pro platformu, kterou si vyberete. Předkompilovaná vydání se instalují mnohem snadněji a najdete je v části „vydání“ na kterémkoli z níže uvedených úložišť. Více se však můžete naučit vlastnoručním stažením a kompilací softwaru.

Okopírování úložiště

Prvním krokem je okopírování úložiště Git, aby se získala kopie zdrojového kódu.

Chcete-li vytvořit místní kopii zvoleného úložiště, použijte příkaz git v domovském adresáři nebo v libovolném adresáři, který používáte pro vývoj:

```
<pre data-type="programlisting">
$ <strong>git clone &lt;Repository Link&gt;</strong>
</pre>
```

Při kopírování úložiště do místního systému byste měli vidět zprávu o postupu:

```
Cloning into 'go-ethereum'...
remote: Enumerating objects: 86915, done.
remote: Total 86915 (delta 0), reused 0 (delta 0), pack-reused 86915
Receiving objects: 100% (86915/86915), 134.73 MiB | 29.30 MiB/s, done.
Resolving deltas: 100% (57590/57590), done.
```

Skvělý! Nyní, když máte lokální kopii Geth, můžete sestavit spustitelný soubor pro vaši platformu.

Sestavení Geth ze zdrojového kódu

Chcete-li sestavit Geth, přejděte do adresáře, kde byl zdrojový kód stažen, a použijte příkaz make:

```
<pre data-type="programlisting">
$ <strong>cd go-ethereum</strong>
$ <strong>make geth</strong>
</pre>
```

Pokud vše půjde dobře, uvidíte kompilátor Go, jak vytváří každou komponentu, dokud nevytvoří spustitelný soubor geth:

```
build/env.sh go run build/ci.go install ./cmd/geth
>>> /usr/local/go/bin/go install -ldflags -X
main.gitCommit=58a1e13e6dd7f52a1d...
github.com/ethereum/go-ethereum/common/hexutil
github.com/ethereum/go-ethereum/common/math
github.com/ethereum/go-ethereum/crypto/sha3
github.com/ethereum/go-ethereum/rlp
github.com/ethereum/go-ethereum/crypto/secp256k1
github.com/ethereum/go-ethereum/common
[...]
github.com/ethereum/go-ethereum/cmd/utils
github.com/ethereum/go-ethereum/cmd/geth
Done building.
Run "build/bin/geth" to launch geth.
$
```

Ujistěte se, že geth funguje, aniž byste ho skutečně spuštili.

```
<pre data-type="programlisting">
$ <strong>./build/bin/geth version</strong>

Geth
Version: 1.9.11-unstable
Git Commit: 0b284f6c6cfc6df452ca23f9454ee16a6330cb8e
Git Commit Date: 20200123
Architecture: amd64
Protocol Versions: [64 63]
Go Version: go1.13.4
Operating System: linux
[...]
</pre>
```

Váš příkaz geth version může zobrazovat mírně odlišné informace, ale měli byste vidět zprávu o verzi, která je podobná té, kterou vidíte zde.

Následující oddíly vysvětlují počáteční synchronizací Ethereum bločenky.

První synchronizace bločenek založených na Ethereu

Tradičně při synchronizaci Ethereum bločenky váš klient stahuje a ověřuje každý blok a každou transakci od samého začátku - tj. od základního (genesis) bloku.

I když je možné plně synchronizovat bločenku tímto způsobem, bude tento typ synchronizace trvat velmi dlouho a bude vyžadovat vysoké nároky na zdroje (bude vyžadovat mnohem více RAM a bude trvat velmi dlouho, pokud nemáte rychlé úložný prostor).

Koncem roku 2016 bylo mnoho bločenek založených na Ethereu obětí útoků odepření služby. Postižené bločenky budou mít při plné synchronizaci tendenci se synchronizovat pomalu.

Například u Etherea nový klient rychle postupuje, dokud nedosáhne bloku 2 283 397. Tento blok byl těžen 18. září 2016 a označuje začátek útoků DoS. Od tohoto bloku až do bloku 2 700 031 (26. listopadu 2016) se ověřování transakcí stává extrémně pomalým,vyžadujícím mnoho operační paměti a vstupně-výstupních operací. Výsledkem je doba ověření delší než 1 minuta na blok. Ethereum implementovalo řadu upgradů pomocí tvrdých rozštěpení, aby vyřešilo základní zranitelnosti, které byly zneužity při útokech DoS. Tyto upgrady také vyčistily bločenku odstraněním přibližně 20 milionů prázdných účtů vytvořených spamovými transakcemi.

Pokud provádíte synchronizaci s plnou validací, váš klient zpomalí a může trvat několik dní nebo možná i déle, než budou ověřeny bloky ovlivněné útoky DoS.

Naštěstí většina Ethereum klientů nyní ve výchozím nastavení provádí „rychlou“ synchronizaci, která přeskočí úplnou validaci transakcí, dokud se nesynchronizuje s aktuálním vrcholem bločenky, a teprve poté pokračuje v plné validaci.

Geth provádí ve výchozím nastavení rychlou synchronizaci pro Ethereum. Možná budete muset postupovat podle konkrétních pokynů pro další vybrané bločenky založené na Ethereu.

Parity ve výchozím nastavení také provádí rychlou synchronizaci.

Geth může provádět rychlou synchronizaci pouze při spuštění s prázdnou databází bloků. Pokud jste již zahájili synchronizaci bez rychlého režimu, Geth se nemůže přepnout. Je rychlejší odstranit datový adresář bločenky a začít rychlou synchronizaci od začátku, než pokračovat v synchronizaci s plnou validací. Při mazání dat bločenky buďte opatrní, abyste neodstranili žádné peněženky!

Spuštění Geth nebo Parity

Nyní, když rozumíte výzvám „první synchronizace“, jste připraveni spustit Ethereum klienta a synchronizovat bločenku. Pro Geth i Parity můžete pomocí volby --help zobrazit všechny konfigurační parametry. Výchozí nastavení je obvykle rozumné a vhodné pro většinu použití. Nejprve můžete nakonfigurovat libovolné volitelné parametry podle svých potřeb, poté spusťte Geth nebo Parity a synchronizujte bločenku. Pak počkejte …

Synchronizace Ethereum bločenky bude trvat od půl dne na velmi rychlém systému se spoustou paměti RAM až po několik dní na pomalejším systému.

Rozhraní JSON-RPC

Ethereum klienti nabízejí rozhraní pro programování aplikací a sada příkazů vzdáleného volání procedur (RPC), které jsou kódovány v JavaScript Object Notation (JSON). Bývá to označované jako *JSON-RPC API*. JSON-RPC API je v zásadě rozhraní, které nám umožňuje psát programy, které používají Ethereum klienta jako *brány*, do sítě Ethereum a bločenky.

Obvykle je rozhraní RPC nabízeno jako služba HTTP na portu 8545. Z bezpečnostních důvodů je ve výchozím nastavení omezeno, aby přijímalo pouze připojení z localhost (IP adresa vašeho vlastního počítače, která je 127.0.0.1).

Pro přístup k JSON-RPC API můžete použít specializovanou knihovnu (napsanou v programovacím jazyce dle vašeho výběru), která poskytuje volání "pahýlu" funkce odpovídajícímu každému dostupnému příkazu RPC, nebo můžete ručně vytvářet požadavky HTTP a odesílat / přijímat JSON kódované žádosti. K volání rozhraní RPC můžete dokonce použít generického HTTP klienta z příkazového řádku, například curl. Zkusme to. Nejprve se ujistěte, že máte Geth spuštěný a nakonfigurovaný s --rpc, který umožňuje HTTP přístup k rozhraní RPC, poté přepněte do nového okna terminálu (např. s pomocí Ctrl-Shift-N nebo Ctrl-Shift-T ve stávajícím okno terminálu), jak je znázorněno zde:

```
<pre data-type="programlisting">
$ <strong>curl -X POST -H "Content-Type: application/json" --data \
  '{"jsonrpc":"2.0","method":"web3_clientVersion","params":[],"id":1}' \
  http://localhost:8545</strong>

{"jsonrpc":"2.0","id":1,
"result":"Geth/v1.9.11-unstable-0b284f6c-20200123/linux-amd64/go1.13.4"}
</pre>
```

V tomto příkladu používáme cur + k navázání HTTP spojení s adresou _http://localhost:8545_. Již běží +geth, které nabízí rozhraní JSON-RPC API jako HTTP službu na portu 8545. Nařídíme curl, aby použil příkaz HTTP POST a označil obsah jako typ application/json. Nakonec předáme požadavek kódovaný JSON jako data součást našeho HTTP požadavku. Většina našeho příkazového řádku právě nastavuje curl, aby HTTP připojení správně pracovalo. Zajímavou součástí je JSON-RPC příkaz, který právě vytváříme:

```
{"jsonrpc":"2.0","method":"web3_clientVersion","params":[],"id":1}
```

JSON-RPC požadavek je formátován podle JSON-RPC 2.0 specifikace [https://www.jsonrpc.org/specification]. Každá žádost obsahuje čtyři prvky:

jsonrpc

Verze protokolu JSON-RPC. MUSÍ to být přesně "2.0".

method

Název metody, která má být vyvolána.

params

Strukturovaná hodnota, která uchovává hodnoty parametrů, které mají být použity během vyvolání metody. Tento prvek může být vynechán.

id

Identifikátor vytvořený klientem, který MUSÍ obsahovat hodnotu String, Number nebo NULL, je-li zahrnuta. Server MUSÍ odpovídat stejnou hodnotou v objektu odpovědi, pokud je zahrnut. Tento člen se používá ke korelaci kontextu mezi dvěma objekty.

Parametr id se používá primárně, když zadáváte více požadavků v jednom JSON-RPC volání, což se nazývá *dávkování*. Dávkování se používá k omezení režie za nová HTTP a TCP připojení pro každý požadavek. Například v Ethereu bychom použili dávkování, pokud bychom chtěli získat tisíce transakcí přes jedno HTTP připojení. Při dávkování nastavíte pro každou žádost jiné id a poté ji spárujeme s odpovídající odpovědí z JSON-RPC serveru s daným id. Nejjednodušší způsob, jak to provést, je udržovat počitadlo a zvyšovat hodnotu pro každý požadavek.

Odpověď, kterou dostáváme, je:

```
{"jsonrpc":"2.0","id":1,
"result":"Geth/v1.9.11-unstable-0b284f6c-20200123/linux-amd64/go1.13.4"}
```

To nám říká, že rozhraní JSON-RPC API je obsluhováno klientem Geth verze 1.13.4.

Zkusme něco zajímavějšího. V dalším příkladu požádáme JSON-RPC API o aktuální cenu plynu ve wei:

```
<pre data-type="programlisting">
$ <strong>curl -X POST -H "Content-Type: application/json" --data \
  '{"jsonrpc":"2.0","method":"eth_gasPrice","params":[],"id":4213}' \
  http://localhost:8545</strong>

{"jsonrpc":"2.0","id":4213,"result":"0x430e23400"}
</pre>
```

Odpověď 0x430e23400 nám říká, že aktuální cena plynu je 18 gwei (gigawei nebo miliard wei). Pokud, stejně jako my, nemyslíte v šestnáctkové soustavě, můžete ji v příkazovém řádku převést na číslo v desítkové soustavě pomocí drobného bash triku:

```
<pre data-type="programlisting">
$ <strong>echo $((0x430e23400))</strong>

18000000000
</pre>
```

Plné rozhraní JSON-RPC API lze prozkoumat na Ethereum wiki [https://github.com/ethereum/wiki/wiki/JSON-RPC].

Parity režim kompatibility s Geth

Parity má speciální „režim kompatibility s Geth“, ve kterém nabízí JSON-RPC API, které je totožné s rozhraním nabízeným Geth. Pro spuštění Parity v tomto režimu použijte přepínač --geth :

```
<pre data-type="programlisting">
$ <strong>parity --geth</strong>
</pre>
```

Ethereum klienti pro vzdálený přístup

Klienti pro vzdálený přístup nabízejí podmnožinu funkčnosti úplného klienta. Neukládají celo Ethereum bločenku, takže se nastavují rychleji a vyžadují mnohem méně uložených dat.

Tito klienti obvykle poskytují možnost provádět jednu nebo více z následujících akcí:

- Spravovat soukromé klíče a Ethereum adresy v peněžence.
- Vytvářet, podepisovat a odesílat transakce.
- Pracovat s chytrými kontrakty pomocí užitečného datového nákladu (data payload)
- Procházet a komunikovat s DApps.
- Nabídka odkazů na externí služby, jako jsou průzkumníci bloků.
- Převod etherových jednotek a načítání směnných kurzů z externích zdrojů.
- Vložit web3 instanci do webového prohlížeče jako JavaScript objekt.
- Použít web3 instanci poskytnutou / napojenou do prohlížeče jiným klientem.
- Přístup ke službám RPC v místním nebo vzdáleném Ethereum uzlu .

Někteří klienti pro vzdálený přístup, například mobilní peněženky, nabízejí pouze základní funkce peněženky. Ostatní vzdálení klienti jsou plnohodnotné prohlížeče DApp. Vzdálení klienti obvykle nabízejí některé funkce Ethereum klienta úplným uzlem, aniž by synchronizovali místní kopii Ethereum bločenky připojením k úplnému uzlu, který se spouští jinde, např. lokálně na vašem počítači nebo na webovém serveru nebo prostřednictvím třetí strany na jejich serverech.

Pojďme se podívat na některé z nejpopulárnějších klientů pro vzdálený přístup a funkce, které nabízejí.

Mobilní peněženky pro chytré telefony

Všechny mobilní peněženky jsou klienty pr ovzdálený přístup, protože chytré telefony nemají dostatečné zdroje pro provozování úplného Ethereum klienta. Odlehčení Ethereum klienti jsou ve stádiu vývoje a obecně se nepoužívají. V případě Parity je odlehčený klient označen jako „experimentální“ a lze jej použít spuštěním parity s volbou --light.

Populární mobilní peněženky zahrnují následující (uvádíme je pouze jako příklady; nejedná se o potvrzení ani o označení bezpečnosti nebo funkčnosti těchto peněženek):

Jaxx [https://jaxx.io]

> Víceměnová mobilní peněženka založená na mnemotechnických semínkách BIP-39 , s podporou Bitcoinu, Litecoinu, Etherea, Etherea Classic, ZCash, různých ERC20 tokenů a mnoha dalších měn. Jaxx je k dispozici pro Android a iOS, jako webová peněženka implementovaná

pomocí rozšíření prohlížeče a jako peněženka pro stolní počítače pro různé operační systémy.

Status [https://status.im]

Mobilní peněženka a prohlížeč DApp s podporou řady tokenů a populárních DApps. K dispozici pro iOS a Android.

Trust Wallet [https://trustwalletapp.com/]

Mobilní víceměnová peněženka, která podporuje Ethereum a Ethereum Classic, stejně jako tokeny ERC20 a ERC223. Trust Wallet je k dispozici pro iOS a Android.

Cipher Browser [https://www.cipherbrowser.com]

Plně vybavený mobilní prohlížeč DApp a Ethereum peněženka, který umožňuje integraci s aplikacemi a Ethereum tokeny. K dispozici pro iOS a Android.

Webové peněženky

Mnoho různých peněženek prohlížečů DApp je k dispozici jako rozšíření webových prohlížečů, jako je Chrome a Firefox. Jedná se o vzdálené klienty, které běží ve vašem prohlížeči.

Mezi nejoblíbenější patří MetaMask, Jaxx, MyEtherWallet a MyCrypto.

MetaMask

MetaMask [https://metamask.io/], představený Základy Etherea je všestranná webová peněženka, RPC klient, a základní průzkumník kontraktů. Je k dispozici v prohlížečích Chrome, Firefox, Opera a Brave.

Na rozdíl od jiných webových peněženek, MetaMask napojuje web3 instanci do kontextu JavaScriptu v prohlížeči, který funguje jako klient RPC, který se připojuje k celé řadě Ethereum bločenek (hlavní síť, testovací síť Ropsten, testovací síť Kovan, místní RPC uzel, atd.). Schopnost vložit web3 instanci a fungovat jako brána k externím RPC službám činí z MetaMasku velmi silný nástroj pro vývojáře i uživatele. Lze jej kombinovat například s MyEtherWallet nebo MyCrypto, funguje jako web3 poskytovatel a RPC brána pro tyto nástroje.

Jaxx

Jaxx [https://jaxx.io], , který byl v předchozí sekci představen jako mobilní peněženka, je k dispozici také jako rozšíření pro Chrome a Firefox a jako peněženka pro stolní počítače.

MyEtherWallet (MEW)

MyEtherWallet [https://www.myetherwallet.com/] webový klient se vzdáleným přístupem naprogramovaný v JavaScriptu, který nabízí:

- Most k populárním hardwarovým peněženkám, jako jsou Trezor a Ledger
- Web3 rozhraní, které se může připojit k web3 instanci jiného klienta (např. MetaMask)
- RPC klient , který se může připojit k úplnému Ethereum klientovi
- Základní rozhraní, které může spolupracovat s chytrými kontrakty, po zadání adresy kontraktu a aplikační binární rozhraní (ABI)
- Mobilní aplikace MEWConnect, která umožňuje používat kompatibilní zařízení Android nebo iOS k ukládání prostředků, podobně jako hardwarová peněženka.
- Softwarová peněženka běžící v JavaScriptu

Při přístupu k MyEtherWallet a dalším webovým peněženkám naprogramovaným v JavaScriptu musíte být velmi opatrní, protože jsou častým cílem phishingu. K přístupu na správnou webovou adresu URL vždy používejte záložku a ne webový vyhledávač.

MyCrypto

In early 2018, the Projekt MyEtherWallet se rozdělil do dvou konkurenčních implementací, vedených dvěma nezávislými vývojovými týmy: „rozštěpení“ jak se tato situace nazývá při vývoj software s otevřeným zdrojovým kódem. Tyto dva projekty se nazývají MyEtherWallet (původní značka) a MyCrypto [https://mycrypto.com/]. MyCrypto nabízí téměř totožnou funkčnost jako MyEtherWallet, ale namísto použití MEWConnect nabízí připojení k mobilní aplikaci Parity Signer. Stejně jako MEWConnect, Parity Signer ukládá klíče do telefonu a rozhraní s MyCrypto podobným způsobem jako hardwarová peněženka.

Mist (zastaralá)

Mist [https://github.com/ethereum/mist] byl první Ethereum prohlížeč, vytvořený Nadací Ethereum. Obsahoval webovou peněženku, která byla první implementací standardu tokenů ERC20 (Fabian Vogelsteller, autor ERC20, byl také hlavním vývojářem Mist). Mist byla také první peněženka, která zavedla kontrolní součet camelCase (EIP-55). V březnu 2019 byla Mist prohlášena za zastaralou a neměla by se nadále používat.

Závěry

V této kapitole jsme prozkoumali Ethereum klienty. Stáhli jste, nainstalovali a synchronizovali klienta, stali jste se účastníkem sítě Ethereum a přispívali jste ke zdraví a stabilitě systému replikováním bločenky na vašem počítači.

Kryptografie

Jednou ze základních technologií Etherea je *kryptografie*, což je odvětví matematiky používané v počítačové bezpečnosti. Kryptografie znamená v řečtině „tajné psaní“, ale studium kryptografie zahrnuje více než jen tajné psaní, které se označuje jako *šifrování*. Kryptografie může být například také použita k prokázání znalosti tajemství bez odhalení tohoto tajemství (např. s digitálním podpisem) nebo k prokázání pravosti dat (např. s digitálními otisky, také známými jako „haš“). Tyto typy kryptografických důkazů jsou matematické nástroje kritické pro fungování platformy Ethereum (a ve skutečnosti všech bločenkových systémů) a jsou také široce používány v Ethereum aplikacích Ethereum.

Všimněte si, že v době napsání této knihy žádná část Ethereum protokolu nezahrnuje šifrování; to znamená, že veškerá komunikace s Ethereum platformou a mezi uzly (včetně transakčních dat) je nešifrovaná a může (dokonce je to nutné) ji přečíst kdokoli. To je tak, aby si každý mohl ověřit správnost aktualizací stavu a dosáhnout konsensu. V budoucnu budou k dispozici pokročilé kryptografické nástroje, jako jsou důkazy o nulových znalostech a homomorfní šifrování, které umožní, aby některé šifrované výpočty byly zaznamenány na bločence a přitom stále umožňovaly shodu; Přestože jsou již naplánovány, nejsou ještě nasazeny.

V této kapitole představíme část kryptografie používané v Ethereum: kryptografii veřejných klíčů (PKC), která se používá k řízení vlastnictví finančních prostředků, ve formě soukromých klíčů a adres.

Klíče a adresy

Jak jsme viděli dříve v této knize, Ethereum má dva různé typy účtů: *externě vlastněné účty* (EOA) a *kontrakty*. Vlastnictví etheru prostřednictvím EOA je řízeno prostřednictvím digitálních *soukromých klíčů*, *Ethereum addres* a *digitálních podpisů*. Soukromé klíče jsou jádrem veškeré uživatelské interakce s Ethereum. Ve skutečnosti jsou adresy účtů odvozeny přímo ze soukromých klíčů: soukromý klíč jednoznačně určuje jednu Ethereum adresu , známou také jako *účet*.

Soukromé klíče se v Ethereum systému nejsou přímo používány; nikdy nejsou přenášeny ani ukládány v Ethereu. To znamená, že soukromé klíče by měly zůstat soukromé a nikdy by se neměly objevovat ve zprávách předávaných do sítě, ani by neměly být ukládány v bločence; v systému Ethereum jsou vždy přenášeny a ukládány pouze adresy účtů a digitální podpisy. Další informace o tom, jak udržovat soukromé klíče v bezpečí, viz Kontrola a odpovědnost a Peněženky.

Přístup k finančním prostředkům a jejich ovládání se dosahuje pomocí digitálních podpisů, které se vytvářejí také pomocí soukromého klíče. Ethereum Transakce vyžadují, aby byl do bločenky zahrnut platný digitální podpis. Každý, kdo má kopii soukromého klíče, má kontrolu nad odpovídajícím účtem a jakýmkoli etherem, který má tento účet v držení. Za předpokladu, že uživatel udržuje svůj soukromý klíč v bezpečí, pprokazují digitální podpisy v Ethereum transakcích skutečného vlastníka prostředků, protože prokazují vlastnictví soukromého klíče.

V systémech založených na kryptografii veřejného klíče, jako je Ethereum, jsou klíče v párech sestávajících ze soukromého (tajného) klíče a veřejného klíče. Veřejný klíč považujte za obdobu číslu bankovního účtu a soukromý klíč jako tajný PIN. Soukromý poskytuje kontrolu nad účtem a veřejný klíč ostatním lidem identifikuje účet. Samotné soukromé klíče jsou uživatele Etherea si velmi zřídka prohlížejí; z větší části jsou uloženy (v šifrované formě) ve speciálních souborech a spravovány softwarem Ethereum peněženky.

V platební části Ethereum transakce je zamýšlený příjemce představován adresou Ethereum, která se používá stejným způsobem jako údaje o účtu příjemce pro bankovní převod. Jak brzy uvidíme podrobněji, Ethereum adresa pro EOA je vytvořena z veřejného klíče a nepřímo z k němu odpovídajícímu soukromému klíči. Ne všechny adresy Ethereum však představují páry veřejného a soukromého klíče; mohou také představovat kontrakty, které, jak uvidíme v Chytré kontrakty a Solidity nejsou pokryty soukromými klíči.

Ve zbývající části této kapitoly nejprve podrobněji prozkoumáme základní kryptografii a vysvětlíme matematiku používanou v Ethereu. Poté se podíváme na to, jak jsou generovány, ukládány a spravovány klíče. Nakonec přezkoumáme různé formáty kódování používané k reprezentaci soukromých klíčů, veřejných klíčů a adres.

Kryptografie veřejného klíče a kryptoměna

Kryptografie veřejného klíče (nazývaná „asymetrická kryptografie“ je základní součástí moderní informační bezpečnosti. Protokol výměny klíčů, prvně publikovali v 70. letech Martin Hellman, Whitfield Diffie a Ralph Merkle, byl to monumentální průlom, který podnítil první velkou vlnu veřejného zájmu v oblasti kryptografie. Před sedmdesátými léty, silné kryptografické znalosti byly drženy v tajnosti vládami.

Kryptografie veřejného klíče používá jedinečné klíče k zabezpečení informací. Tyto klíče jsou založeny na matematických funkcích, které mají speciální vlastnost: je snadné je spočítat, ale je obtížné vypočítat jejich inverzi. Na základě těchto funkcí umožňuje kryptografie vytváření

digitálních tajemství a nepadělatelné digitální podpisy, které jsou zabezpečeny zákony matematiky.

Například vynásobení dvou velkých prvočísel dohromady je jednoduché. Ale k zadanému součinu dvou velkých prvočísel je velmi obtížné najít jeho rozklad na činitele (problém zvaný *prvočíselný rozklad*). Řekněme, že je zadáno číslo 8,018,009 a řekneme vám, že je to produkt dvou prvočísel. Nalezení těchto dvou prvočísel je pro vás mnohem těžší, než bylo pro mě, abych je vynásobil a vytvořil 8 018 009.

Některé z těchto matematických funkcí lze snadno převrátit, pokud znáte nějaké tajné informace. Pokud vám v předchozím příkladu řeknu, že jedním z činitelů je 2 003, můžete triviálně najít toho druhého pomocí jednoduchého dělení: 8 018 009 ÷ 2 003 = 4 003. Takové funkce se často nazývají *funkce se zadními vrátky*, protože je velmi obtížné je invertovat, pokud nedostanete kousek tajné informace, kterou lze použít jako zkratku pro obrácení funkce.

Pokročilejší kategorie matematických funkcí, která je užitečná v kryptografii, je založena na aritmetických operacích na eliptické křivce. V aritmetice eliptické křivky je násobení následované zbytkem po dělení (modulo) prvočíslem jednoduché, ale inverzní funkce dělení je prakticky nemožná. Tomu se říká *problém diskrétního logaritmu* a momentálně neexistují žádné známá zadní vrátka. *Kryptografie eliptických křivek* je široce používána v moderních počítačových systémech a je základem pro použití soukromých klíčů a digitálních podpisů v Ethereum (a dalších kryptoměnách).

Pokud se chcete dozvědět více o kryptografii a matematických funkcích používaných v moderní kryptografii, podívejte se na následující zdroje:

- Kryptografie [http://bit.ly/2DcwNhn]
- Funkce se zadními vrátky [http://bit.ly/2zeZV3c]
- Prvočíselný rozklad [http://bit.ly/2ACJjnV]
- Diskrétní logaritmus [http://bit.ly/2Q7mZYI]
- Kryptografie eliptických křivek [http://bit.ly/2zfeKCP]

V Ethereum používáme kryptografii veřejného klíče (známou také jako asymetrická kryptografie) k vytvoření páru veřejného a soukromého klíče, o kterém jsme hovořili v této kapitole. Jsou považovány za „pár“, protože veřejný klíč je odvozen ze soukromého klíče. Společně představují Ethereum účet tím, že poskytují veřejně přístupný název účtu (adresa) a soukromou kontrolu nad přístupem k jakémukoli etheru na tomto účtu a ověření (autentizaci), kterou účet potřebuje při

používání chytrých kontraktů. Soukromý klíč řídí přístup tím, že je jedinečnou informací potřebnou k vytvoření _ digitálních podpisů_, které jsou požadovány pro podpis transakce a pro možnost utrácet jakékoli prostředky na účtu. Digitální podpisy se používají také k ověření vlastníků nebo uživatelé chytrých kontraktů, jak uvidíme v <<smart_contracts_chapter> >.

Ve většině implementací peněženek jsou pro větší pohodlí soukromé a veřejné klíče ukládány společně jako *pár klíčů*. Veřejný klíč však lze z soukromého klíče jednoduše vypočítat, takže je možné ukládat pouze soukromé klíče.

K jakékoli zprávě lze vytvořit její digitální podpis. U Ethereum transakcí se jako zpráva použijí podrobnosti samotné transakce. Matematika kryptografie - v tomto případě kryptografie eliptických křivek - poskytuje způsob, jak lze zprávu (tj. podrobnosti transakce) zkombinovat se soukromým klíčem a vytvořit kód, který lze vyrobit pouze se znalostí soukromého klíče. Tento kód se nazývá digitální podpis. Všimněte si, že Ethereum transakce je v podstatě žádost o přístup k určitému účtu s konkrétní Ethereum adresou. Pokud je transakce odeslána do Ethereum sítě za účelem přesunu prostředků nebo interakce s chytrým kontraktem, musí být odeslána s digitálním podpisem vytvořeným soukromým klíčem odpovídajícímu dané Ethereum adrese. Matematika eliptické křivky znamená, že *kdokoli* může ověřit, že transakce je platná, a to kontrolou, že digitální podpis odpovídá podrobnostem transakce a Ethereum adrese, ke které se požaduje přístup. Ověření vůbec nezahrnuje soukromý klíč; ten zůstává utajený. Proces ověření však zajišťuje, že nejsou pochybnosti o tom, že transakce mohla přijít pouze od někoho, kdo má soukromý klíč, který odpovídá veřejnému klíči pro danou Ethereum adresu. Toto je „magie“ kryptografie veřejného klíče.

Součást Ethereum protokolu není žádné šifrování - všechny zprávy, které jsou zasílány jako součást provozu Ethereum sítě, může přečíst kdokoli, dokonce je tato vlastnost nutná. Soukromé klíče se proto používají pouze k vytváření digitálních podpisů pro ověřování transakcí.

Soukromé klíče

Soukromý klíč je jednoduše číslo, které je náhodně vybráno. Vlastnictví a kontrola soukromého klíče je základ uživatelské kontroly nad všemi prostředky asociovanými s odpovídající Ethereum adresou a také přístupem ke kontraktům, ovládaných touto adresou. Soukromý klíč se používá k prokazováním vlastnictví prostředků použitých v transakci, k vytváření podpisů požadovaných pro utrácení etheru. Soukromý klíč musí zůstat vždy v tajnosti, protože jeho odhalení třetím stranám je

rovnocenné s tím, že jim poskytuje kontrolu nad etherem a kontrakty zajištěné tímto soukromým klíčem. Soukromý klíč musí být také zálohován a chráněn před náhodnou ztrátou. Pokud dojde ke ztrátě, nelze jej získat zpět a navždy ztratí i prostředky jím zajištěné.

```
Ethereum soukromý klí¬ je jen ¬íslo. Jedním ze zp¬sob¬, jak
náhodn¬ vybrat soukromé klí¬e, je jednoduše použít minci,
tužku a papír: hodit minci 256-krát a máte binární ¬íslice
náhodného soukromého klí¬e, který m¬žete použít v pen¬žence
Ethereum (pravd¬podobn¬ - viz další ¬ást). Ve¬ejný klí¬ a
adresa pak mohou být vygenerovány ze soukromého klí¬e.
```

Tvorba soukromého klíče z náhodného čísla

Prvním a nejdůležitějším krokem při generování klíčů je nalezení bezpečného zdroje entropie nebo náhodnosti. Vytvoření Ethereum soukromého klíče v podstatě zahrnuje výběr čísla mezi 1 a 2^{256}. Přesná metoda, kterou použijete pro výběr tohoto čísla, není důležitá, pokud není předvídatelná nebo deterministická. Ethereum software používá generátor náhodných čísel podkladového operačního systému k vytvoření 256 náhodných bitů. Generátor náhodných čísel operačního systému je obvykle inicializován lidským zdrojem náhodnosti, proto můžete být vyzváni, abyste několik vteřin pohybovali myší nebo stiskli náhodné klávesy na klávesnici. Alternativou by mohl být šum kosmického záření na mikrofonním kanálu počítače.

Přesněji řečeno, soukromý klíč může být libovolné nenulové číslo až do velmi velkého čísla, o trochu menším než 2^{256} - obrovské 78-místné číslo, zhruba $1{,}158 * 10^{77}$. Přesné číslo sdílí prvních 38 číslic s 2^{256} a je definováno jako řád eliptické křivky použité v Ethereum (viz Vysvětlení kryptografie eliptických křivek). Pro vytvoření soukromého klíče náhodně vybereme 256-bitové číslo a zkontrolujeme, zda je v platném rozsahu. Z hlediska programování se toho obvykle dosáhne přidáním ještě většího řetězce náhodných bitů (shromážděných z kryptograficky bezpečného zdroje náhodnosti) do 256-bitového hašovacího algoritmu, jako je Keccak-256 nebo SHA-256, z nichž oba pohodlně vytvoří 256-bitové číslo. Pokud je výsledek v platném rozsahu, máme vhodný soukromý klíč. Jinak to zkusíme znovu s jiným náhodným číslem.

2^{256} - velikost prostoru soukromých klíčů Etherea - je nepochopitelně velké číslo. Je to přibližně 10^{77} v desítkové soustavě; to je číslo se 77 číslicemi. Pro srovnání se odhaduje, že viditelný vesmír obsahuje 10^{80} atomů. Existuje tedy téměř dost soukromých klíčů, aby každý atom ve vesmíru mohl mít vlastní Ethereum účet. Pokud vyberete soukromý klíč náhodně, není myslitelné, že by ho někdo kdy uhádl nebo si ho sám vybral.

Proces generování soukromého klíče je offline. nevyžaduje žádnou komunikaci se sítí Ethereum nebo vůbec žádnou komunikaci s kýmkoli. Aby bylo možné vybrat číslo, které nikdo jiný nikdy nezvolí, musí být skutečně náhodné. Pokud zvolíte číslo sami, je šance, že to někdo zkusí (a poté uteče s vaším etherem) příliš vysoká. Použití generátoru chybných náhodných čísel (jako je pseudonáhodná funkce rand ve většině programovacích jazyků) je ještě horší, protože je výsledné číslo ještě zřetelnější a snadněji replikovatelné. Stejně jako u hesel pro online účty musí být soukromý klíč neuhodnotelný. Naštěstí si nikdy nemusíte pamatovat svůj soukromý klíč, takže pro jeho výběr můžete použít nejlepší možný přístup: skutečnou náhodnost.

Nepište vlastní kód pro vytvoření náhodného čísla ani nepoužívejte „jednoduchý“ generátor náhodných čísel, který nabízí váš programovací jazyk. Je nezbytné, abyste používali kryptograficky bezpečný generátor pseudonáhodných čísel (například CSPRNG) se semenem ze zdroje dostatečné entropie. Prostudujte si dokumentaci vybrané knihovny generátorů náhodných čísel, abyste se ujistili, že je kryptograficky bezpečná. Správná implementace knihovny CSPRNG je rozhodující pro bezpečnost klíčů.

Následuje náhodně vygenerovaný soukromý klíč zobrazený v hexadecimálním formátu (256 bitů zobrazených jako 64 hexadecimálních číslic, každá číslice reprezentuje 4 bity):

```
f8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315
```

Veřejné klíče

Ethereum veřejný key je *bod* na eliptické křivce, což znamená, že je to sada souřadnic x a y, které splňují rovnici eliptické křivky.

Zjednodušeně řečeno, Ethereum veřejný klíč jsou dvě čísla spojená dohromady. Tato čísla jsou

vytvářena ze soukromého klíče výpočtem, který *je jednosměrný*. To znamená, že je snadné vypočítat veřejný klíč, pokud máte soukromý klíč, ale soukromý klíč z veřejného klíče nelze vypočítat.

Bude následovat MATEMATIKA! Nepanikařte. Pokud se v následujících odstavcích začnete ztrácet, můžete přeskočit několik následujících sekcí. Existuje mnoho nástrojů a knihoven, které vám pomohou s matematikou.

Veřejný klíč se počítá ze soukromého klíče pomocí násobení budu na eliptické křivce, které je prakticky nevratné: $K = k * G$, kde k je soukromý klíč, G je konstantní bod nazvaný *generátorový bod*, K je výsledný veřejný klíč a * je speciální operátor „násobení“ na eliptické křivce. Všimněte si, že násobení na eliptické křivce není jako normální násobení. Sdílí funkční atributy s normálním násobením, ale rozdíl je ve způsobu výpočtu. Například reverzní operace (která by byla dělením pro normální čísla), známá jako „nalezení diskrétního logaritmu“ - tj. výpočet k, pokud známe K - je stejně obtížná jako vyzkoušení všech možných hodnot k (hledání hrubou sílou) to bude pravděpodobně trvat déle, než to umožní tento vesmír).

Zjednodušeně řečeno: aritmetika na eliptické křivce se liší od „normální“ celočíselné aritmetiky. Bod (G) může být vynásoben celým číslem (k), čímž vznikne další bod (K). Neexistuje však nic jako *dělení*, takže není možné jednoduše „vydělit“ veřejný klíč K bodem G pro výpočet soukromého klíče k. Toto je jednosměrná matematická funkce popsaná v Kryptografie veřejného klíče a kryptoměna.

Násobení eliptické křivky je typ funkce, kterou kryptografové nazývají „jednosměrnou“ funkcí: je snadné provést v jednom směru (násobení) a nemožné provést v opačném směru (dělení). Vlastník soukromého klíče může snadno vytvořit veřejný klíč a poté jej sdílet se světem, protože ví, že nikdo nemůže vrátit funkci zpět a vypočítat soukromý klíč z veřejného klíče. Tento matematický trik se stává základem pro nepadělatelné a bezpečné digitální podpisy, které prokazují vlastnictví Ethereum finančních prostředků a kontrolu kontraktů.

Než ukážeme, jak vygenerovat veřejný klíč ze soukromého klíče, podívejme se na kryptografii eliptických křivek o něco podrobněji.

Vysvětlení kryptografie eliptických křivek

Kryptografie kryptografie eliptických křivek je typ asymetrické kryptografie neboli kryptografie veřejného klíče založená na problému diskrétního logaritmu vyjádřeného sčítáním a násobením

bodů eliptické křivky.

<<ecc-curve> > je příklad eliptické křivky, podobné křivce používané Ethereem.

Ethereum používá přesně stejnou eliptickou křivku nazvanou secp256k1 jako Bitcoin. To umožňuje opětovné použití mnoha knihoven eliptických křivek a nástrojů z Bitcoinu.

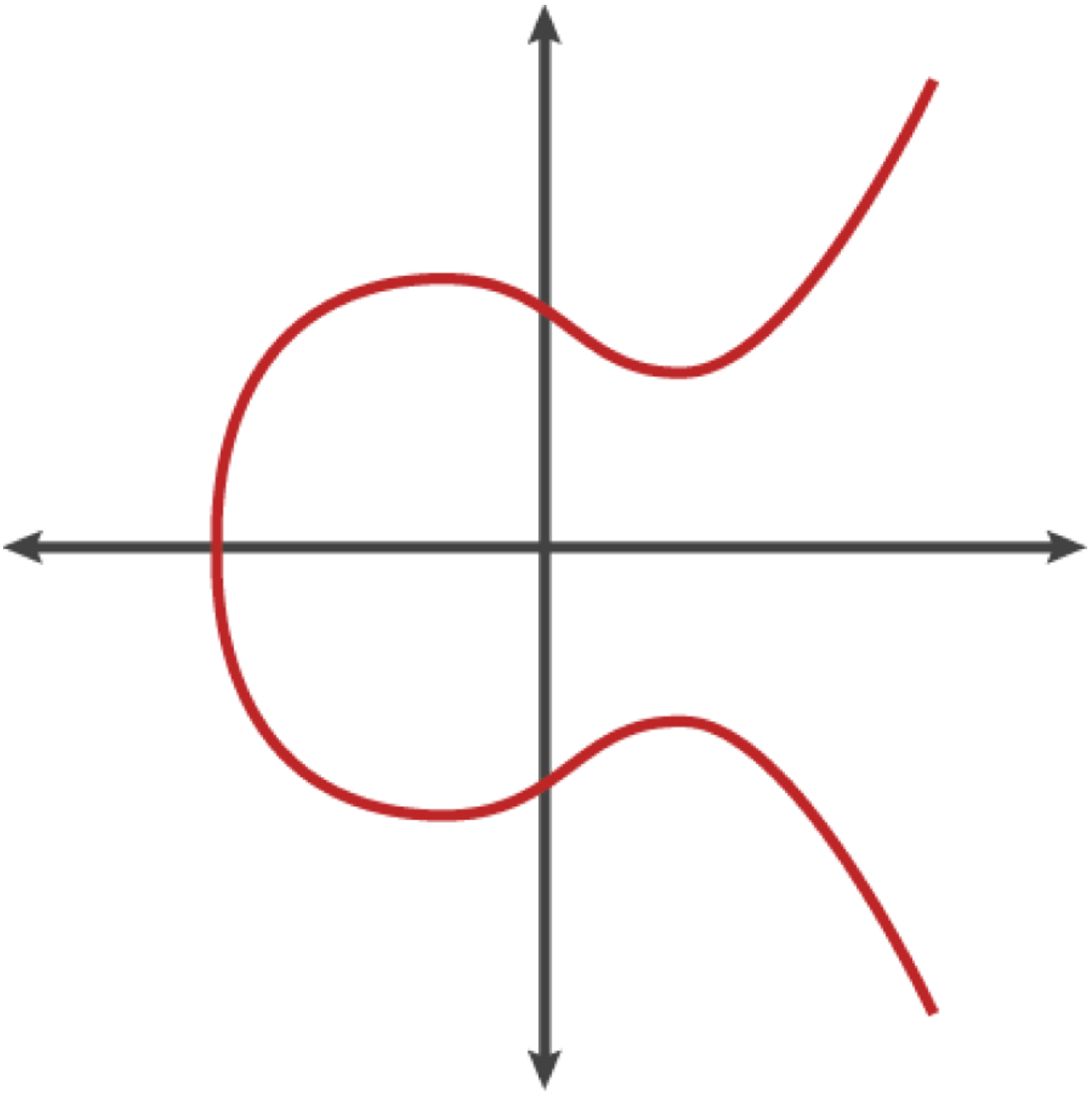

Figure 25. Zobrazení eliptické křivky

Ethereum používá specifickou eliptickou křivku a sadu matematických konstant, jak je definováno ve standardu nazvaném secp256k1, vytvořeném Americkým národním institutem pro standardy a

technologie (NIST). Křivka secp256k1 je definována následující funkcí, která vytváří eliptickou křivku:

```
<div data-type="equation">
<math xmlns="http://www.w3.org/1998/Math/MathML" display="block">
  <mrow>
    <mrow>
      <msup><mi>y</mi> <mn>2</mn> </msup>
      <mo>=</mo>
      <mrow>
        <mo>(</mo>
        <msup><mi>x</mi> <mn>3</mn> </msup>
        <mo>+</mo>
        <mn>7</mn>
        <mo>)</mo>
      </mrow>
    </mrow>
    <mspace width="3.33333pt"/>
    <mtext>nad</mtext>
    <mspace width="3.33333pt"/>
    <mrow>
      <mo>(</mo>
      <msub><mi>Z</mi> <mi>p</mi> </msub>
      <mo>)</mo>
    </mrow>
  </mrow>
</math>
</div>
```

nebo:

```
<div data-type="equation">
<math xmlns="http://www.w3.org/1998/Math/MathML" display="block">
  <mrow>
    <msup><mi>y</mi> <mn>2</mn> </msup>
    <mspace width="3.33333pt"/>
    <mo form="prefix">mod</mo>
    <mspace width="0.277778em"/>
    <mi>p</mi>
    <mo>=</mo>
    <mrow>
      <mo>(</mo>
      <msup><mi>x</mi> <mn>3</mn> </msup>
      <mo>+</mo>
      <mn>7</mn>
      <mo>)</mo>
    </mrow>
    <mspace width="3.33333pt"/>
    <mo form="prefix">mod</mo>
    <mspace width="0.277778em"/>
    <mi>p</mi>
  </mrow>
</math>
</div>
```

Operace zbytku po celočíselném dělení *mod p* (modulo prvočíslo *p*) naznačuje, že tato křivka je nad celočíselným tělesem prvočíselného řádu *p*, také označovaného jako `\( \mathbb{Z}_p \)`, kde $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$, což je velmi vysoké prvočíslo.

Protože je tato křivka definována na konečném tělese prvočíselného řádu místo na reálných číslech, vypadá to jako vzor teček rozptýlené ve dvou rozměrech, což ztěžuje vizualizaci. Matematika je však totožná s matematickou eliptickou křivkou reálných čísel. Například <<ecc-over-F17-math> > ukazuje stejnou eliptickou křivku na mnohem menším konečném tělese prvořadého řádu 17 a ukazuje vzorek teček na mřížce. Ethereum eliptickou křivku secp256k1 lze považovat za mnohem složitější vzor teček na nepochopitelně velké mřížce.

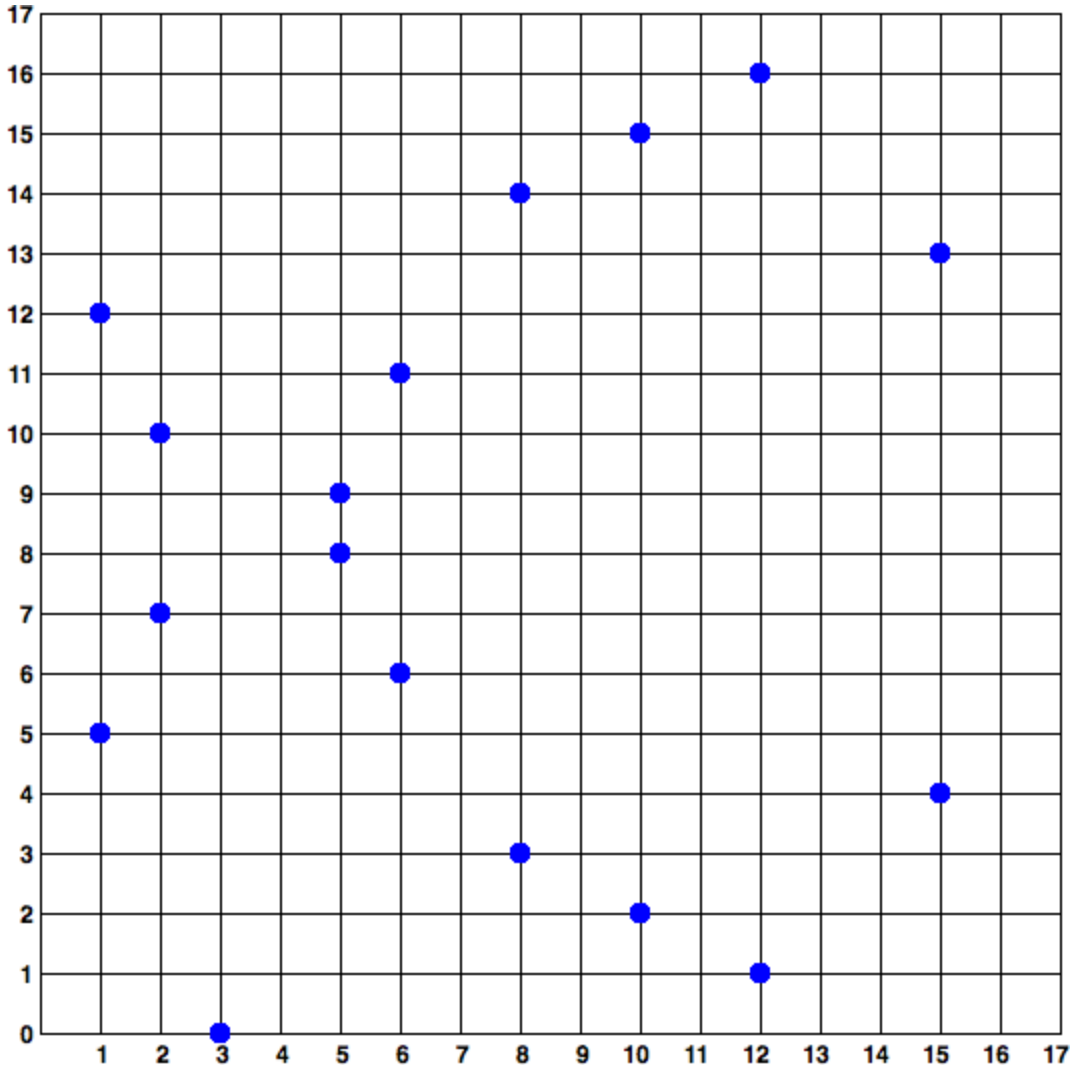

Figure 26. Kryptografie eliptických křivek: vizualizace eliptické křivky nad Z (p), kde p = 17

Například uvádíme následuje bod *Q* se souřadnicemi (*x* _, _ *y*), což je bod na křivce secp256k1:

```
Q =
(4979039082524938448603314435591686460761608352010163868140397374925592453
9515,
5957413216189990004586208649392101578003217529175580739928400772105034129 7
360)
```

[example_1] ukazuje, jak si to můžete ověřit sami pomocí Pythonu. Proměnné x a y jsou souřadnice

bodu *Q*, jako v předchozím příkladu. Proměnná p je prvočíselný řád tělesa eliptické křivky (prvočíslo, které se používá pro všechny operace modulo). Poslední řádek Pythonu je rovnice eliptické křivky (operátor % v Pythonu je operátor modulo). Pokud jsou x a y skutečně souřadnice bodu na eliptické křivce, splní rovnici a výsledek je nula (0L je dlouhé celé číslo s hodnotou nula). Vyzkoušejte to sami zadáním **python** do příkazové řádky a zkopírováním každého řádku (po výzvě >>>) ze seznamu .

```
<div data-type="example" id="example_1">
<h5> Použití Pythonu k potvrzení, že tento bod je na eliptické k¬ivce </h5>
<pre data-type="programlisting">
Python 3.4.0 (default, Mar 30 2014, 19:23:13)
[GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.38)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> <strong>p =
115792089237316195423570985008687907853269984665640564039457584007908834 \
671663</strong>
>>> <strong>x =
497903908252493844860331443559168646076160835201016386814039737492559245 39
515</strong>
>>> <strong>y =
595741321618999000458620864939210157800321752917558073992840077210503412 97
360</strong>
>>> <strong>(x ** 3 + 7 - y**2) % p</strong>
0L
</pre>
</div>
```

Aritmetické operace na eliptické křivce

Hodně matematických operací na eliptické křivce vypadá a funguje velmi podobně jako celočíselná aritmetika, kterou jsme se naučili ve škole. Konkrétně můžeme definovat operátor sčítání, který místo skoku podél číselné čáry přeskočí na další body na křivce. Jakmile máme operátor sčítání, můžeme také definovat násobení bodu a celého čísla, což je ekvivalentní opakovanému sčítání.

Sčítání na eliptické křivce je definováno tak, že vzhledem ke dvěma bodům P_1 and P_2 na eliptické křivce existuje třetí bod $P_3 = P_1 + P_2$, také na eliptické křivce.

Geometricky je tento třetí bod P_3 vypočítán nakreslením čáry mezi P_1 a P_2. Tato čára protíná

eliptickou křivku přesně na jednom dalším místě (úžasně). Nazvěte tento bod $P_3' = (x, y)$. Poté získejte jeho obraz pomocí osové symetrie dle osy x a získejte $P_3 = (x, –y)$.

Pokud jsou P_1 a P_2 stejné body, měla by se spojnice "mezi" P_1 a P_2 prodloužit tak, aby byla v tomto bodě P_1 tečná k eliptické křivce. Tato tečna protne křivku přesně v jednom novém bodě. K určení sklonu tečné čáry můžete použít běžné matematické techniky. Je zajímavé, že tyto techniky fungují, i když omezujeme náš zájem na body na křivce se dvěma celočíselnými souřadnicemi!

V matematice eliptických křivek také existuje bod nazvaný „bod v nekonečnu“, který zhruba odpovídá roli nulové hodnoty při sčítání. Na počítačích je někdy představován znakem $x = y = 0$ (který nesplňuje rovnici eliptické křivky, ale je to zvláštní případ, který jde samostatně zkontrolovat). Existuje několik zvláštních případů, které vysvětlují potřebu bodu v nekonečnu.

V některých případech (např. pokud P_1 a P_2 mají stejné hodnoty x, ale různé hodnoty y), bude čára přesně svislá, v tomto případě P_3 = bod v nekonečnu.

Jestliže P_1 je bod v nekonečnu, pak $P_1 + P_2 = P_2$. Podobně, pokud P_2 je bod v nekonečnu, pak $\_P_1 + P_2 = P_1$. To ukazuje, že bod v nekonečnu hraje stejnou roli, kterou hraje nula v „normální“ aritmetice.

Ukazuje se, že + je asociativní, což znamená, že $(A + B) + C = A + (B + C)$. To znamená, že můžeme napsat $A + B + C$ (bez závorek) aniž by tento zápis byl dvojznačný.

Nyní, když jsme definovali sčítání, můžeme definovat násobení standardním způsobem, který rozšiřuje sčítání. Pro bod P na eliptické křivce, pokud k je celé číslo, pak $k * P = P + P + P + … + P$ (k -krát). Všimněte si, že v tomto případě je k někdy nazýván „exponent“ (což je matoucí)

Vypočtení veřejného klíče

Začneme se soukromým klíčem ve tvaru náhodně vygenerovaného čísla k, vynásobíme jej předem určeným bodem na křivce zvané *generátorový bod G*, abychom vytvořili další bod někde jinde na křivce, což je odpovídající veřejný klíč K:

```
<div data-type="equation">
<math xmlns="http://www.w3.org/1998/Math/MathML" display="block">
  <mrow>
    <mi>K</mi>
    <mo>=</mo>
    <mi>k</mi>
    <mo>*</mo>
    <mi>G</mi>
  </mrow>
</math>
</div>
```

Generátorový bod je specifikován jako součást standardu secp256k1; je to stejné pro všechny implementace secp256k1 a všechny klíče odvozené z této křivky používají stejný bod *G*. Protože generátorový bod je vždy stejný pro všechny Ethereum uživatele, soukromý klíč *k* vynásobený *G* bude mít vždy vést ke stejnému veřejnému klíči *K*. Vztah mezi *k* a *K* je pevný, ale lze jej vypočítat pouze v jednom směru, od *k* ke *K*. Z tohoto důvodu lze Ethereum adresu (odvozenou z *K*) sdílet s kýmkoli a neodhaluje soukromý klíč uživatele (*k*).

Jak jsme popsali v předchozí části, násobení *k* * *G* je ekvivalentní opakovanému sčítání, takže G_ + *G* + *G* + ... + *G*, opakováno *k*-krát. Stručně řečeno, pro vytvoření veřejného klíče *K* ze soukromého klíče *k*, přičteme *k* -krát generátorový bod *G* sám k sobě.

Soukromý klíč lze převést na veřejný klíč, ale veřejný klíč nelze převést zpět na soukromý klíč, protože tato matematika funguje pouze jedním směrem.

Použijte tento výpočet a najděte veřejný klíč pro konkrétní soukromý klíč, který jsme vám ukázali v Soukromé klíče:

Příklad výpočtu veřejného klíče ze soukromého klíče

```
K = f8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315 * G
```

Kryptografická knihovna nám může pomoci vypočítat *K* pomocí násobení eliptické křivky. Výsledný veřejný klíč *K* je definován jako bod:

```
K = (x, y)
```

kde:

```
x = 6e145ccef1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b
y = 83b5c38e5e2b0c8529d7fa3f64d46daa1ece2d9ac14cab9477d042c84c32ccd0
```

V Ethereu můžete vidět veřejné klíče reprezentované jako posloupnost 130 hexadecimálních znaků (65 bajtů). Toto je převzato ze standardního formátu pro reprezentaci navrženého průmyslovým konsorciem Standards for Efficient Cryptography Group (SECG), zdokumentováno v Standards for Efficient Cryptography (SEC1) [http://www.secg.org/sec1-v2.pdf]. Standard definuje čtyři možné předpony, které lze použít k identifikaci bodů na eliptické křivce, uvedené v Předpony reprezentace veřejných klíčů eliptických křivek.

Table 2. Předpony reprezentace veřejných klíčů eliptických křivek

Předpona	**Význam**	**Délka v bytech (včetně předpony)**
0x00	Bod v nekonečnu	1
0x04	Nezkomprimovaný bod	65
0x02	Zkomprimovaný bod se sudým y	33
0x03	Komprimovaný bod s lichými y	33

Ethereum používá pouze nekomprimované veřejné klíče; proto je jedinou předponou, která je relevantní, (hex) 04. Při reprezentaci se zřetězí souřadnice *x* a *y* veřejného klíče:

```
04 + sou¬adnice x (32 byt¬ / 64 hex) + sou¬adnice y (32 byt¬ / 64 hex)
```

Veřejný klíč, který jsme vypočítali dříve, je proto reprezentován jako:

```
046e145ccef1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b83b5c38e
5e2b0 \
c8529d7fa3f64d46daa1ece2d9ac14cab9477d042c84c32ccd0
```

Knihovny eliptických křivek

Existuje několik implementací eliptické křivky secp256k1, které se používají v projektech souvisejících s kryptoměnami:

OpenSSL [https://www.openssl.org/]

Knihovna OpenSSL nabízí komplexní sadu kryptografických primitiv, včetně plné implementace secp256k1. Například pro odvození veřejného klíče lze použít funkci EC_POINT_mul.

libsecp256k1 [https://github.com/bitcoin-core/secp256k1]

Bitcoin Core libcop256k1 je implementace eliptické křivky secp256k1 a dalších kryptografických primitivů v jazyce C. Bylo psáno od nuly a nahrazuje OpenSSL v softwaru Bitcoin Core, a je považováno za lepší v jak ve výkonu tak i v bezpečnosti.

Kryptografické hashovací funkce

Kryptografické hašovací funkce se používají v celém Etherem. Ve skutečnosti jsou hašovací funkce hojně používány téměř ve všech kryptografických systémech - skutečnost, kterou zachytil kryptograf Bruce Schneier [http://bit.ly/2Q79qZp], který řekl „Mnohem více než šifrovací algoritmy jsou jednosměrné hašovací funkce tažnými koňmi moderní kryptografie.“

V této části si probereme hašovací funkce, prozkoumáme jejich základní vlastnosti a uvidíme, jak je tyto vlastnosti činí tak užitečnými v tolika oblastech moderní kryptografie. Řešíme hašovací funkce, protože jsou součástí transformace Ethereum veřejných klíčů na adresy. Mohou být také použity k vytvoření *digitálních otisků*, které pomáhají při ověřování údajů.

Jednoduše řečeno, *hašovací funkce* [http://bit.ly/2CR26gD] je „jakákoli funkce, kterou lze použít k mapování dat libovolné velikosti na data pevné velikosti. “ Vstup do hašovací funkce se nazývá *vzor*, *message* nebo jednoduše *vstup*. Výstup se nazývá *haš*. *Kryptografické hašovací funkce* [http://bit.ly/2Jrn3jM] jsou speciální podkategorií, mají specifické vlastnosti, které jsou užitečné pro zabezpečené platformy, jako je Ethereum.

Kryptografická hašovací funkce je hašovací funkce, která mapuje data libovolné velikosti na řetězec bitů s pevnou velikostí. „Jednosměrná“ povaha znamená, že je výpočetně nemožné znovu vytvořit vstupní data, pokud člověk zná pouze haš výstupu. Jediným způsobem, jak určit možný vstup, je

provést hledání hrubou sílou a zkontrolovat každého kandidáta na odpovídající výstup; vzhledem k tomu, že hledaný prostor je prakticky nekonečný, je snadné pochopit praktickou nemožnost úkolu. I když najdete některá vstupní data, která vytvářejí odpovídající haš, nemusí to být původní vstupní data: haš funkce jsou funkce „nejsou prosté“. Nalezení dvou sad vstupních dat, které jsou hašovány na stejný výstup, se nazývá nalezení *hašovací kolize*. Zhruba řečeno, čím lepší je hašovací funkce, tím vzácnější hašovací kolize jsou. Pro Ethereum jsou skutečně nemožné.

Podívejme se blíže na hlavní vlastnosti kryptografických hašovacích funkcí. Tyto zahrnují:

Determinismus

Daná vstupní zpráva vždy produkuje stejný výstup hašování.

Ověřitelnost

Výpočet haše zprávy je efektivní (lineární složitost).

Nekorelace

Malá změna zprávy (např. 1-bitová změna) by měla změnit výstup hašování tak rozsáhle, že nemůže být korelován s hašem původní zprávy.

Nezvratitelnost

Výpočet zprávy z haše je nemožný, ekvivalentní hledání brutální silou ve všech možných zprávách.

Ochrana proti kolizi

Mělo by být nemožné vypočítat dvě různé zprávy, které produkují stejný výstup hašování.

Odolnost proti hašovacím kolizím je zvláště důležitá pro to, aby se zabránilo padělání digitálního podpisu v Ethereu.

Kombinace těchto vlastností činí kryptografické hašovací funkce užitečné pro širokou škálu bezpečnostních aplikací, včetně:

- Otisky dat
- Integrita zprávy (detekce chyb)
- Důkaz prací
- Ověřování (hašování hesel a roztahování klíčů)

- Pseudonáhodné generátory čísel
- Utajený závazek (mechanismy odevzdání - odhalení)
- Jedinečné identifikátory

Mnoho z nich najdeme v Ethereu, jak postupujeme různými vrstvami systému.

Ethereum kryptografická hašovací funkce: Keccak-256

Ethereum používá kryptografickou hašovací funkci *Keccak-256* na mnoha místech. Keccak-256 byl navržen jako kandidát do soutěže o nový standard kryptografické hašovací funkce SHA-3, kterou pořádal v roce 2007 Národní vědecký a technologický institut (NIST). Keccak byl výherní algoritmus, který se v roce 2015 stal standardem označeným jako Federal Information Processing Standard (FIPS) 202.

V období, kdy bylo Ethereum vyvinuto, však standardizace NIST ještě nebyla dokončena. NIST upravil některé parametry Keccaku po dokončení standardizačního procesu, údajně ke zlepšení jeho účinnosti. K tomu došlo současně, když hrdinským oznamovatel „Snowden, Edward“ Edward Snowden odhalil dokumenty, které naznačují, že NIST mohla být Národní agenturou pro bezpečnost úmyslně oslabena. Konkrétně Generátor náhodných čísel Dual_EC_DRBG, by mohl mít umístěná zadní vrátka do standardního generátoru náhodných čísel. Výsledkem této diskuse byl odpor proti navrhovaným změnám a značné zpoždění ve standardizaci SHA-3. V té době se nadace Ethereum rozhodla implementovat původní algoritmus Keccak, jak navrhli jeho vynálezci, spíše než standard SHA-3 ve znění upraveném NIST.

I když můžete vidět, že v dokumentech a kódu Ethereum je uvedeno „SHA-3“, mnoho, pokud ne všechny tyto případy, ve skutečnosti odkazují na Keccak-256, nikoli na finalizovaný standard FIPS-202 SHA-3. Rozdíly v implementaci jsou malé, pouze v nastavené výplňkových parametrů, ale jsou významné v tom, že Keccak-256 produkuje jiné výstupy hašování než FIPS-202 SHA-3 pro stejný vstup.

Kterou hašovací funkci používám?

Jak zjistíte, zda softwarová knihovna, kterou používáte, implementuje FIPS-202 SHA-3 nebo Keccak-256, pokud se oba mohou jmenovat „SHA-3“?

Snadný způsob, jak to říct, je použít *testovací vektor*, očekávaný výstup pro daný vstup. Nejběžnějším testem pro hašovací funkci je *prázdný vstup*. Pokud spustíte hašovací funkcis prázdným řetězcem jako vstupem, měli byste vidět následující výsledky:

```
Keccak256("") =
  c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470

SHA3("") =
  a7ffc6f8bf1ed76651c14756a061d662f580ff4de43b49fa82d80a4b80f8434a
```

Bez ohledu na to, jak se tato funkce nazývá, můžete pomocí tohoto jednoduchého testu otestovat, zda se jedná o původní Keccak-256 nebo finální NIST standard FIPS-202 SHA-3. Pamatujte, že Ethereum používá Keccak-256, přestože se v kódu často nazývá SHA-3.

V důsledku zmatku vytvořeného rozdílem mezi hašovací funkcí používanou v Ethereum (Keccak-256) a finalizovaným standardem (FIP-202 SHA-3), probíhá úsilí o přejmenování všech instancí sha3 ve všech kódech, parametrech a knihovnách na keccak256. Podrobnosti viz ERC59 [https://github.com/ethereum/EIPs/issues/59].

Dále se podívejme na první aplikaci Keccak-256 v Ethereu, která má vytvářet Ethereum adresy z veřejných klíčů.

Ethereum adresy

Ethereum adresy jsou *jednoznačné* identifikátory, které jsou odvozeny z veřejných klíčů nebo kontraktů pomocí jednosměrné hašovací funkce Keccak-256.

V našich předchozích příkladech jsme začali soukromým klíčem a pro odvození veřejného klíče jsme použili násobení bodu na eliptické křivce:

Private key *k*:

```
k = f8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315
```

Veřejný klíč *K* (souřadnice *x* a *y* zřetězené a zobrazené hexadecimálně):

```
K =
6e145ccef1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b83b5c38e5e
...
```

Je třeba poznamenat, že veřejný klíč není při výpočtu adresy formátován s předponou (hex) 04.

Keccak-256 používáme pro výpočet *haše* tohoto veřejného klíče:

```
Keccak256(K) =
2a5bc342ed616b5ba5732269001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

Pak si ponecháme pouze posledních 20 bytů (nejméně významných bytů), což je naše Ethereum adresa:

```
001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

Nejčastěji uvidíte Ethereum adresy s předponou 0x, což znamená, že jsou hexadecimálně kódované, například:

```
0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

Formáty Ethereum adresy

Etereum adresy jsou hexadecimální čísla, identifikátory odvozené z posledních 20 bajtů Keccak-256 haše veřejného klíče.

Na rozdíl od Bitcoinových adres, které jsou zakódovány v uživatelském rozhraní všech klientů tak, aby obsahovaly vestavěný kontrolní součet pro ochranu před chybně zadanými adresami, jsou Ethereum adresy uváděny jako hrubé hexadecimální číslo bez kontrolního součtu.

Důvodem tohoto rozhodnutí bylo, že Ethereum adresy se mohou skrýt za abstrakcemi (jako jsou například jmenné služby) ve vyšších vrstvách systému a že kontrolní součty by se měly v případě potřeby přidat do vyšších vrstev.

Ve skutečnosti byly tyto vyšší vrstvy vyvinuty příliš pomalu a tato volba návrhu vedla v prvních dnech ekosystému k řadě problémů, včetně ztráty finančních prostředků v důsledku chybně zadaných adres a chyb při ověřování vstupu. Navíc, protože služby Ethereum byly vyvíjeny pomaleji, než se původně očekávalo, vývojáři peněženek přijali alternativní kódování velmi pomalu. Dále se podíváme na několik možností kódování.

Mezibankovní protokol klientských účtů

Mezibankovní protokol klientských účtu (Inter exchange Client Address Protocol; ICAP) je kódování Ethereum adresy, které je částečně kompatibilní s mezinárodním formátem bankovních účtů (IBAN). ICAP nabízí univerzální, kontrolním součtem chráněné a a ke spolupráci vhodné kódování Ethereum adres. Adresy ICAP mohou kódovat Ethereum adresy nebo běžná jména registrovaná v Ethereum registru názvů. Další informace o ICAP si můžete přečíst na Ethereum Wiki [http://bit.ly/2JsZHKu].

IBAN je mezinárodní standard pro identifikaci čísel bankovních účtů, většinou používaných pro bankovní převody. Je široce přijímán v Jednotném evropském platebním prostoru (SEPA) i mimo něj. IBAN je centralizovaná a silně regulovaná služba. ICAP je decentralizovaná, ale kompatibilní implementace pro Ethereum adresy.

Číslo IBAN se skládá z řetězce až 34 alfanumerických znaků (nerozlišují se malá a velká písmena), která obsahují kód země, kontrolní součet a identifikátor bankovního účtu (který je specifický pro danou zemi).

ICAP používá stejnou strukturu zavedením nestandardního kódu země, „XE“, což je zkratka pro „Ethereum“, následovaný dvoumístným kontrolním součtem a třemi možnými variantami identifikátoru účtu:

Přímé (Direct)

> Celé číslo v big-endian base-36 formátu složené z až 30 alfanumerických znaků představujících 155 nejméně významných bitů Ethereum adresy. Protože toto kódování obsahuje méně než celých 160 bitů obecné adresy Ethereum, funguje pouze pro Ethereum adresy, které začínají jedním nebo více nulovými bajty. Výhodou je, že je kompatibilní s IBAN, pokud jde o délku pole a kontrolní součet. Příklad: + XE60HAMICDXSV5QXVJA7TJW47Q9CHWKJD + (33 znaků).

Základní (Basic)

Stejné jako přímé kódování, s tím rozdílem, že má 31 znaků. To mu umožňuje zakódovat jakoukoli adresu Ethereum, ale učiní ji nekompatibilní s ověřením pole IBAN. Příklad: + XE18CHDJBPLTBCJ03FE9O2NS0BPOJVQCU2P + (dlouhý 35 znaků).

Nepřímé (Indirect)

Kóduje identifikátor, který se překládá na adresu Ethereum prostřednictvím poskytovatele registru jmen. Používá 16 alfanumerických znaků, které obsahují identifikátor *aktivum* (např. ETH), jmennou službu (např. XREG) a devítimístný lidsky čitelný název (např. KITTYCATS). Příklad: +XE##ETHXREGKITTYCATS (dlouhý 20 znaků), kde ## by měl být nahrazen dvěma vypočítanými znaky kontrolního součtu.

K vytvoření ICAP adres můžeme použít nástroj příkazového řádku helpeth. Můžete ho nainstalovat následujícím příkazem:

```
<pre data-type="programlisting">
$ <strong>npm install -g helpeth</strong>
</pre>
```

Pokud npm nemáte, možná budete muset nejprve nainstalovat nodeJS, což můžete provést podle pokynů na https://nodeJS.org.

Nyní, když máme helpeth, zkusme vytvořit ICAP adresu pomocí našeho příkladu soukromého klíče (s předponou 0x a předaného jako parametr helpeth).

```
<pre data-type="programlisting">
$ <strong>helpeth keyDetails \
  -p
0xf8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315</strong>

Address: 0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9
ICAP: XE60 HAMI CDXS V5QX VJA7 TJW4 7Q9C HWKJ D
Public key:
0x6e145ccef1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b...
</pre>
```

Příkaz helpeth pro nás vytvoří hexadecimální Ethereum adresu a ICAP adresu. Adresa ICAP pro náš

příklad je:

```
XE60HAMICDXSV5QXVJA7TJW47Q9CHWKJD
```

Protože náš příklad Ethereum adresy začíná nulovým bajtem, může být kódován pomocí metody přímého kódování ICAP, která je platným formátem IBAN. Můžete to říct, protože je dlouhý 33 znaků.

Pokud by naše adresa nezačala nulou, byla by zakódována pomocí základního kódování, které by mělo délku 35 znaků a bylo by neplatné jako IBAN.

Šance na jakoukoli Ethereum adresu začínající nulovým bajtem jsou 1 z 256. Vygenerování jedné takové bude trvat průměrně 256 pokusů, vygenerujeme 256 různých náhodných soukromých klíčů, než najdeme ten, který lze použít jako „přímé“ IBAN kompatibilní kódování ICAP Adresy.

V současné době je ICAP bohužel podporováno pouze několika peněženkami.

Hexadecimální kódování s kontrolním součtem pomocí velikostí písmen (EIP-55)

Vzhledem k pomalému nasazení ICAP a jmenných služeb byl navržen standard https://github.com/Ethereum/EIPs/blob/master/EIPS/eip-55.md [Návrh na vylepšení Etherea 55 (EIP-55)]. EIP-55 nabízí zpětně kompatibilní kontrolní součet pro Ethereum adresy úpravou velikosti písmen hexadecimální adresy. Myšlenka je taková, že Ethereum adresy nerozlišují velká a malá písmena a všechny peněženky by měly přijímat Ethereum adresy vyjádřené velkými nebo malými písmeny bez jakéhokoli rozdílu v interpretaci.

Modifikováním velikosti abecedních znaků v adrese můžeme přidat informaci kontrolním součtu, který lze použít k ochraně integrity adresy před chybami při psaní nebo čtení. Peněženky, které nepodporují kontrolní součty dle EIP-55, jednoduše ignorují skutečnost, že adresa obsahuje smíšení malých a velkých písmen, ale ty, které ji podporují, ji mohou ověřit a detekovat chyby s přesností 99,986%.

Kódování pomocí malých a velkých písmen je jemné a nemusíte si ho nejprve všimnout. Adresa z našeho příkladu je:

```
0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

S kontrolním součtem pomocí velikostí písmen EIP-55 se stává:

```
0x001d3F1ef827552Ae1114027BD3ECF1f086bA0F9
```

Můžete rozeznat rozdíl? Některé z písmenných (A – F) znaků z hexadecimální kódovací abecedy jsou nyní velké, zatímco jiné malé.

Implementace EIP-55 je poměrně jednoduchá. Vezmeme Keccak-256 haš hexadecimální adresy, která má všechna písmena malá. Tento haš funguje jako digitální otisk adresy, což nám poskytuje vhodný kontrolní součet. Jakákoli malá změna ve vstupu (adresa) by měla způsobit velkou změnu ve výsledném haši (kontrolní součet), což nám umožní efektivně detekovat chyby. Haš naší adresy je pak zakódován ve velikosti písmen samotné adresy. Pojďme to rozebrat, krok za krokem:

1. Spočítejte haš adresy zapsané malými písmeny bez předpony 0x:

```
Keccak256("001d3f1ef827552ae1114027bd3ecf1f086ba0f9") =
23a69c1653e4ebbb619b0b2cb8a9bad49892a8b9695d9a19d8f673ca991deae1
```

2. Písmeno v adrese změníme z malého na velké, pokud je odpovídající hexadecimální číslo haše větší nebo rovno 0x8. Toto je snazší ukázat, pokud napíšeme jednotlivé znaky adresu a a jejího haše přímo pod sebe:

```
Addresa: 001d3f1ef827552ae1114027bd3ecf1f086ba0f9
Haš    : 23a69c1653e4ebbb619b0b2cb8a9bad49892a8b9...
```

Naše adresa obsahuje na čtvrté pozici písmeno d. Čtvrtý znak haše je 6, což je méně než 8. Takže ponecháme malé písmeno d. Dalším písmenem v naší adrese je f, na šesté pozici. Šestý znak hexadecimálního haše je c, což je více než 8. Proto změníme písmeno na velké F v adrese atd. Jak vidíte, jako kontrolního součtu používáme pouze prvních 20 bajtů (40 hex znaků) haše, protože máme pouze 20 bajtů (40 hex znaků) v adrese, abychom správně použili velká písmena.

Zkontrolujte si výslednou adresu s velkými a velkými písmeny a zjistěte, zda můžete zjistit, které znaky byly změněny na velká písmena a kterým znakům odpovídají v haši adresy:

```
Addresa: 001d3F1ef827552Ae1114027BD3ECF1f086bA0F9
Haš    : 23a69c1653e4ebbb619b0b2cb8a9bad49892a8b9...
```

Zjištění chyby v adrese zakódované EIP-55

Nyní se podívejme, jak nám adresy EIP-55 pomohou najít chybu. Předpokládejme, že jsme vytiskli Ethereum adresu, která je kódována EIP-55:

```
0x001d3F1ef827552Ae1114027BD3ECF1f086bA0F9
```

Nyní udělejme základní chybu při čtení této adresy. Předposlední znak je velké F. V tomto příkladu předpokládejme, že jsme si to špatně přečetli jako Velké E a do naší peněženky zadáme následující (nesprávnou) adresu:

```
0x001d3F1ef827552Ae1114027BD3ECF1f086bA0E9
```

Naštěstí je naše peněženka kompatibilní s EIP-55! Všimne si smíšeného použití malých a velkých písmen a pokusí se ověřit adresu. Převede jej na malá písmena a vypočítá její haš, získá kontrolní součet:

```
Keccak256("001d3f1ef827552ae1114027bd3ecf1f086ba0e9") =
5429b5d9460122fb4b11af9cb88b7bb76d8928862e0a57d46dd18dd8e08a6927
```

Jak vidíte, i když se adresa změnila pouze o jeden znak (ve skutečnosti pouze jeden bit, protože e a f jsou jeden bit od sebe), haš adresy se radikálně změnil. To je vlastnost hašovacích funkcí, díky které jsou tak užitečné pro kontrolní součty!

Nyní si napište pod sebe adresu a její haš a zkontrolujte velikost písmen v adrese:

```
001d3F1ef827552Ae1114027BD3ECF1f086bA0E9
5429b5d9460122fb4b11af9cb88b7bb76d892886...
```

Všechno je špatně! Několik písmen v adrese má nesprávnou velikost. Nezapomeňte, že velikost písmen je kódování *správného* kontrolního součtu.

Velikost písmen zadané adresy neodpovídá právě vypočítanému kontrolnímu součtu, což znamená, že se v adrese něco změnilo a byla předána chybně. .

Závěry

V této kapitole jsme provedli krátký přehled kryptografie veřejných klíčů a zaměřili se na použití veřejných a soukromých klíčů v Ethereu a použití kryptografických nástrojů, jako jsou hašovací funkce, při vytváření a ověřování Ethereum adres. Také jsme se podívali na digitální podpisy a na to, jak mohou prokázat vlastnictví soukromého klíče, aniž by tento soukromý klíč odhalili. V <<wallets_chapter> > spojíme tyto nápady a podíváme se, jak lze peněženky použít ke správě sbírek klíčů

Peněženky

Slovo „peněženka“ se v Ethereu používá k popisu několika různých věcí.

Na vysoké úrovni je peněženka softwarová aplikace, která slouží jako primární uživatelské rozhraní pro Ethereum. Peněženka řídí přístup k penězům uživatele, správu klíčů a adres, sledování zůstatku a vytváření a podepisování transakcí. Některé Ethereum peněženky mohou také spolupracovat s kontrakty, jako jsou ERC20 tokeny.

Přesněji řečeno z pohledu programátora slovo *peněženka* označuje systém používaný k ukládání a správě klíčů uživatele. Každá peněženka má komponentu správy klíčů. U některých peněženek je to všechno. Ostatní peněženky jsou součástí mnohem širší kategorie, kategorie *prohlížečů*, což jsou rozhraní pro decentralizované aplikace založené na Ethereu, nebo *DApps*, které podrobněji prozkoumáme v <<decentralized_applications_chap> >. Neexistují žádné jasné dané hranice pro rozlišení mezi různými kategoriemi, které patří pod termínem peněženka.

V této kapitole se podíváme na peněženky jako schránky pro soukromé klíče a jako systémy pro správu těchto klíčů.

Přehled technologií peněženky

V této části shrnujeme různé technologie používané k vytváření uživatelsky příjemných, bezpečných a flexibilních Ethereum peněženek.

Jedním z klíčových aspektů při navrhování peněženek je vyvážení pohodlí a soukromí. Nejpohodlnější Ethereum peněženka je ta s jediným soukromým klíčem a adresou, kterou znovu použijete pro všechno. Bohužel takové řešení je noční můrou pro soukromí, protože kdokoli může snadno sledovat a korelovat všechny vaše transakce. Použití nového klíče pro každou transakci je nejlepší z hlediska ochrany osobních údajů, ale je velmi obtížné jej spravovat. Správného vyvážení je obtížné dosáhnout, ale proto je prvořadý dobrý návrh peněženky.

Běžná mylná představa o Ethereu je taková, že Ethereum peněženky obsahují ether nebo tokeny. Ve skutečnosti, velmi přísně řečeno, peněženka drží pouze klíče. Ether nebo jiné tokeny jsou zaznamenány na Ethereum bločence. Uživatelé ovládají své tokeny v síti podpisem transakcí pomocí klíčů v jejich peněženkách. IV jistém smyslu je Ethereum peněženka *klíčenkou*. Vzhledem k tomu, že klíče, které drží peněženka, jsou jedinými věcmi, které jsou zapotřebí k přenosu etheru nebo tokenů

ve prospěch jiné adresy, je v praxi toto rozlišení dosti bezvýznamné. Tam, kde záleží na rozdílu, je změna myšlení od řešení centralizovaného systému konvenčního bankovnictví (kde jen vy a banka můžete vidět peníze na svém účtu a k provedení transakce potřebujete pouze přesvědčit banku, že chcete přesunout finanční prostředky) k decentralizovanému systému bločenkových platforem (kde každý může vidět etherový zůstatek na účtu, i když pravděpodobně nezná majitele účtu a majitel účtu musí přesvědčit každého, že chce pomocí transakce převést své prostředky). V praxi to znamená, že existuje nezávislý způsob, jak zkontrolovat zůstatek na účtu, aniž byste potřebovali peněženku. Kromě toho můžete přesunout správu účtu z aktuální peněženky do jiné peněženky, pokud se vám nelíbí peněženková aplikace, kterou jste začali používat.

Ethereum peněženky obsahují klíče, ne ether nebo tokeny. Peněženky jsou jako klíčenky obsahující páry soukromých a veřejných klíčů. Uživatelé podepisují transakce pomocí soukromých klíčů, čímž prokazují, že vlastní éter. Ether je uložen na bločence.

Existují dva hlavní typy peněženek, které se rozlišují podle toho, zda klíče, které obsahují, spolu souvisí nebo ne.

Prvním typem je *nedeterministická* peněženka, u které je každý klíč nezávisle generován z jiného náhodného čísla . Klíče spolu nesouvisejí. Tento typ peněženky je známý také jako peněženka JBOK, z fráze „pouze hromada klíčů“ (Just a Bunch of Keys).

Druhým typem peněženky je *deterministická peněženka*, kde jsou všechny klíče odvozeny od jednoho hlavního klíč, označovaného jako *semínko*. Všechny klíče v tomto typu peněženky jsou vzájemně propojeny a mohou být vygenerovány znovu z původního semínka. Existuje řada různých metod odvození klíčů používaných v deterministických peněženkách. Nejčastěji používaná odvozovací metoda používá stromovou strukturu, jak je popsáno Hierarchické deterministické peněženky (BIP-32 / BIP-44).

Chcete-li učinit deterministické peněženky o něco bezpečnější proti nehodám při ztrátě dat, jako je například odcizení telefonu nebo jeho pád do záchodu, semínka jsou často kódována jako seznam slov (v angličtině, češtině nebo jiném jazyce), abyste si je mohli zapsat a použít v případě nehody. Používá se označení *mnemotechnická kódová slova*. Samozřejmě, pokud někdo získá vaše mnemotechnická kódová slova, pak může také znovu vytvořit vaši peněženku a získat tak přístup k vašemu etheru a chytrým kontraktům. Proto buďte velmi opatrní se seznamem slov pro obnovení! Nikdy jej neukládejte elektronicky, do souboru, do počítače nebo telefonu. Zapište si jej na papír a uložte na bezpečné místo.

Několik následujících oddílů představuje každou z těchto technologií na vysoké úrovni.

Nedeterministické (náhodné) peněženky

V první Ethereum peněžence (vyrobené pro předprodej Etherea) každý soubor peněženky ukládal jeden náhodně vygenerovaný soukromý klíč. Tyto peněženky jsou nahrazeny deterministickými peněženkami, protože tyto „staré“ peněženky jsou v mnoha ohledech podřadné. Například se považuje za osvědčený postup vyhnout se opětovnému použití Ethereum adresy v rámci maximalizace vašeho soukromí při používání Etherea - tj. Použít novou adresu (která potřebuje nový soukromý klíč) pokaždé, když obdržíte finanční prostředky. Můžete jít dále a použít novou adresu pro každou transakci, i když to může být dražší, pokud hodně nakládáte s tokeny. Aby bylo možné postupovat podle této praxe, bude muset nedeterministická peněženka pravidelně rozšiřovat seznam svých klíčů, což znamená, že budete muset pravidelně zálohovat. Pokud někdy ztratíte svá data (selhání disku, nehoda s rozlitím pití, odcizení telefonu) dříve, než se vám podaří zálohovat vaši peněženku, ztratíte přístup ke svým prostředkům a chytrým kontraktům. Nedeterministické peněženky typu 0 jsou nejsložitější, protože vytvářejí nový soubor peněženky pro každou novou adresu v okamžiku jejího prvního použití.

Mnoho Ethereum klientů (včetně geth) však používá soubor *uložiště klíčů* (keystore), což je soubor kódovaný JSON, který obsahuje jediný (náhodně vygenerovaný) soukromý klíč, šifrovaný přístupovou frází pro zvýšení bezpečnosti. Obsah souboru JSON vypadá takto:

```
{
    "address": "001d3f1ef827552ae1114027bd3ecf1f086ba0f9",
    "crypto": {
        "cipher": "aes-128-ctr",
        "ciphertext":

"233a9f4d236ed0c13394b504b6da5df02587c8bf1ad8946f6f2b58f055507ece",
        "cipherparams": {
            "iv": "d10c6ec5bae81b6cb9144de81037fa15"
        },
        "kdf": "scrypt",
        "kdfparams": {
            "dklen": 32,
            "n": 262144,
            "p": 1,
            "r": 8,
            "salt":

"99d37a47c7c9429c66976f643f386a61b78b97f3246adca89abe4245d2788407"
        },
        "mac":
"594c8df1c8ee0ded8255a50caf07e8c12061fd859f4b7c76ab704b17c957e842"
    },
    "id": "4fcb2ba4-ccdb-424f-89d5-26cce304bf9c",
    "version": 3
}
```

Formát úložiště klíčů používá funkci odvození klíčů _ (KDF), známou také jako algoritmus protahování hesel, který chrání před útokem hrubou chybou, slovníkovým útokem a útokem duhovou tabulkou. Zjednodušeně řečeno, soukromý klíč není šifrován přístupovou frází přímo. Místo toho je přístupová fráze *natažena*, opakovaným hašováním. Hašovací funkce se opakuje pro 262 144 kol, což lze vidět v úložišti klíčů JSON jako parametr crypto.kdfparams.n. Útočník, který se pokouší proniknout hrubou silou, by musel použít 262 144 cyklů hašování pro každý pokus uhodnutí přístupové fráze, což útok dostatečně zpomalí, aby ho učinilo neproveditelným pro přístupové fráze dostatečně složitosti a délky.

Existuje celá řada softwarových knihoven, které umí číst a zapisovat formát úložiště klíčů, například JavaScript knihovna keythereum [https://github.com/ethereumjs/keythereum].

Použití nedeterministických peněženek se nedoporučuje pro nic jiného než pro jednoduché testy. Jsou příliš těžkopádné, než aby jste je mohli zálohovat a používat pro cokoli kromě nejzákladnějších situací. Místo toho použijte průmyslový standard založený na hiearchické deterministické (HD) peněžence s mnemotechnickým semínkem pro zálohování.

Deterministické (semínkové) peněženky

Deterministické nebo „semínkové“ peněženky jsou peněženky, které obsahují soukromé klíče, které jsou všechny odvozeny od jednoho hlavního klíče neboli semínka. Semínko je náhodně generované číslo, které je kombinováno s dalšími daty, jako je indexové číslo nebo „kód řetězu“ (Rozšířené veřejné a soukromé klíče) pro odvození libovolného počtu soukromých klíčů. V deterministické peněžence je semínko dostatečné k získání všech odvozených klíčů, a proto jediná záloha v době jejího vytvoření postačuje k zálohování všech prostředků a chytrých kontraktů v peněžence. Semínko je také dostatečné pro export nebo import peněženky, což umožňuje snadnou migraci všech klíčů mezi různými implementacemi peněženky.

Díky tomuto návrhu je bezpečnost semínka nanejvýš důležitá, protože k získání přístupu k celé peněžence je potřeba pouze semínko. Na druhou stranu, schopnost soustředit bezpečnostní úsilí na jeden kus dat lze považovat za výhodu.

Hierarchické deterministické peněženky (BIP-32 / BIP-44)

Deterministické peněženky byly vyvinuty, aby bylo snadné odvodit mnoho klíčů z jednoho semínka. V současné době je nejpokročilejší formou deterministické peněženky *hierarchická deterministická* (HD) peněženka definovaná v Bitcoinovém *BIP-32 standardu* [http://bit.ly/2B2vQWs]. HD peněženky obsahují klíče odvozené ve stromové struktuře, takže nadřazený klíč může odvodit posloupnost podřízených klíčů, z nichž každý může odvodit posloupnost vnoučat klíčů atd. Tato stromová struktura je znázorněna v HD peněženka: strom klíčů generovaných z jednoho semínka.

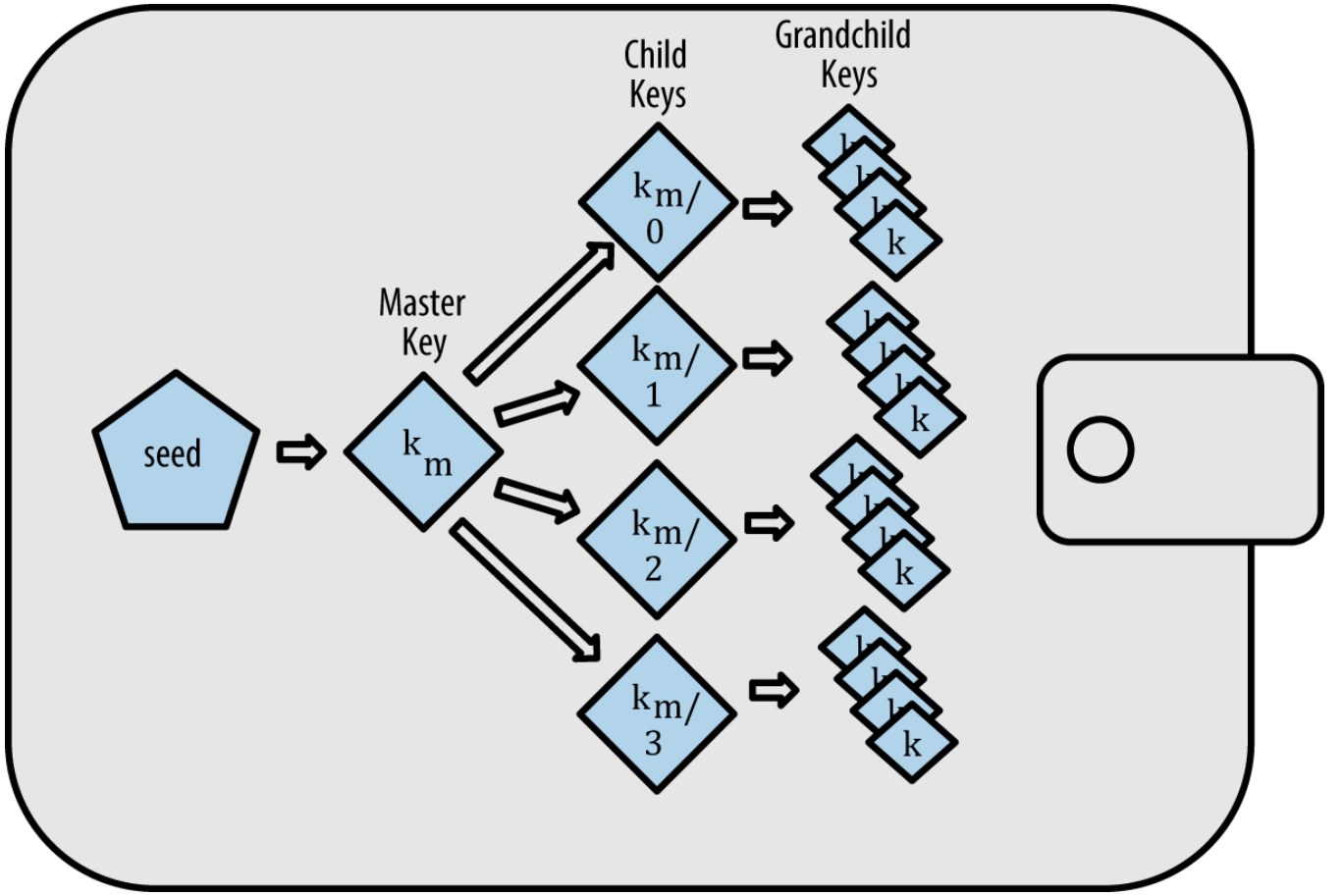

Figure 27. HD peněženka: strom klíčů generovaných z jednoho semínka

HD peněženky nabízejí několik klíčových výhod oproti jednodušším deterministickým peněženkám. Za prvé, stromová struktura může být použita k vyjádření dalšího organizačního schématu, například když je určitá větev podklíčů použita pro příjem příchozích plateb a jiná větev se používá pro příjem vratek odchozích plateb. Větve klíčů lze také použít v podnikovém prostředí, přiřadit různé větve pobočkám, dceřiným společnostem, konkrétním funkcím nebo účetním kategoriím.

Druhou výhodou HD peněženek je to, že uživatelé mohou vytvářet posloupnost veřejných klíčů, aniž by měli přístup k odpovídajícím soukromým klíčům. To umožňuje použití HD peněženek na nezabezpečeném serveru nebo v režimu pouze pro sledování nebo příjem, kde peněženka nemá soukromé klíče, které mohou utratit finanční prostředky.

Semínka a mnemotechnické kódy (BIP-39)

Existuje mnoho způsobů, jak zakódovat soukromý klíč pro bezpečné zálohování a uchování. Aktuálně upřednostňovanou metodou je použití posloupnosti slov, která, pokud jsou použita dohromady ve správném pořadí, můžou jedinečně znovu vytvořit soukromý klíč. Toto je někdy známé jako *mnemotechnický* a přístup byl standardizován BIP-39 [http://bit.ly/2OEMjUz]. Dnes mnoho Ethereum peněženek (stejně jako peněženky pro jiné kryptoměny) používá tento standard a může importovat a exportovat semínka pro zálohování a obnovu pomocí vzájemné spolupráce schopné mnemotechniky.

Abychom zjistili, proč se tento přístup stal populárním, podívejme se na příklad:

Semínko pro deterministickou peněženku, hexadecimálně

```
FCCF1AB3329FD5DA3DA9577511F8F137
```

Semínko pro deterministickou peněženku, 12 mnemotechnických slov

```
wolf juice proud gown wool unfair
wall cliff insect more detail hub
```

Z praktického hlediska je pravděpodobnost chyby při zápisu hexadecimální sekvence nepřijatelně vysoká. Naproti tomu se seznamem známých slov lze poměrně snadno zacházet, hlavně proto, že při psaní slov je vysoká úroveň nadbytečnosti (zejména u anglických slov). Pokud byl „inzect“ zaznamenán náhodou, bylo by možné rychle zjistit, v případě potřeby obnovy peněženky, že „inzect“ není platné anglické slovo a že místo toho by měl být použit „insect“. Hovoříme o zápisu reprezentace seminka, protože to je dobrá praxe při správě HD peněženek: semínko je potřeba k obnovení peněženky v případě ztráty dat (ať už kvůli nehodě nebo krádeži), takže udržování zálohy je velmi obezřetné. Semínko však musí být uchováváno velmi soukromě, proto byste měli pečlivě vyhýbat digitálním zálohám; proto opakujeme dříve zmíněnou rada, zálohujte perem a papírem.

Stručně řečeno, použití seznamu obnovovacích slov pro zakódování semínka pro peněženku HD umožňuje nejjednodušší způsob, jak bezpečně exportovat, přepisovat, zaznamenávat na papír, číst bez chyby a importovat množinu soukromých klíčů do jiné peněženky.

Doporučené postupy pro práci s peněženkou

Vzhledem k tomu, že technologie kryptoměnových peněženek dozrála, objevily se určité společné průmyslové standardy, díky nimž jsou peněženky schopné vzájemné spolupráce a dají se snadno používat a zabezpečit a pružně reagují. Tyto standardy také umožňují peněženkám odvodit klíče pro více různých kryptoměn, vše z jediné mnemotechnické pomůcky. Jedná se o tyto společné normy:

- Mnemonická kódová slova, založená na BIP-39, včetně české sady slov
- HD peněženky, založené na BIP-32
- Víceúčelová struktura HD peněženky, založená na BIP-43
- Peněženky s více měnami a více účty, založené na BIP-44

Tyto standardy se mohou v budoucnu změnit nebo být zastaralé, ale prozatím tvoří sadu vzájemně propojených technologií, které se staly *faktickým* standardem peněženky pro většinu bločenkových platforem a jejich kryptoměny.

Tyto standardy byly přijaty širokou škálou softwarových a hardwarových peněženek, díky čemuž jsou všechny tyto peněženky schopné vzájemné spolupráce. Uživatel může exportovat mnemotechnikou zálohu vytvořenou v jedné z těchto peněženek a importovat ji do jiné peněženky a obnovit všechny klíče a adresy.

Některé příklady softwarových peněženek podporujících tyto standardy zahrnují (jsou uvedeny abecedně) Jaxx, MetaMask, MyCrypto a MyEtherWallet (MEW). Mezi příklady hardwarových peněženek podporujících tyto standardy patří Keepkey, Ledger a Trezor.

Následující sekce podrobně zkoumají každou z těchto technologií.

Pokud implementujete Ethereum peněženku, měla by být postavena jako HD peněženka, se semínkem kódovaným jako mnemotechnický kód pro zálohování podle standardů BIP-32, BIP-39, BIP-43 a BIP-44, jak je popsáno v následujících sekcích.

Mnemotechnická kódová slova (BIP-39)

Mnemotechnická kódová slova je posloupnost slov, která kódují náhodné číslo používané jako

semínko k odvození deterministické peněženky. Posloupnost slov je dostatečná pro opětovné vytvoření semínka a odtud znovu pro vytvoření peněženky a všech odvozených klíčů. Aplikace peněženky, která implementuje deterministické peněženky s mnemotechnickými slovy, ukáže uživateli sekvenci 12 až 24 slov při prvním vytvoření peněženky. Tato posloupnost slov je záloha peněženky a lze ji použít k obnovení a opětovnému vytvoření všech klíčů ve stejné nebo v jakékoli kompatibilní aplikaci peněženky. Jak jsme vysvětlili dříve, seznamy mnemotechnických slov usnadňují uživatelům zálohování peněženek, protože jsou snadno čitelné a správně přepisovatelné.

Mnemonická slova jsou často zaměňována s „mozkovými peněženkami“ (brainwallets). Nejsou stejné. Primární rozdíl spočívá v tom, že mozková peněženka se skládá ze slov zvolených uživatelem, zatímco mnemotechnická slova jsou peněženkou vytvářena náhodně a prezentována uživateli. Tento důležitý rozdíl dělá mnemotechnická slova mnohem bezpečnější, protože lidé jsou velmi špatnými zdroji náhodnosti. Možná ještě důležitější je, že použití termínu „mozková peněženka“ naznačuje, že slova musí být zapamatována, což je hrozný nápad, a recept na to, že nemáte zálohu, když ji potřebujete.

Mnemotechnické kódy jsou definovány v BIP-39. Všimněte si, že BIP-39 je jedna implementace standardu mnemotechnických kódů. Existuje odlišná norma - s jinou sadou slov - používaná peněženkou Electrum Bitcoin a předcházející BIP-39. BIP-39 byl navržen společností stojící za hardwarovou peněženkou Trezor a je nekompatibilní s implementací společnosti Electrum. BIP-39 však nyní dosáhl široké podpory v desítkách vzájemné spolupráce schopných implementací a měl by být považován za průmyslový "faktický" standard. Kromě toho lze BIP-39 použít k výrobě víceměnových peněženek podporujících Ethereum, zatímco Electrum semínka to nemohou.

BIP-39 definuje vytvoření mnemotechnického kódu a semínka, které zde popisujeme v devíti krocích. Pro přehlednost je proces rozdělen do dvou částí: kroky 1 až 6 jsou uvedeny v Generování mnemotechnických slov a kroky 7 až 9 jsou uvedeny v Převod mnemotechnického kódu na semínko.

Generování mnemotechnických slov

Mnemotechnická slova jsou generována automaticky peněženkou pomocí standardizovaného procesu definovaného v BIP-39. Peněženka začíná od zdroje entropie, přidá kontrolní součet a poté entropii převede na slova ze slovníku:

1. Vytvoří kryptograficky náhodnou posloupnost bitů S o velikosti od 128 do 256 bitů.

2. Vytvoří kontrolní součet S tak, že spočte SHA-256 haš S a z něho vezme první bity, konkrétně délka S / 32 bitů.
3. Přidá kontrolní součet na konec náhodné posloupností bitů S.
4. Rozdělí posloupnost bitů vzniklou v předchozím kroku na části o velikosti 11 bitů.
5. Každá 11-bitová posloupnost je pořadovým číslem slova v předem definovaném slovníku obsahujícím 2 048 slov. V současné době existuje 9 slovníků, včetně angličtiny a češtiny.
6. Výsledný mnemotechnický kód tvoří posloupnost slov vybraných v předchozím kroku ze slovníku. Je nutné zachovat pořadí slov, ve kterém byly ze slovníky vybírány.

Generování entropie a její kódování do posloupnosti mnemotechnických slov ukazuje, jak se entropie používá ke generování mnemotechnických slov.

Mnemotechnické kódy: entropie a délka slova ukazuje vztah mezi velikostí entropických dat a délkou mnemotechnických kódů měřenou počtem slov.

Table 3. Mnemotechnické kódy: entropie a délka slova

Entropie (bity)	**Kontrolní součet (bity)**	**Entropie + kontrolní součet (bity)**	**Mnemotechnická délka (slova)**
128	4	132	12
160	5	165	15
192	6	198	18
224	7	231	21
256	8	264	24

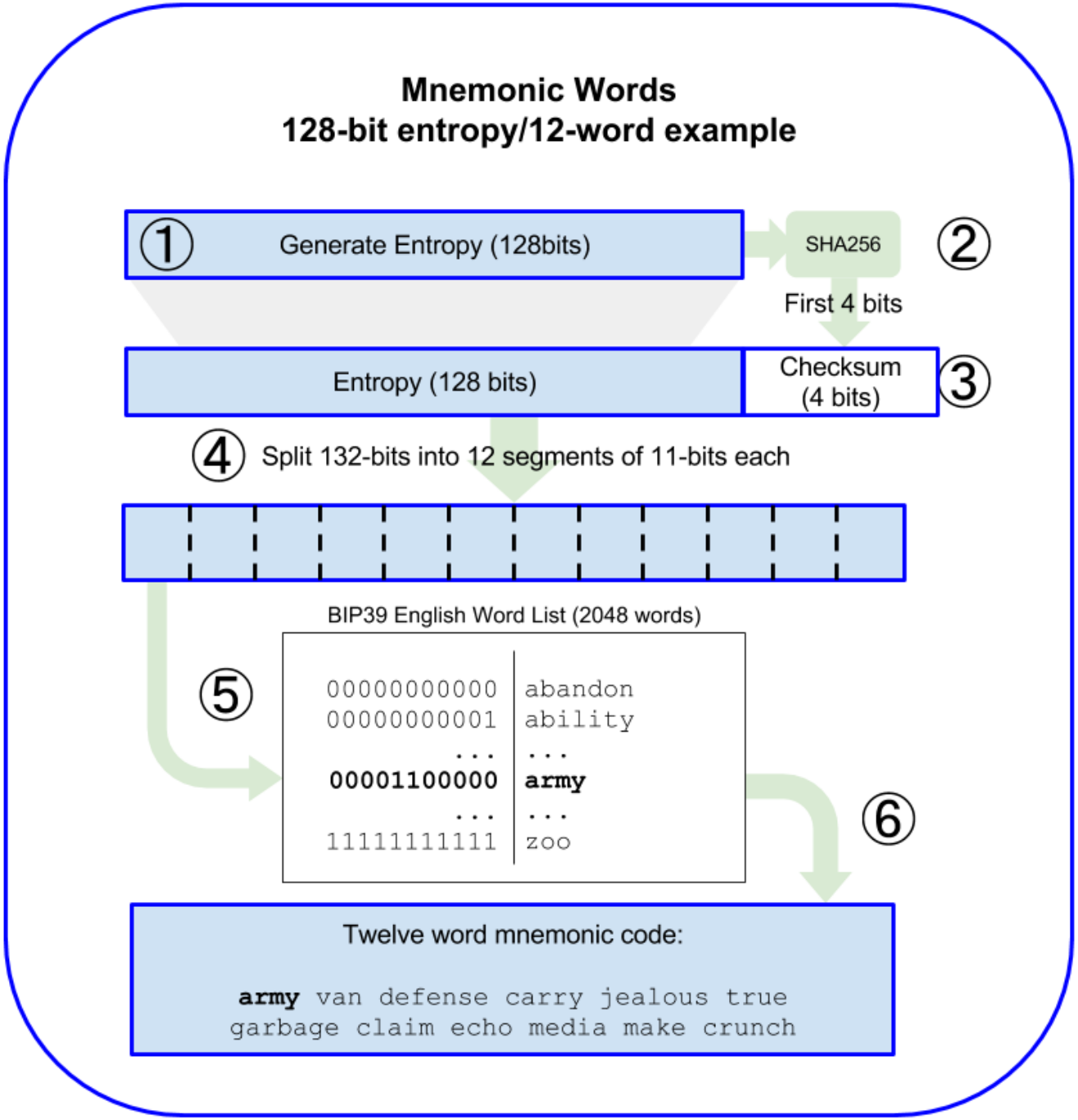

Figure 28. Generování entropie a její kódování do posloupnosti mnemotechnických slov

Převod mnemotechnického kódu na semínko

Mnemotechnická slova představují entropii s délkou 128 až 256 bitů. Entropie se potom používá k odvození delšího (512-bitového) semínka pomocí funkce natahování klíčů PBKDF2. Vytvořené semínko se používá k vytvoření deterministické peněženky a odvození jejích klíčů.

Funkce protahování klíčů má dva parametry: mnemotechnickou posloupnost slov a *sůl*. Účelem soli ve funkci protahování klíčů je ztěžovat sestavení vyhledávací tabulky umožňující útok hrubou silou. Ve standardu BIP-39 má sůl jiný účel: umožňuje zavedení přístupové fráze, která slouží jako další bezpečnostní faktor chránící semínko, jak podrobněji popíšeme v Volitelná přístupová fráze v BIP-39.

Postup popsaný v krocích 7 až 9 pokračuje v postupu popsaného v předchozí části:

7. Prvním parametrem funkce natahování klíčů PBKDF2 je *mnemotechnická posloupnost slov* vytvořená v kroku 6.
8. Druhým parametrem funkce natahování klíčů PBKDF2 je *sůl*. Sůl se skládá z řetězcové konstanty „mnemonic“ zřetězené s volitelným přístupovým heslem dodaným uživatelem.
9. PBKDF2 napíná své parametry (mnemonickou posloupnost slov a sůl) pomocí 2 048 kol hašování algoritmem HMAC-SHA512 a jako konečný výstup vytváří 512-bitovou hodnotu. Tato 512-bitová hodnota je semínko.

Převod mnemotechnické posloupnosti na semínko ukazuje, jak se mnemotechnická posloupnost slov používá ke generování semínka.

