Code 423n4

Slingshot Contest

February 17–21

Findings & Analysis Report

Code 423n4

Overview

About C4

Code 432n4 (C4) is an open organization that consists of security researchers, auditors, developers, and individuals with domain expertise in the area of smart contracts.

A C4 code contest is an event in which community participants (Wardens), review, audit, and analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the contest outlined in this document, C4 conducted an analysis of Slingshot's trading contracts.The contest took place February 17–21, 2021.

Wardens

10 Wardens contributed reports to the Slingshot Code Contest:

- Austin WIlliams
- Christoph Michel
- Gerard Persoon
- Janbro
- Noah Citron
- Nate Oden
- Paulius
- PocoTiempo (Team)
    - Rajeev
    - Mariano Conti
    - Maurelian

Judge

This contest was judged by Zak Cole.

Final report assembled by Zak Cole, John Patten, and Adam Avenir.

Code 423n4

Summary

Overall, the results provided by the C4 analysis indicate that the Slingshot contracts are secure. The analysis yielded an aggregated total of 4 unique vulnerabilities.

Of these vulnerabilities, 1 received a risk rating of **HIGH** severity, 5 received a risk rating of **MEDIUM** severity, and 12 received a risk rating of **LOW** severity. C4 analysis also identified opportunities for **6 gas optimizations.**

Recommendations include providing more concise documentation on how the contracts and their respective functions are expected to interact and receive payloads from the Slingshot backend, as well as inter-module communications. Another recommendation was to separate the approval logic into its own contract.

The Slingshot team responded to the issues identified as a result of this code contest and provided information regarding any changes to the codebase. A small set of vulnerabilities and submissions were disputed by the Slingshot team. For each of these issues, a supporting explanation for these disputes is detailed in the Disputed Findings section.

Scope

Code

The code that was reviewed can be found within the C4 Slingshot code contest repository and comprises 651 lines of code across a total of 10 smart contracts written in the Solidity programming language.

File	Lines of Code
Slingshot.sol	170
ModuleRegistry.sol	61
Initializable.sol	67
Adminable.sol	48
UniswapModule.sol	65
SushiSwapModule.sol	66
CurveModule.sol	63
BalancerModule.sol	59
LibERC20Token.sol	21
Strings.sol	31

System Overview

Slingshot aggregates prices for available token pairs from a variety of liquidity sources across decentralized exchanges and automated market making protocols, including Uniswap, Balancer, Curve, and SushiSwap.

The **`Slingshot.sol`** contract acts as the entrypoint which handles the on-chain logic of executing a trade between liquidity sources. This logic also includes conducting any sanity checks and safety checks to ensure the security of a given transaction.

To interact with a specified market, the Slingshot system implements module contracts for each respective DEX/AMM. For example, when **`Slingshot.sol`** receives an input that defines a trade to be executed within Uniswap, it delegates this transaction to the **`UniswapModule.sol`** contract.

Severity Criteria

C4 assesses severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into 3 primary risk categories:

- Low
- Medium
- High

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided in the C4 website.

Code 423n4

Issues Found By Severity

High Severity

[H-01] Incorrectly encoded arguments to `executeTrades()` can allow tokens to be stolen
This finding combines a couple weaknesses into one attack. The first weakness is a lack of validation on arguments to **`executeTrades`**, the second is that a pre-existing fromToken balance can be used in a trade:

1. Alice wants to convert 1000 DAI to WETH. She calls executeTrades(DAI, WETH, 1000, [], 0, alice).
2. Since trades is an empty array, and finalAmountMin is 0, the result is that 100 DAI are transferred to the Slingshot contract.
3. Eve (a miner or other 'front runner') may observe this, and immediately call executeTrades(DAI, WETH, 0, [{TradeData}], 0, eve).
4. With a correctly formatted array of TradeData, Eve will receive the proceeds of converting Alice's 1000 DAI to WETH.