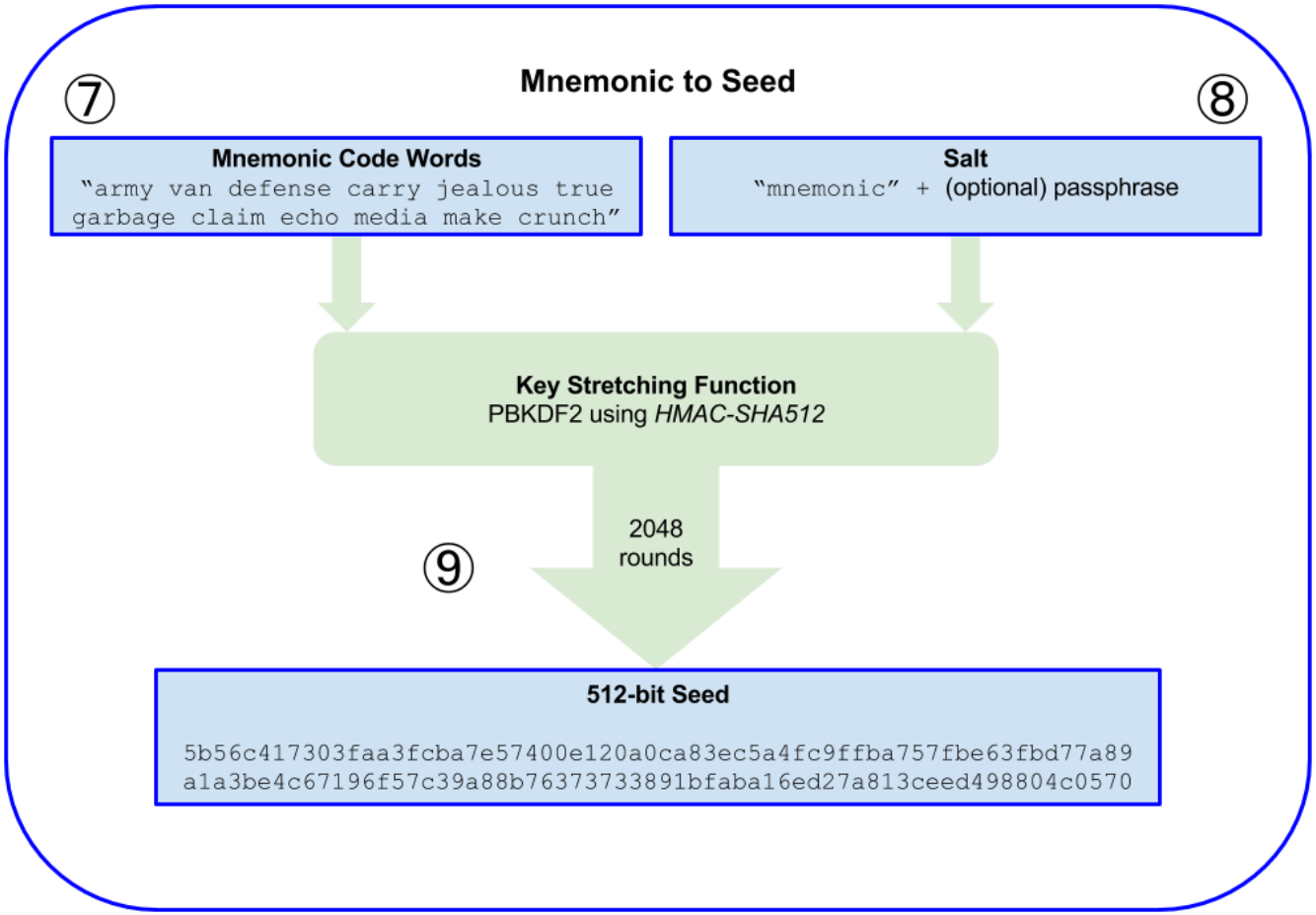

Figure 29. Převod mnemotechnické posloupnosti na semínko

Funkce natahování klíčů s 2 048 koly hašování je docela účinnou ochranou proti útokům hrubou silou proti mnemotechnické nebo přístupové frázi. Je nákladné (při výpočtu) vyzkoušet více než několik tisíc přístupových frází a mnemotechnických kombinací, zatímco počet možných odvozených semínek je obrovský (2^{512}, nebo asi 10^{154}) - mnohem větší než počet atomů ve viditelném vesmíru (asi 10^{80}).

Tabulky <a data-type="xref" data-xrefstyle="select:labelnumber" href="#mnemonic_128_no_pass">#mnemonic_128_no_pass, <a data-type="xref" data-xrefstyle="select:labelnumber" href="#mnemonic_128_w_pass">#mnemonic_128_w_pass, a <a data-type="xref" data-xrefstyle="select:labelnumber" href="#mnemonic_256_no_pass">#mnemonic_256_no_pass ukazují některé příklady

mnemotechnických kódů a semínek, které vytvoří.

Table 4. Mnemonický kód s entropií 128 bitů, bez přístupové fráze, výsledné semínko

Vstupní entropie (128 bitů)	0c1e24e5917779d297e14d45f14e1a1a
Mnemotechnická slova (12)	army van defense carry jealous true garbage claim echo media make crunch
Přístupová fráze	(žádná)
Semínko (512 bitů)	5b56c417303faa3fcba7e57400e120a0ca83ec5a4fc9ffba757fbe63fbd77a89a1a3be4c67196f57c39a88b76373733891bfaba16ed27a813ceed498804c0570

Table 5. Mnemonický kód s entropií 128 bitů, s přístupovou frází, výsledné semínko

Vstupní entropie (128 bitů)	0c1e24e5917779d297e14d45f14e1a1a
Mnemotechnická slova (12)	army van defense carry jealous true garbage claim echo media make crunch
Přístupová fráze	SuperDuperSecret
Semínko (512 bitů)	3b5df16df2157104cfdd22830162a5e170c0161653e3afe6c88defeefb0818c793dbb28ab3ab091897d0715861dc8a18358f80b79d49acf64142ae57037d1d54

Table 6. Mnemonický kód s entropií 256 bitů, bez přístupové fráze, výsledné semínko

***Vstupní entropie (256 bitů)**	2041546864449caff939d32d574753fe684d3c947c3346713dd8423e74abcf8c
Mnemotechnická slova (24)	cake apple borrow silk endorse fitness top denial coil riot stay wolf luggage oxygen faint major edit measure invite love trap field dilemma oblige
Přístupová fráze	(žádná)
Semínko (512 bitů)	3269bce2674acbd188d4f120072b13b088a0ecf87c6e4cae41657a0bb78f5315b33b3a04356e53d062e55f1e0deaa082df8d487381379df848a6ad7e98798404

Volitelná přístupová fráze v BIP-39

Standard BIP-39 umožňuje použití volitelného přístupového fráze při odvozování semínka. Pokud není použita žádná přístupová fráze, mnemotechnická slova se natáhnou solí sestávající z konstantního řetězce „mnemonic“, čímž se získá konkrétní 512-bitové semínko pro daná mnemotechnická slova. Je-li použita přístupová fráze, protahovací funkce vytvoří semínko *rozdílné* od semínka pro téže mnemotechnická slova bez přístupové fráze. Ve skutečnosti, s pro daná mnemotechnická slova, vede každá možná přístupová fráze k jinému semínku. V zásadě neexistuje „špatná“ přístupová fráze. Všechny přístupové fráze jsou platné a všechny vedou k různým semínkům a vytvářejí obrovské množství možných neinicializovaných peněženek. Sada možných peněženek je tak velká (2^{512}), že neexistuje žádná praktická možnost násilného prolomení nebo náhodného uhádnutí již používané peněženky, pokud má přístupová fráze dostatečnou složitost a délku.

V BIP-39 neexistují žádné „nesprávné“ přístupová fráze. Každá přístupová fráze vede k nějaké peněžence, která, pokud nebyla dříve použita, bude prázdná.

Volitelná přístupová fráze přináší dvě důležité funkce:

- Druhý faktor (něco zapamatovaného), díky kterému je mnemotechnická pomůcka sama o sobě k ničemu, uložené mnemotechnické zálohy jsou tak chráněny před kompromitováním zlodějem.
- Forma hodnověrného popření nebo „nátlakové peněženky“, kde zvolené přístupové heslo vede k peněžence s malým množstvím finančních prostředků, používané k odvrácení útočníka od „skutečné“ peněženky, která obsahuje většinu prostředků.

Je však důležité si uvědomit, že použití přístupové fráze také představuje riziko ztráty:

- Pokud je majitel peněženky nezpůsobilý nebo mrtvý a nikdo jiný nezná přístupové heslo, je semínko zbytečné a všechny finanční prostředky uložené v peněžence jsou navždy ztraceny.
- Naopak, pokud vlastník zálohuje přístupovou frázi na stejném místě jako semeno, ztrácí se účel druhého faktoru.

Přístupové fráze jsou sice velmi užitečné, ale měly by být použity pouze v kombinaci s pečlivě naplánovaným procesem zálohování a obnovy, přičemž je třeba vzít v úvahu možnost, aby dědicové, kteří přežijí vlastníka, mohli získat kryptoměnu.

Práce s mnemotechnickými kódy

BIP-39 je implementován jako knihovna v mnoha různých programovacích jazycích. Například:

python-mnemonic **[https://github.com/trezor/python-mnemonic]**

Referenční implementace standardu týmem SatoshiLabs, který navrhl BIP-39, v Pythonu

ConsenSys/eth-lightwallet **[https://github.com/ConsenSys/eth-lightwallet]**

Odlehčená JS Ethereum peněženka pro uzly a prohlížeče (s BIP-39)

npm/bip39 **[https://www.npmjs.com/package/bip39]**

JavaScript implementace Bitcoin BIP-39: Mnemotechnický kód pro generování deterministických klíčů

Existuje také generátor BIP-39 implementovaný do samostatné webové stránky (Generátor BIP-39 jako samostatná webová stránka), což je velmi užitečné pro testování a experimentování. Konvertor mnemotechnických kódů [https://iancoleman.io/bip39/] generuje mnemotechnická slova, semínka a rozšířené soukromé klíče. Může být použit offline v prohlížeči nebo přístupný online.

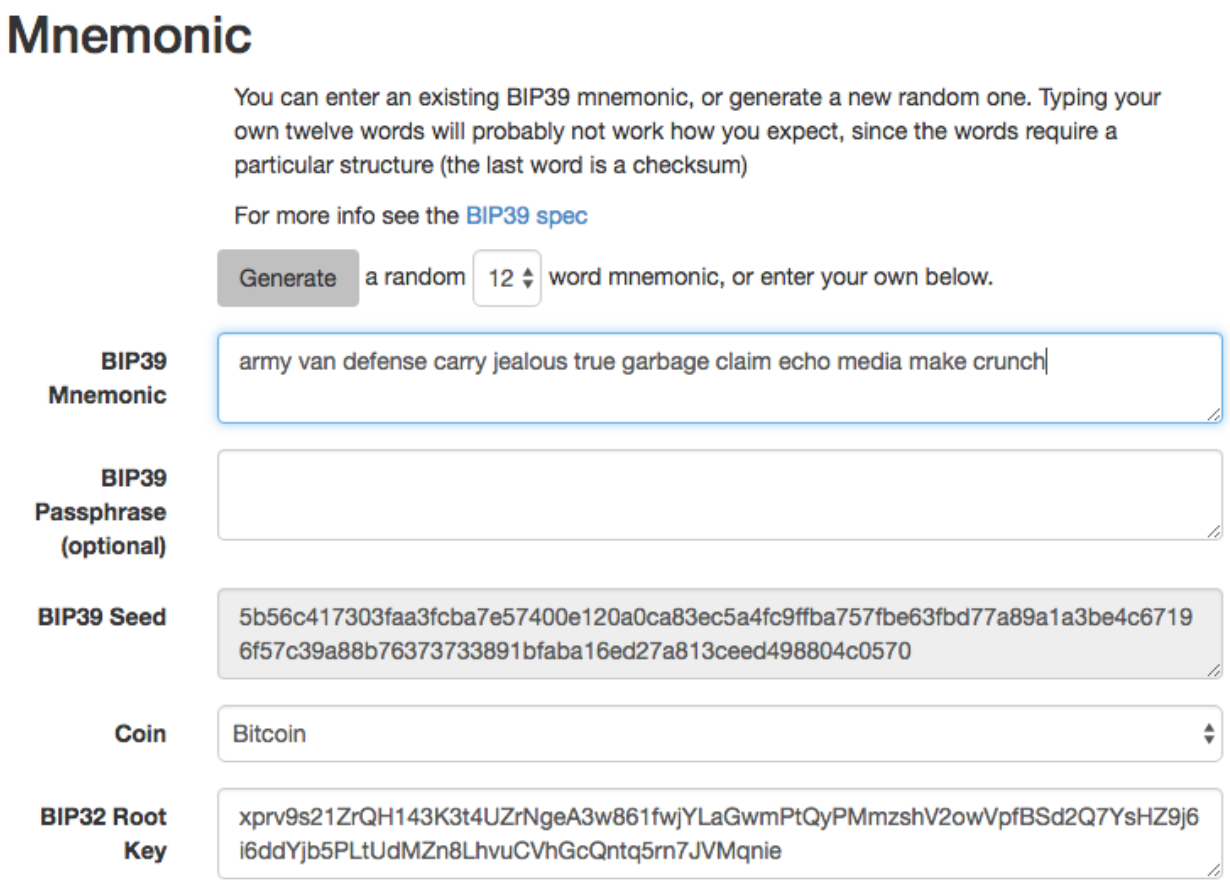

Figure 30. Generátor BIP-39 jako samostatná webová stránka

Vytváření HD peněženky ze semínka

HD peněženky jsou vytvořeny z jediného *kořenového semínka*, což je 128-, 256- nebo 512-bitové náhodné číslo. Nejčastěji je toto semínko generováno z mnemotechnických slov, jak je podrobně popsáno v předchozí části.

Každý klíč v peněžence HD je deterministicky odvozen od tohoto kořenového semínka, což umožňuje znovu vytvořit celou HD peněženku z tohoto semínka v jakékoli kompatibilní HD peněžence. To usnadňuje export, zálohování, obnovu a import HD peněženek obsahujících tisíce nebo dokonce miliony klíčů přenesením pouze mnemotechnických slov, ze které pochází kořenové semínko.

HD peněženky (BIP-32) a jejich vnitřní struktura (BIP-43/44)

Většina HD peněženek podporuje BIP-32 standard, který se *ve skutečnosti* stal průmyslovým standardem pro deterministické generování klíčů.

Nebudeme zde diskutovat o všech detailech BIP-32, pouze o komponentách nezbytných k pochopení toho, jak se používá v peněženkách. Hlavním důležitým aspektem jsou stromové hierarchické vztahy, které mohou mít odvozené klíče, jak je vidět v HD peněženka: strom klíčů generovaných z jednoho semínka. Je také důležité pochopit myšlenky rozšířených klíčů a tvrzených klíčů, které jsou vysvětleny v následujících částech.

Existuje mnoho desítek vzájemné spolupráce schopných implementací BIP-32 nabízených v mnoha softwarových knihovnách. Většinou jsou určeny pro Bitcoinové peněženky, které implementují adresy jiným způsobem, ale sdílejí stejnou implementaci odvození klíčů jako Ethereum peněženky kompatibilní s BIP-32. Použijte jednu navrženou pro Ethereum [https://github.com/ConsenSys/eth-lightwallet],nebo upravte jednu Bitcoinovou přidáním knihovny kódování Ethereum adres.

K dispozici je také BIP-32 generátor implementovaný jako samostatná [http://bip32.org/] webová stránka, která je velmi užitečná pro testování a experimentování s BIP-32.

Samostatný generátor BIP-32 není HTTPS webem. To vám připomíná, že použití tohoto nástroje není bezpečné. Je určen pouze pro testování. Klíče vytvořené touto stránkou byste neměli používat se skutečnými prostředky.

Rozšířené veřejné a soukromé klíče

V terminologii BIP-32 lze klíče „rozšířit“. Se správnými matematickými operacemi lze tyto rozšířené „rodičovské“ klíče použít k odvození "dětských" klíčů, čímž se vytvoří hierarchie výše popsaných klíčů a adres. Rodičovský klíč nemusí být ve vrcholu (kořeni) stromu. lze ho vybrat odkudkoli ve stromové hierarchii. Rozšíření klíče zahrnuje převzetí klíče samotného a připojení zvláštního *kódu řetězce*. Kód řetězce je 256-bitový binární řetězec, který je smíšené s každým klíčem, aby se vytvořily dětské klíče.

Pokud je klíč soukromým klíčem, stává se *rozšířeným soukromým klíčem*, odlišený příponou xprv:

```
xprv9s21ZrQH143K2JF8RafpqtKiTbsbaxEeUaMnNHsm5o6wCW3z8ySyH4UxFVSfZ8n7ESu7fg
ir8i...
```

_Rozšíření veřejného klíče _ se vyznačuje předponou xpub:

```
xpub661MyMwAqRbcEnKbXcCqD2GT1di5zQxVqoHPAgHNe8dv5JP8gWmDproS6kFHJnLZd23tWe
vhdn...
```

Velmi užitečnou vlastností HD peněženek je schopnost odvodit dětské veřejné klíče z rodičovských veřejných klíčů, bez znalosti soukromých klíčů. To nám dává dva způsoby, jak odvodit dětský veřejný klíč: buď přímo z dětského soukromého klíče, nebo z rodičovského veřejného klíče.

Rozšířený veřejný klíč lze proto použít k odvození všech veřejných klíčů (a pouze veřejných klíčů) v této větvi struktury HD peněženky.

Tuto zkratku lze použít k vytvoření velmi bezpečného nasazení veřejného klíče, kde server nebo aplikace má kopii rozšířeného veřejného klíče, ale vůbec žádné soukromé klíče. Takové nasazení může vytvořit nekonečný počet veřejných klíčů a Ethereum adres, ale nemůže utratit žádné peníze zaslané na tyto adresy. Mezitím na jiném, bezpečnějším serveru může rozšířený soukromý klíč odvodit všechny odpovídající soukromé klíče k podepisování transakcí a utrácení peněz.

Jednou z běžných aplikací této metody je instalace rozšířeného veřejného klíče na webový server, který obsluhuje aplikaci elektronického obchodu. Webový server může pomocí funkce odvození veřejného klíče vytvořit novou Ethereum adresu pro každou transakci (např. pro nákupní košík zákazníka) a nebude mít žádné soukromé klíče, které by byly zranitelné v případě krádeže. Bez HD peněženek je jediným způsobem, jak to provést, vygenerovat tisíce Ethereum adres na samostatném zabezpečeném serveru a poté je předem načíst na serveru elektronického obchodu. Tento přístup je těžkopádný a vyžaduje stálou údržbu, aby se zajistilo, že serveru nedojdou klíče, a proto je preferováno použití rozšířených veřejných klíčů z HD peněženek.

Další běžnou aplikací tohoto řešení jsou chladná úložiště nebo hardwarové peněženky. V tomto scénáři lze rozšířený soukromý klíč uložit do hardwarové peněženky, zatímco rozšířený veřejný klíč lze zachovat online. Uživatel může libovolně vytvářet „přijímací“ adresy, zatímco soukromé klíče jsou bezpečně uloženy offline. K utrácení prostředků může uživatel použít rozšířený soukromý klíč v offline Ethereum klientovi nebo podepisovat transakce na hardwarovém peněženkovém zařízení.

Odvození tvrzených dětských klíčů

Schopnost odvodit větev veřejných klíčů z rozšířeného veřejného klíče neboli *xpub* je velmi užitečná, ale přináší potenciální riziko. Přístup k xpub nedává přístup k dětským soukromým klíčům. Protože však xpub obsahuje kód řetězce (používaný k odvození dětských veřejných klíčů z rodičovského veřejného klíče), pokud je dětský soukromý klíč známý nebo nějak unikne, může být použit s kódem řetězce k odvození všech ostatních dětských soukromých klíče. Jediný uniklý soukromý klíč spolu s rodičovským kódem řetězce odhalí všechny soukromé klíče všech dětí. Horší je, že podřízený soukromý klíč spolu s kódem rodičovského řetězce lze použít k odvození nadřazeného soukromého klíče.

Aby se tomuto riziku čelilo, používají HD peněženky alternativní odvozovací funkci nazvanou *tvrzené odvozování*, která „narušuje“ vztah mezi rodičovským veřejným klíčem a kódem dětského řetězce. Funkce tvrzeného odvozování používá rodičovský soukromý klíč k odvození kódu dětského řetězce namísto rodičovského veřejného klíče. Tím se vytvoří „firewall“ v rodičovské / dětské posloupnosti s kódem řetězce, který nelze použít k ohrožení rodičovského nebo sourozeneckého soukromého klíče.

Zjednodušeně řečeno, pokud chcete využít výhod xpub k odvození větví veřejných klíčů, aniž byste se vystavili riziku úniku kódu řetězce, měli byste je odvodit spíše z tvrzeného rodiče než z běžného rodiče. Nejlepší praxí je mít děti úrovně 1 hlavních klíčů vždy odvozené tvrdší derivací, aby nedošlo k ohrožení hlavních klíčů.

Číslování potomků pro normální a tvrzené odvození

Je jasně žádoucí, aby bylo možné odvodit více než jeden dětský klíč z daného rodičovského klíče. Ke správě se používá indexové číslo. Každé číslo indexu, pokud je kombinováno s rodičovským klíčem pomocí speciální funkce odvození dítěte, dává jiný dětský klíč. Indexové číslo použité v BIP-32 odvozovací funkci rodič-dítě je 32-bitové celé číslo. Pro snadné rozlišení mezi klíči odvozenými prostřednictvím normální (netvrzené) odvozovací funkce od klíčů odvozených pomocí tvrzeného odvození je toto indexové číslo rozděleno do dvou rozsahů. Čísla indexu mezi 0 až 2^{31}–1 (0x0 až 0x7FFFFFFF) se používají *pouze* pro normální odvození. Čísla indexů mezi 2^{31} a 2^{32}–1 (0x80000000 až 0xFFFFFFFF) se používají pouze pro tvrzené odvození. Proto, pokud je indexové číslo menší než 2^{31}, dítě je normální, zatímco pokud je indexové číslo rovné nebo vyšší než 2^{31}, dítě je tvrzené.

Pro snazší čtení a zobrazení indexových čísel se indexová čísla pro tvrzené děti zobrazují od nuly, ale se symbolem čárky. První normální dětský klíč je proto zobrazen jako 0, zatímco první tvrzený

dětský klíč (index 0x80000000) je zobrazen jako 0'. Poté by druhý tvrzený klíč měl index 0x80000001 a zobrazoval by se jako 1' atd. Když vidíte index HD peněženky i', znamená to 2^{31} + i.

Cesta ke klíči v HD peněženkách

Klíče v HD peněžence jsou identifikovány pomocí pojmenovávací konvence zvané „cesta“, přičemž každá úroveň stromu je oddělena lomítkem (/) (viz [hd_path_table]). Soukromé klíče odvozené od hlavního soukromého klíče začínají m. Veřejné klíče odvozené od hlavního veřejného klíče začínají M. Proto první dětský soukromý klíč hlavního soukromého klíče je m/0. První dětský veřejný klíč je M/0. Druhým vnoučetem od prvního dítěte je m/0/1 atd.

„Předek“ klíče se čte zprava doleva, dokud nedosáhnete hlavního klíče, ze kterého byl odvozen. Například identifikátor m/x/y/z popisuje klíč, který je z - tým dítětem klíče m/x/y, což je y - té dítě klíče m/x, které je x - tým dítětem m.

Příklady cest v HD peněžence

HD cesta	**Popis klíče**
m/0	První (0) dětský soukromý klíč hlavního soukromého klíče (m)
m/0/0	Soukromý klíč prvního vnuka od prvního dítěte (m/0)
m/0'/0	Soukromý klíč prvního normálního vnoučete od prvního *tvrzeného* dítěte (m/0')
m/1/0	Soukromý klíč prvního vnuka od druhého dítěte (m/1)
M/23/17/0/0	Veřejný klíč prvního prapravnuka od prvního pravnuka od 18-tého vnuka od 24. dítě

Navigace ve stromové struktuře HD peněženky

Stromová struktura HD peněženky je nesmírně flexibilní. Druhou stránkou je, že také umožňuje neomezenou složitost: každý rozšířený klíč rodičů může mít 4 miliardy dětí: 2 miliardy normálních dětí a 2 miliardy tvrzených dětí. Každé z těchto dětí může mít další 4 miliardy dětí atd. Strom může být tak hluboký, jak chcete, s potenciálně nekonečným počtem generací. Se vším tímto potenciálem

může být docela obtížné procházet těmito velmi velkými stromy.

Dva BIPy nabízejí způsob, jak zvládnout tuto potenciální složitost vytvořením standardů pro stromovou strukturu HD peněženky. BIP-43 navrhuje použití prvního kaleného tvrzeného dětského indexu jako zvláštního identifikátoru, který označuje „účel“ stromové struktury. Na základě BIP-43 by HD peněženka měla používat pouze jednu větev stromu úrovně 1, přičemž indexové číslo určuje účel peněženky identifikováním struktury a jmenného prostoru zbytku stromu. Konkrétně, HD peněženka používající pouze větev m/i'/... je určena k označení konkrétního účelu a tento účel je identifikován indexovým číslem i.

BIP-44 rozšiřuje tuto specifikaci a navrhuje strukturu vícečetných účtů s více účty označenou nastavením čísla „účelu“ na 44'. Všechny HD peněženky, které podporují strukturu BIP-44, jsou identifikovány skutečností, že používají pouze jednu větev stromu: m/44'/*.

BIP-44 určuje strukturu skládající se z pěti předdefinovaných úrovní stromu:

```
m / ú¬el' / druh m¬ny' / ú¬et' / vratka / po¬adí adresy
```

První úroveň, účel', je vždy nastavena na 44'. Druhá úroveň, druh měny', specifikuje typ kryptoměnové mince, což umožňuje víceměnové HD peněženky, kde každá měna má pod druhou úrovní vlastní podstrom. V dokumentu standardu s názvem SLIP0044 [https://github.com/satoshilabs/slips/blob/master/slip-0044.md] je definováno několik měn; například Ethereum je m/44'/60', Ethereum Classic je m/44'/61', Bitcoin je m/44'/0' a testovací sítě pro všechny měny jsou m/44'/1'.

Třetí úroveň stromu je + účet'+, což umožňuje uživatelům rozdělit své peněženky do samostatných logických podúčtů pro účely účetnictví nebo organizace. Například HD peněženka může obsahovat dva Ethereum „účty“: m/44'/60'/0' am/44'/60'/1'. Každý účet je kořenem vlastního podstromu.

Protože BIP-44 byl vytvořen původně pro Bitcoin, obsahuje „vtípek“, který není ve světě Ethereum relevantní. Na čtvrté úrovni cesty vratka má HD peněženka dva podstromy: jeden pro vytváření přijímacích adres a druhou pro vytváření vratkových. V Ethereum se používá pouze cesta pro „přijímání“, protože není třeba (a ani není možné) vytvářet vratky, jako je tomu v Bitcoinu. Všimněte si, že zatímco předchozí úrovně používaly tvrzenou derivaci, tato úroveň používá normální derivaci. To umožňuje, aby tato úroveň stromu exportovala rozšířené veřejné klíče pro použití v nezabezpečeném prostředí. Použitelné adresy jsou odvozeny HD peněženkou jako děti čtvrté úrovně, takže pátá úroveň stromu je index adresy. Například třetí přijímací adresa pro Ethereum platby na primárním účtu bude M/44'/60'/0'/0/2. Příklady struktury BIP-44 HD peněženky ukazuje

několik dalších příkladů .

Table 7. Příklady struktury BIP-44 HD peněženky

HD cesta	**Popis klíče**
M/44'/60'/0'/0/2	Třetí přijímající veřejný klíč pro primární Ethereum účet
M/44'/0'/3'/1/14	Veřejný klíč 15-té vratkové adresy 4-tého Bitcoinového účtu
m/44'/2'/0'/0/1	Druhý soukromý klíč přijímací adresy v hlavním Litecoinovém účtu

Závěry

Peněženky jsou základem jakékoli bločenkové aplikace orientované na uživatele. Umožňují uživatelům spravovat kolekce klíčů a adres. Peněženky také umožňují uživatelům prokázat vlastnictví jejich etheru a autorizovat transakce pomocí digitálních podpisů, jak uvidíme v Transakce.

Transakce

Transakce jsou podepsané zprávy pocházející z externě vlastněného účtu, přenášené Ethereum sítí a zaznamenané na Ethereum bločenky. Tato základní definice skrývá mnoho překvapivých a fascinujících detailů. Dalším způsobem, jak se podívat na transakce, je to, že jsou to jediné věci, které mohou vyvolat změnu stavu nebo způsobit vykonání kontraktu v EVM. Ethereum je globální jednoinstanční stavový stroj a transakce jsou důvodem, proč tento stavový stroj „tiká“, mění svůj stav. Kontrakty neběží samy od sebe. Ethereum neběží autonomně. Všechno začíná transakcí.

V této kapitole prozkoumáme transakce, ukážeme, jak fungují, a prozkoumáme podrobnosti. Všimněte si, že velká část této kapitoly je určena těm, kteří mají zájem o správu svých transakcí na nízké úrovni, pravděpodobně proto, že píšou aplikaci peněženky; nemusíte se o to starat, pokud jste spokojeni s používáním existujících aplikací peněženky, i když možná vyhodnotíte podrobnosti jako zajímavé!

Struktura transakce

Nejprve se podívejme na základní strukturu transakce, protože je ve standardizovaném formátu (serializovaně) přenášena v Ethereum síti. Každý klient a aplikace, která přijme transakci ve standardizovaném formátu, ji uloží do paměti pomocí své vlastní interní datové struktury, možná ozdobenou metadaty, která v samotném standardizovaném síťovovém formátu transakci neexistují. Síťová standardizovaný formát je jedinou standardní formou transakce.

Transakce je formátována jako binární zpráva, která obsahuje následující data:

Nonce

: Pořadové číslo vydané původním EOA, používané k zabránění opakovanému provedení transakce

Cena plynu

: Cena plynu (ve wei), kterou je odesilatel ochoten zaplatit

Limit plynu

: Maximální množství plynu, které je původce ochoten zaplatit za celou transakci

Příjemce

Cílová Ethereum adresa

Hodnota

Množství etheru, které se má odeslat do cíle

Data

Užitečné zatížení binárními dayt s proměnnou délkou

v, r, s

Tři složky ECDSA digitálního podpisu EOA odesilatele

Struktura transakční zprávy je formátována pomocí schématu rekurzivní prefix délky (RLP), které bylo vytvořeno speciálně pro jednoduchá, formátování binárních dat v Ethereu. Všechna čísla v Ethereu jsou kódována jako big-endian celá čísla, jejichž délky jsou násobky 8 bitů.

Povšimněte si, že popisky polí (příjemce, limit plynu, atd.) Jsou zde uvedeny pro přehlednost, ale nejsou součástí formátu transakčních dat, která obsahují pole hodnot zakódovaných RLP. Obecně RLP neobsahuje žádné oddělovače polí nebo značky. Předpona délky RLP se používá k identifikaci délky každého pole. Vše, co přesahuje definovanou délku, patří do dalšího pole ve struktuře.

Zatímco jsme si představili skutečnou přenášenou strukturu transakcí, většina interních reprezentací a vizualizací uživatelského rozhraní ji ozdobí dalšími informacemi odvozenými z transakce nebo z bločenky.

Například si můžete všimnout, že zde chybí položka odesilatel transakce. Je to proto, že veřejný klíč EOA lze odvodit ze složek v,r,s ECDSA podpisu. Adresa může být zase odvozena z veřejného klíče. Když vidíte transakci zobrazující pole „odesilatel“, přidal ho tam software použitý k vizualizaci transakce. Další metadata často přidaná do transakce klientským softwarem zahrnují číslo bloku (jakmile je vytěženo a zahrnuto do bločenky) a ID transakce (vypočtený haš). Tato data jsou opět odvozena z transakce a netvoří součást samotné transakční zprávy.

Transakční nonce

Nonce je jednou z nejdůležitějších a nejméně srozumitelných složek transakce. Definice ve Žluté knize (viz Další čtení) zní:

> nonce: Skalární hodnota rovnající se počtu transakcí odeslaných z této adresy nebo, v případě účtů s přidruženým kódem, počtu kontraktů vytvořených tímto účtem.

Přesně vzato je nonce atributem původní adresy; to znamená, že má význam pouze v kontextu odesílací adresy. Nonce však není explicitně uložen jako součást stavu účtu na bločence. Místo toho se vypočítává dynamicky, spočítáním počtu potvrzených transakcí, které pocházejí z dané adresy.

Existují dva scénáře, v nichž je důležitá existence nonce počítající transakce: Vykonávání transakcí v pořadí jejich vytvoření a ochrana před vícenásobným vykonáním transakce. Podívejme se na ukázkový scénář pro každý z nich:

1. Představte si, že chcete provést dvě transakce. Máte důležitou platbu, abyste zaplatili 6 etherů, a také další platbu zaplatit 8 etherů. Nejprve podepíšete a vysíláte transakci 6-etherovou, protože ta je důležitější, a poté podepíšete a vysíláte druhou, 8-etherovou transakci. Bohužel jste přehlédli skutečnost, že váš účet obsahuje pouze 10 etherů, takže síť nemůže akceptovat obě transakce: jedna z nich selže. Protože jste nejdřív poslali důležitější 6-etherovou, pochopitelně očekáváte, že ta projde a 8-etherová bude odmítnuta. V decentralizovaném systému, jako je Ethereum, však mohou uzly přijímat transakce v jakémkoli pořadí; neexistuje žádná záruka, že konkrétní uzel bude mít jednu transakci propagovanou před druhou. Téměř jistě nastane případ, že některé uzly obdrží nejprve 6-etherovou transakci a jiné obdrží 8-etherovou transakci jako první. Bez nonce by bylo náhodné, která z nich bude přijata a která odmítnuta. Avšak s použitím nonce bude mít první transakce, kterou jste odeslali, nonce, řekněme 3, zatímco transakce 8-etherová má další hodnotu nonce (tj. 4). Tato transakce bude tedy ignorována, dokud nebudou zpracovány transakce s noncemi od 0 do 3, i kdyby byla přijata jako první. Uff!
2. Nyní si představte, že máte účet se 100 ethery. Fantastický! Najdete někoho online, který přijme platbu v éteru za nějaké záhadné zařízení (mcguffin-widget), který si opravdu chcete koupit. Pošlete jim 2 ether a vám pošlou objednané záhadné zařízení. Milé. Chcete-li provést tuto 2-etherovou platbu, podepsali jste transakci, která odeslala 2 ethery z vašeho účtu na jejich účet, a poté ji vyslala do sítě Ethereum, kde má být ověřena a zahrnuta do bločenky. Nyní, bez hodnoty nonce v transakci, bude druhá transakce odesílající 2 ethery na stejnou adresu podruhé vypadat přesně stejně jako první transakce. To znamená, že kdokoli, kdo vidí vaši transakci v síti Ethereum (což znamená každý, včetně příjemce transakce nebo vašich nepřátel), může transakci znovu a znovu přeposílat do sítě, dokud váš veškerý ether neodejde jednoduchým okopírováním původní transakce a jejím opětovným odesláním do sítě. Avšak s

hodnotou nonce zahrnutou do transakčních dat je *každá jednotlivá transakce jedinečná*, i když posíláte stejné množství etheru na stejnou adresu příjemce vícekrát. Tím, že máme zvyšující se nonci jako součást transakce, není jednoduše možné, aby někdo „duplikoval“ provedenou platbu.

Když to shrneme, je důležité si uvědomit, že použití nonce je ve skutečnosti pro protokol používající *mechanizmus účtů* životně důležité. V Bitcoinovém protokolu se transakční nonce nemusejí používat (a nepoužívají se), protože místo mechanizmu účtů využívá mechanismus „neutracených výstupů transakcí“ (Unspent Transaction Output; UTXO).

Sledování noncí

Prakticky řečeno, nonce je aktuální počet *potvrzených* transakcí (v bločence), které byly odeslány z účtu. Chcete-li zjistit, co je to nonce, můžete se podívat na bločenku, například přes rozhraní web3. Otevřete JavaScript konzoli v Geth (nebo preferovaném rozhraní web3) na testovací síti Ropsten a zadejte:

```
<pre data-type="programlisting">
&gt;
<strong>web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249
de124f")</strong>
40
</pre>
```

Nonce je počítadlo začínající nulou, což znamená, že první transakce má nonci 0. V tomto příkladu máme počet transakcí 40, což znamená, že uvidíme nonce 0 až 39. Další transakční nonce bude muset být 40.

Vaše peněženka bude sledovat nonce pro každou adresu, kterou spravuje. Je to docela jednoduché, pokud provádíte transakce pouze z jednoho místa. Řekněme, že píšete vlastní software peněženky nebo nějakou jinou aplikaci, která vytváří transakce. Jak sledujete nonce?

Když vytvoříte novou transakci, přiřadíte další nonci v pořadí. Dokud se však transakce nepotvrdí, nebude se počítat do součtu getTransactionCount.

Při použití funkce getTransactionCount k počítání čekajících transakcí buďte opatrní, protože pokud odešlete několik transakcí krátce po sobě, můžete narazit na některé problémy.

Podívejme se na příklad:

```
<pre data-type="programlisting">
&gt;
<strong>web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249
de124f", \
"pending")</strong>
40
&gt; <strong>web3.eth.sendTransaction({from: web3.eth.accounts[0], to: \
"0xB0920c523d582040f2BCB1bD7FB1c7C1ECEbdB34", value:
web3.utils.toWei(0.01, "ether")});</strong>
&gt;
<strong>web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249
de124f", \
"pending")</strong>
41
&gt; <strong>web3.eth.sendTransaction({from: web3.eth.accounts[0], to: \
"0xB0920c523d582040f2BCB1bD7FB1c7C1ECEbdB34", value:
web3.utils.toWei(0.01, "ether")});</strong>
&gt;
<strong>web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249
de124f", \
"pending")</strong>
41
&gt; <strong>web3.eth.sendTransaction({from: web3.eth.accounts[0], to: \
"0xB0920c523d582040f2BCB1bD7FB1c7C1ECEbdB34", value:
web3.utils.toWei(0.01, "ether")});</strong>
&gt;
<strong>web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249
de124f", \
"pending")</strong>
41
</pre>
```

Pokud se pokusíte tyto příklady kódu sami vytvořit v Geth javascriptové konzoli, měli byste použít web3.toWei () místo web3.utils.toWei (). Je to proto, že Geth používá starší verzi knihovny web3.

Jak vidíte, první odeslaná transakce zvýšila počet transakcí na 41, což ukazuje na čekající transakci. Když jsme však poslali další tři transakce rychle za sebou, volání getTransactionCount je nezapočítalo. Počítalo se to pouze jednu, i když byste mohli očekávat, že v paměťovém úložišti nepotvrzených transakcí (mempool) čekají tři. Pokud počkáme několik sekund, než se síťová komunikace usadí, poté volání getTransactionCount vrátí očekávané číslo. Ale prozatím, když čekáme na více než jednu transakci, nemusí nám pomoci.

Při tvorbě aplikace, která vytváří transakce, nemůže spoléhat na `getTransactionCount` pro čekající transakce. Pouze v případě, že čekající a potvrzené počty jsou stejné (všechny nepotvrzené transakce jsou potvrzeny), můžete důvěřovat výstupu getTransactionCount a začít počítat nonce. Proto sledujte nonce ve vaší aplikaci, dokud se každá transakce nepotvrdí.

Parity JSON RPC rozhraní nabízí funkci parity_nextNonce, která vrací další nonci, která by měla být použita v transakci. Funkce parity_nextNonce počítá nonce správně, i když konstruujete několik transakcí v rychlém sledu bez čekání na jejich potvrzení:

```
<pre data-type="programlisting">
$ <strong>curl --data '{"method":"parity_nextNonce", \
  "params":["0x9e713963a92c02317a681b9bb3065a8249de124f"],\
  "id":1,"jsonrpc":"2.0"}' -H "Content-Type: application/json" -X POST \
  localhost:8545</strong>

{"jsonrpc":"2.0","result":"0x32","id":1}
</pre>
```

Parita má webovou konzoli pro přístup k rozhraní JSON RPC, ale zde k přístupu používáme HTTP klienta typu příkazová řádka.

Mezery v noncích, stejné nonce a potvrzení

Je důležité sledovat nonce, pokud vytváříte transakce programem, zejména pokud tak děláte z více nezávislých procesů současně.

Síť Ethereum zpracovává transakce postupně na základě nonce. To znamená, že pokud přenesete transakci s noncí 0 a pak přenesete transakci s noncí 2, druhá transakce nebude zahrnuta do žádných bloků. Bude uložena v paměťovém úložišti nepotvrzených transakcí (mempool), zatímco síť Ethereum čeká, až se objeví chybějící nonce. Všechny uzly budou předpokládat, že chybějící nonce byla prostě zpožděna a že transakce s noncí 2 byla přijata mimo pořadí.

Pokud pak odešlete transakci s chybějícím noncí 1, budou obě transakce (nonce 1 a 2) zpracovány a zahrnuty (samozřejmě, pokud jsou platné). Jakmile vyplníte mezeru, může síť těžit transakci mimo posloupnost, kterou držela v paměťovém úložišti nepotvrzených transakcí.

To znamená, že pokud vytvoříte několik transakcí postupně a jedna z nich nebude oficiálně zahrnuta do žádných bloků, budou všechny následující transakce „zaseknuty“ a budou čekat na chybějící nonci. Transakce může vytvořit neúmyslnou „mezeru“ v posloupnosti noncí, protože je neplatná nebo nemá dostatek plynu. Aby se věci znovu rozpohybovaly, musíte předat platnou transakci s chybějící noncí. Stejně tak byste si měli být vědomi toho, že jakmile je transakce s „chybějící“ noncí ověřena sítí, všechny přenosové transakce s následujícími noncesi postupně nabudou platnosti; není možné „odvolat“ transakci!

Pokud na druhou stranu omylem duplikujete nonce, například přenosem dvou transakcí se stejnou noncí, ale pro různé příjemce nebo s různou hodnotou, pak jedna z nich bude potvrzena a jedna bude odmítnuta. Která z nich je potvrzena, bude určeno pořadím, ve které dorazí k prvnímu validačnímu uzlu, který je přijímá - tj. bude celkem náhodná.

Jak vidíte, sledování potřeb je nutné, a pokud vaše aplikace tento proces nespravuje správně, narazíte na problémy. Bohužel, věci se ještě ztíží, pokud se to snažíte dělat souběžně, jak uvidíme v další části.

Souběh, vytvoření transakce a nonce

Souběžnost je složitý aspekt informatiky někdy se neočekávaně vynoří, zejména v decentralizovaných a distribuovaných systémech v reálném čase, jako je Ethereum.

Zjednodušeně řečeno, souběžnost je, když máte simultánní výpočet několika nezávislými systémy. Ty mohou být ve stejném programu (např. více vláken ve stejném procesu), na stejném počítači (např. více procesorů) nebo na různých počítačích (tj. distribuované systémy). Ethereum je ze své podstaty systém, který umožňuje souběžnost operací (uzly, klienti, DAppy), ale prosazuje jednoinstanční stav na základě konsensu.

Nyní si představte, že máte více nezávislých peněženkových aplikací, které vytvářejí transakce ze stejné adresy nebo adres. Jedním příkladem takové situace by mohla být burza provádějící výběry z horké peněženky (peněženka, jejíž klíče jsou uloženy online, na rozdíl od studené peněženky, jejíž klíče nejsou nikdy online). V ideálním případě byste chtěli mít více než jeden počítač zpracovávající výběr, aby se to nestalo problémovým bodem nebo jediným bodem selhání. To se však rychle stává problematickým, protože mít více než jeden počítač vytvářející výběry bude mít za následek některé trnité problémy souběžného provádění, v neposlední řadě je to výběr noncí. Jak koordinuje více počítačů vytváření, podepisování a odesílání transakcí ze stejného účtu v peněžence?

Jediný počítač můžete použít k přiřazení noncí na principu, kdo první o nonci požádá, ten jí dostane. Tento počítač je však nyní jediným bodem selhání. Horší je, že pokud je přiděleno více noncí a jedna z nich se nikdy nevyužije (z důvodu selhání počítače, který transakci zpracovává s touto noncí), všechny následné transakce uvíznou.

Dalším přístupem by bylo generování transakcí, ale nepřidělování jim noncí (a proto je ponechat nepodepsané - pamatujte, že nonce je nedílnou součástí transakčních dat, a proto musí být zahrnuta do digitálního podpisu, který autentizuje transakci). Pak je můžete umístit do fronty do jediného uzlu, který je podepíše a také sleduje nonce. Opět by to však bylo v tomto procesu úzké místo: podepisování a sledování noncí je ta část vaší operace, která bude pravděpodobně nejvíce vytížená, zatímco vytváření nepodepsané transakce je ta část, kterou ve skutečnosti nemáte potřebu paralelizovat. Měli byste nějakou souběžnost, ale v kritické části procesu by to chybělo.

Nakonec tyto problémy souběžnosti, kromě obtížnosti sledování zůstatků účtů a potvrzení transakcí v nezávislých procesech, nutí většinu implementací k vyhýbání se souběžnosti a vytváření úzkých míst, jako je jediný proces, který řeší všechny transakce výběru z burzy nebo nastavování více horkých peněženek, které mohou zcela samostatně řešit výběry a musí být pouze občas vyváženy.

Transakční plyn

V předchozích kapitolách jsme trochu mluvili o plynu a podrobněji o tom diskutujeme v Plyn. Pojďme však probrat některé základní informace o úloze položek cena plynu (gasPrice) a limit plynu (gasLimit).

Plyn je Ethereum palivo. Plyn není ether - je to samostatná virtuální měna s vlastním směnným kurzem proti etheru. Ethereum používá plyn k řízení množství zdrojů, které může transakce použít, protože bude zpracována na tisících počítačích po celém světě. Výpočetní model s otevřeným koncem (Turingovsky úplný) vyžaduje určitou formu měření spotřeby zdrojů, aby se zabránilo

útokům odmítnutí služby nebo neúmyslně transakcím pohlcujícím zdroje.

Plyn je oddělen od etheru, aby byl systém chráněn před volatilitou, která by mohla nastat spolu s rychlými změnami hodnoty etheru, a také jako způsob řízení důležitých a citlivých poměrů mezi náklady na různé zdroje, které plynem platí (konkrétně výpočet, paměť a úložiště).

Pole cena plynu (gasPrice) v transakci umožňuje odesilateli transakce stanovit cenu, kterou je ochoten zaplatit za plyn. Cena se měří ve wei za jednotku plynu. Například ve vzorové transakci Základy Etherea vaše peněženka nastavila +gasPrice +na 3 gwei (3 gigawei nebo 3 miliardy wei).

Populární web ETH Čerpací stanice [https://ethgasstation.info/] poskytuje informace o aktuálních cenách plynu a dalších relevantních metrikách plynu pro hlavní Ethereum síť.

Peněženky mohou upravit gasPrice v transakcích, z nichž vznikají, aby se dosáhlo rychlejšího potvrzení transakcí. Čím vyšší je gasPrice, tím rychlejší bude transakce pravděpodobně potvrzena. Naopak transakce s nižší prioritou mohou nést sníženou cenu, což má za následek pomalejší potvrzení. Minimální hodnota, na kterou lze nastavit gasPrice, je nula, což znamená transakci bez poplatků. Během období nízké poptávky po prostoru v bloku by se tyto transakce mohly velmi dobře těžit.

Minimální přijatelné gasPrice je nula. To znamená, že peněženky mohou generovat zcela zdarma transakce. V závislosti na kapacitě, nemusí být nikdy potvrzeny, ale v protokolu nic nezakazuje bezplatné transakce. Můžete najít několik příkladů takových transakcí, které byly úspěšně zahrnuty do Ethereum bločenky.

Rozhraní web3 nabízí návrh gasPrice, a to spočítáním střední ceny z několika bloků (k tomu můžeme použít truffle příkazovou řádku nebo jakoukoli JavaScript web3 příkazovou řádku):

```
<pre data-type="programlisting">
> <strong>web3.eth.getGasPrice(console.log)</strong>
> null BigNumber { s: 1, e: 10, c: [ 10000000000 ] }
</pre>
```

Druhé důležité pole související s plynem je limit plynu (gasLimit). Zjednodušeně řečeno, gasLimit udává maximální počet jednotek plynu, které je odesilatel transakce ochoten koupit za účelem

dokončení transakce. Pro jednoduché platby, což znamená transakce, které převádějí ether z jednoho EOA do jiného EOA, je potřebné množství plynu stanoveno na 21 000 jednotek plynu. Chcete-li vypočítat, kolik etheru to bude stát, vynásobíte 21 000 hodnotou gasPrice, kterou jste ochotni zaplatit. Například:

```
<pre data-type="programlisting">
> <strong>web3.eth.getGasPrice(function(err, res) {console.log(res*21000)}
)</strong>
> 2100000000000000
</pre>
```

Pokud je cílovou adresou vaší transakce kontrakt, pak lze potřebné množství plynu odhadnout, ale nelze jej přesně určit. Je to proto, že kontrakt může vyhodnotit různě podmínky, které vedou k různým větvím provádění kódu, s různými celkovými náklady na plyn. Kontrakt může provést pouze jednoduchý nebo složitější výpočet, v závislosti na podmínkách, které jsou mimo vaši kontrolu a nelze je předvídat. Abychom to demonstrovali, podívejme se na příklad: můžeme napsat chytrý kontrakt, která zvýší počitadlo při každém svém zavolání a provede určitou smyčku tolikrát, kolikrát byl zavolán. Možná, že na 100. zavolání rozdá zvláštní cenu, jako loterie, ale pro výpočet ceny je třeba provést další výpočet. Pokud zavoláte smlouvu 99-krát, stane se jedna věc, ale na 100. volání se stane něco velmi odlišného. Množství plynu, za byste měli zaplatit, závisí na tom, kolik dalších transakcí tuto funkci zavolá, než bude vaše transakce zahrnuta do bloku. Možná je váš odhad založen na 99. transakci, ale těsně před potvrzením vaší transakce někdo zavolá kontrakt po 99. Nyní jste 100. transakcí, která chcete kontrakt zavolat, a výpočetní náročnost (a náklady na plyn) je mnohem vyšší.

Chcete-li si vypůjčit běžnou analogii s používáním v Ethereu, můžete si představit gasLimit jako kapacitu palivové nádrže v autě (vaše auto je transakce). Naplňte do nádrž tolik plynu, kolik si myslíte, že bude pro cestu potřeba (výpočet potřebný k ověření vaší transakce). Můžete odhadnout částku do určité míry, ale mohou se vyskytnout neočekávané změny na vaší cestě, jako je odklon (složitější cesta provedení), které zvyšují spotřebu paliva.

Analogie k palivové nádrži je však poněkud zavádějící. Je to spíš úvěrový účet u společnost provozující čerpací stanicí, které zaplatíte po dokončení cesty, podle toho, kolik plynu jste skutečně použili. Při přenosu transakce je jedním z prvních ověřovacích kroků ověření, zda účet, ze kterého pochází, má dostatek etheru k zaplacení gasPrice*gasLimit. Částka však ve skutečnosti nebude odečtena z vašeho účtu, dokud nebude transakce dokončena. Účtuje se vám pouze plyn skutečně

spotřebovaný vaší transakcí, ale před odesláním transakce musíte mít dostatek zůstatku na maximální částku, kterou jste ochotni zaplatit.

Příjemce transakce

Příjemce transakce je uveden v poli to. Tato adresa obsahuje 20-bajtovou Ethereum adresu. Adresa může být EOA nebo adresa kontraktu.

Ethereum již neprovádí další validaci tohoto pole. Jakákoli 20-bajtová hodnota je považována za platnou. Pokud 20-bajtová hodnota odpovídá adrese bez odpovídajícího soukromého klíče nebo bez odpovídajícího kontraktu, je transakce stále platná. Ethereum nemá způsob, jak zjistit, zda byla adresa správně odvozena od veřejného klíče (a tedy ze soukromého klíče).

Ethereum protokol neověřuje adresy příjemců v transakcích. Můžete transakci poslat na adresu, která nemá žádný odpovídající soukromý klíč nebo kontrakt, čímž „spálíte“ ether, čímž jej navždy ztratíte. Ověření by mělo být provedeno na úrovni uživatelského rozhraní.

Odesláním transakce na nesprávnou adresu bude pravděpodobně odeslaný ether *spálen*, což jej učiní navždy nepřístupným (neutratitelným), protože většina adres nemá známý soukromý klíč, a proto nelze vygenerovat žádný podpis nutný pro jeho utracení. Předpokládá se, že k ověření adresy dochází na úrovni uživatelského rozhraní (viz Hexadecimální kódování s kontrolním součtem pomocí velikostí písmen (EIP-55)). Ve skutečnosti existuje řada rozumných důvodů pro spalování etheru - například jako odrazující prvek od podvádění v platebních kanálech a jiných chytrých kontraktech - a protože množství etheru je konečné, pálení etheru efektivně distribuuje spálenou hodnotu všem držitelům etheru. (v poměru k množství etheru, které drží).

Transakční hodnota a data

Hlavní „užitečné zatížení“ transakce je obsaženo ve dvou polích: hodnota (value) and data. Transakce mohou mít hodnotu i data, pouze hodnotu, pouze data, ani hodnotu ani data. Všechny čtyři kombinace jsou platné.

Transakce obsahující pouze hodnotu je *platba*. Transakce obsahující pouze data je *volání*. Transakce s hodnotou i daty je jak platbou, tak i vyvoláním. Transakce bez hodnoty a zároveň bez dat - to je pravděpodobně jen plýtvání plynem! Ale je to stále možné.

Zkusme všechny tyto kombinace. Nejprve nastavíme zdrojovou a cílovou adresu z naší peněženky, abychom ukázku udělali snadněji čitelnou:

```
src = web3.eth.accounts[0];
dst = web3.eth.accounts[1];
```

Naše první transakce obsahuje pouze hodnotu (platbu) a žádné další užitečné datové zatížení:

```
web3.eth.sendTransaction({from: src, to: dst, \
  value: web3.utils.toWei(0.01, "ether"), data: ""});
```

Naše peněženka zobrazuje potvrzovací obrazovku označující hodnotu k odeslání, jak je uvedeno v Peněženka Parity ukazuje transakci s hodnotou, ale bez dat.

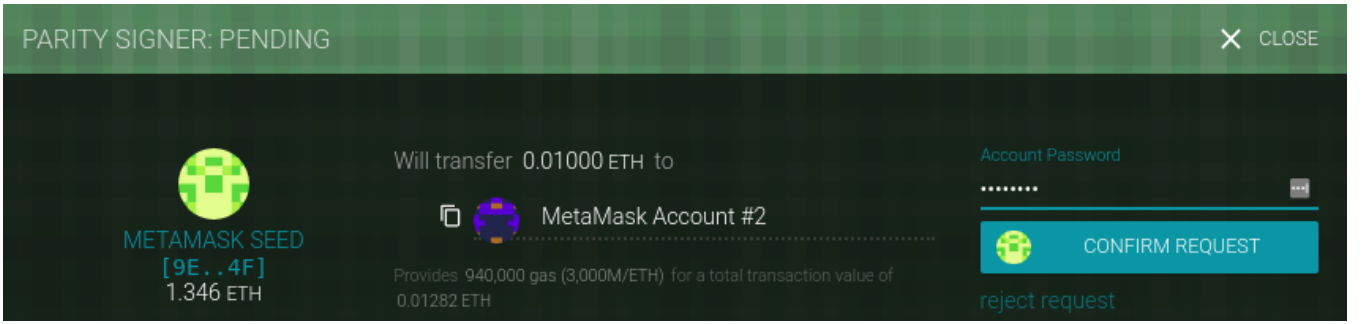

Figure 31. Peněženka Parity ukazuje transakci s hodnotou, ale bez dat

Následující příklad zadává hodnotu i užitečné datové zatížení:

```
web3.eth.sendTransaction({from: src, to: dst, \
  value: web3.utils.toWei(0.01, "ether"), data: "0x1234"});
```

Naše peněženka zobrazuje potvrzovací obrazovku označující hodnotu, která má být odeslána, a užitečné datové zatížení, jak je uvedeno Peněženka Parity ukazuje transakci s hodnotou a daty.

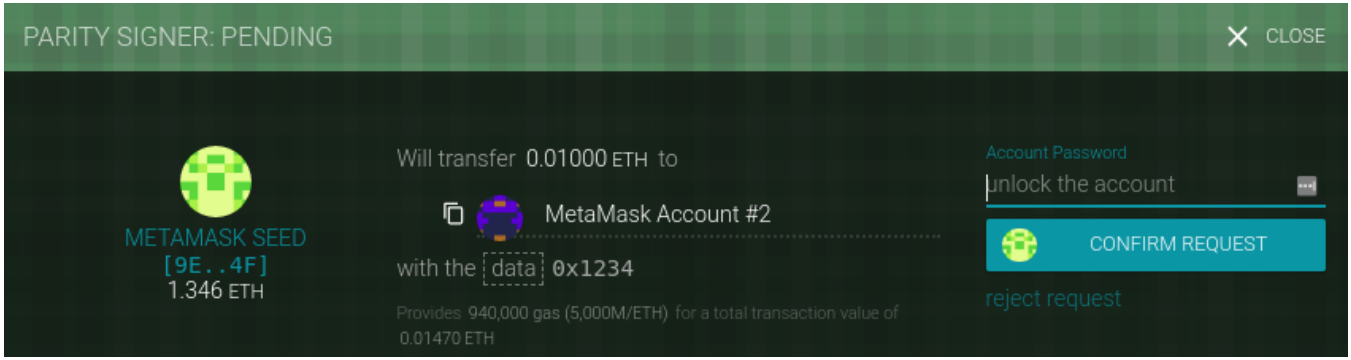

Figure 32. Peněženka Parity ukazuje transakci s hodnotou a daty

Další transakce zahrnuje užitečné datové zatížení, ale její hodnota je nulová:

```
web3.eth.sendTransaction({from: src, to: dst, value: 0, data: "0x1234"});
```

Naše peněženka zobrazuje potvrzovací obrazovku označující nulovou hodnotu a užitečné datové zatížení, jak je uvedeno v Peněženka Parity ukazuje transakci bez hodnoty, pouze data.

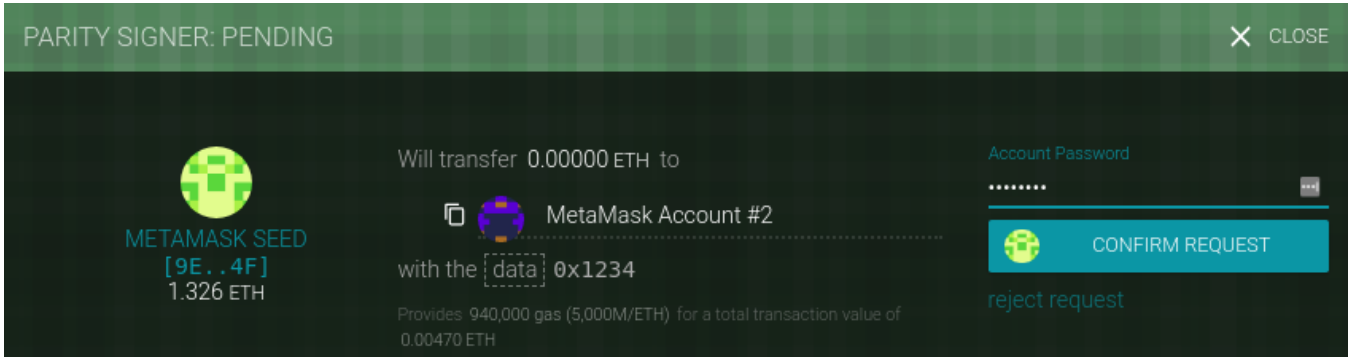

Figure 33. Peněženka Parity ukazuje transakci bez hodnoty, pouze data

Konečně poslední transakce nezahrnuje ani hodnotu k odeslání ani užitečné datové zatížení:

```
web3.eth.sendTransaction({from: src, to: dst, value: 0, data: ""}));
```

Naše peněženka zobrazuje potvrzovací obrazovku označující nulovou hodnotu, jak je uvedeno v Peněženka Parity ukazuje transakci bez hodnoty a bez dat.

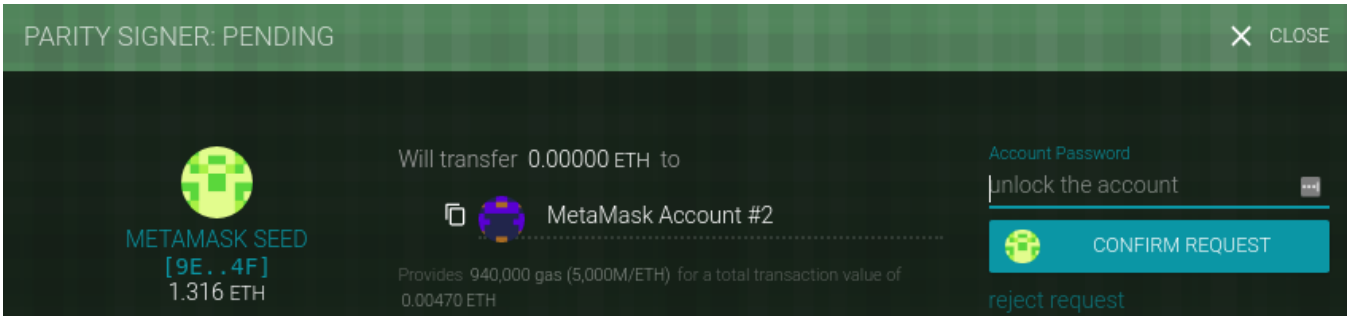

Figure 34. Peněženka Parity ukazuje transakci bez hodnoty a bez dat

Přenos hodnoty ve prospěch EOA a kontratu

Když vytvoříte Ethereum transakci, která obsahuje hodnotu, jedná se o ekvivalent *platby*. Takové transakce se chovají odlišně v závislosti na tom, zda je cílová adresa kontrakt nebo ne.

Pro adresy EOA nebo spíše pro jakoukoli adresu, která není na bločence označena jako kontrakt, Ethereum zaznamená změnu stavu a přidá hodnotu, kterou jste zaslali, do zůstatku adresy. Pokud adresa nebyla dosud použita, hodnota bude přičtena k vnitřní reprezentaci stavu klienta a zůstatek adresy bude inicializován na hodnotu vaší platby.

Pokud je cílová adresa (to) kontrakt, pak EVM spustí kontrakt a pokusí se zavolat funkci uvedenou v datovém zatížení vaší transakce. Pokud v transakci nejsou žádná data, EVM zavolá *nouzovou* funkci a pokud je tato funkce platby přijímající, provede ji, aby se určilo, co dál dělat. Pokud v nouzové funkci neexistuje žádný kód, bude výsledkem transakce zvýšení zůstatku smlouvy, přesně jako platba do peněženky. Pokud neexistuje žádná nouzová funkce nebo nouzová funkce nepřijímá platby, transakce bude vrácena.

Kontrakt může odmítnout příchozí platby okamžitým vyvoláním výjimky při vyvolání funkce nebo podle podmínek kódu funkce. Pokud funkce skončí úspěšně (bez výjimky), pak je stav kontraktu aktualizován, aby odrážel zvýšení etherové zůstatku kontraktu range="endofrange", startref="ix_06transactions-asciidoc7")

Přenos užitečného datového zatížení EOA nebo kontraktu

Pokud vaše transakce obsahuje data, je s největší pravděpodobností adresována na adresu kontraktu. To neznamená, že do EOA nemůžete odeslat užitečné datové zatížení - je to v Ethereum protokolu zcela platné. V takovém případě však interpretace dat závisí na peněžence, kterou

používáte pro přístup k EOA. Ethereum protokol je ignoruje. Většina peněženek také ignoruje veškerá data přijatá v transakci na účet EOA, který řídí. V budoucnu je možné, že se mohou objevit standardy, které umožní peněženkám interpretovat data způsobem, jakým to dělají kontrakty, což umožní transakcím vyvolat funkce běžící uvnitř uživatelských peněženek. Zásadní rozdíl spočívá v tom, že jakákoli interpretace užitečného datového zatížení prostřednictvím EOA nepodléhá Ethereum pravidlům ohledně konsensu, na rozdíl od vykonávání kontraktu.

Prozatím předpokládejme, že vaše transakce doručuje data na adresu kontraktu. In that case, the data will be interpreted by the EVM as a *contract invocation*. ("function invocation" Většina smluv používá tato data konkrétněji pro *volání funkcí*, volání pojmenované funkce a předání naformátovaných parametrů této funkci.

Užitečné datové zatížení odeslané ABI-kompatibilnímu kontraktu (můžeme předpokládat, že to jsou všechny kontrakty) má následující formát v hexadecimálním kódování:

Výběr funkce

: Keccak-256 haš prvních 4 bajtů hlavičky funkce. To umožňuje smlouvě jednoznačně určit, kterou funkci chcete vyvolat.

Parametry funkce

: Parametry funkce, kódované podle pravidel pro různé datové typy definovaných ve specifikaci ABI.

V Faucet.sol: Chytrý kontrakt Kohoutek v Solidity jsme definovali funkci pro výběry:

```
function withdraw(uint withdraw_amount) public {
```

Hlavička funkce je definována jako řetězec obsahující název funkce, následovaný datovými typy každého z jeho parametrů, uzavřených v závorkách a oddělených čárkami. Název funkce je withdraw a vyžaduje jediný parametr, který je typu uint (což je jiné označení pro uint256), takže hlavička withdraw bude:

```
withdraw(uint256)
```

Pojďme vypočítat Keccak-256 haš tohoto řetězce:

```
<pre data-type="programlisting">
> <strong>web3.utils.sha3("withdraw(uint256)");</strong>
'0x2e1a7d4d13322e7b96f9a57413e1525c250fb7a9021cf91d1540d5b69f16a49f'
</pre>
```

První 4 bajty haše jsou 0x2e1a7d4d. To je naše hodnota „výběru funkce“, která kontraktu sdělí, jakou funkci chceme volat.

Dále si vypočtěte hodnotu, která bude předána jako parametr withdraw_amount. Chceme odebrat 0,01 etheru. Zakódujme to do hexadecimálního big-endian 256-bitového celého čísla bez znaménka, v jednotkách wei:

```
<pre data-type="programlisting">
> <strong>withdraw_amount = web3.utils.toWei(0.01, "ether");</strong>
'10000000000000000'
> <strong>withdraw_amount_hex = web3.utils.toHex(withdraw_amount);</strong>
'0x2386f26fc10000'
</pre>
```

Nyní přidáme výběr funkce k množství (zarovnanému na 32 bajtů):

```
2e1a7d4d00000000000000000000000000000000000000000000000002386f26fc10000
```

To je užitečné datové zatížení pro naši transakci, volající funkci withdraw a vyžadující 0,01 etheru jako withdraw_amount .

Zvláštní transakce: Vytvoření kontraktu

Jedním zvláštním případem, který bychom měli zmínit, je transakce, která na bločence *vytvoří nový kontrakt* a připraví ho pro budoucí použití. Transakce vytvoření kontraktu jsou zasílány na zvláštní cílovou adresu zvanou *nulová adresa*; pole to v transakci vytvářející kontrakt obsahuje adresu 0x0. Tato adresa nepředstavuje EOA (neexistuje žádný odpovídající pár soukromých a veřejných klíčů) ani kontrakt. Nikdy nemůže utratit ether ani zahájit transakci. Používá se pouze jako cíl, se zvláštním významem „vytvořit tento kontrakt“.

Zatímco nulová adresa je určena pouze pro vytvoření kontraktu, občas přijímá platby z různých adres. Existují dvě vysvětlení: buď je to omyl, což má za následek ztrátu etheru, nebo je to záměrné spálení etheru (úmyslně ničí ether jeho odesláním na adresu, ze které ho nelze nikdy utratit). Pokud však chcete provést úmyslné spálení etheru, měli byste svůj záměr objasnit síti a místo toho použít speciálně určenou adresu pro pálení:

```
0x000000000000000000000000000000000000dEaD
```

Jakýkoli ether poslaný na určenou adresu pro pálení se stane nevratným a bude navždy ztracen.

Transakce vytvoření kontraktu musí obsahovat pouze užitečné datové zatížení, které obsahuje kompilovaný bajtkód, který vytvoří kontrakt. Jediným účinkem této transakce je vytvoření kontraktu. Pokud chcete nastavit nový kontrakt s počátečním zůstatkem, můžete do pole value přidat částku etheru, ale to je zcela volitelné. Pokud odešlete hodnotu (ether) na adresu pro vytvoření smlouvy bez užitečného datového zatížení (bez kontraktu), pak je účinek stejný jako odesílání na adresu pro pálení - neexistuje žádný kontrakt v jehož prospěch by ether mohl být započítán, takže ether je ztracen.

Jako příklad můžeme vytvořit ručně kontrakt *Faucet.sol* použitý v Základy Etherea vytvořením transakce na nulovou adresu se smlouvou v užitečném datovém zatížení. Smlouva musí být zkompilována do bajtkód reprezentace. To lze provést pomocí Solidity kompilátoru:

```
<pre data-type="programlisting" class="pagebreak-before">
$ <strong>solc --bin Faucet.sol</strong>

Binárn¬:
6060604052341561000f57600080fd5b60e58061001d6000396000f3006060604052600436
1060...
</pre>
```

Stejné informace lze také získat od online kompilátoru Remix.

Nyní můžeme vytvořit transakci:

```
<pre data-type="programlisting">
> <strong>src = web3.eth.accounts[0];</strong>
> <strong>faucet_code = \

"0x606060405234156100f57600080fd5b60e58061001d6000396000f300606...f0029";
</strong>
> <strong>web3.eth.sendTransaction({from: src, to: 0, data: faucet_code, \
  gas: 113558, gasPrice: 200000000000});</strong>

"0x7bcc327ae5d369f75b98c0d59037eec41d44dfae75447fd753d9f2db9439124b"
</pre>
```

Doporučujeme vždy zadat parametr odesilatele to, a to i v případě transakce vytvářející kontrakt (nulovou adresu), protože cena za neúmyslné odeslání etheru na adresu 0x0 a jeho nenávratné ztracení je příliš vysoká. Měli byste také zadat cenu plynu gasPrice a limit plynu gasLimit.

Jakmile je kontrakt vytěžen, můžeme jej vidět v Etherscan průzkumníku bloků, jak je uvedeno v Ethercan zobrazuje úspěšně vytěžený kontrakt.

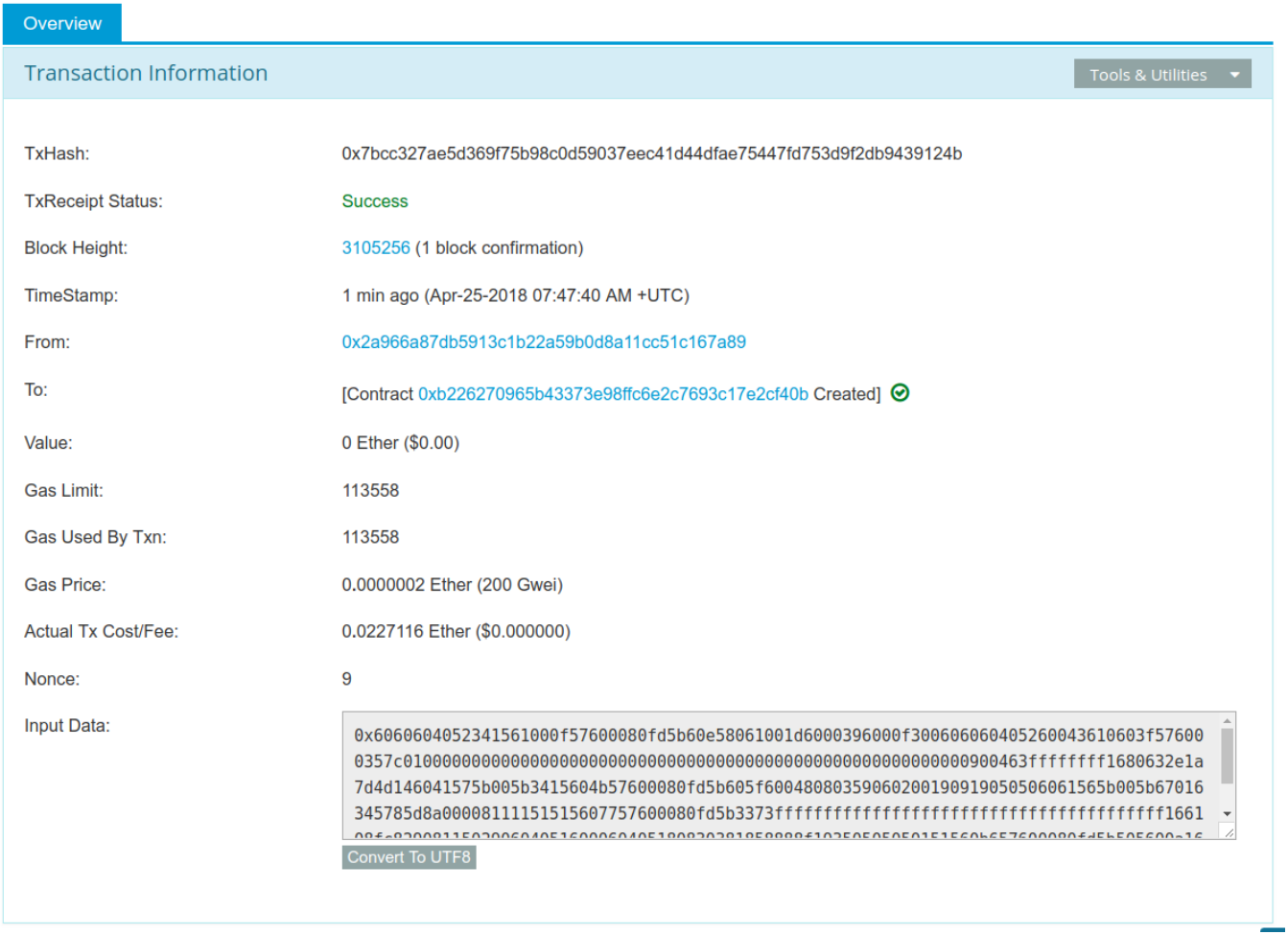

Figure 35. Ethercan zobrazuje úspěšně vytěžený kontrakt

Můžeme se podívat na příjemce transakce a získat informace o kontraktu:

```
<pre data-type="programlisting">
> <strong>web3.eth.getTransactionReceipt( \

"0x7bcc327ae5d369f75b98c0d59037eec41d44dfae75447fd753d9f2db9439124b");</st
rong>

{
  blockHash:
"0x6fa7d8bf982490de6246875deb2c21e5f3665b4422089c060138fc3907a95bb2",
  blockNumber: 3105256,
  contractAddress: "0xb226270965b43373e98ffc6e2c7693c17e2cf40b",
  cumulativeGasUsed: 113558,
  from: "0x2a966a87db5913c1b22a59b0d8a11cc51c167a89",
  gasUsed: 113558,
  logs: [],
  logsBloom: \
    "0x00000000000000000000000000000000000000000000000000...00000",
  status: "0x1",
  to: null,
  transactionHash: \
    "0x7bcc327ae5d369f75b98c0d59037eec41d44dfae75447fd753d9f2db9439124b",
  transactionIndex: 0
}
</pre>
```

To zahrnuje adresu kontraktu, kterou můžeme použít k zasílání prostředků a přijímání prostředků od kontraktu, jak je uvedeno v předchozí části:

```
<pre data-type="programlisting">
> <strong>contract_address =
"0xb226270965b43373e98ffc6e2c7693c17e2cf40b"</strong>
> <strong>web3.eth.sendTransaction({from: src, to: contract_address, \
  value: web3.utils.toWei(0.1, "ether"), data: ""});</strong>

"0x6ebf2e1fe95cc9c1fe2e1a0dc45678ccd127d374fdf145c5c8e6cd4ea2e6ca9f"

> <strong>web3.eth.sendTransaction({from: src, to: contract_address,
value: 0, data: \

"0x2e1a7d4d000000000000000000000000000000000000000000000002386f26fc1000
0"});</strong>

"0x59836029e7ce43e92daf84313816ca31420a76a9a571b69e31ec4bf4b37cd16e"
</pre>
```

Po chvíli jsou obě transakce nviditelné a Etherscan, jak je znázorněno v Etherscan zobrazuje transakce pro odesílání a přijímání prostředků.

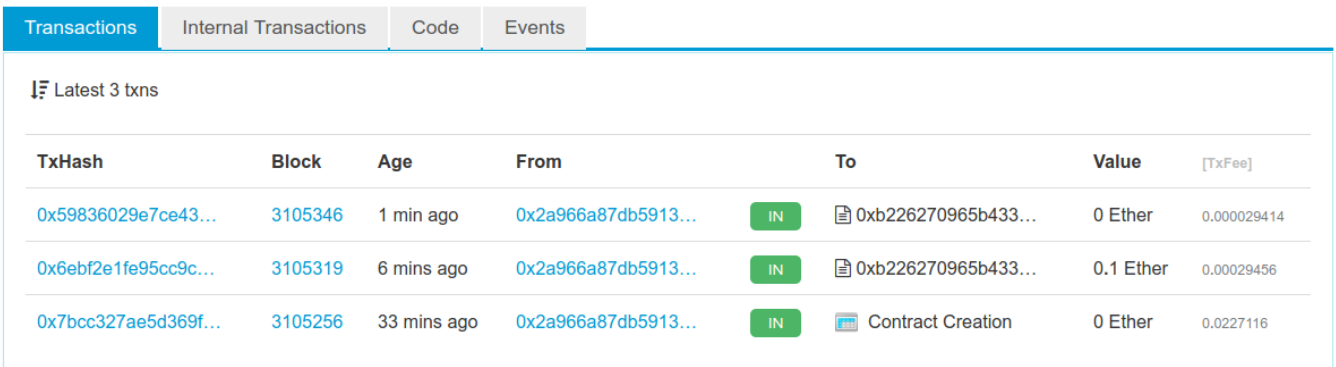

Figure 36. Etherscan zobrazuje transakce pro odesílání a přijímání prostředků

Digitální podpisy

Dosud jsme se nezachytili do podrobností o digitálních podpisech. V této části se podíváme na to, jak digitální podpisy fungují a jak je lze použít k předložení důkazu o vlastnictví soukromého klíče, aniž by tento soukromý klíč byl prozrazen.

Algoritmus digitálního podpisu pomocí eliptické křivky

Algoritmus digitálního podpisu používaný v Ethereu je *algoritmus digitálního podpisu pomocí eliptické křivky* (Elliptic Curve Digital Signature Algorithm; ECDSA). Je založen na dvojicích soukromých a veřejných klíčů, jak je popsáno Vysvětlení kryptografie eliptických křivek.

Digitální podpis slouží v Ethereu ke třem účelům (viz následující postranní panel). Podpis nejprve prokazuje, že vlastník soukromého klíče, který je implicitně majitelem Ethereum účtu, povolil utrácení etheru nebo provedení kontraktu. Za druhé, zaručuje *nepopiratelnost*: důkaz o autorizaci je nepopiratelný. Za třetí, podpis prokazuje, že data transakce nebyla a *nemohla být* po podpisu transakce upravena.

Definice digitálního podpisu na Wikipedii

Elektronický podpis je matematické schéma pro prokazování pravosti digitálních zpráv nebo dokumentů. Platný digitální podpis dává příjemci důvod se domnívat, že zpráva byla vytvořena známým odesílatelem (ověřování), že odesílatel nemůže popřít odeslání zprávy (nepopiratelnost) a že zpráva nebyla změněna při přenosu (integrita) .

Zdroj: https://en.wikipedia.org/wiki/Digital_signature

Jak fungují digitální podpisy

Digitální podpis je matematické schéma, které se skládá ze dvou částí. První část je algoritmus pro vytvoření podpisu pomocí soukromého klíče (podpisový klíč) ze zprávy (což je v našem případě transakce). Druhá část je algoritmus, který umožňuje komukoli ověřit podpis pouze pomocí zprávy a veřejného klíče.

Vytvoření digitálního podpisu

V Ethereum implementaci ECDSA je podepisovanou „zprávou“ transakce nebo přesněji, Keccak-256 haš dat transakce kódovaných RLP. Podpisový klíč je soukromý klíč EOA. Výsledkem je podpis:

```
<div data-type="equation">
<math xmlns="http://www.w3.org/1998/Math/MathML" display="block">
  <mrow>
    <mrow>
      <mi>S</mi>
      <mi>i</mi>
      <mi>g</mi>
    </mrow>
    <mo>=</mo>
    <msub><mi>F</mi> <mrow><mi>s</mi><mi>i</mi><mi>g</mi></mrow> </msub>
    <mrow>
      <mo>(</mo>
      <msub><mi>F</mi>
<mrow><mi>k</mi><mi>e</mi><mi>c</mi><mi>c</mi><mi>a</mi><mi>k</mi><mn>256<
/mn></mrow> </msub>
      <mrow>
        <mo>(</mo>
        <mi>m</mi>
        <mo>)</mo>
      </mrow>
      <mo>,</mo>
      <mi>k</mi>
      <mo>)</mo>
    </mrow>
  </mrow>
</math>
</div>
```

kde:

- k je soukromý klíč provádějící podepisování.
- m je transakce kódovaná RLP.
- $F_{keccak256}$ je hašovací funkce Keccak-256.
- F_{sig} je podpisový algoritmus.
- Sig je výsledný podpis.

Funkce F_{sig} vytváří podpis Sig, který se skládá ze dvou hodnot, které se běžně označují jako r a s:

```
<div data-type="equation">
<math xmlns="http://www.w3.org/1998/Math/MathML" display="block">
  <mrow>
    <mrow>
      <mi>S</mi>
      <mi>i</mi>
      <mi>g</mi>
    </mrow>
    <mo>=</mo>
    <mo>(</mo>
    <mi>r</mi>
    <mo>,</mo>
    <mi>s</mi>
    <mo>)</mo>
  </mrow>
</math>
</div>
```

Ověření podpisu

Pro ověření podpisu je třeba mít podpis (*r* a *s*), naformátovanou transakci a veřejný klíč, který odpovídá soukromému klíči použitému k vytvoření podpisu. V zásadě ověření podpisu znamená „pouze vlastník soukromého klíče, který vygeneroval tento veřejný klíč, mohl vyrobit podpis pro tuto transakci.“

Algoritmus ověření podpisu přebírá zprávu (tj. haš transakce pro naše použití), veřejný klíč podpisovatele a podpis (hodnoty *r* a *s*) a vrací true, pokud je podpis platný pro tuto zprávu a veřejný klíč .

Matematika ECDSA

Jak již bylo zmíněno, podpisy jsou vytvářeny matematickou funkcí F_{sig}, která vytváří podpis složený ze dvou hodnot *r* a *s*. V této sekci se podrobněji podíváme na funkci F_{sig}.

Algoritmus podpisu nejprve kryptograficky bezpečným způsobem vygeneruje *dočasný* soukromý klíč. Tento dočasný klíč se používá při výpočtu hodnot *r* a *s*, aby se zajistilo, že útočníci sledující podepsané transakce v Ethereum síti nemohou vypočítat skutečný soukromý klíč odesílatele.

Jak víme z Veřejné klíče, dočasný soukromý klíč se používá k odvození odpovídajícího (dočasného) veřejného klíče, takže máme:

- Kryptograficky bezpečné náhodné číslo *q*, které se používá jako dočasný soukromý klíč
- Odpovídající dočasný veřejný klíč *Q*, vytvořený z *q* a generátorového bodu eliptické křivky *G*

Hodnota *r* digitálního podpisu je pak *x* souřadnicí dočasného veřejného klíče *Q*.

Odtud algoritmus vypočítá hodnotu *s* podpisu, takže:

```
<ul class="simplelist">
<li><em>s</em> &#8801; <em>q</em><sup>-1</sup>
(<em>Keccak256</em>(<em>m</em>) + <em>r</em> * <em>k</em>)      
(<em>mod p</em>)</li>
</ul>
```

kde:

- *q* je dočasný soukromý klíč.
- *r* je *x* souřadnice souřadnice dočasného veřejného klíče.
- *k* je podpisový soukromý klíč (vlastníka EOA).
- *m* jsou transakční data.
- *p* je prvočíslo, udávající řád konečného tělesa eliptické křivky.

Ověření je inverzní funkce k vytvoření podpisu, pomocí hodnot *r* a *s* a veřejného klíče odesílatele vypočte hodnotu *Q*, což je bod na eliptické křivce (dočasný veřejný klíč používaný při vytváření podpisu). Kroky jsou následující:

1. Zkontrolujte, zda jsou všechny vstupy správně vytvořeny
2. Vypočítejte $w = s^{-1} \ mod \ p$
3. Vypočítejte $u_1 = Keccak256(m) * w \ mod \ p$
4. Vypočítejte $u_2 = r * w \ mod \ p$
5. Nakonec vypočítejte bod na eliptické křivce $Q \equiv u_1 * G + u_2 * K \quad (mod \ p)$

kde:

- *r* a *s* jsou hodnoty podpisu.
- *K* je veřejný klíč toho, kdo provedl podpis (vlastníka EOA).
- *m* jsou transakční data, která byla podepsána.
- *G* je generátorový bod eliptické křivky.
- *p* je prvočíslo, udávající řád konečného tělesa eliptické křivky.

Pokud je souřadnice *x* vypočítaného bodu *Q* rovna *r*, může ověřovatel dojít k závěru, že podpis je platný.

Upozorňujeme, že při ověřování podpisu není soukromý klíč znám ani odhalen.

ECDSA je nutně docela komplikovaný kus matematiky; úplné vysvětlení je nad rámec této knihy. Několik skvělých průvodců online vás provede krok za krokem: hledejte „vysvětlil ECDSA“ nebo zkuste tuto adresu: http://bit.ly/2r0HhGB.

Podepisování transakcí v praxi

Aby bylo možné vytvořit platnou transakci, musí ji její tvůrce zprávu digitálně podepsat pomocí algoritmu digitálního podpisu pomocí eliptické křivky. Když řekneme „podepsat transakci“, máme na mysli „podepsat Keccak-256 haš z transakčních dat naformátovaných RLP“. Podpis se použije na haš transakčních dat, nikoli na transakci samotnou.

Chcete-li podepsat transakci v Ethereu, musí její tvůrce:

1. Vytvořte strukturu transakčních dat, která obsahuje devět polí: nonce, gasPrice, gasLimit, to, value, data, chainID, 0, 0.
2. Vytvořte zprávu tvořenou RLP kódovanou strukturou transakčních dat.
3. Vypočítejte Keccak-256 haš této naformátované zprávy.
4. Vypočítejte ECDSA podpis a podepište haš soukromým klíčem EOA tvůrce transakce.
5. K transakci připojte vypočtené hodnoty v, r, and s ECDSA podpisu.

Speciální proměnná podpisu+v+ označuje dvě věci: ID řetězce a identifikátor zotavení, které

pomohou funkci ECDSArecover kontrolují podpis. Vypočítá se jako buďto jako 27 nebo 28, nebo jako dvojnásobek ID řetězce plus 35 nebo 36. Další informace o ID řetězce viz Tvorba surové transakce dle EIP-155. Identifikátor zotavení (27 nebo 28 v podpisech „ve starém stylu“ nebo 35 nebo 36 v úplných transakcích typu Spurious Dragon) se používá k označení shodnosti (parity) složky y veřejného klíče (viz Hodnota předpony podpisu (v) a obnova veřejného klíče pro více informací).

V bloku # 2 675 000 implementovalo Ethereum tvrdé rozvětvení „Spurious Dragon,“ která mimo jiné zavedla nové schéma podepisování, které zahrnuje ochranu před zopakováním transakce (zabránění transakcím určeným pro jednu síť jejich použití ještě na další síti). Toto nové schéma podepisování je uvedeno v EIP-155. Tato změna ovlivňuje formu transakce a její podpis, takže je třeba věnovat pozornost první ze tří proměnných podpisu (tj. v), která má jednu ze dvou forem a označují datová pole obsažená v transakční zprávě, která je hašována.

Tvorba a podpisování surových transakcí

V této sekci vytvoříme surovou transakci a podepíšeme ji pomocí knihovny ethereumjs-tx, která může být nainstalována pomocí npm. To demonstruje funkce, které by se normálně používaly uvnitř peněženky nebo aplikace, která podepisuje transakce jménem uživatele. Zdrojový kód tohoto příkladu je v souboru *raw_tx_demo.js* v GitHub úložišti knihy [http://bit.ly/2yI2GL3]:

```
// Load requirements first:
//
// npm init
// npm install ethereumjs-tx
//
// Run with: $ node raw_tx_demo.js
const ethTx = require('ethereumjs-tx');

const txData = {
  nonce: '0x0',
  gasPrice: '0x09184e72a000',
  gasLimit: '0x30000',
  to: '0xb0920c523d582040f2bcb1bd7fb1c7c1ecebdb34',
  value: '0x00',
  data: '',
  v: "0x1c", // Ethereum mainnet chainID
  r: 0,
  s: 0
};

tx = new ethTx(txData);
console.log('RLP-Encoded Tx: 0x' + tx.serialize().toString('hex'))

txHash = tx.hash(); // This step encodes into RLP and calculates the hash
console.log('Tx Hash: 0x' + txHash.toString('hex'))

// Sign transaction
const privKey = Buffer.from(
    '91c8360c4cb4b5fac45513a7213f31d4e4a7bfcb4630e9fbf074f42a203ac0b9',
'hex');
tx.sign(privKey);

serializedTx = tx.serialize();
rawTx = 'Signed Raw Transaction: 0x' + serializedTx.toString('hex');
console.log(rawTx)
```

Spuštění vzorového kódu přináší následující výsledky:

```
<pre data-type="programlisting">
$ <strong>node raw_tx_demo.js</strong>
RLP-Encoded Tx:
0xe6808609184e72a0008303000094b0920c523d582040f2bcb1bd7fb1c7c1...
Tx Hash: 0xaa7f03f9f4e52fcf69f836a6d2bbc7706580adce0a068ff6525ba337218e6992
Signed Raw Transaction:
0xf866808609184e72a0008303000094b0920c523d582040f2bcb1...
</pre>
```

Tvorba surové transakce dle EIP-155

EIP-155 „Standard Simple Replay Attack Protection“ specifikuje kódování transakcí chránící je proti opakovanému použití, což zahrnuje *identifikátor řetězu* uvnitř transakčních dat před podpisem. Tím je zajištěno, že transakce vytvořené pro jednu bločenku (např. hlavní Ethereum síť) jsou neplatné na jiném bločence (např. Ethereum Classic nebo testovací síti Ropsten). Transakce vysílané v jedné síti proto nemohou být *zopakovány* v jiné, což vysvětluje název standardu.

EIP-155 přidá tři pole k hlavním šesti polím struktury transakčních dat, jmenovitě řetězový identifikátor, 0 a 0. Tato tři pole jsou přidána k transakčním datům dříve, než jsou zakódována a hašována. Proto mění haš transakce, na kterou se později použije podpis. Zahrnutím identifikátoru řetězu do podepisovaných dat podpis transakce zabrání jakýmkoli změnám, protože podpis je zneplatněn, pokud je identifikátor řetězu změněn. EIP-155 proto znemožňuje, aby byla transakce přehrána v jiném řetězu, protože platnost podpisu závisí na identifikátoru řetězu.

Pole identifikátoru řetězu má hodnotu podle sítě, pro kterou je transakce určena, jak je uvedeno v Identifikátory řetězů.

Table 8. Identifikátory řetězů

Řetěz	**ID řetězu**
Ethereum hlavní síť	1
Morden (již nepodporovaný), Expanse	2
Ropsten	3
Rinkeby	4
Rootstock hlavní síť	30

Řetěz	ID řetězu
Rootstock testovací síť	31
Kovan	42
Ethereum Classic hlavní síť	61
Ethereum Classic hlavní síť	62
Geth soukromá testovací síť	1337

Výsledná transakční struktura je kódována RLP, hašovaná a podepsaná. Algoritmus podpisu je mírně upraven tak, aby kódoval identifikátor řetězce také v předponě v.

Více podrobností viz specifikace EIP-155 [http://bit.ly/2CQUgne].

Hodnota předpony podpisu (v) a obnova veřejného klíče

Jak je uvedeno v <<tx_struct> > transakční zpráva neobsahuje pole „odesilatel“. Je to proto, že veřejný klíč odesilatele lze vypočítat přímo z ECDSA podpisu. Jakmile budete mít veřejný klíč, můžete adresu snadno spočítat. Proces obnovy veřejného klíče podpisu se nazývá obnovení veřejného klíče.

K zadaným hodnotám r a s, které byly vypočítány v Matematika ECDSA můžeme vypočítat dva možné veřejné klíče.

Nejprve vypočítáme dva body eliptické křivky, R a $R^{'}$ z hodnoty jeho x -ové souřadnice r, která je v podpisu. Existují dva body, protože eliptická křivka je symetrická podle osy x, takže pro jakoukoli hodnotu x jsou dvě možné hodnoty, které odpovídají křivce, jedna na každé straně osy x.

Z r počítáme také r^{-1}, což je k r inverzní číslo vzhledem k operaci násobení.

Nakonec vypočítáme z, což je n nejnižších bitů haše zprávy, kde n je řád eliptické křivky.

Dva možné veřejné klíče jsou pak:

```
<ul class="simplelist">
<li><em>K</em><sub>1</sub> = <em>r</em><sup>&#x2013;1</sup> (<em>sR</em>
&#x2013; <em>zG</em>)</li>
</ul>
```

a:

```
<ul class="simplelist">
<li><em>K</em><sub>2</sub> = <em>r</em><sup>&#x2013;1</sup>
(<em>sR</em><sup>'</sup> &#x2013; <em>zG</em>)</li>
</ul>
```

kde:

- K_1 a K_2 jsou dvě možnosti veřejného klíče podpisovatele.
- r^{-1} je k hodnotě podpisu r inverzní číslo vzhledem k operaci násobení.
- s je hodnota podpisu s.
- R a R' jsou dvě možnosti pro dočasný veřejný klíč Q.
- z je n nejnižších bitů haše zprávy.
- G je generátorový bod eliptické křivky.

Aby byly věci efektivnější, podpis transakce obsahuje hodnotu předpony v, která nám říká, která ze dvou možných hodnot R je dočasný veřejný klíč. Pokud je v sudé, pak R je správná hodnota. Pokud je v liché, pak je to R'. Tímto způsobem musíme vypočítat pouze jednu hodnotu pro R a pouze jednu hodnotu pro K.

Oddělení podpisu a přenosu (offline podpis)

Jakmile je transakce podepsána, je připravena k přenosu do sítě Ethereum. Tři kroky vytvoření, podepisování a odeslání transakce se obvykle dějí jako jedna operace, například pomocí web3.eth.sendTransaction. Jak jste však viděli v Tvorba a podpisování surových transakcí můžete transakci vytvořit a podepsat ve dvou samostatných krocích. Jakmile máte podepsanou transakci, můžete ji odeslat pomocí web3.eth.sendSignedTransaction, který hexadecimálně zakódovanou a podepsanou transakci a odešle do sítě Ethereum.

Proč byste chtěli oddělit podpis a přenos transakcí? Nejčastějším důvodem je bezpečnost. Počítač, který podepisuje transakci, musí mít odemčené soukromé klíče načtené v paměti. Počítač, který provádí přenos, musí být připojen k internetu (a musí mít spuštěn Ethereum klienta). Pokud jsou tyto dvě funkce na jednom počítači, máte v online systému soukromé klíče, což je docela nebezpečné. Oddělení funkcí podepisování a přenosu a provádění na různých počítačích (v offline a online zařízení) se nazývá *offline podpis* a je běžnou bezpečnostní praxí.

Offline podpis Ethereum transakcí ukazuje proces:

1. Vytvořte nepodepsanou transakci v online počítači, kde lze získat aktuální stav účtu, zejména aktuální nonce a dostupné prostředky.
2. Převeďte nepodepsanou transakci do offline zařízení s „vzduchovou mezerou“ za účelem podepsání transakce, např. Prostřednictvím QR kódu nebo na USB flash disku.
3. Přeneste podepsanou transakci (zpět) do online zařízení pro její zaslání do Ethereum sítě, např. prostřednictvím QR kódu nebo USB flash disku.

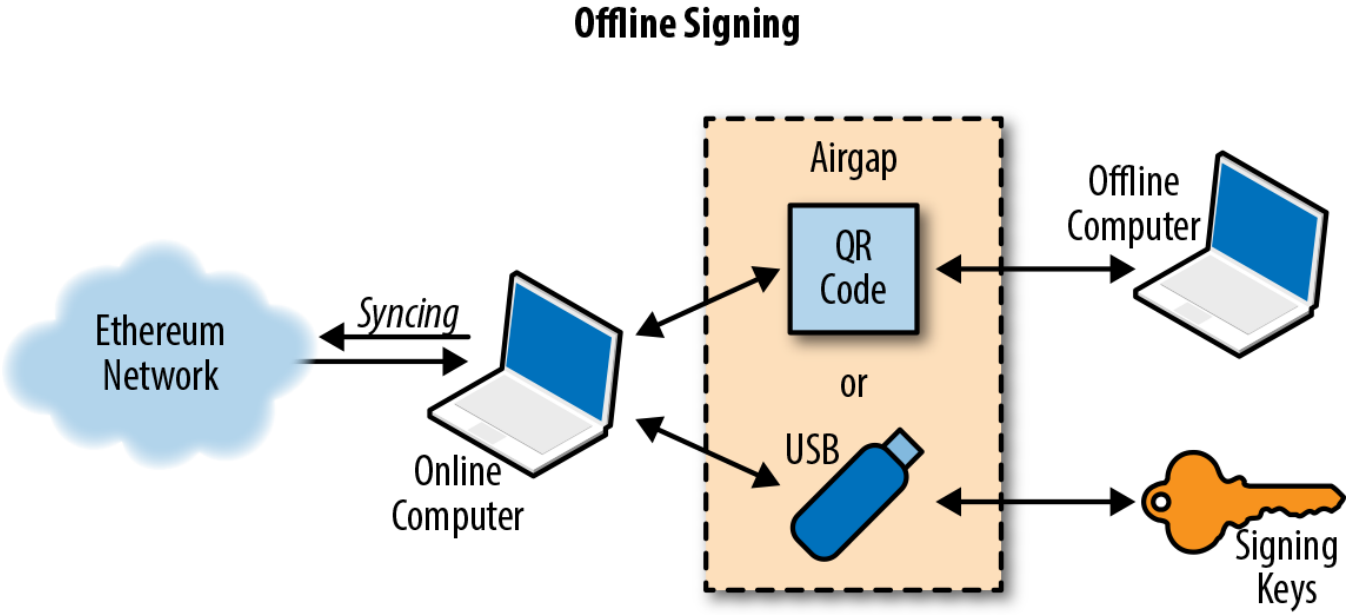

Figure 37. Offline podpis Ethereum transakcí

V závislosti na úrovni zabezpečení, kterou potřebujete, může váš počítač pro „offline podepisování“ mít různé stupně oddělení od online počítače. Tyto možnosti mohou být od izolované a firewallem oddělené podsítě (online, ale oddělené) až po zcela offline systém známý jako systém *se vzdušnou*

mezerou. V systému se vzduchovou mezerou neexistuje vůbec žádná síťová konektivita - počítač je oddělen od online prostředí mezerou „vzduchu“. Chcete-li podepsat transakce, převeďte je do a z počítače se vzduchovou mezerou pomocí média pro ukládání dat nebo (lépe) webové kamery a QR kódu. To samozřejmě znamená, že musíte ručně převést každou transakci, kterou chcete podepsat, a to se nemění.

I když ne mnoho prostředí může využívat plně vzduchové mezery, i malý stupeň izolace má významné bezpečnostní výhody. Například izolovaná podsíť s bránou firewall, která umožňuje pouze protokol fronty zpráv, může nabídnout mnohem menší prostor pro útok a mnohem vyšší zabezpečení než podepisování v online systému. Mnoho společností používá pro tento účel protokol, jako je ZeroMQ (0MQ). S takovým nastavením jsou transakce zformátovány a zařazeny do fronty k podpisu. Protokol fronty přenáší zformátovanou zprávu podobným způsobem jako TCP soket do podepisovacího počítače. Podepisující počítač přečte naformátované transakce z fronty (opatrně), použije podpis s příslušným klíčem a umístí je do odchozí fronty. Odchozí fronta odešle podepsané transakce do počítače s Ethereum klientem, který je vyzvedne z fronty a odešle je.

Propagace transakcí

Ethereum síť používá protokol „směrování vlnou“. Každý Ethereum klient funguje jako *uzel* v síti typu_peer-to-peer (P2P) , *která (ideálně) tvoří síť _smíšené typologie*. Žádný síťový uzel není zvláštní: všichni fungují jako rovnocenní kolegové. Termín „uzel“ budeme používat k označení Ethereum klienta, který je připojen k síti P2P a účastní se jí.

Šíření transakcí začíná původním Ethereum uzlem, který vytváří (nebo přijímá z offline) podepsanou transakci. Transakce je ověřena a poté přenesena do všech ostatních uzlů Ethereum, které jsou *přímo* připojeny k původnímu uzlu. Každý Ethereum uzel udržuje v průměru připojení k nejméně 13 dalším uzlům nazývaným jeho *sousedé*. Každý sousední uzel transakci ověří, jakmile ji obdrží. Pokud se shodnou na tom, že je platná, uloží kopii a rozšíří ji všem svým sousedům (kromě toho, ze které transakce pochází). V důsledku toho se transakce šíří ve vlně směrem ven z původního uzlu *zaplavuje* celou síť, dokud všechny uzly v síti nemají kopii transakce. Uzly mohou filtrovat zprávy, které šíří, ale výchozí je šířit všechny platné transakční zprávy, které obdrží.

Během několika sekund se Ethereum transakce rozšíří do všech Ethereum uzlů po celém světě. Z pohledu každého uzlu není možné rozeznat původ transakce. Soused, který ji poslal do uzlu, může být původcem transakce nebo ji mohl obdržet od jednoho ze svých sousedů. Aby mohl útočník sledovat původ transakcí nebo zasahovat do šíření, musel by ovládat významné procento všech uzlů.

To je součástí návrhu zabezpečení a ochrany soukromí v P2P sítích, zejména v případě bločenkových sítí.

Zaznamenání na bločence

Zatímco všechny Ethereum uzly jsou si rovnoprávné, některé z nich jsou provozovány *těžaři* a zajišťují transakce a bloky pro *těžební farmy*, což jsou počítače s vysoce výkonnými grafickými procesory (GPU). Těžební počítače přidávají transakce do kandidátských bloku a pokoušejí se najít *důkaz prací*, který učiní kandidátských blok platným blokem. Budeme o tom diskutovat podrobněji v Konsensus.

Aniž bychom šli do příliš mnoha podrobností, budou platné transakce nakonec zahrnuty do bloku transakcí, a tedy zaznamenány do Ethereum bločenky. Jakmile se vytěží blok s touto transakcí, transakce také upravují stav jednoinstančního Etherea, a to buď změnou zůstatku na účtu (v případě jednoduché platby), nebo vyvoláním kontraktů, které mění jejich vnitřní stav. Tyto změny se zaznamenávají spolu s transakcí ve formě *účtenky* transakce, která může také zahrnovat *události*. To vše podrobněji prozkoumáme v Ethereum virtuální stroj .

Transakce, která dokončila svou cestu od stvoření přes podpis EOA, propagaci a konečně i těžbu, změnila stav jednoinstančního Etherea a zanechala nesmazatelnou značku na bločence.

Transakce s více podpisy (vícepodpisové)

Pokud jste obeznámeni s možnostmi Bitcoin skriptování, víte, že je možné vytvořit bitcoinový vícepodpisový (multisig) účet, který může utratit finanční prostředky pouze tehdy, když transakci podepíše více stran (např. 2 ze 2 nebo 3 ze 4 podpisů). Základní transakce EOA společnosti Ethereum neobsahují ustanovení pro více podpisů; libovolná omezení podepisování však mohou být vynucena chytrými kontrakty za jakýchkoli podmínek, na které si vzpomenete, za účelem převodu etheru i tokenů.

Aby bylo možné tuto schopnost využít, musí být ether převeden na „peněženkových kontrakt“, který je naprogramován podle požadovaných pravidel utrácení, jako jsou požadavky na více podpisů nebo limity utrácení (nebo kombinace obou). Peněženkový kontrakt poté odešle finanční prostředky, pokud je k tomu vyzván autorizovaným EOA, jakmile jsou splněny podmínky pro jejich odeslání. Chcete-li například chránit ether pomocí vícenásobného podpisu, převeďte ether do vícepodpisového kontraktu. Kdykoli budete chtít poslat finanční prostředky na jiný účet, všichni

oprávnění uživatelé budou muset poslat potvrzující transakci kontraktu pomocí běžné peněženkové aplikace, což účinně opravňuje kontrakt k provedení transakce.

Tyto kontrakty mohou být také navrženy tak, aby vyžadovaly více podpisů před provedením místního kódu nebo ke spuštění jiných kontraktů. Zabezpečení systému je jednoznačně určeno kódem vícepodpisového kontraktu.

Schopnost realizovat transakce s více podpisy jako chytrého kontraktu demonstruje flexibilitu Etherea. Jde však o dvojsečný meč, protože mimořádná flexibilita může vést k chybám, které podrývají zabezpečení schémat s více podpisy. Ve skutečnosti existuje řada návrhů na vytvoření vícepodpisových příkazů v EVM, které by odstraňovaly potřebu chytrých kontraktů, alespoň u jednoduchých vícepodpisových schémat M-z-N. To by bylo rovnocenné s Bitcoinovým vícepodpisovým systémem, který je součástí základních konsensuálních pravidel a prokázal se jako robustní a bezpečný.

Závěry

Transakce jsou výchozím bodem každé činnosti v Ethereum systému. Transakce jsou „vstupy“, které způsobují, že virtuální Ethereum počítač vyhodnocuje kontrakty, aktualizuje zůstatky a obecněji upravuje stav Ethereum bločenky. Dále budeme pracovat s chytrými kontrakty mnohem podrobněji a naučíme se, jak programovat v kontraktově orientovaném programovacím jazyce Solidity

Chytré kontrakty a Solidity

Jak jsme diskutovali v Základy Etherea existují dva různé typy v Ethereum účtů: Externě vlastněné účty (EOA) a účty kontraktů. EOA jsou ovládány uživateli, často prostřednictvím softwaru, jako je peněženková aplikace, která je externí pro platformu Ethereum. Na rozdíl od toho jsou účty kontraktů kontrolovány programovým kódem (také běžně označovaným jako „chytrý kontrakt“), které provádí Ethereum virtuální počítač. Stručně řečeno, EOA jsou jednoduché účty bez přidruženého kódu nebo datového úložiště, zatímco účty kontraktů mají přidružený kód i datové úložiště. EOA jsou kontrolovány transakcemi vytvořenými a kryptograficky podepsanými soukromým klíčem v „reálném světě“ vnějším a nezávislém na protokolu, zatímco účty kontraktů nemají soukromé klíče, a tak se „kontrolují samy“ předem stanoveným způsobem, který stanoví kód jejich chytrého kontraktu. Oba typy účtů jsou identifikovány Ethereum adresou. V této kapitole budeme diskutovat účty kontraktů a programový kód, který je řídí.

Co je to chytrý kontrakt?

Termín *chytrý kontrakt* se v průběhu let používá k popisu široké škály různých věcí. V devadesátých letech kryptograf Nick Szabo vytvořil tento pojem a definoval ho jako „soubor příslibů specifikovaných v digitální podobě, včetně vnitřních protokolů, kterými strany plní přísliby.“ Od té doby se koncept chytrých kontraktů vyvinul, zejména po zavedení decentralizovaných bločenkových platforem s vynálezem Bitcoinu v roce 2009. V kontextu Etherea je tento termín ve skutečnosti trochu mylný, vzhledem k tomu, že chytrý kontrakty v Ethereu nejsou ani chytré ani zákonné smlouvy, ale termín se uchytil. V této knize používáme termín „chytré kontrakty“ pro neměnné počítačové programy, které fungují deterministicky v kontextu Ethereum virtuálního počítače jako součást Ethereum síťového protokolu - tj. Na světovém decentralizovaném Ethereum počítači.

Rozbalme tuto definici:

Počítačové programy

> Chytré kontrakty jsou jednoduše počítačové programy. Slovo „kontrakt“ nemá v této souvislosti právní význam.

Neměnnost

> Po nasazení se kód chytrého kontraktu nemůže změnit. Na rozdíl od tradičního softwaru je jediným způsobem, jak změnit chytrý kontrakt, nasazení nové instance.

Determinismus

Výsledek provedení chytrého kontraktu je stejný pro všechny, kteří ho provedou, vzhledem k danému kontextu transakce, která zahájila jeho provedení, a stavu Ethereum bločenky v okamžiku provedení.

Kontext EVM

Chytré kontrakty pracují ve velmi omezeném kontextu. Mají přístup ke svému vlastnímu stavu, kontextu transakce, která je zavolala, a k některým informacím o posledních blocích.

Decentralizovaný světový počítač

EVM běží jako lokální instance v každém Ethereum uzlu, ale protože všechny instance EVM pracují se stejném počátečním stavu a produkují stejný konečný stav, systém jako celek funguje jako jediný „světový počítač“.

Životní cyklus chytrého kontraktu

Chytré kontrakty jsou obvykle psány ve vysokoúrovňovém jazyce, jako je Solidity. Aby však mohly být spuštěny, musí být kompilovány do nízkoúrovňového bajtkódu, který běží v EVM. Po kompilaci jsou nasazeny na Ethereum platformě pomocí speciální *kontrakt vytvářející* transakce, která je jako taková identifikována odesláním na zvláštní adresu pro vytvoření kontraktu, jmenovitě 0x0 (viz Zvláštní transakce: Vytvoření kontraktu. Každý kontrakt je označen Ethereum adresou, která je odvozena od transakce, které kontrakt vytvořila, jako funkce účtu, ze kterého byla tato transakce odeslána, a transakční nonce. Ethereum Adresa kontraktu může být použita v transakci jako příjemce, pro zasílání prostředků na kontrakt nebo vyvolání jedné z funkcí kontraktu. Upozorňujeme, že na rozdíl od EOA neexistují žádné klíče spojené s účtem vytvořeným pro nový chytrý kontrakt. Jako tvůrce kontraktu nemáte na úrovni protokolu žádná zvláštní oprávnění (i když je můžete explicitně naprogramovat v kódu chytrého kontraktu). Určitě nedostanete soukromý klíč k účtu kontraktu, který ve skutečnosti neexistuje - můžeme říci, že účet chytrého kontraktu vlastní sám sebe.

Důležité je, že *kontrakty jsou spuštěny pouze tehdy, pokud jsou volány transakcí*. Všechny chytré kotrakty v Ethereu jsou v konečném důsledku prováděny z důvodu jejich zahájeni transakcí vyvolanou EOA. Kontrakt může volat jiný kontrakt, který může volat další kontrakt atd., Ale první kontrakt v takovém řetězci provedení bude vždy vyvolán transakcí z EOA. Kontrakty nikdy neběží „samostatně“ nebo „na pozadí“. Kontrakty ve skutečnosti spočívají v klidu, dokud je transakce nezačne vykonávat, ať už přímo nebo nepřímo jako součást řetězce volání kontraktů. Rovněž stojí za

zmínku, že chytré kontrakty nejsou v žádném smyslu prováděny „paralelně“ - světový počítač Ethereum lze považovat za jednovláknový stroj.

Transakce jsou *atomické*, bez ohledu na to, kolik kontraktů volají nebo co tyto kontrakty dělají při volání. Transakce se provádějí v plném rozsahu a veškeré změny v globálním stavu (kontrakty, účty atd.) se zaznamenávají pouze v případě, že celé vykonání kontraktu se úspěšně ukončí. Úspěšné ukončení znamená, že program byl vykonán bez chyby a dosáhl konce provádění. Pokud provádění selže z důvodu chyby, všechny jeho efekty (změny stavu) jsou „vráceny zpět“, jako by transakce nikdy neproběhla. Neúspěšná transakce je stále zaznamenána jako pokus a ether utracený za plyn na její provedení se odečte od původního účtu, ale jinak nemá žádný další dopad na stav kontraktu nebo účtu.

Jak již bylo uvedeno výše, je důležité si uvědomit, že kód kontraktu nelze změnit. Kontrakt však může být „odstraněn“, odstraněním kódu a jeho vnitřního stavu (úložiště) z jeho adresy, přičemž zůstane prázdný účet. Žádné transakce odeslané na tuto adresu účtu po smazání kontraktu nevedou k provedení kódu, protože již neexistuje žádný kód, který by se měl provádět. Chcete-li smazat kontrakt, provedete instrukci EVM nazvanou SELFDESTRUCT (dříve nazývanou SUICIDE). Tato operace stojí „negativní plyn“, navrácení plynu, čímž se stimuluje uvolnění síťových klientských zdrojů z vymazání uloženého stavu. Smazání kontraktu tímto způsobem neodstraní historii transakcí (minulost) kontraktu, protože samotná bločenka je neměnná. Je také důležité si uvědomit, že instrukce SELFDESTRUCT bude k dispozici, pouze pokud autor kontraktu naprogramoval chytrý kontrakt tak, aby tuto funkčnost měla. Pokud kód kontraktu neobsahuje instrukci SELFDESTRUCT nebo je tato instrukce nepřístupná, chytrý kontrakt nelze odstranit.

Úvod do vysokoúrovňových Ethereum jazyků

EVM je virtuální stroj, který spouští speciální formu kódu s názvem *EVM bajtkód*, analogický s procesorem počítače, který spouští strojový kód, například x86_64. Budeme zkoumat fungování a jazyk EVM mnohem podrobněji v Ethereum virtuální stroj . V této části se podíváme na to, jak jsou psány chytré kontrakty, aby mohly být spuštěny na EVM.

I když je možné programovat chytré kontrakty přímo v bajtkódu, EVM bajtkód je poněkud těžkopádný a pro programátory je velmi obtížné ho číst a porozumět mu. Místo toho většina Ethereum vývojářů používá k psaní programů vysokoúrovňové jazyky a kompilátor, který je převádí na bajtkód.

Zatímco jakýkoli vysokoúrovňový jazyk by mohl být upraven tak, aby v něm šlo psát chytré

kontrakty, přizpůsobení libovolného jazyka tak, aby byl kompatibilní s EVM bajtkódem, je tp docela těžkopádné cvičení a obecně by to vedlo k určitému množství nejasností. Chytré kontrakty fungují ve vysoce omezeném a minimalistickém běhovém prostředí (EVM). Kromě toho musí být k dispozici speciální sada systémových proměnných a funkcí EVM. Z tohoto důvodu je snazší vytvořit jazyk chytrých kontraktů od nuly, než upravit obecný jazyk, aby byl vhodný pro psaní chytrých kontraktů. V důsledku toho se objevilo mnoho speciálních jazyků pro programování chytrých kontraktů. Ethereum má několik takových jazyků, spolu s kompilátory potřebnými k vytvoření bajtkódu spustitelného EVM.

Obecně lze programovací jazyky rozdělit do dvou širokých programovacích paradigmat: *deklarativní* a *imperativní*, také známý jako *funkcionální* a *procedurální*. V deklarativním programování píšeme funkce, které vyjadřují *logiku* programu, ale ne jeho *tok*. Deklarativní programování se používá k vytváření programů, ve kterých neexistují žádné vedlejší efekty, což znamená, že mimo funkci nejsou žádné změny stavu. Mezi deklarativní programovací jazyky patří Haskell a SQL. Naopak, v imperativním programování programátor píše sadu funkcí, které kombinují logiku a tok programu. Mezi imperativní programovací jazyky patří C ++ a Java. Některé jazyky jsou „hybridní“, což znamená, že podporují deklarativní programování, ale lze je také použít k vyjádření imperativního programovacího paradigmatu. Mezi takové hybridy patří Lisp, JavaScript a Python. Obecně lze jakýkoli imperativní jazyk použít k napsání deklarativního paradigmatu, ale často to vede k nepovolenému kódu. Pro srovnání, čisté deklarativní jazyky nelze použít k psaní v imperativním paradigmatu. V ryze deklarativních jazycích *neexistují žádné „proměnné“*.

Zatímco imperativní programování je častěji používáno programátory, může být velmi obtížné psát programy, které se provádějí _ přesně podle očekávání_. Schopnost kterékoli části programu změnit stav jakékoli jiné části znesnadňuje uvažování o provádění programu a přináší mnoho příležitostí pro chyby. Deklarativní programování ve srovnání usnadňuje pochopení toho, jak se bude program chovat: protože nemá vedlejší účinky, lze jakoukoli část programu chápat izolovaně.

Ve chytrých kontraktech chyby stojí peníze, a to doslovně. Výsledkem je, že je velmi důležité psát chytré kontrakty bez nezamýšlených účinků. Chcete-li to provést, musíte být schopni jasně uvažovat o očekávaném chování programu. Deklarativní jazyky tedy hrají v chytrých kontraktech mnohem větší roli než v univerzálním softwaru. Jak však uvidíte, nejpoužívanější jazyk pro chytré kontrakty (Solidity) je imperativní. Programátoři, stejně jako většina lidí, odolávají změnám!

Aktuálně podporované vysokoúrovňové programovací jazyky pro chytré kontrakty zahrnují (seřazené podle přibližného věku):

LLL

Funkcionální (deklarativní) programovací jazyk se syntaxí typu Lisp. Byl to první vysokoúrovňový jazyk pro Ethereum chytré kontrakty, ale dnes se používá jen zřídka.

Serpent

Procedurální (imperativní) programovací jazyk se syntaxí podobnou Pythonu. Lze ho také použít k zápisu funkcionálního (deklarativního) kódu, ačkoli to není zcela bez vedlejších účinků.

Solidity

Procedurální (imperativní) programovací jazyk se syntaxí podobnou JavaScriptu, C ++ nebo Java. Nejoblíbenější a často používaný jazyk pro Ethereum chytré kontrakty.

Vyper

Nověji vyvinutý jazyk, podobný Serpentu a opět s Pythonovou syntaxí. Zamýšlel se přiblížit k čistě funkčnímu jazyku Pythonu, na rozdíl od Serpentu, ale nikoli nahradit Serpent.

Bamboo

Nově vyvinutý jazyk, ovlivněný Erlangem, s explicitními stavovými přechody a bez iteračních toků (smyček). Účelem je snížit nežádoucí účinky a zvýšit ověřitelnost správnosti. Velmi nové, ale již široce přijaté.

Jak vidíte, existuje mnoho jazyků, z nichž si můžete vybrat. Avšak ze všech je zdaleka nejoblíbenější Solidity, a to do té míry, že se jedná o *fakticky* vysokoúrovňový jazyk Etherea a dokonce i dalších EVM podobných bločenek. Většinu času strávíme používáním Solidity, ale prozkoumáme také některé příklady v jiných vysokoúrovňových jazycích, abychom pochopili jejich různé filozofie.

Tvorba chytrého kontraktu pomocí Solidity

Solidity byl vytvořen Dr. Gavinem Woodem (spoluautor této knihy) jako jazyk výslovně pro psaní chytrých kontraktů s funkcemi přímo podporujícími provádění v decentralizovaném prostředí světového počítače Ethereum. Výsledné atributy jsou celkem obecné, a tak se nakonec použilo pro programování chytrých kontraktů na několika dalších bločenkových platformách. Mezi vývojáře Solidity patřili Christian Reitiwessner a poté také Alex Beregszaszi, Liana Husikyan, Yoichi Hirai a několik bývalých hlavních Ethereum přispěvatelů. Solidity je nyní vyvíjena a udržována jako nezávislý projekt na GitHubu [https://github.com/ethereum/solidity].

Hlavním „produktem“ projektu Solidity je kompilátor Solidity solc, který převádí programy psané v jazyce Solidity na EVM bajtkód. Projekt také řídí důležitý standard aplikačního binárního rozhraní (ABI) pro Ethereum chytré kontrakty, který podrobně prozkoumáme v této kapitole. Každá verze kompilátoru Solidity odpovídá a zkompiluje konkrétní verzi Solidity jazyka.

Nejprve si stáhneme binární spustitelný soubor kompilátoru Solidity. Poté vytvoříme a sestavíme jednoduchý kontrakt, navazující na příklad, který jsme začínali v Základy Etherea.

Výběr verze Solidity

Solidity následuje model verzí nazvaný *sémantické verzování* [https://semver.org/], který uvádí čísla verzí strukturovaná jako tři čísla oddělená tečkami: *hlavní.vedlejší.záplata* . „Hlavní“ číslo se zvyšuje při velkých a _ zpětně nekompatibilních_ změnách, „vedlejší“ číslo je zvýšeno při přidání zpětně kompatibilní funkce do současné hlavní verze a číslo „záplaty“ je zvyšováno při zpětně kompatibilní opravě chyb.

V době psaní této knihy je Solidity ve verzi 0.4.24. Pravidla pro hlavní verzi 0, která je počáteční vývoj projektu, se liší: cokoli se může kdykoli změnit. V praxi Solidity zachází s „vedlejším“ číslem, jako by to byla hlavní verze, a s číslem „záplaty“, jako by to byla vedlejší verze. Proto je v 0.4.24, 4 považována za hlavní verzi a 24 za vedlejší verzi.

V nejbližší době se očekává vydání hlavní verze Solidity 0.5.

Jak jste viděli v Základy Etherea mohou vaše Solidity programy obsahovat direktivu pragma, která stanoví minimální a maximální verze Solidity, s nimiž je kompatibilní, a lze je použít k sestavení vašeho kontraktu.

Protože se Solidity rychle vyvíjí, je často lepší nainstalovat nejnovější verzi.

Stáhnutí a instalace

Existuje několik metod, které můžete použít ke stažení a instalaci Solidity, buď jako binární vydání, nebo kompilaci ze zdrojového kódu. Podrobné pokyny naleznete v Solidity dokumentaci [http://bit.ly/2RrZmup].

Zde je návod, jak nainstalovat nejnovější binární vydání Solidity do operačního systému Ubuntu / Debian pomocí správce balíčků apt:

```
<pre data-type="programlisting">
$ <strong>sudo add-apt-repository ppa:ethereum/ethereum</strong>
$ <strong>sudo apt update</strong>
$ <strong>sudo apt install solc</strong>
</pre>
```

Po instalaci solc zkontrolujte verzi spuštěním:

```
<pre data-type="programlisting">
$ <strong>solc --version</strong>
solc, the solidity compiler commandline interface
Version: 0.4.24+commit.e67f0147.Linux.g++
</pre>
```

Existuje řada dalších způsobů, jak nainstalovat Solidity, v závislosti na operačním systému a požadavcích, včetně přímého kompilace ze zdrojového kódu. Další informace naleznete na adrese https://github.com/ethereum/solidity [].

Vývojové prostředí

Pro vývoj v Solidity můžete použít libovolný textový editor a solc na příkazovém řádku. Možná však zjistíte, že některé textové editory určené pro vývoj, jako jsou Emacs, Vim a Atom, nabízejí další funkce, jako je zvýraznění syntaxe a makra, která usnadňují vývoj v Solidity.

Existují také webová vývojová prostředí, například Remix IDE [https://remix.ethereum.org/] a EthFiddle [https://ethfiddle.com/].

Použijte nástroje, díky nimž budete produktivní. Nakonec jsou programy Solidity pouhé textové soubory. Zatímco vyspělé editory a vývojová prostředí mohou věci usnadnit, nepotřebujete nic jiného než jednoduchý textový editor, jako je nano (Linux / Unix), TextEdit (macOS) nebo dokonce NotePad (Windows). Jednoduše uložte zdrojový kód programu s příponou *.sol* a kompilátor Solidity jej rozpozná jako Solidity program.

Psaní jednoduchého programu v Solidity

V Základy Etherea jsme napsali náš první Solidity program. Když jsme poprvé vytvořili kontrakt

kohoutek (Faucet), použili jsme Remix IDE ke kompilaci a nasazení kontraktu. V této sekci provedeme revizi, vylepšení a ozdobení kohoutku.

Náš první pokus vypadal jako Faucet.sol: Chytrý kontrakt Kohoutek v Solidity.

Example 2. Faucet.sol: Chytrý kontrakt Kohoutek v Solidity

```
// Version of Solidity compiler this program was written for
pragma solidity ^0.6.0;

// Our first contract is a faucet!
contract Faucet {
    // Accept any incoming amount
    receive () external payable {}

    // Give out ether to anyone who asks
    function withdraw(uint withdraw_amount) public {

        // Limit withdrawal amount
        require(withdraw_amount <= 100000000000000000);

        // Send the amount to the address that requested it
        msg.sender.transfer(withdraw_amount);
    }
}
```

Kompilace pomocí Solidity kompilátoru (solc)

Nyní budeme použijte přímo Solidity kompilátor z příkazové řádky k přímému sestavení našeho programu. Solidity kompilátor solc nabízí celou řadu možností, které můžete vidět předáním parametru --help.

Používáme + solc + parametry --bin a --optimize k vytvoření optimalizovaného binárního souboru našeho vzorového kontraktu:

```
<pre data-type="programlisting">
$ <strong>solc --optimize --bin Faucet.sol</strong>
======= Faucet.sol:Faucet =======
Binárn¬:
6060604052341561000f57600080fd5b60cf8061001d6000396000f3006060604052600436
10603e5
763ffffffff7c0100000000000000000000000000000000000000000000000000000600
0350416
632e1a7d4d81146040575b005b3415604a57600080fd5b603e60043567016345785d8a0000
8111156
06357600080fd5b73ffffffffffffffffffffffffffffffffffffffff331681156108fc028
2604051
600060405180830381858888f1935050505050151560a057600080fd5b505600a165627a7a72
3058203
556d79355f2da19e773a9551e95f1ca7457f2b5fbbf4eacf7748ab59d2532130029
</pre>
```

Výsledkem, který produkuje solc, je hexadecimálně kódovaný binární kód, který lze odeslat doEthereum bločenky.

ABI Ethereum kontraktu

V počítačovém softwaru, *binární rozhraní aplikace* (ABI) je rozhraní mezi dvěma programovými moduly; často mezi operačním systémem a uživatelskými programy. ABI definuje způsob přístupu k datovým strukturám a funkcím ve *strojovém kódu*; to by nemělo být zaměňováno s API, které definuje tento přístup ve vysoce kvalitních, často lidsky čitelných formátech, jako *zdrojovém kódu*. ABI je tedy primárním způsobem kódování a dekódování dat do a ze strojového kódu.

V Ethereum se ABI používá pro kódování volání kontraktů pro EVM a pro čtení dat z transakcí. Účelem ABI je definovat funkce v kontraktu, které lze vyvolat, a popsat, jak každá funkce přijme parametry a vrátí svůj výsledek.

ABI kontraktu je specifikováno jako JSON pole popisů funkcí (viz Funkce) a události (viz Události. Popis funkce je JSON objekt s poli type`, `name`, `inputs`, `outputs`, `constant`, and `payable`. Objekt popisu události obsahuje pole `type`, `name`, `inputs`, and `anonymous`.

Pomocí Solidity kompilátoru solc z příkazové řádky vytvoříme ABI pro náš *Faucet.sol* příklad kontraktu:

```
<pre data-type="programlisting">
$ <strong>solc --abi Faucet.sol</strong>
======= Faucet.sol:Faucet =======
JSON ABI kontraktu
[{"constant":false,"inputs":[{"name":"withdraw_amount","type":"uint256"}],
\
"name":"withdraw","outputs":[],"payable":false,"stateMutability":"nonpayab
le", \
"type":"function"},{"payable":true,"stateMutability":"payable", \
"type":"fallback"}]
</pre>
```

Jak vidíte, kompilátor vytvoří JSON pole popisující dvě funkce, které jsou definovány *Faucet.sol*. Tato JSON data může použít jakákoliv aplikace, která chce po nasazení získat přístup ke kontraktu Faucet. Pomocí ABI může aplikace, jako je peněženka nebo prohlížeč DApp, vytvářet transakce, které volají funkce v Faucet se správnými parametry a jejich typy. Například peněženka by měla vědět, že pro volání funkce withdraw by musela poskytnout parametr uint256 s názvem withdraw_amount. Peněženka může uživatele vyzvat k zadání této hodnoty, poté vytvořit transakci, která ji zakóduje a provede funkci withdraw.

Vše, co aplikace potřebuje k interakci s kontrraktem, je ABI a adresa, na které byl kontrakt nasazen.

Výběr Solidity kompilátoru a verze jazyka

Jak jsme viděli v předchozím kódu, náš kontrakt Faucet se úspěšně kompiluje s verzí Solidity 0.4.21. Ale co kdybychom použili jinou verzi kompilátoru Solidity? Jazyk je stále v neustálém pohybu a věci se mohou neočekávaně změnit. Náš kontrakt je poměrně jednoduchý, ale co když náš program použil funkci, která byla přidána pouze ve verzi Solidity 0.4.19 a pokusili jsme se ho zkompilovat s verzí 0.4.18?

K vyřešení těchto problémů nabízí Solidity *direktivu překladače* známou jako *pragma verze*, která instruuje kompilátoru, že program očekává konkrétní verzi kompilátoru (a jazyka). Podívejme se na příklad:

```
pragma solidity ^0.4.19;
```

Solidity Kompilátor přečte pragmu verze a vytvoří chybu, pokud je verze kompilátoru nekompatibilní s pragmou verze. V našem případě pragma naší verze říká, že tento program může být kompilován kompilátorem Solidity s minimální verzí 0.4.19. Symbol ^ však uvádí, že umožňujeme kompilaci s jakoukoli *vedlejší verzí* nad 0.4.19; např. 0.4.20, ale ne 0.5.0 (což je hlavní verze, nikoli menší verze). Pragma direktivy nejsou kompilovány do EVM bajtkódu. Kompilátor je používá pouze ke kontrole kompatibility.

Pojďme k našemu kontraktu Faucet přidat direktivu pragma. Nový soubor pojmenujeme *Faucet2.sol*, abychom udržovali přehled o změnách, které jsme udělali v tomto příkladu, počínaje Faucet2.sol: Přidání verze pragma do Faucet.

Example 3. Faucet2.sol: Přidání verze pragma do Faucet

```
// Version of Solidity compiler this program was written for
pragma solidity ^0.4.19;

// Our first contract is a faucet!
contract Faucet {

    // Give out ether to anyone who asks
    function withdraw(uint withdraw_amount) public {

        // Limit withdrawal amount
        require(withdraw_amount <= 100000000000000000);

        // Send the amount to the address that requested it
        msg.sender.transfer(withdraw_amount);
    }

    // Accept any incoming amount
    function () external payable {}

}
```

Přidání verze pragma je osvědčeným postupem, protože zabraňuje problémům s různými kompilátory a jazykovými verzemi. V této kapitole prozkoumáme další osvědčené postupy a budeme nadále zlepšujeme kontrakt Faucet.

Programování v Solidity

V této sekci se podíváme na některé schopnosti jazyka Solidity. Jak jsme zmínili v Základy Etherea, náš první příklad kontraktu byl velmi jednoduchý a také vadný různými způsoby. Zde to postupně zlepšíme a prozkoumáme, jak používat Solidity. Nebude to však komplexní výukový kurz Solidity, protože Solidity je poměrně složitý a rychle se vyvíjí. Pokryjeme základy a poskytneme vám dostatek základních dovedností, abyste si mohli zbytek prozkoumat sami. Dokumentaci Solidity naleznete na webu projektu [https://solidity.readthedocs.io/en/latest/].

Datové typy

Nejprve se podívejme na některé ze základních datových typů nabízených v Solidity:

Boolean (bool)

Logický typ, nabývá jedné ze dvou hodnot, pravda (true) nebo nepravda (false), s logickými operátory ! (negace), && (konjunkce; a), || (disjunkce; nebo), == (rovnost) a != (nerovnost).

Celé číslo (int, uint)

Celé číslo se znaménkem (int) a bez znaménka (+ uint) , deklarované v přírůstcích po 8 bitech od +int8 do uint256. Bez přípony velikosti se použije 256-bitová varianta, která odpovídají velikosti slova EVM.

Reálné číslo (fixed, ufixed)

Reálná čísla s pevnou desetinnou čárkou, deklarované (u)`fixed`*`M`*`x`*`N`* kde *M* je velikost v bitech (po násobcích 8 až do 256) a *N* je počet desetinných míst za desetinnou čárkou (až do 18), např. ufixed32x2.

Adresa

20-bajtová Ethereum adresa. Objekt address má mnoho užitečných členských funkcí, přičemž hlavními funkcemi jsou balance (vrací zůstatek na účtu) a pass: [<code>transfer</code>] (převádí ether na účet).

Bajtové pole (statické)

Pole bajtů pevné velikostideklarovaná pomocí bytes1 až bytes32.

Bajtové pole (dynamické)

Pole bajtů proměnlivé délky, deklarované pomocí bytes nebo string.

Enum

Uživatelem definovaný typ pro výčet diskrétních hodnot: enum název {hodnota1, hodnota2, ...}.

Obecné pole

Pole jakéhokoli typu, buď pevné nebo dynamické: uint32[][5] je pole pěti dynamických polí celých čísel bez znaménka.

Struktury

Uživatelem definované datové kontejnery pro seskupení proměnných: `struct název {typ1 prom¬nná1; typ2 prom¬nná2; ...}`.

Mapování

Hašovací vyhledávací tabulky pro dvojice *klíč* => *dvojici*: mapping(typ klíče => typ hodnoty) název.

Kromě těchto datových typů nabízí Solidity také řadu hodnotových konstant, které lze použít k výpočtu různých jednotek:

Časové jednotky

Jednotky sekundy (seconds), minuty (minutes), hodiny (hours), a dny (days) lze použít jako přípony a převést je na násobky základní jednotky seconds.

Ether units

Jednotky wei, finney, szabo, a ether lze použít jako přípony a převést je na násobky základní jednotky wei.

V našem příkladu Faucet jsme použili uint (což je jiné označení pro uint256) pro proměnnou withdraw_amount. Také jsme nepřímo použili proměnnou address, kterou jsme nastavili pomocí msg.sender. Více těchto datových typů použijeme v našich příkladech ve zbytku této kapitoly.

Použijte jeden z multiplikátorů jednotek ke zlepšení čitelnosti našeho příkladu kontraktu. Ve funkci withdraw omezíme maximální výběr, vyjádříme limit ve wei, základní jednotku etheru:

```
require(withdraw_amount <= 100000000000000000);
```

To není příliš snadné číst. Náš kód můžeme vylepšit pomocí multiplikátoru jednotek ether, abychom vyjádřili hodnotu v etheru namísto ve wei:

```
require(withdraw_amount <= 0.1 ether);
```

Předdefinované globální proměnné a funkce

Když je kontrakt prováděn v EVM, má přístup k malému množství globálních objektů. Patří sem objekty blok (block), zpráva (msg), a transakce (tx). Navíc Solidity nabízí řadu instrukcí EVM jako předdefinované funkce. V této části prozkoumáme proměnné a funkce, ke kterým máte přístup v rámci chytrých kontraktů v programu Solidity.

Kontext transakce / volání zprávy

Objekt zpráva (msg) jetransakční volání (vyvolané EOA) nebo volání zprávy (vyvolané kontraktem), které zahájilo vykonávání tohoto kontraktu. Obsahuje řadu užitečných atributů:

msg.sender

Tento už jsme použili. Představuje adresu, která zahájila toto volání kontraktu, ne nutně původní EOA, který transakci odeslal. Pokud byla naše smlouva vyvolána přímo EOA transakcí, jedná se o adresu, která transakci podepsala, jinak to bude adresa kontraktu.

msg.value

Hodnota etheru odeslaného tímto voláním (ve wei).

msg.gas

Množství plynu zbývající v zásobě pro toto prováděcí prostředí. Toto bylo označeno za zastaralé v Solidity v0.4.21 a nahrazeno funkcí gasleft.

msg.data

Užitečné datové zatížení této zprávy v našem kontraktu.

msg.sig

První čtyři bajty užitečného datového zatížení, což je označení funkce, která má být zavolána.

Kdykoli kontrakt volá jiný kontrakt, hodnoty všech atributů zprávy msg se změní, aby odrážely informace o novém volajícím. Jedinou výjimkou je funkce delegatecall, která spouští kód jiného kontraktu / knihovny v původním msg kontextu.

Kontext transakce

Objekt tx poskytuje prostředky pro přístup k informacím o transakci:

tx.gasprice

Cena plynu ve volající transakci.

tx.origin

Adresa EOA, který zavolal tuto transakci. VAROVÁNÍ: nebezpečné!

Kontext bloku

Objekt blok (block) obsahuje informace o aktuálním bloku:

block.blockhash(__blockNumber__)

Haš bloku zadaného čísla bloku, až 256 bloků do minulosti. Zastaralé a nahrazené funkcí blockhash v Solidity v0.4.22.

block.coinbase

Adresa příjemce transakčních poplatků a odměny za vytěžení aktuálního bloku.

block.difficulty: Obtížnost (důkazu práci) aktuálního bloku.

block.gaslimit

Maximální množství plynu, které lze utratit ve všech transakcích zahrnutých v aktuálním bloku.

block.number

Aktuální číslo bloku (výška bločenky).

block.timestamp

Časové razítko umístěné v aktuálním bloku těžařem (počet sekund od 1. ledna 1970).

adresa objektu

"address object")Jakákoli adresa, předaná jako vstup nebo získaná z objektu kontraktu, má řadu atributů a metod:

address.balance

Zůstatek adresy ve wei. Například aktuální zůstatek kontraktu je address(this).balance..

address.transfer(__amount__)

Převede částku (ve wei) na tuto adresu a vyvolá výjimku při jakékoli chybě. Tuto funkci jsme použili v našem příkladu Faucet jako metodu na adrese odesilatele msg.sender, jako msg.sender.transfer.

address.send(__amount__)

Podobně jako transfer, pouze místo vyvolání výjimky vrací false při chybě. VAROVÁNÍ: vždy zkontrolujte návratovou hodnotu send.

address.call(__payload__)

Funkce pro nízkoúrovňové volání CALL — může vytvořit libovolné volání zprávy s užitečným datovým zatížením. Vrací false při chybě. VAROVÁNÍ: nebezpečné - příjemce může (náhodně nebo úmyslně) spotřebovat veškerý plyn, což způsobí zastavení vašeho kontraktu s výjimkou OOG; vždy zkontrolujte návratovou hodnotu volání.

address.callcode(__payload__): Funkce pro nízkoúrovňové volání CALLCODE, obdobné jako address(this).call(...), ale kód kontraktu je nahrazen adresou address. Vrací false při chybě. VAROVÁNÍ: pouze pro pokročilé!

address.delegatecall()

Nízkoúrovňová funkce DELEGATECALL, jako callcode(...), ale s plným kontextem zprávy msg viděným aktuálním kontraktem. Vrací +false +při chybě. VAROVÁNÍ: pouze pro pokročilé!

Vestavěné funkce

Další funkce, které stojí za zmínku, jsou:

addmod, mulmod

Pro sčítání a násobení, na výsledek je aplikována operace zbytku po celočíselném dělení. Například addmod(x,y,k) počítá (x + y) % k

keccak256, sha256, sha3, ripemd160

Funkce pro výpočet hašů s různými standardními hačovacími algoritmy.

ecrecover

Obnoví adresu použitou k podepsání zprávy z podpisu.

selfdestruct(__recipient_address__)

Odstraní aktuální kontrakt a odešle zbývající ether v účtu na adresu příjemce.

this

Adresa účtu aktuálně prováděného kontraktu.

Definice kontraktu

Hlavní typ dat v Solidity je kontrakt; náš příklad Faucet jednoduše definuje objekt `kontraktu` . Podobně jako u jakéhokoli objektu v objektově orientovaném jazyce je kontrakt kontejnerem, který obsahuje data a metody.

Solidity nabízí dva další typy objektů, které jsou podobné kontraktu:

interface

Definice rozhraní je strukturována přesně jako kontrakt, kromě toho, že žádná z funkcí není definována, jsou pouze deklarovány. Tento typ deklarace se často nazývá *pahýl*; vysvětluje argumenty funkcí a typy návratových hodnot bez jakékoli implementace. Rozhraní specifikuje „tvar“ kontraktu; po zdědění musí každá z funkcí deklarovaných v rozhraním být definována v dítěti.

knihovna

Kontrakt typu knihovna je pouze jednou nasazen a následně je využíván jinými kontrakty pomocí metody delegatecall (viz adresa objektu).

Funkce

V rámci kontraktu definujeme funkce, které lze volat EOA transakcí nebo jiným kontraktem. V našem příkladu Faucet máme dvě funkce: withdraw a (nepojmenovaná) *nouzovou* funkce.

Syntaxe, kterou používáme k deklarování funkce v Solidity, je následující:

```
<pre data-type="programlisting">
function JménoFunkce ([<em>parametry</em>]) {public | private | internal |
external}
[pure|constant|view|payable] [<em>midifikátory</em>] [returns (<em>typ
návratové hodnoty</em>)]
</pre>
```

Podívejme se na každou z těchto složek:

NázevFunkce

Název funkce, která se používá k zavolání funkce pomocí transakce (z EOA), z jiné smlouvy nebo dokonce z téže smlouvy. Jedna funkce v každém kontraktu může být definována bez názvu, v tom případě je to funkce *nouzová*, která se volá, když není pojmenována žádná jiná funkce. Nouzová funkce nemůže mít žádné argumenty ani nic vrátit.

parametry

Za jménem funkce specifikujeme parametry, které musí být předány funkci, s jejich jmény a typy. V našem příkladu Faucet jsme definovali uint withdraw_amount jako jediný parametr funkce pass: [<code>withdraw</code>].

Následující sada klíčových slov (public, private, internal, external) určuje *viditelnost* funkce:

public

("public function" Veřejná (public) je výchozí nastavení; takové funkce mohou být vyvolány jinými kontrakty nebo transakcemi EOA, nebo z kontraktu. V našem příkladu Faucet jsou obě

funkce definovány jako veřejné.

external

Externí funkce jsou jako veřejné funkce, s výjimkou, že je nelze vyvolat v rámci kontraktu, pokud není výslovně uvedeno s předponou klíčovým slovem this.

internal

Vnitřní funkce jsou přístupné pouze z kontraktu - nemohou být vyvolány jinou smlouvou nebo EOA transakcí. Mohou být volány odvozenými kontrakty, které jsou potomky tohoto kontraktu. Obdoba klíčového slova protected z C++.

private

Soukromé funkce jsou jako vnitřní funkce, ale nelze volat odvozenými kontrakty.

Mějte na paměti, že pojmy *internal* a *private* jsou poněkud zavádějící. Jakákoli funkce nebo data uvnitř smlouvy jsou vždy _ viditelná_ na veřejné bločence, což znamená, že kdokoli může vidět kód nebo data. Zde popsaná klíčová slova ovlivňují pouze to, jak a kdy může být funkce *volána*.

Druhá sada klíčových slov (pure, constant, view, payable) ovlivňuje chování funkce:

constant or view

Funkce označená jako *view* slibuje, že neupravuje žádný stav. Termín *constant* je jiné označení pro *view*, které bude v budoucí verzi zastaralé. V tuto chvíli kompilátor nevynucuje modifikátor view, pouze vydává varování, ale očekává se, že _view_se stane vynucovaným klíčovým slovem v Solidity v0.5.

pure

Čistá funkce je taková funkce, která v paměti nečte ani nezapisuje žádné proměnné. Může pracovat pouze s argumenty a vracet data, bez odkazu na jakákoli uložená data. Účelem čistých funkcí je podpořit programování v deklarativním stylu bez vedlejších účinků nebo stavu.

payable

Platby přijímající funkce je taková funkce, která může přijímat příchozí platby. Funkce, které nejsou deklarovány jako payable, odmítnou příchozí platby. Existují dvě výjimky z důvodu návrhových rozhodnutí v EVM: mincetvorné (coinbase) platby a SELFDESTRUCT dědictví

budou vyplaceny, i když nouzová funkce není deklarována jako platby přijímající, ale to dává smysl, protože provádění kódu není stejně součástí těchto plateb.

Jak vidíte v našem příkladu Faucet, máme jednu platby přijímající funkci (nouzová funkce), což je jediná funkce, která může přijímat příchozí platby.

Konstruktor kontraktu a samozničení

Existuje speciální funkce, která se používá pouze jednou . Po vytvoření kontraktu se také spustí funkce *constructor*, pokud existuje, k počátečnímu nastavení stavu kontraktu. Konstruktor je spuštěn ve stejné transakci jako vytvoření kontraktu. Funkce konstruktoru je volitelná; všimnete si, že náš příklad Faucet ji nemá.

Konstruktory lze specifikovat dvěma způsoby. Až do a včetně Solidity v0.4.21 je konstruktor funkcí, jejíž název odpovídá názvu kontraktu, jak můžete vidět zde:

```
contract MEContract {
    function MEContract() {
        // This is the constructor
    }
}
```

Problém tohoto formátu spočívá v tom, že pokud se změní název kontraktu a název funkce konstruktoru se nezmění, nejedná se o konstruktor. Podobně, pokud dojde k náhodnému překlepu v pojmenování kontraktu nebo konstruktoru, funkce již nadále není konstruktorem. To může způsobit některé docela ošklivé, neočekávané a obtížně vyhledatelné chyby. Představte si například, že konstruktor nastavuje vlastníka kontrolu pro účely ovládání kontraktu. Pokud funkce ve skutečnosti není konstruktorem kvůli chybě pojmenování, zůstane majitel nejen v okamžiku vytvoření smlouvy nenastavený, ale funkce může být také nasazena jako trvalá a „volitelná“ částí kontraktu, jako například normální funkce, která umožňuje jakékoli třetí straně unést kontrakt a stát se je jeho „vlastníkem“ kdykoli po jeho vytvoření.

Aby bylo možné řešit potenciální problémy s funkcemi konstruktoru založenými na nutnosti stejného názvu jako je název kontraktu, Solidity v0.4.22 zavádí klíčové slovo constructor, které funguje jako funkce konstruktoru, ale nemá jméno. Přejmenování kontraktu nemá na konstruktor vůbec žádný vliv. Rovněž je snazší určit, která funkce je konstruktor. Vypadá to takto:

```
pragma ^0.4.22
contract MEContract {
    constructor () {
        // This is the constructor
    }
}
```

Stručně řečeno, životní cyklus kontraktu začíná kontrakt vytvářející transakcí zaslanou z EOA účtu nebo kontraktu. Pokud existuje konstruktor, je prováděn jako součást vytváření kontraktu, aby inicializoval stav kontraktu po jeho vytvoření. Konstruktor následně již nejde zavolat.

The other end of the contract's life cycle is *contract destruction*. Kontrakty jsou ničeny zvláštní EVM instrukcí s názvem SELFDESTRUCT. Dříve to bylo nazýváno `SUICIDE`, ale toto jméno bylo označené za zastaralé kvůli negativním asociacím slova. V programu Solidity je tato instrukce zpřístupněna na vyšší úrovni jako vestavěná funkce s názvem selfdestruct, která vyžaduje jeden parametr: adresu, na kterou má být zaslán zůstatek etheru na účtu kontraktu. Vypadá to takto:

```
selfdestruct(address recipient);
```

Note that you must explicitly add this command to your contract if you want it to be deletable—this is the only way a contract can be deleted, and it is not present by default. In this way, users of a contract who might rely on a contract being there forever can be certain that a contract can't be deleted if it doesn't contain a `SELFDESTRUCT` instrukci.

Přidání konstruktoru a sebezničení do našeho příkladu kohoutku

Příklad kontraktu Faucet jsme zavedli v Základy Etherea; nemá konstruktor ani selfdestruct funkci. Je to věčná smlouva, kterou nelze odstranit. Pojďme to změnit přidáním konstruktoru a selfdestruct funkce. Pravděpodobně chceme, aby selfdestruct byl zavolatelný *pouze* EOA, který původně vytvořil kontrakt. Obvykle je to podle konvence uloženo v adresové proměnné nazvané owner. Náš konstruktor nastaví proměnnou owner a funkce selfdestruct nejprve zkontroluje, zda ji volal přímo vlastník.

Nejprve náš konstruktor:

```
// Verze kompilátoru Solidity, pro který byl tento program napsán
pragma solidity ^0.4.22;

// Náš první kontrakt je kohoutek!
contract Faucet {

    address owner;

// Inicializovat kontraktu kohoutek: nastavit vlastníka
    constructor() {
        owner = msg.sender;
    }

[...]
```

Změnili jsme direktivu pragma tak, aby byla v tomto příkladu specifikována verze 0.4.22 jako minimální verze, protože používáme nové klíčové slovo constructor zavedené v Solidity v0.4.22. Náš kontrakt nyní obsahuje proměnnou typu adresa pojmenovanou owner. Jméno „vlastníka“ není nijak zvláštní proměnná. Tuto adresní proměnnou bychom mohli nazvat „brambor“ a stále ji používat stejným způsobem. Jméno owner jednodušeji objasní svůj účel.

Dále náš konstruktor, který běží jako součást transakce vytvoření smlouvy, přiřadí adresu z msg.sender k proměnné owner. Pro identifikaci iniciátora žádosti o výběr jsme použili funkci msg.sender ve funkci pass: [<code>withdraw</code>] V konstruktoru je však msg.sender EOA nebo adresa kontraktu, která iniciovala vytvoření kontraktu. Víme, že se jedná o tento případ, *protože* to je funkce konstruktoru: spustí se pouze jednou, během vytváření kontraktu.

Nyní můžeme přidat funkci zničení kontraktu. Musíme se ujistit, že tuto funkci může spustit pouze vlastník, takže k řízení přístupu použijeme příkaz require. Takto to bude vypadat:

```
// Zni¬ení kontraktu, destruktor
function destroy() public {
    require(msg.sender == owner);
    selfdestruct(owner);
}
```

Pokud někdo volá tuto funkci destroy z jiné adresy než owner, funkce selže. Pokud je však tato

funkce zavolána stejnou adresou, jaká byla uložená konstruktorem do proměnné owner, kontrakt se zničí a zbývající zůstatek etheru odešle na adresu owner. Upozorňujeme, že jsme nepoužili nebezpečný tx.origin k určení, zda vlastník chtěl kontrakt zničit - pomocí tx.origin by zlovolné kontrakty mohly váš kontrakt zničit bez vašeho svolení.

Modifikátory funkcí

Solidity nabízí speciální typ funkce zvaný *modifikátor funkce*. Modifikátory použijete na funkce přidáním názvu modifikátoru do deklarace funkce. Modifikátory se nejčastěji používají k vytváření omezujících podmínek, které mají aplikovat v mnoha funkcí v rámci kontraktu. V naší funkci destroy již máme podmínku pro řízení přístupu. Vytvořme modifikátor funkce, který vyjadřuje tuto podmínku:

```
modifier onlyOwner {
    require(msg.sender == owner);
    _;
}
```

Tento modifikátor funkce, nazvaný onlyOwner, nastavuje podmínku pro jakoukoli funkci, kterou upravuje, a vyžaduje, aby adresa uložená jako vlastník kontraktu byla stejná jako adresa odesilatele transakce msg.sender. Toto je základní návrhový vzor pro řízení přístupu, který umožňuje pouze vlastníkovi kontraktu vykonávat jakoukoli funkci, která má modifikátor onlyOwner.

Možná jste si všimli, že náš modifikátor funkce má zvláštní syntaktický „zástupný symbol“, podtržítko následované středníkem (_;). Tento zástupný symbol je nahrazen kódem funkce, která je upravována. Modifikátor je v podstatě „omotán kolem“ modifikované funkce a umístí její kód na místo identifikované znakem podtržítka.

Chcete-li použít modifikátor, přidejte jeho název do deklarace funkce. Na funkci lze použít více než jeden modifikátor; aplikují se v pořadí, v jakém jsou deklarovány, jako seznam oddělený čárkami.

Přepíšeme naši funkci destroy tak, abychom použili modifikátor onlyOwner:

```
function destroy() public onlyOwner {
    selfdestruct(owner);
}
```

Název modifikátoru funkce (onlyOwner) následuje za klíčovým slovem public a říká nám, že funkce destroy je modifikována modifikátorem onlyOwner. V podstatě si to můžete přečíst jako „Tento kontrakt může zničit pouze vlastník.“ V praxi je výsledný kód ekvivalentní „zabalení“ kódu z onlyOwner okolo destroy.

Modifikátory funkcí jsou velmi užitečným nástrojem, protože nám umožňují psát předpoklady pro funkce a důsledně je používat, což usnadňuje čtení kódu a v důsledku toho snadnější audit zabezpečení. Nejčastěji se používají pro řízení přístupu, ale jsou velmi univerzální a lze je použít pro různé jiné účely.

Uvnitř modifikátoru máte přístup ke všem hodnotám (proměnným a argumentům) viditelným pro modifikovanou funkci. V takovém případě máme přístup k proměnné owner, která je deklarována v kontraktu. Inverzní však není pravda: nemůžete získat přístup k žádné z proměnných modifikátoru uvnitř modifikované funkce.

Dědičnost kontraktu

Solidity objekt contract podporuje *dědičnost*, což je mechanismus pro rozšíření základního kontraktu o další funkčnost. Chcete-li použít dědičnost, zadejte rodičovský kontrakt s klíčovým slovem is:

```
contract Child is Parent {
  ...
}
```

S tímto konstruktem zdědí kontrakt Child všechny metody, funkčnost a proměnné svého rodiče Parent. Solidity také podporuje vícenásobnou dědičnost, kterou lze specifikovat názvy kontraktů oddělených čárkami za klíčovým slovem is:

```
contract Child is Parent1, Parent2 {
  ...
}
```

Dědičnost kontraktu nám umožňuje psát naše kontrakty tak, abychom dosáhli modularity, rozšiřitelnosti a opětovného použití. Začínáme s kontrakty, které jsou jednoduché a implementují nejobecnější funkce, a poté je rozšiřujeme zděděním těchto schopností ve specializovanějších kontraktech.

V našem kontraktu Faucet jsme představili konstruktor a destruktor spolu s kontrolou přístupu pro majitele, nastaveným v konstruktoru. Tyto schopnosti jsou celkem obecné: mnoho kontraktů je bude mít. Můžeme je definovat jako obecné kontrakty, ty pak pomocí dědičnosti rozšířit na kontrakt Faucet.

Začneme definováním základního kontraktu vlastněný (owned), který má proměnnou vlastník (owner), kterou nastavíme v konstruktoru kontraktu:

```
contract owned {
    address owner;

// Konstruktor kontraktu: nastaví vlastníka
    constructor() {
        owner = msg.sender;
    }

// Modifikátor ¬ízení p¬ístupu
    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }
}
```

Dále definujeme základní kontrakt smrtelný mortal, který je potomkem owned:

```
contract mortal is owned {
// Zni¬ení kontraktu, destruktor
    function destroy() public onlyOwner {
        selfdestruct(owner);
    }
}
```

Jak vidíte, kontrakt mortal může použít modifikátor funkce onlyOwner, definovaný v owned. Nepřímo také používá adresovou proměnnou owner a konstruktor definovaný v owned. Dědičnost zjednodušuje každý kontrakt a zaměřuje se na její konkrétní funkčnost, což nám umožňuje spravovat podrobnosti modulárním způsobem.

Nyní můžeme dále rozšířit kontrakt owned, vytvořit jeho potomka Faucet, který zdědí jeho

schopnosti:

```
contract Faucet is mortal {
// Vydává ether každému, kdo o to požádá
    function withdraw(uint withdraw_amount) public {
// Limit výb¬ru
        require(withdraw_amount <= 0.1 ether);
// Zašle ¬ástku na adresu, která o ni požádala
        msg.sender.transfer(withdraw_amount);
    }
// P¬ijme jakoukoliv p¬íchozí ¬ástku
    function () external payable {}
}
```

Díky tomu, že je kontrakt Faucet potomkem mortal, který je zase potomkem owned, má kontrakt konstruktor a funkci destroy a definovaného vlastníka. Funkčnost je stejná jako v případě, kdy tyto funkce byly v rámci Faucet, ale nyní můžeme tyto funkce znovu použít v jiných kontraktech, aniž bychom je znovu psali. Opětovné použití kódu a modularita činí náš kód čistějším, čitelnějším a snáze kontrolovatelným.

Zpracování chyb (assert, require, revert)

Volání kontraktu může být ukončeno a vrátit chybu. Zpracování chyb v Solidity je řešeno čtyřmi funkcemi: assert, require, revert, a throw (nyní zastaralá).

Když kontrakt skončí s chybou, všechny změny stavu (změny proměnných, zůstatků atd.) budou vráceny zpět, a to až do konce řetězce volání kontraktů, pokud byl vyvolána více než jeden kontrakt. Tím je zajištěno, že transakce jsou *atomické*, což znamená, že jsou úspěšně dokončeny nebo nemají žádný vliv na stav a jsou zcela vráceny.

Funkce assert a require fungují stejným způsobem, vyhodnocují stav a zastavují provádění s chybou, pokud je podmínka nepravdivá. Obvykle se assert používá, když se očekává, že výsledek bude pravdivý, což znamená, že pomocí assert testujeme vnitřní podmínky. Pro srovnání, require se používá při testování vstupů (jako jsou argumenty funkcí nebo transakční pole), které nastavují naše očekávání pro tyto podmínky.

Použili jsme require v našem modifikátoru funkce onlyOwner, abychom otestovali, že odesílatel zprávy je vlastníkem kontraktu:

```
require(msg.sender == owner);
```

Funkce require funguje jako *podmínková brána*, brání vykonání zbytku funkce a způsobí chybu, pokud není splněna.

Od verze Solidity v0.4.22 může require obsahovat také pomocnou textovou zprávu, kterou lze použít k zobrazení příčiny chyby. Chybová zpráva je zaznamenána v protokolu transakcí. Takže můžeme vylepšit náš kód přidáním chybové zprávy do funkce require :

```
require(msg.sender == owner, "Only the contract owner can call this
function");
```

Funkce revert a throw zastaví provádění kontraktu a vrátí jakékoli změny stavu. Funkce throw je zastaralá a bude odstraněna v budoucích verzích Solidity; místo toho byste měli použít revert. Funkce revert může také přijmout chybovou zprávu jako svůj jediný argument, který je zaznamenán v protokolu transakcí.

Určité podmínky v kontraktech generují chyby bez ohledu na to, zda je výslovně kontrolujeme. Například v našem kontraktu Faucet nekontrolujeme, zda je dostatek etheru k uspokojení žádosti o výběr. Je to proto, že funkce transfer selže s chybou a transakci vrátí, pokud není k provedení převodu dostatečný zůstatek:

```
msg.sender.transfer(withdraw_amount);
```

Může však být lepší explicitně zkontrolovat a poskytnout jasnou chybovou zprávu o selhání. Můžeme to provést přidáním příkazu require před převodem:

```
require(this.balance >= withdraw_amount,
    "Insufficient balance in faucet for withdrawal request");
msg.sender.transfer(withdraw_amount);
```

Další kód pro kontrolu chyb, jako je tento, mírně zvýší spotřebu plynu, ale nabízí lepší hlášení chyb než v případě vynechání. Budete muset najít správnou rovnováhu mezi spotřebou plynu a podrobnou kontrolou chyb na základě očekávaného využití vašeho kontraktu. V případě kontraktu Faucet určené pro testovací síť bychom se pravděpodobně nedopustili chyby přidáním zvláštního

hlášení, i když to stojí více plynu. Oproti tomu na hlavní síti by kontrakt měl být v používání plynu skromnější.

Události

Když se transakce dokončí (úspěšně nebo ne), vytvoří se *transakční účtenka*, jak uvidíme v Ethereum virtuální stroj . Účtenka transakce obsahuje záznamy *protokolu* které poskytují informace o akcích, ke kterým došlo během provádění transakce. *Události* (Events) jsou vysokoúrovňové objekty v Solidity, které se používají k vytváření těchto protokolů.

Události jsou zvláště užitečné pro odlehčené klienty a služby DApp, které mohou „sledovat“ konkrétní události a nahlásit je do uživatelského rozhraní nebo změnit stav aplikace tak, aby odrážely událost v podkladovém kontraktu.

Objekty událostí berou parametry, které jsou naformátovány a zaznamenávány v protokolech transakcí, v bločence. Můžete zadat klíčové slovo indexed před parametrem, abyste vytvořili hodnotovou část indexovací tabulky (hašovací tabulky), kterou může aplikace prohledávat nebo filtrovat.

Do našeho příkladu Faucet jsme do současné doby nepřidali žádné události, udělejme to. Přidáme dvě události, jednu pro přihlášení všech výběrů a jednu pro hlášení všech vkladů. Tyto události budeme nazývat výběr (Withdrawal) a vklad (Deposit). Nejprve definujeme události v kontraktu Faucet:

```
contract Faucet is mortal {
    event Withdrawal(address indexed to, uint amount);
    event Deposit(address indexed from, uint amount);

    [...]
}
```

Rozhodli jsme se udělat adresy indexované (indexed) abychom umožnili vyhledávání a filtrování v jakémkoli uživatelském rozhraní vytvořeném pro přístup k našemu Faucet.

Dále pomocí klíčového slova emit začleníme data událostí do protokolů transakcí:

```
// Vydejte ether každému, kdo o to požádá
function withdraw(uint withdraw_amount) public {
    [...]
    msg.sender.transfer(withdraw_amount);
    emit Withdrawal(msg.sender, withdraw_amount);
}
// P¬ijm¬te veškerou p¬íchozí ¬ástku
function () external payable {
    emit Deposit(msg.sender, msg.value);
}
```

Výsledný kontrakt *Faucet.sol* vypadá jako Faucet8.sol: Upravený kontrakt kohoutek, s událostmi.

Example 4. Faucet8.sol: Upravený kontrakt kohoutek, s událostmi

```
// Version of Solidity compiler this program was written for
pragma solidity ^0.4.22;

contract owned {
    address owner;
    // Contract constructor: set owner
    constructor() {
        owner = msg.sender;
    }
    // Access control modifier
    modifier onlyOwner {
        require(msg.sender == owner,
                "Only the contract owner can call this function");
        _;
    }
}

contract mortal is owned {
    // Contract destructor
    function destroy() public onlyOwner {
        selfdestruct(owner);
    }
}

contract Faucet is mortal {
    event Withdrawal(address indexed to, uint amount);
    event Deposit(address indexed from, uint amount);

    // Give out ether to anyone who asks
    function withdraw(uint withdraw_amount) public {
        // Limit withdrawal amount
        require(withdraw_amount <= 0.1 ether);
        require(this.balance >= withdraw_amount,
            "Insufficient balance in faucet for withdrawal request");
        // Send the amount to the address that requested it
        msg.sender.transfer(withdraw_amount);
        emit Withdrawal(msg.sender, withdraw_amount);
    }
    // Accept any incoming amount
```

```
    function () external payable {
        emit Deposit(msg.sender, msg.value);
    }
}
```

Zachytávání událostí

Dobře, už jsme vytvořili náš kontrakt na vysílání událostí. Jak se podívat výsledky transakce a „zachytit“ události? Knihovna web3.js poskytuje datovou strukturu, která obsahuje protokoly transakcí. V nich vidíme události způsobené transakcí.

Použijeme truffle k provedení testovací transakce na upraveném kontraktu Faucet. Postupujte podle pokynů v Truffle pro nastavení adresáře projektu a kompilaci kódu `Faucet`. Zdrojový kód naleznete v úložišti knihy na GitHubu [https://github.com/ethereumbook/ethereumbook] pod *code/truffle/FaucetEvents*.

```
<pre data-type="programlisting">
$ <strong>truffle develop</strong>
truffle(develop)> <strong>compile</strong>
truffle(develop)> <strong>migrate</strong>
Using network 'develop'.

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ... 0xb77ceae7c3f5afb7fbe3a6c5974d352aa844f53f955ee7d707ef6f3f8e6b4e61
  Migrations: 0x8cdaf0cd259887258bc13a92c0a6da92698644c0
Saving successful migration to network...
  ... 0xd7bc86d31bee32fa3988f1c1eabce403a1b5d570340a3a9cdba53a472ee8c956
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying Faucet...
  ... 0xfa850d754314c3fb83f43ca1fa6ee20bc9652d891c00a2f63fd43ab5bfb0d781
  Faucet: 0x345ca3e014aaf5dca488057592ee47305d9b3e10
Saving successful migration to network...
  ... 0xf36163615f41ef7ed8f4a8f192149a0bf633fe1a2398ce001bf44c43dc7bdda0
Saving artifacts...

truffle(develop)> <strong>Faucet.deployed().then(i => {FaucetDeployed =
i})</strong>
truffle(develop)> <strong>FaucetDeployed.send(web3.utils.toWei(1,
"ether")).then(res => \
                  { console.log(res.logs[0].event, res.logs[0].args)
})</strong>
Deposit { from: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  amount: BigNumber { s: 1, e: 18, c: [ 10000 ] } }
truffle(develop)> <strong>FaucetDeployed.withdraw(web3.utils.toWei(0.1,
"ether")).then(res => \
                  { console.log(res.logs[0].event, res.logs[0].args)
})</strong>
Withdrawal { to: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  amount: BigNumber { s: 1, e: 17, c: [ 1000 ] } }
</pre>
```

Po nasazení kontraktu pomocí funkce deployed provedeme dvě transakce. První transakcí je vklad (pomocí send), který v protokolech transakcí vysílá událost Deposit v transakčním logu:

```
Deposit { from: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  amount: BigNumber { s: 1, e: 18, c: [ 10000 ] } }
```

Dále použijeme funkci withdraw pro výběr. Toto vysílá událost Withdrawal:

```
Withdrawal { to: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  amount: BigNumber { s: 1, e: 17, c: [ 1000 ] } }
```

K získání těchto událostí jsme se podívali na pole protokolu (log) + vrácené jako výsledek (+res) transakcí. První položka protokolu (logs[0]) obsahuje název události v +logs[0].event a parametry události v logs[0].args. Jejich zobrazením v příkazové řádce vidíme název vyslané události a její parametry.

Události jsou velmi užitečným mechanismem, nejen pro komunikaci uvnitř kontraktu, ale také pro ladění během vývoje .

Volání dalších kontraktů (send, call, callcode, delegatecall)

"smart contracts","calling other contracts from within a contract", id="ix_07smart-contracts-solidity-asciidoc21", range="startofrange")Volání dalších kontraktů z vašeho kontraktu je velmi užitečnou, ale potenciálně nebezpečnou operací. Prověříme různé způsoby, jak toho dosáhnout, a vyhodnotíme rizika každé metody. Stručně řečeno, rizika vyplývají ze skutečnosti, že možná nevíte mnoho o kontraktu, který voláte nebo který volá váš kontrakt. Při psaní chytrých kontraktů musíte mít na paměti, že i když většinou můžete očekávat, že se budete zabývat EOA, nic nebrání tomu, aby svévolně složité a možná zhoubné kontrakty byly volány z vašeho kódu nebo váš kód volaly.

Vytvoření nové instance

Nejbezpečnějším způsobem, jak zavolat další kontrakt, je, pokud tento druhý kontrakt vytvoříte sami. Tímto způsobem jste si jisti jeho rozhraními a chováním. Chcete-li to provést, můžete jej jednoduše vytvořit pomocí klíčového slova new, jako v jiných objektově orientovaných jazycích. V Solidity klíčové slovo new vytvoří kontrakt na bločence a vrátí objekt, který můžete použít k odkazování se na tento kontrakt. Řekněme, že chcete vytvořit a zavolat kontrakt Faucet z jiného kontraktu nazvané Token:

```
contract Token is mortal {
    Faucet _faucet;

    constructor() {
        _faucet = new Faucet();
    }
}
```

Tento mechanismus konstrukce kontraktů zajišťuje, že znáte přesný typ kontraktu a její rozhraní. Kontrakt Faucet musí být definován v rámci Token, což můžete udělat s příkazem import, pokud je definice v jiném souboru:

```
import "Faucet.sol";

contract Token is mortal {
    Faucet _faucet;

    constructor() {
        _faucet = new Faucet();
    }
}
```

Při vytváření můžete volitelně zadat hodnotu přenosu etheru a předat argumenty parametry novému kontraktu:

```
import "Faucet.sol";

contract Token is mortal {
    Faucet _faucet;

    constructor() {
        _faucet = (new Faucet).value(0.5 ether)();
    }
}
```

Můžete také zavolat funkce Faucet. V tomto příkladu zavoláme funkci destroy na Faucet. uvnitř funkce destroy v Token:

```
import "Faucet.sol";

contract Token is mortal {
    Faucet _faucet;

    constructor() {
        _faucet = (new Faucet).value(0.5 ether)();
    }

    function destroy() ownerOnly {
        _faucet.destroy();
    }
}
```

Přestože jste vlastníkem kontraktu Token, kontrakt Token vlastní nový kontrakt Faucet, takže ji může zničit pouze kontrakt Faucet.

Adresování existující instance

Dalším způsobem, jak můžete volat kontrakty, je použitím adresy existující instance kontraktu. U této metody použijete známé rozhraní na existující instanci. Je proto velmi důležité, abyste jistě věděli, že instance, kterou adresujete, je ve skutečnosti typu, který předpokládáte. Podívejme se na příklad:

```
import "Faucet.sol";

contract Token is mortal {

    Faucet _faucet;

    constructor(address _f) {
        _faucet = Faucet(_f);
        _faucet.withdraw(0.1 ether)
    }
}
```

Zde vezmeme adresu poskytnutou jako parametr konstruktoru _f a vložíme ji do objektu Faucet. To je mnohem riskantnější než předchozí mechanismus, protože nevíme s jistotou, zda je tato adresa ve

skutečnosti objektem Faucet. Když voláme withdraw, předpokládáme, že přijímá stejné parametry a provádí stejný kód jako naše deklarace Faucet, ale nemůžeme si být jisti. Víme pouze, že by funkce withdraw zavolaná na této adrese mohla provést něco úplně jiného, než co očekáváme, i když je pojmenováno stejně. Použití adres předaných jako vstup a jejich vkládání do konkrétních objektů je proto mnohem nebezpečnější než, když si vytvoříme kontrakt sami.

Call, delegatecall

Solidity nabízí ještě více funkcí „nízké úrovně“ pro volání dalších kontraktů. Ty odpovídají přímo instrukcím EVM se stejným názvem a umožňují nám sestavit volání kontraktu z kontraktu ručně. Představují tedy nejflexibilnější a nejnebezpečnější mechanismy pro volání dalších smluv.

Zde je stejný příklad, při použití metody call:

```
contract Token is mortal {
    constructor(address _faucet) {
        _faucet.call("withdraw", 0.1 ether);
    }
}
```

Jak vidíte, tento typ volání call je *slepé* volání funkce, podobně jako při vytváření surové transakce, pouze z kontextu kontraktu. Může to vystavit váš kontrakt řadě bezpečnostních rizik, zejména *opakovaného volání*, o nichž budeme podrobněji diskutovat v Opětovné zavolání. Funkce call vrátí false, pokud dojde k problému, takže můžete vyhodnotit návratovou hodnotu pro zpracování chyb:

```
contract Token is mortal {
    constructor(address _faucet) {
        if !(_faucet.call("withdraw", 0.1 ether)) {
            revert("Withdrawal from faucet failed");
        }
    }
}
```

Další variantou volání je delegatecall, která nahradila nebezpečnější callcode. Metoda `callcode` bude brzy nepodporována, takže by se neměla používat.

Jak je uvedeno v adresa objektu, delegatecall se liší od call v tom, že se kontext zprávy msg se

nezmění. Například zatímco call mění hodnotu msg.sender na volací kontrakt, delegatecall zachovává stejné msg.sender jako ve volacím kontraktu. Delegatecall v podstatě spouští kód jiného kontraktuv kontextu provádění aktuálního kontraktu. Nejčastěji se používá k vyvolání kódu z knihovny. Také vám umožňuje čerpat ze vzoru používání knihovních funkcí uložených jinde, ale nechat tento kód pracovat s datovým úložiště vašeho kontraktu.

Volání delegate by mělo být používáno s velkou opatrností. Může to mít neočekávané účinky, zejména pokud kontrakt, který voláte, nebyl navržena jako knihovna.

Použijeme příklad kontraktu, abychom demonstrovali různé sémantiky volání, které používají call a delegatecall pro volání knihoven a kontrraktů. V CallExamples.sol: Příklad jiné sémantiky volání pomocí události zaznamenáváme podrobnosti každého volání a sledujeme, jak se mění kontext volání v závislosti na typu volání.

Example 5. CallExamples.sol: Příklad jiné sémantiky volání

```
pragma solidity ^0.4.22;

contract calledContract {
    event callEvent(address sender, address origin, address from);
    function calledFunction() public {
        emit callEvent(msg.sender, tx.origin, this);
    }
}

library calledLibrary {
    event callEvent(address sender, address origin,  address from);
    function calledFunction() public {
        emit callEvent(msg.sender, tx.origin, this);
    }
}

contract caller {

    function make_calls(calledContract _calledContract) public {

        // Calling calledContract and calledLibrary directly
        _calledContract.calledFunction();
        calledLibrary.calledFunction();

        // Low-level calls using the address object for calledContract
        require(address(_calledContract).
                call(bytes4(keccak256("calledFunction()"))));
        require(address(_calledContract).
                delegatecall(bytes4(keccak256("calledFunction()"))));



    }
}
```

Jak vidíte v tomto příkladu, naším hlavním kontraktem je caller, který volá knihovnu calledLibrary a kontrakt calledContract. Jak volaná knihovna, tak smlouva mají identické funkce calledFunction,

které vydávají událost calledEvent. Událost calledEvent zaznamenává tři data: msg.sender, tx.origin, a this. Pokaždé, když je calledFunction zavolána, může mít jiný kontext volání (s potenciálně různými hodnotami pro všechny kontextové proměnné) v závislosti na tom, zda je volána přímo nebo prostřednictvím delegatecall.

V caller voláme nejprve kontrakt a knihovnu přímo zavoláním calledFunction v každém z nich. Potom explicitně používáme nízkoúrovňové funkce call a delegatecall pro volání calledContract.calledFunction. Tímto způsobem můžeme vidět, jak se různé volající mechanismy chovají.

Pojďme to spustit ve vývojovém prostředí Truffle a zachytit události, abychom viděli, jak to vypadá:

```
<pre data-type="programlisting">
truffle(develop)> <strong>migrate</strong>
Using network 'develop'.
[...]
Saving artifacts...
truffle(develop)> <strong>web3.eth.accounts[0]</strong>
'0x627306090abab3a6e1400e9345bc60c78a8bef57'
truffle(develop)> <strong>caller.address</strong>
'0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f'
truffle(develop)> <strong>calledContract.address</strong>
'0x345ca3e014aaf5dca488057592ee47305d9b3e10'
truffle(develop)> <strong>calledLibrary.address</strong>
'0xf25186b5081ff5ce73482ad761db0eb0d25abfbf'
truffle(develop)> <strong>caller.deployed().then( i => { callerDeployed =
i })</strong>

truffle(develop)>
<strong>callerDeployed.make_calls(calledContract.address).then(res => \
                  { res.logs.forEach( log => { console.log(log.args)
})})</strong>
{ sender: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f',
  origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  from: '0x345ca3e014aaf5dca488057592ee47305d9b3e10' }
{ sender: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  from: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f' }
{ sender: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f',
  origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  from: '0x345ca3e014aaf5dca488057592ee47305d9b3e10' }
{ sender: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  from: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f' }
</pre>
```

Uvidíme, co se tady stalo. Zavolali jsme funkci make_calls a předali jsme adresu calledContract, pak jsme zachytili čtyři události, každá odvysílaná jiným voláním. Pojďme se podívat na funkci make_calls a projít si každý krok.

První volání je:

```
_calledContract.calledFunction();
```

Zde voláme přímo calledContract.calledFunction pomocí vysokoúrovňového ABI pro calledFunction. Vysílaná událost je:

```
sender: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f',
origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
from: '0x345ca3e014aaf5dca488057592ee47305d9b3e10'
```

Jak vidíte, msg.sender je adresa kontraktu caller. V tx.origin je adresa našeho účtu, web3.eth.accounts [0], který odeslal transakci caller. Událost byla vyslána calledContract, jak můžeme vidět z posledního argumentu v události.

Další volání make_calls je do knihovny:

```
calledLibrary.calledFunction();
```

Vypadá to identicky s tím, jak jsme volali kontrakt, ale chová se úplně jinak. Podívejme se na druhou odvysílanou událost:

```
sender: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
from: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f'
```

Tentokrát msg.sender není adresa caller. Místo toho je to adresa našeho účtu a je stejná jako odesilatel transakce. Je to proto, že když zavoláte knihovnu, volání je vždy delegatecall a probíhá v kontextu volajícího. Když tedy byl spuštěn kód calledLibrary, zdědil kontext provádění volajícího caller, jako by jeho kód běžel uvnitř caller. Proměnná this (zobrazená jako from v odvysílané události) je adresa caller, i když je přístupná z `calledLibrary`.

Další dvě nízkoúrovňová voláníi call a delegatecall ověřují naše očekávání a vysílají události, které odrážejí to, co jsme právě provedli. saw.

Úvahy o plynu

Plyn, podrobněji popsaný v Plyn, je neuvěřitelně důležitým faktorem při programování chytrých kontraktů. Plyn je zdroj omezující maximální množství výpočtu, které Ethereum umožní transakci spotřebovat. Pokud je při výpočtu překročen limit plynu, dojde k následující sérii událostí:

- Vyhodí se výjimka „došel plyn“.
- Je obnoven (vrácen) stav kontraktu, jaký byl před jeho provedením.
- Veškerý ether používaný k platbě za plyn se považuje za transakční poplatek; *není* vrácen.

Protože plyn platí uživatel, který zahajuje transakci, uživatelé jsou odrazováni od volání funkcí, které mají vysoké náklady na plyn. Je tedy v nejlepším zájmu programátora minimalizovat náklady na plyn na funkce kontraktu. Za tímto účelem existují určité postupy, které se při vytváření chytrých kontraktů doporučují, aby se minimalizovaly náklady na plyn při volání funkce.

Vyhněte se dynamické velikosti polí

Jakýkoli cyklus skrz pole dynamické velikosti, kde funkce provádí operace s každým prvkem nebo hledá konkrétní prvek, zavádí riziko použití příliš velkého množství plynu. Ve skutečnosti může kontraktu dojít plyn před nalezením požadovaného výsledku nebo před zpracování všech prvků, čímž ztrácí čas a ether, aniž by dával jakýkoli výsledek.

Vyhněte se volání jiných kontraktů

Volání jiných kontraktů, zejména pokud nejsou známy náklady na jejich funkce, představuje riziko vyčerpání plynu. Nepoužívejte knihovny, které nejsou dobře testovány a široce používány. Čím méně kontroly knihovna získala od jiných programátorů, tím větší je riziko jejího použití.

Odhad nákladů na plyn

Pokud potřebujete odhadnout plyn nezbytný k provedení určité metody kontraktu s ohledem na jeho parametry, můžete použít následující postup:

```
var contract = web3.eth.contract(abi).at(address);
var gasEstimate = contract.myAweSomeMethod.estimateGas(arg1, arg2,
    {from: account});
```

Proměnná gasEstimate vám řekne počet jednotek plynu potřebných k jeho provedení. Je to odhad kvůli Turingovské úplnosti EVM - je relativně jednoduché vytvořit funkci, která bude vyžadovat naprosto různá množství plynu k provádění různých volání. Dokonce i rozumný kód může jemně změnit způsoby provádění, což vede k velmi rozdílným nákladům na plyn od jednoho volání k druhému. Většina funkcí je však rozumná a estimateGas dá většinu času dobrý odhad.

K získání ceny plynu ze sítě můžete použít:

```
var gasPrice = web3.eth.getGasPrice();
```

A odtud můžete odhadnout náklady na plyn:

```
var gasCostInEther = web3.utils.fromWei((gasEstimate * gasPrice),
'ether');
```

Použijeme naše funkce pro odhad ceny plynu na odhad nákladů na plyn našeho příkladu Faucet pomocí kódu z úložiště knihy [http://bit.ly/2zf0SIO].

Spusťte Truffle ve vývojovém režimu a spusťte JavaScript soubor v gas_estimates.js: Použití funkce estimateGas, *gas_estimates.js*.

Example 6. gas_estimates.js: Použití funkce estimateGas

```
var FaucetContract = artifacts.require("./Faucet.sol");

FaucetContract.web3.eth.getGasPrice(function(error, result) {
    var gasPrice = Number(result);
    console.log("Gas Price is " + gasPrice + " wei"); //
"100000000000000"

// Získání instance smlouvy
    FaucetContract.deployed().then(function(FaucetContractInstance) {

        // K získání plynu použijte klí¬ové slovo „estimateGas“ za
názvem funkce
        // odhad pro tuto konkrétní funkci (aprove)
        FaucetContractInstance.send(web3.utils.toWei(1, "ether"));
        return
FaucetContractInstance.withdraw.estimateGas(web3.utils.toWei(0.1,
"ether"));

    }).then(function(result) {
        var gas = Number(result);

        console.log("gas estimation = " + gas + " units");
        console.log("gas cost estimation = " + (gas * gasPrice) + "
wei");
        console.log("gas cost estimation = " +
                FaucetContract.web3.utils.fromWei((gas * gasPrice),
'ether') + " ether");
    });
});
```

Takto to vypadá vTruffle vývojářské příkazové řádce:

```
<pre data-type="programlisting">
$ <strong>truffle develop</strong>

truffle(develop)> <strong>exec gas_estimates.js</strong>
Using network 'develop'.

Gas Price is 20000000000 wei
gas estimation = 31397 units
gas cost estimation = 627940000000000 wei
gas cost estimation = 0.00062794 ether
</pre>
```

Doporučujeme, abyste v rámci vývojového pracovního postupu vyhodnotili náklady na plyn na funkce, abyste předešli jakýmkoli překvapením při nasazování kontraktu do hlavní sítě..

Závěry

V této kapitole jsme začali podrobně pracovat s chytrými kontrakty a zkoumali jsme programovací jazyk kontraktů Solidity. Vzali jsme jednoduchý příklad kontraktu, *Faucet.sol,* a postupně jsme ho vylepšili a zkomplikovali, abychom ho použili k prozkoumání různých aspektů jazyka Solidity. V Chytré kontrakty a Vyper budeme pracovat s Vyperem, dalším smluvně orientovaným programovacím jazykem. Porovnáme Vyper se Soliditu, ukážeme některé rozdíly v designu těchto dvou jazyků a prohloubíme naše porozumění programování chytrých kontraktů.

Chytré kontrakty a Vyper

Vyper je experimentální, kontraktově orientovaný programovací jazyk pro virtuální stroj Ethereum, který se snaží poskytovat vynikající auditovatelnost, což vývojářům usnadní vytváření srozumitelného kódu. Ve skutečnosti je jedním z principů Vyperu prakticky nemožné, aby vývojáři psali zavádějící kód.

V této kapitole se podíváme na běžné problémy s chytrými kontrakty, představíme programovací jazyk kontraktů Vyper a porovnáme jej se Solidity, čímž ukážeme rozdíly.

Zranitelnosti a Vyper

Nedávná studie [https://arxiv.org/pdf/1802.06038.pdf] analyzovala téměř milion nasazených Ethereum chytrých kontraktů a zjistila, že mnoho z těchto kontraktů obsahovalo vážné zranitelnosti. Během své analýzy vědci nastínili tři základní kategorie sledovaných zranitelností:

Sebevražedné kontrakty

: Chytré kontrakty, které mohou být zabity libovolnými adresami

Chamtivé kontrakty

: Chytré kontrakty, které mohou dosáhnout stavu, ve kterém nemohou uvolnit ether

Marnotratné kontrakty

: Chytré kontrakty, které využít k uvolnění etheru na libovolné adresy

Chyby zabezpečení jsou zaváděny do inteligentních smluv prostřednictvím kódu. Lze silně tvrdit, že tyto a další zranitelnosti nejsou záměrně zavedeny, ale bez ohledu na to, nežádoucí kód chytrých kontraktů zjevně vede k neočekávané ztrátě finančních prostředků pro Ethereum uživatele, a to není ideální. Vyper je navržen tak, aby usnadňoval zápis bezpečného kódu nebo stejně tak ztěžoval náhodný zápis zavádějícího nebo zranitelného kódu.

Srovnání se Solidity

Jedním ze způsobů, jak se Vyper pokouší zkomplikovat psaní nebezpečného kódu, je záměrné *vynechání* některé z funkcí Solidity. Je důležité, aby ti, kdo uvažují o vývoji chytrých kontraktů ve

Vyperu, pochopili, jaké funkce Vyper nemá a proč. Proto v této části prozkoumáme tyto funkce a poskytneme zdůvodnění, proč byly vynechány.

Modifiers

Jak jsme viděli v předchozí kapitole, v Solidity můžete napsat funkci pomocí modifikátorů. Například následující funkce `changeOwner` spustí kód v modifikátoru nazvaném `onlyBy` jako součást jeho provedení:

```
function changeOwner(address _newOwner)
    public
    onlyBy(owner)
{
    owner = _newOwner;
}
```

Tento modifikátor uplatňuje pravidlo ve vztahu k vlastnictví. Jak vidíte, tento konkrétní modifikátor funguje jako mechanismus k provedení předběžné kontroly jménem funkce `changeOwner`:

```
modifier onlyBy(address _account)
{
    require(msg.sender == _account);
    _;
}
```

Modifikátory však nejsou jen proto, aby prováděly kontroly, jak je ukázáno zde. Ve skutečnosti mohou modifikátory významně změnit prostředí chytrého kontraktu v kontextu volající funkce. Jednoduše řečeno, modifikátory jsou *všudypřítomné*.

Podívejme se na další příklad Solidity stylu:

```
enum Stages {
    SafeStage,
    DangerStage,
    FinalStage
}

uint public creationTime = now;
Stages public stage = Stages.SafeStage;

function nextStage() internal {
    stage = Stages(uint(stage) + 1);
}

modifier stageTimeConfirmation() {
    if (stage == Stages.SafeStage &&
                now >= creationTime + 10 days)
        nextStage();
    _;
}

function a()
    public
    stageTimeConfirmation
// Zde se nachází další kód
{
}
```

Na jedné straně by vývojáři měli vždy kontrolovat jakýkoli jiný kód, který volá jejich vlastní kód. Je však možné, že v určitých situacích (například když časová omezení nebo vyčerpání vedou k nedostatečné koncentraci) může vývojář přehlédnout jediný řádek kódu. To je ještě pravděpodobnější, pokud vývojář musí skákat uvnitř velkého souboru a zároveň mentálně sledovat hierarchii volání funkcí a převádět stav proměnných chytrých kontraktů do paměti.

Podívejme se na předchozí příklad trochu hlouběji. Představte si, že vývojář píše veřejnou funkci nazvanou a. Vývojář je v této smlouvě nový a využívá modifikátor napsaný někým jiným. Na první pohled se zdá, že modifikátor `stageTimeConfirmation` jednoduše provádí některé kontroly týkající se věku kontraktu ve vztahu k volající funkci. Vývojář si nemusí uvědomit, že modifikátor také volá jinou funkci `nextStage`. V tomto zjednodušeném demonstračním scénáři pouhé vyvolání veřejné funkce a vede ke změně proměnné `fáze` chytrého kontraktu ze` SafeStage` na `DangerStage`.

Vyper se úplně zbavil modifikátorů. Doporučení od Vypera jsou následující: pokud provádíte pouze kontroly podmínek pomocí modifikátorů, pak jednoduše používejte kontroly přímo v kódu jako součást funkce; pokud změníte stav chytrého kontraktu a tak dále, znovu proveďte tyto změny explicitně jako součástí funkce. Tím se zlepší kontrola správnosti a čitelnost, protože čtenář nemusí mentálně (nebo ručně) „obalit“ kód modifikátoru kolem funkce, aby viděl, co dělá.

Dědičnost třídy

Dědičnost umožňuje programátorům využít sílu již napsaného kódu získáním již existujících funkcí, vlastností a chování ze stávajících softwarových knihoven. Dědičnost je mocná a podporuje opětovné použití kódu. Solidity podporuje vícenásobnou dědičnost i polymorfismus, ale zatímco to jsou klíčové vlastnosti objektově orientovaného programování, Vyper je nepodporuje. Vyper tvrdí, že implementace dědičnosti vyžaduje, aby kodéry a auditoři skákali mezi více soubory, aby pochopili, co program dělá. Vyper také zastává názor, že vícenásobná dědičnost může způsobit, že kód bude příliš komplikovaný na pochopení - pohled mlčky naznačený v Solidity dokumentaci [http://bit.ly/2Q6Azvo], který poskytuje příklad toho, jak může být vícenásobná dědičnost problematická.

Vložený asembler

Vložený asembler poskytuje vývojářům nízkoúrovňový přístup k Ethereum Virtuálnímu Stroji, který umožňuje Solidity programům provádět operace přímým přístupem k instrukcím EVM. Například následující vložený kód asembleru přičte číslo 3 do paměti umístěné na pozici 0x80:

```
3 0x80 mload add 0x80 mstore
```

Vyper považuje cenu za ztrátu čitelnosti příliš vysokou, kterou nedokáže vyvážit ani rozšířená síla, a proto nepodporuje vložený asembler.

Přetěžování funkce

Přetěžování funkcí umožňuje vývojářům psát více funkcí stejného jména. Která funkce se používá při dané příležitosti, závisí na typech dodaných parametrů. Vezměte například následující dvě funkce:

```
function f(uint _in) public pure returns (uint out) {
    out = 1;
}

function f(uint _in, bytes32 _key) public pure returns (uint out) {
    out = 2;
}
```

První funkce (pojmenovaná f) přijímá vstupní parametr typu uint; druhá funkce (také pojmenovaná f) přijímá dva parametry, jeden typu uint a druhý typu bytes32. Mít více definic funkcí se stejným názvem s různými parametry může být matoucí, takže Vyper nepodporuje přetěžování funkcí.

Přetypování proměnných

Existují dva druhy přetypování: *implicitní* a *explicitní*

Implicitní přetypování se často provádí v době kompilace. Například pokud je typová konverze sémanticky správná a pravděpodobně nebude ztracena žádná informace, kompilátor může provést implicitní převod, jako je převod proměnné typu + uint8 + na + uint16 +. Nejstarší verze Vyperu povolily implicitní přetypování proměnných, ale nedávné verze ne.

Explicitní přetypování lze vložit do Solidity. Bohužel mohou vést k neočekávanému chování. Například uložení uint32 do menšího typu uint16 jednoduše odstraní bity vyššího řádu, jak je ukázáno zde:

```
uint32 a = 0x12345678;
uint16 b = uint16(a);
// Prom¬nná b je nyní 0x5678
```

Vyper místo toho má funkci convert provádějící explicitní přetypování. Funkce převodu (nachází se na řádku 82 v *convert.py* [http://bit.ly/2P36ZKT]):

```
def convert(expr, context):
    output_type = expr.args[1].s
    if output_type in conversion_table:
        return conversion_table[output_type](expr, context)
    else:
        raise Exception("Conversion to {} is invalid.".format(output_type))
```

Všimněte si použití conversion_table (nachází se na řádku 90 stejného souboru), který vypadá takto:

```
conversion_table = {
    'int128': to_int128,
    'uint256': to_unint256,
    'decimal': to_decimal,
    'bytes32': to_bytes32,
}
```

Když vývojář volá convert, odkáže se na odkaz conversion_table, což zajišťuje provedení příslušné konverze. Pokud například vývojář předá funkci convert parametr int128, bude provedena funkce to_int128 na řádku 26 stejného souboru (*convert.py*). Funkce to_int128 je následující:

```
@signature(('int128', 'uint256', 'bytes32', 'bytes'), 'str_literal')
def to_int128(expr, args, kwargs, context):
    in_node = args[0]
    typ, len = get_type(in_node)
    if typ in ('int128', 'uint256', 'bytes32'):
        if in_node.typ.is_literal
            and not SizeLimits.MINNUM <= in_node.value <=
SizeLimits.MAXNUM:
            raise InvalidLiteralException(
                "Number out of range: {}".format(in_node.value), expr
            )
        return LLLnode.from_list(
            ['clamp', ['mload', MemoryPositions.MINNUM], in_node,
            ['mload', MemoryPositions.MAXNUM]], typ=BaseType('int128'),
            pos=getpos(expr)
        )
    else:
        return byte_array_to_num(in_node, expr, 'int128')
```

Jak vidíte, proces převodu zajišťuje, že nelze ztratit žádné informace; pokud by to mohlo být, je vyvolána výjimka. Převodový kód zabraňuje zkrácení a dalším anomáliím, které by byly běžně povoleny implicitním přetypováním.

Výběr explicitního místo implicitní přetypování znamená, že vývojář je zodpovědný za provedení všech přetypování. I když tento přístup vede k podrobnějšímu kódu, zvyšuje také bezpečnost a auditovatelnost chytrých kontraktů.

Počáteční a koncové podmínky

Vyper explicitně zpracovává počáteční a koncové podmínky a změny stavu. I když to vytváří nadbytečný kód, umožňuje to také maximální čitelnost a bezpečnost. Při psaní chytrých kontraktů ve Vyperu by vývojář měl dodržovat následující tři body:

Stav

Jaký je aktuální stav Ethereum stavové proměnné?

Efekty

Jaké dopady bude mít tento kód chytrého kontraktu na stav stavových proměnných po jeho provedení? To znamená, co *bude* ovlivněno a *co nebude* ovlivněno? Jsou tyto účinky shodné se záměrem chytrého kontraktu?

Interakce

Po vyčerpávajícím projednávání prvních dvou úvah je čas spustit kód. Před nasazením logicky projděte kód a zvažte všechny možné trvalé výsledky, důsledky a scénáře provádění kódu, včetně interakcí s jinými kontrakty.

V ideálním případě by měl být každý z těchto bodů pečlivě zvážen a následně důkladně zdokumentován v kódu. Tím se zlepší návrh kódu a nakonec se stane čitelnějším a snadněji kontrolovatelným.

Dekorátory

Na začátku každé funkce mohou být použity následující dekorátory:

@private

Dekorátor `@ private` dělá funkci nepřístupnou mimo kontrakt.

@public

Dekorátor `@ public` dělá funkci veřejně viditelnou a spustitelnou. Například i Ethereum peněženka zobrazí takové funkce při prohlížení kontraktu.

@constant

Funkce s dekorátorem `@ konstanty` nemohou měnit stavové proměnné. Pokud se funkce pokusí změnit stavovou proměnnou, kompilátor ve skutečnosti odmítne celý program (s příslušnou chybou).

@payable

Pouze funkce s dekorátorem `@ payable` mohou převádět hodnotu.

Vyper explicitně implementuje logiku dekorátorů [http://bit.ly/2P14RDq]. Například proces kompilace Vyper se nezdaří, pokud funkce má dekorátor `@payable` i dekorátor `@constant`. To má smysl, protože funkce, která přenáší hodnotu, podle definice aktualizovala stav, takže nemůže být `@constant`. Každá Vyper funkcemusí být buď `@public` nebo `@private` (ale ne obojí!).

Funkce a pořadí proměnných

Každý jednotlivý Vyper kontrakt se skládá pouze z jednoho souboru Vyper. Jinými slovy, celý daný kód Vyper chytrého kontraktu, včetně všech funkcí, proměnných atd., existuje na jednom místě. Vyper vyžaduje, aby funkce každého chytrého kontraktu a deklarace proměnných byly fyzicky psány v určitém pořadí. Solidy tento požadavek vůbec nemá. Podívejme se rychle na příklad v Solidity:

```
pragma solidity ^0.4.0;

contract ordering {

    function topFunction()
    external
    returns (bool) {
        initiatizedBelowTopFunction = this.lowerFunction();
        return initiatizedBelowTopFunction;
    }

    bool initiatizedBelowTopFunction;
    bool lowerFunctionVar;

    function lowerFunction()
    external
    returns (bool) {
        lowerFunctionVar = true;
        return lowerFunctionVar;
    }

}
```

V tomto příkladu funkce nazvaná topFunction volá jinou funkci LowerFunction. topFunction také přiřazuje hodnotu proměnné nazvané initiatizedBelowTopFunction. Jak vidíte, Solidity nevyžaduje, aby tyto funkce a proměnné byly fyzicky deklarovány dříve, než budou vyvolány vykonávaným kódem. Toto je platný Solidity kód, který bude úspěšně kompilován.

Vyper požadavky na pořadí nejsou novinkou; ve skutečnosti byly tyto požadavky na pořadí vždy přítomny při programování v Pythonu. Vyper požaduje, aby pořadí bylo přímé a logické, jak ukazuje následující příklad:

```
# Deklarace prom¬nné nazvané theBool
theBool: public(bool)

# Deklarace funkce nazvané topFunction
@public
def topFunction() -> bool:
# P¬i¬adit hodnotu již deklarované prom¬nné nazvané TheBool
    self.theBool = True
    return self.theBool

# Deklarace funkce nazvané lowerFunction
@public
def lowerFunction():
# Zavolání již deklarované funkce nazvané topFunction
    assert self.topFunction()
```

To ukazuje správné pořadí funkcí a proměnných ve Vyper chytrém kontraktu. Všimněte si, jak jsou proměnná theBool deklarovány před tím, než jí bude přiřazena hodnota a funkce topFunction je deklarována dříve než je volána. Pokud by byl theBool deklarován níže topFunction nebo pokud by topFunction byla deklarována níže LowerFunction, tento konktrakt by nebyl sestaven.

Kompilace

Vyper má svůj vlastní https://vyper.online [online editor a kompilátor], který vám umožní psát a poté kompilovat své chytré kontrakty do bajtkódu, ABI a LLL pouze pomocí webového prohlížeče. Online kompilátor Vyper má pro vaše pohodlí řadu předpřipravených chytrých kontraktů, včetně kontraktů na jednoduchou otevřenou aukci, bezpečné vzdálené nákupy, tokeny ERC20 a další. Tento nástroj nabízí pouze jednu verzi kompilačního softwaru. Je pravidelně aktualizován, ale ne vždy zaručuje nejnovější verzi. Etherscan má online Vyper kompilátor [https://etherscan.io/vyper], který vám umožňuje vybrat verzi kompilátoru. Https://remix.ethereum.org [Remix], původně navržený pro Solidity chytré kontrakty, má nyní na kartě nastavení k dispozici zásuvný modul Vyper.

Vyper implementuje ERC20 jako předkompilovaný kontrakt, což umožňuje, aby tento chytrý kontrakt byly snadno použity po vybalení z krabice. Kontrakty ve Vyperu musí být deklarovány jako globální proměnné. Příklad pro deklarování ERC20 proměnné je následující:

```
token: address(ERC20)
```

Kontrakt můžete také zkompilovat pomocí příkazového řádku. Každý Vyper kontrakt je uložen v jediném souboru s příponou *.vy*. Po instalaci můžete sestavit Vyper kontrakt spuštěním následujícího příkazu:

```
vyper ~/hello_world.vy
```

Lidsky čitelný popis ABI (ve formátu JSON) lze poté získat spuštěním následujícího příkazu:

```
vyper -f json ~/hello_world.v.py
```

Ochrana proti chybám přetečení na úrovni kompilátoru

Chyby přetečení v softwaru můžou mít katastrofické dopady při práci se skutečnou hodnotou. Například jedna transakce z poloviny dubna 2018 [http://bit.ly/2yHfvoF] ukazuje pass: [chybný přenos více než 57 896 044 618 658 1008 000 000 000 000 000 000 000 000 000], 000 000 000 000 000 000 000 BEC tokenů. Tato transakce byla výsledkem celočíselného přetečení v kontraktu ERC20 tokenu BeautyChain (*BecToken.sol*). Vývojáři Solidity mají přístup do knihoven, jako je SafeMath [http://bit.ly/2ABhb4l], stejně jako nástroje pro analýzu zabezpečení Ethereum chytrých kontraktů, jako je Mythril OSS [http://bit.ly/2CQRoGU]. Vývojáři však nejsou nuceni používat bezpečnostní nástroje. Jednoduše řečeno, pokud není jazykem vynucována bezpečnost, vývojáři mohou psát nebezpečný kód, který bude úspěšně kompilován a později „úspěšně“ proveden.

Vyper má vestavěnou ochranu proti přetečení, implementovanou ve dvou bodech. Za prvé, Vyper poskytuje ekvivalent SafeMath [http://bit.ly/2PuDfpB], který obsahuje nezbytné případy výjimek pro celočíselnou aritmetiku. Za druhé, Vyper používá svorky vždy, když je načtena konstantní proměnná, hodnota je předána funkci nebo je přiřazena proměnná. Svorky jsou implementovány

pomocí uživatelských funkcí v nízkoúrovňovém kompilátoru jazyku Lisp-like Language (LLL) a nelze je zakázat. (Vyper kompilátor vytváří výstup v LLL místo EVM bajtkódu; to zjednodušuje vývoj samotného Vyperu.)

Čtení a zápis dat

I když je ukládání, čtení a úpravy dat nákladné, jsou tyto operace nezbytnou součástí většiny chytrých kontraktů. Chytré kontrakty mohou zapisovat data na dvě místa:

Globální stav

> Stavové proměnné v daném chytrém kontraktu jsou uloženy v Ethereum globální stavové trii (trie je druh stromové datové struktury). Chytrý kontrakt může pouze ukládat, číst a měnit data patřící dané adrese kontraktu (tj. chytré kontrakty nemohou číst jiné chytré kontrakty nebo do nich zapisovat).

Protokoly

> Chytrý kontrakt může také zapisovat data do Ethereum bločenky prostřednictvím protokolu událostí. Zatímco zpočátku Vyper použil vlastní __`protokolovou`__ syntaxi pro deklarování těchto událostí, byla provedena aktualizace, která přiblížila jeho deklaraci událostí více v souladu s původní Solidity syntaxí. Například Vyper deklarace události nazvané `MyLog` byla původně __log__({arg1: indexed(bytes[3])}). `Syntaxe se nyní zm¬nila na` `MyLog: event({arg1: indexed(bytes[3])})`. Je důležité si uvědomit, že provedení protokolované události ve Vyperu bylo a stále je následující: `log.MyLog("123")`.

Chytré kontrakty mohou zapisovat do Ethereum bločenky (prostřednictvím protokolu událostí), nemohou však číst události protokolu uložené v bločence. Bez ohledu na to je jednou z výhod zápisu do Ethereum bločenky prostřednictvím protokolu událostí to, že protokoly mohou být objeveny a přečteny ve veřejné bločence pomocí odlehčených klientů. Například hodnota logsBloom v těženém bloku může indikovat, zda je či není přítomna událost protokolu. Jakmile je existence událostí protokolu stanovena, lze data protokolu získat z dané transakční účtenky.

Závěry

Vyper je výkonný a zajímavý nový programovací jazyk orientovaný na kontrakty. Je navržen směrem ke „správnosti“ na úkor určité flexibility. To může programátorům umožnit psát lepší chytré kontrakty a vyhnout se určitým nástrahám, které způsobují vážné chyby. Dále se podrobněji

podíváme na zabezpečení chytrého kontraktu. Některé nuance návrhu Vyperu se mohou projevit po přečtení všech možných bezpečnostních problémů, které mohou nastat u chytrých kontraktů.

Bezpečnost chytrých kontraktů

Při psaní chytrých kontraktů je bezpečnost jedním z nejdůležitějších hledisek. V oblasti programování chytrých kontraktů jsou chyby nákladné a snadno zneužitelné. V této kapitole se podíváme na osvědčené postupy zabezpečení a návrhové vzory, jakož i na „bezpečnostní „antivzory“, což jsou postupy a vzorce, které mohou zavést zranitelnost v našich chytrých kontraktech.

Stejně jako u jiných programů bude chytrý kontrakt vykonáván přesně tak, jak je napsán, což není vždy to, co programátor zamýšlel. Kromě toho jsou všechny chytré kontrakty veřejné a každý uživatel s nimi může komunikovat jednoduše vytvořením transakce. Jakákoli zranitelnost může být zneužita a ztráty jsou téměř vždy nenapravitelné. Je proto důležité dodržovat osvědčené postupy a používat osvědčené návrhové vzory.

Doporučené postupy zabezpečení

Defenzivní programování je styl programování, který je zvláště vhodný pro chytré kontrakty. Zdůrazňuje následující, z nichž všechny jsou osvědčenými postupy:

Minimalismus/jednoduchost

> Složitost je nepřítelem bezpečnosti. Čím jednodušší je kód a čím méně dělá, tím menší je pravděpodobnost výskytu chyby nebo nepředvídaného efektu. Při prvním zapojení do programování chytrých kontraktů se vývojáři často pokoušejí napsat spoustu kódu. Místo toho byste se měli podívat na svůj kód chytrého kontraktu a pokusit se najít způsoby, jak udělat méně, s méně řádky kódu, menší složitostí a méně „funkcemi“. Pokud vám někdo řekne, že jejich projekt vytvořil „tisíce řádků kódu“ pro chytré kontrakty, měli byste zpochybnit zabezpečení tohoto projektu. Jednodušší je bezpečnější.

Opětovné použití kódu

> Nezkoušejte znovu vynalézat kolo. Pokud již existuje knihovna nebo kontrakt, která dělá většinu toho, co potřebujete, znovu ji použijte. V rámci svého vlastního kódu dodržujte SUCHÝ princip: Neopakujte se. Pokud vidíte, že se úryvek kódu opakuje vícekrát, zeptejte se sami sebe, zda by mohl být zapsán jako funkce nebo knihovna a znovu použit. Kód, který byl rozsáhle používán a testován, je pravděpodobně bezpečnější než jakýkoli nový kód, který píšete. Dejte si pozor na syndrom „nevytvořil jsem já sám“, kde jste v pokušení „vylepšit“

prvek nebo komponentu vytvořením od nuly. Bezpečnostní riziko je často větší než hodnota zlepšení.

Kvalita kódu

Kód chytrého kontraktu nikdy neodpouští. Každá chyba může vést k peněžním ztrátám. Neměli byste zacházet s programováním chytrého kontraktu stejně jako s univerzálním programováním. Zápis DApps v Solidity není jako vytvoření webového aplikace v JavaScriptu. Spíše byste měli použít přísné techniky a vývoj metodik, jako byste to udělali v leteckém inženýrství nebo v jakékoli podobně neodpouštějící disciplíně. Jakmile „spustíte“ svůj kód, můžete udělat jen málo pro vyřešení problémů.

Čitelnost/kontrolovatelnost

Váš kód by měl být jasný a srozumitelný. Čím snazší je číst, tím snazší je kontrolovat. Chytré kontrakty jsou veřejné, protože každý si může přečíst bajtkód a kdokoli ho může zpětně upravit. Proto je užitečné rozvíjet svou práci na veřejnosti pomocí metodologií založených na spolupráci a otevřeného zdrojového kódu, čerpat z kolektivní moudrosti vývojářské komunity a těžit z nejvyššího společného jmenovatele vývoje otevřených zdrojových kódů. Měli byste napsat kód, který je dobře zdokumentovaný a snadno čitelný, podle stylů a pojmenovávacích konvencí, které jsou součástí Ethereum komunity.

Test pokrytí

Otestujte vše, co můžete. Chytré kontrakty probíhají ve veřejném prováděcím prostředí, kde je může provádět kdokoli s jakýmkoli vstupem, se kterým chce. Nikdy byste neměli předpokládat, že vstup, jako jsou parametry funkce, je dobře formátovaný, správně ohraničený nebo má neškodný účel. Než povolíte pokračování provádění kódu, otestujte všechny parametry, zda jsou v očekávaném rozmezí a zda jsou správně naformátovány.

Bezpečnostní rizika a antivzory

Jako programátor chytrých kontraktů byste měli znát nejvíce běžná bezpečnostní rizika, aby bylo možné zjistit a vyhnout se programovacím vzorům, které vaše smlouvy vystavují těmto rizikům. V následujících několika oddílech se podíváme na různá bezpečnostní rizika, příklady toho, jak mohou vzniknout zranitelnosti, a protiopatření nebo preventivní řešení, která lze použít k jejich řešení.

Opětovné zavolání

Jednou z vlastností Ethereum chytrých kontraktů je jejich schopnost volat a využívat kód z jiných externích kontraktů. Kontrakty také obvykle zpracovávají ether a často posílají ether různým externím uživatelům Tyto operace vyžadují, aby kontrakty odesílaly externí volání. Tyto externí volání mohou být uneseny útočníky, kteří mohou vynutit provést další kód (pomocí nouzové funkce), včetně zpětného volání do sebe. Útoky tohoto druhu byly použity v neslavném The DAO útoku [http://bit.ly/2DamSZT].

Další informace o útocích opětovného volání uvádí Gus Guimareas v příspěvku na blogu [http://bit.ly/2zaqSEY] a Osvědčené postupy Ethereum chytrých kontraktů [http://bit.ly/2ERDMxV].

Zranitelnost

RNDr. Jan Lánský, Ph.D. (zizelevak@gmail.com [mailto:zizelevak@gmail.com]), Vysoká škola finanční a správní, 2020; POZNÁMKA pro editora Nadpisy oddílů „Zranitelnost“ a „Preventivní techniky“ v této kapitole byly změněny z nadpisů na tučné formátování úmyslně, aby nedošlo k nepořádku obsahu opakovaným zněním. RNDr. Jan Lánský, Ph.D. (zizelevak@gmail.com [mailto:zizelevak@gmail.com]), Vysoká škola finanční a správní, 2020;

Tento typ útoku může nastat, když kontrakt pošle ether na neznámou adresu. Útočník může pečlivě vytvořit smlouvu na externí adrese která obsahuje škodlivý kód v nouzové funkci. Když tedy kontrakt pošle ether na tuto adresu, bude vyvolán škodlivý kód. Obvykle škodlivý kód provede funkci zranitelného kontraktu, která nebyla očekávána vývojářem Termín „opětovné volání“ (reentrancy) pochází ze skutečnosti, že externí škodlivé kontrakty mohou zavolat funkci zranitelného kontraktu a opětovně *reenters* do cesty vykonávání kódu

Abychom to objasnili, zvažte jednoduchý zranitelný kontrakt v 2007 EtherStore.sol, který pracuje jako Ethereum trezor, který umožňuje vkladatelům vybrat pouze 1 ether za týden.

Example 7. EtherStore.sol

```
contract EtherStore {

    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(address => uint256) public balances;

    function depositFunds() external payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public {
        require(balances[msg.sender] >= _weiToWithdraw);
        // omezení výb¬ru
        require(_weiToWithdraw <= withdrawalLimit);
        // ¬asové omezení výb¬ru
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
        require(msg.sender.call.value(_weiToWithdraw)());
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
    }
 }
```

Tento kontrakt má dvě veřejné funkce, `depositFunds` a `withdrawFunds`. Funkce `depositFunds` jednoduše zvýší zůstatek odesilatele. Funkce `withdrawFunds` dovolí odesilateli určit množství vybraného wei. Tato funkce je navržena, aby uspěla pouze pokud požadované množství vybíraného wei je menší než 1 ether a výběr nenastal v posledním týdnu.

Zranitelnost ja na řádce 17, kde kontrakt zasílá uživateli jím požadované množství etheru. Uvažujme útočníka, který vytvořil kontrakt v Attack.sol.

Example 8. Attack.sol

```
import "EtherStore.sol";

contract Attack {
  EtherStore public etherStore;

  // inicializujte prom¬nnou etherStore pomocí adresy kontraktu
  constructor(address _etherStoreAddress) {
      etherStore = EtherStore(_etherStoreAddress);
  }

  function attackEtherStore() external payable {
      // útok na nejbližší ether
      require(msg.value >= 1 ether);
      // odešle eth do funkce depositFunds()
      etherStore.depositFunds.value(1 ether)();
      // za¬ne magie
      etherStore.withdrawFunds(1 ether);
  }

  function collectEther() public {
      msg.sender.transfer(this.balance);
  }

  // nouzová function - kde nastane magie
  function () payable {
      if (etherStore.balance > 1 ether) {
          etherStore.withdrawFunds(1 ether);
      }
  }
}
```

Jak by mohlo dojít k zneužití? Za prvé by útočník vytvořil škodlivou smlouvu (řekněme na addrese `0x0...123`) s adresou kontraktu `EtherStore`' jako samostatným parametrem konstruktoru. To by inicializovalo a nasměrovalo veřejnou proměnou `etherStore` do konteraktu, který má být napaden.

Útočník by pak zavolal funkci `attackEtherStore` s nějakým množstvím etheru vyšším nebo rovným 1, předpokládejme `1 ether` prozatím V tomto příkladu předpokládáme také řadu dalších

uživatelů, kteří do tohoto kontraktu vložil ether, takže jeho aktuální zůstatek je `10 ether`. Poté nastane následující:

1. *Attack.sol*, řádek 15: Funkce `depositFunds` kontraktu `EtherStore` bude zavolána s hodnotou `msg.value` of `1 ether` (a hodně plynu). Odesilatel (`msg.sender`) bude škodlivý kontraktu (`0x0…123`). Tedy `balances[0x0..123] = 1 ether`.
2. *Attack.sol*, řádek 17: Škodlivý kontrakt pak zavolá funkci `withdrawFunds` kontraktu `EtherStore` s parametrem `1 ether`. ím budou splněny všechny požadavky (řádky 12–16 `EtherStore` kontraktu) protože nebyly provedeny žádné předchozí výběry.
3. *EtherStore.sol*, řádek 17: Kontrakt pošle `1 ether` zpět škodlivému kontraktu.
4. *Attack.sol*, řádek 25: Platba škodlivému kontraktu bude poté vykonána nouzovou funkcí.
5. *Attack.sol*, řádek 26: Celkový zůstate kontraktu EtherStore contract byl `10 ether` a nyní je `9 ether`, takže if podmíněný příkaz projde.
6. *Attack.sol*, řádek 27: Nouzová funkce zavolá znovu `EtherStore` funkci `withdrawFunds` a '*opětovně vstoupí* do' the `EtherStore` kontraktu
7. *EtherStore.sol*, řádek 11: V tomto druhém volání `withdrawFunds`, je zůstatek útočícího kontraktu stále `1 ether`, protože řádka 18 ještě nebyla vykonána. Takže, my stále máme `balances[0x0..123] = 1 ether`. To je také případ proměnné `lastWithdrawTime`. Znovu jsou splněny všechny požadavky.
8. *EtherStore.sol*, řádka 17: Útočící kontrakt vybere další `1 ether`.
9. Steps 4 je opakován dokud není porušena podmínka `EtherStore.balance > 1`, jak uvádí řádek 26 v *Attack.sol*.
10. *Attack.sol*, řádek 26: Jakmile v kontraktu `EtherStore` zbývá 1 (nebo méně) etheru, selže tato podmínka `if`. To umožní, aby byly provedeny řádky 18 a 19 kontraktu „EtherStore“ (pro každé volání funkce `drawFunds`).
11. *EtherStore.sol*, řádky 18 a 19: Zůstatek `balances` a mapování posledního výběru `lastWithdrawTime` budou nastaveny a provádění skončí.

Konečným výsledkem je, že útočník vybral vše kromě 1 etheru z kontraktu `EtherStore` v jediné transakci.

Peventivní techniky

Existuje řada běžných technik, které pomáhají vyhýbat se možné zranitelnosti chytrého kontraktu útokem vícenásobného volání. Prvním z nich je (pokud je to možné) je použití vestavěné transfer [http://bit.ly/2Ogvnng] function pro zasílání etheru externím kontraktům. Funkce transfer pouze zašle 2300 plynu s externím voláním, což není dostatečné, aby cílová adresa /kontrakt byla schopna zavolat další kontrakt (např. znovu zavolat odesílající kontrakt).

Druhou technikou je zajistit, aby veškerá logika, která mění stav proměnných se udála před zasláním etheru mimo kontrakt (nebo jakýmkoliv externím voláním). V příkladu `EtherStore`, řádky 18 a 19 v *EtherStore.sol* měly by být přesunuty před řádek 17. Je dobrým zvykem, aby byl jakýkoli kód, který provádí externí volání na neznámé adresy poslední operace v lokalizované funkci nebo provedeném kódu. Toto je známo jako vzor kontrola, efekt, interakce [http://bit.ly/2EVo70v].

Třetí technikou je mutex, toje přidání stavové proměnné, která zamyká kontrakt během vykonávná kódu, zabraňující vícenásobnému volání

Použití všech těchto technik (použití všech tří je zbytečné, ale my to děláme) pro demonstrační účely) na *EtherStore.sol*, dává Kontrakt odolný útoku vícenásobného volání

```
contract EtherStore {

    // inicializace mutexu
    bool reEntrancyMutex = false;
    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(address => uint256) public balances;

    function depositFunds() external payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public {
        require(!reEntrancyMutex);
        require(balances[msg.sender] >= _weiToWithdraw);
        // omezení výb¬ru
        require(_weiToWithdraw <= withdrawalLimit);
        // ¬asové omezení výb¬ru
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
        // nastaví mutex reEntrancy p¬ed externím voláním
        reEntrancyMutex = true;
        msg.sender.transfer(_weiToWithdraw);
        // uvolní mutex po externím volání
        reEntrancyMutex = false;
    }
}
```

Skutečný příklad: The DAO

The DAO (Decentralized Autonomous Organization) byl napaden významným útokem, který nastal v rané fázi vývoje Etherea. V tu dobu kontrakt držel přes 150 milionů dolarů. Opětovné volání hrálo hlavní roli v útoku, což nakonec vedlo k tvrdému rozštěpení, která vytvořila Ethereum Classic (ETC). Dobrá analýza zneužití DAO viz http://bit.ly/2EQaLCI. Další informace o historii rozštěpení Etherea, časovém plánu DAO útoku a narození ETC v tvrdém rozštěpení naleznete v Ethereum Standardy.

Aritmetické podtečení / přetečení

Virtuální počítač Ethereum určuje datové typy pevné velikosti pro celá čísla. To znamená, že celočíselná proměnná může představovat pouze určitý rozsah čísel. Například `uint8` může ukládat pouze čísla z intervalu [0,255]. Pokus o uložení `256` do `uint8` vede k výsledku `0`. Pokud tomu nebude věnována pozornost, mohou být proměnné v Solidity zneužity, pokud není zkontrolován vstup uživatele, a přesto jsou provedeny výpočty jejichž výsledné číslo se nachází mimo interval datového typu, do kterého mají být uloženy

Další čtení o aritmetických přetečeních a podtečeních, viz "How to Secure Your Smart Contracts" [https://bit.ly/2nNLuOr], Doporučené postupy pro Ethereum chytré kontrakty [https://bit.ly/2MOfBPv], a "Ethereum, Solidity a přetečení celých čísel: programování bločenek jako v roce 1970" [https://bit.ly/2xvbx1M].

Zranitelnost

Přetečení / podtečení nastane, když je provedena operace, která vyžaduje proměnnou pevné velikosti pro uložení čísla (nebo části dat), které je mimo rozsah datového typu proměnné

Například, odečtením`1` od `uint8` (8 bitové celé číslo bez znaménka) proměnné, jejíž hodnota je `0` vede k číslu `255`. To je *podtečení*. Přiřadili jsme číslo pod dolní mezí `uint8`, takže výsledek je *přetočen* a dává největší číslo `uint8`, které může být uloženo. Podobně přičtení `2^8=256` k `uint8` ponechá proměnnou nezměněnou, protože jsme přetočili o celý rozsah `uint`. Dvě jednoduché analogie tohoto chování jsou Počitadla ujetých kilometrů v automobilech, která měří ujetou vzdálenost (po přetočí se na 000000 po překonání nejvyššího čísla, např. 999999) a o periodické matematické funkce (přičtení 2π k parametru sin ponechá hodnotu nezměněnou).

Přidání čísel větších než je rozsah datového typu se nazývá *přetečení*. Pro ujasnění, přičtení `257` do `uint8` s aktuální hodnotou `0` vede k číslu `1`. Někdy je poučné přemýšlet o proměnných pevné velikosti jako cyklických, u kterých začneme znovu od nuly, pokud přičteme čísla nad největší možné uložené číslo a začneme odpočítávat od největšího čísla, pokud odečítáme od nuly. V případě `int` typů se znaménkem, které _mohou _ reprezentovat záporná čísla, začneme znovu, jakmile dosáhneme největší záporné hodnoty; například pokud se pokusíme odečíst `1` od` int8`, jehož hodnota je `-128`, dostaneme` 127`.

Tyto druhy numerických triků umožňují útočníkům zneužívat kód. neočekávané logické toky. Zvažte

například kontrakt TimeLock v TimeLock.sol.

Example 9. TimeLock.sol

```
contract TimeLock {

    mapping(address => uint) public balances;
    mapping(address => uint) public lockTime;

    function deposit() external payable {
        balances[msg.sender] += msg.value;
        lockTime[msg.sender] = now + 1 weeks;
    }

    function increaseLockTime(uint _secondsToIncrease) public {
        lockTime[msg.sender] += _secondsToIncrease;
    }

    function withdraw() public {
        require(balances[msg.sender] > 0);
        require(now > lockTime[msg.sender]);
        balances[msg.sender] = 0;
        msg.sender.transfer(balance);
    }
}
```

Tento kontrakt je navržen tak, aby fungoval jako časový trezor: uživatelé mohou vložit do kontraktu éter a bude tam uzamčen alespoň týden. Uživatel může prodloužit dobu čekání na déle než 1 týden, pokud si zvolí, ale po uložení si uživatel může být jistý, že je jeho ether bezpečně uzamčen alespoň týden, respektive, takto to bylo zamýšleno.

V případě, že je uživatel nucen předat svůj soukromý klíč, kontrakt jako tento by mohl být užitečný, aby se zajistilo, že jeho ether bude na krátkou dobu nedosažitelný. Ale pokud uživatel se v této smlouvě zamkl v `100 etheru` a předal své klíče útočníkovi, mohl útočník použít přetečení k přijetí etheru, bez ohledu na časový zámek `lockTime`.

Útočník by mohl určit aktuální `lockTime` pro adresu od které nyní drží klíč (je to veřejná proměnná). Zavolejme tento `userLockTime`. Mohl pak zavolat `increaseLockTime` funkci a předat ji

parametr `2^256 - userLockTime`. Toto číslo bude přičteno k aktuální hodnotě `userLockTime` a způsobí přetečení, nastavení `lockTime[msg.sender]` na `0`. Útočník by pak mohl jednoduše zavolat funkci `withdraw` pro získání jeho odměny.

Podívejme se na další příklad (Příklad chyby podtečení z výzvy Ethernaut), tento zhttps://github.com/OpenZeppelin/ethernaut[Ethernaut výzev].

VAROVÁNÍ: SPOILER: *Pokud jste dosud nevyřešili Ethernaut problémy, tohle dává řešení jedné z úrovní.*

Example 10. Příklad chyby podtečení z výzvy Ethernaut

```
pragma solidity ^0.4.18;

contract Token {

  mapping(address => uint) balances;
  uint public totalSupply;

  function Token(uint _initialSupply) {
    balances[msg.sender] = totalSupply = _initialSupply;
  }

  function transfer(address _to, uint _value) public returns (bool) {
    require(balances[msg.sender] - _value >= 0);
    balances[msg.sender] -= _value;
    balances[_to] += _value;
    return true;
  }

  function balanceOf(address _owner) public constant returns (uint
balance) {
    return balances[_owner];
  }
}
```

Jedná se o jednoduchý tokenový kontrakt, který využívá funkci `transfer`, umožňující účastníkům pohybovat se svými tokeny. Vidíte chybu v tomto kontraktu?

Chyba je ve funkci `transfer`. Příkaz require na řádku 13 lze obejít pomocí podtečení. Zvažte uživatele s nulovým zůstatkem. Mohl zavolat funkci `transfer` s jakoukoli nenulovou hodnotou `_value` a obejít příkaz require na řádku 13. To je protože `balances[msg.sender]` je 0 (a `uint256`), takže odečítání jakékoli kladné hodnoty (mimo `2^256`) vede ke kladnému číslu, jak bylo dříve popsáno. To vede ke splnění podmínky na řádku 14, kde bude k zůstatku připsáno kladné číslo. Tak v tomto příkladě, útočník může získat zdarma tokeny kvůli zranitelnosti podtečení.

Peventivní techniky

Současná konvenční technika na ochranu proti zranitelnosti podtečení / přetečení je použití nebo vytváření matematických knihoven, které nahrazují standardní matematické operátory sčítání, odčítání a násobení (rozdělení je vyloučeno, protože nezpůsobuje přetečení / podtečení a EVM se vrací do původního stavu při dělení 0).

OpenZeppelin [https://github.com/OpenZeppelin/openzeppelin-solidity] odvedli skvělou práci při vytváření a auditu zabezpečených knihoven pro komunitu Ethereum. Zejména jejich knihovna+SafeMath+ [http://bit.ly/2ABhb4l] lze použít k zamezení zranitelností podtečení / přetečení.

Abychom demonstrovali, jak jsou tyto knihovny použity v Solidity, opravme kontrakt „`TimeLock` pomocí knihovny „ `SafeMath`. Verze kontraktu bez přetečení je:

```
library SafeMath {

  function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    if (a == 0) {
      return 0;
    }
    uint256 c = a * b;
    assert(c / a == b);
    return c;
  }

  function div(uint256 a, uint256 b) internal pure returns (uint256) {
    // assert(b > 0); // Solidity automaticky vyvolá výjimku p¬i d¬lení 0
    uint256 c = a / b;
    // assert(a == b * c + a % b); // Toto postihuje všechny p¬ípady
    return c;
  }
```

```
  function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    assert(b <= a);
    return a - b;
  }

  function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;
  }
}

contract TimeLock {
    using SafeMath for uint; // použití knihovny pro typ uint
    mapping(address => uint256) public balances;
    mapping(address => uint256) public lockTime;

    function deposit() external payable {
        balances[msg.sender] = balances[msg.sender].add(msg.value);
        lockTime[msg.sender] = now.add(1 weeks);
    }

    function increaseLockTime(uint256 _secondsToIncrease) public {
        lockTime[msg.sender] =
lockTime[msg.sender].add(_secondsToIncrease);
    }

    function withdraw() public {
        require(balances[msg.sender] > 0);
        require(now > lockTime[msg.sender]);
        balances[msg.sender] = 0;
        msg.sender.transfer(balance);
    }
}
```

Všimněte si, že všechny standardní matematické operace byly nahrazeny těmito definovanými v knihovně `SafeMath`. Kontrakt `TimeLock` již dále neprovádí jakoukoli operaci, která je schopna přetečení / podtečení.

Příklady ze života: PoWHC a přetečení dávkového přenosu (CVE-2018–10299)

Proof of Weak Hands Coin (PoWHC), volně přeloženo jako důkaz slabých rukou, byl původně vymyšlen jako vtipné, bylo Ponziho schéma napsané internetovým kolektivem. Bohužel se zdá, že autor (autoři) kontraktu předtím neviděli přetečení / podtečení a následkem toho bylo 866 etheru odcizeno z kontraktu. Eric Banisadr poskytuje dobrý přehled o tom, jak došlo k podtečení (což není příliš odlišné od výše popsané Ethernaut výzvy) na svém https://bit.ly/2wrxIFJ[příspěvku v blogu.

Další příklad [http://bit.ly/2CUf7WG] přináší implementace funkce `batchTransfer()` function ze skupiny kontraktů ERC20 tokenů. Implementace obsahovala chybu přetečení; o podrobnostech si můžete přečíst na vhttps://bit.ly/2HDlIs8[PeckShield účtu].

Neočekávaný ether

Obvykle, když je ether posílán kontraktu, ten musí provést buď nouzovou funkci nebo jinou funkci definovanou v kontraktu. existují dvě výjimky, kdy kontrakt může přijmout ether bez vykonání kódu. Kontrakty, které se spoléhají na vykonání kódu při přijetí jakéhokoli etheru mohou být zranitelné útokem, při kterém je jim zaslán nežádoucí ether.

Další informace k tomuto tématu viz "Jak zabezpčit váš chytrý kontrakt" [https://bit.ly/2MR8Gp0] a "Solidity bezpečnostní vzory - vnucení etheru do kontraktu" [http://bit.ly/2RjXmUWl].

Zranitelnost

Běžná technika defensivního programování, která je užitečná při vynucování správného stavu transakcí nebo ověřování operací je *ověřováni invariantu*. Tato technika zahrnuje definici množiny invariantů (metrik nebo parametrů, které by se neměli měnit) a kontrolování že zůstaly nezměněny po jedné nebo mnoha operacích Toto je obvykle dobrý návrh za předpokladu, že kontrolované invarianty jsou ve skutečnosti invarianty. Jedním příkladem invariantu je celkové množství `totalSupply` pevné emise ERC20 tokenu [http://bit.ly/2CUf7WG]. Protože žádná funkce by neměla upravovat tento invariant, mohli bychom přidat kontrolu, že funkce `transfer` zajišťuje, že `totalSupply` zůstává nezměněno, k zajištění fungování funkce podle očekávání.

Zejména existuje jeden zjevný invariant, který může být lákavé použít ale ten může ve skutečnosti manipulován externím uživatelem (bez ohledu na stanovená pravidla chytrým kontraktem). Toto je

aktuální ether uložený v kontraktu. Když se vývojáři poprvé učí Solidity, mají mylnou představu, že kontrakt může přijmout nebo získat éter pouze pomocí funkce přijímající platby. Tato mylná představa může vést ke kontraktům, které mají falešné předpoklady o zůstatku etheru v nich, což může vést k řadě zranitelností. Kouřící zbraň pro tuto zranitelnost je (nesprávné) použití `this.balance`.

Existují dva způsoby, jak lze ether (násilně) poslat na kontrakt bez použití platby přijímající funkce nebo provedení jakéhokoli kódu kontraktu:

Self-destruct/suicide

> Jakýkoli kontrakt je schopna implementovat `selfdestruct` funkci [http://bit.ly/2RovrDf], která odstraní všechen bajtkód z adresy kontraktu a zašle všechen uloženy ether na adresu specifikovanou parametrem. Pokud tato specifikovaná adresa je také kontrakt, žádná funkce (včetně záložní) není zavolána. Proto funkce `selfdestruct` může být použita k násilnému zaslání etheru do jakéhokoli kontraktu bez ohledu na kód, který může existovat v kontraktu, dokonce i do kontraktu bez platby přijímající funkce. To znamená, že každý útočník může vytvořit kontrakt s funkcí `selfdestruct`, zaslat mu ether a zavolat `selfdestruct(target)` a vynutit zaslání etheru na cílový `target` kontrakt. Martin Swende napsal excelentní příspěvek na blogu [http://bit.ly/2OfLukM] popisující některé vtípky instrukce self-destruct (Vtípek #2) společně s s účtem kontrolujícím nesprávně invarianty, což může vést k poněkud katastrofickému zhroucení Ethereum sítě.

Předposlaný ether

> Dalším způsobem, jak zaslat ether kontraktu, je předposlat ether adrese kontraktu Adresy kontraktů jsou deterministické, ve skutečnosti je tato adresa vypočtena z Keccak-256 haše (běžně označovaného SHA-3) adresy vytvářející kontrakt a transakční nonce použité pro vytvoření kontraktu. contract. Konkrétně má podobu `address = sha3(rlp.encode([account_address,transaction_nonce]))` (viz diskuse Adriana Manninga "Bezklíčový ether" [http://bit.ly/2EPj5Tq] pro některé zábavné případy použití tohoto). To znamená, že kdokoli může spočítat, jaká bude adresa kontraktu, než bude kontrakt vytvořen a poslal ether na tuto adresu. Když je kontrakt vytvořen, bude mít nenulovou etherovou balanci.

Pojďme prozkoumat některá úskalí, která mohou na základě těchto znalostí vzniknout. Zvažte příliš jednoduchý kontrakt v EtherGame.sol.

Example 11. EtherGame.sol

```
contract EtherGame {

    uint public payoutMileStone1 = 3 ether;
    uint public mileStone1Reward = 2 ether;
    uint public payoutMileStone2 = 5 ether;
    uint public mileStone2Reward = 3 ether;
    uint public finalMileStone = 10 ether;
    uint public finalReward = 5 ether;

    mapping(address => uint) redeemableEther;
    // Uživatelé platí 0,5 etheru. Na konkrétních milnících získají
odm¬nu.
    function play() external payable {
        require(msg.value == 0.5 ether); // každá hra stojí 0.5 etheru
        uint currentBalance = this.balance + msg.value;
        // zajistit, aby po dokon¬ení hry žádný hrá¬ již nehrál
        require(currentBalance <= finalMileStone);
        // p¬i milníku odm¬nit hrá¬e
        if (currentBalance == payoutMileStone1) {
            redeemableEther[msg.sender] += mileStone1Reward;
        }
        else if (currentBalance == payoutMileStone2) {
            redeemableEther[msg.sender] += mileStone2Reward;
        }
        else if (currentBalance == finalMileStone ) {
            redeemableEther[msg.sender] += finalReward;
        }
        return;
    }

    function claimReward() public {
        // ujist¬te se, že je hra dokon¬ena
        require(this.balance == finalMileStone);
        // ujist¬te se, že je na vyplacení odm¬ny
        require(redeemableEther[msg.sender] > 0);
        redeemableEther[msg.sender] = 0;
        msg.sender.transfer(transferValue);
    }
 }
```

Tento kontrakt představuje jednoduchou hru (která by samozřejmě zahrnovala souběh), kde hráči posílají kontraktu 0,5 etheru v naději, že budou hráčem, který dosáhne jako první jednoho ze tří milníků. Milníky jsou uvedeny v etheru. První, kdo dosáhne milníku může si vybrat část etheru, když hra skončí. Hra končí, když je dosažen konečný milník (10 ether). Uživatelé si pak mohou nárokovat své odměny.

Problémy s kontraktem EtherGame pocházejí ze špatného používání `this.balance` na obou řádkách 14 (a odvozením 16) a 32. Zlomyslný útočník mohl násilně poslat malé množství etheru, řekněme 0,1 etheru, pomocí funkce `selfdestruct` (diskutováno dříve), aby zabránit budoucím hráčům v dosažení milníku. `this.balance` nikdy nebude násobkem 0,5 etheru díky tomuto příspěvku 0,1 etheru, protože všichni legitimní hráči mohou posílat pouze 0,5-etherové přírůstky. Tím se zabrání všem podmínkám if na řádcích 18, 21, a 24 v jejich splnění.

Ještě horší je, že pomstychtivý útočník, který minul milník, mohl násilně poslat 10 etherů (nebo ekvivalentní množství etheru, které zvedne zůstatek kontraktu and `finalMileStone`), což by zamklo všechny odměny ve smlouvě navždy. Je to proto, že funkce pro nárokování odměny `claimReward` bude vždy neúspěšně ukončena, kvůli require na řádku 32 (tj. protože `this.balance` is vyšší než `finalMileStone`).

Peventivní techniky

Tento druh zranitelnosti obvykle vyplývá ze zneužití `this.balance`. Logika kontraktu, pokud je to možné, by se měla vyhnout závislosti na přesné hodnotě zůstatku kontraktu, protože s ním může být uměle manipulováno. Pokud používáte logiku založenou na `this.balance`, musíte se vyrovnat s neočekávanými zůstatky.

Pokud jsou vyžadovány přesné hodnoty uloženého etheru, je definována proměnná, která by se měla používat pro načítání hodnot získaných platby přijímající funkcí, pro sledování uloženého etheru. Tato proměnná nebude ovlivněna vynuceným zasláním etheru pomocí volání `selfdestruct`.

S ohledem na to mohla opravená verze kontraktu `EtherGame` mohla vypadat jako:

```
contract EtherGame {

    uint public payoutMileStone1 = 3 ether;
    uint public mileStone1Reward = 2 ether;
    uint public payoutMileStone2 = 5 ether;
    uint public mileStone2Reward = 3 ether;
    uint public finalMileStone = 10 ether;
    uint public finalReward = 5 ether;
    uint public depositedWei;

    mapping (address => uint) redeemableEther;

    function play() external payable {
        require(msg.value == 0.5 ether);
        uint currentBalance = depositedWei + msg.value;
        // zajistit, aby po dokon¬ení hry žádný hrá¬ již nehrál
        require(currentBalance <= finalMileStone);
        if (currentBalance == payoutMileStone1) {
            redeemableEther[msg.sender] += mileStone1Reward;
        }
        else if (currentBalance == payoutMileStone2) {
            redeemableEther[msg.sender] += mileStone2Reward;
        }
        else if (currentBalance == finalMileStone ) {
            redeemableEther[msg.sender] += finalReward;
        }
        depositedWei += msg.value;
        return;
    }

    function claimReward() public {
        // ujist¬te se, že je hra dokon¬ena
        require(depositedWei == finalMileStone);
        // ujist¬te se, že je na vyplacení odm¬ny
        require(redeemableEther[msg.sender] > 0);
        redeemableEther[msg.sender] = 0;
        msg.sender.transfer(transferValue);
    }
 }
```

Zde jsme vytvořili novou proměnnou `depositedWei`, která sleduje množství vloženého etheru, a

tuto proměnou používáme pro testování podmínek. Dále již nepoužíváme odkaz na `this.balance`.

Další příklady

Je uvedeno několik příkladů zneužitelných kontraktů v Underhanded Solidity Coding Contest [https://github.com/Arachnid/uscc/tree/master/submissions-2017/], která také poskytuje rozšířené příklady řady nástrah uvedených v této sekci..

DELEGATECALL

Instrukce `CALL` a `DELEGATECALL` jsou užitečné, aby umožnili Ethereum vývojářům rozdělit jejich kód na moduly. Standardní externí volání zprávy do kontraktů je řešeno instrukcí `CALL`, přičemž kód je spuštěn v kontextu externího kontraktu / funkce. Instrukce `DELEGATECALL` je téměř identická, až na to, že kód provedený na cílové adrese běží v kontextu volajícího kontraktu, a `msg.sender` a `msg.value` zůstávají nezměněny. Tato funkcionalita umožňuje implementaci *knihoven*, umožňující vývojářům nasadit opakovaně použitelná kód a volat ho z budoucích kontraktů.

Ačkoli rozdíly mezi těmito dvěma instrukcemi jsou jednoduché a intuitivní, použití `DELEGATECALL` může vést k neočekávanému provádění kódu.

Další čtení viz Loi.Luu Ethereum Stack Exchange otázky na toto téma [http://bit.ly/2AAElb8] a Solidity dokumentace [http://bit.ly/2Oi7UlH].

Zranitelnost

V důsledku kontext zachovávající povahy `DELEGATECALL`, tvorba uživatelských knihoven bez zranitelnosti není tak snadná, jak si člověk může myslet. Kód v knihovnách samotných může být bezpečný a bez zranitelnosti; nicméně při spuštění v kontextu jiné aplikace může dojít k novým zranitelnostem. Uvidíme docela složitý příklad toho, při použití Fibonacciho čísel.

Uvažujme knihovnu v FibonacciLib.sol, která může generovat Fibonacciho posloupnosti a posloupnosti v podobném tvaru (Poznámka: tento kód byl změněn z https://bit.ly/2MReuii.)

Example 12. FibonacciLib.sol

```
// knihovní kontrakt - po¬ítá ¬ísla podobná Fibonacciho
contract FibonacciLib {
    // inicializace standardní Fibonacciho posloupnosti
    uint public start;
    uint public calculatedFibNumber;

    // zm¬na 0-tého ¬ísla v posloupnosti
    function setStart(uint _start) public {
        start = _start;
    }

    function setFibonacci(uint n) public {
        calculatedFibNumber = fibonacci(n);
    }

    function fibonacci(uint n) internal returns (uint) {
        if (n == 0) return start;
        else if (n == 1) return start + 1;
        else return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Knihovna poskytuje funkčnost, která generuje n-té Fibonacciho číslo v posloupnosti. To umožňuje uživatelům změnit počáteční číslo posloupnosti (`start`) a spočítat the *n*-té Fibonacciho číslům podobné číslo v této nové posloupnosti.

Podívejme se nyní náš kontrakt, která tuto knihovnu využívá, zobrazený v FibonacciBalance.sol.

Example 13. FibonacciBalance.sol

```
contract FibonacciBalance {

    address public fibonacciLibrary;
    // aktuální Fibonacciho ¬íslo na výb¬r
    uint public calculatedFibNumber;
    // Po¬áte¬ní ¬íslo Fibonacci posloupnosti
    uint public start = 3;
    uint public withdrawalCounter;
    // Výb¬r Fibonacciho funkce
    bytes4 constant fibSig = bytes4(sha3("setFibonacci(uint256)"));

    // Konstruktor - naplní kontrakt etherem
    constructor(address _fibonacciLibrary) external payable {
        fibonacciLibrary = _fibonacciLibrary;
    }

    function withdraw() {
        withdrawalCounter += 1;
        // Spo¬te Fibonacciho ¬íslo pro aktuálního vybírajícího
uživatele-
        //Nastaví spo¬ítané Fibonacciho ¬íslo calculatedFibNumber
        require(fibonacciLibrary.delegatecall(fibSig,
withdrawalCounter));
        msg.sender.transfer(calculatedFibNumber * 1 ether);
    }

    // umož¬uje uživatel¬m volat knihovnu Fibonacciho funkcí
    function() public {
        require(fibonacciLibrary.delegatecall(msg.data));
    }
}
```

Tento kontrakt umožňuje účastníkovi vybrat ether z kontraktu, jehož množství odpovídá Fibonacciho číslu odpovídajícího pořadovému číslo výběru účastníka. Tj. první účastník získá 1 ether, druhý také 1, třetí 2, čtvrtý dostane 3, pátý 5, a tak dále (dokud zůstatek kontraktu je vyšší než Fibonacciho číslo, které je vybíráno).

V tomto kontraktu je několik prvků, které mohou vyžadovat nějaké vysvětlení. Zaprvé existuje zajímavá proměnná `fibSig`. Ta uchovává 4 bajty Keccak-256 (SHA-3) haše etězce `'setFibonacci(uint256)'`. To je známé jako selektor funkce [http://bit.ly/2RmueMP] a je vložena do `calldata`, aby specifikovala, která funkce chytrého kontraktu bude zavolána. To je používáno funkcí `delegatecall` na řádku 21 pro specifikaci, že chceme spustit funkci `fibonacci(uint256)`. Druhý parametr v `delegatecall` je parametr, který předáváme volané funkci. Za druhé, předpokládáme, že adresa pro `FibonacciLib` knihovnu je správně zadána v konstruktoru. (Externí odkazy na kontrakty diskutuje některé potenciální zranitelnosti související s tímto druhem nastavování odkazu na kontrakt).

Vidíte v tomto kontraktu nějaké chyby? Pokud by někdo měl tento kontrakt nasadit, naplnit jej etherem a zavolejte `withdraw`, pravděpodobně by se neúspěšně vrátil.

Možná jste si všimli, že stavová proměnná `start` se používá jak v knihovně, tak i v hlavním volajícím kontraktu. V knihovním kontraktu `start` se používá k určení začátku Fibonacciho sekvence a je nastavena na `0`, zatímco je nastavena na `3` ve volajícím kontraktu. Možná si také všimli, že nouzová funkce v `FibonacciBalance` kontraktu umožňuje předávání všech volání do knihovního kontraktu, který umožňuje funkci `setStart` knihovního kontraktu, aby byla zavolána. Opakované volání zachovává stav kontraktu, může se zdát, že tato funkce vám umožní změnit stav proměnné `start` v místním `FibonnacciBalance` kontraktu. Pokud ano, umožnilo by to jednomu z nich odebrat další ether, což by mělo za následek `calculatedFibNumber` je závislé na proměnné `start` (jak vidíme v knihovním kontraktu). Ve skutečnosti, funkce `setStart` nemodifikuje (a ani nemůže) proměnou `start` v `FibonacciBalance` kontraktu. Základní zranitelnost tohoto kontraktu je výrazně horší než jen modifikace proměnné `start`.

Než budeme diskutovat o skutečném problému, pojďme rychle odbočit pochopit, jak se vlastně stavové proměnné ukládají v kontraktech. stavové proměnné nebo proměnné v úložišti (proměnné, které přetrvávají jednotlivé transakce) jsou uloženy ve *slotech* postupně, jak jsou uvedeny v kontraktu. (Jsou zde nějaké komplikace, podívejte se na Solidity dokumentaci [http://bit.ly/2JslDWf] pro hlubší pochopení.)

Jako příklad se podívejme na knihovní kontrakt. Má dvě stavové proměnné, `start` a `calculatedFibNumber`. První proměnná `start`, je uložena v úložišti kontraktu v `slot[0]` (tj. první slot). Druhá proměnná `calculatedFibNumber`, je umístěna v druhém možném slotu úložiště `slot[1]`. Funkce `setStart` vezme ze vstupu a nastaví `start` na hodnotu Tato funkce proto nastavuje `slot[0]` na jakýkoli vstup poskytnutý funkcí `setStart`. Podobně Funkce `setFibonacci` nastaví `calculatedFibNumber` na výsledek `fibonacci(n)`. Znovu, to je prostě nastavení úložiště

`slot[1]` na hodnotu `fibonacci(n)`.

Nyní se podívejme na kontrakt `FibonacciBalance`. Úložiště `slot[0]` nyní odpovídá adrese `fibonacciLibrary`, a `slot[1]` odpovídá `calculatedFibNumber`. Tato zranitelnost nastává kvůli tomuto nesprávnému mapování. `delegatecall` *zachovává kontext kontraktu*. To znamená, že kód, který je vykonán `delegatecall` pracuje se stavem (tj. úložištěm) volajícího kontraktu.

Nyní si všimněte, že v `withdraw` na řádku 21 provedeme `fibonacciLibrary.delegatecall(fibSig,withdrawalCounter)`. To volá Funkci `setFibonacci`, která jak jsme diskutovali modifikuje úložiště `slot[1]`, což je v našem současném kontextu `calculatedFibNumber`. To jak je očekáváno (tj. po provedení `calculatedFibNumber` změněno). Nezapomeňte však, že proměnná `start` v `FibonacciLib` kontraktu je uložena v úložišti `slot[0]`, který je adresou `fibonacciLibrary` v současném kontraktu. To znamená, že funkce `fibonacci` bude vracet neočekávané výsledky. To je protože odkazuje na `start` (`slot[0]`), který v aktuálním kontextu volání je adresa `fibonacciLibrary` (která bude často docela velká, když je interpretována jako `uint`). Je tedy pravděpodobné, že funkce `withdraw` bude neúspěšně vrácena, protože není k dispozici `uint(fibonacciLibrary)` množství etheru, které vrátí funkce `calculatedFibNumber`.

Ještě hůře, `FibonacciBalance` kontrakt umožňuje uživatelům volat všechny funkce `fibonacciLibrary` pomocí záložní funkce na řádku 26. Jak jsme uvedli dříve, to zahrnuje funkci `setStart`. Diskutovali, že tato funkce umožňuje komukoli upravit nebo nastavit úložiště `slot[0]`. V tomto případě, úložiště `slot[0]` je adresa `fibonacciLibrary` Útočník by proto mohl vytvořit škodlivý kontrakt, převést adresu na `uint` (to lze udělat snadno v Pythonu použitím `int('<address>',16)`), a poté zavolat `setStart(<attack_contract_address_as_uint>)`. To změní adresu `fibonacciLibrary` na adresu útočníkova kontraktu. Pak kdykoli uživatel volá `withdraw` nebo nouzovou funkci, škodlivý contract bude běžet (což může ukrást celý zůstatek kontraktu) protože jsme upravili skutečnou adresu pro `fibonacciLibrary`. Příkladem takového kontraktu by byl:

```
contract Attack {
    uint storageSlot0; // odpovídá fibonacciLibrary
    uint storageSlot1; // odpovídá calculatedFibNumber

    // nouzová funkce - pob¬ží, pokud nebude nalezena zadaná funkce
    function() public {
        storageSlot1 = 0; // nastavíme calculatedFibNumber na 0, takže
pokud výb¬r
        // je zavolán, neodešleme žádný ether
        <attacker_address>.transfer(this.balance); // vezmeme všechen
ether
    }
 }
```

Všimněte si, že tento útočný kontrakt mění `calculatedFibNumber` změnou úložiště `slot[1]`. Útočník by v zásadě mohl upravit další úložné sloty, které si vybere, aby na tom provedl všechny druhy útoků na tento kontrakt. Doporučujeme vám tyto kontrakty vložit do ttps://remix.ethereum.org[Remix] a experimentovat s různými útočnými kontrakty a změnami stavu pomocí těchto delegatecall` funkcí.

Je také důležité si toho všimnout, když to říkáme `delegatecall` je uchovávající stav, nemluvíme o názvech proměnných kontraktu, ale spíše o skutečných paměťových slotech, na které tyto názvy odkazují. Jak můžete vidět z tohoto příkladu, jednoduchá chyba může vést k tomu, že útočník unese celý kontrakt a jeho ether.

Peventivní techniky

Solidity poskytuje klíčové slovo `library` pro implementaci knihovovního kontraktu(viz dokumentace [http://bit.ly/2zjD8TI] pro další podrobnosti). To zajišťuje, že knihovní kontrakt je beze stavu a nemůže se sám zničit Nutí knihovny být beze stavu zmírňuje složitost úložného kontextu prokázanou v této sekci. Knihovny beze stavu také zabraňují útokům, při kterých útočník přímo upravuje stav knihovny, aby ovlivnil kontrakty, které závisí na kódu knihovny. Obecně platí, při používání `DELEGATECALL` věnujte pozornost možnému kontextu volání jak knihovního kontraktu, tak i volajícího kontraktu. a kdykoli je to možné, stavět bezestavové knihovny.

Příklad ze života: Parity vícepodpisová peněženka (druhé napadení)

Druhé napadení Parity vícepodpisové peněženky je příkladem jak dobře napsaná knihovna může být napadena, pokud běží mimo zamýšlený kontext. Existuje mnoho dobrých vysvětlení toto napadení, jako je "Znovu napadena Parity vícepodpisová peněženka" [http://bit.ly/2Dg7GtW] a "Hluboký pohled na chyby Parity vícepodpisové peněženky" [http://bit.ly/2Of06B9].

Chcete-li přidat tyto reference, prozkoumejme kontrakty, které byly úspěšně napadeny. Knihovní a peněženkové kontrakty naleznete na GitHubu [http://bit.ly/2OgnXQC].

Knihovní kontrakt je následující:

```
contract WalletLibrary is WalletEvents {

  ...

  // vyhodí výjimku dokud není kontrakt inicializován.
  modifier only_uninitialized { if (m_numOwners > 0) throw; _; }

  // konstruktor - pouze projde pole vlastník¬ pro více vlastnictví a
  // omezení denním limitem
  function initWallet(address[] _owners, uint _required, uint _daylimit)
      only_uninitialized {
    initDaylimit(_daylimit);
    initMultiowned(_owners, _required);
  }

  // zabije kontrakt zasláním všeho na `_to`.
  function kill(address _to) onlymanyowners(sha3(msg.data)) external {
    suicide(_to);
  }

  ...

}
```

A tady je peněženkový kontrakt

```
contract Wallet is WalletEvents {

  ...

  // METODY

  // vyvolá se, když neodpovídá žádná jiná funkce
  function() payable {
    // práv¬ vám byly zaslány n¬jaké peníze?
    if (msg.value > 0)
      Deposit(msg.sender, msg.value);
    else if (msg.data.length > 0)
      _walletLibrary.delegatecall(msg.data);
  }

  ...

  // DATOVÉ POLOŽKY
  address constant _walletLibrary =
    0xcafecafecafecafecafecafecafecafecafecafe;
}
```

Všimněte si, že kontrakt `Wallet` v podstatě přesměruje všechna volání do knihovního kontraktu `WalletLibrary` pomocí volání delegate. Konstanta adresa `_walletLibrary` v této ukázce kódu funguje jako zástupný symbol pro aktuálně nasazený knihovní kontrakt `WalletLibrary` (která je `0x863DF6BFa4469f3ead0bE8f9F2AAE51c91A907b4`).

Zamýšlené fungování těchto kontraktů mělo mít jednoduché nízkonákladové nasaditelné peněženkové kontrakty, jejichž kódy a hlavní funkcionalita byla v kontraktu `WalletLibrary`. Bohužel, kontrakt `WalletLibrary` je sama o sobě kontraktem a udržuje svůj vlastní stav Vidíte, vidíte, proč by to mohl být problém?

Je možné odesílat volání `WalletLibrary` samotným kontraktem. Konkrétně, `WalletLibrary` kontrakt by mohl být inicializován a stát se vlastněným. Ve skutečnosti to uživatel provedl a zavolal `initWallet` funkci v `WalletLibrary` kontraktu a stal se majitelem knihovního kontraktu. stejný uživatel následně zavolall funkci `kill`. Protože uživatel byl vlastníkem knihovního kontraktu, modifikátor prošel a knihovní kontrakt se zničil. Jak všechny existující `Wallet` kontrakty odkazovali na tento knihovní kontrakt a neobsahují žádnou metodu pro změnu tohoto odkazu, veškerá jejich funkčnost, včetně schopnosti vybírat ether, byla ztracena spolu s kontraktem „WalletLibrary“.

Výsledkem je veškerý ether ve všech parity vícepodpisových peněženkách tohoto typu okamžitě ztracený nebo trvale nedostupný.

Výchozí viditelnost

Funkce v Solidity mají specifikátory viditelnosti, které diktují jak mohou být volány. Viditelnost určuje, zda funkce mohou uživatelé volat externě, jinými odvozenými kontrakty, pouze interně, nebo pouze externě. Existují čtyři specifikátory viditelnosti, které jsou podrobně popsány v Solidity dokumentaci [http://bit.ly/2ABiv7j]. Výchozí specifikátor funkce je `public`, umožňující uživatelům volat je externě. Nyní uvidíme, jak nesprávné použití specifikátorů viditelnosti může vést k některým devastujícím zranitelnostem v chytrých kontraktech

Zranitelnost

Výchozí viditelnost funkcí je `public`, takže funkce které neuvádějí svojí viditelnost, budou moci volat externí uživatelé. Problém nastává, když vývojáři omylem vynechají specifikace viditelnosti funkcí, které by měly být soukromé (nebo viditelné pouze v rámci samotného kontraktu).

Pojďme rychle prozkoumat triviální příklad:

```
contract HashForEther {

    function withdrawWinnings() {
        // Vít¬z, pokud posledních 8 hex znak¬ adresy je 0
        require(uint32(msg.sender) == 0);
        _sendWinnings();
     }

     function _sendWinnings() {
         msg.sender.transfer(this.balance);
     }
}
```

Tento jednoduchý kontrakt je navržena tak, aby fungoval jako hra odměnující za odhady adres Pro získání zůstatku kontraktu musí uživatel vygenerovat Ethereum adresu, jejíž posledních 8 hexadecimálních znaků je 0. Jakmile toho dosáhne může zavolat funkci „drawWinnings“ a získat tak odměnu.

Viditelnost funkcí bohužel nebyla stanovena. Zejména funkce `_sendWinnings` je `public` (výchozí), a tedy libovolná adresa může volat tuto funkci a ukrást odměnu.

Peventivní techniky

Je dobrou praxí vždy specifikovat viditelnost všech funkcí v kontraktu, i když jsou úmyslně `public`. Poslední verze solc zobrazuje varování pro funkce, které nemají explicitně nastavenou viditelnost, aby tuto praxi podpořili.

Příklad ze života: Parity vícepodpisová peněženka (první napadení)

V prvním napadení Parity vícepodpisové peněženky bylo ukradeno Ethereum v hodnotě okolo $31M, většinou ze tří peněženek. Dobrá rekapitulace toho, jak přesně se to stalo dává Haseeb Qureshi [https://bit.ly/2vHiuJQ].

V podstatě, vícepodpisová peněženka je vytvořen ze základního kontraktu `Wallet`, který volá knihovnu kontraktu obsahující základní funkčnost (jak je popsáno v Příklad ze života: Parity vícepodpisová peněženka (druhé napadení)). Knihovní kontrakt obsahuje kód pro inicializaci peněženky jak můžeme vidět z následujícího úryvku:

```
contract WalletLibrary is WalletEvents {

  ...

  // METODY

  ...

  // konstruktoru je dán po¬et podpis¬ pot¬ebných k zajišt¬ní ochrany
  // "onlymanyowners" transakce, stejn¬ jako výb¬r adres
  // schopné je potvrdit
  function initMultiowned(address[] _owners, uint _required) {
    m_numOwners = _owners.length + 1;
    m_owners[1] = uint(msg.sender);
    m_ownerIndex[uint(msg.sender)] = 1;
    for (uint i = 0; i < _owners.length; ++i)
    {
      m_owners[2 + i] = uint(_owners[i]);
      m_ownerIndex[uint(_owners[i])] = 2 + i;
    }
    m_required = _required;
  }

  ...

  // konstruktor - pouze projde pole vlastník¬ pro více vlastnictví a
  // omezení denním limitem
  function initWallet(address[] _owners, uint _required, uint _daylimit) {
    initDaylimit(_daylimit);
    initMultiowned(_owners, _required);
  }
}
```

Všimněte si, že žádná z funkcí nespecifikuje jejich viditelnost, obě ji mají výchozí `public`. Funkce `initWallet` se nazývá konstruktor peněženky a nastavuje majitele vícepodpisové peněženky, jak je vidět ve funkci `initMultiowned`. Protože tyto funkce byly náhodně ponechány `public`,, útočník byl schopen volat tyto funkce na nasazených kontraktech, nastavením vlastnictví na adresu útočníka. Když byl vlastníkem, útočník vyčerpal z peněženek všechen jejich ether.

Iluze entropie

("security (smart contracts)","entropy illusion threat"Všechny transakce na Ethereum bločence jsou deterministické operace přechodu stavu. To znamená, že každá transakce modifikuje globální stav ekosystému Etherea vypočítatelným způsobem, bez nejistoty. To má zásadní důsledek, že v Ethereu není žádný zdroj entropie nebo náhodnosti. Dosažení decentralizované entropie (náhodnosti) je známý problém, pro který bylo navrženo mnoho řešení, včetně RANDAO [https://github.com/randao/randao], nebo použití hašů bločenky, jak popsal Vitalik Buterin v blogu "Validátor pořadí a náhodnosti v PoS" [https://vitalik.ca/files/randomness.html].

Zranitelnost

Některé z prvních kontraktů vytvořených na platformě Ethereum byly založeny kolem hazardu. Hazardní hry v zásadě vyžadují nejistotu (něco na co vsadit), což dělá stavění hazardního systému na bločence (a deterministickém systému) poněkud obtížným. Je jasné, že nejistota musí pocházet ze zdroje externího k bločence. To je možné pro Sázky mezi hráči (viz např. technika odevzdání a odhalení [http://bit.ly/2CUh2KS]); je však výrazně obtížnější, pokud chcete implementovat kontrakt, aby fungoval jako kasíno (jako blackjack nebo ruleta). Společným úskalím je použití budoucích blokových proměnných, tj. proměnné obsahující informace o transakčním bloku, jehož hodnoty ještě nejsou známy, například haše, časové značky, čísla bloků nebo limity plynu. Problém s nimi je že jsou ovládáni těžařem, který vytěží blok, a jako takové nejsou opravdu náhodné. Zvažte například chytrý kontrakt ruleta s logikou, která vrací černé číslo, pokud další haš bloku končí na sudé číslo. Těžař (nebo těžařská skupina) mohli vsadit $ 1M na černou. Pokud oni vyřeší další blok a najdou haš končící na liché číslo, mohou naštěstí nezveřejňují svůj blok a těžit další, dokud nenajdou řešení s hašem bloku, který končí na sudé číslo (za předpokladu, že blok odměna a poplatky jsou nižší než 1 milion USD). Používání minulých nebo současných proměnných může být ještě ničivější, jak prokazuje Martin Swende ve svém vynikajícím příspěvku na blogu [http://martin.swende.se/blog/Breaking_the_house.html]. Navíc použití výhradně blokových proměnných znamená, že pseudonáhodné číslo bude stejné pro všechny transakce v bloku, tedy útočník může znásobit své výhry provedením mnoha transakcí v rámci bloku (mělo by být omezení na maximální sázku).

Peventivní techniky

Zdroj entropie (náhodnost) musí být mimo bločenku. To lze provést mezi vrstevníky v systémech, jako je odevzdat–odhalit [http://bit.ly/2CUh2KS], nebo změnou modelu důvěry na skupinu účastníků

(jako v RandDAO [https://github.com/randao/randao]). To lze také provést pomocí centralizované entity, která funguje jako orákulum náhodnosti. Blokové proměnné (obecně existují některé výjimky) by neměly být použity jako zdroj entropie, protože mohou být manipulovány těžaři

Příklad ze života: PRNG kontrakt

V únoru 2018 Arseny Reutov blogoval [http://bit.ly/2Q589lx] o jeho analýze 3 649 živých chytrých kontraktů, které využívaly nějaký druh generátoru pseudonáhodných čísel (PRNG); našel 43 kontraktů, které by mohly být napadeny.

Externí odkazy na kontrakty

Jednou z výhod Ethereum světového počítače je schopnost znovu použít kód a komunikovat s kontrakty již nasazenými v síti. Výsledkem je, že velký počet kontraktů odkazuje na externí kontrakty, obvykle prostřednictvím externího volání zpráv. Tato externí volání zpráv mohou maskovat škodlivé záměry účastníků některými nenápadnými způsoby, které nyní prozkoumáme.

Zranitelnost

V Solidity lze libovolnou adresu dosadit do kontraktu bez ohledu na to, zda kód na adrese představuje typ dosazovaného kontraktu. Tento může způsobit problémy, zejména pokud se autor kontraktu snaží skrýt škodlivý kód. Pojďme to ilustrovat příkladem.

Zvažte kousek kódu jako Rot13Encryption.sol, který neohrabaně implementuje ROT13 cipher [https://en.wikipedia.org/wiki/ROT13].

Example 14. Rot13Encryption.sol

```
// šifrovací kontrakt
contract Rot13Encryption {

   event Result(string convertedString);

    // rot13-šifrování ¬et¬zce
    function rot13Encrypt (string text) public {
        uint256 length = bytes(text).length;
        for (var i = 0; i < length; i++) {
            byte char = bytes(text)[i];
            // vkládaný asembler pro úpravu ¬et¬zce
            assembly {
                // vrátí první bajt
                char := byte(0,char)
                // pokud je první znak [n,z], tj. p¬eto¬ení
                if and(gt(char,0x6D), lt(char,0x7B))
                // ode¬te ASCII hodnotu 'a',
                // rozdíl mezi znakem <char> a 'z'
                { char:= sub(0x60, sub(0x7A,char)) }
                if iszero(eq(char, 0x20)) // ignoruje mezery
                // p¬i¬te 13 do znaku
                {mstore8(add(add(text,0x20), mul(i,1)), add(char,13))}
            }
        }
        emit Result(text);
    }

    // rot13-dešifrování ¬et¬zce
    function rot13Decrypt (string text) public {
        uint256 length = bytes(text).length;
        for (var i = 0; i < length; i++) {
            byte char = bytes(text)[i];
            assembly {
                char := byte(0,char)
                if and(gt(char,0x60), lt(char,0x6E))
                { char:= add(0x7B, sub(char,0x61)) }
                if iszero(eq(char, 0x20))
                {mstore8(add(add(text,0x20), mul(i,1)), sub(char,13))}
            }
```

```
        }
        emit Result(text);
    }
}
```

Tento kód jednoduše vezme řetězec (písmen a až z, bez ověření) and *zašifruje* ho posunem každého znaku o 13 míst doprava (přetočení okolo z) around z) tj. a posune na n a x posune na k. Sestavení v předcházejícím kontraktu nepotřebuje být pochopena, abychom ocenili problém, který diskutujeme, takže čtenáři neobeznámení se sestavením to mohou bezpečně ignorovat.

Nyní zvažte následující kontrakt, který používá tento kód pro jeho šifrování:

```
import "Rot13Encryption.sol";

// šifruje vaše nejv¬tší tajemství
contract EncryptionContract {
    // knihovna pro šifrování
    Rot13Encryption encryptionLibrary;

    // konstruktor - inicializuje knihovnu
    constructor(Rot13Encryption _encryptionLibrary) {
        encryptionLibrary = _encryptionLibrary;
    }

    function encryptPrivateData(string privateInfo) {
        // zde m¬žete provád¬t n¬které operace
        encryptionLibrary.rot13Encrypt(privateInfo);
    }
}
```

Problém s tímto kontraktem spočívá v tom, že adresa `encryptionLibrary` není ne veřejná nebo konstantní. Ten, kdo smlouvu nasadil, by tedy mohl v konstruktoru uvést adresu, která odkazuje na tuto smlouvu:

```
// šifrovací kontrakt
contract Rot26Encryption {

   event Result(string convertedString);
```

```
    // rot13-šifrování ¬et¬zce
    function rot13Encrypt (string text) public {
        uint256 length = bytes(text).length;
        for (var i = 0; i < length; i++) {
            byte char = bytes(text)[i];
            // vkládaný asembler pro úpravu ¬et¬zce
            assembly {
                // vrátí první bajt
                char := byte(0,char)
                // pokud je první znak [n,z], tj. p¬eto¬ení
                if and(gt(char,0x6D), lt(char,0x7B))
                // ode¬te ASCII hodnotu 'a',
                // rozdíl mezi znakem <char> a 'z'
                { char:= sub(0x60, sub(0x7A,char)) }
                // ignoruje mezery
                if iszero(eq(char, 0x20))
                // p¬i¬te 26 ke znaku!
                {mstore8(add(add(text,0x20), mul(i,1)), add(char,26))}
            }
        }
        emit Result(text);
    }

    // rot13-dešifrování ¬et¬zce
    function rot13Decrypt (string text) public {
        uint256 length = bytes(text).length;
        for (var i = 0; i < length; i++) {
            byte char = bytes(text)[i];
            assembly {
                char := byte(0,char)
                if and(gt(char,0x60), lt(char,0x6E))
                { char:= add(0x7B, sub(char,0x61)) }
                if iszero(eq(char, 0x20))
                {mstore8(add(add(text,0x20), mul(i,1)), sub(char,26))}
            }
        }
        emit Result(text);
    }
}
```

Tento kontrakt implementuje šifru ROT26, která posune každý znak o 26 míst (tj. nedělá nic). Opět

není třeba pochopit sestavení v tomto kontraktu Zjednodušeně, útočník mohl propojit následující kontrakt se stejným účinkem:

```
contract Print{
    event Print(string text);

    function rot13Encrypt(string text) public {
        emit Print(text);
    }
 }
```

Pokud byla adresa některého z těchto kontraktů uvedena v konstruktoru, `encryptPrivateData` funkce ubde jednoduše vytvářet událost vypisující nezašifrovaná soukromá data.

I když v tomto příkladu knihovního kontraktu byl nastaven konstruktor, je to často v případě, že privilegovaný uživatel (například vlastník) může změnit adresu knihovního kontraktu. Pokud napojený kontrakt tuto volanou funkci neobsahuje nouzová funkce je vykonána. Například, s řádkou `encryptionLibrary.rot13Encrypt()`, pokud kontrakt specifikován `encryptionLibrary` byl:

```
 contract Blank {
     event Print(string text);
     function () {
         emit Print("Here");
         // sem vložte škodlivý kód a bude spušt¬n
     }
 }
```

pak by byla vyslána událost s textem `Here`. Pokud tedy uživatelé mohou změnit knihovní kontrakt, mohou v zásadě přimět ostatní uživatele, aby nevědomky spustili libovolný kód.

Zde uvedené kontrakty jsou pouze pro demonstrační účely a nepředstavují správné šifrování. Neměly by být použity šifrování.

Peventivní techniky

Jak již bylo dříve prokázáno, bezpečné kontrakty mohou (v některých případech) být nasazeny tak, aby se chovaly škodlivě. Auditor mohl veřejně ověřit kontrakt a nechat jejího majitele, aby ho

nasadil škodlivým způsobem, což má za následek veřejně auditovaný kontrakt, který má slabé stránky nebo zákeřný záměr.

Těmto scénářům brání řada technik.

Jednou z technik je použití klíčového slova `new` pro vytváření kontraktů. V předchozím příkladě, konstruktor může být napsán jako:

```
constructor() {
    encryptionLibrary = new Rot13Encryption();
}
```

Tímto způsobem se při nasazení vytvoří instance odkazovaného kontraktu a osoba, která provedla nasazení, nemůže vyměnit `Rot13Encryption` kontrakt bez jeho změnění.

Dalším řešením je natvrdo uvést v kódu adresy externích chytrých kontraktů.

Obecně by kód, který volá externí kontrakty, měl být vždy pečlivě prověřen. Pro vývojáře, při definování externích kontraktů, může být dobrý nápad udělat adresy kontraktu veřejné (což není případ příkladu hrnce medu v následující sekci) case in the example in the following section) aby uživatelé mohli snadno zkoumat kód uvedený v kontraktu. Naopak, pokud má kontrakt soukromou proměnou adresy kontraktu, může to být přiznak adresa variabilní smlouvy může být znakem, že se někdo chová škodlivě (jak je uvedeno v příkladu reálného světa). Pokud uživatel může změnit adresu kontraktu, která se používá pro volání externích funkcí, může to být důležité (v kontextu decentralizovaného systému) implementovat mechanismus časového blokování a / nebo hlasování, který uživatelům umožní podívat se, jaký kód se mění, nebo dát účastníkům šanci se rozhodnout pro ukončení užívání tohoto kontraktu

Příklad ze života: Hrnec medu s vícenásobným voláním

V poslední době na hlavní síti bylo vydáno množství hrnců medu (honey pot). Tyto kontrakty se snaží přechytračit Ethereum hackery, kteří se snaží využívat chyb kontraktů, ale nakonec jsou to oni, kdo ztrácí ether v kontraktu, který plánovali vykrást. Jedním příkladem zahrnujícím tento útok náhradou očekávaného kontraktu, zlomyslným kontraktem v konstruktoru. Kód lze najít zde [http://bit.ly/2JtdqRi]:

```
pragma solidity ^0.4.19;

contract Private_Bank
{
    mapping (address => uint) public balances;
    uint public MinDeposit = 1 ether;
    Log TransferLog;

    function Private_Bank(address _log)
    {
        TransferLog = Log(_log);
    }

    function Deposit()
    public
    payable
    {
        if(msg.value >= MinDeposit)
        {
            balances[msg.sender]+=msg.value;
            TransferLog.AddMessage(msg.sender,msg.value,"Deposit");
        }
    }

    function CashOut(uint _am)
    {
        if(_am<=balances[msg.sender])
        {
            if(msg.sender.call.value(_am)())
            {
                balances[msg.sender]-=_am;
                TransferLog.AddMessage(msg.sender,_am,"CashOut");
            }
        }
    }

    function() external payable{}

}

contract Log
{
```

```
    struct Message
    {
        address Sender;
        string  Data;
        uint Val;
        uint  Time;
    }

    Message[] public History;
    Message LastMsg;

    function AddMessage(address _adr,uint _val,string _data)
    public
    {
        LastMsg.Sender = _adr;
        LastMsg.Time = now;
        LastMsg.Val = _val;
        LastMsg.Data = _data;
        History.push(LastMsg);
    }
}
```

Tento text [http://bit.ly/2Q58VyX] jednoho uživatele reddit vysvětluje jak přišel o 1 ether v tomto kontraktu tím, že se pokusili využít chybu vícenásobného volání, kterou očekávali, že bude přítomna v kontraktu.

Krátká adresa / útok na parametry

Tento útok se neprovádí u Solidity kontraktů samotných, ale v aplikacích třetích stran, které s nimi mohou interagovat. Tato sekce je přidána pro úplnost a dává čtenáři povědomí o tom, jak mohou být parametry zmanipulované ve kontraktech.

Další čtení viz "Vysvětlení ERC20 útoku krátkou adresou" [http://bit.ly/2yKme14], "Zranitelnost ICO chytrých kontraktů: útok krátkou adresou" [http://bit.ly/2yFOGRQ], nebo tento Reddit příspěvek [http://bit.ly/2CQjBhc].

Zranitelnost

Při předávání parametrů do chytrého kontraktu jsou parametry kódovány podle ABI specifikace [http://bit.ly/2Q5VIG9]. Je možné poslat zakódované parametry, které jsou kratší než očekávaná délka parametru (například odeslání adresy, která je pouze 38 hex znaků (19 bajtů) místo standardních 40 hex znaků (20 bajtů). V takovém případě EVM přidá nuly na konec kódovaných parametrů pro vytvoření očekávané délky.

To se stává problémem, když aplikace třetích stran neověřují vstupy. Nejjasnějším příkladem je burza, která neověřuje adresu ERC20 tokenu když uživatel požaduje výběr. Tento příklad je podrobněji popsán v příspěvku Petera Vessenese "Vysvětlení ERC20 útoků krátkou adresou" [http://bit.ly/2Q1ybpQ].

Uvažujme standard ERC20 [http://bit.ly/2CUf7WG] transfer funkční rozhraní s ohledem na pořadí parametrů:

```
function transfer(address to, uint tokens) public returns (bool success);
```

Nyní zvažte burzu s velkým množstvím tokenů (řekněme `REP`) a uživatele, který si chce vybrat svůj podíl 100 žetonů. Uživatel zadá svou adresu,`0xdeaddeaddeaddeaddeaddeaddeaddeaddeaddead`, a množství tokenů, `100`. Burza by zakódovala tyto parametry v pořadí určeném funkcí `transfer`; to je `address` následovaná `tokens`. Zakódovaným výsledkem by bylo:

```
a9059cbb000000000000000000000000deaddeaddea \
ddeaddeaddeaddeaddeaddead0000000000000
0000000000000000000000000000000056bc75e2d63100000
```

První 4 bajty (`a9059cbb`) jsou `transfer` selektor selektor [http://bit.ly/2RmueMP], dalších 32 bajtů je adresa a posledních 32 bytů představuje počet tokenů `uint256`. Všimněte si, že hex `56bc75e2d63100000` na konci odpovídá 100 tokenům (s 18 desetinnými místy, jak je specifikováno `REP` tokenovým kontraktem).

Podívejme se nyní, co by se stalo, kdyby někdo zaslal adresu s chybějícím 1 bajtem (2 hexadecimální číslice). Konkrétně řekněme útočník zašle `0xdeaddeaddeaddeaddeaddeaddeaddeaddeadde` jako adresu (chybí poslední dvě číslice) a stejných `100` žetony k výběru. Pokud burza neověřuje tento vstup, bude zakódována jako:

```
a9059cbb000000000000000000000000deaddeaddea \
ddeaddeaddeaddeaddeaddeadde00000000000000
000000000000000000000000000000000056bc75e2d6310000000
```

Rozdíl je jemný. Všimněte si, že `00` bylo přidáno na konec kódování, aby doplnilo krátkou adresu, která byla odeslána. Když se to dostane do chytrého kontraktu, parametr `address` bude čten jako `0xdeaddeaddeaddeaddeaddeaddeaddeaddeadde00` a hodnota bude čtena jako `56bc75e2d6310000000` (všimněte si dvou 0 navíc). Tato hodnota je nyní `25600` tokenů (hodnota byla vynásobena `256`). V tomto příkladu, pokud by burza držela tolik tokenů, uživatel by vybral `25600` tokenů (zatímco burza si myslí, že uživatel vybral pouze `100`) na upravenou adresu. Útočník samozřejmě nebude mít adresa v tomto příkladu, ale pokud by měl útočník vygenerovat jakoukoli adresu, která by skončila na 0 (která může být snadno spočtena hrubou silou) a použil tuto vygenerovanou adresu, mohl ukrást tokeny z z nic netušící burzy

Peventivní techniky

Všechny vstupní parametry v externích aplikacích by měly být kontrolovány dříve než jsou odeslány do bločenky. Mělo by být také poznamenáno, že zde hraje důležitou roli uspořádání parametrů. K zarovnání dochází pouze na konci, pečlivé řazení parametrů v chytrém kontraktu může zmírnit některé formy tohoto útoku.

Nekontrolované návratové hodnoty CALL

V Solidity existuje řada způsobů, jak provádět externí volání. Odesílání etheru na externí účet se běžně provádí metodou `transfer`. Lze však také použít funkci `send` a pro více univerzální externí volání instrukce `CALL` může být přímo použita v Solidity. Funkce `call` a `send` vrací Boolean označující, zda volání bylo úspěšné nebo neúspěšné. Tyto funkce tedy mají jednoduché varování že transakce, která vykonává tyto funkce, se nevrátí, pokud externí volání (zahájené `call` nebo `send`)selže, spíše, funkce jednoduše vrátí `false`. Častou chybou je že vývojář očekává, že k navrácení dojde, pokud se externí volání nezdaří a nekontroluje návratovou hodnotu.

Další čtení viz #4 na the DASP Top 10 roku 2018 [http://www.dasp.co/#item-4] a "Scanning Živých Ethereum kontraktů pro chyby nekontrolovaného odeslání" [http://bit.ly/2RnS1vA].

Zranitelnost

Zvažte následující příklad:

```
contract Lotto {

    bool public payedOut = false;
    address public winner;
    uint public winAmount;

    // ... další funkce zde

    function sendToWinner() public {
        require(!payedOut);
        winner.send(winAmount);
        payedOut = true;
    }

    function withdrawLeftOver() public {
        require(payedOut);
        msg.sender.send(this.balance);
    }
}
```

To reprezentuje kontrakt podobný Lotto, kde `winner` obdrží `winAmount` etheru, který obvykle ponechává trochu zbylého etheru pro výběr kýmkoliv jiným

Tato chyba zabezpečení existuje na řádku 11, kde je použit `send` bez kontroly odpovědi. V tomto triviálním příkladu `winner`, jehož transakce selže (buď nedostatkem plynu nebo je to kontrakt, který úmyslně vyhodí výjimku v nouzové funkci) umožní nastavení `payedOut` na `true` bez ohledu na to, zda byl odeslán ether. V takovém případě může kdokoli vybrat odměnu určenou `winner`' pomocí funkce `withdrawLeftOver`.

Peventivní techniky

Kdykoli je to možné, použijte spíše funkci `transfer` než `send`, protože `transfer` se neúspěšně vrátí, pokud se externí transakce neúspěšně vrátí. Pokud `send` požadováno, vždy zkontrolujte vrácenou hodnotu.

Robustnější doporučení [http://bit.ly/2CSdF7y] je přijmout *výběrový vzor*. V tomto řešení musí každý uživatel volat izolovanou withdraw funkci který zpracovává odesílání etheru z kontraktu a vypořádává se s důsledky neúspěšných odesílacích transakcí. Záměrem je logicky izolovat externí funkci odesílání od zbytek zdrojového kódu a zatížit břemenem potenciálního selhání transakci u koncového uživatele, která volá funkci withdraw.

Příklad ze života: Etherpot and King of the Ether

Etherpot [http://bit.ly/2OfHalK] was Loterijní chytrý kontrakt, příliš odlišné od výše uvedeného vzorového kontraktu. Pád tohoto kontraktu byl způsoben především nesprávným použitím hašů bloku (je možné použít pouze posledních 256 hashů bloku, viz Aakil Fernandes text [http://bit.ly/2Jpzf4x] o tom, jak Etherpot toto nezohlednil správně). Nicméně, tento také trpěla nekontrolovanou hodnotou volání. Zvažte funkci `cash` v lotto.sol: úryvek kódu.

Example 15. lotto.sol: úryvek kódu

```
...
  function cash(uint roundIndex, uint subpotIndex){

        var subpotsCount = getSubpotsCount(roundIndex);

        if(subpotIndex>=subpotsCount)
            return;

        var decisionBlockNumber =
getDecisionBlockNumber(roundIndex,subpotIndex);

        if(decisionBlockNumber>block.number)
            return;

        if(rounds[roundIndex].isCashed[subpotIndex])
            return;
        //Vedlejší výhry mohou být proplaceny pouze jednou. Tím se
zabrání dvojím výplatám

        var winner = calculateWinner(roundIndex,subpotIndex);
        var subpot = getSubpot(roundIndex);

        winner.send(subpot);

        rounds[roundIndex].isCashed[subpotIndex] = true;
        //Ozna¬í kolo jako proplacené
}
...
```

Všimněte si, že na řádku 21 není návratová hodnota funkce `send` kontrolována, a následující řádek pak nastaví booleovský znak označující, že výherci byly zaslány prostředky. Tato chyba může povolit stav, ve kterém vítěz nedostal svůj ether, ale stav kontraktu může být přesvědčen, že vítěz již byl vyplacen.

Vážnější verze této chyby se vyskytla v King of the Ether [http://bit.ly/2ACsfi1]. Výborná posmrtná [http://bit.ly/2ESoaub] analýza kontraktu byla napsána, že podrobnosti o tom, jak nekontrolované selhání `send` může být použito k útoku na kontrakt.

Souběh / předbíhání

The Kombinace externích volání jiných kontraktů a víceuživatelská povahy podkladové bločenky vede k různorodosti potenciálních úskalí Solidity, kvůli kterým uživatelé *souběžně* vykonávající kód získají neočekávané stavy. Opětovné volání (diskutováno dříve v této kapitole) je jedním z příkladů takových souběhů (race condition). V této sekci budeme diskutovat jiné druhy souběhů, které se mohou vyskytnout v Ethereum bločence. K tomuto tématu existuje celá řada dobrých příspěvků, včetně Race Conditions on the Ethereum Wiki [http://bit.ly/2yFesFF], #7 z DASP Top10 roku 2018 [http://www.dasp.co/#item-7], a Osvědčené postupy Ethereum chytrých kontraktů [http://bit.ly/2Q6E4lP].

Zranitelnost

Jako u většiny bločenek, Ethereum uzly shromažďují transakce a tvoří z nich bloky. Transakce jsou považovány za platné, pouze pokud těžař vyřešil mechanismus konsensu (v současné době Ethash [http://bit.ly/2yI5Dv7] PoW pro Ethereum). Těžař, který blok řeší, si také vybere, které z nashromážděných transakcí budou zahrnuty do bloku, obvykle je řadí podle `gasPrice` jednotlivých transakcí. Zde je potenciální možnost útoku. Útočník může sledovat shromážděné transakční a vyhledávat ty, které mohou obsahovat řešení problému a měnit nebo odvolávat řešitelovo povolení nebo měnit stav kontraktu škodlivě vůči řešiteli. Útočník pak může získat data z této transakce a vytvořit vlastní transakci s vyšší ` `gasPrice`, tak jeho transakce je zahrnuta v bloku před originálem.

Uvidíme, jak by to mohlo fungovat s jednoduchým příkladem. Zvažte kontrakt uvedený v FindThisHash.sol.

Example 16. FindThisHash.sol

```
contract FindThisHash {
    bytes32 constant public hash =

0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a;

    constructor() external payable {} // naplnit etherem

    function solve(string solution) public {
        // Pokud najdete p¬edobraz haše,  získáte 1000 ether¬
        require(hash == sha3(solution));
        msg.sender.transfer(1000 ether);
    }
}
```

Řekněme, že tento kontrakt obsahuje 1 000 etherů. Uživatel, který může najít předobraz následujícího SHA-3 haše:

```
0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a
```

může odeslat řešení a získat 1 000 etherů. Řekněme, že jeden uživatel našel řešení `Ethereum!`. Zavolá `solve` s `Ethereum!` jako parametrem. Útočník byl bohužel chytrý a sleduje nashromážděné transakce a hledá v nich transakci odesílající řešení. podívá se na toto řešení, zkontrolujte jeho platnost a poté odešlete podobnou transakci transakce s mnohem vyšší cenou „gasPrice“ než původní transakce. Těžař, který blok řeší, pravděpodobně dá přednost transakci útočníka díky vyšší `gasPrice`, a vytěží jeho transakci před transakcí původního řešitele. Útočník získá 1 000 etherů a uživatel, který problém vyřešil, nedostane nic. Mějte na paměti, že v případě tohoto typu „předbíhajících útoků“ zranitelnosti jsou těžaři jedinečně motivováni k provádění útoků jimi samými (nebo mohou být upláceni za účelem provedení těchto útoků za extravagantní poplatky). Nelze podceňovat možnost, že by útočník byl sám těžař.

Peventivní techniky

Existují dvě třídy účastníků, kteří mohou provádět tyto druhy předbíhacích útoků: uživatelé (kteří změnili `gasPrice` jejich transakce) a těžaři samotní (kteří mohou přeuspořádat transakce v bloku,

jak uznají za vhodné). Kontrakt, který je zranitelný vůči první třídě (uživatelé) je výrazně horší, než je zranitelný vůči druhé (těžaři), protože těžaři mohou provést útok pouze tehdy, když vyřeší a blok, což je nepravděpodobné pro jakéhokoli jednotlivého těžaře, který cílí na konkrétní blok. Zde uvádíme několik opatření ke zmírnění vztahujících se k oběma třídám útočníků.

Jednou z metod je umístit horní hranici `gasPrice`. To zabraňuje uživatelům zvyšovat `gasPrice` za horní hranic a získávat přednost při řazení transakcí Toto opatření pouze chrání před první třídou útočníků (libovolní uživatelé). Těžaři v tomto scénáři mohou stále útočit na kontrakt, protože mohou uspořádávat transakce v jejich bloku bez ohledu, bez ohledu na cenu plynu.

Odolnější metoda je použití odeslat–odhalit [http://bit.ly/2CUh2KS] schéma. Takové schéma určuje, že uživatelé posílají transakce se skrytými informacemi (obvykle haš). Poté, co byla transakce byla zahrnuta do bloku, uživatel odešle transakci odhalení dat, která byla odeslána (fáze odhalení). Tato metoda zabraňuje jak těžařům, tak uživatelům, aby prováděli přebíhající transakce, protože nemůžou určit obsah transakce. Tato metoda však nemůže skrýt hodnotu transakce (což v některých případech je cenná informace, kterou je třeba skrýt). ENS [https://ens.domains/] chytrý kontrakt umožňuje uživatelům zaslat transakce, jejichž potvrzená data zahrnovala množství etheru, který oni byli ochotni utratit. Uživatelé pak mohli odesílat libovolné transakce hodnoty. Během fáze odhalení byl uživatelům vrácen rozdíl mezi částkou odeslanou v transakci a částkou, která byli ochotni utratit.

Další návrh od autorůLorenz Breidenbach, Phil Daian, Ari Juels, a Florian Tramèr je použití "ponorka zaslání" [http://bit.ly/2SygqQx]. Účinné provedení této myšlenky vyžaduje instrukci `CREATE2` která v současné době nebyla přijata, ale zdá se, že pravděpodobně bude v nadcházejícím tvrdém rozštěpení.

Příklady ze života: ERC20 a Bancor

ERC20 standard [http://bit.ly/2CUf7WG] je docela dobře známý pro vytváření tokenů on Ethereu. Tento Standard má potenciální chybu přebíhání, která se vyskytuje v ve funkci `approve`. Mikhail Vladimirov and Dmitry Khovratovich [http://bit.ly/2DbvQpJ] napsali toto dobré vysvětlení zranitelnost (a způsoby, jak zmírnit útok).

Norma specifikuje funkci `approve` jako:

```
function approve(address _spender, uint256 _value) returns (bool success)
```

Tato funkce umožňuje uživateli povolit jiným uživatelům převod tokenů jejich jménem. K zranitelnosti chybou předbíhání dochází ve scénáři, kde uživatel Alice *schválí* jejímu příteli Bobovi, aby utratil 100 tokenů. Alice se později rozhodne, že chce odvolat Bobův souhlas utratit, řekněme: 100 tokenů, takže vytvoří transakci, která určuje Bobův příděl 50 tokenů. Bob, který bločenku pečlivě sleduje, vidí tuto transakci a vytvoří transakci jeho vlastního výdaje 100 tokenů. Na svou transakci vloží vyšší `gasPrice` než Alice, takže jeho transakce dostane přednost před její. Některé implementace `approve` by Bobovi umožnily převést jeho 100 tokenů a poté, když je transakce Alice schválena, nastaví se Bobův příděl na 50 tokenů, ve skutečnosti umožňuje Bobovi přístup k 150 tokenům.

Dalším příkladem ze života je Bancor [https://www.bancor.network/]. Ivan Bogatyy a jeho tým zdokumentoval ziskový útok na počáteční implementaci Bancor. Jeho blog příspěvek [http://bit.ly/2EUlLzb] a DevCon3 řeč [http://bit.ly/2yHgkhs] podrobně diskutují o tom, jak se to stalo. Ceny tokenů jsou v zásadě ceny určené na základě hodnoty transakce; uživatelé mohou sledovat transakční úložiště pro Bancor transakce a útokem přebíháním vydělávat na rozdílech cen. Tento útok byl vyřešen týmem Bancor.

Odepření služby (DoS)

Tato kategorie je velmi široká, ale v zásadě sestává z útoků, kde uživatelé mohou učinit smlouvu nefunkční po určitou dobu, nebo v některých případech trvale. To může v těchto smlouvách uvěznit ether navždy, jak tomu bylo v případě Příklad ze života: Parity vícepodpisová peněženka (druhé napadení).

Zranitelnost

Existují různé způsoby, jak se kontrakt může stát nefunkční. Tady jsme zdůraznit jen několik méně zřejmých Solidity vzorů kódu, které mohou vést ke zranitelnosti DoS:

Průchod extrémně zmanipulovaným mapováním nebo polem

> Tento vzor se obvykle objeví, když si majitel přeje distribuovat tokeny investorům pomocí funkce `distribute` jako v tomto příkladu kontraktu:

```
contract DistributeTokens {
    address public owner; // dostane se n¬kam
    address[] investors; // pole investor¬
    uint[] investorTokens; // množství token¬, které každý investor
dostane

    // ... další funkce, v¬etn¬ transfertoken()

    function invest() external payable {
        investors.push(msg.sender);
        investorTokens.push(msg.value * 5); // pošle 5-krát wei
        }

    function distribute() public {
        require(msg.sender == owner); // pouze majitel
        for(uint i = 0; i < investors.length; i++) {
            // zde transferToken(to,amount) p¬evede "amount"
            // token¬ adrese "to"
            transferToken(investors[i],investorTokens[i]);
        }
    }
}
```

Všimněte si, že smyčka v tomto kontraktu běží přes pole, které může být uměle nahuštěné. Útočník může vytvořit mnoho uživatelských účtů a udělat pole investorů příliš velké. V zásadě to lze provést tak, že plyn potřebný k provedení smyčky for přesáhne limit plynu v bloku, v podstatě způsobuje nefunkčnost funkce `distribute`.

Operace vlastníka

Dalším obvyklým vzorem je, kde majitelé mají specifická privilegia v kontraktech a musí provést nějakou úlohu, aby kontrakt postoupit do dalšího stavu. Jedním příkladem by mohla být kontrakt počáteční nabídka mincí (ICO), který vyžaduje, aby vlastník `finalize` kontrakt, který poté umožňuje převody tokenů. Například:

```
bool public isFinalized = false;
address public owner; // dostane se n¬kam

function finalize() public {
    require(msg.sender == owner);
    isFinalized == true;
}

// ... extra ICO funkcionalita

// p¬etížená p¬evodová funkce
function transfer(address _to, uint _value) returns (bool) {
    require(isFinalized);
    super.transfer(_to,_value)
}

...
```

V takových případech, pokud privilegovaný uživatel ztratí soukromé klíče nebo se stane neaktivní, celý tokenový kontrak se stane nefunkční. V tomto případě, pokud vlastník nemůže zavolat `finalize`, žádné tokeny nemohou být převedeny; celá obsluha tokenového ekosystému závisí na jedné address.

Progresivní stav založený na externích voláních

Kontrakty jsou někdy psány tak, že postup do nového stavu vyžaduje odeslání etheru na adresu nebo čekání na nějaký vstup z externího zdroje. Tyto vzory mohou vést k DoS útokům v případě, že externí volání selže nebo je mu zabráněno externím důvodem. V příkladu odesílání etheru může uživatel vytvořit kontrakt, který nepřijímá ether. Pokud kontrakt vyžaduje, aby byl ether vybrán, aby postupoval do nového státu (zvažte a kontrakt časového uzamčení, která vyžaduje, aby byl veškerý ether vybrán před tím, než bude znovu použitelný), kontrakt nikdy nedosáhne nového stavu, protože éter nemůže být nikdy zaslán na uživatelský kontrakt, který nepřijímá ether.

Peventivní techniky

V prvním příkladu by kontrakty neměly procházet smyčkou datovými strukturami, které mohou být manipulovány externími uživateli. Výběrový vzor je doporučen, přičemž každý z investorů volá

funkci withdraw pro vyzvednutí tokenů nezávisle.

Ve druhém příkladu byl ke změně stavu vyžadován privilegovaný uživatel kontraktu. V takových příkladech může být použita pojistka pro případ, že se majitel stane nezpůsobilým. Jedno řešení je, aby se majitel stal vícepodpisovým kontraktem. Další řešení je použití časového zámku: v daném příkladu require na řádku 5 může obsahovat a mechanismus založený na čase, například `require(msg.sender == owner || now > unlockTime)`, který umožňuje libovolnému uživateli dokončení po uplynutí doby zadané proměnnou `unlockTime`. Tento druh zmírňovací techniky lze použít také ve třetím příkladu. Pokud externí volání jsou nutná pro postup do nového stavu, účet pro jejich možné selhání a případně přidat stav založený na uplynulém čase pro postup v případě, že požadované volání nikdy nepřijde.

K těmto návrhům samozřejmě existují centralizované alternativy: Lze přidat `maintenanceUser` kdo může přijít a opravit v případě potřeby problémy s útoky na bázi DoS. Typicky tyto druhy kontraktů mají problémy s důvěrou, kvůli moci takové entity.

Příklady ze života: GovernMental

GovernMental [http://governmental.github.io/GovernMental/] bylo staré Ponziho schéma, které nashromáždilo poměrně velké množství etheru (1 100 etherů v jeden okamžik). Bohužel to bylo citlivé na zranitelnosti DoS zmíněné v této sekci. A Reddit příspěvek [http://bit.ly/2DcgvFc] od etherik popisuje, jak kontrakt vyžadovala výmaz velkého mapování za účelem odebrání etheru. Vypuštění tohoto mapování mělo náklady na plyn, která v té době překračovaly hranici limitu plynu v bloku, a tak bylo nemožné vybrat 1 100 etherů. Adresa kontraktu je 0xF45717552f12Ef7cb65e95476F217Ea008167Ae3 [http://bit.ly/2Oh8j7R], a můžete vidět z transakce 0x0d80d67202bd9cb6773df8dd2020e719 0a1b0793e8ec4fc105257e8128f0506b [http://bit.ly/2Ogzrnn], že 1 100 etheru bylo nakonec získáno pomocí transakce, která použita 2.5M plynu (když se limit plynu v bloku dostatečně zvýšil, aby umožnil takovou transakci).

Manipulace s časovou značkou bloku

Časové značky bloků byla historicky používány pro rozmanité aplikace, jako je entropie pro náhodná čísla (viz Iluze entropie pro další podrobnosti), zamykání finančních prostředků na časové období a různé podmíněné změny stavu, příkazy, které jsou časově závislé. Těžaři mají schopnost mírně přizpůsobit časové značky, což se ukázalo být nebezpečným, pokud časové značky jsou v

chytrých kontraktech používány nesprávně.

Užitečné odkazy k tomu zahrnují Solidity dokumentaci [http://bit.ly/2OdUC9C] a Joris Bontje's Ethereum Stack Exchange otázky [http://bit.ly/2CQ8gh4] na toto téma.

Zranitelnost

`block.timestamp` a její alias `now` mohou být manipulovány těžaři, pokud mají k tomu určitou motivaci. Pojďme postavit jednoduchou hru, ukázanou v roulette.sol, která by byla zranitelná vůči zneužití těžaři.

Example 17. roulette.sol

```
contract Roulette {
    uint public pastBlockTime; // vynutí jednu sázku na blok

    constructor() external payable {} // po¬áte¬ní financování
kontraktu

    // nouzová funkce použita pro položení sázky
    function () external payable {
        require(msg.value == 10 ether); // musí zaslat 10 etheru pro
hraní
        require(now != pastBlockTime); // pouze jedna transakce v bloku
        pastBlockTime = now;
        if(now % 15 == 0) { // vít¬z
            msg.sender.transfer(this.balance);
        }
    }
}
```

Tento kontrakt se chová jako jednoduchá loterie. Jedna transakce v bloku může vsadit 10 etherů na šanci získat zůstatek kontraktu. předpokládá se, že poslední dvě číslice `block.timestamp` jsou rovnoměrně rozděleny. Pokud by tomu tak bylo, byla by 1 ku 15 šance vyhrát tuto loterii.

Jak však víme, těžaři mohou upravit časovou značku, pokud to budou potřebovat. V tomto konkrétním případě, pokud je ve smlouvě dostatek etherových zdrojů, těžař, který řeší blok, je

motivován, aby si vybral takovou časovou značku `block.timestamp` nebo `now`, která je modulo 15 rovna `0`. Může tak vyhrát ether zamčený v tomto kontraktu spolu s blokovou odměnou. Protože existuje pouze jedna osoba, která může sázet v bloku, je to také zranitelné útokem předbíháním (viz Souběh / předbíhání pro další detaily).

V praxi se časová značky bloku monotónně zvyšují, a tak těžaři nemohou zvolit libovolná časové značky bloku (musí být pozdější než jejich předchůdci). Jsou také omezeny na nastavení časů bloků, které nejsou příliš daleko v budoucnu, protože tyto bloky bude síť pravděpodobně odmítat (uzly nebudou ověřovat bloky, jejichž časové značky jsou v budoucnosti).

Peventivní techniky

Časové značky bloku by neměla být použita pro entropii nebo generování náhodných čísel tj. neměly by být rozhodujícím faktorem (buď přímo nebo prostřednictvím nějakého odvození) za výhru hry nebo změnu důležitého stavu.

Někdy je vyžadována časově citlivá logika; např. pro odblokování kontraktů (time-locking), dokončení ICO po několika týdnech nebo vynucení vypršení platnosti dat. Někdy se doporučuje použít `block.number` [http://bit.ly/2OdUC9C] a průměrný čas bloku pro odhad doby; s a `10 sekund` jako dobu tvorby bloku, `1 týden` odpovídá přibližně `60480 blok¬m`. Určení čísla bloku, ve kterém lze změnit stav kontraktu, může být bezpečnější, protože těžaři nemohou snadno manipulovat s číslem bloku. BAT ICO [http://bit.ly/2AAebFr] kontrakt využil tuto strategii.

To může být zbytečné, pokud se kontrakty zvlášť netýkají těžařské manipulace s časovou značkou bloku, ale je to něco, o čem byste měli vědět při nasazení kontraktů.

Příklad ze života: GovernMental

GovernMental [http://governmental.github.io/GovernMental/], staré Ponzi schéma zmíněné výše, bylo také zranitelné útokem založeným na časových značkách. Kontrakt vyplácel hráče, který byl posledním hráčem, který se připojil (alespoň na jednu minutu) v daném kole. Těžař, který byl hráčem, tak mohl upravit časovou značku (na budoucí čas, aby to vypadalo, že uběhla minuta) aby se ukázalo, že byl posledním hráčem, který se připojil na déle než minutu (dokonce i když to ve skutečnosti neplatilo). Více podrobností najdete v "Historii Ethereum bezpečnosti a zranitelností, útoků a oprav [http://bit.ly/2Q1AMA6] od Tanya Bahrynovska.

Konstruktory s péčí

Konstruktory jsou speciální funkce, které často vykonávají kritické, privilegované úkoly při inicializaci kontraktů. Před Solidity v0.4.22, konstruktory byli definovány jako funkce, které měly stejné jméno jako kontrakt, který je obsahoval. V takových případech, když se změní název kontraktu v vývoji, pokud se nezmění také jméno konstruktoru, stane se normální, volitelnou funkcí. Jak si dokážete představit, může to vést k některým zajímavým napadeném kontraktů.

Pro další nahlédnutí může mít čtenář zájem, pokusit se o Ethernaut výzvy [https://github.com/OpenZeppelin/ethernaut] (příslušné úrovně Fallout).

Zranitelnost

IPokud je název kontraktu změněn nebo je v něm překlep jméno konstruktoru pak neodpovídá názvu kontraktu, konstruktor se bude chovat jako normální funkce. To může vést k hrozným důsledkům, zejména pokud konstruktor provádí privilegované operace. Zvažte následující kontrakt:

```
contract OwnerWallet {
    address public owner;

    // konstruktor
    function ownerWallet(address _owner) public {
        owner = _owner;
    }

    // Nouzová funkce, p¬ijímá ether
    function () payable {}

    function withdraw() public {
        require(msg.sender == owner);
        msg.sender.transfer(this.balance);
    }
}
```

Tento kontrakt shromažďuje ether a umožňuje jej vybrat pouze vlastníkovi, zavoláním funkce `withdraw`. Problém nastává, pokud konstruktor není pojmenován přesně stejně jako kontrakt: první písmeno je jiné! Tak, jakýkoliv uživatel smí zavolat funkci `ownerWallet`, nastavit sám sebe jako

vlastníka, a poté vzít veškerý ether v kontraktu zavoláním `withdraw`.

Peventivní techniky

Tento problém byl vyřešen ve verzi 0.4.22 Solidity překladače. Tato verze představila klíčové slovo `constructor`, které specifikuje konstruktor, spíše než vyžadování jména funkce odpovídajícího jménu kontraktu. Doporučuje se určování konstruktorů pomocí tohoto klíčového slova, aby se zabránilo problémům s pojmenováním.

Příklad ze života: Rubixi

Rubixi [http://bit.ly/2ESWG7t] bylo další pyramidové schéma, které se vystavovalo tomuto druhu zranitelnosti. Bylo původně nazváno `DynamicPyramid`, ale název kontraktu byl změněn před nasazením na `Rubixi`. jméno konstruktoru se nezměnilo, což každému uživateli umožnilo stát se tvůrcem. Najdete zde zajímavou diskusi týkající se této chyby na Bitcointalk [http://bit.ly/2P0TRWw]. Nakonec to uživatelům umožnilo bojovat o status tvůrce a vybírat poplatky z pyramidového schématu. Více podrobností o této konkrétní chybě lze nalézt v "Historii Ethereum bezpečnosti, zranitelností. útoků a jejich oprav" [http://bit.ly/2Q1AMA6].

Neinicializované ukazatele úložiště

("uninitialized storage pointers security threat", id="ix_09smart-contracts-security-asciidoc45", range="startofrange"EVM ukládá data buď do úložiště nebo do paměti. Porozumění tomu, jak přesně se to dělá a výchozím typům lokálních proměnných funkcí je vysoce doporučeno při vývoji kontraktů. To je protože je možné vytvořit zranitelné kontrakty nevhodnou inicializací proměnných.

Další informace o úložišti a paměti v EVM najdete v dokumentaci Solidity umístění dat [http://bit.ly/2OdUU0l], rozložení stavových proměnných v úložišti [http://bit.ly/2JslDWf], a rozložení paměti [http://bit.ly/2Dch2Hc].

Tato sekce je založena na vynikajícím textu od Stefana Beyera [http://bit.ly/2ERI0pb]. Další informace k tomuto tématu, inspirované Stefanem, najdete v tomto Reddit vlákně [http://bit.ly/2OgxPtG].

Zranitelnost

Lokální proměnné ve výchozím nastavení funkcí úložiště nebo paměti závisí na jejich typu. Neinicializované proměnné místního úložiště mohou obsahovat hodnoty jiných proměnných úložiště v kontraktu; tato skutečnost může způsobit neúmyslné zranitelnosti nebo může být záměrně zneužita.

Podívejme se relativně jednoduchý registrační kontrakt v NameRegistrar.sol.

Example 18. NameRegistrar.sol

```
// Zam¬ený registr jmen
contract NameRegistrar {

    bool public unlocked = false;  // registr zam¬ený, nelze m¬nit
jména

    struct NameRecord { // mapa haš¬ na addresy
        bytes32 name;
        address mappedAddress;
    }

    // záznamy, kdo zaregistroval jména
    mapping(address => NameRecord) public registeredNameRecord;
    // p¬ekládá haše na adresy
    mapping(bytes32 => address) public resolve;

    function register(bytes32 _name, address _mappedAddress) public {
        // nastaví nový NameRecord
        NameRecord newRecord;
        newRecord.name = _name;
        newRecord.mappedAddress = _mappedAddress;

        resolve[_name] = _mappedAddress;
        registeredNameRecord[msg.sender] = newRecord;

        require(unlocked); // Povolit registrace pouze v p¬ípad¬, že je
kontrakt  odem¬en
    }
}
```

Tento registrátor jednoduchých jmen má pouze jednu funkci. Když je kontrakt odemčený `unlocked`, umožňuje komukoli zaregistrovat jméno (jako `bytes32` haš) a namapujte toto jméno na adresu. Registrátor je nejprve zamčený, a `require` na řádku 25 zabraňuje `register` přidávat nové záznamy. Zdá se, že kontrakt je nepoužitelný Neexistuje žádný způsob, jak odemknout registr! Existuje však zranitelnost která umožňuje registraci jména bez ohledu na proměnnou `unlocked`.

K projednání této chyby zabezpečení musíme nejprve porozumět tomu, jak úložiště pracuje v

Solidity. Jako přehled na vysoké úrovni (bez jakýchkoli technických podrobností doporučujeme přečíst si Solidity dokumentaci pro pro řádnou kontrolu, stavové proměnné jsou ukládány postupně v *slotech* tak, jak jsou objeví se v kontraktu (mohou být seskupeny dohromady, ale nejsou v tomto příkladu, takže se tím nebudeme bát). Tedy, `unlocked` existuje v `slot[0]`, `registeredNameRecord` v `slot[1]`, a `resolve` v `slot[2]`, atd. Každý z těchto slotů má velikost 32 bajtů (jsou přidány složitosti s mapováním, které prozatím nebudeme ignorovat). Booleovský `unlocked` bude vypadat jako `0x000...0` (64 0, kromě `0x`) pro `false` nebo `0x000...1` (63 0) pro `true`. Jak vidíte, existuje v tomto konkrétním příkladu se značně plýtvá úložištěm.

Další kousek skládačky je, že Solidity ve výchozím nastavení při inicializaci ukládá do úložiště složité datové typy, jako jsou struct jako lokální proměnné. Takže, `newRecord` na řádce 18 jde standardně do úložiště. Tato zranitelnost je způsobena tím, že `newRecord` je neinicializován. Protože je výchozí úložiště, je to mapováno na slot úložiště slot[0], který v současné době obsahuje ukazatel na `unlocked`. Všimněte, že na řádcích 19 a 20 jsme nastavili `newRecord.name` na `_name` a `newRecord.mappedAddress` na `_mappedAddress`; Tím se aktualizují umístění úložiště slot[0] a slot[1], což modifikuje obojí `unlocked`a slot úložišt¬ spojený s `registeredNameRecord`.

To znamená, že `unlocked` lze přímo upravit jednoduše pomocí `bytes32 _name` parametru funkce `register`. Proto pokud poslední bajt `_name` je nenulový, změní poslední bajt storage `slot[0]` a přímo změní `unlocked` na `true`. Taková hodnota `_name` způsobí, že volání `require` na řádku 25 je úspěšné, protože jsme nastavili `unlocked` na `true`. Vyzkoušejte to v Remix. Všimněte si, že funkce projde pokud použijete `_name` ve tvaru:

```
0x0000000000000000000000000000000000000000000000000000000000000001
```

Peventivní techniky

Kompilátor Solidity zobrazuje varování pro neinicializované proměnné úložiště; vývojáři by měli těmto upozorněním věnovat náležitou pozornost, při tvorbě chytrých kontraktů. Aktuální verze Mist (0.10) neumožňují kompilovat tyto kontrakty. To je často dobrá praxe při řešení složitých typů explicitně používat specifikátory memory` nebo `storage`, pro zajištění, aby se chovaly podle očekávání.

Příklad ze života: OpenAddressLottery a CryptoRoulette hrnce medu

Hrnec medu pojmenovaný OpenAddressLottery [http://bit.ly/2AAVnWD] byl nasazen, tak že používal neinicializovanou proměnnou úložiště, aby vybral ether od některých nezkušených útočníků. Kontrakt je spíše zapojen, tak necháme analýzu na Reddit vlákně [http://bit.ly/2OgxPtG] kde je útok zcela jasně vysvětlen.

Další hrnec medu, CryptoRoulette [http://bit.ly/2OfNGJ2], také využívá tento trik pro pokus získat od vás nějaký ether. Pokud nemůže zjistit, jak útok funguje, viz " Analýza několika Ethereum hrnců medu" [http://bit.ly/2OVkSL4] pro přehled tohoto kontraktu a dalších.

Plovoucí desetinná čárka a přesnost

V době psaní této knihy (v0.4.24), Solidity nepodporuje reálná čísla s pevnou a plovoucí desetinnou čárku To znamená, že plovoucí desetinná čárka musí být reprezentována pomocí celočíselných typů v Solidity. To může vést k chybám a zranitelnosti, pokud to není implementováno správně.

Další čtení viz Ethereum kontrakty, bezpečnostní techniky a tipy wiki [http://bit.ly/2Ogp2Ia].

Zranitelnost

Protože není žádný datový typ s pevnou desetinou čárkou v Solidity, vývojáři jsou nuceni implementovat vlastní pomocí standardních celočíselných datových typů. Existuje množství úskalí pro vývojáře, které mohou nastat během tohoto procesu. Pokusíme se některé z nich zdůraznit v této sekci.

Začněme příkladem kódu (pro zjednodušení budeme ignorovat problémy s přetečením / podtečením, o kterých jsme hovořili dříve v této kapitole):

```
contract FunWithNumbers {
    uint constant public tokensPerEth = 10;
    uint constant public weiPerEth = 1e18;
    mapping(address => uint) public balances;

    function buyTokens() external payable {
        // p¬evod wei na eth, poté vynásobí kurzem tokenu
        uint tokens = msg.value/weiPerEth*tokensPerEth;
        balances[msg.sender] += tokens;
    }

    function sellTokens(uint tokens) public {
        require(balances[msg.sender] >= tokens);
        uint eth = tokens/tokensPerEth;
        balances[msg.sender] -= tokens;
        msg.sender.transfer(eth*weiPerEth);
    }
}
```

Tento jednoduchý kontrakt nakupující / prodávající tokeny má některé zřejmé problémy. Přestože matematické výpočty pro nákup a prodej tokenů jsou správné, nedostatek plovoucí čísel s plovoucí desetinnou čárkou poskytne chybné výsledky. Například při nákupu tokenů na řádce 8, pokud je hodnota menší než `1 ether` počáteční dělení bude mít výsledek `0`, výsledek konečného násobení zůstane jako `0` (např., `200 wei` vyděleno `1e18` `weiPerEth` se rovná `0`). Podobně při prodeji tokenů, libovolný počet tokenů menší než `10` bude mít také za následek `` `0 ether ``. Ve skutečnosti je zaokrouhlování vždy dolů, takže výsledkem bude prodej `29 tokens` za `2 ether`.

Problém s tímto kontraktem spočívá v tom, že přesnost se týká pouze nejbližšího etheru (tj. 1e18 wei). To může být obtížné, když se zabýváte desetinnými čísly v ERC20 [https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md] tokenech, když potřebujete vyšší přesnost.

Peventivní techniky

Udržování vhodné přesnosti ve vašem chytrém kontraktu je velmi důležité, zejména při řešení poměrů a sazeb odrážejících ekonomická rozhodnutí.

Měli byste zajistit, aby všechny použité poměry nebo sazby umožňovaly velká čísla ve zlomcích. Například jsme použili kurz `tokensPerEth` v našem příkladě. Bylo by lepší použít `weiPerTokens`,

což bude velké číslo. K výpočtu odpovídajícího počtu tokenů bychom mohli udělat `msg.value/weiPerTokens`. To by poskytlo přesnější výsledek.

Další taktika, kterou je třeba mít na paměti, je mít na paměti pořadí operací. V našem příkladu byl výpočet nákupu tokenů proveden `msg.value/weiPerEth*tokenPerEth`. Všimněte si, že k dělení dojde před násobením. (Solidity, na rozdíl od některých jazyků, zaručuje provádění operací v pořadí, v jakém jsou psány.) Tento příklad by dosáhl větší přesnost, pokud výpočet provedl nejprve násobení a poté dělení, tj. `msg.value*tokenPerEth/weiPerEth`.

A konečně, při definování libovolné přesnosti pro čísla to může být dobrý nápad převést hodnoty na vyšší přesnost, provést všechny matematické operace, pak nakonec převést zpět na přesnost požadovanou pro výstup. Obvykle uint256 je použit (z důvodu optimálního využití plynu); tyto dávají přibližně 60 řádů v jejich rozsahu, z nichž některé lze věnovat přesnosti matematických operací. Může se stát, že je lepší si to nechat všechny proměnné s vysokou přesností v Solidity a převést je zpět na nižší přesnosti v externích aplikacích (to je v podstatě jak decimals` proměnné fungují v ERC20 tokenových kontraktech). Chcete-li vidět příklad, jak toho lze dosáhnout, doporučujeme podívat se na `https://github.com/dapphub/ds-math[DS-Math]. Používá některé zvláštní názvy (“wads” a “rays”), ale koncept je užitečný.

Příklad ze života: Ethstick

The Ethstick kontrakt [http://bit.ly/2Qb7PSB] nepoužívá rozšířenou přesnost; jedná se však o wei. Tak, tento kontrakt bude mít problémy se zaokrouhlováním, ale pouze na úrovni wei přesnosti. Má některé závažnější nedostatky, ale ty se týkají obtížnosti získat entropii na bločence (viz Iluze entropie). Pro další diskusi o Ethstick kontraktu, odkážeme vás na další příspěvek od Petera Vessenesa, "Ethereum Kontraky jsou cukrovím pro útočníky" [http://bit.ly/2SwDnE0].

Tx.Origin ověřování

Solidity má globální proměnnou `tx.origin`, která prochází celým zásobníkem volání a obsahuje adresu účtu, který původně odeslal volání (nebo transakci). Použití této proměnné pro ověřování v chytrém kontraktu způsobí, že kontrakt bude zranitelný vůči útoku podvržení identity.

Další čtení viz dbryson Ethereum Stack Exchange otázky [http://bit.ly/2PxU1UM], “Tx.Origin and Ethereum Oh My!” [http://bit.ly/2qm7ocJ] od Petera Vessenese, a “Solidity: Tx Origin Attacks” [http://bit.ly/2P3KVA4] od Chrise Coverdala.

Zranitelnost

Kontrakty, které uživatele opravňují pomocí proměnné `tx.origin` jsou obvykle jsou náchylné k útokům podvržení identity, při kterých mohou být uživatelé oklamáni, aby provedli ověřovací akci na zranitelném kontraktu.

Zvažme jednoduchý kontrakt v Phishable.sol.

Example 19. Phishable.sol

```
contract Phishable {
    address public owner;

    constructor (address _owner) {
        owner = _owner;
    }

    function () external payable {} // p¬ijímá ether

    function withdrawAll(address _recipient) public {
        require(tx.origin == owner);
        _recipient.transfer(this.balance);
    }
}
```

Všimněte si, že na řádku 11 kontrakt povoluje funkci `withdrawAll` pomocí `tx.origin`. Tento kontrakt umožňuje útočníkovi vytvořit útočný kontrakt ve tvaru:

```
import "Phishable.sol";

contract AttackContract {

    Phishable phishableContract;
    address attacker; // Adresa úto¬níka pro získání finan¬ních prost¬edk¬

    constructor (Phishable _phishableContract, address _attackerAddress) {
        phishableContract = _phishableContract;
        attacker = _attackerAddress;
    }

    function () payable {
        phishableContract.withdrawAll(attacker);
    }
}
```

Útočník by mohl maskovat tento kontrakt jako svojí vlastní soukromou adresu a sociálním působením donutit oběť (vlastníka Phishable kontraktu), aby na adresu poslal nějakou formu transakce - možná zaslal tomuto kontraktu nějaké množství etheru. Oběť, pokud není opatrná, si toho nemusí všimnout zde je kód na adrese útočníka, nebo by jej mohl útočník předat jako peněženku s více podpisy nebo jako peněženku pro pokročilé úložiště (pamatujte že zdrojový kód veřejných kontraktů není ve výchozím nastavení k dispozici).

V každém případě, pokud oběť odešle transakci s dostatečným množstvím plynu na adresu `AttackContract`, vyvolá nouzovou funkci, ve které se zavolá funkce `withdrawAll` kontraktu `Phishable` s parametrem`attacker`. To povede ke vybrání všech prostředků z kontraktu `Phishable` na adresu `attacker` . Tohle je protože adresa, která poprvé iniciovala volání, byla obětí (tj. vlastník kontraktu `Phishable`). Proto `tx.origin` se bude rovnat `owner`a `require` na řádku 11 `Phishable` kontraktu projde.

Peventivní techniky

`tx.origin` by nemělo být použito pro autorizaci v chytrých kontraktech. To neznamená, že proměnná `tx.origin` by nikdy neměla být použita. To má nějaké legitimní případy použití v chytrých kontraktech. Například, pokud by někdo chtěl odmítnout externí kontrakty volané z aktuálního kontraktu, mohl by implementovat `require` ve tvaru `require(tx.origin ==`

`msg.sender`). To brání zprostředkujícím kontraktům, aby byly používány pro volání aktuálního kontraktu, čímž se kontrakt omezuje běžné adresy, které nejsou kontraktem.range="endofrange", startref="ix_09smart-contracts-security-asciidoc49").

Knihovny kontraktů

K dispozici je spousta existujících kódů pro opakované použití, a to jak nasazených v bločence jako dostupné knihovny, tak mimo bločenku jako knihovny šablon kódů. Platformové knihovny, které byly nasazeny, existují jako chytré kontrakty v bajtkódu, takže před jejich reálným použitím je třeba věnovat velkou pozornost. Používání dobře zavedených existujících knihoven na platformě však přináší mnoho výhod, například možnost těžit z nejnovějších rozšíření a šetří vám peníze a prospívá ekosystému Ethereum snížením celkového počtu živých kontraktů v Ethereum síti.

V Ethereum je nejčastěji používaným zdrojem OpenZeppelin [https://openzeppelin.org/], rozsáhlá knihovna kontraktů od implementace tokenů ERC20 a ERC721, po mnoho příchutí modelů veřejného prodeje, až po jednoduchá chování běžně vyskytující se v kontraktech, jako jsou `Ownable`, `Pausable`, nebo `LimitBalance`. Kontrakty v tomto úložišti byly důkladně testovány a v některých případech dokonce fungují jako standardní implementace. Jsou zdarma k použití a jsou vytvářeny a udržovány pomocí https://zeppelin.solutions [Zeppelin] spolu se stále rostoucím seznamem externích přispěvatelů.

Od Zeppelin je také ZeppelinOS [https://zeppelinos.org/], platforma služeb a nástrojů s otevřeným zdrojovým kódem, pro bezpečný vývoj a správu aplikací chytrých kontraktů. ZeppelinOS poskytuje vrstvu na vrcholu EVM, která vývojářům usnadňuje spuštění rozšiřitelných DApps spojených s bločenkovou knihovnou prověřených kontraktů, které jsou samy rozšiřitelné. Na platformě Ethereum mohou existovat různé verze těchto knihoven a vouching systém umožňuje uživatelům navrhovat nebo prosazovat vylepšení v různých směrech. Platforma rovněž poskytuje sadu nástrojů mimo bločenku pro ladění, testování, nasazení a sledování decentralizovaných aplikací.

Cílem projektu ethpm je uspořádat různé zdroje, které se v ekosystému vyvíjejí, poskytováním systému správy balíků. Jejich registr jako takový poskytuje další příklady pro procházení:

- Web: https://www.ethpm.com/
- Úložiště: https://www.ethpm.com/registry
- GitHub odkaz: https://github.com/ethpm

- Dokumentace: https://www.ethpm.com/docs/integration-guide

Závěry

Každý vývojář, který pracuje v oblasti chytrých kontraktů, má toho hodně co vědět a pochopit. Dodržováním osvědčených postupů při navrhování chytrých kontraktů a psaní kódu se vyhnete mnoha vážným nástrahám a pastím.

Snad nejzákladnějším principem zabezpečení softwaru je maximalizace opětovného použití důvěryhodného kódu. V kryptografii je to tak důležité, že bylo zhuštěno do přísloví: „Nevyhazujte své krypto.“ V případě chytrých kontraktů se jedná o co největší zisk z volně dostupných knihoven, které byly komunitou důkladně prověřeny.

Tokeny

Slovo „token“ pochází ze staroanglického slova „tācen“, což znamená znaménko nebo symbol. Obyčejně je používáno při odkazování na soukromě vydávané, mincím podobné věci se speciálním účelem, nízké vnitřní hodnoty. Příklady mohou být přepravní žetony, žetony do prádelen a žetony do arkádové hry. Poznámka překladate: Pro token se používá český překlad žeton, pouze pokud má hmotnou podobu. Nehmotné tokeny se používaly v informatice již dávno před vznikem ktyptoměn a ponechávaly se bez překladu.

V současné době „tokeny“ spravované na bločenkách předefinují slovo tak, že znamenají abstrakce založené na bločence, které mohou být ve vlastněné a které představují aktiva, měnu nebo přístupová práva.

Asociace mezi slovem „token“ a nevýznamnou hodnotou má hodně co do činění s omezeným použitím fyzických verzí tokenů (nazývaných žetony). Fyzické tokeny, které jsou často omezeny na konkrétní firmy, organizace nebo umístění, nejsou snadno vyměnitelné a obvykle mají pouze jednu funkci. U bločenkových tokenů jsou tato omezení zrušena - nebo přesněji zcela předefinovatelná. Mnoho bločenkových tokenů slouží globálně pro různé účely a lze je na obchodovat na světových likvidních trzích za jiné tokeny nebo měny. Omezení na vlastnictví a používání zmizelo, stejně tak očekávání „zanedbatelné hodnoty“ je také minulostí.

V této kapitole se podíváme na různá použití tokenů a na to, jak jsou vytvořeny. Diskutujeme také o vlastnostech tokenů, jako je zastupitelnost a přirozenost. Nakonec zkoumáme standardy a technologie, na nichž jsou založeny, a experimentujeme vytvořením vlastních tokenů.

Jak se používají tokeny

Nejviditelnějším použitím tokenů jsou digitální soukromé měny. Toto je však pouze jedno možné použití. Tokeny lze naprogramovat tak, aby sloužily mnoha různým funkcím, které se často překrývají. Například token může současně zprostředkovat hlasovací právo, přístupové právo a vlastnictví zdroje. Jak ukazuje následující seznam, měna je pouze první „aplikace“:

Měna

: Token může sloužit jako forma měny s hodnotou určenou soukromým obchodem.

Zdroj

Token může představovat zdroj získaný nebo vytvořený ve sdílené ekonomice nebo prostředcích sdílení zdrojů; například úložiště nebo CPU token představující prostředky, které lze sdílet v síti.

Aktivum

Token může představovat vlastnictví vlastního nebo vnějšího, hmotného nebo nehmotného aktiva; například zlata, nemovitosti, auta, ropy, energie, předmětů z online her (MMOG), atd.

Přístup

Token může představovat přístupová práva a udělit přístup k digitálnímu nebo fyzickému majetku, jako je diskusní fórum, exkluzivní web, hotelový pokoj nebo půjčovna aut.

Vlastní kapitál

Token může představovat vlastní kapitál v digitální organizaci (např. DAO) nebo právnické osobě (např. v korporaci).

Hlasování

Token může představovat hlasovací práva v digitálním nebo právním systému.

Sběratelství

Token může představovat digitální sběratelský (collectible) předmět (např. CryptoPunks) nebo fyzický sběratelský předmět (např. obraz).

Identita

Token může představovat digitální identitu (např. Avatar) nebo legální identitu (např. občanský průkaz).

Osvědčení

Token může představovat certifikaci nebo potvrzení skutečnosti některou autoritou nebo decentralizovaným systémem reputace (např. záznam manželství, rodný list, vysokoškolský titul).

Obsluha

Token lze použít k přístupu nebo platbě za službu.

Jeden token často zahrnuje několik těchto funkcí. Někdy je mezi nimi těžké rozeznat, protože fyzické ekvivalenty byly vždy neoddělitelně spjaty. Například ve fyzickém světě je řidičský průkaz (osvědčení) rovněž dokladem totožnosti (totožnost) a tyto dva nelze oddělit. V digitální sféře mohou být dříve smíšené funkce odděleny a vyvíjeny samostatně (např. anonymní osvědčení).

Tokeny a zaměnitelnost

https://en.wikipedia.org/wiki/Fungibility [Wikipedia] říká: “ V ekonomice je zastupitelnost vlastností zboží, jehož jednotlivé jednotky jsou v podstatě zaměnitelné. “

Tokeny jsou zastupitelné, když můžeme nahradit jakoukoli jednu jednotku tokenu jinou, aniž by došlo k rozdílu v jeho hodnotě nebo funkci.

Přísně vzato, pokud je možné sledovat historický původ tokenu, pak není úplně zastupitelný. Schopnost sledovat původ může vést k seznamu zakázaných (blacklist) nebo povolených (whitelist) adres, snížení nebo eliminaci zastupitelnosti.

Nezaměnitelné tokeny jsou tokeny, z nichž každý představuje jedinečnou hmotnou nebo nehmotnou položku, a proto nejsou zaměnitelné. Například token, který představuje vlastnictví *konkrétního* obrazu od Van Gogha, není rovnocenný s jiným tokenem, který představuje obraz od Picassa, přestože mohou být součástí stejného systému „tokeny vlastnictví umění“. Podobně token představující specifický digitální sběratelský materiál, jako je konkrétní krypto koťátko (CryptoKitty), není zaměnitelný s jiným krypto koťátkem. Každý nezaměnitelný token je spojen s jedinečným identifikátorem, jako je například sériové číslo.

V další části této kapitoly si ukážeme příklady zaměnitelných i nezaměnitelných tokenů.

Všimněte si, že „zaměnitelný“ se často používá jako „přímo směnitelný za peníze“ (například žeton kasina může být „proplacen“, zatímco žetony prádla obvykle nemohou). Toto není smysl, ve kterém zde používáme toto slovo.

Riziko protistrany

Riziko protistrany je riziko, že strana v transakci nesplní své závazky. Některé typy transakcí jsou vystaveny dodatečnému riziku protistrany, protože jsou zapojeny více než dvě strany. Pokud například držíte depozitní certifikát pro drahý kov a prodáváte jej někomu, jsou v této transakci

alespoň tři strany: prodávající, kupující a depozitář drahého kovu. Ten, kdo drží fyzické aktivum, se nutně stává stranou při plnění transakce a zvyšují riziko protistrany u každé transakce týkající se tohoto aktiva. Obecně platí, že pokud se s aktivem obchoduje nepřímo prostřednictvím výměny symbolu vlastnictví, existuje další riziko protistrany ze strany depozitáře aktiva. Mají to aktivum? Rozpoznají (nebo povolí) převod vlastnictví na základě převodu tokenu (jako je certifikát, listina, titul nebo digitální token)? Ve světě digitálních tokenů představujících aktiva, stejně jako v nedigitálním světě, je důležité pochopit, kdo drží aktivum reprezentované tokenem a jaká pravidla platí pro dané podkladové aktivum.

Tokeny a vnitřnost

Slovo „vnitřnost“ ("intrinsic) je odvozeno z latinského slova „intra“, což znamená „zevnitř“.

Některé tokeny představují digitální položky, které jsou uvnitř bločenky. Tato digitální aktiva se řídí pravidly konsensu, stejně jako samotné tokeny. To má důležitý důsledek: tokeny, které představují vlastní aktiva, nepředstavují další riziko protistrany. Pokud držíte klíč pro krypto koťátko, neexistuje žádná jiná strana, která by vaše krypto koťátko držela za vás - vlastníte ji přímo. Platí pravidla konsenzu bločenky a vaše vlastnictví (tj. kontrola) soukromých klíčů je ekvivalentní vlastnictví aktiva bez jakéhokoli zprostředkovatele.

Naopak, mnoho žetonů se používá k reprezentaci *vnějších* (extrinsic) věcí, jako jsou nemovitosti, akcie s hlasovacím právem společnosti, ochranné známky a zlaté cihly. Vlastnictví těchto položek, které nejsou „uvnitř“ bločenky, se řídí zákonem, zvykem a politikou, odděleně od pravidel konsensu, kterými se řídí token. Jinými slovy, vydavatelé tokenů a vlastníci mohou stále záviset na skutečných vnějších smlouvách. Výsledkem je, že tato vnější aktiva nesou další riziko protistrany, protože jsou držena depozitáři, evidována v externích registrech nebo kontrolována zákony a politikou mimo prostředí bločenky.

Jedním z nejdůležitějších důsledků tokenů založených na bločence je schopnost převádět vnější aktiva na vnitřní aktiva a tím odstranit riziko protistrany. Dobrým příkladem je přechod z vlastního kapitálu v korporaci (vnější) na kmenový nebo hlasovací token v *DAO* nebo podobné (vnitřní) organizaci.

Používání tokenů: užitek nebo majetkový podíl

Téměř všechny projekty v Ethereu se dnes spouští s jakýmsi tokenem. Potřebují však všechny tyto

projekty tokeny? Existují nějaké nevýhody používání tokenu, nebo uvidíme slogan „tokenizovat všechny věci“? Použití tokenů lze v zásadě chápat jako konečný nástroj pro správu nebo organizaci. V praxi integrace bločenkových platforem, včetně Etherea, do stávajících struktur společnosti, má omezenou použitelnost, dosud existuje mnoho omezení.

Začněme objasněním role tokenu v novém projektu. Většina projektů používá tokeny jedním ze dvou způsobů: buď jako „obslužné tokeny“ (utility) nebo jako „tokeny vlastního kapitálu“ (equity). Tyto dvě role jsou velmi často spojeny.

Užitkové tokeny jsou ty, u nichž je pro získání přístupu ke službě, aplikaci nebo zdroji nutné použít token. Příklady obslužných tokenů zahrnují tokeny, které představují prostředky, jako je sdílené úložiště nebo přístup ke službám, jako jsou sítě sociálních médií.

Majetkové tokeny jsou ty, které představují podíly na ovládání nebo vlastnictví něčeho, jako je startup. Majetkové tokeny mohou být stejně omezené jako akcie bez hlasovacího práva pro distribuci dividend a zisků, nebo mohou být expanzivní jako akcie s hlasovacím právem v decentralizované autonomní organizaci, kde je řízení platformy prostřednictvím nějakého složitého systému správy založeného na hlasování držitelů tokenů.

Je to kachna!

Mnoho startupů čelí složitému problému: tokeny jsou skvělým mechanismem financování, ale nabízení cenných papírů (majetkový podíl) pro veřejnost je ve většině jurisdikcí regulovaná činnost. Zamaskováním tokenů vlastního kapitálu jako obslužných tokenů mnoho startupů doufá, že obejdou tato regulační omezení a získají peníze z veřejné nabídky a zároveň je prezentují jako předprodej „poukázek na přístup ke službám“ nebo, jak tomu říkáme, obslužných tokenů. Zda budou tyto pokusy maskované nabídky vlastního kapitálu schopny uspokojit regulátory, ještě uvidíme.

Jak říká lidové přísloví: „Pokud chodí jako kachna a kváká jako kachna, je to kachna.“ Regulátoři pravděpodobně nebudou těmito sémantickými drobnostmi rozptylováni; právě naopak, je pravděpodobné, že vidí takovou právní kličku jako pokus podvádět veřejnost.

====Obslužné Tokeny: Kdo je potřebuje?

Skutečným problémem je, že obslužné tokeny představují pro startupy značná rizika a překážky v adopci. Možná se ve vzdálené budoucnosti stane „tokenizace všech věcí“ realita, ale v současné době je množina lidí, kteří mají znalosti a touhu používat token, podmnožinou již tak malého trhu

kryptoměn.

Každá inovace představuje pro startup riziko a tržní filtr. Inovace vede po cestě nejméně vyšlapané a odchází z tradiční cesty. Je to skutečně osamělá procházka. Pokud se startup pokouší inovovat v nové oblasti technologie, jako je sdílení úložiště přes P2P sítě, je to dost osamělá cesta. Přidání obslužného tokenu do této inovace a vyžadování, aby uživatelé přijali tokeny za účelem využití služby, zvyšuje riziko a zvyšuje překážky přijetí. Je to krok z již osamělé stezky inovace P2P úložiště přímo do divočiny.

Každou inovaci považujte za filtr. Omezuje přijímání na podmnožinu trhu, která se může stát raným osvojitelem této inovace. Přidání druhého filtru dále omezuje adresovatelný trh. Žádáte své dřívější osvojitele, aby přijali nejen jednu, ale dvě zcela nové technologie: novou aplikaci / platformu / službu, kterou jste vytvořili, a tokenovou ekonomiku.

Pro startupy, každá inovace představuje rizika, která zvyšují pravděpodobnost selhání startupu. Pokud využijete svůj již riskantní nápad pro startup a přidáte obslužný token, přidáváte všechna rizika základní platformy (Ethereum), širší ekonomiky (burzy, likvidita), regulačního prostředí (regulátory akcií / komodit) a technologií (chytré kontrakty , standardy tokenů). To je pro startup velké riziko.

Zastánci „tokenize všech věcí“ budou pravděpodobně uvádět opačné argumenty, že přijetím tokenů také zdědí tržní nadšení, dřívější osvojitele, technologii, inovaci a likviditu celé tokenové ekonomiky. To je také pravda. Otázkou je, zda přínosy a nadšení převažují nad riziky a nejistotami.

Nicméně některé z nejinovativnějších podnikatelských nápadů se skutečně odehrávají v kryptoměně. Pokud regulační orgány nejsou dostatečně rychlé na to, aby přijaly zákony a podporovaly nové obchodní modely, budou se podnikatelé a související talenty snažit působit v jiných jurisdikcích, které jsou více kryptografické. To se již děje.

Nakonec jsme na začátku této kapitoly při zavádění tokenů hovořili o hovorovém významu „tokenu“ jako o „něčem bezvýznamném“. Základním důvodem pro zanedbatelnou hodnotu většiny tokenů je to, že je lze použít pouze ve velmi úzkém kontextu: jedna autobusová společnost, jedna prádelna, jedna podloubí, jeden hotel nebo jeden firemní obchod. Omezená likvidita, omezená použitelnost a vysoké náklady na konverzi snižují hodnotu tokenů, dokud nejsou pouze „token“. Když tedy na svou platformu přidáte obslužný token, ale tento token lze použít pouze na jedné platformě s malým trhem, znovu vytváříte podmínky, díky nimž jsou fyzické žetony bezcenné. To může být opravdu správný způsob, jak začlenit tokenizaci do vašeho projektu. Pokud však uživatel potřebuje použít

vaši platformu, musí převést něco na váš obslužný token, použít jej a poté převést zbytek zpět na něco obecně užitečnějšího, vytvořili jste firemní měnu s nuceným oběhem. Náklady na přepínání digitálního tokenu jsou řádově nižší než u fyzického tokenu bez trhu, ale nejsou nulové. Obslužné tokeny, které fungují napříč celým odvětvím, budou velmi zajímavé a pravděpodobně docela cenné. Pokud však nastavíte startup tak, aby musel zavést celý průmyslový standard, abyste uspěli, možná jste již selhali.

Jednou z výhod nasazení služeb na univerzálních platformách, jako je Ethereum, je schopnost propojit chytré kontrakty (a tedy i obslužné tokeny) napříč projekty, což zvyšuje potenciál likvidity a užitečnosti tokenů.

Toto rozhodnutí učiňte ze správných důvodů. Zaveďte token, protože vaše aplikace nemůže fungovat bez tokenu_. Zaveďte jej, protože token odstraňuje zásadní překážku na trhu nebo řeší problém s přístupem. Nezavádějte obslužný token, protože je to jediný způsob, jak můžete rychle získat peníze a musíte předstírat, že se nejedná o veřejnou nabídku cenných papírů .

Ethereum tokeny

Bločenkové tokeny existovaly před Ethereem. V některých ohledech je první bločenková měna, Bitcoinem, sám o sobě token. Před platformou Ethereum bylo také vyvinuto mnoho tokenových platforem na Bitcoinu a dalších kryptoměnách. Zavedení prvního standardu tokenů na Ethereu však vedlo k explozi tokenů.

Vitalik Buterin navrhl tokeny jako jednu z nejzjevnějších a nejužitečnějších aplikací zobecněného programovatelné bločenky, jako je Ethereum. Ve skutečnosti v prvním rocea Etherea bylo běžné, že Vitalik a další měli na sobě trička zdobená logem Etherea a na zadní straně kus chytrého kontraktu. Tato trička měla několik variant, ale nejběžnější ukazovala implementaci tokenu.

Než se pustíme do podrobností o vytváření Ethereum tokenů, je důležité mít přehled o tom, jak tokeny fungují na Ethereu. Tokeny se liší od etheru, protože o nich Ethereum protokol nic neví. Posílání etheru je vlastní činností platformy Ethereum, ale odesílání nebo dokonce vlastnění tokenů není. Etherový zůstatek Ethereum účtů je zpracován na úrovni protokolu, zatímco tokenový zůstatek Ethereum účtů je zpracován na úrovni chytrých kontraktů. Chcete-li vytvořit nový token na Ethereu, musíte vytvořit nový chytrý kontrakt. Po nasazení chytrý kontrakt zpracovává vše, včetně vlastnictví, převodů a přístupových práv. Svůj chytrý kontrakt můžete napsat, abyste mohli provést všechny potřebné akce jakýmkoli způsobem, ale pravděpodobně je nejmoudřejší sledovat existující

standardy. Dále se podíváme na takovéto standardy. Na konci kapitoly diskutujeme o výhodách a nevýhodách následujících standardů.

Standard tokenu ERC20

První standard byl představen v listopadu 2015 Fabianem Vogelstellerem jako žádost o připomínku k Ethereu (ERC). Bylo mu automaticky přiděleno číslo 20 na GitHubu, čímž vznikl název „ERC20 token“. Převážná většina tokenů je v současné době založena na standardu ERC20. Žádost o připomínky ERC20 se nakonec stala návrhem na vylepšení 20 (EIP-20), ale většinou se na něj stále odkazuje původní název ERC20.

ERC20 je standard pro *zaměnitelné tokeny*, což znamená, že různé jednotky ERC20 tokenu jsou vzájemně zaměnitelné a nemají žádné jedinečné vlastnosti.

ERC20 standard [http://bit.ly/2CUf7WG] definuje společné rozhraní pro kontrakty implementující token, takže ke každému kompatibilnímu tokenu lze přistupovat a používat ho stejným způsobem. Rozhraní se skládá z řady funkcí, které musí být přítomny v každé implementaci standardu, jakož i z některých volitelných funkcí a atributů, které mohou vývojáři přidat.

ERC20 požadované funkce a události

Kontrakt tokenu v souladu s ERC20 musí poskytovat alespoň následující funkce a události:

totalSupply::Vrátí celkové množství jednotek tohoto tokenu, které aktuálně existují. Tokeny ERC20 mohou mít pevný nebo variabilní počet jednotek.

balanceOf

Pro danou adrese vrací tokenový zůstatek této adresy.

transfer

Pro zadanou adresu a množství převede toto množství tokenů na tuto adresu ze zůstatku adresy, která provedla převod.

transferFrom

Pro zadaného odesílatele, příjemce a množství převádí tokeny z jednoho účtu na druhý. Používá se v kombinaci s approved.

approve

Pro zadanou adresu příjemce a množství opravňuje tuto adresu k provedení několika převodů z účtu, který vydal schválení, až do tohoto uvedeného množství.

allowance

Pro zadanou adresu majitele a adresu disponenta vrátí zbývající částku, kterou má disponent schválenou k výběru od vlastníka.

Transfer

Událost spuštěná po úspěšném převodu (volání transfer nebo transferFrom) (i pro převody s nulovým množstvím).

Approval

Událost zaznamenaná po úspěšném volání approve.

Volitelné funkce ERC20

Kromě požadovaných funkcí uvedených v předchozí části jsou standardem definovány také následující volitelné funkce:

name

Vrací lidsky čitelné jméno (např. „americké dolary“) tokenu.

symbol

Vrací lidsky čitelný symbol (např. „USD“) tokenu.

decimals

Vrací počet desetinných míst použitých k rozdělení množství tokenů. Pokud má například decimals hodnotu 2, pak je částka tokenu vydělena 100, aby se získalo jeho uživatelské zobrazení.

Rozhraní ERC20 definované v Solidity

Zde je uvedeno, jak specifikace ERC20 rozhraní vypadá v Solidity:

```
contract ERC20 {
   function totalSupply() constant returns (uint theTotalSupply);
   function balanceOf(address _owner) constant returns (uint balance);
   function transfer(address _to, uint _value) returns (bool success);
   function transferFrom(address _from, address _to, uint _value) returns
      (bool success);
   function approve(address _spender, uint _value) returns (bool success);
   function allowance(address _owner, address _spender) constant returns
      (uint remaining);
   event Transfer(address indexed _from, address indexed _to, uint
_value);
   event Approval(address indexed _owner, address indexed _spender, uint
_value);
}
```

Struktura dat ERC20

Pokud prozkoumáte jakoukoli implementaci ERC20, uvidíte, že obsahuje dvě datové struktury, jednu pro sledování zůstatků a druhou pro sledování povolení. V Solidity jsou implementovány pomocí *mapování dat*.

První mapování dat implementuje interní tabulku bilancí tokenů podle vlastníka. To umožňuje tokenovému kontraktu sledovat, kdo vlastní tokeny. Každý převod je odečten z jednoho zůstatku a přičten k jinému zůstatku:

```
mapping(address => uint256) balances;
```

Druhou datovou strukturou je datové mapování povolení. Jak uvidíme v další části, s tokeny ERC20 může vlastník tokenu delegovat oprávnění na disponenta (spender), což mu umožní utratit určité povolené množství (allowance) ze zůstatku vlastníka. ERC20 kontrakt sleduje povolenky pomocí dvourozměrného mapování, přičemž primárním klíčem je adresa majitele tokenu, mapovaná na adresu disponenta a povolené množství:

```
mapping (address => mapping (address => uint256)) public allowed;
```

Pracovní postupy ERC20: "transfer" a "approve & transferFrom"

Tokenový standard ERC20 má dvě převodové funkce. Možná se divíte, proč.

ERC20 umožňuje dva různé pracovní postupy. První je jednoduchá transakce, přímý pracovní postup pomocí funkce transfer. Tento pracovní postup je ten, který používají peněženky k odesílání tokenů na jiné peněženky. Převážná většina převodů tokenů se odehrává v rámci pracovního postupu transfer.

Provedení převodu v rámci kontraktu je velmi jednoduché. Pokud chce Alice poslat Bobovi 10 tokenů, její peněženka odešle transakci na adresu tokenového kontraktu, volá funkci `transfer` s parametry Bobova adresa a 10. Tokenový kontrakt upraví zůstatek Alice (–10) a Bobův zůstatek (+10) a vyšle událost Transfer.

Druhým pracovním postupem jsou dvě transakce, nejprve volání approval následované voláním transferFrom. Tento pracovní postup umožňuje vlastníkovi tokenu delegovat jeho kontrolu na jinou adresu. Nejčastěji se používá k delegování kontroly na kontrakt o distribuci tokenů, ale lze ho také použít burzami.

Například pokud společnost prodává žetony pro ICO, může schválit (approve) adrese kontraktu veřejného prodeje distribuovat určité množství tokenů. Prodejní kontrakt pak může převést (transferFrom) zůstatek majitele tokenového kontraktu na každého kupujícího tokenu, jak je znázorněno v Schválení převodu a převod ERC20 tokenů.

Počáteční nabídka mincí (Initial Coin Offering; CO) je mechanismus skupinového financování používaný společnostmi a organizacemi k získávání peněz prodejem tokenů. Termín je odvozen od počáteční veřejné nabídky (Initial Public Offering; IPO), což je proces, kterým veřejná společnost nabízí akcie k prodeji investorům na burze. Na rozdíl od vysoce regulovaných trhů IPO jsou ICO otevřené, globální a chaotické. Příklady a vysvětlení ICO v této knize nejsou schvalováním tohoto typu získávání prostředků.

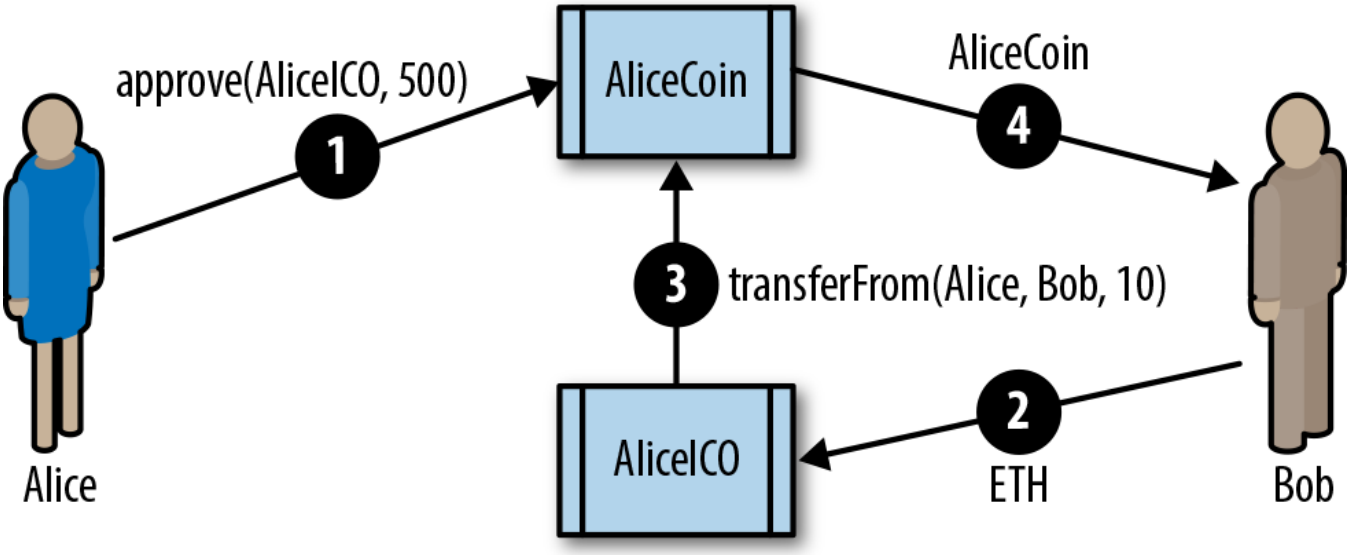

Figure 38. Schválení převodu a převod ERC20 tokenů

Pro pracovní postup approve & transferFrom jsou nutné dvě transakce. Řekněme, že Alice chce povolit kontraktu AliceICO, aby prodal 50% všech tokenů AliceCoin kupujícím, jako je Bob a Charlie. Nejprve Alice nasadí kontrakt AliceCoin ERC20 a vydá veškeré AliceCoin tokeny ve prospěch své vlastní adresy. Poté Alice nasadí AliceICO kontrakt, která může prodávat tokeny za ether. Poté Alice zahájí pracovní postup approve & transferFrom. Odešle transakci kontraktu AliceCoin, volá schválení approval s adresou kontraktu AliceICO a 50% z totalSupply jako parametry. Tím se spustí událost schválení Approval. Nyní může kontrakt AliceICO prodávat AliceCoin.

Když kontrakt AliceICO obdrží od Boba ether, musí na oplátku poslat Bobovi nějaké tokeny AliceCoin. V rámci kontraktu AliceICO je směnný kurz mezi AliceCoin tokeny a etherem. Směnný kurz, který Alice nastavila, když vytvořila kontrakt +AliceICO, určuje, kolik tokenů Bob obdrží za dané množství etheru zaslané do kontraktu AliceICO. Když kontrakt AliceICO volá funkci AliceCoin transferFrom, nastaví adresu Alice jako odesílatele a adresu Boba jako příjemce a pomocí směnného kurzu určí, kolik tokenů AliceCoin bude převedeno na Boba v poli hodnota value. Kontrakt AliceCoin převádí zůstatek z adresy Alice na Bobovu adresu a spouští událost Transfer. Kontrakt AliceICO může volat transferFrom neomezeně krát, pokud nepřekročí Alicí stanovený limit množství tokenů. Kontrakt AliceICO může sledovat, kolik tokenů AliceCoin může prodat voláním funkce allowance.

Implementace ERC20

I když je možné implementovat token kompatibilní s ERC20 na přibližně 30 řádcích kódu Solidity, většina implementací je složitější. To je z důvodu zabezpečení proti možným chybám. Standard EIP-20 uvádí dvě implementace:

Consensys EIP20 [http://bit.ly/2EUYCMR]

Jednoduchá a snadno čitelná implementace tokenu kompatibilního s ERC20.

OpenZeppelin StandardToken [https://bit.ly/2xPYck6]

Tato implementace je kompatibilní s ERC20 a s dalšími bezpečnostními opatřeními. Tvoří základ knihoven OpenZeppelin, které implementují složitější tokeny kompatibilní s ERC20 s limitem skupinového investování, aukcemi, dočasného zmražení (vesting schedules) a dalšími funkcemi.

Spuštění vlastního ERC20 tokenu

"ERC20 token standard","METoken creation/launch example", id="ix_10tokens-asciidoc9", range="startofrange")Let's create and launch our own token. For this example, we will use the Truffle framework. The example assumes you have already installed truffle and configured it, and are familiar with its basic operation (for details, see Truffle).

Nazveme náš token „Mastering Ethereum Token“ se symbolem „MET“.

Tento příklad najdete v GitHub úložišti knihy [https://github.com/ethereumbook/ethereumbook/blob/develop/code/truffle/METoken].

Nejprve vytvořme a inicializujeme adresář projektu Truffle. Spusťte tyto čtyři příkazy a přijměte výchozí odpovědi na jakékoli otázky:

```
<pre data-type="programlist">
$ <strong>mkdir METoken</strong>
$ <strong>cd METoken</strong>
METoken $ <strong>truffle init</strong>
METoken $ <strong>npm init</strong>
</pre>
```

Nyní byste měli mít následující strukturu adresářů:

```
METoken/
+---- contracts
|   `---- Migrations.sol
+---- migrations
|   `---- 1_initial_migration.js
+---- package.json
+---- test
+---- truffle-config.js
`---- truffle.js
```

Upravte konfigurační soubor *truffle.js* nebo *truffle-config.js* a nastavte prostředí Truffle nebo jej zkopírujte z repozitáře [http://bit.ly/2DdP2mz].

Pokud použijete příklad *truffle-config.js*, nezapomeňte vytvořit soubor *.env* ve složce *METoken* obsahující vaše testovací soukromé klíče pro testování a nasazení ve veřejných testovacích Ethereum sítích, jako jsou Ropsten nebo Kovan. Váš soukromý klíč pro testovací síť můžete exportovat z MetaMasku.

Poté by měl váš adresář vypadat takto:

```
METoken/
+---- contracts
|   `---- Migrations.sol
+---- migrations
|   `---- 1_initial_migration.js
+---- package.json
+---- test
+---- truffle-config.js
+---- truffle.js
`---- .env *new file*
```

Používejte pouze testovací klíče nebo testovací mnemotechnická slova, které nejsou používány k držení prostředků v hlavní Ethereum síti. *Nikdy* nepoužívejte klíče, které drží skutečné peníze pro testování.

V našem příkladu importujeme knihovnu OpenZeppelin, která implementuje některé důležité

bezpečnostní kontroly a lze ji snadno rozšířit:

```
<pre data-type="programlist">
$ <strong>npm install openzeppelin-solidity@1.12.0</strong>

+ openzeppelin-solidity@1.12.0
added 1 package from 1 contributor and audited 2381 packages in 4.074s
</pre>
```

Balíček openzeppelin-solidity přidá asi 250 souborů do adresáře *node_modules*. Knihovna OpenZeppelin obsahuje mnohem více než ERC20 token, ale použijeme pouze malou část.

Dále pojďme napsat náš kontrakt tokenu. Vytvořte nový soubor *METoken.sol* a zkopírujte kód příkladu z GitHubu [http://bit.ly/2qfIFH0].

Náš kontrakt uvedený v METoken.sol: Solidity kontrakt implementující ERC20 token je velmi jednoduchý, protože zdědí veškerou svou funkčnost z knihovny OpenZeppelin.

Example 20. METoken.sol: Solidity kontrakt implementující ERC20 token

```
pragma solidity ^0.4.21;

import 'openzeppelin-solidity/contracts/token/ERC20/StandardToken.sol';

contract METoken is StandardToken {
    string public constant name = 'Mastering Ethereum Token';
    string public constant symbol = 'MET';
    uint8 public constant decimals = 2;
    uint constant _initial_supply = 2100000000;

    function METoken() public {
        totalSupply_ = _initial_supply;
        balances[msg.sender] = _initial_supply;
        emit Transfer(address(0), msg.sender, _initial_supply);
    }
}
```

Zde definujeme volitelné proměnné name, symbol, a decimals. Definujeme také proměnnou

_initial_supply nastavenou na 21 milionů tokenů; se dvěma desetinnými místy rozdělení, což dává celkem 2,1 miliardy jednotek. V funkci inicializace kontraktu (konstruktoru) jsme nastavili totalSupply tak, aby se rovnalo _initial_supply a alokovali všechny _initial_supply do zůstatku na účtu (msg.sender), který vytváří kontrakt METoken.

Nyní zkompilujeme METoken kód pomocí truffle:

```
<pre data-type="programlist">
$ <strong>truffle compile</strong>
Compiling ./contracts/METoken.sol...
Compiling ./contracts/Migrations.sol...
Compiling openzeppelin-solidity/contracts/math/SafeMath.sol...
Compiling openzeppelin-solidity/contracts/token/ERC20/BasicToken.sol...
Compiling openzeppelin-solidity/contracts/token/ERC20/ERC20.sol...
Compiling openzeppelin-solidity/contracts/token/ERC20/ERC20Basic.sol...
Compiling openzeppelin-solidity/contracts/token/ERC20/StandardToken.sol...
</pre>
```

Jak vidíte, truffle obsahuje potřebné závislosti z knihoven OpenZeppelin a tyto kontrakty také kompiluje.

Nastavíme migrační skript pro nasazení kontraktu METoken. Ve složce *METoken/migrations* vytvořte nový soubor s názvem *2_deploy_contracts.js*. Zkopírujte obsah z příkladu v úložišti GitHub [http://bit.ly/2P0rHLl]:

2_deploy_contracts: Migrace pro nasazení METoken

```
var METoken = artifacts.require("METoken");

module.exports = function(deployer) {
  // Deploy the METoken contract as our only task
  deployer.deploy(METoken);
};
```

Než to nasadíme do jedné z Ethereum testovacích sítí , spustíme místní bločenku, který vše otestuje. Spusťte bločenku ganache, buď z příkazové řádky pomocí ganache-cli, nebo z grafického uživatelského rozhraní.

Jakmile bude spuštěn ganache, můžeme nasadit náš METoken token a zjistit, zda všechno funguje podle očekávání:

```
<pre data-type="programlist">
$ <strong>truffle migrate --network ganache</strong>
Using network 'ganache'.

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ... 0xb2e90a056dc6ad8e654683921fc613c796a03b89df6760ec1db1084ea4a084eb
  Migrations: 0x8cdaf0cd259887258bc13a92c0a6da92698644c0
Saving successful migration to network...
  ... 0xd7bc86d31bee32fa3988f1c1eabce403a1b5d570340a3a9cdba53a472ee8c956
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying METoken...
  ... 0xbe9290d59678b412e60ed6aefedb17364f4ad2977cfb2076b9b8ad415c5dc9f0
  METoken: 0x345ca3e014aaf5dca488057592ee47305d9b3e10
Saving successful migration to network...
  ... 0xf36163615f41ef7ed8f4a8f192149a0bf633fe1a2398ce001bf44c43dc7bdda0
Saving artifacts...
</pre>
```

Na ganache příkazové řádce bychom měli vidět, že naše implementace vytvořila čtyři nové transakce, jak je znázorněno v Nasazení METoken na ganache.

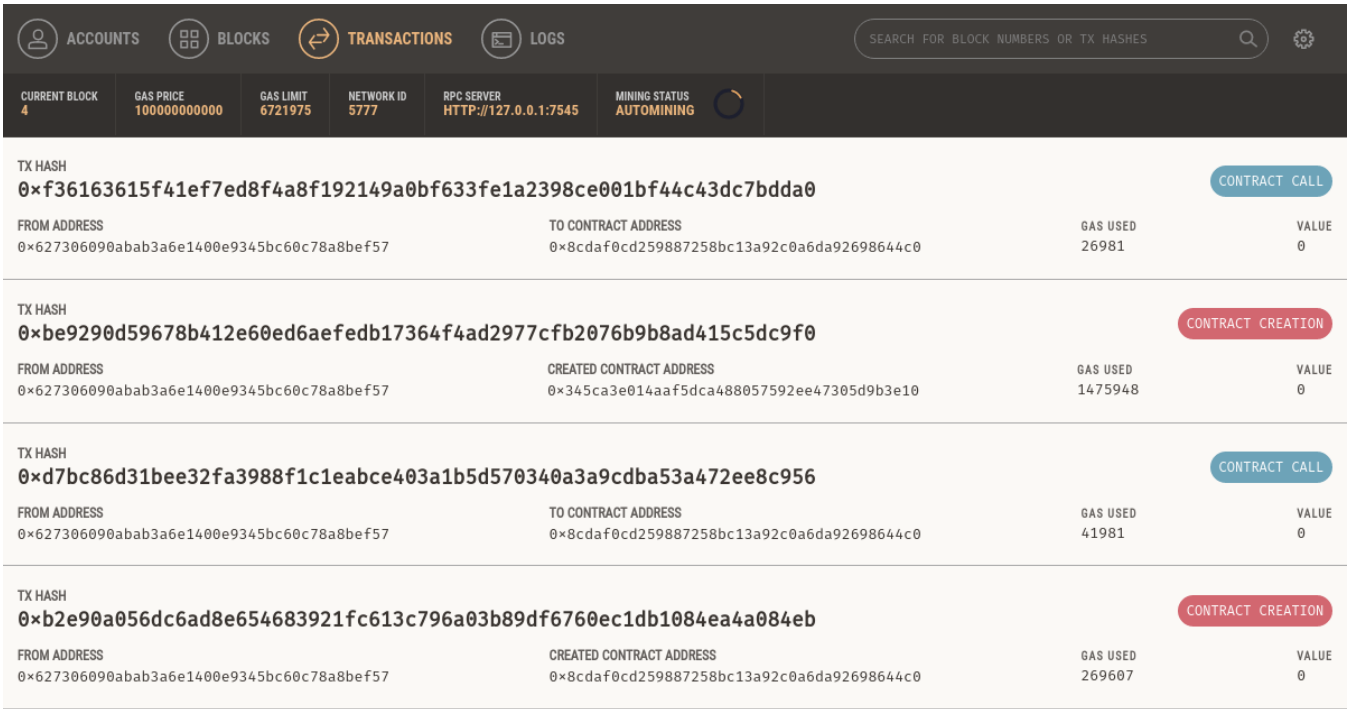

Figure 39. Nasazení METoken na ganache

Interakce s METoken pomocí příkazové řádky Truffle

Můžeme komunikovat s naším kontraktem na ganache bločence pomocí příkazové řádky Truffle. Toto je interaktivní prostředí JavaScriptu, které poskytuje přístup do prostředí Truffle a přes web3 do bločenky. V tomto případě připojíme konzolu Truffle k ganache bločence:

```
<pre data-type="programlist">
$ <strong>truffle console --network ganache</strong>
truffle(ganache)&gt;
</pre>
```

Výzva truffle(ganache)> ukazuje, že jsme připojeni k ganache bločence a jsme připraveni napsat naše příkazy. Příkazová řádka Truffle podporuje všechny příkazy truffle, takže bychom mohli kompilovat a migrovat z příkazové řádky. Tyto příkazy již byly spuštěny, takže pojďme přímo k samotnému kontraktu. Kontrakt METoken existuje jako JavaScript objekt v prostředí Truffle. Na výzvu zadejte **METoken**+ a vypíše se celá definice kontraktu:

```
<pre data-type="programlist">
truffle(ganache)&gt; <strong>METoken</strong>
{ [Function: TruffleContract]
  _static_methods:

[...]

currentProvider:
 HttpProvider {
   host: 'http://localhost:7545',
   timeout: 0,
   user: undefined,
   password: undefined,
   headers: undefined,
   send: [Function],
   sendAsync: [Function],
   _alreadyWrapped: true },
network_id: '5777' }
</pre>
```

Objekt METoken odhaluje také několik atributů, například adresu kontraktu (jak je nasazena příkazem migrate):

```
<pre data-type="programlist">
truffle(ganache)&gt; <strong>METoken.address</strong>
'0x345ca3e014aaf5dca488057592ee47305d9b3e10'
</pre>
```

Pokud chceme komunikovat s nasazeným kontraktem, musíme použít asynchronní volání ve formě JavaScriptu „slibu“. Pomocí funkce deployed získáme instanci kontraktu a poté zavoláme funkci totalSupply:

```
<pre data-type="programlist">
truffle(ganache)&gt; <strong>METoken.deployed().then(instance =>
instance.totalSupply())</strong>
BigNumber { s: 1, e: 9, c: [ 2100000000 ] }
</pre>
```

Dále použijeme účty vytvořené ganache ke kontrole našeho METoken zůstatku a pošleme nějaký METoken na jinou adresu. Nejprve získáme adresy účtu:

```
<pre data-type="programlist">
truffle(ganache)&gt; <strong>let accounts</strong>
undefined
truffle(ganache)&gt; <strong>web3.eth.getAccounts((err,res) => { accounts
= res })</strong>
undefined
truffle(ganache)&gt; <strong>accounts[0]</strong>
'0x627306090abab3a6e1400e9345bc60c78a8bef57'
</pre>
```

Seznam accounts nyní obsahuje všechny účty vytvořené pomocí ganache+ a account[0] je účet, který nasadil kontrakt METoken. Měl by mít zůstatek METoken, protože náš METoken konstruktor dává celý tokenovou zásobu na adresu, která ho vytvořila. Zkontrolujme:

```
<pre data-type="programlist">
truffle(ganache)&gt; <strong>METoken.deployed().then(instance =></strong>
                  <strong>{
instance.balanceOf(accounts[0]).then(console.log) })</strong>
undefined
truffle(ganache)&gt; <strong>BigNumber { s: 1, e: 9, c: [ 2100000000 ]
}</strong>
</pre>
```

Nakonec převedeme 1 000,00 METoken z account[0] na account[1] voláním funkce kontraktu transfer:

```
<pre data-type="programlist">
truffle(ganache)&gt; <strong>METoken.deployed().then(instance =>
                  { instance.transfer(accounts[1], 100000) })</strong>
undefined
truffle(ganache)&gt; <strong>METoken.deployed().then(instance =>
                  { instance.balanceOf(accounts[0]).then(console.log)
})</strong>
undefined
truffle(ganache)&gt; <strong>BigNumber { s: 1, e: 9, c: [ 2099900000 ]
}</strong>
undefined
truffle(ganache)&gt; <strong>METoken.deployed().then(instance =>
                  { instance.balanceOf(accounts[1]).then(console.log)
})</strong>
undefined
truffle(ganache)&gt; <strong>BigNumber { s: 1, e: 5, c: [ 100000 ]
}</strong>
</pre>
```

METoken má 2 desetinná místa přesnosti, což znamená, že 1 METoken je v kontraktu 100 jednotek. Když převedeme 1 000 METoken, určíme hodnotu ve volání 100000 při volání funkce transfer.

Jak vidíte, v v příkazové řádce má account[0] nyní 20 999 000 MET a account[1] má 1 000 MET.

Pokud přepnete do grafického uživatelského rozhraní+ganache+, viz Přesun METoken na ganache uvidíte transakci, která volala funkci transfer .

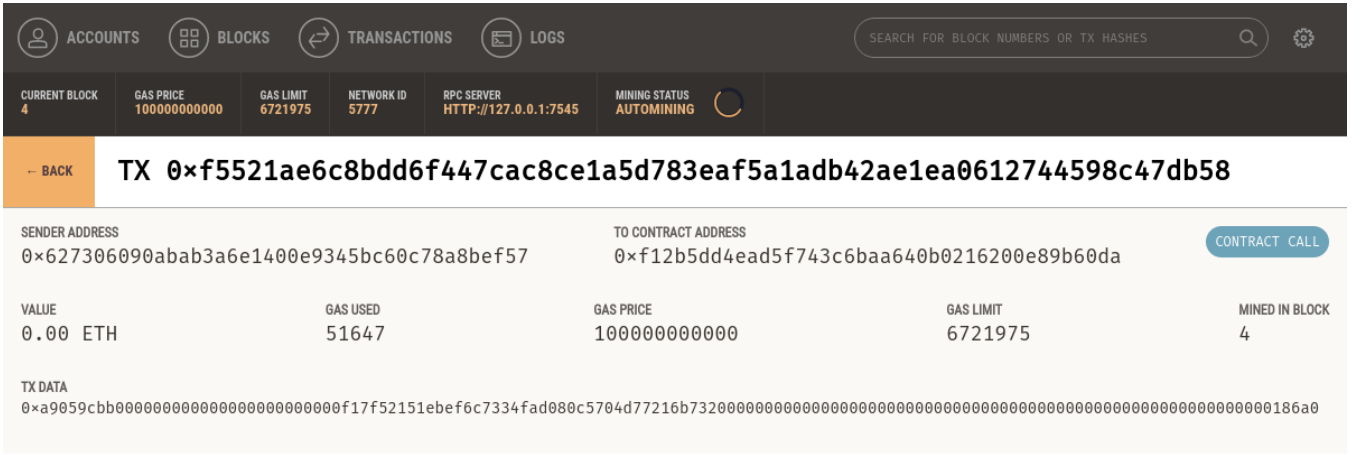

Figure 40. Přesun METoken na ganache

Odesílání ERC20 tokenů na adresy kontraktu

Zatím jsme nastavili ERC20 token a převedli některé tokeny z jednoho účtu na druhý. Všechny účty, které jsme použili pro tyto demonstrace, jsou externě vlastněné účty, což znamená, že jsou ovládány soukromým klíčem, nikoli kontraktem. Co se stane, pokud pošleme MET na adresu kontraktu? Pojďme to zjistit!

Nejprve nasaďme do našeho testovacího prostředí další kontrakt. V tomto příkladu použijeme náš první kontrarkt *Faucet.sol*. Přidáme jej do projektu METoken zkopírováním do adresáře *contracts*. Náš adresář by měl vypadat takto:

```
METoken/
+---- contracts
|   +---- Faucet.sol
|   +---- METoken.sol
|   `---- Migrations.sol
```

Přidáme také migraci, abychom nasadili Faucet odděleně od METoken:

```
var Faucet = artifacts.require("Faucet");

module.exports = function(deployer) {
// Nasadit kontrakt Faucet jako náš jediný úkol
  deployer.deploy(Faucet);
};
```

Zkompilujme a migrujme kontrakty z konzole Truffle:

```
<pre data-type="programlist">
$ <strong>truffle console --network ganache</strong>
truffle(ganache)&gt; <strong>compile</strong>
Compiling ./contracts/Faucet.sol...
Writing artifacts to ./build/contracts

truffle(ganache)&gt; <strong>migrate</strong>
Using network 'ganache'.

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ... 0x89f6a7bd2a596829c60a483ec99665c7af71e68c77a417fab503c394fcd7a0c9
  Migrations: 0xa1ccce36fb823810e729dce293b75f40fb6ea9c9
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Replacing METoken...
  ... 0x28d0da26f48765f67e133e99dd275fac6a25fdfec6594060fd1a0e09a99b44ba
  METoken: 0x7d6bf9d5914d37bcba9d46df7107e71c59f3791f
Saving artifacts...
Running migration: 3_deploy_faucet.js
  Deploying Faucet...
  ... 0x6fbf283bcc97d7c52d92fd91f6ac02d565f5fded483a6a0f824f66edc6fa90c3
  Faucet: 0xb18a42e9468f7f1342fa3c329ec339f254bc7524
Saving artifacts...
</pre>
```

Skvělý. Nyní pošleme nějaký MET do kontraktu Faucet:

```
<pre data-type="programlist">
truffle(ganache)&gt; <strong>METoken.deployed().then(instance =>
                  { instance.transfer(Faucet.address, 100000) })</strong>
truffle(ganache)&gt; <strong>METoken.deployed().then(instance =>
                  {
instance.balanceOf(Faucet.address).then(console.log)})</strong>
truffle(ganache)&gt; <strong>BigNumber { s: 1, e: 5, c: [ 100000 ]
}</strong>
</pre>
```

Dobře, převedli jsme 1 000 MET na kontrakt Faucet. Jak nyní vybereme tyto tokeny?

Pamatujte, *Faucet.sol* je velmi jednoduchý kontrakt. Má pouze jednu funkci, `withdraw`, což je pro vybrání *etheru*. Nemá funkci pro vybrání MET nebo jiného ERC20 tokenu. Použijeme-li withdraw, pokusí se odeslat ether, ale protože Faucet zatím nemá zůstatek etheru, funkce selže.

Kontrakt METoken ví, že Faucet má zůstatek, ale jediný způsob, jak může tento zůstatek převést, je, pokud obdrží volání transfer z adresy kontraktu. Nějak potřebujeme, aby kontrakt Faucet volal funkci transfer v `METoken`.

Pokud vás zajímá, co dělat dál, tak neexistuje řešení tohoto problému. MET zaslaný na Faucet je zaseknutý navždy. Převést ho může pouze kontrakt Faucet a kontrakt Faucet nemá kód pro volání funkce transfer kontraktu ERC20 tokenu.

Možná jste tento problém předvídali. Pravděpodobně ale ne. Ve skutečnosti ani stovky Ethereum uživatelů, kteří omylem převedli různé tokeny na kontrakty, které neměly žádnou schopnost pracovat s ERC20. Podle některých odhadů se tokeny v hodnotě více než zhruba 2,5 milionu USD (v době psaní) „takto zasekly“ a jsou navždy ztraceny.

Jedním ze způsobů, jak uživatelé ERC20 tokenů mohou nechtěně ztratit své tokeny při přesunu, je pokus o přesun na burzu nebo jinou službu. Zkopírují Ethereum adresu z webové stránky burzy a myslí si, že na ni mohou jednoduše poslat tokeny. Mnoho burz však zveřejňuje přijímací adresy, které jsou ve skutečnosti kontrakty! Účelem těchto kontraktů je přijímat pouze éter, nikoliv ERC20 tokeny, nejčastěji přeposílají všechny prostředky, které jim byly zaslány do „studeného úložiště“ nebo jiné centralizované peněženky. Přes mnoho varování, která říká „neposílejte tokeny na tuto adresu,“ se tímto způsobem ztratí mnoho tokenů.

Demonstrace pracovního postupu „Schválit a převést“

Náš kontrakt Faucet nemohl zpracovat ERC20 tokeny. Odeslání tokenů pomocí funkce transfer vedlo ke ztrátě těchto tokenů. Pojďme nyní přepsat kontrakt a nechat ji zpracovat ERC20 tokeny. Konkrétně z toho uděláme kohoutek, který rozdá MET každému, kdo o to požádá.

V tomto příkladu vytvoříme kopii projektového adresáře *truffle* (nazveme jej *METoken_METFaucet*), inicializujeme truffle a npm, instalujeme závislosti OpenZeppelin a zkopírujeme kontrakt *METoken.sol*. Viz náš první příklad, v Spuštění vlastního ERC20 tokenu pro podrobné pokyny.

Náš nový kontrakt kohoutku *METFaucet.sol*, bude vypadat jako METFaucet.sol: kohoutek pro METoken.

Example 21. METFaucet.sol: kohoutek pro METoken

```
// Version of Solidity compiler this program was written for
pragma solidity ^0.4.19;

import 'openzeppelin-solidity/contracts/token/ERC20/StandardToken.sol';


// A faucet for ERC20 token MET
contract METFaucet {

    StandardToken public METoken;
    address public METOwner;

    // METFaucet constructor, provide the address of METoken contract
and
    // the owner address we will be approved to transferFrom
    function METFaucet(address _METoken, address _METOwner) public {

        // Initialize the METoken from the address provided
        METoken = StandardToken(_METoken);
        METOwner = _METOwner;
    }

    function withdraw(uint withdraw_amount) public {

        // Limit withdrawal amount to 10 MET
        require(withdraw_amount <= 1000);

        // Use the transferFrom function of METoken
        METoken.transferFrom(METOwner, msg.sender, withdraw_amount);
    }

    // REJECT any incoming ether
    function () external payable { revert(); }

}
```

V základním příkladu Faucet jsme provedli několik změn. Protože METFaucet bude používat funkci

transferFrom v METoken,, bude potřebovat dvě další proměnné. Jedna bude udržovat adresu nasazeného METoken kontraktu. Druhá bude držet adresu majitele MET, který schválí výběry z kohoutku. Kontrakt METFaucet zavolá METoken.transferFrom a dá jí pokyn, aby přesunul MET od vlastníka na adresu, odkud přišel požadavek na výběr z kohoutku.

Zde deklarujeme tyto dvě proměnné:

```
StandardToken public METoken;
address public METOwner;
```

Protože náš kohoutek musí být inicializován se správnými adresami pro METoken a METOwner, musíme naprogramovat vlastní konstruktor:

```
// METFaucet konstruktor - zadejte adresu kontraktu METoken a
// adresu vlastníka, který bude schválený k transferFrom
function METFaucet(address _METoken, address _METOwner) public {

// Inicializujte METokenu z poskytnuté adresy
    METoken = StandardToken(_METoken);
    METOwner = _METOwner;
}
```

Další změna je ve funkci withdraw. Namísto volání transfer, METFaucet používá funkci transferFrom v METoken a požádá METoken o převod MET na uživatele kohoutku, která požádal o výběr:

```
// Použijte funkci transferFrom METoken
METoken.transferFrom(METOwner, msg.sender, withdraw_amount);
```

A konečně, protože náš kohoutek již neodesílá ether, měli bychom pravděpodobně zabránit tomu, aby někdo poslal ether na METFaucet, protože bychom nechtěli, aby se zasekl. Změníme nouzovou platby přijímající funkci tak, abychom odmítli příchozí ether, pomocí funkce revert vrátíme všechny příchozí platby:

```
// ODMÍTNOUT jakýkoli p¬íchozí ether
function () external payable { revert(); }
```

Nyní, když je náš kód *METFaucet.sol* připraven, musíme upravit migrační skript pro nasazení kontraktu. Tento migrační skript bude o něco složitější, protože METFaucet závisí na adrese METoken. K nasazení obou kontraktů postupně použijeme JavaScript příslib. Vytvořte *2_deploy_contracts.js* takto:

```
var METoken = artifacts.require("METoken");
var METFaucet = artifacts.require("METFaucet");
var owner = web3.eth.accounts[0];

module.exports = function(deployer) {

// Nejd¬íve nasa¬te kontrakt METoken
    deployer.deploy(METoken, {from: owner}).then(function() {
// Poté nasa¬te METFaucet a p¬edejte mu adresu METokenu a
// adresu vlastníka všech MET, který schválí METFaucet
        return deployer.deploy(METFaucet, METoken.address, owner);
    });
}
```

Nyní můžeme vyzkoušet vše v příkazové řádce Truffle. Nejprve použijeme migrate k nasazení kontraktu. Když je METoken nasazený, přidělí všechny MET k účtu, který jej vytvořil, web3.eth.accounts[0]. Potom zavoláme funkci approve `METoken` pro schválení METFaucet k zasílání až 1 000 MET jménem web3.eth.accounts[0]. Nakonec vyzkoušíme kohoutek a zavoláme METFaucet.withdraw z web3.eth.accounts [1] a pokusíme se vybrat 10 MET. Zde jsou příkazy příkazové řádky:

```
<pre data-type="programlist">
$ <strong>truffle console --network ganache</strong>
truffle(ganache)&gt; <strong>migrate</strong>
Using network 'ganache'.

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ... 0x79352b43e18cc46b023a779e9a0d16b30f127bfa40266c02f9871d63c26542c7
  Migrations: 0xaa588d3737b611bafd7bd713445b314bd453a5c8
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Replacing METoken...
  ... 0xc42a57f22cddf95f6f8c19d794c8af3b2491f568b38b96fef15b13b6e8bfff21
  METoken: 0xf204a4ef082f5c04bb89f7d5e6568b796096735a
  Replacing METFaucet...
  ... 0xd9615cae2fa4f1e8a377de87f86162832cf4d31098779e6e00df1ae7f1b7f864
  METFaucet: 0x75c35c980c0d37ef46df04d31a140b65503c0eed
Saving artifacts...
truffle(ganache)&gt; <strong>METoken.deployed().then(instance =>
                  { instance.approve(METFaucet.address, 100000)
})</strong>
truffle(ganache)&gt; <strong>METoken.deployed().then(instance =>
                  {
instance.balanceOf(web3.eth.accounts[1]).then(console.log) })</strong>
truffle(ganache)&gt; <strong>BigNumber { s: 1, e: 0, c: [ 0 ] }</strong>
truffle(ganache)&gt; <strong>METFaucet.deployed().then(instance =>
                  { instance.withdraw(1000, {from:web3.eth.accounts[1]}) }
)</strong>
truffle(ganache)&gt; <strong>METoken.deployed().then(instance =>
                  {
instance.balanceOf(web3.eth.accounts[1]).then(console.log) })</strong>
truffle(ganache)&gt; <strong>BigNumber { s: 1, e: 3, c: [ 1000 ]
}</strong>
</pre>
```

Jak můžete vidět z výsledků, můžeme pomocí pracovního postupu approve & transferFrom autorizovat jeden kontrakt k převodu tokenů definovaných v jiném tokenu. Při správném použití mohou ERC20 tokeny používat EOA a další kontrakty.

Břemeno správného řízení ERC20 tokenů je však posunuto do uživatelského rozhraní. Pokud se

uživatel nesprávně pokusí přenést ERC20 tokeny na adresu kontraktu a tento kontrakt není vybaven pro příjem ERC20 tokenů, tokeny budou ztraceny. .

Problémy s ERC20 tokeny

Přijetí standardu tokenů ERC20 bylo skutečně výbušné. Byly spuštěny tisíce tokenů, jak pro experimentování s novými schopnostmi, tak pro získávání finančních prostředků v různých aukcích „skupinového financování“ a ICO. Existují však potenciální úskalí, jak jsme viděli v otázce převodu tokenů na smluvní adresy.

Jedním z méně zřejmých problémů s ERC20 tokeny je to, že odhalují jemné rozdíly mezi tokeny a samotným etherem. Pokud je ether převeden transakcí, ta má jako cíl adresu příjemce. Převod tokenů probíhá v rámci *špecifického stavu tokenového kontraku* a má tokenový kontrakt jako svůj cíl, nikoli adresu příjemce. Tokenový kontrakt sleduje zůstatky a vydává události. Při převodu tokenu není příjemci tokenu ve skutečnosti odeslána žádná transakce. Místo toho je adresa příjemce přidána do mapování v rámci samotného tokenového kontraktu. Transakce odesílající ether na adresu mění stav adresy. Transakce převádějící token na adresu mění pouze stav tokenového kontraktu, nikoli stav adresy příjemce. Dokonce ani peněženka, která podporuje ERC20 tokeny, si neuvědomuje zůstatek tokenů, pokud uživatel výslovně nepřidá konkrétní „tokenový“ kontrakt. Některé peněženky sledují nejpopulárnější tokenové kontrakty k detekci zůstatků držených adresami, které kontrolují, ale to je omezeno na malý zlomek existujícího ERC20 kontraktů.

Ve skutečnosti je nepravděpodobné, že by uživatel *chtěl* sledoval všechny zůstatky ve všech možných ERC20 tokenových kontraktech. Mnoho ERC20 tokenů je spíš jako e-mailový spam než použitelné tokeny. Automaticky vytvářejí zůstatky pro účty, které mají etherovou aktivitu, aby přilákaly uživatele. Pokud máte Ethereum adresu s dlouhou historií činnosti, zejména pokud byla vytvořena v předprodeji, najdete ji plnou „nezdravých“ tokenů, které se objevily z ničeho. Adresa samozřejmě není opravdu plná tokenů; jsou to tokenové kontrakty, které mají v sobě vaši adresu. Tyto zůstatky uvidíte pouze v případě, že tyto tokenové kontrakty sleduje průzkumník bloků nebo peněženka, kterou používáte k zobrazení vaší adresy.

Tokeny se nechovají stejně jako ether. Ether je odesílán funkcí send a přijímán jakoukoli platby přijímající funkcí v kontraktu nebo na jakékoli externě vlastněné adrese. Tokeny se odesílají pomocí funkcí transfer nebo approve & transferFrom, které existují pouze v ERC20 kontraktu, a nespouštějí (alespoň v ERC20) žádné platby přijímající funkce v kontraktu příjemce. Tokeny mají fungovat jako kryptoměna, jako je ether, ale přicházejí s určitými rozdíly, které tuto iluzi narušují.

Zvažte další problém. Chcete-li poslat ether nebo použít jakýkoliv Ethereum kontrakt, musíte etherem zaplatit za plyn. Chcete-li poslat tokeny, *potřebujete také ether*. Nemůžete zaplatit za transakční plyn tokenem a tokenový kontrakt nemůže za vás zaplatit. To se může někdy ve vzdálené budoucnosti změnit, ale mezitím to může způsobit poněkud podivné uživatelské zkušenosti. Řekněme například, že používáte burzu nebo ShapeShift k převodu nějakých bitcoinů na token. „Dostanete“ token do peněženky, která sleduje tokenový kontrakt a zobrazuje váš zůstatek. Vypadá to stejně jako všechny ostatní kryptoměny, které máte v peněžence. Zkuste však poslat token a vaše peněženka vás informuje, že k tomu potřebujete ether. Můžete být zmatení - přece jen nepotřebujete ether, abyste dostali token. Možná nemáte ether. Možná jste ani nevěděli, že token je ERC20 token na Ethereu; Možná jste si mysleli, že to byla kryptoměna s vlastní bločenkou. Iluze se právě rozplynula.

Některé z těchto problémů jsou specifické pro ERC20 tokeny. Jiné jsou obecnější problémy, které se týkají abstrakce a hranic rozhraní v rámci Etherea. Některé mohou být vyřešeny změnou tokenového rozhraní, zatímco jiné mohou potřebovat změny základních struktur v rámci Etherea (jako je rozlišení mezi EOA a kontrakty a mezi transakcemi a zprávami). Některé nemusí být „řešitelné“ přesně a mohou vyžadovat návrh uživatelského rozhraní, aby skryly nuance a zajistily konzistentnost uživatelského prostředí bez ohledu na základní rozdíly.

V dalších částech se podíváme na různé návrhy, které se snaží některé z těchto problémů řešit.

ERC223: Navrhovaný standard rozhraní tokenového kontraktu

Návrh ERC223 se pokouší vyřešit problém neúmyslného převodu tokenů na kontrakt (který může nebo nemusí podporovat tokeny) zjišťováním, zda je cílová adresa kontraktem nebo ne. ERC223 vyžaduje, aby kontrakty určené k přijetí tokenů implementovaly funkci nazvanou tokenFallback. Pokud je cílem převodu kontrakt a ten nemá podporu pro tokeny (tj. Neimplementuje tokenFallback), převod se nezdaří.

K detekci, zda je cílová adresa kontraktem, používá referenční implementace ERC223 malý segment vloženého bajtkódu poměrně kreativním způsobem:

```
function isContract(address _addr) private view returns (bool is_contract)
{
  uint length;
    assembly {
// na¬te velikost kódu na cílové adrese; to vyžaduje asembler
       length := extcodesize(_addr)
    }
    return (length>0);
}
```

Specifikace rozhraní ERC223 kontraktu je:

```
interface ERC223Token {
  uint public totalSupply;
  function balanceOf(address who) public view returns (uint);

  function name() public view returns (string _name);
  function symbol() public view returns (string _symbol);
  function decimals() public view returns (uint8 _decimals);
  function totalSupply() public view returns (uint256 _supply);

  function transfer(address to, uint value) public returns (bool ok);
  function transfer(address to, uint value, bytes data) public returns
(bool ok);
  function transfer(address to, uint value, bytes data, string
custom_fallback)
      public returns (bool ok);

  event Transfer(address indexed from, address indexed to, uint value,
                 bytes indexed data);
}
```

ERC223 není široce implementován a je debatována v ERC diskuzním [https://github.com/ethereum/EIPs/issues/223] vlákně o zpětné kompatibilitě a kompromisem mezi implementačními změnami v rozhraní kontraktu versus uživatelské rozhraní. Debata pokračuje.

ERC777: Navržený standard rozhraní smluvních tokenů

Další návrh na vylepšení standardu tokenového kontraktu je ERC777 [https://eips.ethereum.org/EIPS/eip-777]. Tento návrh má několik cílů, včetně:

- Nabídnout rozhraní kompatibilní s ERC20
- Převod tokenů pomocí funkce send, podobně jako převody etheru *Kompatibilita s ERC820 pro registraci tokenového kontraktu
- Povolit, aby kontrakty a adresy kontrolovaly, které tokeny odesílají pomocí funkce „tokensToSend“, která se volá před odesláním
- Umožnit, aby kontrakty a adresy byly upozorňovány na příjem tokenů voláním funkce `tokensReceived` v příjemci, a snížením pravděpodobnosti, že budou tokeny uzamčeny v kontraktech tím, že požaduje, aby kontrakty poskytovaly funkci `tokensReceived`
- Povolit existujícím kontraktům používat zástupné kontrakty pro funkce tokensToSend a tokensReceived
- Pracovat stejným způsobem, ať už se jedná o zaslání kontraktu nebo EOA
- Poskytovat konkrétní události pro ražbu a pálení tokenů
- Umožnit operátorům (důvěryhodným třetím stranám, které mají být ověřené kontrakty) pohybovat tokeny jménem držitele tokenů
- Poskytovat metadata o transakcích přenosu tokenů v polích userData a operatorData

Probíhající diskusi o ERC777 lze nalézt na GitHubu [https://github.com/ethereum/EIPs/issues/777].

Specifikace rozhraní kontraktu ERC777 je:

```
interface ERC777Token {
    function name() public constant returns (string);
    function symbol() public constant returns (string);
    function totalSupply() public constant returns (uint256);
    function granularity() public constant returns (uint256);
    function balanceOf(address owner) public constant returns (uint256);

    function send(address to, uint256 amount, bytes userData) public;

    function authorizeOperator(address operator) public;
    function revokeOperator(address operator) public;
    function isOperatorFor(address operator, address tokenHolder)
        public constant returns (bool);
    function operatorSend(address from, address to, uint256 amount,
                          bytes userData,bytes operatorData) public;

    event Sent(address indexed operator, address indexed from,
               address indexed to, uint256 amount, bytes userData,
               bytes operatorData);
    event Minted(address indexed operator, address indexed to,
                 uint256 amount, bytes operatorData);
    event Burned(address indexed operator, address indexed from,
                 uint256 amount, bytes userData, bytes operatorData);
    event AuthorizedOperator(address indexed operator,
                             address indexed tokenHolder);
    event RevokedOperator(address indexed operator, address indexed
tokenHolder);
}
```

ERC777 háčky

Specifikace háčku pro odesílatele tokenů ERC777 je:

```
interface ERC777TokensSender {
    function tokensToSend(address operator, address from, address to,
                          uint value, bytes userData, bytes operatorData)
public;
}
```

Implementace tohoto rozhraní je vyžadována pro jakoukoli adresu, která má být informována, zpracována nebo zabraňuje odečtu tokenů. Adresa, pro kterou kontrakt provádí toto rozhraní, musí být zaregistrována prostřednictvím ERC820, ať už kontrakt implementuje rozhraní pro sebe nebo pro jinou adresu.

Specifikace háčku příjemce tokenu ERC777 je:

```
interface ERC777TokensRecipient {
  function tokensReceived(
     address operator, address from, address to,
    uint amount, bytes userData, bytes operatorData
  ) public;
}
```

Implementace tohoto rozhraní je vyžadována pro každou adresu, která chce být informována, zpracovávat nebo odmítat příjem tokenů. Stejná logika a požadavky se vztahují na příjemce tokenů jako na rozhraní odesílatele tokenů, s dodatečným omezením, že kontrakty příjemce musí implementovat toto rozhraní, aby se zabránilo zablokování tokenů. Pokud kontrakt příjemce nezaregistruje adresu implementující toto rozhraní, přenos tokenů selže.

Důležitým aspektem je, že na jednu adresu lze zaregistrovat pouze jednoho odesílatele tokenu a jednoho příjemce tokenu. Tudíž pro každý převod ERC777 tokenů jsou volány stejné háčky funkcí před odepsáním nebo připsáním každého přesunu ERC777 tokenů. Specifický token může být v těchto funkcích identifikován pomocí odesílatele zprávy, což je konkrétní adresa tokenového kontraktu, pro zpracování konkrétního případu použití.

Na druhou stranu, mohou být stejné háčky pro odesílání tokenů a příjemce tokenů zaregistrovány pro více adres a háčky mohou rozlišit, kdo jsou odesílatelem a zamýšleným příjemcem, pomocí parametrů `from` a `` to``.

K návrhu je připojena referenční implementace [http://bit.ly/2qkAKba] ERC777. ERC777 závisí na paralelním návrhu registračního kontraktu, uvedeném v ERC820. Část debaty o ERC777 se týká složitosti přijímání dvou velkých změn najednou: nový standard tokenů a standard registrace. Diskuze pokračuje.

ERC721: Standard nezaměnitelného tokenu (vlastnická listina)

Všechny standardy tokenů, na které jsme se dosud dívali, jsou pro *zaměnitelné* tokeny, což znamená, že jednotky tokenů je navzájem zaměnitelné. Standard tokenů ERC20 sleduje pouze konečný zůstatek každého účtu a (explicitně) nesleduje původ žádného tokenu.

The Návrh ERC721 [http://bit.ly/2Ogs7Im] je standard pro *nezaměnitelné* tokeny, také známé jako *vlastnické listiny*.

Z Oxfordského slovníku:

> *vlastnická listina*: Právní dokument, který je podepsán a doručen, zejména dokument týkající se vlastnictví nebo zákonných práv.

Použití slova „vlastnická listina“ má zachycovat „vlastnictví majetku“, přestože v žádné jurisdikci nejsou dosud uznány jako „právní dokumenty“. Je pravděpodobné, že v budoucnu bude právní vlastnictví založené na digitálních podpisech na bločenkové platformě legálně uznáno.

Nezaměnitelné tokeny sledují vlastnictví jedinečné věci. Vlastněná může být digitální věc, například předmět ve hře nebo digitální sběratelská věc; nebo věcí může být fyzická věc, jejíž vlastnictví je sledováno tokenem, jako je dům, auto nebo umělecké dílo. Vlastnické listiny mohou také představovat věci se zápornou hodnotou, jako jsou půjčky (dluhy), zástavní práva, věcná břemena atd. Standard ERC721 neomezuje ani neočekává povahu věci, jejíž vlastnictví je sledováno vlastnickou listinou, a vyžaduje pouze to, aby byla jednoznačně identifikováno, což je v případě tohoto standardu dosaženo 256-bitovým pass: [identifikátorem].

Podrobnosti o standardu a diskuzi jsou sledovány na dvou různých GitHub místech:

- Počáteční návrh [https://github.com/ethereum/EIPs/issues/721]
- Pokračující diskuze [https://github.com/ethereum/EIPs/pull/841]

Abychom pochopili základní rozdíl mezi ERC20 a ERC721, stačí se podívat na strukturu interních dat použitých v ERC721:

```
// Mapování ID vlastnické listiny na majitele
mapping (uint256 => address) private deedOwner;
```

Zatímco ERC20 sleduje zůstatky, které patří každému majiteli, přičemž vlastníkem je primární klíč mapování, ERC721 sleduje každé ID vlastnické listiny a kdo je jeho vlastníkem, přičemž ID vlastnické listiny je primárním klíčem mapování. Z tohoto základního rozdílu plynou všechny vlastnosti nezaměnitelného tokenu.

Specifikace rozhraní ERC721 kontraktu je:

```
interface ERC721 /* is ERC165 */ {
    event Transfer(address indexed _from, address indexed _to, uint256
_deedId);
    event Approval(address indexed _owner, address indexed _approved,
                   uint256 _deedId);
    event ApprovalForAll(address indexed _owner, address indexed
_operator,
                         bool _approved);

    function balanceOf(address _owner) external view returns (uint256
_balance);
    function ownerOf(uint256 _deedId) external view returns (address
_owner);
    function transfer(address _to, uint256 _deedId) external payable;
    function transferFrom(address _from, address _to, uint256 _deedId)
        external payable;
    function approve(address _approved, uint256 _deedId) external payable;
    function setApprovalForAll(address _operateor, boolean _approved)
payable;
    function supportsInterface(bytes4 interfaceID) external view returns
(bool);
}
```

ERC721 také podporuje dvě *volitelná* rozhraní, jedno pro metadata a druhé pro výčet vlastnických listin a majitelů.

Volitelné rozhraní ERC721 pro metadata je:

```
interface ERC721Metadata /* is ERC721 */ {
    function name() external pure returns (string _name);
    function symbol() external pure returns (string _symbol);
    function deedUri(uint256 _deedId) external view returns (string
_deedUri);
}
```

Volitelné rozhraní ERC721 pro výčet je :

```
interface ERC721Enumerable /* is ERC721 */ {
    function totalSupply() external view returns (uint256 _count);
    function deedByIndex(uint256 _index) external view returns (uint256
_deedId);
    function countOfOwners() external view returns (uint256 _count);
    function ownerByIndex(uint256 _index) external view returns (address
_owner);
    function deedOfOwnerByIndex(address _owner, uint256 _index) external
view
        returns (uint256 _deedId);
}
```

Používání tokenového standardu

V předchozí části jsme přezkoumali několik navrhovaných standardů a několik široce nasazených standardů pro tokenové kontrakty. Co přesně tyto standardy dělají? Měli byste používat tyto standardy? Jak byste je měli používat? Měli byste přidat funkce nad rámec těchto standardů? Jaké standardy byste měli použít? Některé z těchto otázek dále prozkoumáme.

Jaké jsou tokenové standardy? Jaký je jejich účel?

Tokenové standardy jsou *minimální* specifikací pro implementaci. To znamená, že pro splnění požadavků, řekněme, ERC20, musíte minimálně implementovat funkce a chování stanovené standardem ERC20. Můžete také volně *přidat* funkcionalitu implementací funkcí, které nejsou součástí standardu.

Primárním účelem těchto standardů je podpora *schopnosti spolupráce* mezi kontrakty. Všechny

peněženky, výměny, uživatelská rozhraní a další komponenty infrastruktury se tedy mohou *propojit* (interface) předvídatelným způsobem s jakýmkoliv kontraktem, který následuje specifikaci. Jinými slovy, pokud nasadíte kontrakt, který odpovídá standardu ERC20, mohou všichni existující uživatelé peněženky hladce začít obchodovat s vaším tokenem bez jakéhokoli upgradu nebo úsilí z vaší strany.

Standardy jsou zamýšleny jako _ popisné_, nikoli *doslovné*. Jak se rozhodnete tyto funkce implementovat, záleží na vás - vnitřní fungování kontraktu není pro normu relevantní. Mají určité funkční požadavky, které řídí chování za určitých okolností, ale nepředepisují implementaci. Příkladem je chování funkce transfer, pokud je hodnota nastavena na nulu.

Měli byste používat tyto standardy?

Vzhledem ke všem těmto standardům čelí každý vývojář dilematu: používat stávající standardy nebo inovovat nad rámec omezení, která stanoví?

Toto dilema není snadné vyřešit. Standardy nutně omezují vaši schopnost inovovat vytvořením úzké „koleje“, kterou musíte dodržovat. Na druhou stranu základní standardy vyplynuly ze zkušeností se stovkami aplikací a často dobře zapadají do převážné většiny případů použití.

Součástí této úvahy je ještě větší problém: hodnota schopnosti spolupracovat a široké přijetí. Pokud se rozhodnete použít existující standard, získáte hodnotu všech systémů navržených pro práci s tímto standardem. Pokud se rozhodnete odchýlit se od standardu, musíte zvážit náklady na vybudování veškeré podpůrné infrastruktury na vlastní pěst nebo přesvědčit ostatní, aby podpořili vaši implementaci jako nový standard. Tendence k vytvoření vlastní cesty a ignorování existujících standardů je známá jako syndrom „nevymýšlel jsem já“ a je protikladnou ke kultuře otevřeného zdrojového kódu. Na druhé straně pokrok a inovace závisí na občasném odklonu od tradice. Je to složitá volba, proto ji pečlivě zvažte!

Na Wikipedii, nevymyslel jsem já“ [https://en.wikipedia.org/wiki/Not_invented_here] je postoj přijatý sociálními, podnikovými nebo institucionálními kulturami, který zabraňuje používání nebo nákupu již existujících produktů, výzkumu, standardů nebo znalostí. kvůli jejich vnějšímu původu a nákladům, jako jsou licenční poplatky.

Bezpečnost zralostí

Kromě volby standardu existuje paralelní výběr *implementace*. Pokud se rozhodnete použít normu,

jako je ERC20, musíte se poté rozhodnout, jak implementovat kompatibilní návrh. Existuje celá řada existujících „referenčních“ implementací, které jsou v ekosystému Etherea široce využívány, nebo byste mohli napsat svojí vlastní od nuly. Tato volba opět představuje dilema, které může mít vážné bezpečnostní důsledky.

Stávající implementace jsou „testovány bitvou“. I když není možné prokázat, že jsou bezpečné, mnoho z nich podporuje tokeny v hodnotě milionů dolarů. Byli napadeni opakovaně a energicky. Dosud nebyly objeveny žádné významné chyby. Vlastní psaní není snadné - existuje mnoho jemných způsobů, jak může být kontrakt ohrožen. Je mnohem bezpečnější používat osvědčenou a široce používanou implementaci. V našich příkladech jsme použili implementaci OpenZeppelin standardu ERC20, protože tato implementace je od základu zaměřena na bezpečnost.

Pokud používáte existující implementaci, můžete ji také rozšířit. S tímto impulsem však buďte opět opatrní. Složitost je nepřítelem bezpečnosti. Každý jednotlivý řádek kódu, který přidáte, rozšíří *plochu pro útok* na váš kontrakt a může v budoucnu představovat zranitelnost. Možná si nevšimnete problému, dokud do kontraktu nevložíte vysokou hodnotu a někdo kontrakt nerozbije.

Standardy a možnosti implementace jsou důležitými součástmi celkového bezpečného návrhu chytrého kontraktu ale nejsou to jediné úvahy. Viz Bezpečnost chytrých kontraktů

Rozšíření rozhraní tokenového standardu

Standardy tokenů popsané v této kapitole poskytují velmi minimální rozhraní s omezenou funkčností. Mnoho projektů vytvořilo rozšířené implementace na podporu funkcí, které potřebují pro své aplikace. Mezi tyto funkce patří:

Kontrola vlastníka

Schopnost poskytovat konkrétní adrese nebo množině adres (tj. schémata více podpisů), speciální schopnosti, jako je tvoření seznamu zakázaných a povolených adres, ražení nových tokenů, obnova ztracených tokenů, atd.

Pálení

Schopnost úmyslně zničit (“burn”) tokeny jejich převodem na adresu, ze které nejde odeslat transakci, nebo vymazáním zůstatku a snížením celkového množství tokenů.

Ražba: Schopnost přidat k celkovému množství tokenů předvídatelnou rychlostí nebo „rozhodnutím“ (fiat) tvůrce nové tokeny.

Skupinové financování

Schopnost nabízet tokeny k prodeji, například prostřednictvím aukce, tržního prodeje, reverzní aukce atd.

Zastropování: Schopnost nastavit předdefinované a neměnné limity na celkové množství tokenů (opak funkce „ražba“).

Zadní vrátka pro obnovu

Funkce pro zpětné získání finančních prostředků, zpětné převody nebo rozebrání tokenu, kterou lze aktivovat na určené adrese nebo množině adres.

Seznam povolených adres

Schopnost omezit akce (například přenos tokenů) na konkrétní adresy. Nejčastěji se používá k poskytování tokenů „akreditovaným investorům“ po prověření podle pravidel různých jurisdikcí. Obvykle existuje mechanismus pro aktualizaci seznamu povolených adres.

Seznam zakázaných adres: Schopnost omezit přenos tokenů zakázáním určitých adres. Obvykle existuje funkce pro aktualizaci černé listiny.

Pro mnoho z těchto funkcí existují referenční implementace, například v knihovně OpenZeppelin. Některé z nich jsou specifické pro konkrétní případ a jsou implementovány pouze v několika tokenech. Nyní neexistují široce přijímané standardy pro rozhraní těchto funkcí.

Jak již bylo uvedeno, rozhodnutí rozšířit tokenový standard o další funkce představuje kompromis mezi inovacemi / riziky a schopností spolupráce / zabezpečením.

Tokeny a ICO

Tokeny prošly v ekosystému Etherea výbušným vývojem. Je pravděpodobné, že se stanou velmi důležitou součástí všech platforem chytrých kontraktů, jako je Ethereum.

Důležitost a budoucí dopad těchto standardů by se však neměl zaměňovat s podporou současných tokenových nabídek. Stejně jako v jakékoli rané fázi technologie, první vlna produktů a společností téměř jistě selže, a některé selhají významně. Mnoho tokenů, které jsou dnes v Ethereu nabízeny,

jsou špatně maskované podvody, pyramidová schémata a pasti na peníze.

Trik spočívá v oddělení dlouhodobé vize a dopadu této technologie, která bude pravděpodobně obrovská, od krátkodobé bubliny ICO tokenů, které jsou plné podvodů. Standardy tokenů a platforma přežijí současnou mánii tokenů a pak pravděpodobně změní svět.

Závěry

Tokeny jsou v Ethereu velmi silným konceptem a mohou tvořit základ mnoha důležitých decentralizovaných aplikací. V této kapitole jsme se podívali na různé typy tokenů a standardů tokenů a vytvořili jste první token a související aplikaci. Vrátíme se tokenům znovu v Decentralizované aplikace (DApps), kde použijeme nezaměnitelné tokeny jako základ DApp aukce.

Orákula

V této kapitole diskutujeme *orákula*, což jsou systémy, které mohou poskytovat externí zdroje dat pro Ethereum chytré kontrakty. Termín „oracle“ (věštec) pochází z řecké mytologie, kde se odkazoval na člověka ve spojení s bohy, který viděl vize budoucnosti. V kontextu bločenek je orákulum systém, který může odpovídat na otázky, které jsou pro Ethereum vnější. Ideálně orákula jsou systémy, které jsou *důvěru nevyžadující* (trustless), což znamená, že nemusí nevyžadují důvěru, protože pracují na decentralizovaných principech.

Proč jsou potřeba orákula

Klíčovou součástí Ethereum platformy je Ethereum virtuální počítač (EVM) se schopností provádět programy a aktualizovat stav Etherea, omezený pravidly konsensu, na jakémkoli uzlu v decentralizované síti. Za účelem udržení konsensu musí být provádění EVM zcela deterministické a založené pouze na sdíleném kontextu Ethereum stavu a podepsaných transakcích. To má dva zvláště důležité důsledky: první je, že nemůže existovat žádný vlastní zdroj náhodnosti pro EVM a pro chytré kontrakty, které by ho mohly využívat; Druhým je to, že vnější data lze zavést pouze jako užitečné datové zatížení transakce.

Rozeberme tyto dva důsledky dále. Abychom pochopili zákaz skutečné náhodné funkce v EVM pro poskytování náhodnost pro chytré kontrakty, zvažte dopad na pokusy o dosažení konsensu po provedení takové funkce: uzel A by provedl příkaz a uložil 3 jménem chytrého kontraktu ve svém úložišti, zatímco uzel B provádějící stejný chytrý kontrakt by místo toho uložil 7. Uzly A a B by tedy dospěly k různým závěrům o tom, jaký by měl být výsledný stav, přestože by ve stejném kontextu běžel přesně stejný kód. Ve skutečnosti by se mohlo stát, že by se při každém vyhodnocení chytrého kontraktu dosáhlo jiného výsledného stavu. Síť s množstvím uzlů, které běží nezávisle na celém světě, by tak nikdy nemohla dospět k decentralizovanému konsensu o tom, jaký by měl být výsledný stav. V praxi by to bylo mnohem horší, než tento příklad, velmi rychle, kvůli řetězovému efektu, zahrnujícímu přesuny etheru, by exponenciálně narůstal počet možných stavů.

Všimněte si, že pseudonáhodné funkce, jako jsou kryptograficky bezpečné hašovací funkce (které jsou deterministické, a proto mohou být a skutečně jsou součástí EVM), nestačí pro mnoho aplikací. Vezměte hazardní hru, která simuluje házení mincí, abyste vyřešili výplaty sázek, musíte náhodně zvolit hlavu nebo orla - těžař může získat výhodu hraním hry a zahrnutím svých transakcí pouze do bloků, ve kterých vyhraje. Jak se tedy tento problém obejde? Všechny uzly se mohou dohodnout na obsahu podepsaných transakcí, takže lze jako datovou část transakcí odeslaných do sítě zavést vnější

informace, včetně zdrojů náhodnosti, informací o ceně, předpovědi počasí atd. Takovým datům však nelze jednoduše důvěřovat, protože pocházejí z neověřitelných zdrojů. Právě jsme problém odložili. Používáme orákula k pokusu o vyřešení těchto problémů, které budeme podrobně diskutovat ve zbývající části této kapitoly.

Případy užití orákula a příklady

Orákula, ideálně, poskytují nedůvěryhodný (nebo přinejmenším téměř nedůvěryhodný) způsob, jak získat vnější (tj. „ze skutečného světa“; nebo mimo bločenkové) informace, jako jsou výsledky fotbalových her, cena zlata nebo skutečně náhodná čísla, na Ethereum platformu pro použití chytrými kontrakty. Lze je také použít k přímému přenosu dat na rozhraní DApp. Orákula lze proto považovat za mechanismus pro překlenutí mezery mezi světem mimo bločenku a chytrým kontraktem. Umožňují chytrým kontraktům vymáhat smluvní vztahy založené na skutečných událostech a datech, čímž výrazně rozšiřují jejich rozsah. To však může také představovat vnější rizika pro Ethereum bezpečnostní model. Zvažte kontrakt „chytré závěti“, která distribuuje aktiva, když člověk zemře. To je něco, o čem se často diskutuje v oblasti chytrých kontraktů, a upozorňuje na rizika důvěru vyžadujícího orákula. Pokud je výše dědictví kontrolovaná takovýmto kontraktem dostatečně vysoká, existuje velmi vysoká motivace k hacknutí orákula a spuštění rozdělování aktiv *dříve*, než vlastník zemře.

Upozorňujeme, že některá orákula poskytují údaje, které jsou specifické pro konkrétní zdroj soukromých dat, jako jsou akademické certifikáty nebo občanské průkazy. Zdroj takových údajů, jako je univerzita nebo státní správa, jednoznačně vyžaduje důvěru a pravdivost údajů je subjektivní (pravda je určena pouze odvoláním na autoritu zdroje). Taková data proto nemohou být poskytována jako důvěru nevyžadující (trustless) - tj. bez důvěru vyžadujícího zdroje - protože neexistuje žádná samostatně ověřitelná objektivní pravda. Jako takové zahrneme tyto zdroje dat do naší definice toho, co se označuje jako orákula protože také poskytují datový most pro chytré kontrakty. Údaje, které poskytují, mají obecně podobu osvědčení, jako jsou pasy nebo záznamy o dosaženém úspěchu. Osvědčení se v budoucnu stanou velkou součástí úspěchu bločenkových platforem, zejména ve vztahu k souvisejícím otázkám ověření identity nebo pověsti, takže je důležité prozkoumat, jak mohou být zajišťovány bločenkovými platformami.

Mezi další příklady údajů, které by mohly poskytovat orkákula, patří:

- Náhodná čísla / entropie z fyzických zdrojů, jako jsou kvantové / tepelné procesy: např. spravedlivý výběr vítěze v loterijním chytrém kontraktu

- Parametrické spouštěče indexované podle přírodních rizik: např. dluhopisový chytrý kontrakt, spouštějící výplatu po nastalé katastrofě, např. zemětřesení daného stupně Richterovy škály.
- Data směnného kurzu: např. pro přesné navázání kryptoměn na fiat měnu
- Údaje o kapitálových trzích: např. cenové koše tokenizovaných aktiv / cenných papírů
- Referenční hodnoty: např. začlenění úrokových sazeb do chytrých finančních derivátů
- Statická / pseudostatická data: identifikátory cenných papírů, kódy zemí, kódy měn atd.
- Časová a intervalová data: pro spouštěče událostí spojené s přesným měřením času
- Údaje o počasí: např. výpočty pojistného na základě předpovědí počasí Politické události: pro řešení trhu předpovědí
- Sportovní události: pro řešení trhu předpovědí a kontraktů z oblasti fantasy sportů.
- Geolokační data: např. použitá při sledování dodavatelského řetězce
- Ověření poškození: u pojistných kontraktů
- Události vyskytující se na jiných bločenkách: funkce vzájemné spolupráce
- Tržní cena etheru: např. pro orákula udávající cenu plynu ve fiat měně
- Statistiky letu: např. používané skupinami a kluby pro společný nákup letenek

V následujících sekcích prozkoumáme některé způsoby, jak lze orákula implementovat, včetně základních vzorů orákul, výpočetních orákul, decentralizovaných orákul a implementace orákulového klienta v Solidity.

Návrhové vzory orákul

Všechny orákula mají podle definice několik klíčových funkcí. Mezi ně patří schopnost:

- Sbírat data ze zdroje mimo bločenku.
- Přenos dat v bločence pomocí podepsané zprávy.
- Zpřístupnění dat jejich umístěním do úložiště chytrého kontraktu.

Jakmile jsou data k dispozici v úložišti chytrých kontraktů, lze k nim přistupovat pomocí jiných chytrých kontraktů prostřednictvím zpráv, které vyvolávají funkci „načtení“ chytrého kontraktu orákula; lze k nim přistupovat také prostřednictvím Ethereum uzlů nebo klientů podporujících síť

přímo „nahlédnutím“ do úložiště orákula.

Tři hlavní způsoby, jak nastavit orákulum, lze kategorizovat jako *požadavek – odpověď, publikace-odběr* a *bezprostřední čtení*.

Počínaje nejjednoduššími, orákula s „bezprostřením čtením“ poskytují údaje, které jsou potřebné pouze pro okamžité rozhodnutí, jako „Jaká je adresa pro *ethereumbook.info*?“ nebo „Je tato osoba starší 18 let?“ Ti, kteří se chtějí dotazovat na tento druh údajů, mají tendenci tak činit na základě „právě teď“, vyhledávání se provádí, když jsou informace potřebné a možná už nikdy znovu. Mezi příklady takových orákulí patří ty, které uchovávají údaje o organizacích nebo vydaná organizacemi, jako jsou akademické certifikáty, vytáčecí kódy, institucionální členství, identifikátory letišť, identifikační průkazy, atd. Tento typ orákula ukládá data jednou do úložiště kontraktu, odkudkoli jiný chytrý kontrakt je může vyhledat pomocí požadavkového volání kontraktu orákula. Může to být změněno. Data v úložišti orákula jsou také k dispozici pro přímé vyhledávání pomocí bločenkových aplikací (tj. Ethereum připojených klientů), aniž by bylo nutné být průkopníkem a vynaložit náklady na plyn při provedení transakce. Obchod, který chce zkontrolovat věk zákazníka, který si chce koupit alkohol, by mohl použít orákulum tímto způsobem. Tento typ věštce je atraktivní pro organizaci nebo společnost, která by jinak musela provozovat a udržovat servery, aby odpověděla na takové požadavky na data. Všimněte si, že data uložená orákulem pravděpodobně nebudou prvotní data, která orákulum poskytuje, např. z důvodů efektivity nebo ochrany soukromí. Univerzita by mohla zřídit orákulum pro osvědčení o akademickém úspěchu minulých studentů. Ukládání všech podrobností o osvědčeních (které by mohly odkazovat na stránky absolvovaných kurzů a dosažených známek) by však bylo příliš náročné. Místo toho stačí haš certifikátu. Stejně tak by vláda mohla chtít umístit občanské průkazy na platformu Ethereum, přičemž je třeba zachovat soukromí údajů. Opět by hašování dat (pečlivěji v Merkle stromech se solemi) a ukládání kořenového haše do úložiště chytrých kontraktů bylo účinným způsobem organizace takové služby.

Další nastavení je *publikace – odběr*, kde orákulum, který efektivně poskytuje službu vysílání dat, u nichž se očekává, že změna (možná jak pravidelná, tak i častá) je vyvolávána chytrým kontraktem z bločenky, nebo aktualizačním démonem mimo bločenku. Tato kategorie má podobnou strukturu jako kanály RSS, WebSub a podobně, kde je orákulum aktualizováno novými informacemi a signalizuje, že nová data jsou dostupná těm, kteří se považují za „přihlášené“. Zainteresované strany musí orákulum vyzvat, aby zkontrolovalo, zda se změnily nejnovější informace, nebo poslouchat aktualizace kontraktu orákula a jednat, když k nim dojde. Příkladem jsou cenové zdroje, informace o počasí, hospodářské nebo sociální statistiky, provozní údaje atd. Dotazování na změnu je ve světě webových serverů velmi neefektivní, ale ne v peer-to-peer kontextu bločenkových platforem: Ethereum klienti musí držet krok se všemi změnami stavu, včetně změn úložiště kontraktu, takže

dotazování na změny dat je místní volání synchronizovaného klienta. Protokoly Ethereum událostí usnadňují aplikacím vyhledávat aktualizace orákula, takže tento vzor lze v některých ohledech dokonce považovat za službu „tlačit“ (push). Pokud se však dotazování provádí z chytrého kontraktu, který může být nezbytný pro některé decentralizované aplikace (např. tam, kde aktivační pobídky nejsou možné), mohou vzniknout značné výdaje na plyn.

událostí Kategorie *požadavek-odpověď* je nejsložitější: v tomto případě je datový prostor příliš velký na to, aby byl uložen v chytrým kontraktu a od uživatelů se očekává, že budou současně potřebovat pouze malou část celkového souboru dat. Je to také vhodný model pro podniky poskytující údaje. Prakticky by takové orákulum mohlo být implementován jako systém chytrýho kontraktů na dálku a infrastruktury mimo bločenku používané k monitorování požadavků a získávání a vracení dat. Požadavek na data z decentralizované aplikace by obvykle byl asynchronní proces zahrnující řadu kroků. V tomto vzoru nejprve EOA zašle transakci decentralizované aplikaci, což má za následek interakci s funkcí definovanou v chytrém kontraktu orákula. Tato funkce iniciuje požadavek na orákulum, s přidruženými parametry upřesňujícími požadovaná data a doplňujícími informacemi, které mohou zahrnovat funkce zpětného volání a parametry plánování. Jakmile je tato transakce potvrzena, lze žádost orákulu považovat za událost EVM vycházející z kontraktu orákula nebo jako změnu stavu; parametry lze načíst a použít k provedení skutečného dotazu zdroje dat mimo bločenku. Orákulum může také vyžadovat platbu za zpracování žádosti, platbu plynu za zpětné volání a oprávnění k přístupu k požadovaným datům. Nakonec jsou výsledná data podepsána vlastníkem orákula, což potvrzuje platnost dat v daném čase, a jsou dodávány v transakci decentralizované aplikaci, která žádost podala - buď přímo, nebo prostřednictvím orákulum kontraktu. V závislosti na parametrech plánování může orákulum vysílat další transakce aktualizující data v pravidelných intervalech (např. Informace o cenách na konci dne).

Kroky orákula typu požadavek-odpověď lze shrnout takto:

1. Přijme dotaz od DApp.
2. Analyzuje dotaz.
3. Zkontroluje platbu a zda jsou k dispozici oprávnění k přístupu k datům.
4. Načtě relevantní data ze zdroje mimo bločenku (a v případě potřeby je zašifrujte).
5. Podepíše transakci(transakcee) se zahrnutými údaji.
6. Odešle transakci do sítě
7. Naplánuje další nezbytné transakce, jako jsou oznámení atd.

Je také možná řada dalších schémat; například data mohou být vyžádána a vrácena přímo EOA, čímž se odstraní potřeba kontraktu orákula. Podobně by mohl být požadavek a odpověď podán z a na hardwarového senzoru umožňujícího internet věcí. Proto může být orákulum člověk, software nebo hardware.

Zde popsaný vzor požadavek-odpověď je běžně vidět v architekturách typu klient-server. I když se jedná o užitečný vzor zasílání zpráv, který umožňuje aplikacím vést obousměrnou konverzaci, za určitých podmínek je to pravděpodobně nevhodné. Například chytrý dluhopis vyžadující úrokovou sazbu od orákula může vyžadovat data denně podle vzoru požadavek-odpověď, aby se zajistilo, že sazba je vždy správná. Vzhledem k tomu, že úrokové sazby se mění jen zřídka, může být vhodnější zde vzor publikace-odběr - zejména při zohlednění omezené velikosti Ethereum bloků.

Publikace - odběr je vzor, kdy vydavatelé (v tomto kontextu orákula) neposílají zprávy přímo příjemcům, ale místo toho rozdělují publikované zprávy do různých tříd. Odběratelé jsou schopni vyjádřit zájem o jednu nebo více tříd a načíst pouze ty zprávy, které jsou předmětem jejich zájmu. Podle takového vzoru by orákulum mohla zapsat úrokovou sazbu do svého vlastního interního úložiště pokaždé, když se změní. Více přihlášených DApps je jednoduše přečte z orákulum kontraktu, čímž se sníží dopad na obsazenost bloků a minimalizují se náklady na ukládání dat.

Ve vzoru vysílání nebo vícesměrného vysílání, orákulum odesílá všechny zprávy na kanál a odběratelské kontrakty by poslouchaly kanál v různých režimech předplatného. Například orákulum může zveřejňovat zprávy na kanálu směnného kurzu. Odběratelský chytrý kontrakt by mohl požadovat plný obsah kanálu, pokud by vyžadovalo časové řady např. pro výpočet klouzavého průměru; jiný by mohl vyžadovat pouze nejnovější sazbu pro výpočet okamžité ceny. Vzor vysílání je vhodný, pokud orákulum nemusí znát totožnost předplatného kontraktu.

Ověření dat

Pokud předpokládáme, že zdroj dat, která jsou dotazována pomocí DApp, je autoritativní a důvěryhodný (není to nevýznamný předpoklad), zůstává nevyřešená otázka: vzhledem k tomu, že orákulum a mechanismus reakce na žádost mohou být provozovány různými entitami, jak můžeme tomuto mechanismu důvěřovat? Existuje zřetelná možnost, že s daty může být manipulováno během přenosu, takže je důležité, aby metody mimo bločenku dokázaly integritu vrácených dat. Dva běžné přístupy k autentizaci dat jsou *důkazy pravosti* a *důvěryhodná prováděcí prostředí* (TEE).

Důkazy pravosti jsou kryptografické záruky, že s údaji nebylo manipulováno. Na základě různých technik osvědčování (např. digitálně podepsaných důkazů) účinně přesouvají důvěru z datového

nosiče na atestátora (tj. poskytovatele osvědčení). Ověření důkazem pravosti v bločence umožňuje chytrým kontraktům ověřit integritu dat, před provedením operace na těchto datech. Oraclize [http://www.oraclize.it/] je příkladem služby orákula, která využívá řadu důkazů pravosti. Jeden takový důkaz, který je v současné době k dispozici pro datové dotazy z hlavní sítě Ethereum, je TLSNotary důkaz. TLSNotary důkazy umožňují klientovi poskytnout třetí straně důkaz o tom, že mezi klientem a serverem došlo k HTTPS webovému provozu. Zatímco HTTPS je sám o sobě bezpečný, nepodporuje podepisování dat. Důkazy TLSNotary se proto spoléhají na podpisy TLSNotary (prostřednictvím PageSigner). Protokoly TLSNotary využívají protokol Transport Layer Security (TLS), což umožňuje hlavní klíč TLS, který podepisuje data po přístupu, rozdělit mezi tři strany: server (orákulum), auditovaný subjekt (oraclize) a auditor. Společnost Oraclize používá jako auditora instanci virtuálního počítače Amazon Web Services (AWS), kterou lze ověřit jako nezměněnou od okamžiku vytvoření. Tato instance AWS ukládá tajemství TLSNotary, což mu umožňuje poskytovat čestné důkazy. Přestože nabízí vyšší záruky proti manipulaci s daty, než je mechanismus čistého požadavku a odpovědi, vyžaduje tento přístup předpoklad, že Amazon sám nebude manipulovat s instancí VM.

Town Crier [http://www.town-crier.org/] orákulum pro autentizované datové zdroje založený na přístupu TEE; takové metody využívají zabezpečené enklávy založené na hardwaru k zajištění integrity dat. Town Crier používá Intel Software Guard eXtensions (SGX) k zajištění toho, aby odpovědi z dotazů HTTPS mohly být ověřeny jako autentické. SGX poskytuje záruky integrity, čímž zajišťuje, že aplikace spuštěné v enklávě jsou chráněny CPU proti neoprávněné manipulaci jiným procesem. Poskytuje také důvěrnost a zajišťuje, že stav aplikace je neprůhledný vůči jiným procesům při běhu v enklávě. A konečně, SGX umožňuje atestaci tím, že vytváří digitálně podepsaný důkaz, že aplikace - bezpečně identifikovaná hašem jeho sestavení - skutečně běží v enklávě. Ověřením tohoto digitálního podpisu je možné, aby decentralizovaná aplikace dokázala, že instance Town Crier běží bezpečně v enklávě SGX. To zase dokazuje, že s instancí nebylo manipulováno a že data odesílaná Town Crier jsou tedy autentická. Vlastnost důvěrnosti dále umožňuje Town Crieru zpracovávat soukromá data tím, že umožňuje šifrování datových dotazů pomocí veřejného klíče instance Town Crier. Provozování mechanismu dotazů a odpovědí orákula v enklávě, jako je SGX, nám efektivně umožňuje myslet na to, že běží bezpečně na důvěryhodném hardwaru třetích stran a zajišťuje, že požadovaná data jsou vrácena nezfalšována (za předpokladu, že věříme Intel / SGX).

Výpočetní orákula

Dosud jsme diskutovali pouze orákula v souvislosti s vyžádáním a doručením dat. Orákula však lze také použít k provádění libovolného výpočtu, což je funkce, která může být zvláště užitečná vzhledem k omezení na maximální množství plynu, které můžou spotřebovat transakce v rámci

jednoho Ethereum bloku, a poměrně vysokým nákladům výpočet v růmci EVM. Namísto pouhého předávání výsledků dotazu lze výpočetní orákulum použít k provedení výpočtu na sadě vstupů a vrátit vypočtený výsledek, který by nebylo možné na bločence spočítat. Například je možné použít výpočetní orákulum pro výpočet výpočetně náročného regresního výpočtu za účelem odhadu výnosu z dluhopisové smlouvy.

IPokud jste ochotni důvěřovat centralizované, ale auditovatelné službě, můžete znovu jít do Oraclize. Poskytují službu, která umožňuje decentralizovaným aplikacím požadovat výstup výpočtu provedeného ve virtuálním počítači AWS v izolovaném prostředí. Instance AWS vytvoří spustitelný kontejner z uživatelsky konfigurovaného Dockerfile zabaleného v archivu, který je nahrán do Meziplanetárního systému souborů (IPFS; viz Úložiště dat). Na požádání Oraclize načte tento archiv pomocí haše a poté inicializuje a spustí kontejner Docker na AWS a předá veškeré argumenty, které jsou aplikaci poskytnuty jako proměnné prostředí. Kontejnerizovaná aplikace provede výpočet s časovým omezením a zapíše výsledek na standardní výstup, kde jej lze získat pomocí Oraclize a vrátit se do decentralizované aplikace. Oraclize aktuálně nabízí tuto službu v auditovatelné instanci t2.micro AWS, takže pokud má výpočet nějakou netriviální hodnotu, je možné zkontrolovat, zda byl proveden správný kontejner Docker. Nejedná se však o skutečně decentralizované řešení.

Koncept „cryptlet“ jako standardu pro ověřitelné pravdy orákul byl formalizován jako součást širšího rámce ESC společnosti Microsoft. Cryptlety se provádějí v zašifrované kapsli, která odtrhuje infrastrukturu, jako je I / O, a má připojený CryptoDelegate, takže příchozí a odchozí zprávy jsou automaticky podepsány, ověřovány a osvědčené. Cryptlety podporují distribuované transakce, takže logika kontraktu může ACID provádět komplexní vícekrokové, vícebločenkové a externí systémové transakce ACID způsobem. To umožňuje vývojářům vytvářet přenosná, izolovaná a soukromá řešení pravdy pro použití v chytrých kontraktech. Cryptlety mají následující formát:

```
public class SampleContractCryptlet : Cryptlet
  {
        public SampleContractCryptlet(Guid id, Guid bindingId, string
name,
            string address, IContainerServices hostContainer, bool
contract)
            : base(id, bindingId, name, address, hostContainer, contract)
        {
            MessageApi = new CryptletMessageApi(GetType().FullName,
                new SampleContractConstructor())
```

V případě decentralizovanějšího řešení se můžeme obrátit na TrueBit [https://truebit.io/], které nabízí řešení pro škálovatelný a ověřitelný výpočet mimo bločenku. Používají systém řešitelů a ověřovatelů, kteří jsou motivováni k provádění výpočtů a ověřování těchto výpočtů. Pokud je řešení napadeno, provádí se opakovací proces ověřování na podmnožinách výpočtu - druh „ověřovací hry“. Hra pokračuje řadou kol, z nichž každé rekurzivně kontroluje menší a menší podmnožinu výpočtu. Hra nakonec dorazí do finálového kola, kde je výzva natolik triviální, že rozhodčí - Ethereum těžaři - mohou rozhodnout o tom, zda byla výzva splněna, na bločence. Ve skutečnosti je TrueBit implementací výpočetního trhu, který umožňuje decentralizovaným aplikacím platit za ověřitelné výpočty, které mají být prováděny mimo bločenku, ale spoléhat se na Ethereum, aby prosadil pravidla ověřovací hry. Teoreticky to umožňuje důvěryhodným chytrým kontraktům bezpečně provádět jakýkoli výpočetní úkol.

Pro systémy, jako je TrueBit, existuje celá řada aplikací, od strojového učení až po ověření důkazu prací. Příkladem toho je most Doge – Ethereum, který používá TrueBit k ověření Dogecoinového důkazu práci (Scrypt), což je paměťově náročná a výpočetně náročná funkce, kterou nelze vypočítat v rámci limitu plynu bloku Etherea. Provedením tohoto ověření na TrueBit bylo možné bezpečně ověřit transakce dogecoinů v rámci chytrého kontraktu na testovací síti Ethereum Rinkeby.

Decentralizovaná orákula

Zatímco centralizovaná datová nebo výpočetní orákula postačují pro mnoho aplikací, představují jediné body selhání v síti Ethereum. V souvislosti s myšlenkou decentralizovaných orákulí jako prostředku zajištění dostupnosti dat a vytvoření sítě jednotlivých poskytovatelů údajů se systémem agregace údajů v bločence bylo navrženo několik systémů.

ChainLink [https://www.smartcontract.com/link] navrhl decentralizovanou síť orákulí skládající se ze klíčových chytrých kontraktů: reputarční kontrakt, kontrakt párování objednávek a agregační kontrakt a —A mimo bločenkový registr poskytovatelů dat. Reputační kontrakt se používá ke sledování výkonnosti poskytovatelů dat. Skóre v reputačním kontraktu se používá k naplnění registru mimo bločenku. Kontrakt párování objednávek vybírá nabídky orákulí při zohlednění reputace kontraktu. Následně dokončí dohodu o úrovni služeb, která zahrnuje parametry dotazu a požadovaný počet orákulí. To znamená, že kupující nemusí obchodovat přímo s jednotlivými orákuly. Agregační kontrakt shromažďuje odpovědi (předložené pomocí systému odevzdání a odhalení) z několika orákulí, vypočítá konečný kolektivní výsledek dotazu a nakonec výsledky vrátí zpět do reputačního kontraktu.

Jednou z hlavních výzev takového decentralizovaného přístupu je formulace agregační funkce. ChainLink navrhuje vypočítat váženou odpověď, což umožňuje vykazovat skóre platnosti pro každou odpověď orákula. Detekce „neplatného“ skóre je zde nesnadná, protože se spoléhá na předpoklad, že okrajové datové body, měřené odchylkami od odpovědí poskytnutých kolegy, jsou nesprávné. Při výpočtu skóre platnosti na základě umístění odpovědi orákula mezi distribucí odpovědí hrozí penalizace správných odpovědí oproti průměrným. Proto ChainLink nabízí standardní sadu agregačních smluv, ale také umožňuje specifikovat přizpůsobené agregační smlouvy.

Příbuznou myšlenkou je protokol SchellingCoin. Zde uvádí několik účastníků hodnoty a medián je považován za „správnou“ odpověď. Účastníci jsou povinni poskytnout vklad, který je přerozdělován ve prospěch hodnot, které jsou blíže střední hodnotě, a proto stimuluje vykazování hodnot, které jsou podobné ostatním. Očekává se, že společná hodnota, známá také jako Schellingův bod, který by respondenti mohli považovat za přirozený a zřejmý cíl, kolem kterého se koordinovat, bude blízká skutečné hodnotě.

Jason Teutsch z TrueBit nedávno představil nový návrh pro decentralizované orákulum dostupnosti dat mimo bločenku. Tento návrh využívá specializovanou bločenku využívající důkaz prací, která je schopna správně informovat o tom, zda jsou registrovaná data k dispozici během dané epochy. Těžaři se pokoušejí stahovat, ukládat a šířit všechna aktuálně registrovaná data, čímž je zaručena lokální dostupnost dat. I když je takový systém nákladný v tom smyslu, že každý těžební uzel ukládá a šíří všechna zaregistrovaná data, systém umožňuje opětovné použití úložiště uvolněním dat po skončení registračního období.

Rozhraní orákulového klienta v Solidity

Pomocí Oraclize aktualizujte směnný kurz ETH / USD z externího zdroje je příklad v Solidity, který ukazuje, jak lze Oraclize použít k nepřetržitému dotazování na cenu ETH / USD z API a výsledek uložit použitelným způsobem.

Example 22. Pomocí Oraclize aktualizujte směnný kurz ETH / USD z externího zdroje

```
/*
Cenový ticker ETH / USD využívající API CryptoCompare

Tento kontrakt udržuje v pam¬ti aktualizovanou cenu ETH / USD,
která je aktualizována každých 10 minut.
 */

pragma solidity ^0.4.1;
import "github.com/oraclize/ethereum-api/oraclizeAPI.sol";

/*
   P¬ipravené metody „oraclize_“ zna¬í d¬di¬nost z „usingOraclize“
 */
contract EthUsdPriceTicker is usingOraclize {

    uint public ethUsd;

    event newOraclizeQuery(string description);
    event newCallbackResult(string result);

    function EthUsdPriceTicker() payable {
        // signalizuje generování d¬kaz¬ TLSN jejich ukládání na IPFS
        oraclize_setProof(proofType_TLSNotary | proofStorage_IPFS);

        // požadovaný dotaz
        queryTicker();
    }

    function __callback(bytes32 _queryId, string _result, bytes _proof)
public {
        if (msg.sender != oraclize_cbAddress()) throw;
        newCallbackResult(_result);

        /*
         * Z výsledného ¬et¬zce získá celé ¬íslo bez znaménka pro
použití na blo¬ence.
         * Použije zd¬d¬ného pomocníka "parseInt" z "usingOraclize",
umož¬ujícího
         * výsledný ¬et¬zec "123.45" p¬evést na uint 12345.
```

```
         */
        ethUsd = parseInt(_result, 2);

        // zavoláno ze zp¬tného volání, protože se dotazujeme na cenu
        queryTicker();
    }

    function queryTicker() external payable {
        if (oraclize_getPrice("URL") > this.balance) {
            newOraclizeQuery("Oraclize query was NOT sent, please add
some ETH
                to cover for the query fee");
        } else {
            newOraclizeQuery("Oraclize query was sent, standing by for
the
                answer...");

            // parametry dotazu jsou (spožd¬ní v sekundách,  typ
datového zdroje,
            // parametr datového zdroje)
            // udává JSONPath k vyzvednutí ur¬ené ¬ástí výsledku JSON
API
            oraclize_query(60 * 10, "URL",
                "json(https://min-api.cryptocompare.com/data/price?\
                fsym=ETH&tsyms=USD,EUR,GBP).USD");
        }
    }
}
```

Pro integraci s Oraclize musí být kontrakt EthUsdPriceTicker dítětem `usingOraclize`; kontrakt usingOraclize je definován v souboru *oraclizeAPI*. Požadavek na data se provádí pomocí funkce oraclize_query, která je zděděna ze kontraktu usingOraclize. Toto je přetížená funkce, která očekává alespoň dva argumenty:

- Podporovaný zdroj dat, který se má použít, například URL, WolframAlpha, IPFS nebo výpočet
- Argument pro daný zdroj dat, který může zahrnovat použití pomocníků pro analýzu JSON nebo XML

Cenový dotaz se provádí ve funkci queryTicker. Za účelem provedení dotazu Oraclize vyžaduje

zaplacení malého poplatku v etheru, pokrývající náklady na plyn za zpracování výsledku a přenos do funkce zpětného volání __callback a doprovodný příplatek za službu. Tato částka závisí na zdroji dat a, je-li to stanoveno, na typu důkazu o pravosti, který je vyžadován. Jakmile jsou data načtena, je funkce __callback vyvolána účtem řízeným Oraclize, který má oprávnění provádět zpětné volání; předá hodnotu odpovědi a jedinečný argument queryId, který lze například použít ke zpracování a sledování více čekajících zpětných volání z Oraclize.

Poskytovatel finančních údajů Thomson Reuters rovněž poskytuje službu orákula pro Ethereum nazvanou BlockOne IQ, která umožňuje požadovat tržní a referenční údaje prostřednictvím chytrých kontraktů provozovaných v soukromých nebo povolených sítích. Kontrakt volající službu BlockOne IQ pro tržní data zobrazuje rozhraní pro orákulum a klientský kontrakt, která žádost podá.

Example 23. Kontrakt volající službu BlockOne IQ pro tržní data

```
pragma solidity ^0.4.11;

contract Oracle {
    uint256 public divisor;
    function initRequest(
       uint256 queryType, function(uint256) external onSuccess,
       function(uint256
    ) external onFailure) public returns (uint256 id);
    function addArgumentToRequestUint(uint256 id, bytes32 name, uint256
arg) public;
    function addArgumentToRequestString(uint256 id, bytes32 name,
bytes32 arg)
        public;
    function executeRequest(uint256 id) public;
    function getResponseUint(uint256 id, bytes32 name) public constant
        returns(uint256);
    function getResponseString(uint256 id, bytes32 name) public
constant
        returns(bytes32);
    function getResponseError(uint256 id) public constant
returns(bytes32);
    function deleteResponse(uint256 id) public constant;
}

contract OracleB1IQClient {

    Oracle private oracle;
    event LogError(bytes32 description);

    function OracleB1IQClient(address addr) external payable {
        oracle = Oracle(addr);
        getIntraday("IBM", now);
    }

    function getIntraday(bytes32 ric, uint256 timestamp) public {
        uint256 id = oracle.initRequest(0, this.handleSuccess,
this.handleFailure);
        oracle.addArgumentToRequestString(id, "symbol", ric);
        oracle.addArgumentToRequestUint(id, "timestamp", timestamp);
```

```
        oracle.executeRequest(id);
    }

    function handleSuccess(uint256 id) public {
        assert(msg.sender == address(oracle));
        bytes32 ric = oracle.getResponseString(id, "symbol");
        uint256 open = oracle.getResponseUint(id, "open");
        uint256 high = oracle.getResponseUint(id, "high");
        uint256 low = oracle.getResponseUint(id, "low");
        uint256 close = oracle.getResponseUint(id, "close");
        uint256 bid = oracle.getResponseUint(id, "bid");
        uint256 ask = oracle.getResponseUint(id, "ask");
        uint256 timestamp = oracle.getResponseUint(id, "timestamp");
        oracle.deleteResponse(id);
        // Ud¬lejte n¬co s údaji o cen¬
    }

    function handleFailure(uint256 id) public {
        assert(msg.sender == address(oracle));
        bytes32 error = oracle.getResponseError(id);
        oracle.deleteResponse(id);
        emit LogError(error);
    }

}
```

Požadavek na data je iniciován pomocí funkce initRequest, která umožňuje specifikovat typ dotazu (v tomto příkladu požadavek na vnitrodenní cenu), navíc dvě funkce zpětného volání. Vrací uint256 identifikátor, který lze použít k poskytnutí dalších parametrů. Funkce addArgumentToRequestString se používá k určení kódu nástroje Reuters (RIC), zde pro akcie IBM a `addArgumentToRequestUint` umožňuje zadat časové razítko. Nyní předání aliasu pro block.timestamp načte aktuální cenu pro IBM. Požadavek je poté proveden funkcí executeRequest. Jakmile je žádost zpracována, kontrakt orákula vyvolá funkci onSuccess zpětného volání s identifikátorem dotazu, což umožní získat výsledná data; v případě selhání vyhledávání, `onFailure` zpětné volání vrátí chybový kód. Mezi dostupná pole, která lze získat při úspěchu, patří otevírací (open), maximální (high), minimální (low), zavírací OHLC (close) a nabídková/poptávková (bid/ask) cena.

Závěry

Jak vidíte, orákula poskytují klíčovou službu chytrým kontraktům: orovádění kontraktů obohacují o externí data. S tím také samozřejmě představují značné riziko - pokud jsou důvěru vyžadujícími zdroji a mohou být ohroženy, mohou vést k ohroženému plnění chytrého kontraktu, kterému dodávají data.

Obecně platí, že při zvažování použití orákula buďte velmi opatrní ohledně modelu důvěry. Pokud se domníváte, že orákulum může být důvěru vyžadující, můžete narušit bezpečnost vašeho chytrého kontraktu tím, že ji vystavíte potenciálně falešným vstupům. To znamená, že orákula mohou být velmi užitečná, pokud budou pečlivě zváženy předpoklady zabezpečení.

Decentralizovaná orákula mohou některé z těchto problémů vyřešit a nabídnout Ethereum chytrým kontraktům zdroj důvěru nevyžadujících externích dat. Vyberte si pečlivě a můžete začít zkoumat most mezi Ethereem a „skutečným světem“, který nabízejí orákula.

Decentralizované aplikace (DApps)

V této kapitole prozkoumáme svět *decentralizovaných aplikací* nebo *DApps*. Od počátků Etherea byla vize zakladatelů mnohem širší než „chytré kontrakty“: ne méně než znovuobjevení webu a vytvoření nového světa DApps, vhodně nazývaného *web3*. Chytré kontrakty jsou způsob, jak decentralizovat řídicí logiku a platební funkce aplikací. Web3 DApps jsou o decentralizaci všech ostatních aspektů aplikace: úložiště, zasílání zpráv, pojmenování atd. (Viz Web3: Decentralizovaný web využívající chytré kontrakty a technologie P2P).

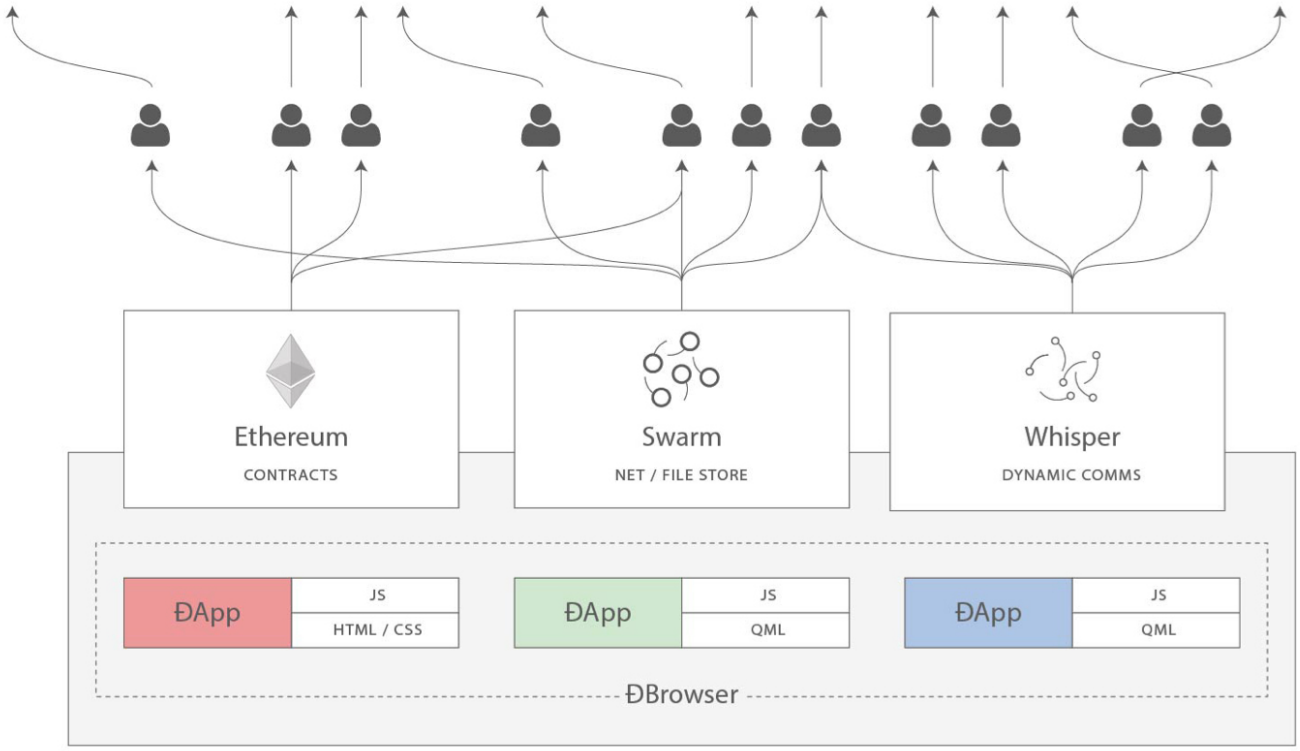

Figure 41. Web3: Decentralizovaný web využívající chytré kontrakty a technologie P2P

Zatímco „decentralizované aplikace“ jsou odvážnou vizí budoucnosti, termín „DApp“ je často používán pro každý chytrý kontrakt s webovým rozhraním. Některé z těchto tzv. DApps jsou vysoce centralizované aplikace (CApps?). Dejte si pozor na falešné DApps!

V této kapitole budeme vyvíjet a zavádět ukázkovou aukční platformu DApp. Zdrojový kód najdete v

úložišti knihy na adrese *code/auction_dapp* [http://bit.ly/2DcmjyA]. Podíváme se na každý aspekt aukční aplikace a uvidíme, jak můžeme aplikaci co nejvíce decentralizovat. Nejprve se však podrobněji podíváme na definující vlastnosti a výhody DApps.

Co je to DApp?

DApp je aplikace, která je většinou nebo zcela decentralizovaná.

Zvažte všechny možné aspekty aplikace, která můžou být decentralizované:

- Backend software (aplikační logika)
- Frontend software (uživatelské rozhraní)
- Datové úložiště
- Komunikace zpráv
- Rozlišení názvu

Každý z nich může být poněkud centralizovaný nebo trochu decentralizovaný. Například frontend může být vyvinut jako webová aplikace, která běží na centralizovaném serveru nebo jako mobilní aplikace, která běží na vašem zařízení. Backend a úložiště mohou být na soukromých serverech a proprietárních databázích, nebo můžete použít chytrý kontrakt a úložiště P2P.

Při vytváření DApp existuje mnoho výhod, které typická centralizovaná architektura nemůže poskytnout:

Pružnost

Protože obchodní logika je řízena chytrým kontraktem, backend DApp bude plně distribuován a spravován na bločenkové platformě. Na rozdíl od aplikace nasazené na centralizovaném serveru nebude mít DApp žádné prostoje a bude nadále k dispozici, dokud bude platforma stále funkční.

Transparentnost

Charakter bločenkové DApp umožňuje každému nahlédnout do kódu a být si jistější jeho funkcí. Jakákoli interakce s DApp bude uložena navždy v bločence

Odolnost proti cenzuře

> Dokud má uživatel přístup k Ethereum uzlu (v případě potřeby jeden spustí), bude vždy schopen interagovat s DApp bez rušení centralizovanou kontrolou. Žádný poskytovatel služeb nebo dokonce vlastník chytrého kontraktu nemůže změnit kód, jakmile je nasazen v síti.

V současném Ethereum ekosystému dnes existuje jen velmi málo skutečně decentralizovaných aplikací - většina z nich se na svou část provozu stále spoléhá na centralizované služby a servery. V budoucnu očekáváme, že bude možné, aby každá část jakéhokoli DApp byla provozována plně decentralizovaným způsobem.

Backend (chytrý kontrakt)

V DApp se chytré kontrakty používají k ukládání obchodní logiky (kód programu) a související stav vaší aplikace. Můžete si představit chytrý kontrakt, která nahradí komponentu na straně serveru (aka „backend“) v běžné aplikaci. Toto je samozřejmě přílišné zjednodušení. Jedním z hlavních rozdílů je to, že jakýkoli výpočet provedený v chytrém kontraktu je velmi nákladný, a proto by měl být co nejmenší. Je proto důležité určit, které aspekty aplikace potřebují důvěryhodnou a decentralizovanou prováděcí platformu.

Ethereum chytré kontrakty vám umožňují stavět architektury, ve kterých síť chytrých kontraktů se mezi sebou volá a předává si data, čte a zapisuje své vlastní stavové proměnné, jak jsou v provozu, přičemž jejich složitost je omezena pouze limitem plynu v bloku. Po nasazení chytrého kontraktu by vaši obchodní logiku mohlo v budoucnu dobře využít mnoho dalších vývojářů.

Jedním z hlavních aspektů návrhu architektury chytrého kontraktu je nemožnost změnit kód chytrého kontraktu po jeho nasazení. Může být vymazán, pokud je programován s přístupnou instrukcí SELFDESTRUCT , ale kromě úplného odstranění nelze kód nijak změnit.

Druhým hlavním aspektem návrhu architektury chytrého kontraktu je velikost DApp. Skutečně velký monolitický chytrý kontrakt může stát hodně plynu při jeho zavedení a použití. Některé aplikace se proto mohou rozhodnout pro mimo bločenkový výpočet a externí zdroj dat. Mějte však na paměti, že základní obchodní logika DApp závisející na externích datech (např. z centralizovaného serveru) znamená, že vaši uživatelé budou muset těmto externím zdrojům důvěřovat.

Frontend (webové uživatelské rozhraní)

Na rozdíl od obchodní logiky DApp, která vyžaduje, aby vývojář porozuměl EVM a novým jazykům, jako je Solidity, může rozhraní DApp na straně klienta používat standardní webové technologie (HTML, CSS, JavaScript atd.). To umožňuje tradičnímu vývojáři webu používat známé nástroje, knihovny a rámce. Interakce s Ethereem, jako je podepisování zpráv, odesílání transakcí a správa klíčů, jsou často prováděny prostřednictvím rozšíření webového prohlížeče, jako je MetaMask (viz Základy Etherea).

I když je možné vytvořit také mobilní DApp, v současné době existuje jen málo zdrojů, které by pomohly vytvořit mobilní rozhraní DApp, zejména kvůli nedostatku mobilních klientů, kteří mohou sloužit jako odlehčený klient s funkcemi správy klíčů.

Frontend je obvykle propojen s programem Ethereum prostřednictvím knihovny JavaScript *web3.js*, která je spojena s prostředky frontendu a je obsluhována v prohlížeči webovým serverem.

Úložiště dat

Vzhledem k vysokým nákladům na plyn a aktuálně nízkému limitu plynu v bloku nejsou chytré kontrakty vhodné pro ukládání nebo zpracování velkého množství dat. Proto většina DApps využívá služby ukládání dat mimo bločenku, což znamená, že ukládají objemná data mimo Ethereum bločenku, na platformě pro ukládání dat. Tato platforma pro ukládání dat může být centralizována (například typická cloudová databáze) nebo mohou být data decentralizována, ukládána na platformě P2P, jako je IPFS nebo na vlastní platformě Ethereum Swarm.

Decentralizované úložiště P2P je ideální pro ukládání a distribuci velkých statických dat, jako jsou obrázky, videa a zdroje rozhraní frontend webového rozhraní (HTML, CSS, JavaScript atd.). Podíváme se na několik dalších možností.

IPFS

Celoplanetární souborový systém (Inter-Planetary File System; IPFS) je decentralizovaný, obsahem adresovatelný úložný systém, který distribuuje uložené objekty mezi uzly v P2P síti. „Adresovatelným obsahem“ se rozumí, že každá část obsahu (souboru) je hašovaná a haš se používá k identifikaci tohoto souboru. Poté můžete načíst libovolný soubor z libovolného uzlu IPFS tak, že jej požádáte o soubor s daným hašem.

Cílem IPFS je nahradit HTTP jako zvolený protokol pro doručování webových aplikací. Místo ukládání webové aplikace na jeden server jsou soubory ukládány do IPFS a lze je získat z libovolného uzlu IPFS.

Více informací o IPFS naleznete na https://ipfs.io.

Swarm

Swarm je další úložný systém P2P, který lze adresovat, podobně jako IPFS. Swarm byl vytvořen Nadací Ethereum, jako součást sady nástrojů Go-Ethereum. Stejně jako IPFS umožňuje ukládat soubory, které jsou šířeny a replikovány uzly Swarm. K libovolnému souboru Swarm se dostanete tak, že na něj odkazuje haš. Funkce Swarm umožňuje přístup k webové stránce z decentralizovaného systému P2P místo z centrálního webového serveru.

Domovská stránka Swarm je sama uložena ve službě Swarm a je přístupná ve vašem uzlu Swarm nebo v bráně: https://swarm-gateways.net/bzz:/theswarm.eth/.

Decentralizované komunikační protokoly zpráv

Další hlavní součástí každé aplikace je meziprocesová komunikace. To znamená, že si lze vyměňovat zprávy mezi aplikacemi, mezi různými instancemi aplikace nebo mezi uživateli aplikace. Tradičně se toho dosahuje spoléháním na centralizovaný server. Existuje však celá řada decentralizovaných alternativ k serverovým protokolům, které nabízejí zasílání zpráv prostřednictvím sítě P2P. Nejvýznamnějším protokolem pro zasílání P2P zpráv pro DApps je *Whisper* [http://bit.ly/2CSls5h], který je součástí Go-Ethereum sady nástrojů Nadace Ethereum.

Posledním aspektem aplikace, kterou lze decentralizovat, je rozlišení názvu. Podrobněji se podíváme na Ethereum jmennou službu dále v této kapitole; teď se však podívejme na příklad.

Základní příklad DApp: Aukční DApp

IV této sekci začneme budovat příklad DApp, abychom prozkoumali různé decentralizační nástroje. Náš DApp bude decentralizovanou aukci.

Aukční DApp umožňuje uživateli zaregistrovat „vlastnickou listinu tokenu“ token, který představuje nějaké jedinečné aktivum, například dům, auto, ochranná známka atd. Jakmile je token zaregistrován, převede se vlastnictví tokenu na aukční DApp, což umožňuje jeho uvedení na prodej.

Aukční DApp obsahuje seznam všech registrovaných tokenů, což umožňuje ostatním uživatelům zadávat nabídky. Během každé aukce se mohou uživatelé připojit k chatovací místnosti vytvořené speciálně pro tuto aukci. Po dokončení aukce je vlastnictví tokenu vlastnictví převedeno na vítěze aukce.

Celkový aukční proces je uveden v Aukce DApp: Jednoduchý příklad aukční DApp;.

Hlavní komponenty naší aukční DApp jsou:

- Chytrý kontrakt implementující ERC721 nezaměnitelné „vlastnické listiny“ tokenů (`DeedRepository`)
- Chytrý kontrakt provádějící dražbu (AuctionRepository) na prodej vlastnických listin
- Webové rozhraní využívající JavaScript rámec Vue / Vuetify
- Knihovna *web3.js* pro připojení k bločence Etherea (přes MetaMask nebo jiné klienty)
- Klient Swarm pro ukládání zdrojů, jako jsou obrázky
- Klient Whisper, pro vytvoření chatovací místnosti pro pro všechny účastníky aukce

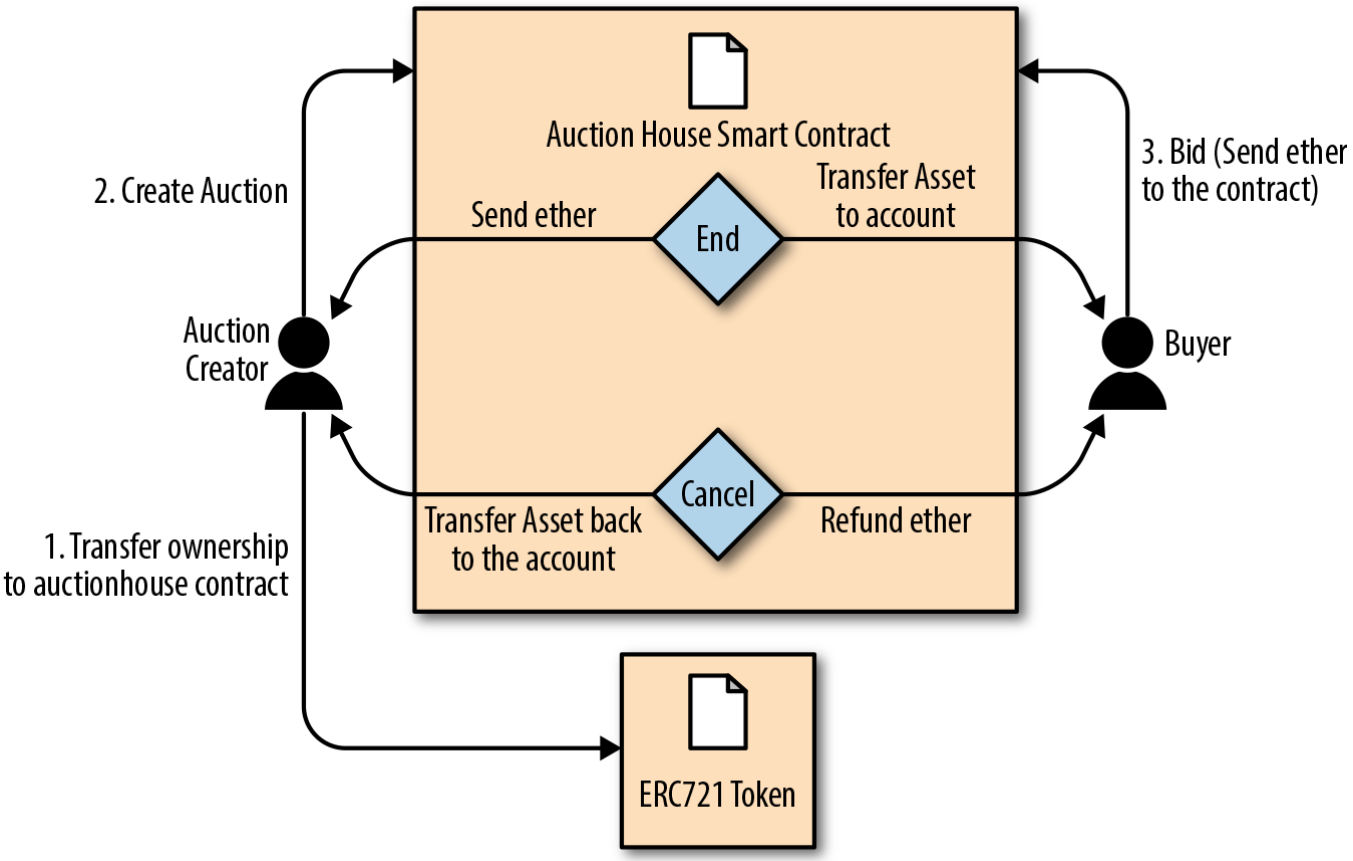

Figure 42. Aukce DApp: Jednoduchý příklad aukční DApp

Zdrojový kód aukční DApp najdete v úložišti knihy [http://bit.ly/2DcmjyA].

Aukční DApp: Backend chytrý kontrakt

Náš příklad aukční DApp je podporován dvěma chytrými kontrakty, které musíme nasadit na Ethereum bločenku, abychom podporovali aplikaci: `AuctionRepository` a DeedRepository.

Začněme tím, že se podíváme na DeedRepository v DeedRepository.sol: Token ERC721 pro použití v aukci. This ERC721 kontrakt je nezaměnitelný token (viz ERC721: Standard nezaměnitelného tokenu (vlastnická listina)).

Example 24. DeedRepository.sol: Token ERC721 pro použití v aukci

```
pragma solidity ^0.4.17;
import "./ERC721/ERC721Token.sol";

/**
 * @title Repository of ERC721 Deeds
 * This contract contains the list of deeds registered by users.
 * This is a demo to show how tokens (deeds) can be minted and added
 * to the repository.
 */
contract DeedRepository is ERC721Token {


    /**
    * @dev Created a DeedRepository with a name and symbol
    * @param _name string represents the name of the repository
    * @param _symbol string represents the symbol of the repository
    */
    function DeedRepository(string _name, string _symbol)
        public ERC721Token(_name, _symbol) {}

    /**
    * @dev Public function to register a new deed
    * @dev Call the ERC721Token minter
    * @param _tokenId uint256 represents a specific deed
    * @param _uri string containing metadata/uri
    */
    function registerDeed(uint256 _tokenId, string _uri) public {
        _mint(msg.sender, _tokenId);
        addDeedMetadata(_tokenId, _uri);
        emit DeedRegistered(msg.sender, _tokenId);
    }

    /**
    * @dev Public function to add metadata to a deed
    * @param _tokenId represents a specific deed
    * @param _uri text which describes the characteristics of a given
deed
    * @return whether the deed metadata was added to the repository
    */
```

```
    function addDeedMetadata(uint256 _tokenId, string _uri) public
returns(bool){
        _setTokenURI(_tokenId, _uri);
        return true;
    }

    /**
    * @dev Event is triggered if deed/token is registered
    * @param _by address of the registrar
    * @param _tokenId uint256 represents a specific deed
    */
    event DeedRegistered(address _by, uint256 _tokenId);
}
```

Jak vidíte, kontrakt DeedRepository je přímá implementace tokenu kompatibilního s ERC721.

Naše aukční DApp používá kontrakt DeedRepository k vydávání a sledování tokenů pro každou aukci. Aukce samotná je řízena kontraktem AuctionRepository. Tento kontrakt je příliš dlouhý na to, aby zde byl zahrnut v plném rozsahu, ale AuctionRepository.sol: Hlavní chytrý kontrakt aukční DApp ukazuje hlavní definici struktury kontraktu a dat. Celý kontrakt je k dispozici v GitHub úložišti knihy [https://bit.ly/2IaOo9i].

Example 25. AuctionRepository.sol: Hlavní chytrý kontrakt aukční DApp

```
contract AuctionRepository {

    // Pole se všemi aukcemi
    Auction[] public auctions;

    // Mapování indexu aukce na uživatelské nabídky
    mapping(uint256 => Bid[]) public auctionBids;

    // Mapování z vlastníka na seznam vlastn¬ných aukcí
    mapping(address => uint[]) public auctionOwner;

    // Struktura nabídek obsahuje uchaze¬e a ¬ástku
    struct Bid {
        address from;
        uint256 amount;
    }

    // Struktura aukce, která obsahuje všechny požadované informace
    struct Auction {
        string name;
        uint256 blockDeadline;
        uint256 startPrice;
        string metadata;
        uint256 deedId;
        address deedRepositoryAddress;
        address owner;
        bool active;
        bool finalized;
    }
```

Kontrakt AuctionRepository spravuje všechny aukce s následujícími funkcemi:

```
getCount()
getBidsCount(uint _auctionId)
getAuctionsOf(address _owner)
getCurrentBid(uint _auctionId)
getAuctionsCountOfOwner(address _owner)
getAuctionById(uint _auctionId)
createAuction(address _deedRepositoryAddress, uint256 _deedId,
              string _auctionTitle, string _metadata, uint256 _startPrice,
              uint _blockDeadline)
approveAndTransfer(address _from, address _to, address
_deedRepositoryAddress,
                   uint256 _deedId)
cancelAuction(uint _auctionId)
finalizeAuction(uint _auctionId)
bidOnAuction(uint _auctionId)
```

Tyto kontrakty můžete nasadit do Ethereum bločenky dle vašeho výběru (např. Ropsten) pomocí truffle v úložišti knihy:

```
<pre data-type="programlisting">
$ <strong>cd code/auction_dapp/backend</strong>
$ <strong>truffle init</strong>
$ <strong>truffle compile</strong>
$ <strong>truffle migrate --network ropsten</strong>
</pre>
```

Správa DApp

Pokud si přečtete dva chytré kontrakty Aukčního DApp, všimnete si něčeho důležitého: neexistuje žádný zvláštní účet nebo role, která má nad DApp zvláštní oprávnění. Každá aukce má majitele s některými speciálními schopnostmi, ale aukční DApp sám nemá žádného privilegovaného uživatele.

Toto je uvážená volba decentralizovat správu DApp a vzdát se jakékoli kontroly, jakmile bude nasazena. Některé DApps mají pro srovnání jeden nebo více privilegovaných účtů se speciálními schopnostmi, jako je například schopnost ukončit kontrakt DApp, přepsat nebo změnit jeho konfiguraci nebo „vetovat“ určité operace. Obvykle jsou tyto funkce správy zavedeny v DApp, aby se zabránilo neznámým problémům, které mohou nastat v důsledku chyby.

Otázka privilegovaného účtu je obzvláště obtížná, protože představuje dvojsečný meč. Na jedné straně jsou privilegované účty nebezpečné; v případě ohrožení mohou narušit zabezpečení DApp. Na druhou stranu bez privilegovaného účtu neexistují žádné možnosti obnovy, pokud je nalezena chyba. Obě tato rizika jsme viděli v Ethereum DApps. V případě The DAO (Skutečný příklad: The DAO a Historie Ethereum rozštěpení) existovaly některé privilegované účty zvané „kurátoři“, ale jejich možnosti byly velmi omezené. Tyto účty nebyly schopny přepsat prostředky, které útočník DAO vybral. V novějším případě zažila decentralizovaná burza Bancor masivní krádež, protože byl ohrožen privilegovaný účet pro správu. Ukázalo se, že Bancor nebyl tak decentralizovaný, jak se původně předpokládalo.

Při vytváření DApp se musíte rozhodnout, zda chcete učinit chytré kontrakty skutečně nezávislými, spustit je a poté je ponechat bez kontroly, nebo si vytvořit privilegované účty a vystavit se riziku ohrožení. Každá volba nese riziko, ale z dlouhodobého hlediska nemohou skutečné DApps mít specializovaný přístup pro privilegované účty - to není decentralizované.

Aukční DApp: Frontend uživatelské rozhraní

Jakmile jsou kontrakty aukční DApp nasazeny, můžete s nimi komunikovat pomocí své oblíbené JavaScript příkazové řádky a web3.js nebo jiné knihovny web3. Většina uživatelů však bude potřebovat snadno použitelné rozhraní. Naše uživatelské rozhraní Aukčního DApp je vytvořeno pomocí rámce JavaScript Vue2 / Vuetify od společnosti Google.

Kód uživatelského rozhraní najdete ve složce *code/auction_dapp/frontend* v úložišti knihy [https://github.com/ethereumbook/ethereumbook]. Adresář má následující strukturu a obsah:

```
frontend/
|-- build
|   |-- build.js
|   |-- check-versions.js
|   |-- logo.png
|   |-- utils.js
|   |-- vue-loader.conf.js
|   |-- webpack.base.conf.js
|   |-- webpack.dev.conf.js
|   `-- webpack.prod.conf.js
|-- config
|   |-- dev.env.js
|   |-- index.js
|   `-- prod.env.js
|-- index.html
|-- package.json
|-- package-lock.json
|-- README.md
|-- src
|   |-- App.vue
|   |-- components
|   |   |-- Auction.vue
|   |   `-- Home.vue
|   |-- config.js
|   |-- contracts
|   |   |-- AuctionRepository.json
|   |   `-- DeedRepository.json
|   |-- main.js
|   |-- models
|   |   |-- AuctionRepository.js
|   |   |-- ChatRoom.js
|   |   `-- DeedRepository.js
|   `-- router
|       `-- index.js
```

Jakmile nasadíte kontrakty, upravte konfiguraci frontendu v *frontend/src/config.js* a zadejte adresy kontraktů+DeedRepository+ a AuctionRepository podle nasazení. Frontend aplikace také potřebuje přístup k Ethereum uzlu nabízejícímu JSON-RPC a rozhraní WebSockets. Jakmile nakonfigurujete rozhraní, spusťte jej pomocí webového serveru v místním počítači:

```
<pre data-type="programlisting">
$ <strong>npm install</strong>
$ <strong>npm run dev</strong>
</pre>
```

Bude spuštěn frontend aukce DApp, který bude přístupný prostřednictvím libovolného webového prohlížeče na adrese http://localhost:8080.

Pokud vše půjde dobře, měla by se zobrazit obrazovka uvedená v Uživatelské rozhraní aukční DApp která ilustruje aukční DApp spuštěný ve webovém prohlížeči.

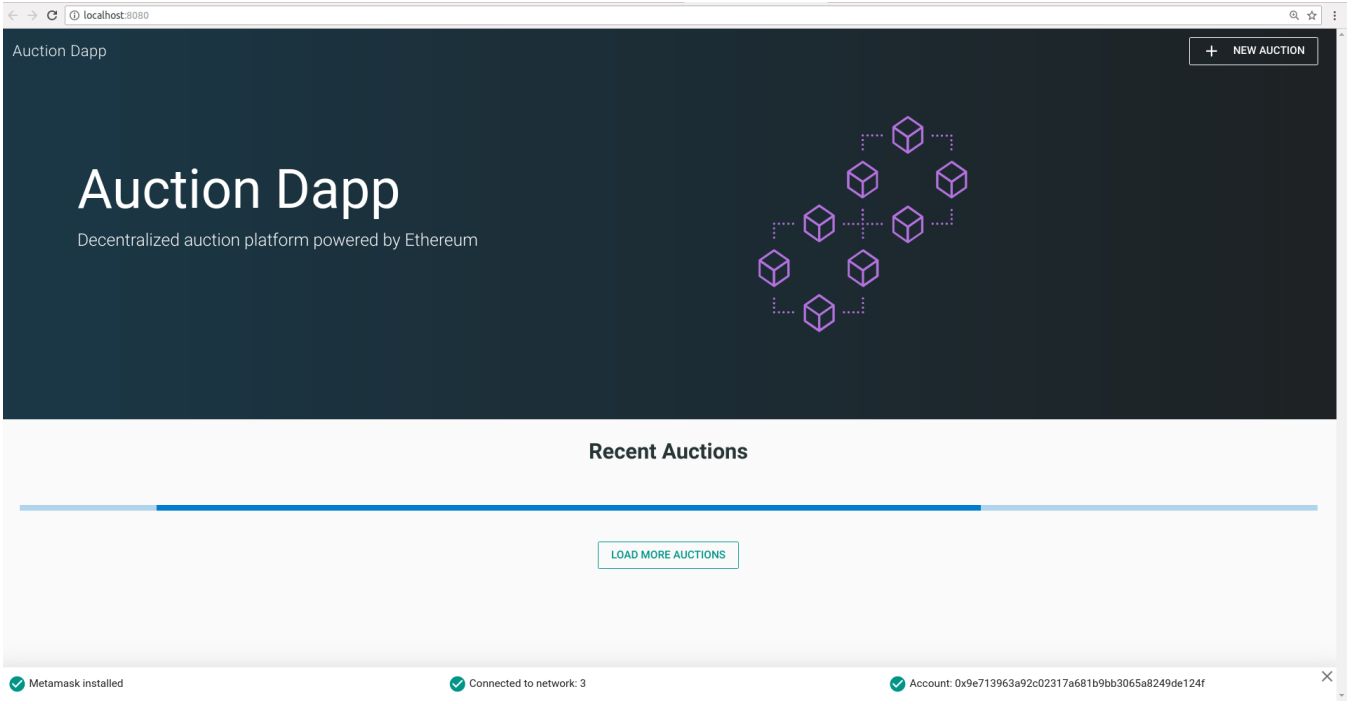

Figure 43. Uživatelské rozhraní aukční DApp

Další decentralizace aukčního DApp

Náš DApp je již docela decentralizovaný, ale můžeme věci vylepšit.

Kontrakt AuctionRepository funguje nezávisle na jakémkoli dohledu, otevřeně pro kohokoli. Jakmile je nasazen, nemůže být zastaven, ani nemůže být kontrolována žádná aukce. Každá aukce má

samostatnou chatovací místnost, která umožňuje komukoli komunikovat o aukci bez cenzury nebo identifikace. Různá aukční aktiva, jako je popis a související obrázek, jsou uložena na Swarm, takže je obtížné je cenzurovat nebo blokovat.

Kdokoli může komunikovat s DApp vytvořením transakcí ručně nebo spuštěním Vue frontend na svém lokálním počítači. Samotný kód DApp má otevřený zdroj a je skupinově vyvíjen se na veřejném úložišti.

Pro decentralizaci a odolnost tohoto DApp můžeme udělat dvě věci:

- Uložit celý kód aplikace do Swarm nebo IPFS.
- Přístupovat k DApp odkazem na jméno pomocí služby Ethereum jmenou službu.

První možnost prozkoumáme v další části a druhou možnost uvedeme v Ethereum jmenná služba (ENS).

Uložení aukční DApp na Swarm

Představili jsme Swarm v Swarm, dříve v této kapitole. Naše aukční DApp již používá Swarm k uložení obrázku ikony pro každou aukci. Toto je mnohem efektivnější řešení než pokus o ukládání dat na Ethereum, což je drahé. Je také mnohem odolnější, než kdyby tyto obrázky byly uloženy v centralizované službě, jako je webový server nebo souborový server.

Ale můžeme věci posunout ještě o krok dále. Můžeme uložit celý frontend samotného DApp ve službě Swarm a spustit jej přímo z uzlu Swarm, namísto spuštění webového serveru.

Příprava Swarm

Abyste mohli začít, musíte nainstalovat Swarm a inicializovat uzel Swarm. Swarmj je součástí Go-Ethereum sady nástrojů Nadace Ethereum. Postupujte podle pokynů k instalaci Go-Ethereum v Go-Ethereum (Geth) nebo při instalaci binárního vydání Swarm postupujte podle pokynů v Swarm dokumentaci [http://bit.ly/2Q75KXw].

Po instalaci Swarm si můžete ověřit, že funguje správně spuštěním příkazu version:

```
<pre data-type="programlisting">
$ <strong>swarm version</strong>
Version: 0.3
Git Commit: 37685930d953bcbe023f9bc65b135a8d8b8f1488
Go Version: go1.10.1
OS: linux
</pre>
```

Abyste mohli spustit Swarm, musíte mu sdělit, jak se připojit k instanci Geth, abyste získali přístup k JSON-RPC API. Začínáme podle pokynů v Průvodci začátečníka [https://swarm-guide.readthedocs.io/en/latest/gettingstarted.html].

Když spustíte Swarm, měli byste vidět něco takového:

```
Maximum peer count                       ETH=25 LES=0 total=25
Starting peer-to-peer node               instance=swarm/v0.3.1-
225171a4/linux...
connecting to ENS API                    url=http://127.0.0.1:8545
swarm[5955]: [189B blob data]
Starting P2P networking
UDP listener up
self=enode://f50c8e19ff841bcd5ce7d2d...
Updated bzz local addr                   oaddr=9c40be8b83e648d50f40ad3...
uaddr=e
Starting Swarm service
9c40be8b hive starting
detected an existing store. trying to load peers
hive 9c40be8b: peers loaded
Swarm network started on bzz address: 9c40be8b83e648d50f40ad3d35f...
Pss started
Streamer started
IPC endpoint opened
url=/home/ubuntu/.ethereum/bzzd.ipc
RLPx listener up
self=enode://f50c8e19ff841bcd5ce7d2d...
```

Připojením k místnímu webovému rozhraní Swarm brány můžete potvrdit, že váš uzel Swarm funguje správně: http://localhost:8500.

Měli byste vidět obrazovku jako na Swarm brána na localhost a být schopni dotazovat se na libovolný Swarm haš nebo ENS jméno.

Figure 44. Swarm brána na localhost

Nahrávání souborů do Swarm

Jakmile bude spuštěn váš místní uzel a brána Swarm, můžete nahrát do Swarm soubory a ty budou přístupné na libovolném uzlu Swarm, jednoduše odkazem na haš souboru.

Vyzkoušejte to nahráním souboru:

```
<pre data-type="programlisting">
$ <strong>swarm up code/auction_dapp/README.md</strong>
ec13042c83ffc2fb5cb0aa8c53f770d36c9b3b35d0468a0c0a77c97016bb8d7c
</pre>
```

Swarm nahrál soubor *README.md* a vrátil hash, který můžete použít pro přístup k souboru z libovolného uzlu Swarm. Můžete například použít veřejnou Swarm bránu [https://bit.ly/2znWUP9].

I když je nahrávání jednoho souboru poměrně jednoduché, nahrání celého rozhraní DApp je o něco složitější. Je to proto, že různé prostředky DApp (HTML, CSS, JavaScript, knihovny atd.) Mají vložené odkazy na sebe. Webový server obvykle převádí adresy URL do místních souborů a poskytuje správné zdroje. Totéž můžeme dosáhnout pro Swarm zabalením našeho DApp.

V aukční DApp je skript pro zabalení všech zdrojů:

```
<pre data-type="programlisting">
$ <strong>cd code/auction_dapp/frontend</strong>
$ <strong>npm run build</strong>

> <strong>frontend@1.0.0 build
/home/aantonop/Dev/ethereumbook/code/auction_dapp/frontend</strong>
> <strong>node build/build.js</strong>

Hash: 9ee134d8db3c44dd574d
Version: webpack 3.10.0
Time: 25665ms
Asset     Size
static/js/vendor.77913f316aaf102cec11.js  1.25 MB
static/js/app.5396ead17892922422d4.js   502 kB
static/js/manifest.87447dd4f5e60a5f9652.js  1.54 kB
static/css/app.0e50d6a1d2b1ed4daa03d306ced779cc.css  1.13 kB
static/css/app.0e50d6a1d2b1ed4daa03d306ced779cc.css.map  2.54 kB
static/js/vendor.77913f316aaf102cec11.js.map  4.74 MB
static/js/app.5396ead17892922422d4.js.map   893 kB
static/js/manifest.87447dd4f5e60a5f9652.js.map  7.86 kB
index.html  1.15 kB

Build complete.
</pre>
```

Výsledkem tohoto příkazu bude nový adresář *code/auction_dapp/frontend/dist*, který obsahuje celý frontend aukčního DApp, zabalený dohromady:

```
dist/
|-- index.html
`-- static
    |-- css
    |   |-- app.0e50d6a1d2b1ed4daa03d306ced779cc.css
    |   `-- app.0e50d6a1d2b1ed4daa03d306ced779cc.css.map
    `-- js
        |-- app.5396ead17892922422d4.js
        |-- app.5396ead17892922422d4.js.map
        |-- manifest.87447dd4f5e60a5f9652.js
        |-- manifest.87447dd4f5e60a5f9652.js.map
        |-- vendor.77913f316aaf102cec11.js
        `-- vendor.77913f316aaf102cec11.js.map
```

Nyní můžete nahrát celý DApp do Swarm pomocí příkazu up a volitelného parametru pro rekurzi --recursive . Zde také řekneme Swarmu, že index.html je výchozí cesta defaultpath pro načtení této DApp:

```
<pre data-type="programlisting">
$ <strong>swarm --bzzapi http://localhost:8500 --recursive \
  --defaultpath dist/index.html up dist/</strong>

ab164cf37dc10647e43a233486cdeffa8334b026e32a480dd9cbd020c12d4581
</pre>
```

Nyní je celý náš aukční DApp hostován ve službě Swarm a je přístupný prostřednictvím Swarm URL:

```
<ul class="simplelist">
<li><em>bzz://ab164cf37dc10647e43a233486cdeffa8334b026e32a480dd9cbd020c12d
4581</em></li>
</ul>
```

Udělali jsme určitý pokrok v decentralizaci naší DApp, ale ztížili jsme použití. Taková adresa URL je mnohem méně uživatelsky přívětivá než pěkné jméno, jako je *auction_dapp.com*. Jsme nuceni obětovat použitelnost, abychom získali decentralizaci? Ne nutně. V další části prozkoumáme jmennou službu Etherea, která nám umožňuje používat snadno čitelná jména, ale stále zachovává decentralizovanou povahu naší aplikace.

Ethereum jmenná služba (ENS)

Můžete navrhnout nejlepší chytrý kontrakt na světě, ale pokud uživatelům neposkytnete dobré rozhraní, nebudou mít k němu přístup.

Na tradičním internetu nám doménový jmenný systém (Domain Name System; DNS) umožňuje používat v prohlížeči lidsky čitelná jména a převádět je na IP adresy nebo jiné identifikátory v zákulisí. Na Ethereum bločence řeší *Ethereum jmenný systém* (Ethereum Naming System; ENS) stejný problém, ale decentralizovaným způsobem.

Například dárcovská adresa Nadace Ethereum Foundation je 0xfB6916095ca1df60bB79Ce92cE3Ea74c37c5d359; v peněžence, která podporuje ENS, je to jednoduše ethereum.eth.

ENS je víc než chytrý kontrakt; je to samotný základní DApp, nabízející decentralizovanou jmennou službu. ENS je navíc podporována řadou DApps pro registraci, správu a aukce registrovaných jmen. ENS ukazuje, jak můžou DApps spolupracovat: je to DApp postavený tak, aby sloužil jiným DApps, podporovaný ekosystémem DApps, zabudovaný v jiných DApps atd.

V této části se podíváme na to, jak funguje ENS. Ukážeme vám, jak si můžete nastavit své vlastní jméno a propojit jej s peněženkou nebo Ethereum adresou, jak můžete vložit ENS do jiného DApp a jak můžete pomocí ENS pojmenovat své DApp prostředky, aby bylo jejich použití snadnější.

Historie Ethereum jmenné služby

Registrace jména byla první neobvyklou aplikací bločenky, která byla propagována kryptoměnou Namecoin. Ethereum Bílá kniha [http://bit.ly/2Of1gfZ] dala jako jeden ze svých příkladů použití dvouřádkový registrační systém typu Namecoin.

Včasná vydání Geth a C++ Ethereum klienta měla vestavěný kontrakt namereg (již se nepoužívá) a bylo vytvořeno mnoho návrhů a ERC pro jmenné služby, ale teprve tehdy, když Nick Johnson začal pracovat pro Nadaci Ethereum v roce 2016 a vzal projekt pod svá křídla, seriózní práce na registrátorovi začaly.

ENS byl spuštěn na den Hvězdných válek, 4. května 2017 (po neúspěšném pokusu o spuštění na den čísla Pi, 15. března).

Specifikace ENS

ENS je specifikováno zejména ve třech návrzích na vylepšení Etherea: EIP-137, který specifikuje základní funkce ENS; EIP-162, který popisuje aukční systém pro kořen .eth; a EIP-181, který určuje reverzní rozlišení adres.

ENS se řídí „sendvičovou“ filozofií návrhu: velmi jednoduchá vrstva dole, následovaná vrstvami složitějšího, ale vyměnitelného kódu, s velmi jednoduchou horní vrstvou, která udržuje všechny prostředky na samostatných účtech.

Dolní vrstva: vlastníci jmen a překladače

ENS pracuje na „uzlech“ místo na lidsky čitelných jména: lidsky čitelné jméno je převedeno na uzel pomocí algoritmu „Namehash“.

Základní vrstva ENS je chytře jednoduchý kontrakt (méně než 50 řádků kódu) definovaný v ERC137, který umožňuje pouze vlastníkům uzlů nastavovat informace o jejich jménech a vytvářet poduzly (ENS ekvivalent DNS subdomén).

Jediné funkce na základní vrstvě jsou ty, které umožňují vlastníkovi uzlu nastavit informace o vlastním uzlu (konkrétně překladač, čas života nebo převést vlastnictví) a vytvořit vlastníky nových podřízených uzlů.

Algoritmus Namehash

Namehash je rekurzivní algoritmus, který dokáže převést libovolné jméno na haš, který identifikuje jméno.

„Rekurzivní“ znamená, že problém vyřešíme vyřešením podproblému, který je menším problémem stejného typu, a poté pomocí řešení podproblému vyřešíme původní problém.

Namehash rekurzivně hašuje komponenty názvu a vytváří jedinečný řetězec pevné délky (nebo „uzel“) pro libovolnou platnou vstupní doménu. Například Namehash uzel subdomain.example.eth je `keccak('<example.eth>' uzlu) + keccak ('<subdomain>') `. Díl¬í problém, který musíme vy¬ešit, je spo¬ítat uzel pro +example.eth+, což je ``keccak('<.eth>' node) + keccak('<example>')`. Abychom mohli začít, musíme vypočítat uzel pro eth, což je `keccak(<root node>) + keccak ('<eth>')`.

Kořenový uzel je to, čemu říkáme „konec rekurze“ a samozřejmě jej nemůžeme definovat rekurzivně, jinak se algoritmus nikdy nezastaví! Kořenový uzel je definován jako „0x0000000000000000000000000000000000000000000000000000000000000000“ (32 nulových bajtů).

Když to dáme celé dohromady, uzel subdomain.example.eth je proto `keccak(keccak(keccak(0x0...0 + keccak('eth')) + keccak('example')) + keccak('subdomain'))`.

Zobecněně, můžeme definovat funkci Namehash následovně (kořenový uzel nebo prázdný název, následovaný rekurzivním krokem):

```
namehash([]) =
0x0000000000000000000000000000000000000000000000000000000000000000
namehash([label, ...]) = keccak256(namehash(...) + keccak256(label))
```

V Pythonu se to provede:

```
def namehash(name):
  if name == '':
    return '\0' * 32
  else:
    label, _, remainder = name.partition('.')
    return sha3(namehash(remainder) + sha3(label))
```

Mastering-ethereum.eth bude tedy zpracováno následujícím způsobem:

```
namehash('mastering-ethereum.eth')
¬ sha3(namehash('eth') + sha3('mastering-ethereum'))
¬ sha3(sha3(namehash('') + sha3('eth')) + sha3('mastering-ethereum'))
¬ sha3(sha3(('\0' * 32) + sha3('eth')) + sha3('mastering-ethereum'))
```

Samotné subdomény mohou samozřejmě mít i subdomény: může existovat sub.subdomain.example.eth za+ subdomain.example.eth+, potom sub.sub.subdomain.example.eth atd. Abychom se vyhnuli nákladnému výpočtu, protože Namehash závisí pouze na samotném názvu, lze uzel pro dané jméno předběžně spočítat a vložit do kontraktu což odstraní potřebu manipulace s řetězci a umožní okamžité vyhledávání záznamů ENS bez ohledu na počet komponent v surovém

jménu.

Jak vybrat platné jméno

Jména se skládají z posloupnosti názvů oddělených tečkami. Ačkoli jsou povolena velká a malá písmena, všechny názvy by měly projít normalizačním procesem UTS # 46, který sjednotí velikost písmen před jejich hašováním, takže názvy s odlišnými velkými a malými písmeny získají stejný Namehash.

Můžete použít názvy a domény libovolné délky, ale kvůli kompatibilitě se starým DNS jsou doporučena následující pravidla:

- názvy by neměly mít více než 64 znaků.
- Celé ENS jméno by nemělo mít více než 255 znaků.
- Názvy by neměly začínat nebo končit pomlčkami nebo začínat číslicemi.

Vlastnictví kořenového uzlu

Jedním z výsledků tohoto hierarchického systému je to, že se spoléhá na vlastníky kořenového uzlu, kteří jsou schopni vytvářet domény nejvyšší úrovně (TLD).

Zatímco konečným cílem je přijmout decentralizovaný rozhodovací proces pro nové TLD, v době psaní této knihy je kořenový uzel řízen 4-z-7 vícepodpisem (multisig), který drží lidé v různých zemích (analogie se 7 držiteli klíčů v systému DNS). Výsledkem je, že k provedení jakékoli změny je vyžadována většina alespoň 4 ze 7 držitelů klíčů.

V současné době je účelem a cílem těchto držitelů klíčů pracovat ve shodě s komunitou na:

- Migraci a vylepšení dočasného vlastnictví .eth TLD na trvalejší kontrakt, po vyhodnocení současného stavu systému.
- Povolit přidávání nových TLD, pokud komunita odsouhlasí, že jsou potřeba.
- Migrace vlastnictví kořene pomocí vícepodpisového kontraktu na decentralizovanější smlouvu, až bude takový systém dohodnut, otestován a implementován.
- Slouží jako poslední možnost řešení jakýchkoli chyb či zranitelností v registrech nejvyšší úrovně.

Překladače

Základní ENS kontrakt nemůže k názvu přidat metadata; to je práce tzv. „překladatelských kontraktů“. Jedná se o kontrakty vytvořené uživatelem, které mohou odpovídat na otázky týkající se názvu, jako je například jaká je Swarm adresa spojená s aplikací, jaká adresa přijímá platby do aplikace (v etheru nebo tokenu) nebo jaký je haš aplikace (ověření její integrity)

Střední vrstva: .eth uzly

V době psaní této knihy, jedinou doménou nejvyšší úrovně, kterou lze jedinečně registrovat v chytrém kontraktu, je .eth.

Probíhá práce na tom, aby tradiční vlastníci DNS domén mohli nárokovat vlastnictví ENS. Teoreticky by to mohlo fungovat pro .com, jediná doména, která byla doposud implementována, je .xyz, a pouze pro testovací síť Ropsten [http://bit.ly/2SwUuFC].

Domény + .eth + jsou distribuovány prostřednictvím aukčního systému. Neexistuje žádný rezervovaný seznam ani priorita a jediným způsobem, jak získat jméno, je použití systému. Aukční systém je složitý kus kódu (přes 500 řádků); většina z počátečních vývojových úsilí (a chyb!) v ENS byla v této části systému. Je však také vyměnitelná a vylepšitelná bez rizika pro prostředky - o tom později.

Vickrey aukce

Jména jsou distribuována prostřednictvím upravené aukce Vickrey. V tradiční aukci Vickrey každý uchazeč předloží zapečetěnou nabídku a všechny z nich budou odhaleny současně, v tom okamžiku nejvyšší nabídka vyhraje aukci, ale zaplatí pouze druhou nejvyšší nabídku. Proto jsou uchazeči motivováni, aby nenabízeli méně než skutečnou hodnotu jména, protože nabízení jejich skutečné hodnoty zvyšuje pravděpodobnost, že vyhrají, ale neovlivní cenu, kterou nakonec zaplatí.

Na bločence jsou nutné některé změny:

- Aby se zajistilo, že uchazeči nebudou předkládat nabídky, které nemají v úmyslu platit, musí předem uzamknout hodnotu rovnající se nebo vyšší než jejich nabídka, aby byla zaručena platnost nabídky.

- Protože nemůžete skrýt tajemství na bločence, uchazeči musí provést alespoň dvě transakce (proces odevzdání - odhalení), aby skryli původní hodnotu a jméno, na které nabízejí.
- Protože nemůžete odhalit všechny nabídky současně v decentralizovaném systému, uchazeči musí sami odhalit své vlastní nabídky; pokud tak neučiní, propadají jejich blokované prostředky. Bez tohoto propadnutí by člověk mohl učinit mnoho nabídek a rozhodnout se odhalit pouze jednu nebo dvě, čímž se aukce uzavřených nabídek promění v tradiční aukci s rostoucí cenou.

Aukce je tedy čtyřkrokovým procesem:

1. Zahajte aukci. To je vyžadováno pro vysílání záměru zaregistrovat jméno. Tím se vytvoří všechny aukční termíny. Jména jsou hašovaná, takže pouze ti, kteří mají ve svém slovníku toto jméno, budou vědět, která aukce byla otevřena. To umožňuje určité soukromí, což je užitečné, pokud vytváříte nový projekt a nechcete o něm sdílet podrobnosti. Můžete otevřít více falešných aukcí současně, takže pokud vás někdo sleduje, nemůže jednoduše podat nabídku na všechny otevřené aukce.
2. Udělejte zapečetěnou nabídku. Musíte to udělat před uzávěrkou nabídkového řízení tím, že dané množství éteru připojíte k haši tajné zprávy (obsahující mimo jiné haš jména, skutečnou částku nabídky a sůl). Můžete zamknout více etheru, než jste ve skutečnosti nabízeli, abyste maskovali své skutečné ocenění.
3. Odhalte nabídku. Během období odhalení musíte provést transakci, která odhalí nabídku, která poté vypočítá nejvyšší nabídku a druhou nejvyšší nabídku a odešle ether zpět neúspěšným uchazečům. Pokaždé, když je nabídka odhalena, aktuální vítěz se přepočítá; Poslední vítěz, který bude stanoven před uplynutím lhůty pro odhalení, se proto stává celkovým vítězem.
4. Nakonec vyčistěte. Pokud jste vítězem, můžete aukci dokončit, abyste získali zpět rozdíl mezi vaší nabídkou a druhou nejvyšší nabídkou. Pokud jste zapomněli odhalit, můžete udělat pozdní odhalení a získat zpět část své nabídky.

Vrchní vrstva: vlastnické listiny

Vrchní vrstva ENS je další velmi jednoduchý kontrakt s jediný účel: držet prostředky.

Když vyhrajete jméno, finanční prostředky nejsou nikde posílány, ale jsou pouze uzamčeny na dobu, kdy chcete jméno držet (alespoň rok). Funguje to jako zaručené odkupy: pokud majitel již nechce

jméno, může jej prodat zpět do systému a obnovit svůj ether (takže náklady na držení jména jsou příležitostné náklady na provedení něčeho s návratností větší než nula).

Samozřejmě, že mít jediný kontrakt držící miliony dolarů v etheru, se ukázalo být velmi riskantní, takže místo toho ENS vytváří kontrakt o vlastnictví pro každé nové jméno. Kontrakt vlastnické lilstiny je velmi jednoduchý (asi 50 řádků kódu) a umožňuje pouze převádět finanční prostředky na jediný účet (vlastník listiny) a vyvolat je jediným subjektem (registrační kontrakt). Tento přístup drasticky redukuje prostor pro útok, kde chyby mohou ohrozit finanční prostředky.

Registrace jména

Registrace jména v ENS je čtyřkrokový proces, jak jsme viděli v Vickrey aukce. Nejprve umístíme nabídku na jakékoli dostupné jméno a poté po 48 hodinách nabídku zveřejníme, abychom jméno zajistili. Časová osa pro registraci ENS je schéma ukazující časovou osu registrace.

Registrujme naše první jméno!

K vyhledání dostupných jmen použijeme jedno z několika uživatelsky příjemných rozhraní, umístíme nabídku na jméno ethereumbook.eth, odhalíme nabídku a zabezpečíme jméno.

Existuje celá řada webových rozhraní s ENS, která nám umožňují interagovat s DApp ENS. V tomto příkladu použijeme MyCrypto rozhraní [https://mycrypto.com/] ve spojení s MetaMaskem jako naší peněženkou.

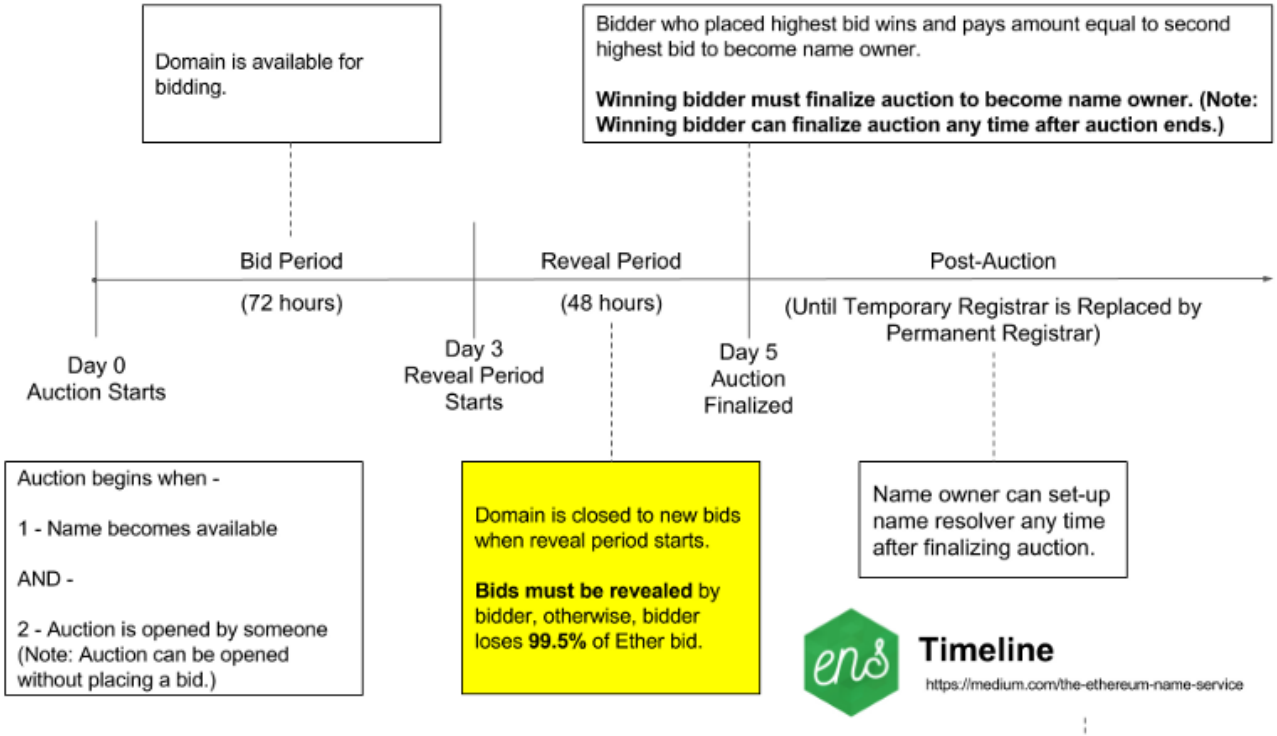

Figure 45. Časová osa pro registraci ENS

Nejprve se musíme ujistit, že je k dispozici požadované jméno. Při psaní této knihy jsme opravdu chtěli zaregistrovat jméno mastering.eth, ale bohužel < <ens-name-search>> jsme zjistili, že je již zabráno! Protože registrace ENS trvají pouze jeden rok, je možné, že bude možné toto jméno v budoucnu zabezpečit. Do té doby použijeme ethereumbook.eth (Vyhledávání ENS jmen na MyCrypto.com).

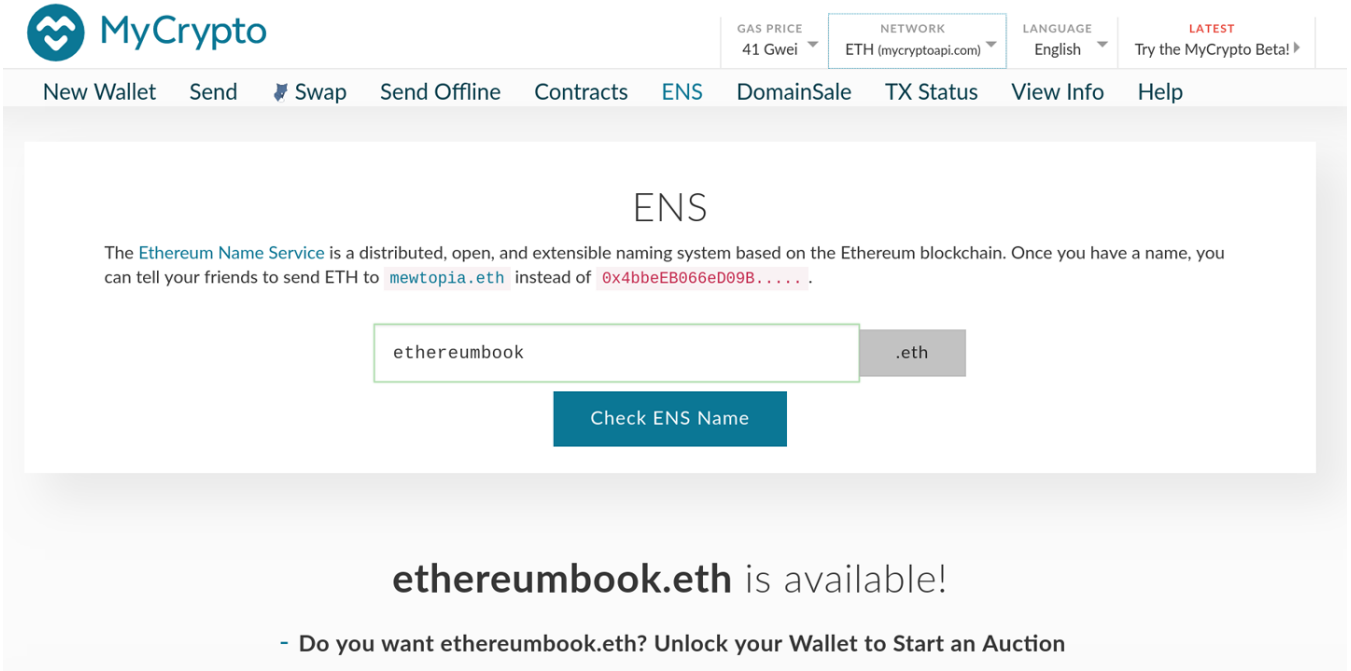

Figure 46. Vyhledávání ENS jmen na MyCrypto.com

Skvělý! Název je k dispozici. Abychom ji mohli zaregistrovat, musíme se pohnout vpřed pomocí Vyvolání aukce pro ENS jméno ;. Pojďme odemknout MetaMask a zahájit aukci pro ethereumbook.eth.

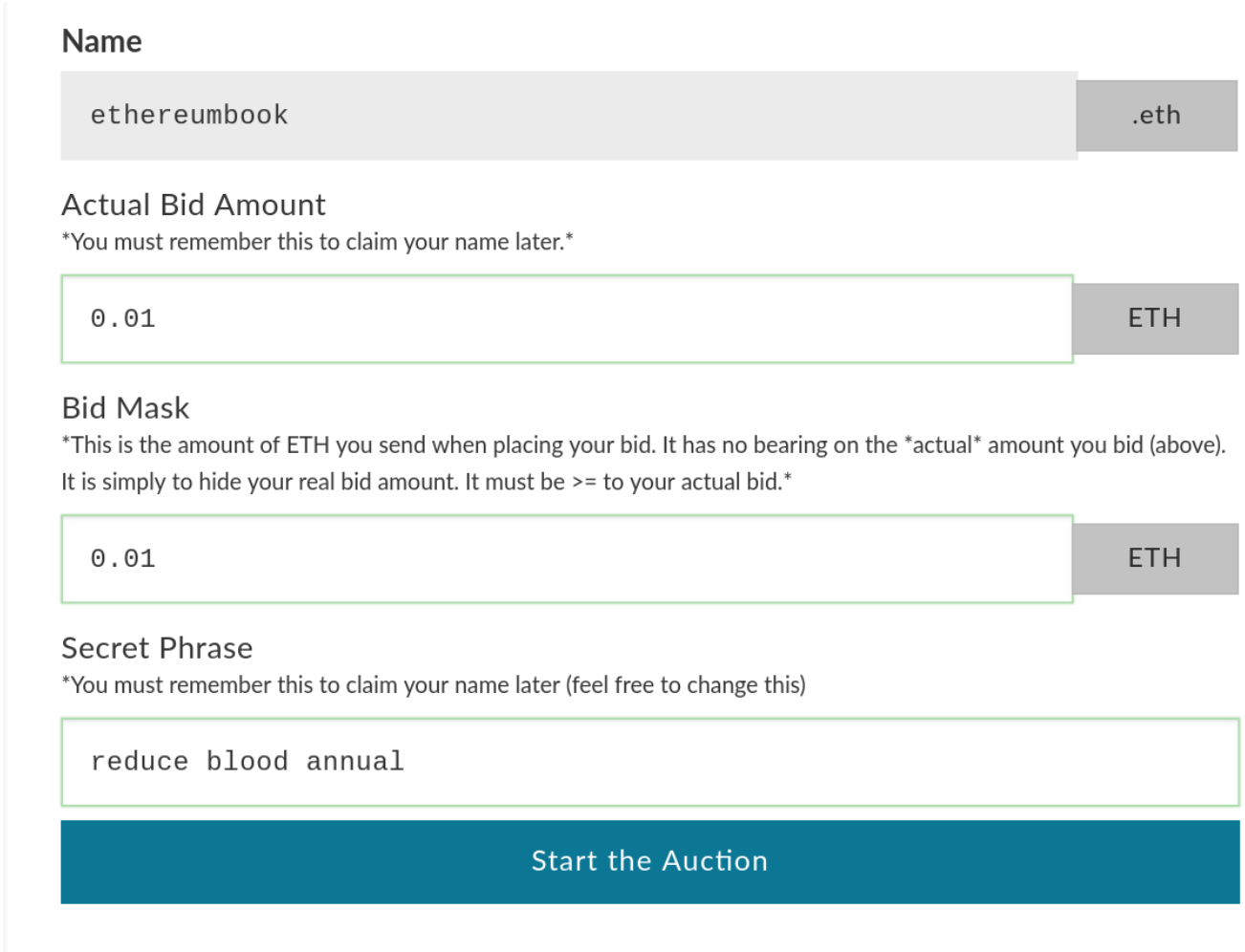

Figure 47. Vyvolání aukce pro ENS jméno

Udělejme naši nabídku. Abychom to mohli udělat, musíme postupovat podle kroků v Udělání nabídky na ENS jméno .

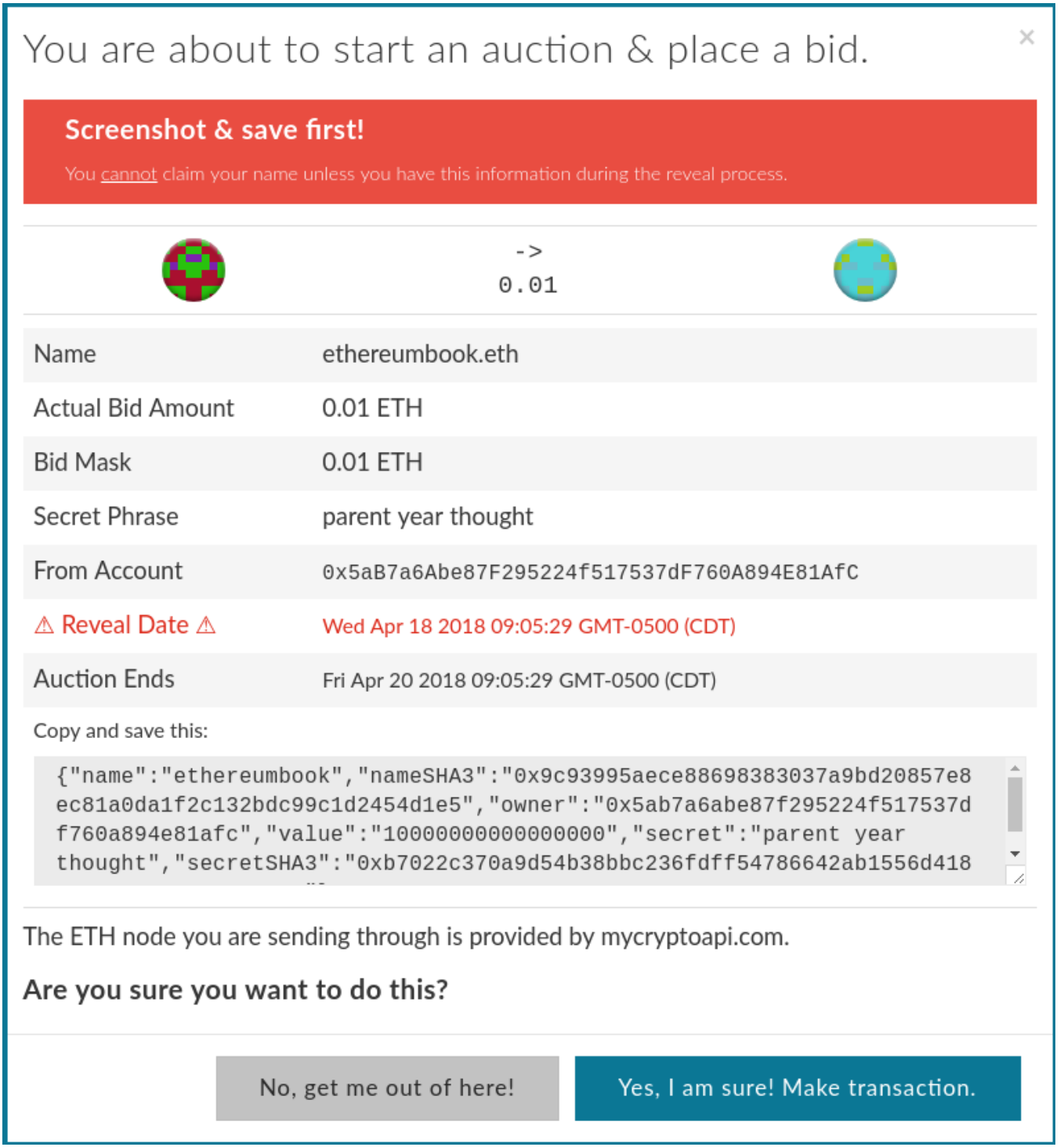

Figure 48. Udělání nabídky na ENS jméno

Jak je uvedeno v Vickrey aukce musíte nabídku zveřejnit do 48 hodin po dokončení dražby, nebo *ztratíte prostředky v nabídce*. Zapomněli jsme to udělat a sami jsme ztratili 0,01 ETH? Vsadíte se, že jsme to udělali.

Udělejte snímek obrazovky, uložte tajnou frázi (jako zálohu nabídky) a přidejte do kalendáře připomenutí ohledně data a času odhalení, abyste nezapomněli a neztratili prostředky.

Nakonec transakci potvrdíme kliknutím na velké zelené tlačítko Odeslat zobrazené v MetaMask transakce obsahující vaši nabídku.

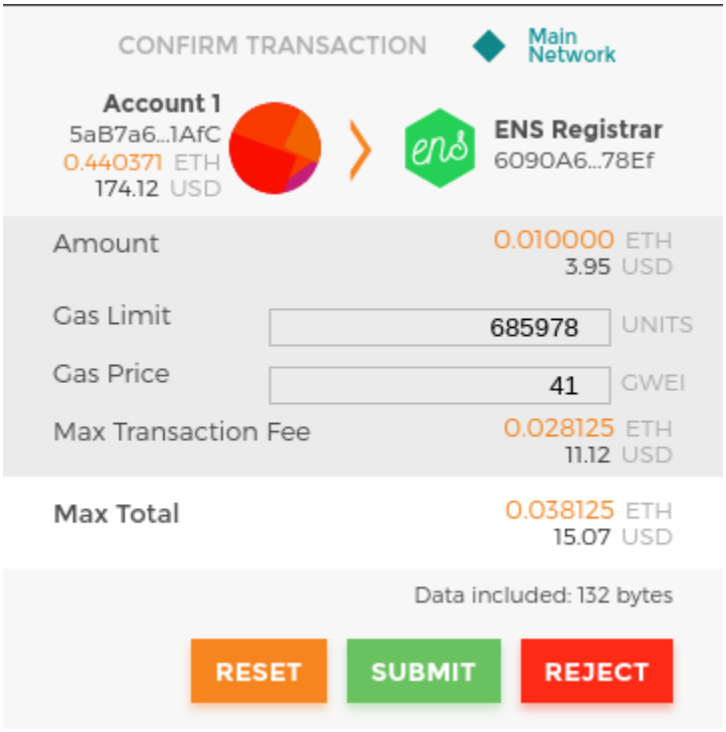

Figure 49. MetaMask transakce obsahující vaši nabídku

Pokud vše půjde dobře, po odeslání transakce tímto způsobem můžete se do do 48 hodin vrátit a odhalit nabídku a požadované jméno bude zaregistrováno na vaši Ethereum adresu.

Správa vašeho ENS jména

Jakmile zaregistrujete ENS jméno, můžete jej spravovat pomocí jiného uživatelsky přívětivého

rozhraní: https://manager.ens.domains/ [Správce ENS].

Poté zadejte do vyhledávacího pole jméno, které chcete spravovat (viz < <ens-manager>>). Musíte mít odblokovanou peněženku Ethereum (např. MetaMask), aby DApp Správce ENS mohla spravovat jméno podle vašich pokynů.

Figure 50. Webové rozhraní Správce ENS

Z tohoto rozhraní můžeme vytvořit subdomény, nastavit kontrakt překladače (více o tom později) a připojit každé jméno k příslušnému zdroji, jako je Swarm adresa frontendu DApp.

Vytvoření subdomény ENS

Nejprve vytvořme subdoménu pro náš příklad Aukční DApp (viz Přidávání subdomény auction.ethereumbook.eth). Pojmenujeme subdoménu auction, takže plně kvalifikovaný název bude auction.ethereumbook.eth.

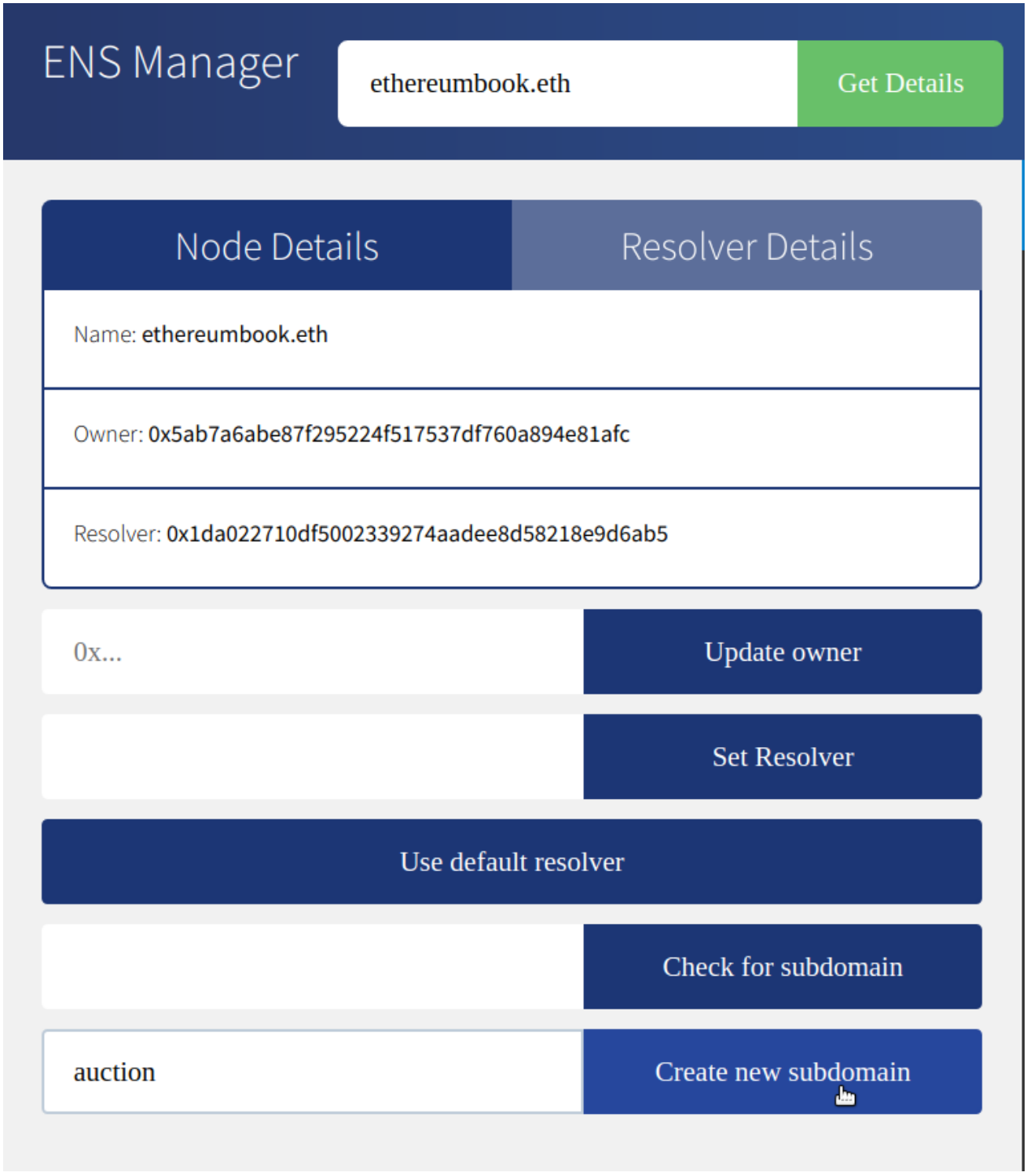
ENS Manager
ethereumbook.eth
Get Details
Node Details
Resolver Details
Name: ethereumbook.eth
Owner: 0x5ab7a6abe87f295224f517537df760a894e81afc
Resolver: 0x1da022710df5002339274aadee8d58218e9d6ab5
0x...
Update owner
Set Resolver
Use default resolver
Check for subdomain
auction
Create new subdomain

Figure 51. Přidávání subdomény auction.ethereumbook.eth

Jakmile jsme vytvořili subdoménu, můžeme do vyhledávacího pole zadat auction.ethereumbook.eth a spravovat ji, stejně jako jsme spravovali doménu ethereumbook.eth dříve.

ENS překladače

V ENS je rozlišení názvu dvoufázový proces:

1. Registr ENS se volá s názvem, který má být po jeho zahašování přeložen. Pokud záznam existuje, registr vrátí adresu jeho překládače.
2. Překladač je volán pomocí metody vhodné pro požadovaný zdroj. Překladač vrátí požadovaný výsledek.

Tento dvoustupňový proces má několik výhod. Oddělení funkčnosti překladačů od samotného pojmenovacího systému nám dává mnohem větší flexibilitu. Vlastníci jmen mohou pomocí vlastních překladačů vyřešit jakýkoli typ nebo prostředek a rozšířit tak funkčnost ENS. Například, pokud jste v budoucnu chtěli propojit geolokační prostředek (zeměpisná délka / šířka) s názvem ENS, můžete vytvořit nový překladač, který odpovídá dotazu `geolocation`. Kdo ví, jaké aplikace mohou být užitečné v budoucnosti? U vlastních překladačů je jediným omezením vaše představivost.

Pro zjednodušení existuje výchozí veřejný překladač, který dokáže vyřešit různé zdroje, včetně adresy (pro peněženky nebo kontrakty) a obsahu (Swarm haš pro DApp nebo zdrojový kód kontraktu).

Protože chceme propojit naši aukční DApp se Swarm hašem, můžeme použít veřejný překladač, který podporuje rozlišení obsahu, jak je uvedeno v<<ens-manager-set-default-resolver> >; nemusíme kódovat ani nasazovat vlastní překladač.

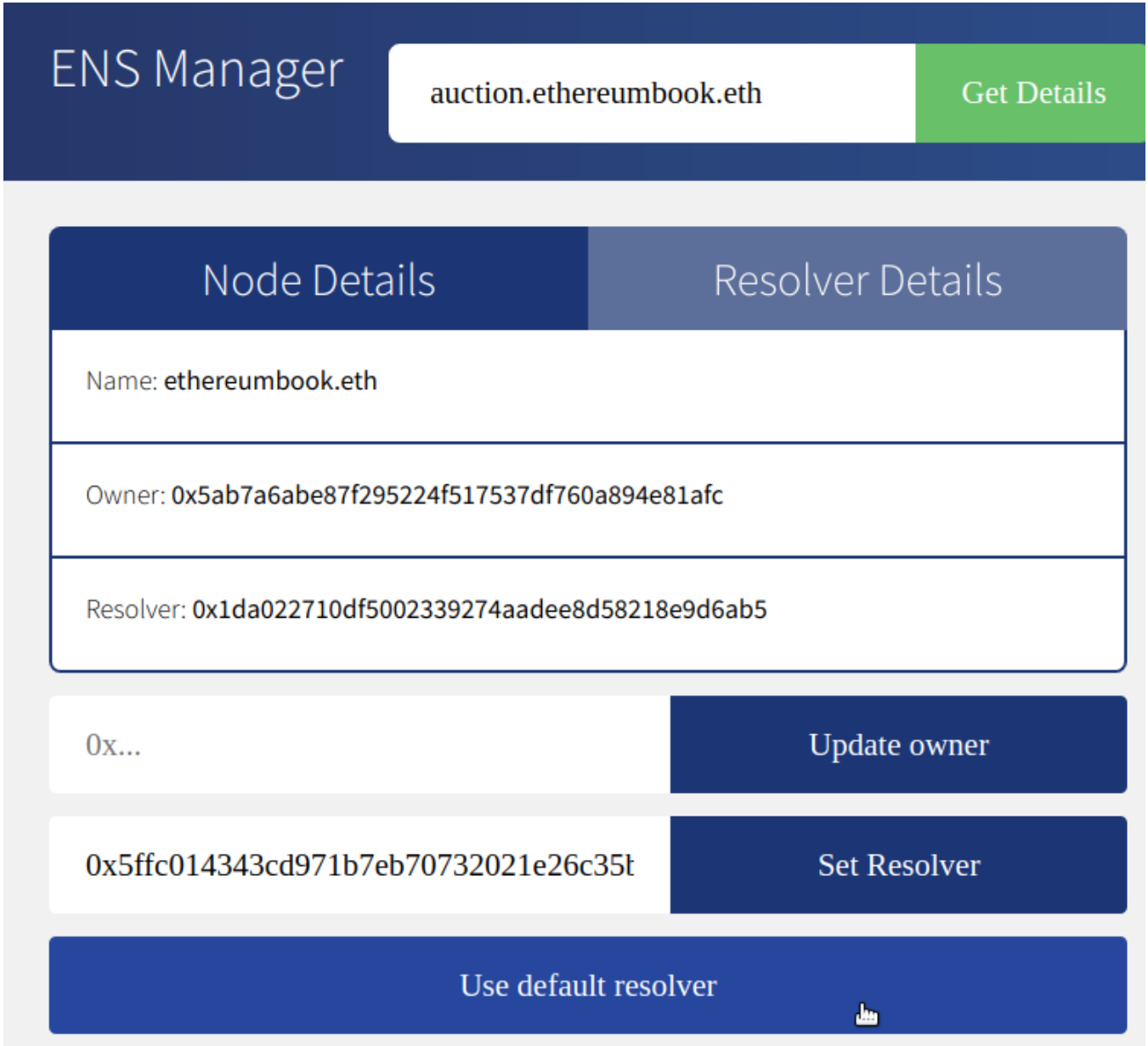

Figure 52. Nastavení výchozího veřejného překladače pro auction.ethereumbook.eth

Překlad jména na Swarm haš (obsah)

Jakmile je překladač pro +auction.ethereumbook.eth nastaven jako veřejný překladač, můžeme jej nastavit tak, aby vrátil Swarm haš ako obsah našeho jména (viz Nastavení 'obsahu' pro návrat na

auction.ethereumbook.eth).

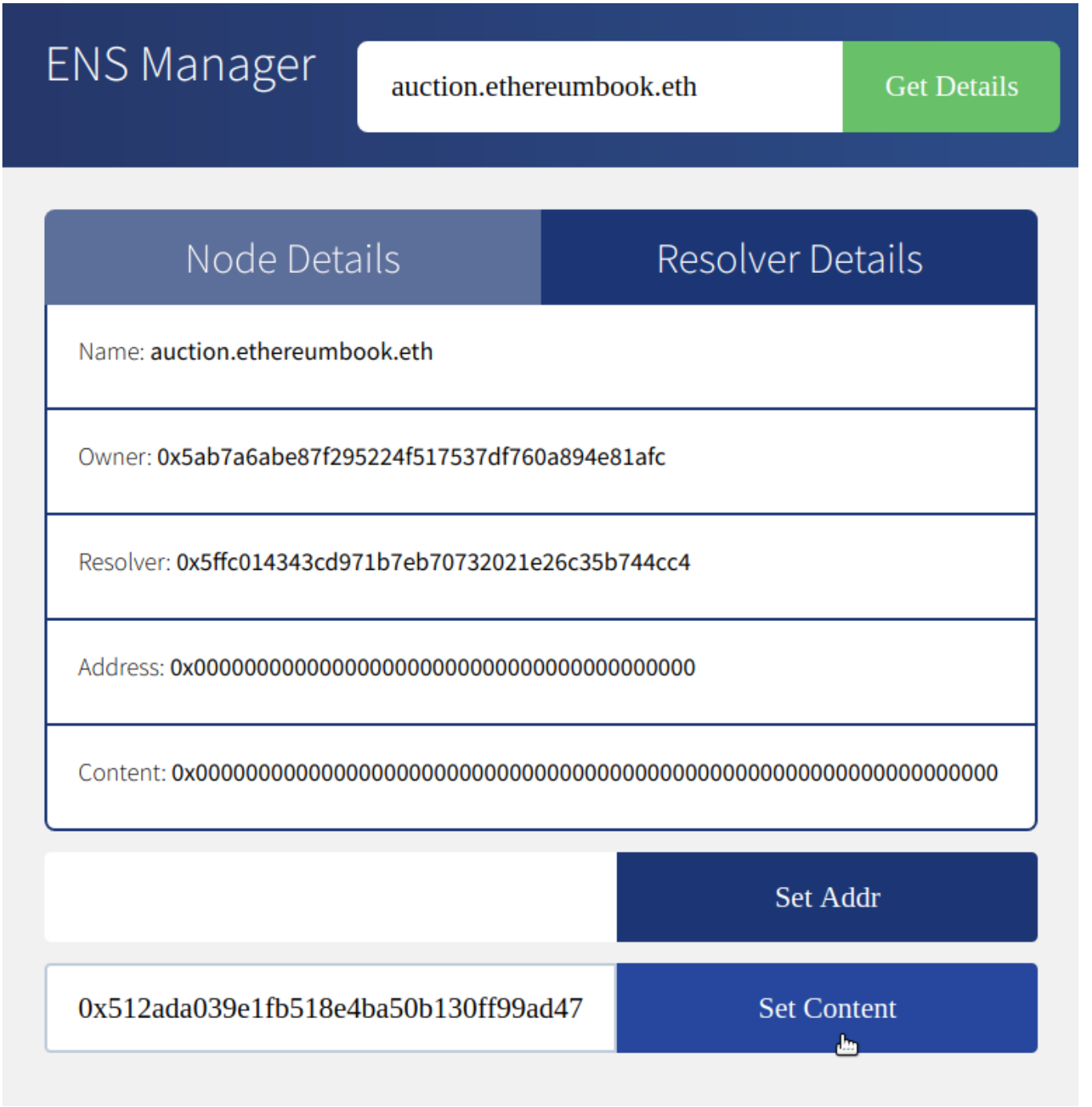

Figure 53. Nastavení 'obsahu' pro návrat na auction.ethereumbook.eth

Po chvíli čekání na potvrzení naší transakce bychom měli být schopni název správně přeložit. Před nastavením názvu, šlo najít naší aukční DApp na Swarm bráně podle haše:

```
<ul class="simplelist">
<li><em>https://swarm-
gateways.net/bzz:/ab164cf37dc10647e43a233486cdeffa8334b026e32a480dd9cbd020
c12d4581</em></li>
</ul>
```

nebo v prohlížeči DApp nebo Swarm bráně hledáním Swarm URL:

```
<ul class="simplelist">
<li><em>bzz://ab164cf37dc10647e43a233486cdeffa8334b026e32a480dd9cbd020c12d
4581</em></li>
</ul>
```

Nyní, když jsme ji připojili ke jménu, je to mnohem snazší:

```
<ul class="simplelist">
<li><em>http://swarm-gateways.net/bzz:/auction.ethereumbook.eth/</em></li>
</ul>
```

Nalezneme jej také hledáním „aukce.ethereumbook.eth“ v libovolné ENS-kompatibilní peněžence nebo DApp prohlížeči (např. Mist). .

Od aplikace do DApp

Během několika posledních sekcí jsme postupně vytvořili decentralizovanou aplikaci. Začali jsme s párem chytrých kontraktů, abychom mohli zahájit aukci za vlastnickými listinami ERC721. Tyto kontrakty byly navrženy tak, aby neobsahovaly žádné spravované ani privilegované účty, takže jejich provoz je skutečně decentralizovaný. Přidali jsme frontend implementovaný do JavaScriptu, který nabízí pohodlné a uživatelsky přívětivé rozhraní pro náš DApp. Aukce DApp používá k ukládání prostředků aplikace, jako jsou obrázky, decentralizovaný úložný systém Swarm. DApp také používá decentralizovaný komunikační protokol Whisper a nabízí šifrovanou chatovací místnost pro každou aukci bez centrálních serverů.

Celý frontend jsme nahráli do Swarmu, takže náš DApp nespoléhá na žádné webové servery, které by soubory obsluhovaly. Nakonec jsme přidělili jméno pro náš DApp pomocí ENS a připojili jej k rozhraní Swarm haše frontendu, aby k němu uživatelé měli přístup pomocí jednoduchého a snadno zapamatovatelného lidsky čitelného jména.

S každým z těchto kroků jsme zvýšili decentralizaci naší aplikace. Konečným výsledkem je DApp, který nemá centrální autoritu, žádný centrální bod selhání a vyjadřuje vizi „web3“.

Architektura aukční DApp ukazuje úplnou architekturu aukčního DApp.

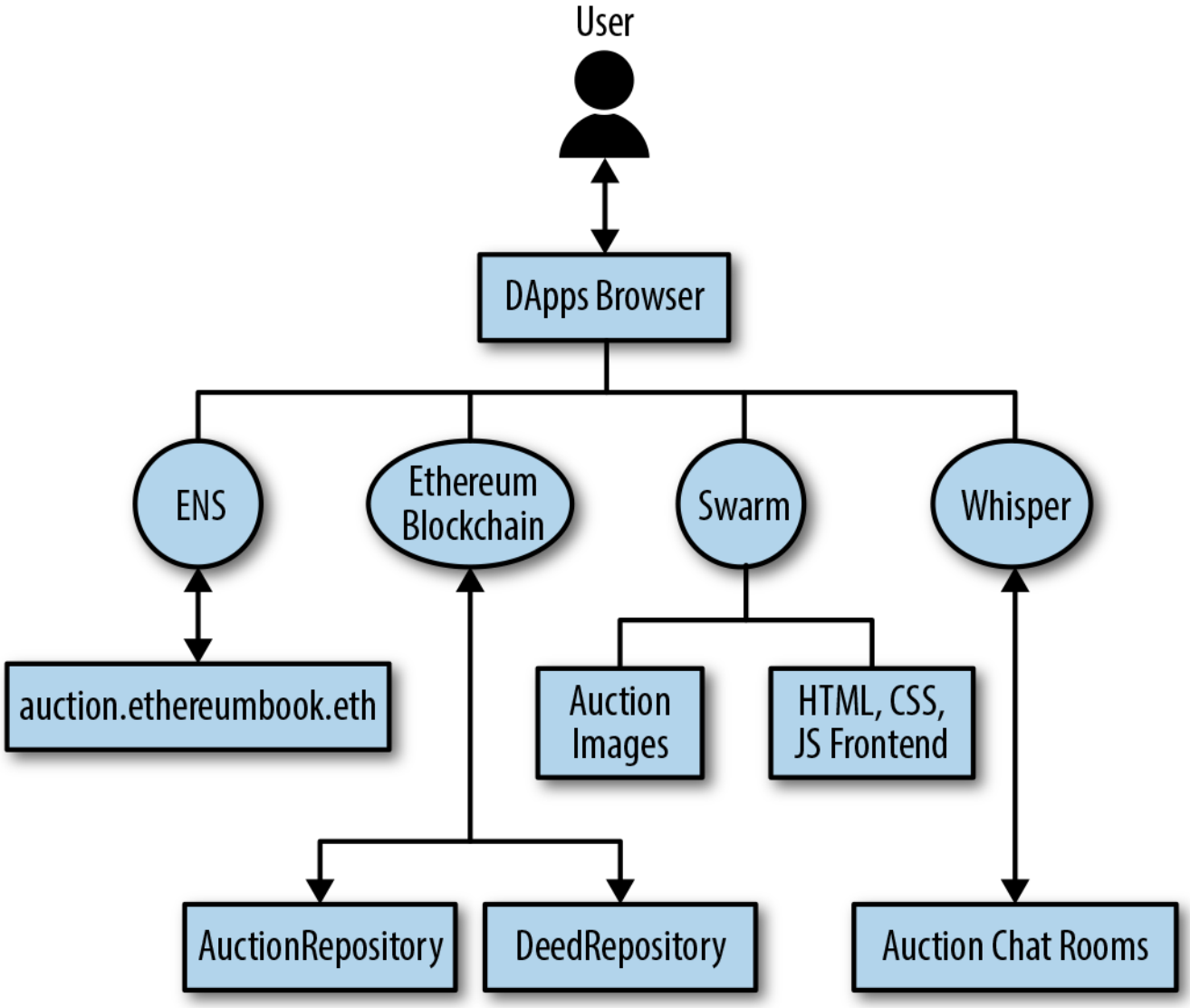

Figure 54. Architektura aukční DApp

Závěry

Decentralizované aplikace jsou vyvrcholením vize Etherea, jak ji vyjádřili zakladatelé od nejranějších návrhů. Zatímco mnoho aplikací se dnes nazývá „DApps“, většina z nich není plně decentralizovaná. Je však již možné vytvářet aplikace, které jsou téměř zcela decentralizované. Postupem času, jak se technologie dále zraje, lze stále více našich aplikací decentralizovat, což vede k odolnějšímu, necenzurovatelnému a bezplatnému webu.

Ethereum virtuální stroj

V jádru Ethereum protokolu a funkčnosti je Ethereum, virtuální stroj (Ethereum Virtual Machine; EVM). Jak byste asi mohli uhodnout ze jména, jedná se o výpočetní stroj, který není nijak výrazně odlišný od virtuálních strojů Microsoft .NET Framework nebo interpretů jiných programovacích jazyků kompilovaných podle bajtkódu, jako je Java. V této kapitole se podrobně podíváme na EVM, včetně jeho instrukční sady, struktury a fungování, v kontextu aktualizací Ethereum stavu.

Co je EVM?

EVM je součást Etherea, která zajišťuje nasazení a provádění chytrých kontraktů. Transakce s jednoduchým převodem hodnot z jednoho EOA do druhého nemusí prakticky EVM zahrnovat, ale všechno ostatní bude zahrnovat aktualizaci stavu vypočítanou EVM. Na vysoké úrovni lze EVM běžící na Ethereum bločence považovat za globální decentralizovaný počítač obsahující miliony spustitelných objektů, každý s vlastním trvalým datovým úložištěm.

EVM je stroj kvazi – Turingovsky úplný stavový stroj; „kvazi“, protože všechny procesy provádění jsou omezeny na konečný počet výpočtových kroků množstvím plynu dostupného pro jakékoli dané vykonání chytrého kontraktu. Jako takový je vyřešen problém zastavení Turingova stroje (všechna spuštění programu se zastaví) a je zabráněno situaci, kdy by provádění (náhodně nebo škodlivě) mohlo běžet navždy, čímž by se zastavila Ethereum platforma jako celek.

EVM má architekturu založenou na zásobníku, která ukládá všechny hodnoty v paměti do zásobníku. Pracuje s velikostí slova 256 bitů (hlavně pro usnadnění nativního hašování a operací eliptických křivek) a má několik adresovatelných datových komponent:

- Neměnný *programový kód ROM*, s načteným bajtovým kódem chytrého kontraktu, který má být proveden
- Dočasná *paměť*, s každým paměťovým místem explicitně inicializovaným na nulu Trvalé *úložiště*, které je součástí Ethereum stavu , také inicializované nulami

Během provádění je k dispozici také sada proměnných prostředí a dat. Podrobněji je projdeme dále v této kapitole.

Architektura a prováděcí kontext Ethereum virtuálního počítače (EVM) zobrazuje kontext architektury a provádění EVM.

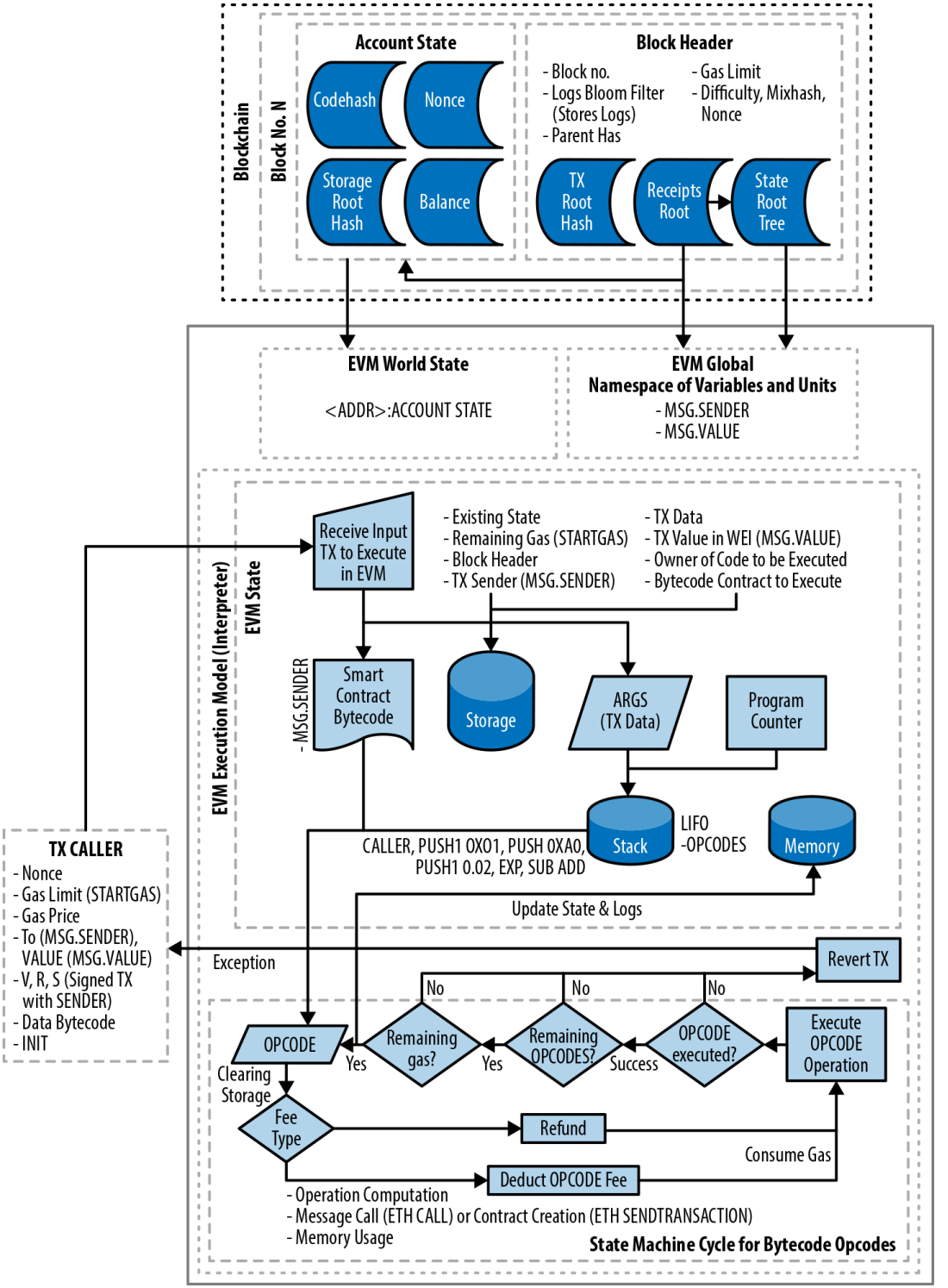
Blockchain
Block No. N
Account State
Codehash
Nonce
Storage Root Hash
Balance
Block Header
- Block no.
- Logs Bloom Filter (Stores Logs)
- Parent Has
- Gas Limit
- Difficulty, Mixhash, Nonce
TX Root Hash
Receipts Root
State Root Tree
EVM World State
<ADDR>:ACCOUNT STATE
EVM Global Namespace of Variables and Units
- MSG.SENDER
- MSG.VALUE
EVM Execution Model (Interpreter)
EVM State
Receive Input TX to Execute in EVM
- Existing State
- Remaining Gas (STARTGAS)
- Block Header
- TX Sender (MSG.SENDER)
- TX Data
- TX Value in WEI (MSG.VALUE)
- Owner of Code to be Executed
- Bytecode Contract to Execute
- MSG.SENDER
Smart Contract Bytecode
Storage
ARGS (TX Data)
Program Counter
Stack
LIFO
-OPCODES
Memory
CALLER, PUSH1 0X01, PUSH 0XA0, PUSH1 0.02, EXP, SUB ADD
Update State & Logs
TX CALLER
- Nonce
- Gas Limit (STARTGAS)
- Gas Price
- To (MSG.SENDER), VALUE (MSG.VALUE)
- V, R, S (Signed TX with SENDER)
- Data Bytecode
- INIT
Exception
Revert TX
No
No
No
OPCODE
Remaining gas?
Remaining OPCODES?
OPCODE executed?
Execute OPCODE Operation
Yes
Yes
Success
Clearing Storage
Fee Type
Refund
Consume Gas
Deduct OPCODE Fee
- Operation Computation
- Message Call (ETH CALL) or Contract Creation (ETH SENDTRANSACTION)
- Memory Usage
State Machine Cycle for Bytecode Opcodes

Figure 55. Architektura a prováděcí kontext Ethereum virtuálního počítače (EVM)

Srovnání se stávající technologií

Termín „virtuální stroj“ se často používá pro virtualizaci skutečného počítače, obvykle pomocí „hypervizora“, jako je VirtualBox nebo QEMU, nebo celé instance operačního systému, například KVM Linuxu. Musí poskytovat softwarovou abstrakci skutečného hardwaru a systémových volání a dalších funkcí jádra.

EVM pracuje v mnohem omezenější doméně: je to pouze výpočetní stroj, a jako takový poskytuje abstrakci pouze výpočetního a úložného prostoru, podobně jako například specifikace Java Virtual Machine (JVM). Z pohledu na vysoké úrovni je JVM navržen tak, aby poskytoval běhové prostředí, které je nezávislý na hostitelskému operačnímu systému nebo hardwaru a umožňuje kompatibilitu v celé řadě systémů. Vysokoúrovňové programovací jazyky, jako je Java nebo Scala (které používají JVM) nebo C# (které používají .NET), jsou kompilovány do sady bajtód instrukcí jejich příslušného virtuálního stroje. Stejně tak EVM vykonává svou vlastní instrukční sadu bajtkódu (popsanou v následující části), do které jsou kompilovány nadřazené inteligentní programovací jazyky kontraktů, jako jsou LLL, Serpent, Mutan nebo Solidity.

EVM proto nemá žádnou možnost plánování, protože pořadí vykonávání instrukcí je organizováno externě - Ethereum klienti procházejí ověřenými bloky transakcí, aby určili, které chytré kontrakty je třeba provést a v jakém pořadí. V tomto smyslu je světový počítač Ethereum jednovláknový, jako JavaScript. EVM také nemá žádnou manipulaci se „systémovým rozhraním“ ani „hardwarovou podporu“ - neexistuje žádný fyzický stroj, s nímž by bylo možné propojit. Počítač světa Ethereum je zcela virtuální.

Instrukční sada EVM (bajtkódové operace)

Instrukční sada EVM nabízí většinu operací, které můžete očekávat, zahrnuje:

- Aritmetické a bitové logické operace
- Provedení kontextových dotazů
- Přístup k zásobníku, paměti a úložišti
- Řízení toku operací
- Protokolování, volání a další operátory

Kromě typických operací s bajtkódem má EVM také přístup k informacím o účtu (např. adresa a zůstatek) a informacím o blocích (např. číslo bloku a aktuální cena plynu).

Začněme podrobněji zkoumáním EVM, když se podíváme na dostupné instrukce a co dělají. Jak byste mohli očekávat, všechny operandy jsou převzaty ze zásobníku a výsledek (pokud je použitelný) je často dát zpět na vrchol zásobníku.

Kompletní seznam instrukcí a k ním odpovídající ceny plynu jsou uvedeny v Instrukce Ethereum EVM a spotřeba plynu.

Dostupné instrukce lze rozdělit do následujících kategorií:

Aritmetické operace

Aritmetické instrukce:

```
ADD        //Se¬te dv¬ vrchní položky v zásobníku
MUL        //Vynásobí dv¬ vrchní položky v zásobníku
SUB        //Ode¬te dv¬ vrchní položky v zásobníku
DIV        //celo¬íselné d¬lení
SDIV       //celo¬íselné d¬lené znaménkových ¬ísel
MOD        //zbytek po celo¬íselném d¬lení (modulo)
SMOD       //zbytek po celo¬íselném d¬lení znaménkových ¬ísel
ADDMOD     //s¬ítání, na které je následn¬ aplikován zbytek po
celo¬íselném d¬lení
MULMOD     //násobení, na které je následn¬ aplikován zbytek po
celo¬íselném d¬lení
EXP        //mocnina
SIGNEXTEND //Zv¬tší délku celého ¬ísla se znaménkem zapsaného ve
dvojkovém dopl¬ku
SHA3       //Spo¬te Keccak-256 haš bloku pam¬ti
```

Všimněte si, že veškerá aritmetika se provádí modulo 2^{256} (pokud není uvedeno jinak) a že nula na nultou, 0^0, je považována za 1.

Operace zásobníku

Instrukce pro správu zásobníku, paměti a úložiště:

```
POP     //Odebere položku z vrcholu zásobníku
MLOAD   //Na¬te slovo z pam¬ti
MSTORE  //Uloží slovo do pam¬ti
MSTORE8 //Uloží byte do pam¬ti
SLOAD   //Na¬te slovo z úložišt¬
SSTORE  //Uloží slovo do úložišt¬
MSIZE   //Vrátí velikost aktivní pam¬ti v bytech
PUSHx   //Vloží x-bytovou položku do zásobníku, kde x m¬že být celé
¬íslo
        // od 1 do 32 (plné slovo) v¬etn¬
DUPx    //Zdvojí x-tou položku zásobníku, kde x m¬že být celé ¬íslo
        // od 1 do 16 v¬etn¬
SWAPx   //Vym¬ní první a (x+1)-ní položku zásobníku, kde x m¬že být
        // celé ¬íslo od 1 to 16 v¬etn¬
```

Operace řízení toku

Instrukce pro řízení toku

```
STOP      //Ukon¬í provád¬ní
JUMP      //Nastaví programový ¬íta¬ na libovolnou hodnotu
JUMPI     //Podmín¬n¬ nastaví programový ¬íta¬
PC        //Vrátí hodnotu ¬íta¬e instrukcí, p¬ed zvýšením
          //odpovídajícímu provedení této instrukce
JUMPDEST  //Ozna¬í platný cíl pro skoky
```

Systémové operace

Instrukce systém vykonávající program

```
LOGx          //P¬ipojí x záznam¬ do protokolu, kde x je celé ¬íslo
              //od 0 do 4 v¬etn¬
CREATE        //Vytvo¬í nový ú¬et s p¬i¬azeným zdrojovým kódem
CALL          //Volání zprávy jiným ú¬tem, nap¬. spušt¬ní kódu
              //jiného ú¬tu
CALLCODE      //Vyvolání zprávy do tohoto ú¬tu pomocí jiného ú¬tu
              //jiného ú¬tu
RETURN        //Zastavení provád¬ní a vrácení výstupních dat
DELEGATECALL  //Volání zprávy do tohoto ú¬tu s alternativním
              //zdrojovým kódem ú¬tu, ale zachováním sou¬asných
              //hodnot odesilatele a hodnoty
STATICCALL    //Statické volání zpráv do ú¬tu
REVERT        //Provád¬ní zastavení, vrácení zm¬n stavu, ale vrácení
              //dat a zbývajícího plynu
INVALID       //Úmyslná neplatná instrukce
SELFDESTRUCT  //Zastavení provád¬ní a ozna¬ení ú¬tu k odstran¬ní
```

Logické operace

Instrukce pro porovnání a bitovou logiku:

```
LT     //Porovnání menší než
GT     //Porovnání v¬tší než
SLT    //Porovnání menší než pro ¬ísla se znaménkem
SGT    //Porovnání v¬tší než pro ¬ísla se znaménkem
EQ     //Porovnání rovnosti
ISZERO //Logický operátor negace (NOT)
AND    //Operace bitové konjunkce (AND)
OR     //Operace bitové negace (OR)
XOR    //Operace bitové exkuluzivní disjunkce (XOR)
NOT    //Operace bitové negace (NOT)
BYTE   //Na¬te jeden bajt z 256-bitového slova (plné ší¬ky)
```

Operace prostředí

Instrukce zabývající se informacemi o prostředí provádění:

```
GAS            //Vrátí množství dostupného plynu (po snížení za
               //tuto instrukci)
ADDRESS        //Vrátí adresu aktuáln¬ provád¬jícího ú¬tu
BALANCE        //Vrátí z¬statek na zadaném ú¬tu
ORIGIN         //Vrátí adresu EOA, která zahájila toto EVM
               //vykonávání EVM
CALLER         //Vrátí adresu bezprost¬edn¬ odpov¬dného volajícího
               //za toto vykonání EVM
CALLVALUE      //Vrátí množství etheru uloženého odpov¬dným volajícím
               //za toto vykonání EVM
CALLDATALOAD   //Vrátí vstupní data zaslaná volajícím odpov¬dným
               //za toto vykonání
CALLDATASIZE   //Vrátí velikost vstupních dat
CALLDATACOPY   //Okopíruje vstupní data do pam¬ti
CODESIZE       //Vrátí velikost kódu b¬žícího v tomto prost¬edí
CODECOPY       //Okopíruje kód b¬žící v tomto prost¬edí do
               //pam¬ti
GASPRICE       //Vrátí cenu plynu ur¬enou po¬áte¬ní
               //transakcí
EXTCODESIZE    //Vrátí velikost kódu ú¬tu
EXTCODECOPY    //Okopíruje kód ú¬tu do pam¬ti
RETURNDATASIZE //Vrátí velikost výstupních dat z p¬edchozího volání
               //v sou¬asném prost¬edí
RETURNDATACOPY //Okopíruje výstupní data p¬edchozího volání do pam¬ti
```

Blokové operace

Instrukce pro přístup k informacím o aktuálním bloku:

```
BLOCKHASH  //Vrátí haš jednoho z 256 posledních blok¬
           //blo¬enky
COINBASE   //Vrátí adresu p¬íjemce odm¬ny za vytvo¬ení bloky
TIMESTAMP  //Vrátí ¬asovou zna¬ku bloku
NUMBER     //Vrátí ¬íslo bloku
DIFFICULTY //Vrátí obtížnost bloku
GASLIMIT   //Vrátí omezení na množství plynu v bloku
```

Ethereum stav

Úkolem EVM je aktualizovat stav Etherea výpočtem platných přechodů stavu v důsledku provádění kódu chytrého kontraktu, jak je definováno Ethereum protokolem. Tento aspekt vede k popisu Ethereum jako *transakčního stavového stroje*, což odráží skutečnost, že externí aktéři (tj. majitelé účtů a těžaři) iniciují přechody stavu vytvářením, přijímáním a zařazováním transakcí. V tomto bodě je užitečné zvážit, co představuje stav Etherea.

Na nejvyšší úrovni máme *světový stav* Ethereua. Světový stav je mapování Ethereum adres (160-bitové hodnoty) na *účty.accounts*. Na spodní úrovni každá Ethereum adresa představuje účet obsahující *zůstatek* etheru (uložený jako počet wei ve vlastnictví účtu), *nonci* (představující počet transakcí úspěšně odeslaných z tohoto účtu, pokud jde o EOA, nebo počet kontraktů vytvořených tímto účtem. Účty chytrých kontraktů navíc obsahují *úložiště* účtu (což jsou trvale uložené údaje) a *zdrojový kód* účtu. EOA nemá nikdy zdrojový kód a má vždy prázdné úložiště.

Když transakce vyústí v provedení kódu chytrého kontraktu, EVM je inicializován všemi požadovanými informacemi ve vztrahu k aktuálně vytvářenému bloku a konkrétní právě vytvářené transakci. Zejména je programový ROM kód EVM načten s kódem volaného účtu kontraktu, čítač programu je nastaven na nulu, úložiště je načteno z úložiště účtu kontraktu, paměť je zcela vynulována a všechny proměnné prostředí a bloku jsou nastaveny. Klíčovou proměnnou je zásoba plynu pro toto provedení, která je nastavena na množství plynu zaplacené odesílatelem na začátku transakce (viz Plyn pro více informací). Jak postupuje provádění kódu, je zásoba plynu snižována v závislosti na nákladech na plyn na provedené operace. Pokud je v kterémkoli okamžiku zásoba plynu snížena na nulu, dostaneme výjimku „došel plyn“ (Out of Gas; OOG); provádění se okamžitě zastaví a transakce je přerušena. Stav Etherea se nezmění, s výjimkou zvýšení nonce odesílatele a snížení zůstatku etheru, aby se tvůrci bloku zaplatily prostředky použité k provedení kódu do okamžiku jeho zastavení. V tomto okamžiku si můžete představit, že EVM běží na kopii světového stavu Etherea v karanténě, přičemž změny dosažené v této karanténě je zcela anulovány, pokud provedení nelze z jakéhokoli důvodu dokončit. Pokud se však provádění úspěšně dokončí, aktualizuje se skutečný stav tak, aby odpovídal stavu karantény, včetně jakýchkoli změn v datech úložiště volaného kontraktu, všech vytvořených nových kontraktů a všech iniciovaných převodů zůstatku etheru.

Všimněte si, protože chytrý kontrakt může sám účinně iniciovat transakce, je provádění kódu rekurzivním procesem. Kontrakt může zavolat jiné kontrakty přičemž každé volání má za následek, že další EVM karanténa je vytvořena kolem nového cíle volání. Každá instance má svůj světový stav v karanténě inicializovaný z karantény EVM na vyšší úrovni volání. Každé instanci je také dáno

určité množství plynu jako její zásoba plynu (samozřejmě nepřekračující množství plynu zbývajícího nad vyšší úrovní volání), a tak se může sama zastavit s výjimkou kvůli příliš malému množství plynu, menšímu než je nutné k dokončení jeho provádění . V takových případech je opět stav karantény smazán a provádění se vrací ke keranténě EVM na vyšší úrovni volání.

Kompilace Solidity EVM bajtkódu

Kompilaci Solidity zdrojového kódu do bajtkódu EVM lze provést několika způsoby. V Základy Etherea jsme použili online překladač Remix. V této kapitole použijeme spustitelný soubor solc na příkazové řádce. Chcete-li zobrazit seznam možností, spusťte následující příkaz:

```
<pre data-type="programlisting">
$ <strong>solc --help</strong>
</pre>
```

Vytváření surového proudu instrukcí ze Solidity zdrojového kódu lze snadno dosáhnout pomocí volby příkazového řádku --opcodes . Tento proud instrukcí vynechává některé informace (volba --asm poskytuje úplné informace), ale pro tuto diskusi je to dostačující. Například kompilace vzorového souboru Solidity *Example.sol* a odeslání výstupních instrukcí do adresáře s názvem *BytecodeDir* se provede pomocí následujícího příkazu:

```
<pre data-type="programlisting">
$ <strong>solc -o BytecodeDir --opcodes Example.sol</strong>
</pre>
```

nebo:

```
<pre data-type="programlisting">
$ <strong>solc -o BytecodeDir --asm Example.sol</strong>
</pre>
```

Následující příkaz vytvoří binární bajtkód pro náš příkladový program:

```
<pre data-type="programlisting">
$ <strong>solc -o BytecodeDir --bin Example.sol</strong>
</pre>
```

Vytvářené výstupní soubory instrukčního kódu budou záviset na konkrétních kontraktech obsažených v Solidity zdrojovém souboru. Náš jednoduchý Solidity soubor *Example.sol* má pouze jeden kontrakt s názvem example:

```
pragma solidity ^0.4.19;

contract example {

  address contractOwner;

  function example() {
    contractOwner = msg.sender;
  }
}
```

Jak vidíte, celý tento kontrakt obsahuje jednu trvalou stavovou proměnnou, která je nastavena jako adresa posledního účtu, který tento kontrakt spouští.

Pokud se podíváte do adresáře *BytecodeDir*, uvidíte soubor instrukcí *example.opcode*, který obsahuje EVM instrukce kontraktu example. Otevřením souboru *example.opcode* v textovém editoru se zobrazí následující:

```
PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE ISZERO PUSH1 0xE JUMPI PUSH1 0x0
DUP1
REVERT JUMPDEST CALLER PUSH1 0x0 DUP1 PUSH2 0x100 EXP DUP2 SLOAD DUP2
PUSH20
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF MUL NOT AND SWAP1 DUP4 PUSH20
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND MUL OR SWAP1 SSTORE POP
PUSH1
0x35 DUP1 PUSH1 0x5B PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN STOP PUSH1 0x60
PUSH1
0x40 MSTORE PUSH1 0x0 DUP1 REVERT STOP LOG1 PUSH6 0x627A7A723058 KECCAK256
JUMP
0xb9 SWAP14 0xcb 0x1e 0xdd RETURNDATACOPY 0xec 0xe0 0x1f 0x27 0xc9 PUSH5
0x9C5ABCC14A NUMBER 0x5e INVALID EXTCODESIZE 0xdb 0xcf EXTCODESIZE 0x27
EXTCODESIZE 0xe2 0xb8 SWAP10 0xed 0x
```

Kompilace příkladu s volbou --asm vytvoří v našem adresáři *BytecodeDir* soubor s názvem *example.evm*. Obsahuje mírně vyšší úroveň instrukcí bajtkódu EVM spolu s některými užitečnými popisy:

```
/* "Example.sol":26:132  contract example {... */
  mstore(0x40, 0x60)
    /* "Example.sol":74:130  function example() {... */
  jumpi(tag_1, iszero(callvalue))
  0x0
  dup1
  revert
tag_1:
    /* "Example.sol":115:125  msg.sender */
  caller
    /* "Example.sol":99:112  contractOwner */
  0x0
  dup1
    /* "Example.sol":99:125  contractOwner = msg.sender */
  0x100
  exp
  dup2
  sload
  dup2
  0xffffffffffffffffffffffffffffffffffffffff
  mul
```

```
  not
  and
  swap1
  dup4
  0xffffffffffffffffffffffffffffffffffffffff
  and
  mul
  or
  swap1
  sstore
  pop
    /* "Example.sol":26:132  contract example {... */
  dataSize(sub_0)
  dup1
  dataOffset(sub_0)
  0x0
  codecopy
  0x0
  return
stop

sub_0: assembly {
        /* "Example.sol":26:132  contract example {... */
      mstore(0x40, 0x60)
      0x0
      dup1
      revert

    auxdata:
0xa165627a7a7230582056b99dcb1edd3eece01f27c9649c5abcc14a435efe3b...
}
```

Volba --bin-runtime vytváří strojově čitelný hexadecimální bajtkód:

```
60606040523415600e57600080fd5b336000806101000a81548173
ffffffffffffffffffffffffffffffffffffffff
021916908373
ffffffffffffffffffffffffffffffffffffffff
16021790555060358060 5b6000396000f3006060604052600080fd00a165627a7a72305820
56b...
```

Podrobně si můžete prohlédnout, co se zde děje, pomocí seznamu instrukcí uvedených vInstrukční sada EVM (bajtkódové operace). Je to však docela úkol, takže začněme prozkoumáním prvních čtyř instrukcí:

```
PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE
```

Zde máme PUSH1 následovaný surovým bajtem hodnoty 0x60. Tato instrukce EVM vezme jeden bajt za kódem instrukce v programovém kódu (jako doslovnou hodnotu) a vloží jej do zásobníku. Je možné do zásobníku posunout hodnoty velikosti až 32 bajtů, jako v:

```
PUSH32 0x436f6e67726174756c6174696f6e732120536f6f6e20746f206d617374657221
```

Druhá instrukce PUSH1 z *example.opcode* ukládá 0x40 na na vrchol zásobníku (nad hodnotu 0x60, která byla do teď jediným prvkem zásobníku a tedy na jeho vrcholu a nyní bude jednu pozici pod vrcholem.

Další je MSTORE, což je operace ukládání do paměti, která ukládá hodnotu do paměti EVM. Vyžaduje dva parametry a stejně jako většina operací EVM je získá ze zásobníku. Výběr každého z těchto parametrů ze zásobníku probíhá následujícím způsobem. Je odebrána horní hodnota zásobníku (vrchol) a všechny ostatní hodnoty v zásobníku jsou posunuty o jednu pozici nahoru. Prvním parametrem pro MSTORE je adresa slova v paměti, kam bude vložena hodnota, která má být uložena. Pro tento program máme ve vrcholu zásobníku 0x40, takže je ze zásobníku odebrána a použita jako adresa paměti. Druhým argumentem je hodnota, která se má uložit, což je zde 0x60. Po provedení operace MSTORE je náš zásobník znovu prázdný, ale máme v hodnotu 0x60 (desítkově 96) v paměti na adrese 0x40.

Další instrukcí je CALLVALUE, což je instrukce prostředí, která vloží na vrchol zásobníku množství etheru (měřeno ve wei) odeslané pomocí volání zprávy, které zahájila toto vykonání.

Mohli bychom pokračovat v tomto programu tímto způsobem, dokud jsme si plně neuvědomili změny stavu na nízké úrovni, které tento kód způsobí, ale v této fázi by nám to nepomohlo. Vrátíme se k němu později v kapitole.

Kód nasazení kontraktu

Mezi kódem použitým při vytváření a zavádění nového kontraktu na platformě Ethereum a kódem samotného kontraktu existuje významný, ale jemný rozdíl. K vytvoření nového kontraktu je nutná zvláštní transakce, jejíž pole to je nastaveno na speciální adresu 0x0 a pole data je nastaveno na *zdrojový kód vytvoření kontraktu*. Místo toho je EVM inicializován s kódem v poli data transakce načtené do jeho programového kódu ROM a poté je výstup provádění tohoto nasazovacího kódu považován za kód pro nový účet kontraktu. Je to tak, že nové kontrakty lze programově nastavit pomocí světového státu Ethereum v době nasazení, stanovením hodnot v úložišti kontraktu a dokonce odesláním etheru nebo vytvořením dalších nových kontraktů.

Při kompilaci kontraktu offline, např. Pomocí solc na příkazové řádce, můžete získat *bajtkód nasazení* nebo *běhový bajtkód*.

Bajtkód nasazení se používá pro všechny aspekty počátečního nastavení nového účtu kontraktu, včetně bajtkódu, který bude skutečně zavolán, když transakce budou volat tento nový kontrakt (tj. běhový bajtkód) a kód pro počáteční nastavení všeho na základě konstruktoru kontraktu.

Běhový bajtkód, na druhé straně, je přesně bajtkód, který je vykonán po zavolání nového kontraktu, a nic víc; nezahrnuje bajtkód potřebný k počátečnímu nastavení kontraktu během jeho nasazení.

Vezměme si jako příklad jednoduchý kontrakt *Faucet.sol*, který jsme vytvořili dříve:

```
// Verze kompilátoru Solidity, pro který byl tento program napsán
pragma solidity ^0.4.19;

// Our first contract is a faucet!
contract Faucet {

  // Vydá ether každému, kdo o to požádá
  function withdraw(uint withdraw_amount) public {

      // Maximální povolená ¬ástka pro výb¬r
      require(withdraw_amount <= 100000000000000000);

      // Zašlete ¬ástku na adresu, která o ni požádala
      msg.sender.transfer(withdraw_amount);
    }

  // P¬ijm¬te jakoukoliv p¬íchozí ¬ástku
  function () external payable {}

}
```

Pro získání bajtkódu nasazení bychom spustili `solc --bin Faucet.sol`. Pokud bychom místo toho chtěli pouze běhový bajtkód, spustili by jsme `solc --bin-runtime Faucet.sol`.

Pokud porovnáte výstup těchto příkazů, uvidíte, že běhový bajtkód je podmnožinou bajtkódu nasazení. Jinými slovy, běhový bajtkód je zcela obsažen v bajtkódu nasazení.

Rozklad bajtkódu na assembler

Rozklad bajtkódu EVM je skvělý způsob, jak pochopit, jak vysokoúrovňové Solidity působí v EVM. K tomu můžete použít několik dekompilátorů (disassembler):

- *Porosity* [https://github.com/comaeio/porosity] je populární dekompilátor s otevřeným zdrojovým kódem.
- *Ethersplay* [https://github.com/trailofbits/ethersplay] je EVM zásuvný modul pro dekompilátor Binary Ninja.
- *IDA-Evm* [https://github.com/trailofbits/ida-evm] je EVM zásuvný modul pro další dekompilátor IDA.

V této sekci budeme používat zásuvný modul Ethersplay pro Binary Ninja a začneme [Faucet_disassembled]. Po získání běhového bajtkódu *Faucet.sol* můžeme ho vložit do Binary Ninja (po načtení zásuvného modulu Ethersplay) a podívat se, jak vypadají instrukce EVM.

Rozklad Faucet.sol na běhový bajtkód image::images/Faucet_disassembled.png["Rozklad Faucet.sol na běhový bajtkód"]

Když odešlete transakci na chytrý kontrakt kompatibilní s ABI (lze předpokládat, že to splňují všechny kontrakty), transakce nejprve interaguje s *dispečerem* tohoto chytrého kontraktu. Dispečer přečte transakční pole data a odešle příslušnou jeho část příslušné funkci. Na začátku našeho rozebraného běhového bajtkódu *Faucet.sol* vidíme příklad dispečera. Po známé instrukci MSTORE vidíme následující instrukce:

```
PUSH1 0x4
CALLDATASIZE
LT
PUSH1 0x3f
JUMPI
```

Jak jsme viděli, PUSH1 0x4 umístí 0x4 na vrchol zásobníku, který je do té doby prázdný. CALLDATASIZE získá bajtovou velikost dat odeslaných transakcí (známé jako *data volání*) a toto číslo vloží do zásobníku. Po provedení těchto instrukcí vypadá zásobník takto:

Stack
<length of calldata from tx>
0x4

Tato další instrukce je LT, zkratka pro „ménší než“. Instrukce LT zkontroluje, zda je horní položka v zásobníku menší než další položka v zásobníku. V našem případě zkontroluje, zda je výsledek CALLDATASIZE menší než 4 bajty.

Proč EVM kontroluje, zda data volání transakce jsou nejméně 4 bajty? Kvůli mechanizmu identifikátorů funkcí. Každá funkce je identifikována prvními 4 bajty Keccak-256 haše. Použitím názvu funkce a parametrů jako vstupu hašovací funkce keccak256, můžeme odvodit její identifikátor funkce. V našem případě máme:

```
keccak256("withdraw(uint256)") = 0x2e1a7d4d...
```

Identifikátor funkce pro funkci withdraw(uint256) je tedy 0x2e1a7d4d, protože se jedná o první 4 bajty výsledného haše. Identifikátor funkce je vždy 4 bajty dlouhý, takže pokud celé pole data transakce odeslané do kontraktu je menší než 4 bajty, neexistuje žádná funkce, s níž by transakce mohla komunikovat, pokud není definována *nozouvá* funkce. Protože jsme implementovali takovou nouzovou funkci v *Faucet.sol*, EVM skočí na tuto funkci, když je délka dat volání menší než 4 bajty.

LT vyzvedne ze zásobníku dvě horní hodnoty, a pokud je pole dat + transakce menší než 4 bajty, vloží na vrchol zásobníku +1. Jinak vloží na vrchol zásobníku 0. V našem příkladu předpokládejme, že pole data transakce zaslané našemu kontraktu_ bylo menší než 4 bajty.

Instrukce PUSH1 0x3f vloží bajt 0x3f na vrchol zásobníku. Po této instrukci vypadá zásobník takto:

Stack
0x3f
1

Další instrukce je JUMPI, což je zkratka pro "podmíněný skok". Funguje to takto:

```
jumpi(label, cond) // Sko¬ na místo programu ozna¬ené "náv¬štím" pokud
"podmínka" je vyhodnocena jako pravdivá
```

V našem případě návěští je 0x3f, což je místo, kde zíje naše nouzová funkce v našem chytrém kontraktu. Parametr podmínka + je +1, což byl výsledek předchozí instrukce LT. Chcete-li tuto celou posloupnost vyjádřit slovy, smlouva skočí na nouzovou funkci, pokud jsou transakční data menší než 4 bajty.

Na 0x3f následuje pouze instrukce STOP, protože ačkoli jsme deklarovali nouzovou funkci, ponechali jsme ji prázdnou. Jak můžete vidět v JUMPI instrukce vedoucí k nouzové funkci, neimplementovali jsme záložní funkci, kontrakt by měl místo toho vyvolat výjimku.

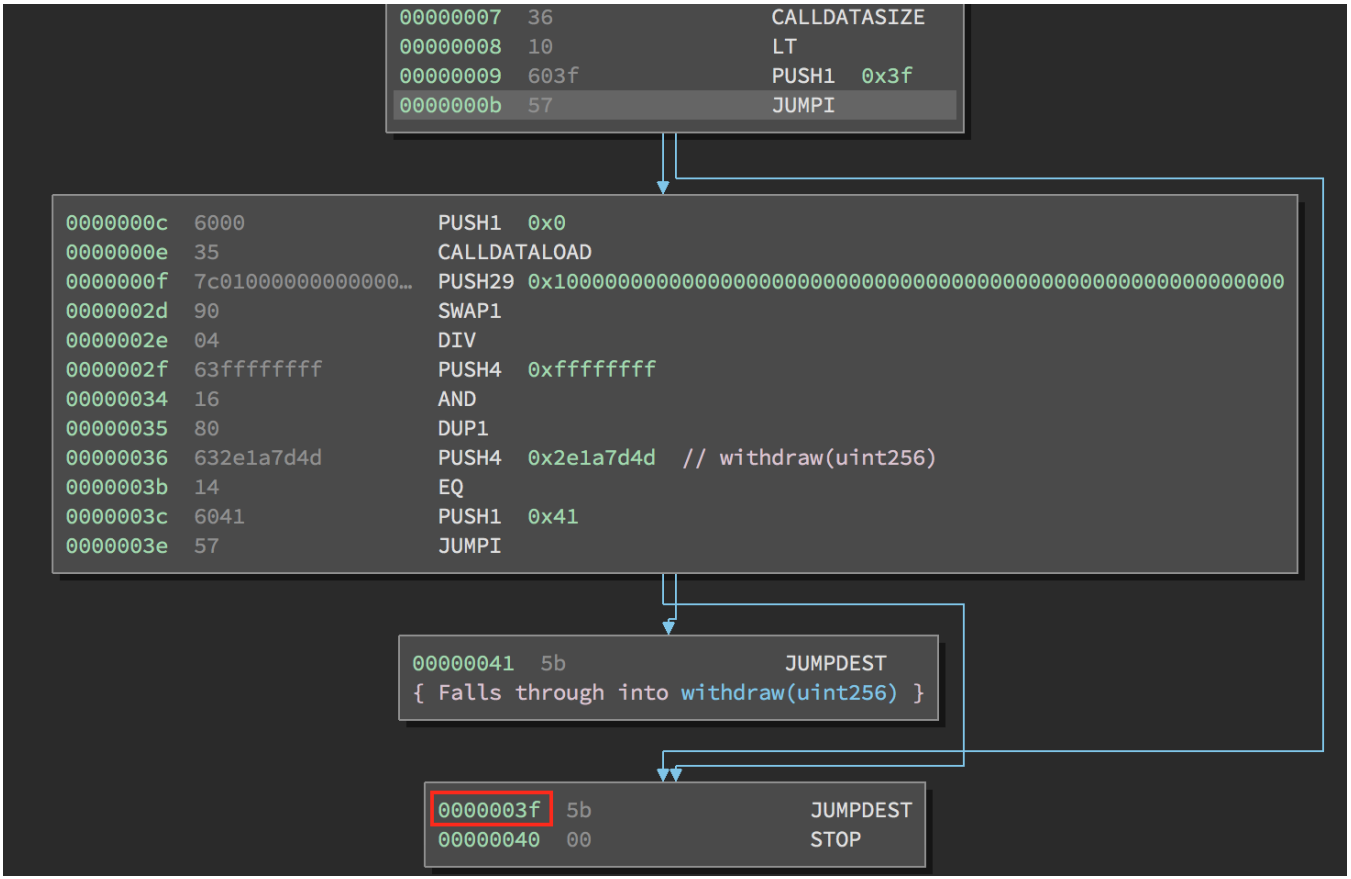

Figure 56. JUMPI instrukce vedoucí k nouzové funkci

Podívejme se na centrální blok dispečera. Za předpokladu, že jsme obdrželi data volání, která byla *větší* delší než 4 bajty, instrukce JUMPI by neskočila na záložní funkci. Místo toho by provádění kódu pokračovalo podle následujících instrukcí:

```
PUSH1 0x0
CALLDATALOAD
PUSH29 0x1000000...
SWAP1
DIV
PUSH4 0xffffffff
AND
DUP1
PUSH4 0x2e1a7d4d
EQ
PUSH1 0x41
JUMPI
```

PUSH1 0x0 vloží 0 na vrchol zásobníku zásobníku, který je nyní jinak znovu prázdný. CALLDATALOAD přijímá jako parametr index v rámci dat volání zasílaných do chytrého kontaktu a čte 32 bajtů z tohoto indexu, například:

```
calldataload(p) //na¬te 32 bajt¬ dat volání od zadané pozice p
```

Protože příkaz PUSH1 0x0 předal index 0, CALLDATALOAD přečte 32 bajtů dat volání počínaje bajtem 0 a poté je vkládá na vrchol zásobníku (po odebrání původního 0x0). Po instrukci PUSH29 0x1000000 ... je zásobník:

Stack
0x1000000... (celkem dlouhé 29 bajtů)
<32 bytes of calldata starting at byte 0>

SWAP1 prohazuje vrchol zásobníku s *i* - tým prvkem za ním. V tomto případě zaměňuje 0x1000000 ... s daty volání. Nový zásobník je:

Stack
<32 bytes of calldata starting at byte 0>
0x1000000... (celkem dlouhé 29 bajtů)

Další instrukce DIV, která pracuje následovně:

```
div(x, y) // celo¬íselné d¬lení x / y
```

V tomto případě x = 32 bajtů dat volání počínaje bajtem 0 a y = 0x100000000 ... (celkem 29 bajtů). Přemýšlíte, proč dispečer dělá dělení? Tady je nápověda: nejprve čteme 32 bajtů z dat volání, počínaje indexem 0. První 4 bajty těchto dat volání je identifikátor funkce.

Data 0x100000000..., která jsme vložili na zásobník dříve, jsou 29 bajtů dlouhá, sestávající z 1 na začátku, následovaných samými 0. Rozdělením našich 32 bajtů dat volání touto hodnotou nám zůstanou pouze _vrchní 4 bajtů načtení dat volání, počínaje indexem 0. Tyto 4 bajty - první 4 bajty v datech volání začínající indexem 0 - jsou identifikátor funkce, a tímto způsobem EVM extrahuje toto pole.

Pokud vám tato část není jasná, myslete na to takto: v desítkové soustavě, 1234000/1000 = 1234. V šestnáctkové soustavě se to neliší. Namísto toho, aby každé místo bylo násobkem 10, je to násobek 16. Stejně jako dělení 10^3 (1000) v našem menším příkladu ponechalo pouze nejvyšší číslice, přičemž naše 32-bajtové hodnota o základu 16 se vydělila 16^{29} stejným způsobem.

Výsledek DIV (identifikátor funkce) je vložen na vrchol zásobníku a náš zásobník je nyní:

Stack
<function identifier sent in data>

Protože instrukce PUSH4 0xffffffff a AND jsou nadbytečné, můžeme je zcela ignorovat, protože zásobník zůstane stejný i po jejich dokončení. Instrukce DUP1 zdvojí vrchol zásobníku, což je identifikátor funkce. Další instrukce, PUSH4 0x2e1a7d4d, vloží na vrchol zásobníku vypočítaný identifikátor funkce `withdraw(uint256)`. Zásobník je nyní:

Stack
0x2e1a7d4d
<function identifier sent in data>
<function identifier sent in data>

Další instrukce EQ vybere z vrcholu zásobníku dvě položky a porovná je. Toto je místo, kde dispečer vykonává svou hlavní úlohu: porovnává, zda se identifikátor funkce odeslaný v poli msg.data transakce shoduje s identifikátorem `withdraw(uint256)`. Pokud se shodují, EQ vloží na vrchol

zásobníku 1, která nakonec bude použita pro skok na funkci pro výběr prostředků. V opačném případě, EQ vloží na vrchol zásobníku 0.

Za předpokladu, že transakce zaslaná našemu kontraktu skutečně začala identifikátorem funkce pro withdraw(uint256), náš zásobník nyní vypadá následovně:

Stack
1
<function identifier sent in data> (nyní známá jako 0x2e1a7d4d)

Dále máme PUSH1 0x41, což je adresa, na které funkce withdraw(uint256) žije ve smlouvě. Po této instrukci vypadá zásobník takto:

Stack
0x41
1
Identifikátor funkce zaslaný v msg.data

Další je instrukce JUMPI + a jako své parametry bere opět dva horní prvky v zásobníku. V tomto případě máme +jumpi(0x41, 1), která říká EVM, aby provedl skok na místo funkce withdraw(uint256), a pokračoval v provádění kódu této funkce.

Turingova úplnost a plyn

Jak jsme se již dříve zmínili, jednoduše řečeno, systém nebo programovací jazyk je *Turingovsky úplný*, pokud dokáže spustit libovolný program. Tato schopnost však přichází s velmi důležitou výzvou: některé programy běží navždy. Důležitým aspektem je to, že nemůžeme říct, pouhým pohledem na program, zda bude běžet věčně nebo ne. Musíme skutečně projít celým během programu a počkat na jeho ukončení, abychom to zjistili. Samozřejmě, bude-li běžet věčně, budeme muset čekat věčně. Tomu se říká *problém zastavení* a pro Ethereum by to byl obrovský problém, kdyby nebyl vyřešen.

Kvůli problému zastavení hrozí, žeEthereum světový počítač bude požádán o provedení programu, který se nikdy nezastaví. Mohlo by to být náhodou nebo zlomyslností. Diskutovali jsme o tom, že

Ethereum funguje jako stroj s jedním vláknem, bez jakéhokoli plánovače, a tak pokud by se zasekl v nekonečné smyčce, znamenalo by to, že by se stal nepoužitelným.

Existuje řešení, použití plynu: pokud po provedení předepsaného maximálního množství výpočtu není provádění ukončeno, EVM provádění programu zastaví. To činí z EVM „kvazi Turingovsky úplný stroj;“ může spustit jakýkoli program, který do něj vložíte, ale dokončí ho pouze v případě, že program skončí v rámci určitého množství výpočtu. Tento limit není pevně nastaven v Ethereu. Můžete zaplatit, abyste jej zvýšili na maximum (nazývané „blokový limit plynu“) a každý může souhlasit se zvýšením tohoto maxima v průběhu času. Nicméně kdykoli existuje limit, transakce, které při provádění spotřebovávají příliš mnoho plynu, jsou zastaveny.

V následujících částech se podíváme na plyn a prozkoumáme, jak to funguje.

Plyn

Plyn je Ethereum jednotka pro měření výpočetních a úložných zdrojů potřebných k provádění akcí na Ethereum bločence. Na rozdíl od Bitcoinu, jejichž transakční poplatky zohledňují pouze velikost transakce v kilobajtech, musí Ethereum brát v úvahu každý výpočetní krok prováděný transakcemi a prováděním kódu chytrého kontraktu.

Každá operace prováděná transakcí nebo smlouvou stojí pevné množství plynu. Několik příkladů z Etereum Žluté knihy:

- Sečtení dvou čísel stojí 3 jednotky plynu
- Výpočet Keccak-256 haše stojí 30 plynu + plynu za každých 256 bitů hašovaných dat
- Zaslání transakce stojí 21 000 plynu

Plyn je klíčovou součástí Etherea a plní dvojí roli: jako nárazník mezi (volatilní) cenou Etherea a odměnou těžařům za práci, kterou vykonávají, a jako obrana proti útokům odepření služby. Aby se předešlo náhodným nebo škodlivým nekonečným smyčkám nebo jiným plýtvajícím výpočtům v síti, je odesilatel každé transakce povinen stanovit limit na výši výpočtu, za kterou je ochoten zaplatit. Plynový systém tak odradí útočníky od odesílání „spamových“ transakcí, protože musí úměrně platit za výpočetní zátěž a úložné prostředky, které spotřebovávají.

Účtování plynu během provádění

Když je k dokončení transakce potřeba EVM, je mu nejprve poskytnuta zásoba plynu rovnající se zadanému množství plynu v transakci. Každá spuštěná instrukce má stanovenou cenu plynu, a tak se zásoba plynu v EVM snižuje, jak EVM prochází programem. Před každou operací EVM zkontroluje, zda je dostatek plynu k provedení operace. Pokud není dostatek plynu, provádění se zastaví a transakce je anulována.

Pokud EVM dosáhne konce úspěšně konce provádění, bez vyčerpání plynu, jsou náklady na plyn uhrazeny těžaři jako transakční poplatek převedený na ether na základě ceny plynu uvedené v transakci:

```
poplatek t¬ža¬i = spot¬ebované množství plynu * cena plynu
```

Plyn zbývající v zásobě plynu je vrácen odesílateli, opět převeden na ether na základě ceny plynu uvedené v transakci:

```
zbývající plyn = limit plynu - spot¬ebovaný plyn
vrácený ether = zbývající plyn * cena plynu
```

Pokud transakci dojde během provádění plyn, dojde k jejímu okamžitému ukončení a je vyvolána výjimka „došel plyn“. Transakce je anulována a všechny změny stavu jsou vráceny zpět.

Přestože byla transakce neúspěšná, bude odesílateli účtován poplatek za transakci, protože těžaři již do té doby provedli výpočetní práci a musí být za to odškodněni.

Úvahy o účtování plynu

Relativní náklady na plyn při různých operacích, které může EVM provádět, byly pečlivě vybrány, aby co nejlépe chránily Ethereum bločenku před útokem. Můžete vidět podrobnou tabulku nákladů na plyn pro různé EVM instrukce v EVM instrukce a náklady na plyn.

Výpočtově náročnější operace stojí více plynu. Například provedení funkce SHA3 je 10-krát dražší (30 plynu) než operace ADD (3 jednotky plynu). Ještě důležitější je, že některé operace, například EXP, vyžadují dodatečnou platbu na základě velikosti operandu. Používání paměti EVM a ukládání dat do úložiště kontraktu v bločence také stojí plyn.

Důležitost přizpůsobení nákladů na plyn skutečným nákladům na zdroje byla prokázána v roce 2016, kdy útočník našel a využil nesoulad nákladů. Útok vytvářel transakce, které byly velmi výpočetně drahé, a přiměl Ethereum síť téměř se zastavit. Tento nesoulad byl vyřešen tvrdým rozštěpením (nazvaným „Tangerine Whistle“), které vyladilo relativní náklady na plyn.

Náklady na plyn versus cena plynu

Zatímco *náklady na pyn* jsou mírou výpočtu a skladování používaného v EVM, samotný plyn má také *cenu* měřenou v etheru. Při provádění transakce odesílatel stanoví cenu plynu, kterou je ochotni zaplatit (v etheru) za každou jednotku plynu, což trhu umožňuje rozhodnout o vztahu mezi cenou etheru a náklady na výpočetní operace (měřeno v plynu) :

```
transak¬ní poplatek = množství použitého plynu * zaplacena cena za
jednotku plynu (v ether)
```

Při tvorbě nového bloku si těžaři v síti Ethereum mohou vybrat mezi čekajícími transakcemi výběrem těch, které nabízejí zaplatit vyšší cenu za jednotku plynu. Nabízení vyšší ceny plynu proto motivuje těžaře, aby zahrnuli vaši transakci a získali ji rychlejší.

V praxi odesílatel transakce stanoví limit plynu, který je vyšší nebo roven množství plynu, které se očekává, že se použije. Je-li limit plynu nastaven na vyšší hodnotu, než je množství spotřebovaného plynu, bude odesílateli vrácena nadbytečná částka, protože těžaři dostanou zaplaceno pouze za práci, kterou skutečně vykonají.

Je důležité si ujasnit rozdíl mezi *náklady na plyn* a *cenou plynu*. Shrnutí:

- Náklady na plyn jsou počet jednotek plynu potřebných k provedení určité operace.
- Cena plynu je množství etheru, které jste ochotni zaplatit za jednotku plynu, když odešlete svou transakci do Ethereum sítě.

I když má plyn cenu, nemůže být „vlastněn“ ani „utracen“. Plyn existuje pouze uvnitř EVM, jako měřítko, kolik výpočetní práce se provádí. Odesílateli je účtován transakční poplatek v etheru, který je poté převeden na plyn pro účetnictví EVM a poté zpět na ether jako transakční poplatek zaplacený těžařům.

Záporné náklady na plyn

Ethereum podporuje vymazání použitých uložených proměnných a účtů tím, že vrátí část plynu použitého během provádění kontraktu.

V EVM jsou dvě operace se zápornými náklady na plyn:

- Smazání kontraktu (SELFDESTRUCT) je odměněno navrácením 24 000 plynu.
- Změna adresy v úložišti z nenulové hodnoty na nulovou (SSTORE[x] = 0) je odměněno vrácením 15 000 plynu.

Aby se zabránilo využívání mechanismu vrácení prostředků, je maximální náhrada za transakci stanovena na polovinu celkového množství použitého plynu (zaokrouhleno dolů).

Blokový limit plynu

Blokový limit plynu je maximální množství plynu, které může být spotřebováno všemi transakcemi v bloku, a omezuje, kolik transakcí se vejde do bloku.

Řekněme například, že máme 5 transakcí, jejichž limity plynu byly nastaveny na 30 000, 30 000, 40 000, 50 000 a 50 000. Pokud je limit plyny v bloku 180 000, pak se do jednoho bloku mohou zapojit libovolné čtyři z těchto transakcí, zatímco pátá bude muset čekat na budoucí blok. Jak již bylo řečeno, těžaři rozhodují, které transakce mají být v bloku zahrnuty. Různí těžaři pravděpodobně vyberou různé kombinace, hlavně proto, že přijímají transakce ze sítě v jiném pořadí.

Pokud se těžař pokusí zahrnout transakci, která vyžaduje více plynu, než je aktuální limit plynu v bloku, blok bude sítí odmítnut. Většina Ethereum klientů vás zastaví ve vydávání takové transakce tak, že vás varuje v řádku „transakce překračuje limit plynu v bloku“. Blokový limit plynu v Ethereum hlavní síti je v době psaní podle protokolu https://etherscan.io 8 miliónů plynu, což znamená, že do bloku se vejde zhruba 380 základních transakcí (každá spotřebovává 21 000 plynu).

Kdo rozhoduje, jaký je limit plynu v bloku?

Těžaři v síti společně rozhodují o limitu plynu v bloku. Jednotlivci, kteří chtějí těžit v síti Ethereum, používají těžební program, například Ethminer, který se připojuje k Geth nebo Parity Ethereum klientovi. Protokol Ethereum má vestavěný mechanismus, kde horníci mohou hlasovat o limitu plynu, takže kapacita může být zvýšena nebo snížena v následujících blocích. Těžař bloku může

hlasovat pro úpravu limitu plynu v bloku faktorem 1/1 024 (0,0976%) v obou směrech. Výsledkem je nastavitelná velikost bloku na základě aktuálních potřeb sítě. Tento mechanismus je spojen s výchozí těžařskou strategií, kdy těžaři hlasují o limitu plynu, který je nejméně 4,7 milionu plynu, ale který se zaměřuje na hodnotu 150% průměru průměrné nedávné celkové spotřeby plynu na blok (pomocí exponenciálního plovoucího průměru 1 024 bloků).

Závěry

V této kapitole jsme prozkoumali Ethereum virtuální stroj, sledovali provádění různých chytrých kontraktů a zkoumali, jak EVM provádí bajtkód. Také jsme se podívali na plyn, účetní mechanismus EVM, a viděli jsme, jak řeší problém zastavení a chrání Ethereum před útoky odepření služby. Dále vKonsensus se podáváme na mechanizmus použitý Ethereum k decentralizovanému dosahování konsensu.

Konsensus

V této knize jsme již hovořili o„pravidlech konsensu“; pravidlech na kterých se musí všichni shodnout, aby systém pracoval decentralizovaně, ale deterministicky. , V informatice termín *konsensus* je starší než bločenka a souvisí s širším problémem synchronizace stavu v distribuovaných systémech, takže různí účastníci v distribuovaném systému se všichni (případně) dohodnou na jediném stavu celého systému. Tomu se říká „dosažení konsensu“.

Pokud jde o základní funkci decentralizovaného uchovávání a ověřování záznamů, může být problematické spoléhat se pouze na důvěru, aby bylo zajištěno, že informace odvozené z aktualizací stavu jsou správné. Tato poměrně obecná výzva je zvláště výrazná v decentralizovaných sítích, protože neexistuje žádná centrální entita, která by rozhodovala o tom, co je pravda. Absence ústřední rozhodovací entity je jednou z hlavních výhod bločenkových platforem, a to kvůli výsledné schopnosti odolat cenzuře a nedostatečné závislosti na oprávnění k přístupu k informacím. Tyto výhody však stojí za to: bez důvěryhodného rozhodce musí být neshody, podvody nebo rozdíly sladěny jinými prostředky. Algoritmy konsensu jsou mechanismem používaným pro sladění bezpečnosti a decentralizace.

V bločenkách je konsenzus kritickou vlastností systému. Jednoduše řečeno, v sázce jsou peníze! V kontextu bločenek je tedy *konsensus* o možnosti dospět ke společnému stavu při zachování decentralizace. Jinými slovy, cílem konsensu je vytvořit systém *přísných pravidel bez vládce*. Neexistuje žádná osoba, organizace nebo skupina, moc a kontrola jsou rozptýleny po široké síti účastníků, jejichž vlastním zájmem je dodržování pravidel a chování se čestně.

Schopnost dospět ke konsensu v distribuované síti za rozporuplných podmínek bez centralizované kontroly je základním principem všech otevřených veřejných bločenek. Komunita pokračuje v experimentování s různými modely konsensu, aby vyřešila tuto výzvu a zachovala si cennou vlastnost decentralizace. Tato kapitola zkoumá tyto konsensuální modely a jejich očekávaný dopad na bločenky chytrých kontraktů, jako je Ethereum.

Zatímco konsensuální algoritmy jsou důležitou součástí fungování bločenek, fungují v základní vrstvě, daleko pod abstrakcí chytrých kontraktů. Jinými slovy, většina podrobností o konsensu se skrývá před tvůrci chytrých kontraktů. Nemusíte vědět, jak fungují, aby používali Ethereum, o nic víc, než potřebujete vědět, jak funguje směrování při používání internetu.

Konsensus pomocí důkazu prací (PoW)

Tvůrce původní bločenky, Bitcoinu, vynalezl *konsensuální algoritmus _ nazvaný _důkaz prací* (Proof of Work; PoW). PoW je pravděpodobně nejdůležitější vynález, na kterém je založen Bitcoin. Hovorovým termínem pro PoW je „těžba“, což vytváří nedorozumění ohledně primárního účelu konsensu. Lidé často předpokládají, že účelem těžby je vytvoření nové měny, protože účelem těžby v reálném světě je těžba drahých kovů nebo jiných zdrojů. Skutečným účelem těžby (a všech ostatních konsensuálních modelů) je spíše *zabezpečit bločenku* a zároveň udržet kontrolu nad systémem decentralizovanou a rozptýlenou mezi co nejvíce účastníků. Odměna nově ražené měny je pobídkou pro ty, kteří přispívají k bezpečnosti systému: prostředek k dosažení cíle. V tomto smyslu je odměna prostředkem a decentralizovaná bezpečnost je cíl. V PoW konsensu existuje také odpovídající „trest“, což jsou náklady na energii potřebné k účasti na těžbě. Pokud se účastníci neřídí pravidly a nezískají odměnu, riskují finanční prostředky, které již vynaložili na elektřinu, na těžbu. Konsenzus PoW je tedy pečlivou rovnováhou mezi rizikem a odměnou, která nutí účastníky, aby se chovali čestně z vlastního zájmu.

Ethereum je v současné době PoW bločenka v tom, že používá PoW algoritmus se stejným základním motivačním systémem pro tentýž základní cíl: zajištění bločenky při decentralizované kontrole. Algoritmus PoW Etherea se mírně liší od Bitcoinového a nazývá se *Ethash*. Budeme zkoumat funkční a návrhové charakteristiky algoritmu v Ethash: Ethereum algoritmus pro důkaz prací.

Konsensus pomocí důkazu podílem (PoS)

Historicky nebyl důkaz prací prvním navrhovaným algoritmem konsensu. Před zavedením důkazu prací mnoho vědců navrhlo varianty konsensuálních algoritmů založených na finančním podílu, nyní nazvaném *důkaz podílem* (Proof of Stake; PoS). V některých ohledech byl důkaz práce vynalezen jako alternativa k důkazu podílem. Po úspěchu Bitcoinu, mnoho bločenek využilo důkaz prací. Exploze výzkumu konsensuálních algoritmů však také vzkřísila důkaz podílem, což významně pomohlo ve stavu technologie. Zakladatelé Etherea od začátku doufali, že nakonec převedou svůj konsensuální algoritmus na důkaz podílem Ve skutečnosti existuje úmyslné znevýhodnění důkazu prací v Ethereu, nazvaného „obtížnostní bomba“, jehož cílem je dělat postupně v Ethereu těžbu pomocí důkazu prací stále obtížnější, což nutí k přechodu k důkazu podílem.

V době vydání této knihy Ethereum stále používá důkaz praci, ale probíhající výzkum alternativy důkazu podílem se blíží ke konci. Plánovaný algoritmus PoS Etherea se nazývá *Casper*. Zavedení Casperu jako náhrada za Ethash bylo v posledních dvou letech několikrát odloženo, což vyžadovalo

zásahy, které zneškodnily obtížnostní bombu a odložily nucené zastarávání důkazu práci.

Algoritmus PoS obecně funguje následovně. Bločenka sleduje množinu validátorů a každý, kdo drží základní kryptoměnu bločenky (v případě Ethereum, ether), se může stát validátorem zasláním zvláštního typu transakce, která zamkne jejich ether do vkladu. Validátoři se střídají při navrhování a hlasování o dalším platném bloku a váha hlasování každého validátora závisí na velikosti jeho vkladu (tj. podílu). Důležité je, že validátor riskuje ztrátu svého vkladu, pokud blok, na který vsadili, je odmítnut většinou validátorů. Naopak, validátoři získávají malou odměnu, úměrnou jejich vkladu, za každý blok, který přijímá většina. PoS tedy nutí validátory jednat čestně a dodržovat pravidla konsensu, a to prostřednictvím systému odměn a trestů. Hlavní rozdíl mezi PoS a PoW je v tom, že trest v PoS je vlastní bločence (např. ztráta vsázeného etheru), zatímco v PoW je trest vnější (např. ztráta finančních prostředků vynaložených na elektřinu).

Ethash: Ethereum algoritmus pro důkaz prací

Ethash je Ethereum PoW algoritmus. Používá odvozený algoritmus Dagger–Hashimoto, který je kombinací Dagger algoritmu Vitalika Buterina a Hashimotova algoritmu Thaddeus Dryja.Ethash je závislý na generování a analýze velkého souboru dat, známého jako *orientovaný acyklický graf* (nebo jednodušeji, DAG) měl počáteční velikost asi 1 GB a bude se i nadále pomalu a lineárně zvětšovat, přičemž velikost se aktualizuje jednou za každou epochu (30 000 bloků nebo zhruba 125 hodin).

Účelem použití DAG je, aby byl PoW algoritmus Ethash závislý na udržování velké, často přístupné datové struktury. To má zase za cíl učinit Ethash „odolným vůči ASIC“, což znamená, že je obtížnější vyrobit zařízení *aplikačně specifické integrované obvody* (ASIC), která jsou řádově rychlejší než rychlá *grafická zpracovatelská jednotka* (GPU). Zakladatelé Etherea se chtěli vyhnout centralizaci těžby PoW, kde by těžební infrastruktura mohla pomoci v dominanci těm, kteří mají přístup ke specializovaným výrobním závodům na výrobu křemíku a velkým rozpočtům, a podkopávat bezpečnost algoritmu konsensu.

Použití GPU na uživatelské úrovni pro provádění PoW v Ethereum síti znamená, že do procesu těžby se může zapojit více lidí po celém světě. Čím více jsou těžaři nezávislí, tím decentralizovanější je těžební síla, což znamená, že se můžeme vyhnout situaci jako v Bitcoinu, kde je velká část těžební síly soustředěna do rukou několika velkých průmyslových těžebních skupin. Nevýhodou používání GPU pro těžbu je to, že v roce 2017 vyvolalo celosvětový nedostatek GPU, což způsobilo, že jejich cena prudce stoupla a vyvolala stížnosti hráčů. To vedlo k omezení nákupu u maloobchodníků, což omezovalo kupující na jednoho nebo dva GPU na zákazníka.

Až donedávna hrozba ASIC těžařů v Ethereum síti téměř neexistovala. Použití ASIC pro Ethereum vyžaduje návrh, výrobu a distribuci vysoce přizpůsobeného hardwaru. Jejich výroba vyžaduje značné investice času a peněz. Dlouho vyjádřené plány vývojářů společnosti Ethereum přejít na algoritmus konsensu PoS pravděpodobně zabránily dodavatelům ASIC zaměřit se na síť Ethereum po dlouhou dobu. Jakmile se Ethereum přesune na PoS, ASIC určené pro algoritmus PoW se stanou zbytečnými - to znamená, pokud je těžaři nemohou použít k těžbě jiných kryptoměn. Druhá možnost je nyní realitou s řadou dalších dostupných konvenčních mincí založených na Ethashu, jako jsou PIRL a Ubiq, a Ethereum Classic, které se zavázaly, že v dohledné budoucnosti zůstane PoW bločenkou. To znamená, že pravděpodobně uvidíme, že se ASIC těžba začne stávat silou v Ethereum síti, zatímco bude stále fungovat na základě konsensu.

Casper: Ethereum algoritmus důkazu podílem

Casper je navrhovaný název pro Ethereum algoritmus konsensu PoS. Je stále aktivně zkoumán a vyvíjen a není implementován na Ethereum bločence v době vydání této knihy.Casper se vyvíjí ve dvou konkurenčních „příchutích“:

- Casper FFG: "The Friendly Finality Gadget" (přátelské koncové zařízení)
- Casper CBC: "The Friendly GHOST/Correct-by-Construction" (Přátelský DUCH/Správná konstrukce)

Zpočátku byl Casper FFG navržen jako hybridní PoW / PoS algoritmus, který má být implementován jako přechod k trvalejšímu „čistému PoS“ algoritmu. V červnu 2018 se Vitalik Buterin, který vedl výzkumné práce na Casper FFG, rozhodl „sešrotovat“ hybridní model ve prospěch čistě PoS algoritmu. Nyní jsou Casper FFG a Casper CBC vyvíjeny paralelně. Jak vysvětluje Vitalik:

> Hlavním rozdílem mezi FFG a CBC je to, že CBC vypadá, že má hezčí teoretické vlastnosti, ale zdá se, že FFG je snadnější implementovat.

Více informací o historii společnosti Casper, probíhajícím výzkumu a budoucích plánech naleznete na následujících odkazech:

- Ethereum Casper (důkaz podílem) [http://bit.ly/2RO5HAl]
- Casper historie, část 1 [http://bit.ly/2FlBojb]
- Casper historie, část 2 [http://bit.ly/2QyHiic]

- Casper historie, část 3 [http://bit.ly/2JWWFyt]
- Casper historie, část 4 [http://bit.ly/2FsaExI]
- Casper historie, část 5 [http://bit.ly/2PPhhOv]

Principy konsensu

Principy a předpoklady konsensuálních algoritmů lze lépe pochopit položením několika klíčových otázek:

- Kdo může změnit minulost a jak? (Toto je také známé jako *neměnitelnost*.)
- Kdo může změnit budoucnost a jak? (Toto je také známé jako *konečnost*.)
- Jaké jsou náklady na provedení takových změn?
- Jak decentralizovaná je pravomoc provádět takové změny?
- Kdo bude vědět, jestli se něco změnilo, a jak to bude vědět?

Algoritmy konsensu se rychle vyvíjejí a snaží se odpovědět na tyto otázky stále inovativnějším způsobem.

Spor a soutěž

V tuto chvíli by vás zajímalo: Proč potřebujeme tolik různých konsensuálních algoritmů? Který z nich funguje lépe? Odpověď na poslední otázku je středem nejzajímavější oblasti výzkumu distribuovaných systémů za poslední desetiletí. To vše se scvrkává na to, co považujete za „lepší“, což je v kontextu počítačové vědy o předpokladech, cílech a nevyhnutelných kompromisech.

Je pravděpodobné, že žádný algoritmus nemůže optimalizovat napříč všemi dimenzemi problému decentralizovaného konsensu. Když někdo navrhne, že jeden konsensuální algoritmus je „lepší“ než ostatní, měli byste začít klást otázky, které objasňují: Lepší v čem? Neměnitelnost, konečnost, decentralizace, náklady? Na tyto otázky neexistuje jasná odpověď, alespoň dosud. Kromě toho je návrh konsensuálních algoritmů středem odvětví s miliardami dolarů a vyvolává obrovské kontroverze a žhavé argumenty. Nakonec nemusí existovat „správná“ odpověď, stejně jako mohou existovat různé odpovědi pro různé aplikace.

Celý bločenkový průmysl je obrovským experimentem, ve kterém budou tyto otázky testovány za

rozporuplných podmínek s obrovskou peněžní hodnotou v sázce. Nakonec historie rozsoudí tento spor.

Závěry

Ethereův konsensuální algoritmus je v době dokončení této knihy stále v pohybu. V budoucím vydání pravděpodobně přidáme další podrobnosti o Casper a dalších souvisejících technologiích, jak vyzrajou a budou nasazeny na Ethereum. Tato kapitola představuje konec naší cesty a dokončení *Mastering Ethereum*. Další referenční materiál je uveden v dodatcích. Děkujeme vám za přečtení této knihy a blahopřeji k dosažení konce!

Appendix A: Historie Ethereum rozštěpení

Většina tvrdých rozštěpení je plánována jako součást plánu vylepšení a sestává z aktualizací, se kterými komunita obecně souhlasí (tj. existuje společenská shoda). Některá tvrdá rozštěpení však nejsou výsledkem konsensu, což vede k několika odlišným bločenkám. Události, které vedly k rozdělení Ethereum / Ethereum Classic, jsou jedním z takových případů a jsou diskutovány v této příloze.

Ethereum Classic (ETC)

Ethereum Classic vzniklo poté, co členové Ethereum komunity implementovali časově citlivé tvrdé rozštěpení (kódově nazvané DAO;). Dne 20. července 2016, v bloku výšky 1,92 milionu, zavedlo Ethereum nepravidelnou změnu stavu prostřednictvím tvrdého rozštěpení ve snaze vrátit přibližně 3,6 milionu etheru, který byl vybrán z chytrého kontraktu známého jako The DAO. Téměř všichni souhlasili s tím, že odebraný éter byl ukraden a že ponechání všeho v rukou zloděje by bylo na úkor rozvoje ekosystému Ethereum i samotné platformy.

Vrácení etheru příslušným vlastníkům, jako by The DAO nikdy neexistovalo, bylo technicky snadné, i když spíše politicky kontroverzní. Řada lidí v ekosystému nesouhlasí s touto změnou a věří, že neměnitelnost by měla být základním principem Ethereum bločenky bez výjimky; rozhodli se pokračovat v původní bločence pod názvem Ethereum Classic. Zatímco samotné rozdělení bylo původně ideologické, oba řetězce se od té doby vyvinuly do samostatných entit.

Decentralizovaná autonomní organizace (The DAO)

The The DAO kontrakt byl vytvořen Slock.it s cílem poskytovat komunitní financování a správu projektů. Hlavní myšlenkou bylo, že návrhy budou předloženy, kurátoři budou řídit návrhy, budou získávány finanční prostředky od investorů v komunitě Ethereum, a pokud by se projekty osvědčily, investoři by získali podíl na zisku.

The DAO byl také jedním z prvních experimentálních tokenů v Ethereu. Namísto financování projektů přímo s etherem by účastníci obchodovali se svým etherem za tokeny DAO, použili je k hlasování o financování projektu a později je mohli vyměnit za ether.

The DAO tokeny byly k dispozici k zakoupení ve veřejném prodeji, který probíhal od 5. do 30. dubna 2016 a shromáždil téměř 14% [https://econ.st/2qfJO1g] z celkového existujícího etheru, v té době v

hodnotě ~ 150 miliónů USD.

Chyba vícenásobného volání

Dne 9. června 2016 vývojáři Peter Vessenes a Chriseth oznámili, že většina kontraktů na bázi Etherea, které spravují finanční prostředky, byla potenciálně zranitelná [http://bit.ly/2AAaDmA], že tyto kontrakty mohou ztratit finanční prostředky. O několik dní později, 12 června, Stephen Tual (spoluzakladatel Slock.it) oznámil, že The DAO's kód není zranitelný [http://bit.ly/2qmo3g1] chybou popsanou Peterem a Chrisethem. Znepokojení investoři do The DAO vydechli úlevou, nicméně o pět dní později neznámý útočník začal vysávat The DAO [http://bit.ly/2Q7zR1h] pomocí využití chyby podobné té, pro kterou bylo varování vydáno. Nakonec útočník vysál z The DAO ~ 3,6 milionu etherů.

Zároveň shromáždění dobrovolníků, kteří si říkali Skupina Robina Hooda (Robin Hood Group; RHG), začalo používat stejnou chybu k výběru zbývajících prostředků, aby je zachránilo před ukradením DAO útočníkem. Dne 21. června RHG oznámila [http://bit.ly/2PtX4xl], že mají zajištěno asi 70% z fondů DAO (zhruba 7,2 milionu etheru), s plány vrátit je do komunity (což úspěšně udělali na síti ETC a po rozštěpení nemuseli dělat na Ethereum síti). RHG dostalo mnoho poděkování a vyznamenání za jejich rychlé přemýšlení a rychlé akce, které pomohly zajistit většinu ehteru komunity.

Technické detaily

Zatímco podrobnější a důkladnější vysvětlení chyby je dáno Phil Daian [http://bit.ly/2EQaLCI], krátké vysvětlení je, že zásadní funkce v The DAO měla dva řádky kódu v nesprávném pořadí, což znamená, že útočník mohl požádat o výběr etheru opakovaně, než byla dokončena kontrola, zda má útočník nárok na výběr Tento typ chyby zabezpečení je popsán v části Opětovné zavolání.

Průběh útoku

Představte si, že jste na svém bankovním účtu měli 100 USD a mohli byste své bankovce přinést libovolný počet výběrových lístků. Pokladník vám dá peníze za každý tiket v pořádku, a teprve po zpracování všech tiketů zaznamená váš výběr. Co když jim přinesete tři lístky, z nichž každý požaduje výběr 100 $? Co kdybys jim přinesl tři tisíce lístků?

The DAO útok probíhal následovně:

1. DAO útočník požádal kontrakt The DAO o výběr DAO tokenů (DAO).
2. Útočník požádal The DAO *znovu* o výběr, dříve než kontrakt aktualizoval záznamy, že byl proveden výběr.
3. Útočník opakoval krok 2 tolikrát, kolikrát bylo možné.
4. Kontrakt The DAO nakonec zaznamenal jeden výběr, ztrácí přehled o výběrech, které se mezitím staly.

The DAO tvrdé rozštěpení

Naštěstí bylo v The DAO zabudováno několik ochranných opatření: zejména všechny žádosti o výběr byly předmětem 28-denního zpoždění. Komunitě to dalo čas k diskuzi o tom, co s využitím chyby udělat, protože zhruba od 17. června do 20. července nebude útočník DAO schopen převést své DAO tokeny na ether.

Několik vývojářů se zaměřilo na nalezení životaschopného řešení a v tomto krátkém časovém období bylo prozkoumáno několik cest. Mezi nimi bylo *DAO měkké rozštěpení* [http://bit.ly/2qhruEK], ohlášené 24. června, aby odložilo výběr DAO až do dosažení konsensu, a *DAO tvrdé rozštěpení* [http://bit.ly/2AAGjIu], oznámené 15. července, které zvrátilo účinky DAO útoku výjimečnou změnou stavu.

Dne 28. června vývojáři objevili možnost DoS útoku na DAO měkkém rozštěpení [http://bit.ly/2zgOxUn] a dospěli k závěru, že DAO tvrdé rozštěpení bude jedinou schůdnou možností, jak situaci plně vyřešit. DAO tvrdé rozštěpení přenese veškerý ether, který byl do DAO investován, do nového chytrého kontraktu na vrácení prostředků investorům, což umožní původním majitelům etheru požadovat plné náhrady. To poskytlo řešení pro vrácení odcizených prostředků, ale také to znamenalo zásah do zůstatků konkrétních adres v síti, i když byly izolované. V částech DAO, které se nazývají *childDAOs*, by také existoval nějaký zbylý ether. Skupina správců by ručně povolila zbylý ether v hodnotě ~$6–7 milionů [http://bit.ly/2RuUrJh] v té době.

Po nějakém čase několik Ethereum vývojových týmů vytvořilo klienty, kteří uživateli umožnily rozhodnout se, zda chtějí toto rozštěpení povolit. Tvůrci klientů se však chtěli dohodnout, zda tuto volbu nastavit jako doporučenou (ve výchozím nastavení není rozštěpení) nebo nedoporučenou (ve výchozím nastavení je rozštěpení). Dne 15. července bylo zahájeno hlasování *carbonvote.com* [http://bit.ly/2ABkTuV]. Další den, v bloku číslo 1,894,000 [http://bit.ly/2yHb7Gl], bylo dokončeno. Volilo pouze 5.5% celkového množství etherů [http://bit.ly/2RuUrJh], 80% hlasů (4.5% celkového množství

etherů) volilo pro rozštěpení jako doporučené nastavení. Jedna čtvrtina z těchto hlasů pocházela z jediné adresy.

Nakonec se rozhodnutí stalo doporučeným, takže ti, kteří se postavili proti tvrdému rozštěpení DAO, museli výslovně vyjádřit svou opozici změnou možnosti konfigurace v softwaru, který provozovali.

1. července v bloku číslo 1,920,000 [http://bit.ly/2zfaIKB], Ethereum implementovalo tvrdé rozštěpení DAO [http://bit.ly/2yJxZ83] a tak byly vytvořeny dvěEthereum sítě: jedna přijala změny stavu a druhá ji odmítla.

Když Ethereum s tvrdým rozštěpením (dnešní Ethereum) získalo většinu těžební síly, mnozí předpokládali, že bylo dosaženo konsensu a menšinový řetězec zmizí, jako v předchozích rozštěpeních. Navzdory tomu začala značná část komunity Ethereum (zhruba 10% hodnoty a těžební síly) podporovat bločenku bez tvrdého rozštěpení, která se stal známá jako Ethereum Classic.

Během několika dnů od rozštěpení začalo několik burz uvádět jak Ethereum („ETH“), tak Ethereum Classic („ETC“). Vzhledem k povaze tvrdých rozštěpení všichni uživatelé Etherea, kteří v době rozštěpení drželi ether, pak drželi prostředky na obou bločenkách a brzy byla stanovena tržní hodnota pro ETC pomocí Poloniex [http://bit.ly/2qhuNvP] uvedl ETC 24. července.

Časová osa tvrdého rozštěpení DAO

- 5. dubna 2016: Slock.it představuje přehled zabezpečení [http://bit.ly/2Db4boE] obecného rámce DAO chytrých kontraktů od Deja Vu Security.
- 30. dubna 2016: veřejný prodej The DAO spuštěn [http://bit.ly/2qhwhpI].
- 27. května 2016: The DAO veřejný prodej končí.
- 9. června 2016: Byl objevena obecná chyba rekurzivního volání [http://bit.ly/2AAaDmA] a předpokládá se, že ovlivňuje mnoho Solidity kontraktů, které uchovávají zůstatky uživatelů.
- 12. června 2016: Stephen Tual prohlásil [http://bit.ly/2qmo3g1], že finanční prostředky The DAO nejsou ohroženy.
- 17. června 2016: The DAO byl napaden [http://bit.ly/2EQaLCI] variantou objevené chyby (nazývané „chyba vícenásobného vstupu“), bylo zahájeno vysávání finančních prostředků, nakonec zmizelo ~ 30% z etheru.
- 21. června 2016: RHG oznámila [http://bit.ly/2zgl3Gk] zajištění dalších ~ 70% etheru uloženého v

DAO.

- 24. června 2016: hlasování o měkkém rozštěpenií [http://bit.ly/2qhruEK] je vyhlášeno prostřednictvím volitelné nepřednastavené signalizace prostřednictvím klientů Geth a Parity, jehož účelem je dočasně zadržet prostředky, dokud se komunita nemůže lépe rozhodnout, co dělat.
- 28. června 2016: zranitelnost [http://bit.ly/2zgOxUn] je objevena na měkkém rozštěpení a je opuštěno.
- 28. června 2016 do 15. července: Uživatelé debatují o tom, zda se má provést tvrdé rozštěpení, většina veřejné debaty o hlasování se odehrává na fóru reddit v tématu */r/ethereum*
- 15. července 2016: Je navrženo tvrdé rozštěpení DAO [http://bit.ly/2qmo3g1], aby se vrátily prostředky získané při útoku DAO.
- 15. července 2016: koná se hlasování [http://bit.ly/2ABkTuV] na CarbonVote, aby se rozhodlo, zda bude tvrdé rozštěpení DAO nepřednastavené (ve výchozím nastavení není rozštěpení) nebo přednastavené (ve výchozím nastavení je rozštěpení).
- 16. července 2016: 5.5% z celkového počtu celkového etheru [http://bit.ly/2RuUrJh]; ~80% hlasů (~4.5% z celkového množství) hlasuje pro přednastavené tvrdé rozštěpení, jedna čtvrtina souhlasných hlasů pochází z jedné adresy.
- 20. července 2016: tvrdé rozštěpení [http://bit.ly/2yJxZ83] se vyskytuje v bloku 1 920 000.
- 20. července 2016: Ti, kteří jsou proti tvrdému rozštěpení DAO, pokračují v používání starého klientského softwaru; to vede k problémům s přeposílání transakcí na obou bločenkách [http://bit.ly/2qjJm27].
- 24. července 2016: Poloniex uvádí [http://bit.ly/2qhuNvP] původní bločenku Ethereum pod symbolem ETC; je to první burza, která to dělá.
- 10. srpna 2016: RHG převádí 2,9 [http://bit.ly/2JrLpK2] milionu zpětně získaných ETC na Poloniex za účelem jejich převodu na ETH na radu Bity SA; 14% z celkového objemu RHG je převedeno z ETC na ETH a další kryptoměny a Poloniex zmrazí [http://bit.ly/2ETDdUc] zbývajících 86% uloženého ETH.
- 30. srpna 2016: Zmrazené finanční prostředky jsou společností Poloniex zaslány zpět do RHG, která poté vytvoří kontrakt pro navrácení prostředků v bločence ETC.
- 11. prosince 2016: Vytvoření vývojového týmu ETC IOHK, vedeného zakládajícím členem Etherea Charlesem Hoskinsonem.

- 13. ledna 2017: Síť ETC je aktualizována, aby vyřešila problémy s opakováním prováděním transakcí; bločenky jsou nyní funkčně oddělené.
- 20. února 2017: Vytvořen ETCDEVTeam vedený raným vývojářem ETC Igorem Artamonovem (splix).

Ethereum and Ethereum Classic

Zatímco počáteční rozdělení bylo soustředěno kolem The DAO, dvě sítě, Ethereum a Ethereum Classic, jsou nyní oddělené projekty, ačkoli většina vývoje je stále prováděna komunitou Etherea a jednoduše portována do zdrojových kódů Etherea Classic. Celý soubor rozdílů se však neustále vyvíjí a je příliš rozsáhlý na to, aby byl pokryty v této příloze. Je však třeba poznamenat, že bločenky se výrazně liší v jejich hlavním vývoji a struktuře komunity. Dále je popsáno několik technických rozdílů.

The EVM

Z velké části (v době psaní) zůstávají obě sítě vysoce kompatibilní: kód kontraktu vytvořený pro jednu bločenku běží podle očekávání na straně druhé; ale existují malé rozdíly v EVM instrukcích(viz EIPs 140 [http://bit.ly/2yIajkF], 145 [http://bit.ly/2qhKz9Y], a 214 [http://bit.ly/2SxsrFR]).

Vývoj základní sítě

Protože jsou bločenkové platformy otevřené projekty, mají často mnoho uživatelů a přispěvatelů. Vývoj jádrové sítě (tj. kódu, který provozuje síť) je však často prováděn malými skupinami kvůli odbornosti a znalostem potřebným pro vývoj tohoto typu softwaru. Pokud jde o Ethereum, tuto práci provádí Nadace Ethereum a dobrovolníci. Na Ethereum Classic to provádí ETCDEV, IOHK a dobrovolníci.

Další významná Ethereum rozštěpení

Ellaism [https://ellaism.org/about/] je síť založená na Ethereu, která hodlá používat k zabezpečení bločenky výhradně PoW. Nemá žádné předtěžení a žádné povinné poplatky pro vývojáře, veškerou podporu a vývoj poskytuje komunita zdarma. Jeho vývojáři se domnívají, že to dělá jejich, jeden z nejpřímějších čistých Ethereum projektů a ten, který je jedinečně zajímavý jako platforma pro seriózní vývojáře, pedagogy a nadšence. Ellaismus je čistě platforma chytrých kontraktů. Jeho cílem

je vytvořit platformu chytrých kontraktů, která bude spravedlivá a důvěryhodná. Principy platformy jsou následující:

> - Všechny změny a rozšíření protokolu by se měly snažit udržovat a posilovat tyto principy Ellaism.
> - Měnová politika: 280 milionů mincí.
> - Žádná cenzura: Nikdo by neměl být schopen zabránit tomu, aby byly platné transakce potvrzeny.
> - Open-Source: Zdrojový kód Ellaism by měl být vždy otevřený pro kohokoli, aby ho mohl číst, upravovat, kopírovat, sdílet.
> - Nevyžadující povolení: Žádní libovolní strážci by nikdy neměli zabránit nikomu být součástí sítě (uživatel, uzel, těžař atd.).
> - Pseudo-anonymní: neměla by být vyžadována žádná identifikace pro vlastnění a používání Ellaism.
> - Zaměnitelné: Všechny mince jsou stejné a měly by být stejně utratitelné.
> - Nevratné transakce: Potvrzené bloky by měly být vytesány do kamene. Historie bločenky by měla být neměnná.
> - Žádná sporná tvrdá rozštěpení: Nikdy neprovádět tvrdá rozštěpení bez konsensu celé komunity. Stávající konsenzus přerušit pouze v případě nezbytné potřeby.
> - Mnoho vylepšení funkcí lze provádět bez tvrdého rozštěpení, jako je zlepšení výkonu EVM.

Na Ethereu se také objevilo několik dalších rozštěpení. Některé z nich jsou tvrdými rozštěpeními v tom smyslu, že se oddělují přímo od existující sítě Etherea. Jiné jsou softwarová rozštěpení: používají klientský / uzlový software Ethereum, ale provozují zcela oddělené sítě bez historie sdílené s Ethereem. Během života Etherea bude pravděpodobně více rozštěpení.

Existuje také několik dalších projektů, které prohlašují, že se jedná o rozštěpení Etherea, ale ve skutečnosti jsou založeny na ERC20 tokenech a běží na síti Ethereum.Dva příklady z nich jsou EtherBTC (ETHB) a Ethereum Modification (EMOD). Nejedná se o rozštěpení v tradičním slova smyslu a někdy je lze nazvat „výsadky“ (airdrop).

Zde je stručné shrnutí některých z pozoruhodnějších rozštěpení, která se vyskytla:

- *Expanse* bylo první rozštěpení Ethereum bločenky, která získala ohlas. Bylo oznámeno prostřednictvím fóra Bitcoin Talk 7. září 2015. Skutečné rozštěpení se objevilo o týden později 14. září 2015 v bloku číslo 800 000. Původně jej založili Christopher Franko a James Clayton. Jejich stanovenou vizí bylo vytvořit pokročilou bločenku pro: „identitu, vládnutí, charitu, obchod a rovnost“.
- *EthereumFog* (ETF) byl spuštěn 14. prosince 2017 a vidličkou byl v bloku číslo 4 730 660. Cílem projektu je vyvinout „světovou decentralizovanou výpočetní mlhovinu“ se zaměřením na výpočetní mlhovinu a decentralizované úložiště. Stále existuje jen málo informací o tom, co to ve skutečnosti bude.
- *EtherZero* (ETZ) byl zahájen 19. ledna 2018 v bloku číslo 4 936 270. Mezi jeho významné inovace patřilo zavedení architektury hlavních uzů a odstranění transakčních poplatků za chytré kontrakty, které umožnily širší rozmanitost DApps. Někteří významní členové komunity Etherea, MyEtherWallet a MetaMask, projekt kritizovali kvůli nedostatečné jasnosti okolního vývoje a některým obviněním z možné snahy o získávání citlivých údajů uživatelů.
- *EtherInc* (ETI) byl zahájen 13. února 2018 v bloku číslo 5 078 585 se zaměřením na budování decentralizovaných organizací. Stanovené cíle zahrnují zkrácení doby tvorby bloku, zvýšení odměn pro těžaře, odstranění strýčkových (ommer) odměn a stanovení stropu na vytěžitelné mince. EtherInc používá stejné soukromé klíče jako Ethereum a implementoval ochranu proti opakování transakce, aby ochránil ether v původní bločence bez rozštěpení.

Appendix B: Ethereum Standardy

Návrhy na vylepšení Etherea (EIP)

Úložiště Návrhů na vylepšení Etherea (Ethereum Improvement Proposal) se nachází na https://github.com/ethereum/EIPs/. Pracovní postup je znázorněn v Pracovní postup návrhu na vylepšení Etherea.

Z EIP-1 [https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1.md]:

> EIP je zkratka pro návrh na vylepšení Etherea. EIP je konstrukční dokument poskytující informace komunitě Etherea nebo popisující novou vlastnost pro Ethereum nebo jeho procesy nebo prostředí. EIP by měl poskytnout stručnou technickou specifikaci prvku a zdůvodnění tohoto prvku. Autor EIP je zodpovědný za budování konsensu v komunitě a za dokumentování nesouhlasných názorů.

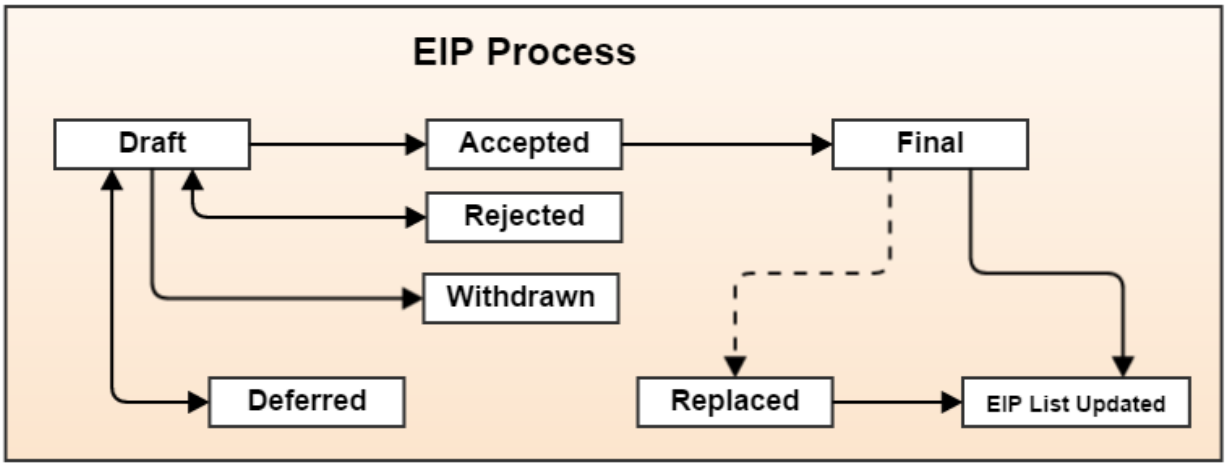

Figure 57. Pracovní postup návrhu na vylepšení Etherea

Nejvýznamnější EIP a ERC

Table 9. Důležité EIP a ERC

EIP/ERC #	Název/Popis	Autor	Vrstva	Status	Vytvořeno
EIP-1 [http://bit.ly/2OVq6qa]	Účel EIP a pokyny	Martin Becze, Hudson Jameson	Meta	Final	
EIP-2 [http://bit.ly/2yJtTNa]	Homestead tvrdé rozštěpení	Vitalik Buterin	Core	Final	
EIP-5 [http://bit.ly/2Jrx93V]	Použití plynu pro `RETURN` a `CALL*`	Christian Reitwiessner	Core	Draft	
EIP-6 [http://bit.ly/2OYbc2t]	Přejmenování instrukce SUICIDE	Hudson Jameson	Interface	Final	
EIP-7 [http://bit.ly/2JxdBeN]	`DELEGATECALL`	Vitalik Buterin	Core	Final	
EIP-8 [http://bit.ly/2Q6Oly6]	devp2p Požadavky zpětné kompatibility pro Homestead	Felix Lange	Networking	Final	

EIP/ERC #	Název/Popis	Autor	Vrstva	Status	Vytvořeno
EIP-20 [http://bit.ly/2CUf7WG]	ERC-20 standard tokenů. Popisuje standardní funkce, které může implementovat tokenový kontrakt, aby umožnil DApps a peněženkám zpracovávat tokeny přes více rozhraní / DApps. Metody zahrnují: `totalSupply`, balanceOf(address), `transfer`, `transferFrom`, `approve`, `allowance`. Události zahrnují: `Transfer` (triggered when tokens are transferred), `Approval` (spuštěno při zavolání `approve`).	Fabian Vogelsteller, Vitalik Buterin	ERC	Final	Frontier

EIP/ERC #	Název/Popis	Autor	Vrstva	Status	Vytvořeno
EIP-55 [http://bit.ly/2Q6R4YB]	Smíšená velikost písmen využitá jako kontrolní součet adresy	Vitalik Buterin	ERC	Final	

EIP/ERC #	Název/Popis	Autor	Vrstva	Status	Vytvořeno
EIP-86 [http://bit.ly/2OgE5la]	Abstrakce původu transakce a podpisu. Nastavuje fázi „abstrakce“ zabezpečení účtu a umožnění uživatelům vytvářet „účtu kontraktů“, směřující k modelu, kde v dlouhodobém horizontu jsou všechny účty kontrakty, které mohou platit za plyn, a uživatelé mohou volně definovat své vlastní bezpečnostní modely, které provádějí jakékoli požadované ověřování podpisů a kontroly nonce (namísto použití mechanismu v protocol, kde	Vitalik Buterin	Core	Deferred (to be replaced)	Constantinople

EIP/ERC #	Název/Popis	Autor	Vrstva	Status	Vytvořeno
EIP-96 [http://bit.ly/2QedSFC]	Haš bloku a změny stavu kořene. Ukládá haše blok§ ve stavu, aby se snížila složitost protokolu a potřeba komplexních klientských implementací pro zpracování operačního kódu BLOCKHASH. Rozšiřuje rozsah toho, jak daleko může kontrola zpětného haše bloku jít, s vedlejším účinkem vytváření přímých vazeb mezi bloky s velmi vzdálenými čísly bloků, aby se usnadnila mnohem efektivnější počáteční synchronizace odlehčeného klienta.	Vitalik Buterin	Core	Deferred	Constantinople

EIP/ERC #	Název/Popis	Autor	Vrstva	Status	Vytvořeno
EIP-100 [http://bit.ly/2AC05DM]	Změňte nastavení obtížnosti tak, aby bylo dosaženo cílové střední doby tvorby bloku včetně strýců (ommer).	Vitalik Buterin	Core	Final	Metropolis Byzantinium

EIP/ERC #	Název/Popis	Autor	Vrstva	Status	Vytvořeno
EIP-101 [http://bit.ly/2Jr1zDv]	Serenity Currency a Crypto Abstraction. Abstraktce etheru na vyšší úroveň s výhodou umožňující, aby se s etherem a závislými tokeny zacházelo podobně jako s kontrakty, snížila se úroveň přesměrování vyžadovaná pro účty vlastní politiky, jako jsou vícepodpisové účty, a čistí základní protokol Ethereum snížením minimální složitosti implementace konsensu.	Vitalik Buterin	Active	Serenity feature	Serenity Casper

EIP/ERC #	Název/Popis	Autor	Vrstva	Status	Vytvořeno
EIP-105 [http://bit.ly/2Q5sdEv]	Binární střepy (sharding) plus kontrakt volající sémantiku. "Lešení pro střepy“ (EIP), který umožňuje paralelizaci transakcí Etherea pomocí binárního mechanismu střepů a připravuje půdu pro pozdější systém střepů. Probíhá výzkum; viz https://github.com/ethereum/sharding.	Vitalik Buterin	Active	Serenity feature	Serenity Casper
EIP-137 [http://bit.ly/2yG2Dzi]	Ethereum doménová jmenná služba - specifikace	Nick Johnson	ERC	Final	

EIP/ERC #	Název/Popis	Autor	Vrstva	Status	Vytvořeno
EIP-140 [http://bit.ly/2yJtWZm]	Nová instrukce: REVERT. Přidává instrukci REVERT, který zastaví provádění a vrátí zpět změny stavu provádění EVM, aniž by spotřebovala veškerý poskytovaný plyn (místo toho kontrakt musí platit pouze za paměť) nebo ztratil protokoly, a vrátí volajícímu ukazatel na paměťové místo s chybovým kódem nebo zprávou.	Alex Beregszaszi, Nikolai Mushegian	Core	Final	Metropolis Byzantinium
EIP-141 [http://bit.ly/2CQMXfe]	Navržena neplatná instrukce EVM	Alex Beregszaszi	Core	Final	
EIP-145 [http://bit.ly/2qhKz9Y]	Operace bitových posunů v EVM	Alex Beregszaszi, Paweł Bylica	Core	Deferred	

EIP/ERC #	Název/Popis	Autor	Vrstva	Status	Vytvořeno
EIP-150 [http://bit.ly/2qhxflQ]	Změny nákladů na plyn pro vstupně výstupní operace	Vitalik Buterin	Core	Final	
EIP-155 [http://bit.ly/2CQUgne]	Jednoduchá ochrana proti útoku opětovnému volání funkce. Replay Attack umožňuje, aby se každá transakce používající uzel nebo klienta Ethereum před EIP-155 stala podepsanou, takže je platná a prováděná v obou bločenkách Ethereum a Ethereum Classic.	Vitalik Buterin	Core	Final	Homestead
EIP-158 [http://bit.ly/2JryBmT]	Čištění stavu	Vitalik Buterin	Core	Superseded	
EIP-160 [http://bit.ly/2CR6VGY]	Zvýšení náklaů na instrukci EXP	Vitalik Buterin	Core	Final	

EIP/ERC #	Název/Popis	Autor	Vrstva	Status	Vytvořeno
EIP-161 [http://bit.ly/2OfU96M]	Čištění stavu trie (invariantně zachovaná alternativa)	Gavin Wood	Core	Final	
EIP-162 [http://bit.ly/2JxdKil]	Registrátor počátečního ENS haše	Maurelian, Nick Johnson, Alex Van de Sande	ERC	Final	
EIP-165 [http://bit.ly/2OgsOkO]	ERC-165 Detekce standardního rozhraní	Christian Reitwiessner et al.	Interface	Draft	
EIP-170 [http://bit.ly/2OgCWu1]	Omezení velikosti kódu kontraktu	Vitalik Buterin	Core	Final	
EIP-181 [http://bit.ly/2ERNv7g]	ENS podpora pro zpětný překlad Ethereum adres	Nick Johnson	ERC	Final	
EIP-190 [http://bit.ly/2P0wPz5]	Standard balíčkování Ethereum Smart chytrých kontraktů	Piper Merriam et al.	ERC	Final	

EIP/ERC #	Název/Popis	Autor	Vrstva	Status	Vytvořeno
EIP-196 [http://bit.ly/2SwNQiz]	Předkompilované kontrakty na sčítání a skalární násobení na eliptické křivce alt_bn128. Vyžadováno za účelem ověření zkSNARK v rámci blokového limitu plynu.	Christian Reitwiessner	Core	Final	Metropolis Byzantinium
EIP-197 [http://bit.ly/2ETDC9a]	Předkompilované kontrakty pro optimální kontrolu ate párování na eliptické křivce alt_bn128. V kombinaci s EIP-196.	Vitalik Buterin, Christian Reitwiessner	Core	Final	Metropolis Byzantinium
EIP-198 [http://bit.ly/2DdTCRN]	Velké celočíselné modulární násobení. Předkompilace umožňující ověření podpisu RSA a další kryptografické aplikace.	Vitalik Buterin	Core	Final	Metropolis Byzantinium

EIP/ERC #	Název/Popis	Autor	Vrstva	Status	Vytvořeno
EIP-211 [http://bit.ly/2qjYJr3]	Nové instrukce: `RETURNDATASIZE` a `RETURNDATACOPY`. Přidává podporu pro vracení hodnot s proměnnou délkou uvnitř EVM s jednoduchým nabíjením plynem a minimální změnou na volající instrukce pomocí nových instrukcí `RETURNDATASIZE` a `RETURNDATACOPY`. Zpracovává podobně jako existující `calldata`, přičemž po volání jsou vrácená data uchovávána ve virtuální vyrovnávací paměti, ze které jej může volající	Christian Reitwiessner	Core	Final	Metropolis Byzantinium

EIP/ERC #	Název/Popis	Autor	Vrstva	Status	Vytvořeno
EIP-214 [http://bit.ly/2OgV0Eb]	Nová instrukce: `STATICCALL`. Povoluje nestabilní volání sama sebe nebo na jiného kontraktu, zatímco znemožňuje jakékoli změny stavu během volání (a jeho závislých volání, pokud jsou přítomny), aby se zvýšila bezpečnost chytrých kontraktů a zajistilo vývojářům, že z volání nemohou vzniknout chyby vícenásobného volání. Zavolá dítě s příznakem „STATIC“ nastaveným na „pravda“ pro provedení dítěte, což způsobí	Vitalik Buterin, Christian Reitwiessner	Core	Final	Metropolis Byzantinium

EIP/ERC #	Název/Popis	Autor	Vrstva	Status	Vytvořeno
EIP-225 [http://bit.ly/2JssHlJ]	Rinkeby testnet používající důkaz autoritou, kde bloky jsou těženy pouze důvěryhodnými podepisovači.	Péter Szilágyi			Homestead
EIP-234 [http://bit.ly/2yPBavd]	Přidá `blockHash` do JSON-RPC nastavení filtrů	Micah Zoltu	Interface	Draft	
EIP-615 [http://bit.ly/2yKrBNM]	Podprogramy a statické skoky pro EVM	Greg Colvin, Paweł Bylica, Christian Reitwiessner	Core	Draft	
EIP-616 [http://bit.ly/2AzGX99]	SIMD instrukce pro EVM	Greg Colvin	Core	Draft	
EIP-681 [http://bit.ly/2qjYX1n]	URL formát pro transakční požadavky	Daniel A. Nagy	Interface	Draft	
EIP-649 [http://bit.ly/2OYgE5n]	Metropolis odložení obtížnostní bomby o 1 rok a snížení odměny za vytvoření bloku z 5 na 3 ethery.	Afri Schoedon, Vitalik Buterin	Core	Final	Metropolis Byzantinium

EIP/ERC #	Název/Popis	Autor	Vrstva	Status	Vytvořeno
EIP-658 [http://bit.ly/2RoGCvH]	Vložení kódu stavu transakce do účtenek. Načte a vloží stavové pole indikující stav úspěchu nebo neúspěchu na účtenkách transakcí pro volající, protože již není možné předpokládat, že transakce selhala, a to pouze tehdy, pokud spotřebovala veškerý plyn po zavedení instrukce „REVERT“ v EIP-140.	Nick Johnson	Core	Final	Metropolis Byzantinium
EIP-706 [http://bit.ly/2Ogwpzs]	DEVp2p úhledná komprese	Péter Szilágyi	Networking	Final	

EIP/ERC #	Název/Popis	Autor	Vrstva	Status	Vytvořeno
EIP-721 [http://bit.ly/2AAkCIP]	ERC-721 Standard nezaměnitelných tokenů. Standardní API, které umožňuje, aby chytré kontrakty fungovaly jako jedinečné obchodovatelné nezaměnitelné tokeny (NFT), které mohou být sledovány ve standardizovaných peněženkách a obchodovány na burzách jako aktiva s hodnotou, podobná ERC20. CryptoKitties byla první populárně zavedená implementace digitálního NFT v ekosystému Ethereum.	William Entriken, Dieter Shirley, Jacob Evans, Nastassia Sachs	Standard	Draft	

EIP/ERC #	Název/Popis	Autor	Vrstva	Status	Vytvořeno
EIP-758 [http://bit.ly/2qmuDmJ]	Předplatné a filtry pro dokončené transakce	Jack Peterson	Interface	Draft	
EIP-801 [http://bit.ly/2RnqlHy]	ERC-801 Kanárský standard	ligi	Interface	Draft	

EIP/ERC #	Název/Popis	Autor	Vrstva	Status	Vytvořeno
EIP-827 [http://bit.ly/2DdTKkf]	ERC827 standard tokenů. Rozšíření standardního rozhraní ERC20 pro tokeny s metodami, které umožňují provádění volání uvnitř transfer+ a schvalování. Tento standard poskytuje základní funkce pro přenos tokenů a také umožňuje tokenům, aby byly schváleny, aby mohly být utraceny jinou třetí stranou v řetězci. Rovněž umožňuje vývojáři provádět volání při převodech a schvalování.	Augusto Lemble	ERC	Draft	

EIP/ERC #	Název/Popis	Autor	Vrstva	Status	Vytvořeno
EIP-930 [http://bit.ly/2Jq2hAM]	ERC930 Věčné úložiště. Kontrakt ES (Eternal Storage) je vlastněn adresou, která má oprávnění k zápisu. Úložiště je veřejné, což znamená, že každý má oprávnění ke čtení. Ukládá data do mapování pomocí jednoho mapování podle typu proměnné. Použití tohoto kontraktu umožňuje vývojáři snadno přenést úložiště do jiného kontraktu, pokud je to nutné..	Augusto Lemble	ERC	Draft	

Appendix C: Instrukce Ethereum EVM a spotřeba plynu

Tento dodatek je založen na konsolidaci provedené lidmi https://github.com/trailofbits/evm-opcodes jako reference pro instrukční informace Ethereum VM (EVM) licencované podle Apache License 2.0 [http://bit.ly/2zfrv0b].

Table 10. EVM instrukce a náklady na plyn

Instrukce	Název	Popis	Dodatečné informace	Plyn
`0x00`	STOP	Zastaví běh	-	0
`0x01`	ADD	Operace sčítání	-	3
`0x02`	MUL	Operace násobení	-	5
`0x03`	SUB	Operace odečítání	-	3
`0x04`	DIV	Operace celočíselného dělení	-	5
`0x05`	SDIV	Operace znaménkového celočíselného dělení (oříznutá)	-	5
`0x06`	MOD	Operace zbytku po celočíselném dělení	-	5
`0x07`	SMOD	Operace zbytku po znaménkovém celočístelném dělení	-	5
`0x08`	ADDMOD	Operace sčítání následovaná zbytkem po dělení	-	8
`0x09`	MULMOD	Operace násobení následovaná zbytkem po dělení	-	8

Instrukce	Název	Popis	Dodatečné informace	Plyn
`0x0a`	EXP	Operace mocnění	-	10***
`0x0b`	SIGNEXTEND	Dvojkový doplněk znaménkového celého čísla prodloužené délky	-	5
`0x0c` - `0x0f`	Nepoužito	Nepoužito	-	
`0x10`	LT	Porovnání menší než	-	3
`0x11`	GT	Porovnání větší než	-	3
`0x12`	SLT	Znaménkové porovnání menší než	-	3
`0x13`	SGT	Znaménkové porovnání větší než	-	3
`0x14`	EQ	Porovnání na rovnost	-	3
`0x15`	ISZERO	Jednoduchý operátor negace NOT	-	3
`0x16`	AND	Operace bitové konjunkce AND	-	3
`0x17`	OR	Operace bitové disjunkce OR	-	3
`0x18`	XOR	Operace bitové exkluzivní disjunkce XOR	-	3
`0x19`	NOT	Operace bitové negace NOT	-	3

Instrukce	Název	Popis	Dodatečné informace	Plyn
`0x1a`	BYTE	Načte jeden bajt ze slova	-	3
`0x1b` - `0x1f`	Nepoužito	Nepoužito	-	
`0x20`	SHA3	Spočte Keccak-256 haš	-	30
`0x21` - `0x2f`	Nepoužito	Nepoužito	-	
`0x30`	ADDRESS	Vrátí adresu právě prováděného účtu	-	2
`0x31`	BALANCE	Vrátí zůstatek daného účtu	-	400
`0x32`	ORIGIN	Vrátí adresu tvůrce transakce	-	2
`0x33`	CALLER	Vrátí adresu volajícího	-	2
`0x34`	CALLVALUE	Vrátí uloženou hodnotu instrukce/transakce odpovědné za toto vykonání	-	2
`0x35`	CALLDATALOAD	Vrátí vstupní data současného prostředí	-	3
`0x36`	CALLDATASIZE	Vrátí velikost vstupních dat v současném prostředí	-	2
`0x37`	CALLDATACOPY	Zkopíruje vstupní data ze současného prostředí do paměti	-	3

Instrukce	Název	Popis	Dodatečné informace	Plyn
`0x38`	CODESIZE	Vrátí velikost běžícího kódu v současném prostředí	-	2
`0x39`	CODECOPY	Okopíruje běžící kód současného prostředí do paměti	-	3
`0x3a`	GASPRICE	Vrátí cenu plynu v současném prostředí	-	2
`0x3b`	EXTCODESIZE	Vrátí velikost kódu kontraktu	-	700
`0x3c`	EXTCODECOPY	Okopíruje kód kontraktu do paměti	-	700
`0x3d`	RETURNDATASIZE	Vloží do zásobníku velikost vrácených dat	EIP-211 [http://bit.ly/2zaBcNe]	2
`0x3e`	RETURNDATACOPY	Okopíruje do paměti vrácená data	EIP-211 [http://bit.ly/2zaBcNe]	3
`0x3f`	Nepoužito	-	-	
`0x40`	BLOCKHASH	Vrátí haš jednoho z posledních 256 bloků	-	20
`0x41`	COINBASE	Vrátí adresu příjemce odměny za vytvoření bloku	-	2
`0x42`	TIMESTAMP	Vrátí časovou značku bloku	-	2

Instrukce	Název	Popis	Dodatečné informace	Plyn
`0x43`	NUMBER	Vrátí číslo bloku	-	2
`0x44`	DIFFICULTY	Vrátí obtížnost bloku	-	2
`0x45`	GASLIMIT	Vrátí limit plynu v bloku	-	2
`0x46` - `0x4f`	Nepoužito	-	-	
`0x50`	POP	Odstraní slovo ze zásobníku	-	2
`0x51`	MLOAD	Načte slovo z paměti	-	3
`0x52`	MSTORE	Uloží slovo do paměti	-	3*
`0x53`	MSTORE8	Uloží bajt do paměti	-	3
`0x54`	SLOAD	Načte slovo z úložiště	-	200
`0x55`	SSTORE	Uloží slovo do úložiště	-	0*
`0x56`	JUMP	Změní programový čítač	-	8
`0x57`	JUMPI	Podmíněně změní programový čítač	-	10
`0x58`	GETPC	Vrátí hodnotu programového čítače před zvýšením	-	2
`0x59`	MSIZE	Vrátí velikost aktivní paměti v bajtech	-	2

Instrukce	Název	Popis	Dodatečné informace	Plyn
`0x5a`	GAS	Vrátí množství dostupného plynu, včetně odpovídajícího snížení množství dostupného plynu	-	2
`0x5b`	JUMPDEST	Označí platný cíl skoku	-	1
`0x5c` - `0x5f`	Nepoužito	-	-	
`0x60`	PUSH1	Umístí 1 bajt dat na zásobník	-	3
`0x61`	PUSH2	Umístí 2 bajty dat na zásobník	-	3
`0x62`	PUSH3	Umístí 3 bajty dat na zásobník	-	3
`0x63`	PUSH4	Umístí 4 bajty dat na zásobník	-	3
`0x64`	PUSH5	Umístí 5 bajtů dat na zásobník	-	3
`0x65`	PUSH6	Umístí 6 bajtů dat na zásobník	-	3
`0x66`	PUSH7	Umístí 7 bajtů dat na zásobník	-	3
`0x67`	PUSH8	Umístí 8 bajtů dat na zásobník	-	3
`0x68`	PUSH9	Umístí 9 bajtů dat na zásobník	-	3
`0x69`	PUSH10	Umístí 10 bajtů dat na zásobník	-	3

Instrukce	Název	Popis	Dodatečné informace	Plyn
`0x6a`	PUSH11	Umístí 11 bajtů dat na zásobník	-	3
`0x6b`	PUSH12	Umístí 12 bajtů dat na zásobník	-	3
`0x6c`	PUSH13	Umístí 13 bajtů dat na zásobník	-	3
`0x6d`	PUSH14	Umístí 14 bajtů dat na zásobník	-	3
`0x6e`	PUSH15	Umístí 15 bajtů dat na zásobník	-	3
`0x6f`	PUSH16	Umístí 16 bajtů dat na zásobník	-	3
`0x70`	PUSH17	Umístí 17 bajtů dat na zásobník	-	3
`0x71`	PUSH18	Umístí 18 bajtů dat na zásobník	-	3
`0x72`	PUSH19	Umístí 19 bajtů dat na zásobník	-	3
`0x73`	PUSH20	Umístí 20 bajtů dat na zásobník	-	3
`0x74`	PUSH21	Umístí 21 bajtů dat na zásobník	-	3
`0x75`	PUSH22	Umístí 22 bajtů dat na zásobník	-	3
`0x76`	PUSH23	Umístí 23 bajtů dat na zásobník	-	3
`0x77`	PUSH24	Umístí 24 bajtů dat na zásobník	-	3
`0x78`	PUSH25	Umístí 25 bajtů dat na zásobník	-	3

Instrukce	Název	Popis	Dodatečné informace	Plyn
`0x79`	PUSH26	Umístí 26 bajtů dat na zásobník	-	3
`0x7a`	PUSH27	Umístí 27 bajtů dat na zásobník	-	3
`0x7b`	PUSH28	Umístí 28 bajtů dat na zásobník	-	3
`0x7c`	PUSH29	Umístí 29 bajtů dat na zásobník	-	3
`0x7d`	PUSH30	Umístí 30 bajtů dat na zásobník	-	3
`0x7e`	PUSH31	Umístí 31 bajtů dat na zásobník	-	3
`0x7f`	PUSH32	Umístí 32 bajtů dat (plné slovo) na zásobník	-	3
`0x80`	DUP1	Zdvojí 1. položku (vrchol) v zásobníku	-	3
`0x81`	DUP2	Zdvojí 2. položku v zásobníku	-	3
`0x82`	DUP3	Zdvojí 3. položku v zásobníku	-	3
`0x83`	DUP4	Zdvojí 4. položku v zásobníku	-	3
`0x84`	DUP5	Zdvojí 5. položku v zásobníku	-	3
`0x85`	DUP6	Zdvojí 6. položku v zásobníku	-	3
`0x86`	DUP7	Zdvojí 7. položku v zásobníku	-	3

Instrukce	Název	Popis	Dodatečné informace	Plyn
0x87	DUP8	Zdvojí 8. položku v zásobníku	-	3
0x88	DUP9	Zdvojí 9. položku v zásobníku	-	3
0x89	DUP10 Zdvojí 10. položku v zásobníku	-	3	0x8a
DUP11	Zdvojí 11. položku v zásobníku	-	3	0x8b
DUP12	Zdvojí 12. položku v zásobníku	-	3	0x8c
DUP13	Zdvojí 13. položku v zásobníku	-	3	0x8d
DUP14	Zdvojí 14. položku v zásobníku	-	3	0x8e
DUP15	Zdvojí 15. položku v zásobníku	-	3	0x8f
DUP16	Zdvojí 16. položku v zásobníku	-	3	0x90
SWAP1	Vymění 1. a 2. položku v zásobníku	-	3	0x91
SWAP2	Vymění 1. a 3. položku v zásobníku	-	3	0x92
SWAP3	Vymění 1. a 4. položku v zásobníku	-	3	0x93

Instrukce	Název	Popis	Dodatečné informace	Plyn
SWAP4	Vymění 1. a 5. položku v zásobníku	-	3	`0x94`
SWAP5	Vymění 1. a 6. položku v zásobníku	-	3	`0x95`
SWAP6	Vymění 1. a 7. položku v zásobníku	-	3	`0x96`
SWAP7	Vymění 1. a 8. položku v zásobníku	-	3	`0x97`
SWAP8	Vymění 1. a 9. položku v zásobníku	-	3	`0x98`
SWAP9	Vymění 1. a 10. položku v zásobníku	-	3	`0x99`
SWAP10	Vymění 1. a 11. položku v zásobníku	-	3	`0x9a`
SWAP11	Vymění 1. a 12. položku v zásobníku	-	3	`0x9b`
SWAP12	Vymění 1. a 13. položku v zásobníku	-	3	`0x9c`
SWAP13	Vymění 1. a 14. položku v zásobníku	-	3	`0x9d`

Instrukce	Název	Popis	Dodatečné informace	Plyn
SWAP14	Vymění 1. a 15. položku v zásobníku	-	3	`0x9e`
SWAP15	Vymění 1. a 16. položku v zásobníku	-	3	`0x9f`
SWAP16	Vymění 1. a 17. položku v zásobníku	-	3	`0xa0`
LOG0	Připojí do protokolu 0 témat	-	375	`0xa1`
LOG1	Připojí do protokolu 1 témata	-	750	`0xa2`
LOG2	Připojí do protokolu 2 témata	-	1125	`0xa3`
LOG3	Připojí do protokolu 3 témata	-	1500	`0xa4`
LOG4	Připojí do protokolu 4 témata	-	1875	`0xa5` - `0xaf`
Nepoužito	-	-		`0xb0`
JUMPTO	Předběžný libevmasm má různá čísla [http://bit.ly/2Sx2Vkg]	EIP 615 [http://bit.ly/2CR77pu]		`0xb1`
JUMPIF	Předběžný	EIP-615 [http://bit.ly/2CR77pu]		`0xb2`
JUMPSUB	Předběžný	EIP-615 [http://bit.ly/2CR77pu]		`0xb4`
JUMPSUBV	Předběžný	EIP-615 [http://bit.ly/2CR77pu]		`0xb5`

Instrukce	Název	Popis	Dodatečné informace	Plyn
BEGINSUB	Předběžný	EIP-615 [http://bit.ly/2CR77pu]		`0xb6`
BEGINDATA	Předběžný	EIP-615 [http://bit.ly/2CR77pu]		`0xb8`
RETURNSUB	Předběžný	EIP-615 [http://bit.ly/2CR77pu]		`0xb9`
PUTLOCAL	Předběžný	EIP-615 [http://bit.ly/2CR77pu]		`0xba`
GETLOCA	Předběžný	EIP-615 [http://bit.ly/2CR77pu]		`0xbb` - `0xe0`
Nepoužito	-	-		`0xe1`
SLOADBYTES	Pouze uvedené v pyethereum	-	-	`0xe2`
SSTOREBYTES	Pouze uvedené v pyethereum	-	-	`0xe3`
SSIZE	Pouze uvedené v pyethereum	-	-	`0xe4` - `0xef`
Nepoužito	-	-		`0xf0`
CREATE	Vytvoří nový účet s přiřazeným kódem	-	32000	`0xf1`
CALL	Volání zprávy do účtu	-	Complicated	`0xf2`
CALLCODE	Volání zprávy do tohoto účtu s alternativním kódem účtu	-	Complicated	`0xf3`
RETURN	Ukončí provádění a vrátí výstupní data	-	0	`0xf4`

Instrukce	**Název**	**Popis**	**Dodatečné informace**	**Plyn**
DELEGATECALL	Volání zprávy do účtu s alternativním kódem účtu, ale přetrvávajícím na tomto účtu s alternativním kódem účtu	-	Complicated	`0xf5`
CALLBLACKBOX	-	-	40	`0xf6` - `0xf9`
Nepoužito	-	-		`0xfa`
STATICCALL	Podobné CALL, ale nemění stav	-	40	`0xfb`
CREATE2	Vytvoří nový účet a nastaví adresu vytvoření na `sha3(sender + sha3(init code)) % 2**160`	-		`0xfc`
TXEXECGAS	Není ve Žluté knize FIXME	-	-	`0xfd`
REVERT	Zastaví provádění a anuluje změnu stavu bez spotřebování všeho potřebného plynu a poskytnutí důvodu	-	0	`0xfe`
INVALID	Záměrně navržená neplatná instrukce	-	0	`0xff`

Appendix D: Vývojové nástroje, rámce a knihovny

Rámce

("frameworks", id="ix_appdx-dev-tools-asciidoc0", range="startofrange"Pomocí rámců lze usnadnit vývoj Ethereum chytrých kontraktů. Když uděláte všechno sami, získáte lepší představu o tom, jak všechno zapadá do sebe, ale je to spousta únavné a opakující se práce. Rámce popsané v této části mohou automatizovat určité úkoly a usnadnit vývoj.

Truffle

GitHub: https://github.com/trufflesuite/truffle

Web: https://truffleframework.com

Dokumentace: https://truffleframework.com/docs

Truffle vzorové projekty: http://truffleframework.com/boxes/

npm úložiště balíčku: https://www.npmjs.com/package/truffle

Instalace Truffle rámce

Rámec Truffle zahrnuje několik balíčků Node.js. Před instalací truffle musíte mít aktuální a funkční instalaci Node.js a Node Package Manager (npm).

Doporučeným způsobem instalace Node.js a npm je použití Správce verzí uzlů (nvm). Po instalaci nvm se za vás postará o všechny závislosti a aktualizace. Postupujte podle pokynů na adrese http://nvm.sh.

Jakmile je nvm nainstalované ve vašem operačním systému, installing Node.js je jednoduchá. Použijte nastavení --lts, abyste oznámili nvm, že chcete nejnovější dlouhodobě podporovanou (LTS) verzi Node.js:

```
<pre data-type="programlisting">
$ <strong>nvm install --lts</strong>
</pre>
```

Ujistěte se, že jste nainstalovali node a npm:

```
<pre data-type="programlisting">
$ <strong>node -v</strong>
v8.9.4
$ <strong>npm -v</strong>
5.6.0
</pre>
```

Dále vytvořte skrytý soubor *.nvmrc*, který obsahuje verzi Node.js podporovanou vaším DApp, takže vývojáři stačí spustit `nvm install` v kořenovém adresáři projektového adresáře a automaticky se nainstaluje a přepne na použití této verze:

```
<pre data-type="programlisting">
$ <strong>node -v &gt; .nvmrc</strong>
$ <strong>nvm install</strong>
</pre>
```

Vypadá to dobře, nyní nainstalujeme truffle:

```
<pre data-type="programlisting">
$ <strong>npm -g install truffle</strong>

+ truffle@4.0.6
installed 1 package in 37.508s
</pre>
```

Integrace předpřipraveného Truffle projektu (Truffle Box)

Pokud chcete použít nebo vytvořit DApp, který staví na předdefinované základové desce, přejděte na web Truffle Boxes, vyberte existující projekt Truffle a poté spusťte následující příkaz pro jeho stažení a rozbalení:

```
<pre data-type="programlisting">
$ <strong>truffle unbox <em>BOX_NAME</em></strong>
</pre>
```

Vytvoření adresáře projektu truffle

Pro každý projekt, kde budete používat truffle, vytvořte adresář projektu a inicializujte truffle v tomto adresáři. truffle vytvoří nezbytnou strukturu adresářů v adresáři projektu. Je obvyklé dát adresáři projektu název, který projekt popisuje. V tomto příkladu použijeme truffle k nasazení našeho kontraktu Faucet Jednoduchá smlouva: Testovací Ethereum kohoutek, a proto pojmenujeme adresář projektu *Faucet*:

```
<pre data-type="programlisting">
$ <strong>mkdir Faucet</strong>
$ <strong>cd Faucet</strong>
Faucet $
</pre>
```

Jakmile jsme v adresáři *Faucet*, inicializujeme truffle:

```
<pre data-type="programlisting">
Faucet $ <strong>truffle init</strong>
</pre>
```

truffle vytvoří strukturu adresářů a některé výchozí soubory:

```
Faucet
+---- contracts
|   `---- Migrations.sol
+---- migrations
|   `---- 1_initial_migration.js
+---- test
+---- truffle-config.js
`---- truffle.js
```

Kromě samotného truffle také použijeme řadu podporovaných balíčků JavaScriptu (Node.js).

Můžeme je nainstalovat s npm. Inicializujeme strukturu adresáře npm a přijímáme výchozí hodnoty navržené npm:

```
<pre data-type="programlisting">
$ <strong>npm init</strong>

package name: (faucet)
version: (1.0.0)
description:
entry point: (truffle-config.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to Faucet/package.json:

{
  "name": "faucet",
  "version": "1.0.0",
  "description": "",
  "main": "truffle-config.js",
  "directories": {
    "test": "test"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" &amp;&amp; exit 1"
  },
  "author": "",
  "license": "ISC"
}


Is this ok? (yes)
</pre>
```

Nyní můžeme nainstalovat závislosti, pomocí kterých budeme snadněji pracovat s truffle:

```
<pre data-type="programlisting">
$ <strong>npm install dotenv truffle-wallet-provider ethereumjs-
wallet</strong>
</pre>
```

Nyní máme adresář *node_modules* s několika tisíci soubory uvnitř našeho *Faucet* adresáře.

Před nasazením DApp do cloudového produkčního nebo kontinuálního integračního prostředí je důležité zadat pole engines , aby byl váš DApp vytvořen se správnou verzí Node.js a byly nainstalovány související závislosti. Podrobnosti o konfiguraci tohoto pole naleznete v dokumentaci [http://bit.ly/2zp2GPF].

Konfigurace truffle

truffle vytváří pouze prázdné konfigurační soubory, *truffle.js* a *truffle-config.js*. V operačním systému Windows_truffle.js_ může způdobit potíže, když se pokusíte spustit příkaz truffle a Windows se místo toho pokusí spustit *truffle.js*. Abysme se tomu vyhnuli, smažeme *truffle.js* a místo toho použijeme *truffle-config.js* (na podporu uživatelů Windows, kteří, upřímně, už dost trpí):

```
<pre data-type="programlisting">
$ <strong>rm truffle.js</strong>
</pre>
```

Nyní upravíme soubor *truffle-config.js* a nahradíme obsah ukázkovou konfigurací uvedenou zde:

```
module.exports = {
  networks: {
    localnode: { // Jakákoli sí¬, ke které se náš místní uzel p¬ipojuje
      network_id: "*", // Odpovídající ID sít¬
      host: "localhost",
      port: 8545,
    }
  }
};
```

Tato konfigurace je dobrým výchozím bodem. Nastaví jednu výchozí Ethereum síť (pojmenovanou

localnode), která předpokládá, že provozujeme klienta Ethereum, jako je Parity, buď jako úplný uzel nebo jako odlehčený klient. Tato konfigurace dá pokyn truffle ke komunikaci s lokálním uzlem přes RPC, na portu 8545. truffle použije jakoukoli Ethereum síť, k níž je místní uzel připojen, jako je Ethereum hlavní síť nebo testovací síť jako Ropsten. Lokální uzel bude také poskytovat funkce peněženky.

V následujících sekcích nakonfigurujeme další sítě pro použití truffle, jako je ganache lokální testovací bločenka a Infura, hostovaný poskytovatel sítě. Když přidáváme další sítě, bude konfigurační soubor složitější, ale také nám poskytne více možností pro náš pracovní postup testování a vývoje.

Použití truffle k nasazení kontraktu

Nyní máme základní pracovní adresář pro náš projekt *Faucet* a máme nakonfigurovaný truffle a jeho závislosti. Kontrykty se nacházejí v podadresáři *contracts* našeho projektu. Adresář již obsahuje „pomocný“ kontakt *Migrations.sol*, který pro nás spravuje vylepšování kontraktu. V další části prozkoumáme použití *Migrations.sol*.

Okopírujme kontrakt *Faucet.sol* (z Faucet.sol: Chytrý kontrakt Kohoutek v Solidity) do podadresáře *contracts*, takže projektový adresář bude vypadat následovně:

```
Faucet
+---- contracts
|   +---- Faucet.sol
|   `---- Migrations.sol
...
```

Nyní můžeme požádat truffle, aby pro nás sestavil smlouvu:

```
<pre data-type="programlisting">
$ <strong>truffle compile</strong>
Compiling ./contracts/Faucet.sol...
Compiling ./contracts/Migrations.sol...
Writing artifacts to ./build/contracts
</pre>
```

Truffle migrace a porozumění nasazovacím skriptům

Truffle nabízí systém nasazení zvaný *migrace*. Pokud jste pracovali v jiných rámcích, možná jste viděli něco podobného: Ruby on Rails, Python Django a mnoho dalších jazyků a rámců má příkaz migrate.

Ve všech těchto rámcích je účelem migrace zpracovat změny v datovém schématu mezi různými verzemi softwaru. Účel migrace v Ethereum je poněkud odlišný. Protože Ethereum kontrakty jsou neměnné a jejich zavedení je nákladné, Truffle nabízí mechanismus migrace, který sleduje, které kontrakty (a které verze) již byly zavedeny. V komplexním projektu s desítkami kontraktů a složitými závislostmi byste nemuseli platit za opakované nasazení kontraktů, které se nezměnily. Také byste nechtěli ručně sledovat, které verze kontraktů již byly nasazeny. Mechanismus Truffle migrace to vše provádí nasazením chytrého kontraktu *Migrations.sol*, která pak sleduje všechna další nasazení kontraktů.

Máme pouze jeden kontrakt, *Faucet.sol*, což znamená, že migrační systém je přinejmenším kanónem na vrabce. Bohužel ho musíme použít. Když se však naučíme, jak je použít pro jednu smlouvu, můžeme začít procvičovat některé dobré návyky pro náš vývojový pracovní postup. Úsilí se vyplatí, jak se věci komplikují.

Truffle *migrační* adresář je místo, kde se nachází migrační skripty. Právě teď existuje pouze jeden skript *1_initial_migration.js*, který nasazuje samotný kontrakt *Migrations.sol*:

```
var Migrations = artifacts.require("./Migrations.sol");

module.exports = function(deployer) {
  deployer.deploy(Migrations);
};
```

K nasazení *Faucet.sol* potřebujeme druhý migrační skript. Nazvěme ho *2_deploy_contracts.js*. Je to velmi jednoduché, stejně jako *1_initial_migration.js*, s několika malými změnami. Ve skutečnosti můžete zkopírovat obsah *1_initial_migration.j* a jednoduše nahradit všechny výskyty Migrations za Faucet:

```
var Faucet = artifacts.require("./Faucet.sol");

module.exports = function(deployer) {
  deployer.deploy(Faucet);
};
```

Skript inicializuje proměnnou Faucet, označí tím Solidity zdrojový kód *Faucet.sol* Solidity jako artefakt, který definuje Faucet. Poté zavolá funkci `deploy` pro nasazení tohoto kontraktu.

Vše je připraveno. K nasazení použijte truffle migrate. Musíme specifikovat, která síť se má použít, pomocí parametru --network. V konfiguračním souboru jsme zadali pouze jednu síť, kterou jsme nazvali localnode. Zkontrolujte, zda je spuštěn váš Ethereum místní klient a potom zadejte:

```
<pre data-type="programlisting">
Faucet $ <strong>truffle migrate --network localnode</strong>
</pre>
```

Protože k připojení k Ethereum síti a správě naší peněženky používáme místní uzel, musíme autorizovat transakci, kterou truffle vytvoří. Provozujeme parity připojeného k testovací bločence Ropsten, takže během migrace uvidíme vyskakovací okno jako v Parity žádající o potvrzení nasazení Faucet on Parity webové příkazové řádce.

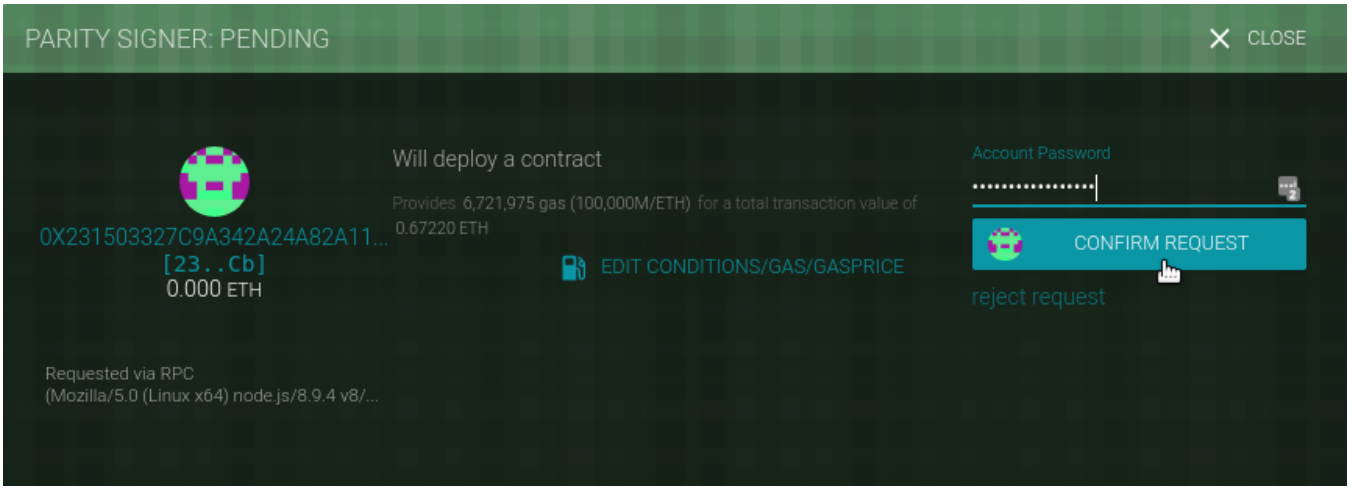

Figure 58. Parity žádající o potvrzení nasazení Faucet

Celkem existují čtyři transakce: jedna k nasazení Migrations, jedna k aktualizaci čítače nasazení na

1, jedna k nasazení Faucet a jedna k aktualizaci čítače nasazení na 2.

Truffle zobrazí dokončení migrace, zobrazí každou transakci a adresy kontraktů:

```
<pre data-type="programlisting">
$ <strong>truffle migrate --network localnode</strong>
Using network 'localnode'.

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ... 0xfa090db179d023d2abae543b4a21a1479e70ca7d35a469a5d1a98bfc6bd80fe8
  Migrations: 0x8861c27715550bed8362c0345add158489df6db0
Saving successful migration to network...
  ... 0x985c4a32716826ddbe4eae284104bef8bc69e959899f62246a1b27c9dfcd6c03
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying Faucet...
  ... 0xecdbeef77f0558edc689440e34b7bba0a3ba7a45e4b680b071b47c30a930e9d6
  Faucet: 0xd01cd8e7bd29e4bff8c1693f59eee46137a9f300
Saving successful migration to network...
  ... 0x11f376bd7307edddfd40dc4a14c3f7cb84b6c921ac2465602060b67d08f9fd8a
Saving artifacts...
</pre>
```

Používání Truffle příkazové řádky

Truffle nabízí JavaScript příkazovou řádku, kterou můžeme použít k interakci s Ethereum sítí (prostřednictvím místního uzlu), interakci s nasazenými kontrakty a interakci s poskytovatelem peněženky. V naší současné konfiguraci (localnode) je poskytovatelem uzlů a peněženek náš místní klient Parity.

Spusťte Truffle příkazovou řádku a vyzkoušejte některé příkazy:

```
<pre data-type="programlisting">
$ <strong>truffle console --network localnode</strong>
truffle(localnode)>
</pre>
```

Truffle zobrazí výzvu ukazující vybranou konfiguraci sítě (localnode).

Je důležité si zapamatovat a uvědomit si, kterou síť používáte. Nechtěli byste nechtěně nasadit zkušební kontrakt nebo provést zkušební transakci v hlavní Ethereum síti. To by mohla být nákladná chyba!

Příkazová řádka Truffle nabízí funkci automatického doplňování, která nám usnadňuje prozkoumávat prostředí. Pokud stiskneme Tab po částečně dokončeném příkazu, Truffle dokončí příkaz pro nás. Dvojím stisknutím klávesy Tab se zobrazí všechna možná dokončení, pokud našemu vstupu odpovídá více než jeden příkaz. Pokud na prázdnou výzvu stiskneme dvakrát klávesu Tab, Truffle zobrazí všechny dostupné příkazy:

```
<pre data-type="programlisting" class="codewrap">
truffle(localnode)&gt;
Array Boolean Date Error EvalError Function Infinity JSON Math NaN Number
Object RangeError ReferenceError RegExp String SyntaxError TypeError
URIError decodeURI decodeURIComponent encodeURI encodeURIComponent eval
isFinite isNaN parseFloat parseInt undefined

ArrayBuffer Buffer DataView Faucet Float32Array Float64Array GLOBAL
Int16Array Int32Array Int8Array Intl Map Migrations Promise Proxy Reflect
Set StateManager Symbol Uint16Array Uint32Array Uint8Array
Uint8ClampedArray WeakMap WeakSet WebAssembly XMLHttpRequest _ assert
async_hooks buffer child_process clearImmediate clearInterval clearTimeout
cluster console crypto dgram dns domain escape events fs global http http2
https module net os path perf_hooks process punycode querystring readline
repl require root setImmediate setInterval setTimeout stream
string_decoder tls tty unescape url util v8 vm web3 zlib

__defineGetter__ __defineSetter__ __lookupGetter__ __lookupSetter__
__proto__ constructor hasOwnProperty isPrototypeOf propertyIsEnumerable
toLocaleString toString valueOf
</pre>
```

Převážnou většinu funkcí souvisejících s peněženkami a uzly poskytuje objekt web3, což je instance knihovny web3.js. Objekt web3 abstrahuje rozhraní RPC do našeho Parity uzlu. Všimněte si také dvou objektů se známými jmény: Migrations a Faucet. To jsou kontrakty, které jsme právě nasadili. K interakci s kontrakty použijeme Truffle příkazovou řádku. Nejprve zkontrolujeme naši peněženku

pomocí objektu web3:

```
<pre data-type="programlisting">
truffle(localnode)&gt; <strong>web3.eth.accounts</strong>
[ '0x9e713963a92c02317a681b9bb3065a8249de124f',
  '0xdb5dc1a13e3a55cf3b4587cd8d1e5fdeb6738145' ]
</pre>
```

Náš Parity klient má dvě peněženky, s testovacím etherem na Ropstenu. Atribut web3.eth.accounts obsahuje seznam všech účtů. Můžeme zkontrolovat zůstatek prvního účtu pomocí funkce getBalance:

```
<pre data-type="programlisting">
truffle(localnode)&gt;
<strong>web3.eth.getBalance(web3.eth.accounts[0]).toNumber()</strong>
191198572800000000
truffle(localnode)&gt;
</pre>
```

web3.js je velká JavaScript knihovna, která nabízí komplexní rozhraní systému Ethereum prostřednictvím poskytovatele, jako je například místní klient. Podrobněji prozkoumáme web3.js v web3.js tutorial. Nyní se pokusíme spolupracovat s našimi kontrakty

```
<pre data-type="programlisting">
truffle(localnode)&gt; <strong>Faucet.address</strong>
'0xd01cd8e7bd29e4bff8c1693f59eee46137a9f300'
truffle(localnode)&gt;
<strong>web3.eth.getBalance(Faucet.address).toNumber()</strong>
0
truffle(localnode)&gt;
</pre>
```

Dále použijeme sendTransaction k odeslání testovacího etheru k financování kontraktu Faucet. Všimněte si použití web3.utils.toWei k převodu jednotek etheru pro nás. Zadání 18 nul, aniž by došlo k chybě, je obtížné a nebezpečné, takže je vždy lepší použít pro převod jednotek převodník jednotek. Takto odesíláme transakci:

```
<pre data-type="programlisting">
truffle(localnode)&gt;
<strong>web3.eth.sendTransaction({from:web3.eth.accounts[0],
                    to:Faucet.address, value:web3.utils.toWei(0.5,
'ether')});</strong>
'0xf134c75b985dc0e0c27c2f0412251e0860eb530a5055e660f21e7483ab336808'
</pre>
```

Pokud přepneme na webové rozhraní Parity, zobrazí se vyskakovací okno s žádostí o potvrzení této transakce. Jakmile bude transakce vytěžena, uvidíme zůstatek v našem Faucet kontraktu:

```
<pre data-type="programlisting">
truffle(localnode)&gt;
<strong>web3.eth.getBalance(Faucet.address).toNumber()</strong>
500000000000000000
</pre>
```

Zavolejme nyní funkci withdraw, abychom ze smlouvy odebrali nějaký testovací ether:

```
<pre data-type="programlisting">
truffle(localnode)&gt; <strong>Faucet.deployed().then(instance =>
                      {instance.withdraw(web3.utils.toWei(0.1,
                      'ether'))}).then(console.log)</strong>
</pre>
```

Znovu budeme muset transakci schválit ve webovém rozhraní Parity. Pokud se znovu podíváme, uvidíme, že zůstatek kontraktu Faucet se snížil a naše testovací peněženka obdržela 0,1 etheru:

```
<pre data-type="programlisting">
truffle(localnode)&gt;
<strong>web3.eth.getBalance(Faucet.address).toNumber()</strong>
400000000000000000
truffle(localnode)&gt; <strong>Faucet.deployed().then(instance =>
                    {instance.withdraw(web3.utils.toWei(1,
'ether'))})</strong>
StatusError: Transaction: 0xe147ae9e3610334...8612b92d3f9c
  exited with an error (status 0).
</pre>
```

Embark

GitHub: https://github.com/embark-framework/embark/

Documentation: https://embark.status.im/docs/

npm package repository: https://www.npmjs.com/package/embark

Embark je rámec vytvořený tak, aby vývojářům umožnil snadný vývoj a nasazení decentralizovaných aplikací. Embark se integruje s Ethereum, IPFS, Whisper a Swarm a nabízí následující funkce:

- automatické nasazení kontraktů a jejich zpřístupnění v JS kódu.
- Sledujte změny a aktualizujte kontrakty, abyste v případě potřeby je mohli znovu nasadit.
- Správa a interakce s různými bločenkami (např. testovací, lokální, hlavní).
- Spravujte složité systémy vzájemně závislých kontraktů.
- Ukládá a získává data, včetně nahrávání a načítání souborů uložených v IPFS.
- Usnadňuje proces nasazení celé aplikace na IPFS nebo Swarm.
- Odesílá a přijímá zprávy prostřednictvím Whisper.

Můžete jej nainstalovat pomocí npm:

```
<pre data-type="programlisting">
$ <strong>npm -g install embark</strong>
</pre>
```

OpenZeppelin

GitHub: https://github.com/OpenZeppelin/openzeppelin-solidity

Web: https://openzeppelin.org/

Dokumentace: https://openzeppelin.org/api/docs/open-zeppelin.html

OpenZeppelin [https://openzeppelin.org/] je otevřený rámec opakovaně použitelných a bezpečných chytrých kontraktů v jazyce Solidity.

Je řízen komunitou pod vedením týmu Zeppelin [https://zeppelin.solutions/] s více než stovkou externích přispěvatelů. Hlavním zaměřením rámce je bezpečnost, dosažená použitím standardních bezpečnostních vzorů chytrých kontraktů a osvědčených postupů, čerpající ze všech zkušeností, které vývojáři Zeppelin získali z auditu [https://blog.zeppelin.solutions/tagged/security] a obrovského množství zakázek a prostřednictvím neustálého testování a auditu ze strany komunity, která používá rámec jako základ pro jejich aplikace v reálném světě.

Rámec OpenZeppelin je nejrozšířenějším řešením Ethereum chytrých kontraktů. Rámec má v současné době bohatou knihovnu kontraktů, včetně implementace tokenů ERC20 a ERC721, mnoho variant modelů pro skupinový prodej a jednoduchá chování běžně vyskytující se v kontraktech, jako jsou `Ownable`, `Pausable`, nebo `LimitBalance`. Kontrakty v tomto úložišti fungují v některých případech jako standardní implementace.

Rámec je licencován na základě licence MIT a všechny kontrakty byly navrženy s modulárním přístupem, který zaručuje snadné opětovné použití a rozšíření. Jedná se o čisté a základní stavební kameny, připravené k použití v příštím Ethereum projektu. Nastavme rámec a vytvoříme jednoduchý kontrakt skupinového prodeje pomocí OpenZeppelin kontraktu, abychom ukázali, jak snadné je ho používat. Tento příklad také zdůrazňuje důležitost opětovného použití bezpečných součástí namísto jejich napsání námi samotnými.

Nejprve budeme muset nainstalovat knihovnu openzeppelin-solidity do našeho pracovního prostoru. Nejnovější vydání v době tohoto psaní je v1.9.0, takže použijeme toto:

```
<pre data-type="programlisting">
$ <strong>mkdir sample-crowdsale</strong>
$ <strong>cd sample-crowdsale</strong>
$ <strong>npm install openzeppelin-solidity@1.9.0</strong>
$ <strong>mkdir contracts</strong>
</pre>
```

V době psaní této knihy obsahuje OpenZeppelin několik základních kontraktů tokenů, které dodržují standardy ERC20, ERC721 a ERC827, s různými charakteristikami emisí, limitů, získávání, životního cyklu atd.

Vytvořme ERC20 token, který je razitelný což znamená, že počáteční zásoba začíná na 0 a nový token může být vytvořen vlastníkem tokenu (v našem případě při veřejném prodeji) kontraktu) a prodáván nakupujícím. Za tímto účelem vytvoříme soubor *contracts/SampleToken.sol* s následujícím obsahem:

```
pragma solidity 0.4.23;

import 'openzeppelin-solidity/contracts/token/ERC20/MintableToken.sol';

contract SampleToken is MintableToken {
  string public name = "SAMPLE TOKEN";
  string public symbol = "SAM";
  uint8 public decimals = 18;
}
```

OpenZeppelin již poskytuje kontrakt MintableToken, který můžeme použít jako základnu pro náš token, takže definujeme pouze podrobnosti, které jsou specifické pro náš případ. Dále udělejme kontrakt pro veřejný prodej. Stejně jako u tokenů, OpenZeppelin již nabízí širokou škálu variant. V současné době najdete kontrakty na různé scénáře zahrnující distribuci, emise, cenu a validaci. Řekněme tedy, že chcete nastavit cíl pro náš veřejný prodej a pokud není splněn do konce prodeje, chcete vrátit peníze všem našim investorům. K tomu můžete použít RefundableCrowdsale [http://bit.ly/2yHoh65] kontrakt. Nebo možná budete chtít definovat veřejný prodej s rostoucí cenou, abyste motivovali začínající kupce; tady IncreasingPriceCrowdsale [http://bit.ly/2PtWOys] kontrakt dělá přesně tohle. Veřejný prodej můžete také ukončit, jakmile kontrakt obdrží určité množství etheru(CappedCrowdsale [http://bit.ly/2OVsCN8]), nebo uplyne daný čas TimedCrowdsale [http://bit.ly/2zp2Nuz] kontrakt, nebo vytvořit seznam povolených adres kupujících WhitelistedCrowdsale

[http://bit.ly/2CN8Hc9] kontrakt.

Jak jsme již uvedli, OpenZeppelin kontrakty jsou základními stavebními kameny. Tyto kontrakty veřejného prodeje byly navrženy tak, aby byly kombinovatelné; přečtěte si zdrojový kód základního Crowdsale [http://bit.ly/2ABIQSI] kontraktu pro pokyny jak ho rozšířit. Pro veřejný prodej našeho tokenu potřebujeme razit tokeny, když je kontrakt veřejného prodeje získá ether, použijme. MintedCrowdsale [http://bit.ly/2Sx3HOc] jako základ. A aby to bylo ještě zajímavější, udělejme to také PostDeliveryCrowdsale [http://bit.ly/2Qef0Jm] takže tokeny mohou být odebrány až po skončení veřejného prodeje. Za tímto účelem napíšeme následující *contracts/SampleCrowdsale.sol*:

```
pragma solidity 0.4.23;

import './SampleToken.sol';
import 'openzeppelin-
solidity/contracts/crowdsale/emission/MintedCrowdsale.sol';
import 'openzeppelin-solidity/contracts/crowdsale/ \
  distribution/PostDeliveryCrowdsale.sol';

contract SampleCrowdsale is PostDeliveryCrowdsale, MintedCrowdsale {

  constructor(
    uint256 _openingTime,
    uint256 _closingTime
    uint256 _rate,
    address _wallet,
    MintableToken _token
  )
    public
    Crowdsale(_rate, _wallet, _token)
    PostDeliveryCrowdsale(_openingTime, _closingTime)
  {
  }
}
```

Opět jsme skoro nemuseli psát jakýkoli kód; znovu jsme použili bitvou testovaný kód, který komunita OpenZeppelin zpřístupnila. Je však důležité si uvědomit, že tento případ je odlišný od případu našeho kontraktu `SampleToken`. Pokud jdete na Crowdsale automated tests [http://bit.ly/2Q8lQ3o] uvidíte, že jsou testovány izolovaně. Když integrujete různé jednotky kódu do větší komponenty, nestačí otestovat všechny jednotky samostatně, protože vzájemné působení mezi nimi

může způsobit chování, které jste neočekávali. Zejména uvidíte, že jsme zde zavedli vícenásobnou dědičnost, které může vývojáře překvapit, pokud nerozumí detailům Solidity. Náš `SampleCrowdsale` kontrakt je jednoduchý a bude fungovat tak, jak očekáváme, protože rámec byl navržen tak, aby případy jako tyto byly jednoduché; ale nepolevte ve vaši ostražitost kvůli jednoduchosti, kterou tento rámec zavádí. Pokaždé, když integrujete části rámce OpenZeppelin za účelem vytvoření komplexnějšího řešení, musíte plně otestovat všechny aspekty svého řešení, abyste se ujistili, že všechny interakce jednotek fungují podle vašich představ.

A konečně, když jsme s naším řešením spokojeni a je důkladně vyzkoušeno, musíme jej nasadit. OpenZeppelin se dobře integruje s Truffle, takže můžeme napsat pouze migrační soubor, jako je následující (*migrations/2_deploy_contracts.js*), jak je vysvětleno v Truffle migrace a porozumění nasazovacím skriptům:

```
const SampleCrowdsale = artifacts.require('./SampleCrowdsale.sol');
const SampleToken = artifacts.require('./SampleToken.sol');

module.exports = function(deployer, network, accounts) {
  const openingTime = web3.eth.getBlock('latest').timestamp + 2; // 2s in
future
  const closingTime = openingTime + 86400 * 20; // 20 days
  const rate = new web3.BigNumber(1000);
  const wallet = accounts[1];

  return deployer
    .then(() => {
      return deployer.deploy(SampleToken);
    })
    .then(() => {
      return deployer.deploy(
        SampleCrowdsale,
        openingTime,
        closingTime,
        rate,
        wallet,
        SampleToken.address
      );
    });
};
```

Byl to jen rychlý přehled několika kontraktů, které jsou součástí rámce OpenZeppelin. Můžete se připojit k vývojové komunitě OpenZeppelin a učit se a přispívat.

ZeppelinOS

GitHhub: https://github.com/zeppelinos

Web: https://zeppelinos.org

Blog: https://blog.zeppelinos.org

ZeppelinOS [https://github.com/zeppelinos] je distribuovaná platforma nástrojů a služeb s otevřeným zdrojovým kódem, postavená na vrcholu EVM k bezpečnému vyvíjení a správě aplikací chytrých kontraktů.

Na rozdíl od kódu OpenZeppelin, který je třeba při každé aplikaci znovu nasadit s každou aplikací, zůstává kód ZeppelinOS v bločence. Aplikace, které potřebují danou funkčnost, řekněme ERC20 token, a to nejen, že nemusí přepracovávat a opakovat jeho implementaci (něco, co OpenZeppelin vyřešil), ale ani jej nemusí implementovat. S ZeppelinOS, aplikace interaguje s implementací tokenu přímo bločence, téměř stejným způsobem, jako aplikace stolního počítače interaguje se součástmi svého základního OS.

("proxy"Jádrem ZeppelinOS je velmi chytrý kontrakt známý jako *proxy*. Proxy je kontrakt, která je schopen zabalit jakýkoli jiný kontrakt, odhalit jeho rozhraní, aniž by pro něj musela ručně implementovat funkce nastavení a čtení parametrů a může ho vylepšovat bez ztráty stavu. Z hlediska Solidity to lze považovat za normální kontrakt, jejíž obchodní logika je obsažena v knihovně, kterou lze kdykoli vyměnit za novou knihovnu, aniž by došlo ke ztrátě jejího stavu. Způsob, jakým se proxy připojuje k jeho implementaci, je pro vývojáře zcela automatizovaný a zapouzdřený. Prakticky může být každý kontrakt vylepšován s malou až žádnou změnou jejo kódu. Více o proxy mechanismu ZeppelinOS lze nalézt v blog [http://bit.ly/2OfuNpu], a příklad použití lze nalézt na GitHubu [http://bit.ly/2OfuE5q].

Vývoj aplikací pomocí ZeppelinOS je podobný vývoji JavaScript aplikací pomocí npm. AppManager zpracovává balíček aplikací pro každou verzi aplikace. Balíček je jednoduše adresář kontraktů, z nichž každá může mít jeden nebo více vylepšovatelných proxy. AppManager poskytuje nejen proxy pro kontrakty specifické pro aplikaci, ale také pro implementace ZeppelinOS, ve formě standardní

knihovny. Úplný příklad toho naleznete na adrese examples/complex [http://bit.ly/2PtyJb3].

Přestože je ZeppelinOS v současné době ve vývoji, usiluje o poskytování široké škály dalších funkcí, jako jsou vývojářské nástroje, plánovač, který automatizuje operace na pozadí v rámci kontraktů, vývojové odměny, tržiště, které usnadňuje komunikaci a výměnu hodnoty mezi aplikacemi, a ještě mnohem více. To vše je popsáno v ZeppelinOS Bílé knize [http://bit.ly/2QcxV7K].

Nástroje

EthereumJS helpeth: nástroj příkazové řádky

GitHub: https://github.com/ethereumjs/helpeth

helpeth je nástroj příkazové řádky pro manipulaci s klíči a transakcemi, který vývojářům v mnohém usnadňuje práci.

Je součástí kolekce EthereumJS knihoven a nástrojů založených na JavaScriptu:

```
Použití: helpeth [command]

P¬íkazy:
  signMessage <message>                        Podepíše zprávu
  verifySig <hash> <sig>                       Ov¬¬í podpis
  verifySigParams <hash> <r> <s> <v>           Ov¬¬í parametry podpisu
  createTx <nonce> <to> <value> <data>         Podepíše zprávu
  <gasLimit> <gasPrice>
  assembleTx <nonce> <to> <value> <data>       Složí transakci z
  <gasLimit> <gasPrice> <v> <r> <s>            ¬ástí
  parseTx <tx>                                  Zpracuje surovou transakci
  keyGenerate [format] [icapdirect]            Vytvo¬í nový klí¬
  keyConvert                                   P¬evede klí¬ do V3 formátu
  keyDetails                                   Vytiskne podrobnosti o klí¬i
  bip32Details <path>                          Vytiskne podrobnosti o klí¬i
zadané cesty
  addressDetails <address>                     Vytiskne podrobnosti o adrese
  unitConvert <value> <from> <to>              P¬evod mezi Ethereum
jednotkami

Nastavení:
  -p, --private      Soukromý klí¬ jako hex ¬et¬zec
[string]
  --password         Heslo pro soukromý klí¬
[string]
  --password-prompt  Výzva k zadání soukromého klí¬e
[boolean]
  -k, --keyfile      Soubor se zakódovaným klí¬em
[string]
  --show-private     Zobrazí podrobnosti soukromého kli¬e         [boolean]
  --mnemonic         Mnemotechnická slovo pro odvození HD klí¬e   [string]
  --version          Zobrazí ¬íslo verze
[boolean]
  --help             Zobrazí nápov¬du
[boolean]
```

dapp.tools

Web: https://dapp.tools/

dapp.tools je komplexní sada vývojových nástrojů orientovaných na bločenku vytvořená v duchu filozofie Unixu. Zahrnuty jsou tyto nástroje:

Dapp

Dapp je základní uživatelsky orientovaný nástroj pro vytváření nových DApps, spouštění testů Solidity jednotek, ladění a nasazování kontraktů, spouštění testovacích sítí a další.

Seth

Seth se používá pro vytváření transakcí, dotazování bločenky, převod mezi datovými formáty, provádění vzdálených volání a podobných každodenních úkolů.

Hevm

Hevm je implementace Haskell EVM s hbitým nástrojem pro ladění Solidity. Používá se k testování a ladění DApps.

evmdis

evmdis je dekompilátor EVM; provádí statickou analýzu v bajtkódu, aby poskytl vyšší úroveň abstrakce než surové instrukce EVM.

SputnikVM

SputnikVM [https://github.com/etcdevteam/sputnikvm] je samostatný zásuvný virtuální stroj pro různé bločenky na bázi Etherea. Je psán v Rustu a lze jej použít jako binární, přepravní bednu nebo sdílenou knihovnu, nebo integrovat prostřednictvím rozhraní FFI, Protobuf a JSON. Má samostatný binární soubor sputnikvm-dev určený pro testovací účely, který emuluje většinu rozhraní JSON-RPC API a těžby bloku.

Knihovny

web3.js

web3.js ("web3.js"je rozhraní API kompatibilní s Ethereum pro komunikaci s klienty prostřednictvím JSON-RPC, vyvinuté Nadací Ethereum.

GitHub: https://github.com/ethereum/web3.js

npm úložiště balíčku: https://www.npmjs.com/package/web3

Dokumentace pro web3.js API 0.2x.x: http://bit.ly/2Qcyq1C

Dokumentace pro web3.js API 1.0.0-beta.xx: http://bit.ly/2CT33p0

web3.py

web3.py je Python knihovna pro interakce s Ethereum bločenkou, spravovaný Nadací Ethereum.

GitHub: https://github.com/ethereum/web3.py

PyPi: https://pypi.python.org/pypi/web3/4.0.0b9

Dokumentace: https://web3py.readthedocs.io/

EthereumJS

EthereumJS kolepce knihoven a nástrojů pro Ethereum.

GitHub: https://github.com/ethereumjs

Web: https://ethereumjs.github.io/

web3j

web3j is a Java and Android knihovna pro pro integraci s Ethereum klienty a práci s chytrými kontrakty.

GitHub: https://github.com/web3j/web3j

Web: https://web3j.io

Dokumentace: https://docs.web3j.io

EtherJar

EtherJar je další Java knihovna pro integraci s Ethereem a práci s chytrými kontrakty. Je navržen pro

projekty na straně serveru založené na Java 8+ a poskytuje nízkoúrovňový přístup a vysokou úroveň obalů kolem RPC, datových struktur Etherea a přístupu k chytrým kontraktům.

GitHub: https://github.com/infinitape/etherjar

Nethereum

Nethereum je .Net integrovaná knihovna pro Ethereum.

GitHub: https://github.com/Nethereum/Nethereum

Web: http://nethereum.com/

Dokumentace: https://nethereum.readthedocs.io/en/latest/

ethers.js

The ethers.js knihovna je kompaktní, úplná, plně vybavená, rozsáhle testovaná Ethereum knihovna s licencí MIT, která od Nadace Ethereum získala grant DevEx na její rozšíření a údržbu.

GitHub odkaz: https://github.com/ethers-io/ethers.js

Dokumentace: https://docs.ethers.io

Emerald Platform

Emerald Platform poskytuje knihovny a komponenty uživatelského rozhraní k vytváření DApps nad Ethereem. Emerald JS a Emerald JS UI poskytují sady modulů a komponenty React pro vytváření JavaScript aplikací a webových stránek; Emerald SVG Icons je sada ikon souvisejících s bločenkou. Kromě JavaScript knihoven má Emerald knihovnu Rust pro provozování soukromých klíčů a podpisů transakcí. Všechny knihovny a součásti Emerald jsou licencovány na základě licence Apache, verze 2.0.range="endofrange", startref="ix_appdx-dev-tools-asciidoc10")

GitHub: https://github.com/etcdevteam/emerald-platform

Dokumentace: https://docs.etcdevteam.com

Testování chytrých kontraktů

Existuje několik běžně používaných testovacích rámců pro vývoj chytrých kontraktů, shrnuto v Shrnutí rámců testování chytrých kontraktů:

Table 11. Shrnutí rámců testování chytrých kontraktů

Rámec	**jazyk testů**	**Testovací rámec**	**Emulátor bločenky**	**Web**
Truffle	JavaScript/Solidity	Mocha	TestRPC/Ganache	https://truffleframework.com/
Embark	JavaScript	Mocha	TestRPC/Ganache	https://embark.status.im/docs/
Dapp	Solidity	ds-test (custom)	ethrun (Parity)	https://dapp.tools/dapp/
Populus	Python	pytest	Python chain emulator	https://populus.readthedocs.io

Truffle

Truffle umožňuje psaní testů jednotek v JavaScriptu (založeném na Mocha) nebo Solidity. Tyto testy probíhají proti Ganache.

Embark

Embark se integruje s Mochou pro spuštění jednotkových testů napsaných v JavaScriptu. Testy jsou zase prováděny proti kontraktům nasazeným na TestRPC / Ganache. Rámec Embark automaticky zavádí chytré kontrakty a automaticky znovu nasadí kontrakty, které byly změny. Sleduje také nasazené kontraktů a nasazuje kontrakty, pouze pokud je to skutečně nutné. Embark obsahuje testovací knihovnu, která vám umožní rychle provozovat a testovat vaše kontrakty v EVM, s funkcemi jako assert.equal. Příkaz embark test spustí všechny testovací soubory v adresáři *test*.

Dapp

Dapp používá čistý Solidity kód (knihovna pojmenována ds-test) a vestavěnou Parity Rust knihovnou zvanou ethrun pro vykonávání Ethereum bajtkódu a poté správnost volání assert. Knihovna ds-test poskytuje assert funkce pro ověřování správnosti a události pro

protokolování dat z příkazové řádky.

Assert funkce obsahují:

```
assert(bool condition)
assertEq(address a, address b)
assertEq(bytes32 a, bytes32 b)
assertEq(int a, int b)
assertEq(uint a, uint b)
assertEq0(bytes a, bytes b)
expectEventsExact(address target)
```

Příkazy protokolování budou vypisovat informace do příkazové řádky, což je činí užitečnými pro ladění:

```
logs(bytes)
log_bytes32(bytes32)
log_named_bytes32(bytes32 key, bytes32 val)
log_named_address(bytes32 key, address val)
log_named_int(bytes32 key, int val)
log_named_uint(bytes32 key, uint val)
log_named_decimal_int(bytes32 key, int val, uint decimals)
log_named_decimal_uint(bytes32 key, uint val, uint decimals)
```

Populus

Populus používá Python a jeho vlastní emulátor bločenky k spouštění kontraktů napsaných v Solidity. Testy jednotek jsou psány v Pythonu s knihovnou pytest. Populus podporuje psaní kontraktů speciálně pro testování. Tyto názvy souborů kontraktů by se měly shodovat s globálním vzorem *Test*.sol* a měly by být umístěny kdekoli v adresáři testů projektů *tests*.

Testování na bločence

Ačkoli většina testů by neměla nastat u nasazených kontraktů, chování kontraktu lze zkontrolovat prostřednictvím Ethereum klientů. Následující příkazy lze použít k posouzení stavu chytrého kontraktu Tyto příkazy by měly být zadány na terminálu geth, ačkoli je bude podporovat také jakákoli příkazové řádky web3.

Adresu kontraktu získáte na adrese *txhash*, použitím:

```
<pre data-type="programlisting">
web3.eth.getTransactionReceipt(<em>txhash</em>);
</pre>
```

Tento příkaz získá kód kontraktu nasazený na *contractaddress*; to lze použít k ověření správného nasazení:

```
<pre data-type="programlisting">
web3.eth.getCode(<em>contractaddress</em>)
</pre>
```

Tím získáte úplné protokoly kontraktu umístěného na adrese uvedené v *options*,, což je užitečné pro prohlížení historie volání v kontraktu:

```
<pre data-type="programlisting">
web3.eth.getPastLogs(<em>options</em>)
</pre>
```

Nakonec tento příkaz získá úložiště umístěné na *address* na pozici *position*:

```
<pre data-type="programlisting">
web3.eth.getStorageAt(<em>address</em>, <em>position</em>)
</pre>
```

Ganache: Místní testovací bločenka

Ganache je místní testovací bločenka, který můžete použít k nasazení kontraktů, vývoji aplikací a provádění testů. Je k dispozici jako aplikace pro stolní počítače (s grafickým uživatelským rozhraním) pro Windows, MacOS a Linux. Je také k dispozici jako obslužný program příkazové řádky s názvem ganache-cli. Další podrobnosti a pokyny k instalaci stolní aplikace Ganache najdete na stránce https://trufflеframework.com/ganache.

The ganache-cli kód se nechází na https://github.com/trufflesuite/ganache-cli/.

Chcete-li nainstalovat příkazový řádek ganache-cli, pomocí npm:

```
<pre data-type="programlisting">
$ <strong>npm install -g ganache-cli</strong>
</pre>
```

Pomocí ganache-cli můžete spustit místní bločenku pro testování následujícím způsobem:

```
<pre data-type="programlisting">
$ <strong>ganache-cli \
  --networkId=3 \
  --port="8545" \
  --verbose \
  --gasLimit=8000000 \
  --gasPrice=4000000000;</strong>
</pre>
```

Několik poznámek k této příkazové řádce:

- Zkontrolujte, zda hodnoty příznaků `--networkId` a `--port` odpovídají vašemu nastavení v *truffle.js*.
- Zkontrolujte, zda hodnota příznaku `--gasLimit` odpovídá poslední hodnotě limitu plynu na hlavní síti (např. 8,000,000 plynu) zobrazené na https://ethstats.net, abyste se vyhnuli nechtěnému vyvolání vyjímky "nedostatek plynu". Všimněte si, že „`--gasPrice` ve výši 4000000000 představuje cenu plynu 4 gwei.
- Volitelně můžete zadat hodnotu příznaku `--mnemonic` a obnovit předchozí HD peněženku a přidružené adresy.

Appendix E: web3.js tutorial

Popis

Tento tutoriál je založen na web3@1.0.0-beta.29 web3.js. Je zamýšlen jako úvod do web3.js.

JavaScript knihovna web3.js je kolekce modulů, které obsahují specifické funkce pro ekosystém Etherea, spolu s JavaScript rozhraním API kompatibilním s Ethereem, které implementuje obecnou specifikaci JSON RPC.

Chcete-li spustit tento skript, nemusíte spouštět svůj vlastní místní uzel, protože používá Infura služby [https://infura.io].

Základní interakce web3.js kontraktu neblokovaným (Async) způsobem

Zkontrolujte, zda máte platnou verzi npm:

```
<pre data-type="programlisting">
$ <strong>npm -v</strong>
5.6.0
</pre>
```

Pokud jste ještě neudělali, inicializujte npm:

```
<pre data-type="programlisting">
$ <strong>npm init</strong>
</pre>
```

Instalace základních závislostí:

```
<pre data-type="programlisting">
$ <strong>npm i command-line-args</strong>
$ <strong>npm i web3</strong>
$ <strong>npm i node-rest-client-promise</strong>
</pre>
```

Tím se aktualizuje váš konfigurační soubor *package.json* o vaše nové závislosti.

Provádění skriptu Node.js

Základní vykonávání:

```
<pre data-type="programlisting">
$ <strong>node code/web3js/web3-contract-basic-interaction.js</strong>
</pre>
```

Použijte svůj vlastní token Infura (registrujte na https://infura.io/ a uložte api-klíč do místního souboru s názvem *infura_token*):

```
<pre data-type="programlisting">
$ <strong>node code/web3js/web3-contract-basic-interaction.js \
  --infuraFileToken /path/to/file/with/infura_token</strong>
</pre>
```

nebo:

```
<pre data-type="programlisting">
$ <strong>node code/web3js/web3-contract-basic-interaction.js \
  /path/to/file/with/infura_token</strong>
</pre>
```

Tím bude soubor čten s vlastním tokenem a předán jako parametr aktuálního příkazu na příkazové řádce.

Prohlížení ukázkového skriptu

Dále se podívejme na náš ukázkový skript, *web3-contract-basic-interaction*.

Použijeme objekt Web3 k získání základního poskytovatele web3:

```
var web3 = new Web3(infura_host);
```

Poté můžeme komunikovat s web3 a vyzkoušet některé základní funkce. Uvidíme verzi protokolu:

```
web3.eth.getProtocolVersion().then(function(protocolVersion) {
      console.log(`Protocol Version: ${protocolVersion}`);
  })
```

Nyní se podívejme na aktuální cenu plynu:

```
web3.eth.getGasPrice().then(function(gasPrice) {
      console.log(`Gas Price: ${gasPrice}`);
  })
```

Jaký je poslední vytěžený blok v aktuální bločence?

```
web3.eth.getBlockNumber().then(function(blockNumber) {
      console.log(`Block Number: ${blockNumber}`);
  })
```

Interakce kontraktu

Nyní vyzkoušejme některé základní interakce s kontraktem. Pro tyto příklady použijeme WETH9_ kontrakt [https://bit.ly/2MPZZLx] na testovací síti Kovan.

Nejprve inicializujme adresu našeho kontraktu:

```
var our_contract_address = "0xd0A1E359811322d97991E03f863a0C30C2cF029C";
```

Pak se můžeme podívat na jeho zůstatek:

```
web3.eth.getBalance(our_contract_address).then(function(balance) {
      console.log(`Balance of ${our_contract_address}: ${balance}`);
})
```

a podívejte se na jeho bajtkód:

```
web3.eth.getCode(our_contract_address).then(function(code) {
      console.log(code);
})
```

Dále připravíme naše prostředí pro interakci s API Etherscan průzkumníka.

Inicializujte URL našeho kontraktu v API Etherscan prohlížeče pro bločenku Kovan:

```
var etherscan_url =
  "https://kovan.etherscan.io/api?module=contract&action=getabi&
  address=${our_contract_address}"
```

A inicializujme REST klienta pro interakci s API Etherscan:

```
var client = require('node-rest-client-promise').Client();
```

a získat klienta promise:

```
client.getPromise(etherscan_url)
```

Jakmile dostaneme platného klienta promise, můžeme získat ABI našeho kontraktu z API Etherscan:

```
.then((client_promise) => {
  our_contract_abi = JSON.parse(client_promise.data.result);
```

A nyní můžeme vytvořit náš objekt kontraktu jako příslib k pozdější konzumaci:

```
  return new Promise((resolve, reject) => {
      var our_contract = new web3.eth.Contract(our_contract_abi,
                                               our_contract_address);
      try {
        // Pokud všechno p¬jde dob¬e
        resolve(our_contract);
      } catch (ex) {
        // Pokud se n¬co pokazí
        reject(ex);
      }
    });
})
```

Pokud se náš kontrakt promise vrátí úspěšně, můžeme s ním začít komunikovat:

```
.then((our_contract) => {
```

Podívejme se na naši adresu kontraktu:

```
console.log(`Our Contract address:
            ${our_contract._address}`);
```

nebo alternativně:

```
console.log(`Our Contract address in another way:
            ${our_contract.options.address}`);
```

Nyní se podívejme na ABI našeho kontraktu:

```
console.log("Our contract abi: " +
            JSON.stringify(our_contract.options.jsonInterface));
```

Celkovou zásobu plynu našeho kontraktu vidíme pomocí zpětného volání:

```
our_contract.methods.totalSupply().call(function(err, totalSupply) {
    if (!err) {
        console.log(`Total Supply with a callback:  ${totalSupply}`);
    } else {
        console.log(err);
    }
});
```

Nebo můžeme použít vrácený promise místo předávání zpětného volání:

```
our_contract.methods.totalSupply().call().then(function(totalSupply){
    console.log(`Total Supply with a promise:  ${totalSupply}`);
}).catch(function(err) {
    console.log(err);
});
```

Asynchronní provoz s čekáním

Nyní, když jste viděli základní tutoriál, můžete vyzkoušet stejné interakce pomocí asynchronního await konstruktu. Zkontrolujte *web3-contract-basic-interaction-async-await.js* skript na *code/web3js* [http://bit.ly/2ABrFkl] a porovnejte ho s tímto tutoriálem, abyste viděli, jak se liší. Async-await se snáze čte, protože způsobuje, že se asynchronní interakce chová spíše jako sekvence blokovaných volání.

Index

@

"address object", 208
"block, defined", 19
"ERC20 token standard","METoken creation/launch example", id="ix_10tokens-asciidoc9", range="startofrange", 331
"forks", 23
"frameworks", id="ix_appdx-dev-tools-asciidoc0", range="startofrange", 493
"function invocation", 171
"message call", 26
"MetaMask","basics", id="ix_02intro-asciidoc3", range="startofrange", 51
"miners", 27
"proxy", 510
"public function", 210
"security (smart contracts)","entropy illusion threat", 283
"smart contracts","calling other contracts from within a contract", id="ix_07smart-contracts-solidity-asciidoc21", range="startofrange", 225
"uninitialized storage pointers security threat", id="ix_09smart-contracts-security-asciidoc45", range="startofrange", 307
"web3.js", 513
"Wood, Dr. Gavin","and birth of Ethereum", 35
$ symbol, 89
(contract creation transaction, 193, 21
(Ethash, 22
(Federal Information Processing Standard (FIPS), 123
(gas
    defined, 24
(Homestead, 25
(utility currency, ether as, 33
.eth nodes, 400
„finney“, 23
„odměna
    definováno“, 29
„ommer
    definované“, 27
„Parita“
    „definovaná“, 28
„Snowden, Edward“, 123
„Standard BIP-43“, 154
„Standard BIP-44“, 154
„Zvědavý drak“, 30

A

ABI (application binary interface)
    id=ix_07smart-contracts-solidity-asciidoc7
        range=startofrange, 201
account
    contract
        see=smart contracts, 45
    defined, 19
    world state and, 424
addresses
    defined, 19
    exploring transaction history of
        id=ix_02intro-asciidoc8, 60
    formats, 125
    hex encoding with checksum in

capitalization (EIP-55)
id=ix_04keys-addresses-asciidoc19, 128
ICAP encoding
id=ix_04keys-addresses-asciidoc16, 126
id=ix_04keys-addresses-asciidoc14
range=startofrange, 124
air-gapped system, 189
airdrops, 455
application binary interface (ABI)
id=ix_07smart-contracts-solidity-asciidoc8
range=startofrange, 201
application-specific integrated circuits (ASIC), 445
approve & transferFrom workflow, 329
id=ix_10tokens-asciidoc15
range=startofrange, 343
arithmetic over/underflows
id=ix_09smart-contracts-security-asciidoc6
range=startofrange, 261
preventative techniques
id=ix_09smart-contracts-security-asciidoc11, 264
real-world examples: PoWHC and batch transfer overflow, 266
vulnerability
id=ix_09smart-contracts-security-asciidoc10, 261
ASIC (application-specific integrated circuits), 445
assert function
defined, 19
Solidity and, 218
assymetric cryptography
see=public key cryptography, 45
attack surface, 358
attribution, 5
Auction DApp
backend smart contracts
id=ix_12dapps-asciidoc4, 383
ENS and
id=ix_12dapps-asciidoc9, 396
frontend user interface, 388
further decentralizing of, 390
id=ix_12dapps-asciidoc2
range=startofrange, 381
storing on Swarm
id=ix_12dapps-asciidoc6, 391
authenticity proofs, 366
await construct, 526

B

backward compatibility, Ethereum vs. Bitcoin, 44
balance, world state and, 424
Bamboo, 197
Bancor, 300
batching, 98
batchTransfer function, 266
big-endian, defined, 19
BIP-32 standard
extended public and private keys, 150
HD wallets and
id=ix_05wallets-asciidoc12, 150
BIP-39 standard, 139
deriving seed from mnemonic words, 144
generating code words with, 141
id=ix_05wallets-asciidoc6
range=startofrange, 140
libraries, 148
optional passphrase with, 147
working with mnemonic codes, 148

BIPs
see=Bitcoin improvement proposals, 45
Bitcoin
as token, 325
development culture, 44
Ethereum blockchain compared to Bitcoin blockchain, 38
Ethereum compared to, 33
Ethereum definition compared to, 83
limitations of, 35
Bitcoin Core, 34
Bitcoin improvement proposals (BIPs), 19
Hierarchical Deterministic Wallets (BIP-32/BIP-44), 137
Mnemonic Code Words (BIP-39), 139
id=ix_05wallets-asciidoc7, 140
Multipurpose HD Wallet Structure (BIP-43)
id=ix_05wallets-asciidoc10, 150
bitcoind client, 34
blind calls, 228
block gas limit, 441
block object, 207
block timestamp manipulation security threat
id=ix_09smart-contracts-security-asciidoc41
range=startofrange, 303
preventative techniques, 305
real-world example: GovernMental, 305
vulnerability, 304
blockchain
components of, 34, 38
creating contract on
id=ix_02intro-asciidoc14, 70
defined, 20
Ethereum as developer's blockchain, 45
first synchronization of
id=ix_03clients-asciidoc9, 95
on-blockchain testing, 517
recording transactions on, 190
warnings and cautions, 6
BlockOne IQ, 373
brainwallets, mnemonic words vs., 141
broadcast (multicast) oracle, 366
browser wallets, 101
burn
see=ether burn, 45
Buterin, Vitalik, 31
and birth of Ethereum, 35
and Casper, 446
and Dagger algorithm, 445
on tokens, 325
bytecode
seealso=EVM bytecode, 20
bytecode operations
id=ix_13evm-asciidoc1
range=startofrange, 419
Byzantium fork, 20, 38

C

CALL opcode, 272
calls, external
id=ix_09smart-contracts-security-asciidoc33
range=startofrange, 293
Casper, 446
Casper CBC, 446
Casper FFG, 446
chain code, 150
chain identifier, 185
ChainLink, 369
checks-effects-interactions pattern, 259
checksum

EIP-55 and
id=ix_04keys-addresses-asciidoc20, 128
in Ethereum address formats, 125
in mnemonic code word generation, 141
child private keys, 152
Cipher Browser, 101
class inheritance, 242
clients, Ethereum
and JSON-RPC API
id=ix_03clients-asciidoc12, 96
Ethereum-based networks and
id=ix_03clients-asciidoc1, 83
first synchronization of Ethereum-based blockchains
id=ix_03clients-asciidoc10, 95
full node hardware requirements, 87
Geth and
id=ix_03clients-asciidoc7, 93
id=ix_03clients-asciidoc0
range=startofrange, 83
Parity and, 90
remote
id=ix_03clients-asciidoc15, 99
running
id=ix_03clients-asciidoc5, 87
software requirements for building/running
id=ix_03clients-asciidoc6, 89
code examples, obtaining and using, 5
cold-storage wallets, 151
command-line interface, 89
compiler directive, 202
compiling
defined, 20
Faucet.sol contract
id=ix_02intro-asciidoc12, 68
protecting against overflow errors at the compiler level, 249
Vyper, 248
concurrency, nonces and, 163
consensus
Casper as Ethereum PoS algorithm, 446
controversy and competition, 447
defined, 20
Ethash as Ethereum PoW algorithm, 445
id=ix_14consensus-asciidoc0
range=startofrange, 443
principles of, 447
via proof of stake, 444
via proof of work, 444
consensus rules, 20
constant (function keyword), 211
Constantinople fork, 20, 38
constructor function, 212
adding to faucet example, 213
contract name modification security threat, 306
constructor/contract name modification security threat
preventative techniques, 307
real-world example: Rubixi, 307
vulnerability, 306
contract accounts
creating
seealso=Faucet.sol contract, 64
defined, 21
EOAs compared to, 193
seealso=smart contracts, 63
contract creation transaction
id=ix_06transactions-asciidoc12
range=startofrange, 172

contract data type, 209
contract definition, Solidity, 209
contract destruction, 213
contract invocation, 171
contract name modification/constructor security threat
    preventative techniques, 307
    real-world example: Rubixi, 307
    vulnerability, 306
contract object
    id=ix_07smart-contracts-solidity-asciidoc14 range=startofrange, 216
convert function (Vyper), 243
counterparty risk, 321
cryplet, 368
cryptographic hash functions
    id=ix_04keys-addresses-asciidoc11 range=startofrange, 121
cryptography
    asymmetric
        see=public key cryptography, 45
    defined, 105
    Ethereum addresses and
        id=ix_04keys-addresses-asciidoc15, 124
    hash functions
        id=ix_04keys-addresses-asciidoc12, 121
    id=ix_04keys-addresses-asciidoc0 range=startofrange, 105
    keys and addresses, 105
    public key cryptography and cryptocurrency
        id=ix_04keys-addresses-asciidoc1, 106
    public keys
        id=ix_04keys-addresses-asciidoc4, 110
CryptoRoulette honey pot, 311
currency units, 47

D

DAG (directed acyclic graph), 445
Dagger-Hashimoto algorithm, 445
DAO (Decentralized Autonomous Organization), 37
    about, 449
    and Ethereum Classic origins, 449
    defined, 21
    hard fork
        id=ix_appdx-forks-history-asciidoc4, 451
    id=ix_appdx-forks-history-asciidoc3 range=startofrange, 449
    reentrancy attack, 260, 388
Dapp, 516
dapp.tools, 512
DApps (decentralized applications)
    Auction DApp example
        seealso=Auction DApp, 381
    backend (smart contract), 379
    data storage, 380
    decentralized message communication protocols, 381
    defined, 21
    elements of
        id=ix_12dapps-asciidoc1, 378
    Ethereum as platform for, 42
    frontend (web user interface), 380
    id=ix_12dapps-asciidoc0 range=startofrange, 377
    IPFS and, 380

Swarm and, 381
web3 and, 43
data authentication, oracles and, 366
data field
id=ix_06transactions-asciidoc6
range=startofrange, 167
data mapping, 328
data payload, transmitting to EOAs and contracts
id=ix_06transactions-asciidoc8
range=startofrange, 170
data storage
DApps and, 380
IPFS, 380
Swarm, 381
decentralized applications
see=DApps, 45
Decentralized Autonomous Organization
see=DAO, 45
declarative programming, 196
decorators, Vyper, 245
deed
and ERC721 non-fungible token standard
id=ix_10tokens-asciidoc20, 354
as ENS top layer, 401
Auction DApp and, 381
defined, 21, 354
default function
see=fallback function, 45
default visibility specifier security problem
id=ix_09smart-contracts-security-asciidoc23
range=startofrange, 280
preventative techniques, 281
real world example: Parity Multisig Wallet hack, 281
vulnerability, 280
defensive programming, 253
delegatecall method, 228
DELEGATECALL opcode security threat
id=ix_09smart-contracts-security-asciidoc16
range=startofrange, 272
preventative techniques, 277
real-world example: Parity Multisig Wallet hack
id=ix_09smart-contracts-security-asciidoc20, 278
vulnerability
id=ix_09smart-contracts-security-asciidoc18, 272
denial of service (DoS) attacks
id=ix_09smart-contracts-security-asciidoc39
range=startofrange, 300
preventative techniques, 302
real-world example: GovernMental, 303
vulnerability, 300
deployment bytecode, 430
deployment scripts
id=ix_appdx-dev-tools-asciidoc4
range=startofrange, 499
deterministic (seeded) wallets
about, 137
defined, 134
hierarchical
see=hierarchical deterministic wallets, 45
development culture, Ethereum, 44
difficulty setting, 21
Diffie, Whitfield, 106
digital fingerprint, 121
digital signatures, 106

creating, 178
defined, 21
ECDSA and, 178
ECDSA math
id=ix_06transactions-asciidoc16, 180
id=ix_06transactions-asciidoc14
range=startofrange, 177
mechanism of operation, 178
private key and, 108
separating signing and transmission
id=ix_06transactions-asciidoc18, 187
signature prefix value (v) and public key recovery, 186
signing in practice, 182
verifying, 180
Wikipedia definition, 178
directed acyclic graph (DAG), 445
discrete logarithm problem, 107
distributed state machine, Ethereum as, 38
DoS attacks
see=denial of service attacks, 45
Dual_EC_DRBG, 123
duress wallet, 147
dynamically sized arrays, 234

E

ECDSA
see=Elliptic Curve Digital Signature Algorithm, 45
EIP-155 Simple Replay Attack Protection standard, 185
EIP-55 (Ethereum Improvement Proposal 55)
checksum for addresses
id=ix_04keys-addresses-asciidoc21, 128
detecting an error in an encoded address, 130
EIPs (Ethereum Improvement Proposals), 457
defined, 22
workflow, 457
Ellaism, 454
elliptic curve cryptography, 107
arithmetic operations, 117
basics
id=ix_04keys-addresses-asciidoc7, 111
id=ix_04keys-addresses-asciidoc8
range=startofrange, 111
libraries, 121
public key generation
id=ix_04keys-addresses-asciidoc5, 110
public key generation with, 118
Elliptic Curve Digital Signature Algorithm (ECDSA)
about, 178
defined, 22
mathematics of
id=ix_06transactions-asciidoc17, 180
signature creation, 178
transaction signing in practice, 182
Embark, 505, 516
Emerald Platform, 515
Emerald Wallet, 49
EMOD (Ethereum Modification), 455
empty input test, 124
encryption
seealso=keys and addresses, 105
ENS (Ethereum Name Service), 22
bottom layer: name owners and resolvers
id=ix_12dapps-asciidoc13, 397
choosing a valid name, 399

DApps and
id=ix_12dapps-asciidoc11, 396
design of
id=ix_12dapps-asciidoc12, 397
history of, 396
id=ix_12dapps-asciidoc10
range=startofrange, 396
managing your ENS name, 407
middle layer: .eth nodes, 400
Namehash algorithm, 397
registering a name
id=ix_12dapps-asciidoc14, 402
resolvers, 400
resolving a name
id=ix_12dapps-asciidoc16, 410
resolving a name to a Swarm hash, 411
root node ownership, 399
specification, 397
top layer: deeds, 401
Vickrey auctions, 400
entropy
defined, 22
private key generation and, 109
entropy illusion security threat, 283
preventative techniques, 283
real world example: PRNG contracts, 284
vulnerability, 283
EOA (Externally Owned Account)
basics, 63
contract accounts compared to, 193
defined, 22
keys and addresses, 105
transmitting data payload to
id=ix_06transactions-asciidoc9, 170
transmitting value to, 170
ephemeral private key, 180
equity tokens
defined, 323
utility tokens as, 323
ERC (Ethereum Request for Comments)
seealso=EIPs (Ethereum Improvement Proposals), 22
ERC20 token standard
data structures, 328
front-running vulnerability, 299
id=ix_10tokens-asciidoc5
range=startofrange, 326
interface defined in Solidity, 327
issues with ERC20 tokens, 348
optional functions, 327
required functions and events, 326
transfer functions
id=ix_10tokens-asciidoc7, 329
Vyper implementation of, 249
ERC223 token standard proposal, 349
ERC721 non-fungible token standard
id=ix_10tokens-asciidoc21
range=startofrange, 354
ERC777 token standard proposal
id=ix_10tokens-asciidoc18
range=startofrange, 351
error handling, Solidity, 218
ETC
see=Ethereum Classic, 45
ETF (EthereumFog), 456
eth nodes, 400
Ethash, 445
ETHB (EtherBTC), 455
Ether (cryptocurrency), 22
ether (generally)
gas and, 42
testnet, 56

unexpected ether security threat
id=ix_09smart-contracts-security-asciidoc12, 266
ether burn, 167
EtherBTC (ETHB), 455
Ethereum (generally)
about
id=ix_01what-is-asciidoc0, 33
and EVM, 63
as general-purpose blockchain, 38
basics
id=ix_02intro-asciidoc0, 47
birth of, 35
Bitcoin compared to, 33
blockchain components, 34, 38
clients
see=clients, Ethereum, 45
control and responsibility
id=ix_02intro-asciidoc1, 49
currency units, 47
DApps and, 42
development culture, 44
EIPs, 457
EOAs and contracts, 63
Ethereum Classic compared to, 454
fork history
id=ix_appdx-forks-history-asciidoc0, 449
four stages of development, 37
MetaMask basics
id=ix_02intro-asciidoc2, 51
purpose of, 33
reasons to learn, 45
standards, 457
Turing completeness and, 40
wallet choices, 48
web3 and, 43
Ethereum Classic (ETC)
Emerald Wallet and, 49
Ethereum compared to, 454
origins, 449, 452
Ethereum Improvement Proposals
see=EIP entries, 45
Ethereum Modification (EMOD), 455
Ethereum Name Service
see=ENS, 45
Ethereum Virtual Machine
see=EVM, 45
EthereumFog (ETF), 456
EthereumJS, 183, 514
EthereumJS helpeth, 127, 511
EtherInc (ETI), 456
EtherJar, 514
Etherpot smart contract lottery, 295
ethers.js, 515
EtherZero (ETZ), 456
ethpm project, 316
Ethstick contract, 313
ETI (EtherInc), 456
events
catching, 223
defined, 23
Solidity
id=ix_07smart-contracts-solidity-asciidoc18, 220
EVM (Ethereum Virtual Machine)
about, 417
and Ethereum high-level languages
id=ix_07smart-contracts-solidity-asciidoc1, 195
as world computer, 63
block gas limit, 441

comparison with existing technology, 419
compiling Solidity to EVM bytecode
id=ix_13evm-asciidoc4, 425
contract deployment code
id=ix_13evm-asciidoc7, 430
defined, 23
disassembling bytecode
id=ix_13evm-asciidoc9, 431
Ethereum state updating, 424
gas accounting considerations, 439
gas accounting during execution, 439
gas and, 42
id=ix_13evm-asciidoc11, 438
gas cost vs. gas price, 440
id=ix_13evm-asciidoc0
range=startofrange, 417
instruction set (bytecode operations)
id=ix_13evm-asciidoc2, 419
opcodes and gas consumption, 479
Turing completeness and Gas, 437
EVM assembly language, 23
EVM bytecode, 195
compiling source file to
id=ix_13evm-asciidoc5, 425
contract deployment code
id=ix_13evm-asciidoc8, 430
disassembling
id=ix_13evm-asciidoc10, 431
instruction set
id=ix_13evm-asciidoc3, 419
EVM OPCODES, 454
Expanse, 456
explicit typecasting, 243
extended keys, 150
external calls
id=ix_09smart-contracts-security-
asciidoc34
range=startofrange, 293
external contract referencing security threat
id=ix_09smart-contracts-security-
asciidoc27
range=startofrange, 284
preventative techniques, 288
real-world example: reentrancy honey pot
id=ix_09smart-contracts-security-
asciidoc30, 289
vulnerability
id=ix_09smart-contracts-security-
asciidoc29, 284
external function, 211
Externally Owned Account
see=EOA, 45

F

fallback function, 210, 23, 67
fast synchronization, 96
faucet, defined, 23
Faucet.sol contract (test example)
adding constructor and selfdestruct to, 213
adding events to
id=ix_07smart-contracts-solidity-
asciidoc20, 220
catching events, 223
compiling, 200
id=ix_02intro-asciidoc13, 68
creating
id=ix_02intro-asciidoc11, 64
interacting with
id=ix_02intro-asciidoc17, 73
METoken approve & transferFrom
workflow demonstration
id=ix_10tokens-asciidoc16, 343

METoken project
id=ix_10tokens-asciidoc13, 340
on the blockchain
id=ix_02intro-asciidoc15, 70
sending ether to
id=ix_02intro-asciidoc18, 75
viewing contract address in a block explorer, 73
withdrawing funds from
id=ix_02intro-asciidoc19, 76
fees
see=gas, 45
Fibonacci sequences
id=ix_09smart-contracts-security-asciidoc19
range=startofrange, 272
FIPS (Federal Information Processing Standard), 123
FIPS-202, 123
first synchronization
and JSON-RPC API
id=ix_03clients-asciidoc13, 96
Geth or Parity for, 96
of Ethereum-based blockchains
id=ix_03clients-asciidoc11, 95
floating-point representation security risk
id=ix_09smart-contracts-security-asciidoc47
range=startofrange, 311
preventative techniques, 312
real-world example: Ethstick, 313
vulnerability, 311
flood routing, 189
forks
DAO, 449
id=ix_appdx-forks-history-asciidoc5, 451
ETC, 449, 454
Ethereum and Ethereum Classic split, 454
reentrancy bug, 450
seealso=hard forks
id=ix_appdx-forks-history-asciidoc1, 449
frameworks
Embark, 505
for smart contract development
id=ix_appdx-dev-tools-asciidoc11, 516
OpenZeppelin suite, 316
id=ix_appdx-dev-tools-asciidoc8, 506
Truffle
id=ix_appdx-dev-tools-asciidoc1, 493
ZeppelinOS, 316, 510
front-running attacks
id=ix_09smart-contracts-security-asciidoc37
range=startofrange, 297
preventative techniques, 298
real-world examples: ERC 20 and Bancor, 299
vulnerability, 297
Frontier, 23, 37
full node
advantages/disadvantages, 85
Ethereum-based networks and
id=ix_03clients-asciidoc3, 84
hardware requirements, 87
function declarations, ordering of, 246
function modifiers, 215
function overloading, 242
fungible tokens
seealso=ERC20 token standard, 321

G

Ganache
advantages/disadvantages, 86
defined, 23
local blockchain testing with, 518
gas
accounting considerations, 439
accounting during execution, 439
as counter to Turing completeness, 42
avoiding calls to other contracts, 234
basics, 59
block gas limit, 441
conserving
id=ix_07smart-contracts-solidity-asciidoc24, 234
cost vs. price, 440
dynamically sized arrays and, 234
estimating cost of
id=ix_07smart-contracts-solidity-asciidoc27, 234
EVM and
id=ix_13evm-asciidoc12, 438
EVM opcodes and gas consumption, 479
negative costs, 441
on test networks, 60
tokens and, 349
transactions and
id=ix_06transactions-asciidoc3, 164
gas cost, gas price vs., 440
gas limit, 24
gasLimit field, 165
gasPrice field, 165
generator point, 111, 118
genesis block, 24
Geth (Go-Ethereum), 90
basics
id=ix_03clients-asciidoc8, 93
building from source code, 94
cloning Git repo for, 93
defined, 24
for first synchronization, 96
git, 90
global state trie, 250
Go
seealso=Geth (Go-Ethereum), 90
GovernMental Ponzi scheme
block timestamp-based attack, 305
DoS vulnerability, 303
graphics processing unit (GPU), mining and, 445

H

halting problem, 40, 437
hard forks, 24
id=ix_appdx-forks-history-asciidoc2 range=startofrange, 449
id=ix_appdx-forks-history-asciidoc6 range=startofrange, 451
seealso=DAO; other specific hard forks, e.g.: Spurious Dragon, 37
hardened derivation
for child private keys, 152
index numbers for, 152
hardware wallets, 140, 151
hash collision, 122
hash functions
id=ix_04keys-addresses-asciidoc13 range=startofrange, 121
Keccak-256, 123
main properties, 122
test vector for determining, 123
hash, defined, 24
HD wallet seed, 24

Hellman, Martin, 106
helpeth command-line tool, 127, 511
hierarchical deterministic wallets (BIP-32/BIP-44), 137
creating from root seed, 149
defined, 24
extended public and private keys, 150
hardened child key derivation, 152
HD wallets (BIP-32) and paths (BIP-43/44) id=ix_05wallets-asciidoc11, 150
index numbers for normal/hardened derivation, 152
key identifier, 153
tree structure, 153
Homestead, 37
honey pots
id=ix_09smart-contracts-security-asciidoc31 range=startofrange, 289
hybrid programming languages, 196

I

IBAN (International Bank Account Number), 126
ICAP (Inter-exchange Client Address Protocol), 25
id=ix_04keys-addresses-asciidoc17 range=startofrange, 126
Ice Age, 25, 37
ICOs
see=Initial Coin Offerings, 45
IDE (Integrated Development Environment), 25
immediate-read oracles, 364
immutable deployed code problem, 25
imperative programming, 196
implicit typecasting, 243
index numbers, for normal/hardened derivation, 152
infinite loops, 42
inheritance, 242
id=ix_07smart-contracts-solidity-asciidoc15 range=startofrange, 216
Initial Coin Offerings (ICOs)
defined, 329
DoS attacks and, 301
tokens and, 329, 359
inline assembly, 242
defined, 30
Solidity compared to Vyper, 242
Integrated Development Environment (IDE), 25
intended audience, 4
Inter-exchange Client Address Protocol (ICAP), 25
id=ix_04keys-addresses-asciidoc18 range=startofrange, 126
interface object type, 209
internal function, 211
internal transaction (message), 25, 79
International Bank Account Number (IBAN), 126
invariant checking, 266
invocation, 167, 171
IPFS (InterPlanetary File System), 25, 380

J

Jaxx, 49
desktop version, 102
mobile version, 100
JBOK wallets
seealso=nondeterministic (random) wallets, 134
JSON-RPC API
id=ix_03clients-asciidoc14 range=startofrange, 96

K

Keccak-256 hash function, 123, 26
key derivation function (KDF), 136, 25
key derivation methods, 134
key exchange protocol, 106
key pairs, 106, 108
key-stretching function, 144
key-value tuple, 38
keychains, 133
keys
    extended, 150
    path naming convention, 153
keys and addresses, 105
    seealso=cryptography; private keys; public keys, 6
keystore file, 135, 26
King of the Ether, 295

L

LevelDB, 26
libraries
    Emerald Platform, 515
    EtherJar, 514
    ethers.js, 515
    id=ix_appdx-dev-tools-asciidoc10 range=startofrange, 513
    Nethereum, 515
    web3.js, 513
    web3.py, 514
    web3j, 514
library contract, 210, 26
libsecp256k1 cryptographic library, 121
light/lightweight client, 26, 85
LLL, 197
local blockchain simulation, 86
logs, Vyper, 250

M

Mastering Ethereum Token
    see=METoken, 45
Merkle Patricia Tree, 26
Merkle, Ralph, 106
message call, 206
message communication protocols, 381
message, defined, 26
MetaMask, 49
    and testnet ether, 56
    as browser wallet, 101
    exploring transaction history of an address with id=ix_02intro-asciidoc9, 60
    network choices, 55
    sending ether from id=ix_02intro-asciidoc6, 58
    wallet setup with id=ix_02intro-asciidoc4, 52
METoken (Mastering Ethereum Token)
    approve & transferFrom workflow demonstration id=ix_10tokens-asciidoc17, 343
    creation/launch example id=ix_10tokens-asciidoc10, 331
    defined, 26
    interacting with via Truffle console id=ix_10tokens-asciidoc11, 336
    sending to contract addresses id=ix_10tokens-asciidoc14, 340
Metropolis, 27, 38
MEW
    see=MyEtherWallet, 45
migrations
    id=ix_appdx-dev-tools-asciidoc5 range=startofrange, 499

miners, 190
mining farms, 190
Mist (browser-based wallet), 103, 27
mnemonic code words, 134
    BIP-39, 139
        id=ix_05wallets-asciidoc8, 140
    generating, 141
    MetaMask and
        id=ix_02intro-asciidoc5, 53
    optional passphrase in BIP-39, 147
mobile (smartphone) wallets, 100
modifiers, 240
msg object, 206
multicast (broadcast) oracle, 366
multiple-signature (multisig) transactions, 190
mutex, 259
MyCrypto (wallet), 102
MyEtherWallet (MEW), 102, 49

N

Namehash algorithm, 397
names/naming
    see=ENS (Ethereum Name Service), 45
National Institute of Science and Technology
        (NIST), 123
negative gas, 195
neighbor nodes, 189
Nethereum, 515
networks (Ethereum)
    clients and
        id=ix_03clients-asciidoc2, 83
    defined, 27
    full nodes and
        id=ix_03clients-asciidoc4, 84
    local blockchain simulation
            advantages/disadvantages, 86
    MetaMask and, 27
    public testnet advantages/disadvantages,
            86
NFTs
    see=nonfungible tokens, 45
NIST (National Institute of Science and
        Technology), 123
node
    defined, 27
    transaction propagation, 189
Node.js, 493
non-fungible tokens (NFTs), 321
    ERC721 non-fungible token standard
        id=ix_10tokens-asciidoc22, 354
non-repudiation, 178
nonces
    concurrency, 163
    confirmation, 163
    defined, 27
    duplicated, 163
    gaps in sequence of, 162
    id=ix_06transactions-asciidoc1
        range=startofrange, 158
    keeping track of, 160
    transaction origination, 163
    world state and, 424
nondeterministic (random) wallets
    id=ix_05wallets-asciidoc2
        range=startofrange, 134
nonfungible tokens (NFTs)
    Auction DApp and, 381
    defined, 27

O

offline signing
    id=ix_06transactions-asciidoc19

range=startofrange, 187
one-way functions, 111, 121
open source licenses, 6
OpenAddressLottery honey pot, 311
OpenSSL cryptographic library, 121
OpenZeppelin, 264, 316
id=ix_appdx-dev-tools-asciidoc9
range=startofrange, 506
oracles
and data authentication, 366
broadcast/multicast, 366
client interfaces in Solidity
id=ix_11oracles-asciidoc3, 370
computation oracles
id=ix_11oracles-asciidoc2, 367
data authentication with, 366
decentralized, 369
design patterns
id=ix_11oracles-asciidoc1, 363
id=ix_11oracles-asciidoc0
range=startofrange, 361
immediate-read, 364
publish-subscribe, 364
reasons for using, 361
use cases/examples, 362
Oraclize, 367-368
id=ix_11oracles-asciidoc4
range=startofrange, 370
overflow
defined, 261
id=ix_09smart-contracts-security-asciidoc7
range=startofrange, 261
protecting against, 249

P

Parity
basics, 90
for first synchronization, 96
installing, 91
libraries for, 90
nonce counting, 162
Parity Multisig Wallet
first hack, 281
second hack
id=ix_09smart-contracts-security-asciidoc21, 278
passphrases, 147
password stretching algorithm, 136
payable function, 211
payment, 167
PBKDF2 function, 144
Populus, 517
PoS
see=proof of stake, 45
PoW
see=proof of work, 45
PoWHC
see=Proof of Weak Hands Coin, 45
pre-image, 121
prime factorization, 107
private blockchain, 86
private function, 211
private keys
defined, 29
extended, 150
generating from random number, 109
hardened child key derivation, 152
id=ix_04keys-addresses-asciidoc3
range=startofrange, 108
seealso=keys and addresses, 105
wallets and, 48, 50
PRNG (pseudorandom number generator)

contracts, 284
proof of stake (PoS)
Casper as Ethereum PoS algorithm, 446
consensus via, 444
defined, 28
Proof of Weak Hands Coin (PoWHC), 266
proof of work (PoW)
consensus via, 444
defined, 28
Ethash as Ethereum PoW algorithm, 445
propagation of transactions, 189
prototype of a function, 171
pseudorandom number generator (PRNG)
contracts, 284
public key cryptography
id=ix_04keys-addresses-asciidoc2
range=startofrange, 106
public key recovery, 186
public keys
defined, 28
elliptic curve cryptography and
id=ix_04keys-addresses-asciidoc9,
111
extended, 151
generating, 118
seealso=keys and addresses
id=ix_04keys-addresses-asciidoc6,
110
public testnets, 86
publish-subscribe oracles, 364
pure function, 211

Q

QR codes, 6

R

race conditions
seealso=front-running security threat;
reentrancy attack, 297
random (nondeterministic) wallets
id=ix_05wallets-asciidoc3
range=startofrange, 134
random numbers, private key generation from,
109
Range = "endofrange"
startref = "ix_10tokens-asciidoc6", 331
range) = "endofrange"
startref = "ix_10tokens-asciidoc5"), 331
range="endofrange", startref="ix_06transactions-
asciidoc7", 170
range="endofrange", startref="ix_09smart-
contracts-security-asciidoc49", 316
range="endofrange", startref="ix_appdx-dev-
tools-asciidoc10", 515
range=endofrange
startref=ix_01what-is-asciidoc0, 45
startref=ix_02intro-asciidoc0, 81
startref=ix_02intro-asciidoc1, 51
startref=ix_02intro-asciidoc10, 67
startref=ix_02intro-asciidoc11, 67
startref=ix_02intro-asciidoc12, 70
startref=ix_02intro-asciidoc13, 70
startref=ix_02intro-asciidoc14, 80
startref=ix_02intro-asciidoc15, 73
startref=ix_02intro-asciidoc16, 73
startref=ix_02intro-asciidoc17, 74
startref=ix_02intro-asciidoc18, 76
startref=ix_02intro-asciidoc19, 80
startref=ix_02intro-asciidoc2, 63
startref=ix_02intro-asciidoc20, 80
startref=ix_02intro-asciidoc3, 63

startref=ix_02intro-asciidoc4, 55
startref=ix_02intro-asciidoc5, 55
startref=ix_02intro-asciidoc6, 60
startref=ix_02intro-asciidoc7, 60
startref=ix_02intro-asciidoc8, 63
startref=ix_02intro-asciidoc9, 63
startref=ix_03clients-asciidoc0, 103
startref=ix_03clients-asciidoc1, 87
startref=ix_03clients-asciidoc10, 99
startref=ix_03clients-asciidoc11, 99
startref=ix_03clients-asciidoc12, 99
startref=ix_03clients-asciidoc13, 99
startref=ix_03clients-asciidoc14, 99
startref=ix_03clients-asciidoc2, 87
startref=ix_03clients-asciidoc3, 86
startref=ix_03clients-asciidoc4, 86
startref=ix_03clients-asciidoc5, 95
startref=ix_03clients-asciidoc6, 95
startref=ix_03clients-asciidoc9, 99
startref=ix_04keys-addresses-asciidoc0, 131
startref=ix_04keys-addresses-asciidoc1, 108
startref=ix_04keys-addresses-asciidoc10, 117
startref=ix_04keys-addresses-asciidoc11, 124
startref=ix_04keys-addresses-asciidoc12, 124
startref=ix_04keys-addresses-asciidoc13, 124
startref=ix_04keys-addresses-asciidoc14, 131
startref=ix_04keys-addresses-asciidoc15, 131
startref=ix_04keys-addresses-asciidoc16, 128
startref=ix_04keys-addresses-asciidoc17, 128
startref=ix_04keys-addresses-asciidoc18, 128
startref=ix_04keys-addresses-asciidoc19, 131
startref=ix_04keys-addresses-asciidoc2, 108
startref=ix_04keys-addresses-asciidoc20, 131
startref=ix_04keys-addresses-asciidoc21, 131
startref=ix_04keys-addresses-asciidoc3, 110
startref=ix_04keys-addresses-asciidoc4, 121
startref=ix_04keys-addresses-asciidoc5, 121
startref=ix_04keys-addresses-asciidoc6, 121
startref=ix_04keys-addresses-asciidoc7, 118
startref=ix_04keys-addresses-asciidoc8, 118
startref=ix_04keys-addresses-asciidoc9, 117
startref=ix_05wallets-asciidoc0, 155
startref=ix_05wallets-asciidoc1, 139-140
startref=ix_05wallets-asciidoc10, 155
startref=ix_05wallets-asciidoc11, 155
startref=ix_05wallets-asciidoc12, 153
startref=ix_05wallets-asciidoc2, 137
startref=ix_05wallets-asciidoc3, 137
startref=ix_05wallets-asciidoc4, 137
startref=ix_05wallets-asciidoc5, 155
startref=ix_05wallets-asciidoc6, 148
startref=ix_05wallets-asciidoc7, 148
startref=ix_05wallets-asciidoc8, 148
startref=ix_05wallets-asciidoc9, 148
startref=ix_06transactions-asciidoc0, 191
startref=ix_06transactions-asciidoc1, 164
startref=ix_06transactions-asciidoc10, 172
startref=ix_06transactions-asciidoc11, 172
startref=ix_06transactions-asciidoc12, 177
startref=ix_06transactions-asciidoc13, 177

startref=ix_06transactions-asciidoc14, 186
startref=ix_06transactions-asciidoc15, 186
startref=ix_06transactions-asciidoc16, 182
startref=ix_06transactions-asciidoc17, 182
startref=ix_06transactions-asciidoc18, 189
startref=ix_06transactions-asciidoc19, 189
startref=ix_06transactions-asciidoc2, 164
startref=ix_06transactions-asciidoc20, 189
startref=ix_06transactions-asciidoc3, 167
startref=ix_06transactions-asciidoc4, 167
startref=ix_06transactions-asciidoc5, 172
startref=ix_06transactions-asciidoc6, 170
startref=ix_06transactions-asciidoc8, 172
startref=ix_06transactions-asciidoc9, 172
startref=ix_07smart-contracts-solidity-asciidoc0, 237
startref=ix_07smart-contracts-solidity-asciidoc1, 197
startref=ix_07smart-contracts-solidity-asciidoc10, 233
startref=ix_07smart-contracts-solidity-asciidoc11, 206
startref=ix_07smart-contracts-solidity-asciidoc12, 209
startref=ix_07smart-contracts-solidity-asciidoc13, 212
startref=ix_07smart-contracts-solidity-asciidoc14, 218
startref=ix_07smart-contracts-solidity-asciidoc15, 218
startref=ix_07smart-contracts-solidity-asciidoc16, 218
startref=ix_07smart-contracts-solidity-asciidoc17, 218
startref=ix_07smart-contracts-solidity-asciidoc18, 225
startref=ix_07smart-contracts-solidity-asciidoc19, 225
startref=ix_07smart-contracts-solidity-asciidoc2, 197
startref=ix_07smart-contracts-solidity-asciidoc20, 225
startref=ix_07smart-contracts-solidity-asciidoc21, 233
startref=ix_07smart-contracts-solidity-asciidoc22, 233
startref=ix_07smart-contracts-solidity-asciidoc23, 233
startref=ix_07smart-contracts-solidity-asciidoc24, 237
startref=ix_07smart-contracts-solidity-asciidoc25, 237
startref=ix_07smart-contracts-solidity-asciidoc26, 237
startref=ix_07smart-contracts-solidity-asciidoc27, 237
startref=ix_07smart-contracts-solidity-asciidoc3, 237
startref=ix_07smart-contracts-solidity-asciidoc4, 237
startref=ix_07smart-contracts-solidity-asciidoc5, 201
startref=ix_07smart-contracts-solidity-asciidoc6, 201
startref=ix_07smart-contracts-solidity-asciidoc7, 203
startref=ix_07smart-contracts-solidity-asciidoc8, 203
startref=ix_07smart-contracts-solidity-asciidoc9, 203
startref=ix_08smart-contracts-vyper-asciidoc0, 251

startref=ix_08smart-contracts-vyper-asciidoc1, 245
startref=ix_08smart-contracts-vyper-asciidoc2, 245
startref=ix_09smart-contracts-security-asciidoc0, 317
startref=ix_09smart-contracts-security-asciidoc1, 317
startref=ix_09smart-contracts-security-asciidoc10, 264
startref=ix_09smart-contracts-security-asciidoc11, 265
startref=ix_09smart-contracts-security-asciidoc12, 272
startref=ix_09smart-contracts-security-asciidoc13, 272
startref=ix_09smart-contracts-security-asciidoc14, 272
startref=ix_09smart-contracts-security-asciidoc15, 270
startref=ix_09smart-contracts-security-asciidoc16, 280
startref=ix_09smart-contracts-security-asciidoc17, 280
startref=ix_09smart-contracts-security-asciidoc18, 277
startref=ix_09smart-contracts-security-asciidoc19, 277
startref=ix_09smart-contracts-security-asciidoc2, 316
startref=ix_09smart-contracts-security-asciidoc20, 280
startref=ix_09smart-contracts-security-asciidoc21, 280
startref=ix_09smart-contracts-security-asciidoc22, 280
startref=ix_09smart-contracts-security-asciidoc23, 282
startref=ix_09smart-contracts-security-asciidoc24, 282
startref=ix_09smart-contracts-security-asciidoc25, 282
startref=ix_09smart-contracts-security-asciidoc26, 282
startref=ix_09smart-contracts-security-asciidoc27, 291
startref=ix_09smart-contracts-security-asciidoc28, 291
startref=ix_09smart-contracts-security-asciidoc29, 288
startref=ix_09smart-contracts-security-asciidoc3, 260
startref=ix_09smart-contracts-security-asciidoc30, 291
startref=ix_09smart-contracts-security-asciidoc31, 291
startref=ix_09smart-contracts-security-asciidoc32, 291
startref=ix_09smart-contracts-security-asciidoc33, 296
startref=ix_09smart-contracts-security-asciidoc34, 296
startref=ix_09smart-contracts-security-asciidoc35, 296
startref=ix_09smart-contracts-security-asciidoc36, 296
startref=ix_09smart-contracts-security-asciidoc37, 300
startref=ix_09smart-contracts-security-asciidoc38, 300
startref=ix_09smart-contracts-security-asciidoc39, 303

startref=ix_09smart-contracts-security-asciidoc4, 260
startref=ix_09smart-contracts-security-asciidoc40, 303
startref=ix_09smart-contracts-security-asciidoc41, 305
startref=ix_09smart-contracts-security-asciidoc42, 305
startref=ix_09smart-contracts-security-asciidoc43, 311
startref=ix_09smart-contracts-security-asciidoc44, 311
startref=ix_09smart-contracts-security-asciidoc45, 311
startref=ix_09smart-contracts-security-asciidoc46, 310
startref=ix_09smart-contracts-security-asciidoc47, 313
startref=ix_09smart-contracts-security-asciidoc48, 313
startref=ix_09smart-contracts-security-asciidoc5, 258
startref=ix_09smart-contracts-security-asciidoc6, 266
startref=ix_09smart-contracts-security-asciidoc7, 266
startref=ix_09smart-contracts-security-asciidoc8, 266
startref=ix_09smart-contracts-security-asciidoc9, 266
startref=ix_10tokens-asciidoc0, 360
startref=ix_10tokens-asciidoc1, 321
startref=ix_10tokens-asciidoc10, 348
startref=ix_10tokens-asciidoc11, 339
startref=ix_10tokens-asciidoc12, 339
startref=ix_10tokens-asciidoc13, 342
startref=ix_10tokens-asciidoc14, 342
startref=ix_10tokens-asciidoc15, 348
startref=ix_10tokens-asciidoc16, 348
startref=ix_10tokens-asciidoc17, 348
startref=ix_10tokens-asciidoc18, 353
startref=ix_10tokens-asciidoc19, 353
startref=ix_10tokens-asciidoc2, 325
startref=ix_10tokens-asciidoc20, 356
startref=ix_10tokens-asciidoc21, 356
startref=ix_10tokens-asciidoc22, 356
startref=ix_10tokens-asciidoc23, 356
startref=ix_10tokens-asciidoc24, 358
startref=ix_10tokens-asciidoc25, 358
startref=ix_10tokens-asciidoc3, 325
startref=ix_10tokens-asciidoc4, 356
startref=ix_10tokens-asciidoc7, 330
startref=ix_10tokens-asciidoc8, 330
startref=ix_10tokens-asciidoc9, 348
startref=ix_11oracles-asciidoc0, 376
startref=ix_11oracles-asciidoc1, 366
startref=ix_11oracles-asciidoc2, 369
startref=ix_11oracles-asciidoc3, 375
startref=ix_11oracles-asciidoc4, 375
startref=ix_11oracles-asciidoc5, 375
startref=ix_12dapps-asciidoc0, 415
startref=ix_12dapps-asciidoc1, 381
startref=ix_12dapps-asciidoc10, 413
startref=ix_12dapps-asciidoc11, 413
startref=ix_12dapps-asciidoc12, 402
startref=ix_12dapps-asciidoc13, 400
startref=ix_12dapps-asciidoc14, 407
startref=ix_12dapps-asciidoc15, 407
startref=ix_12dapps-asciidoc16, 413
startref=ix_12dapps-asciidoc2, 414
startref=ix_12dapps-asciidoc3, 414
startref=ix_12dapps-asciidoc4, 388

startref=ix_12dapps-asciidoc5, 388
startref=ix_12dapps-asciidoc6, 395
startref=ix_12dapps-asciidoc7, 395
startref=ix_12dapps-asciidoc8, 395
startref=ix_12dapps-asciidoc9, 413
startref=ix_13evm-asciidoc0, 442
startref=ix_13evm-asciidoc1, 423
startref=ix_13evm-asciidoc10, 437
startref=ix_13evm-asciidoc11, 442
startref=ix_13evm-asciidoc12, 442
startref=ix_13evm-asciidoc2, 423
startref=ix_13evm-asciidoc3, 423
startref=ix_13evm-asciidoc4, 429
startref=ix_13evm-asciidoc5, 429
startref=ix_13evm-asciidoc6, 429
startref=ix_13evm-asciidoc7, 431
startref=ix_13evm-asciidoc8, 431
startref=ix_13evm-asciidoc9, 437
startref=ix_14consensus-asciidoc0, 448
startref=ix_appdx-dev-tools-asciidoc0, 511
startref=ix_appdx-dev-tools-asciidoc1, 504
startref=ix_appdx-dev-tools-asciidoc11, 519
startref=ix_appdx-dev-tools-asciidoc12, 519
startref=ix_appdx-dev-tools-asciidoc13, 519
startref=ix_appdx-dev-tools-asciidoc2, 504
startref=ix_appdx-dev-tools-asciidoc3, 497
startref=ix_appdx-dev-tools-asciidoc4, 501
startref=ix_appdx-dev-tools-asciidoc5, 501
startref=ix_appdx-dev-tools-asciidoc6, 501
startref=ix_appdx-dev-tools-asciidoc7, 504
startref=ix_appdx-dev-tools-asciidoc8, 510
startref=ix_appdx-dev-tools-asciidoc9, 510
startref=ix_appdx-forks-history-asciidoc0, 456
startref=ix_appdx-forks-history-asciidoc1, 456
startref=ix_appdx-forks-history-asciidoc2, 456
startref=ix_appdx-forks-history-asciidoc3, 454
startref=ix_appdx-forks-history-asciidoc4, 454
startref=ix_appdx-forks-history-asciidoc5, 454
startref=ix_appdx-forks-history-asciidoc6, 454
startref=ix_appdx-web3js-tutorial-asciidoc0, 526
startref=ix_appdx-web3js-tutorial-asciidoc1, 526
receipt, defined, 28
Recursive Length Prefix (RLP), 158, 29
reentrancy attacks
  blind calls and, 228
  defined, 28
  id=ix_09smart-contracts-security-asciidoc3 range=startofrange, 255
  preventative techniques, 259
  real-world example: DAO attack, 260
  vulnerability
    id=ix_09smart-contracts-security-asciidoc5, 255
reentrancy bug, 450
  attack flow, 450
  technical details, 450
reentrancy honey pot security threat
  id=ix_09smart-contracts-security-asciidoc32 range=startofrange, 289
reference specification, 34
registering a name
  id=ix_12dapps-asciidoc15

range=startofrange, 402
Remix IDE, 68
id=ix_02intro-asciidoc16
range=startofrange, 71
remote clients
browser wallets, 101
id=ix_03clients-asciidoc16
range=startofrange, 99
mobile wallets, 100
wallet compared to, 85
Remote Procedure Call (RPC) commands
see=JSON-RPC API, 45
require function, 218
resolver contracts, 400
revert function, 219
RLP (Recursive Length Prefix), 158, 29
root seeds, creating HD wallets from, 149
Ropsten Test Network, 56
RPC (Remote Procedure Call) commands
see=JSON-RPC API, 45
Rubixi pyramid scheme, 307
runtime bytecode, 430
Rust, 90

S

SafeMath library, 264
salts, 144
Satoshi Nakamoto, 29
SchellingCoin protocol, 370
Schneier, Bruce, 121
SECG (Standards for Efficient Cryptography Group), 120
secp256k1 elliptic curve, 119, 121
id=ix_04keys-addresses-asciidoc10
range=startofrange, 112
secret keys
seealso=private keys, 29
Secure Hash Algorithm
see=SHA entries, 45
security (smart contracts)
arithmetic over/underflow threat
id=ix_09smart-contracts-security-asciidoc8, 261
best practices, 253
block timestamp manipulation threat
id=ix_09smart-contracts-security-asciidoc42, 303
constructors and contract name-change threat, 306
contract libraries for, 316
default visibility specifier threat
id=ix_09smart-contracts-security-asciidoc24, 280
DELEGATECALL opcode threat
id=ix_09smart-contracts-security-asciidoc17, 272
denial of service attacks
id=ix_09smart-contracts-security-asciidoc40, 300
external contract referencing threat
id=ix_09smart-contracts-security-asciidoc28, 284
floating-point problem
id=ix_09smart-contracts-security-asciidoc48, 311
id=ix_09smart-contracts-security-asciidoc0
range=startofrange, 253
race conditions/front running threat
id=ix_09smart-contracts-security-asciidoc38, 297
reentrancy attacks
id=ix_09smart-contracts-security-

asciidoc4, 255
risks and antipatterns
id=ix_09smart-contracts-security-asciidoc2, 254
short address/parameter attack, 291
token standard implementation choices, 357
tx.origin authentication threat
id=ix_09smart-contracts-security-asciidoc49, 313
unchecked CALL return value threat
id=ix_09smart-contracts-security-asciidoc35, 293
unexpected ether threat
id=ix_09smart-contracts-security-asciidoc13, 266
uninitialized storage pointer threat
id=ix_09smart-contracts-security-asciidoc43, 307
seeded wallets
see=deterministic wallets, 45
seeds
deriving from mnemonic code words, 144
mnemonic code words for, 134
seealso=mnemonic code words, 139
optional passphrase with, 147
seealso=root seeds, 24
selfdestruct function, 213, 267
SELFDESTRUCT opcode, 195, 213
semantic versioning, 198
Serenity, 29, 38
Serpent, 197, 29
SGX (Software Guard eXtensions), 367
SHA (Secure Hash Algorithm), 29
SHA-3 Hash Function, 123
shell commands, 89
short address/parameter attack, 291
preventative techniques, 293
vulnerability, 292
side effects, 196
single-instance private blockchain, 86
singleton, 29
smart contracts
ABI
id=ix_07smart-contracts-solidity-asciidoc9, 201
addressing an existing instance, 227
and Ethereum high-level languages
id=ix_07smart-contracts-solidity-asciidoc2, 195
as DApp backend, 379
id=ix_12dapps-asciidoc5, 383
basics, 63
building with Solidity
id=ix_07smart-contracts-solidity-asciidoc3, 197
call method
id=ix_07smart-contracts-solidity-asciidoc23, 228
constructor function, 212
creating new instance, 225
defined, 193, 21, 29
delegatecall method, 228
deleting, 195
EOAs compared to, 193
ether and, 63
gas considerations
id=ix_07smart-contracts-solidity-asciidoc25, 234
id=ix_07smart-contracts-solidity-asciidoc0
range=startofrange, 193
inheritance

id=ix_07smart-contracts-solidity-asciidoc16, 216
life cycle of, 194
on-platform libraries, 316
security
id=ix_09smart-contracts-security-asciidoc1, 253
Solidity and
id=ix_07smart-contracts-solidity-asciidoc5, 197
test frameworks
id=ix_appdx-dev-tools-asciidoc12, 516
transmitting data payload to
id=ix_06transactions-asciidoc10, 170
transmitting value to, 170
using Truffle to deploy, 498
Vyper and
see=Vyper, 45
smartphones
see=mobile (smartphone) wallets, 45
Software Guard eXtensions (SGX), 367
solc (Solidity compiler), 200
Solidity, 197
adding constructor/selfdestruct to faucet example, 213
building a smart contract with
id=ix_07smart-contracts-solidity-asciidoc4, 197
calling other contracts from within a contract
id=ix_07smart-contracts-solidity-asciidoc22, 225
class inheritance, 242
compiling source file to EVM bytecode
id=ix_13evm-asciidoc6, 425
contract constructor function, 212
contract definition, 209
contract destruction, 213
contract inheritance
id=ix_07smart-contracts-solidity-asciidoc17, 216
data types
id=ix_07smart-contracts-solidity-asciidoc11, 204
default visibility specifier problem
id=ix_09smart-contracts-security-asciidoc25, 280
defined, 30
development environment, 199
downloading/installing, 198
error handling, 218
event objects
id=ix_07smart-contracts-solidity-asciidoc19, 220
faucet.sol and, 64
function modifiers, 215
function ordering, 246
function overloading, 242
function syntax, 210
functions
id=ix_07smart-contracts-solidity-asciidoc13, 210
gas considerations
id=ix_07smart-contracts-solidity-asciidoc26, 234
modifiers, 240
oracle client interfaces in
id=ix_11oracles-asciidoc5, 370
predefined global variables/functions
id=ix_07smart-contracts-solidity-asciidoc12, 206

programming with
id=ix_07smart-contracts-solidity-asciidoc10, 204
selecting compiler and language version, 202
selecting version of, 198
selfdestruct function, 213
smart contracts and
id=ix_07smart-contracts-solidity-asciidoc6, 197
variable ordering, 246
variable typecasting, 243
Vyper compared to
id=ix_08smart-contracts-vyper-asciidoc1, 239
writing a simple program in, 199
Solidity compiler (solc), 200
Solidity inline assembly, 242, 30
Spurious Dragon, 183, 37
SputnikVM, 513
Standards for Efficient Cryptography Group (SECG), 120
Status (mobile wallet), 101
storage
see=data storage, 45
storage pointers, uninitialized
id=ix_09smart-contracts-security-asciidoc44
range=startofrange, 307
stub, 209
submarine sends, 299
SUICIDE
see=SELFDESTRUCT opcode, 45
Swarm, 381
defined, 30
installing and initializing, 391
resolving a name to a Swarm hash, 411
storing Auction DApp on
id=ix_12dapps-asciidoc7, 391
uploading files to
id=ix_12dapps-asciidoc8, 393
Swarm hash, resolving a name to, 411
synchronization
see=fast synchronization, 45
see=first synchronization, 45
szabo, defined, 30
Szabo, Nick, 193

T

Tangerine Whistle, 30, 37
TEEs (trusted execution environments), 367
terminal applications, 89
test ether
obtaining, 56
sending
id=ix_02intro-asciidoc7, 58
test frameworks
for smart contract development
id=ix_appdx-dev-tools-asciidoc13, 516
on-blockchain testing, 517
test vector, determining hash functions with, 123
testnet
defined, 30
ether for, 56
public, 86
throw function, 219
TLSNotary proofs, 367
token standards (generally)
defined, 356
extensions to, 358
implementation choices, 357

purpose of, 356
reasons to use, 357
seealso=specific standards, e.g.: ERC20 token standard
id=ix_10tokens-asciidoc24, 356
tokens
counterparty risk, 321
ERC20 standard
id=ix_10tokens-asciidoc6, 326
ERC223 standard proposal, 349
ERC721 non-fungible token standard
id=ix_10tokens-asciidoc23, 354
ERC777 standard proposal
id=ix_10tokens-asciidoc19, 351
fungibility, 321
gas and, 349
ICOs and, 359
id=ix_10tokens-asciidoc0
range=startofrange, 319
intrinsicality, 322
on Ethereum
id=ix_10tokens-asciidoc4, 325
reasons to adopt, 325
uses of
id=ix_10tokens-asciidoc1, 319
using token standards
id=ix_10tokens-asciidoc25, 356
utility/equity types
id=ix_10tokens-asciidoc2, 322
Town Crier, 367
transaction call, 206
transaction fees
see=gas, 45
transaction receipt, 220
transactions
as atomic, 195
basic structure, 157
contract creation
id=ix_06transactions-asciidoc13, 172
defined, 30
digital signatures and
id=ix_06transactions-asciidoc15, 177
gas
id=ix_06transactions-asciidoc4, 164
id=ix_06transactions-asciidoc0
range=startofrange, 157
multiple-signature, 190
nonces
id=ix_06transactions-asciidoc2, 158
propagation of, 189
raw transaction creation with EIP-1455, 185
raw transaction creation/signing, 183
recipient of, 167
recording on the blockchain, 190
separating signing and transmission
id=ix_06transactions-asciidoc20, 187
signature prefix value (v) and public key recovery, 186
signing in practice, 182
smart contracts and, 194
transmitting data payload to EOAs and contracts
id=ix_06transactions-asciidoc11, 170
transmitting value to EOAs and contract, 170
value and data fields
id=ix_06transactions-asciidoc5, 167
warnings and cautions, 6
transfer function
ERC20 token standard
id=ix_10tokens-asciidoc8, 329

to reduce reentrancy vulnerabilities, 259
trapdoor functions, 107
tree structure, navigating, 153
TrueBit, 369
Truffle
as test framework, 516
configuring, 497
console
id=ix_appdx-dev-tools-asciidoc7, 501
contract deployment with, 498
creating a project directory
id=ix_appdx-dev-tools-asciidoc3, 495
defined, 31
id=ix_appdx-dev-tools-asciidoc2
range=startofrange, 493
installing, 493
integrating a prebuilt Truffle project, 494
interacting with METoken via Truffle console
id=ix_10tokens-asciidoc12, 336
migrations
id=ix_appdx-dev-tools-asciidoc6, 499
running test transaction with, 223
Truffle Box, 494
Trust Wallet, 101
trusted execution environments (TEEs), 367
trustless systems
seealso=oracles, 361
Turing completeness
as feature, 41
defined, 31
Ethereum and, 40
EVM and, 417, 437
implications of, 41
Turing, Alan, 40
tx object, 207
tx.origin authentication security threat
preventative techniques, 315
vulnerability, 314
typecasting, 243
typographical conventions, 4

U

unchecked CALL return value security threat
id=ix_09smart-contracts-security-asciidoc36
range=startofrange, 293
preventative techniques, 294
real-world example: Etherpot and King of the Ether, 295
vulnerability, 294
underflow, 261
id=ix_09smart-contracts-security-asciidoc9
range=startofrange, 261
unexpected ether
preventative techniques, 270
security threat from
id=ix_09smart-contracts-security-asciidoc14, 266
vulnerability
id=ix_09smart-contracts-security-asciidoc15, 266
uninitialized storage pointers security threat
preventative techniques, 310
real-world examples: OpenAddressLottery and CryptoRoulette honey pots, 311
vulnerability
id=ix_09smart-contracts-security-asciidoc46, 308
Universal Turing machine (UTM), 41
user interface, as DApp frontend, 380

utilities, 511
dapp.tools, 512
EthereumJS helpeth, 511
SputnikVM, 513
utility tokens
defined, 323
equity tokens disguised as, 323
issues to consider when using
id=ix_10tokens-asciidoc3, 323
UTM (Universal Turing machine), 41

V

value field
id=ix_06transactions-asciidoc7
range=startofrange, 167
variable declarations, ordering of, 246
version pragma, 202
Vickrey auctions, 400
view (function keyword), 211
visibility specifiers
id=ix_09smart-contracts-security-asciidoc26
range=startofrange, 280
vulnerabilities
seealso=security; specific attacks/vulnerabilities, 239
Vyper, 197
class inheritance, 242
compilation, 248
contract vulnerabilities and, 239
decorators, 245
defined, 31
function ordering, 246
function overloading, 242
id=ix_08smart-contracts-vyper-asciidoc0
range=startofrange, 239
modifiers, 240
overflow protection, 249
preconditions/postconditions, 245
reading/writing data, 250
Solidity compared to
id=ix_08smart-contracts-vyper-asciidoc2, 239
variable ordering, 246
variable typecasting, 243

W

wallets
browser wallets, 101
choosing, 48
cold-storage wallets, 151
creating HD wallets from root seed, 149
defined, 133, 31, 48
deterministic, 134, 137
duress wallet, 147
Emerald Wallet, 49
HD
see=hierarchical deterministic wallets, 45
id=ix_05wallets-asciidoc0
range=startofrange, 133
Jaxx, 100, 102, 49
MetaMask
see=MetaMask, 45
Mist, 103, 27
mnemonic codes (BIP-39), 139
id=ix_05wallets-asciidoc9, 140
mobile, 100
MyCrypto, 102
MyEtherWallet, 102, 49
nondeterministic
id=ix_05wallets-asciidoc4, 134

Parity Multisig Wallet hacks, 281
id=ix_09smart-contracts-security-asciidoc22, 278
remote clients compared to, 85
technology overview
id=ix_05wallets-asciidoc1, 133
testnet ether and, 56
Trust, 101
warnings and cautions
avoid sending money to addresses appearing in book, 6
private key protection, 108
when using test and example material appearing in book, 6
web user interface, as DApp frontend, 380
web3, 377, 43
seealso=DApps, 31
web3.js
asynchronous operation with await, 526
contract basic interaction in nonblocked (Async) fashion, 521
contract interaction
id=ix_appdx-web3js-tutorial-asciidoc1, 523
node.js script execution, 522
reviewing demo script, 523
tutorial
id=ix_appdx-web3js-tutorial-asciidoc0, 521
web3.py, 514
web3j, 514
wei, defined, 31
Whisper, 31, 381
withdrawal of funds from contract
id=ix_02intro-asciidoc20
range=startofrange, 76
Wood, Dr. Gavin, 24
and Solidity, 197, 64
and web3, 43
world computer, Ethereum as, 33, 63
world state, 424

Y

Yellow Paper specification, 83

Z

ZeppelinOS, 316, 510
zero address
contract creation, 172
contract registration, 70
defined, 32
ZeroMQ (0MQ), 189