This issue is essentially identical to the one described in Ethereum is a Dark Forest, where locked tokens are available to anyone, and thus recovery is susceptible to front running.
It also provides an unauthorized alternative to **`rescueTokens()`**, however it is still a useful function to have, as it provides a method to recover the tokens without allowing a front runner to simulate and replay it.

Medium Severity

[M-01] Front Running/Sandwich Attacks
If a **`finalAmountMin`** is chosen that does not closely reflect the received amount one would get at the market rate (even with just 1% slippage), this could lead to the trade being frontrun and to less tokens than with a tighter slippage amount.Balancer and Curve modules don't have any slippage protection at all which makes it easy for attackers to profit from such an attack. The min amount returned is hardcoded to 1 for both protocols. The Sushiswap/Uniswap modules are vulnerable as well, depending on the calldata that is defined by the victim trader.

[M-02] Front Running/Sandwich Attacks
If tokens are accidently sent to Slingshot, arbitrary trades can be executed and those funds can be stolen by anyone. This vulnerability impacts the `rescueToken` functionality and any funds trapped in Slingshot's contract. Tokens and/or Eth have a higher likelihood of becoming trapped in Slingshot if `finalAmountMin` is not utilized properly. Recommend validating parameters in the calldata passed to modules and ensuring the fromToken and amount parameter from executeTrades is equivalent to the token being swapped and amount passed to `swap()`. Additionally, approval values can be limited to value being traded and cleared after trades are executed.

[M-03] Infinite Approval abused by malicious admin
Current Slingshot contracts assume a rapid development environment so they use a proxy pattern with a trusted admin role. We do not expect any malicious behavior from admin, however we agree that in the current setup admin potentially would be able to use unlimited approvals to steal user's funds. We consider this medium severity.

[M-04] Stuck tokens can be stolen
Any tokens in the Slingshot contract can be stolen by creating a fake token and a Uniswap pair for the stuck token and this fake token. Consider 10 WETH being stuck in the Slingshot contract. One can create a fake ERC20 token contract FAKE and a WETH <> FAKE Uniswap pair. The attacker provides a tiny amount of initial WETH liquidity (for example, 1 gwei) and some amount of FAKE tokens. The attacker then executes `executeTrades` action such that the Slingshot contract uses its Uniswap module to trade the 10 WETH into this pair.

[M-05] Admin role lockout
The `initializeAdmin()` function in Adminable.sol sets/updates admin role address in one-step. If an incorrect address (zero address or other) is mistakenly used then future administrative access or even recovering from this mistake is prevented because all `onlyAdmin` modifier functions (including `postUpgrade()` with `onlyAdminIfInitialized`, which ends up calling `initializeAdmin()`) require `msg.sender` to be the incorrectly used admin address (for which private keys may not be available to sign transactions). In such a case, contracts would have to be redeployed. Suggest using a two-step process where the new admin address first claims ownership in one transaction and a second transaction from the new admin address takes ownership.

Low Severity

[L-01] Custom pair drain
Any tokens in the module contracts that allow custom pairs to be added can be stolen by creating a fake token and a pool pair for the stuck token and this fake token. The attacker provides a tiny amount of liquidity to this pool. Then one can directly call the swap function on

the module contract to swap the stuck tokens in the pool for useless tokens. Afterwards, the stuck tokens are in the pool and can be withdrawn by withdrawing liquidity. A similar attack is even possible for modules where no custom pairs can be created by providing lots of liquidity first using flash loans, then trading in the pool and withdrawing liquidity again.

[L-02] Setting the admin in initialize or `postUpgrade` can be frontrun by an attacker
These functions can be called by anyone the first time which allows an attacker to set the owner of the contract to themselves leading to a denial of service attack whenever the project party tries to deploy these contracts on their own.

[L-03] The `ModuleRegistry::slingshot` storage variable is never read.
Recommend removing it along with the `setSlingshot` function.

[L-04] `tradeAll` can be exploited
Setting **`tradeAll`** to true and **`swapExactTokensForTokens`** to false in the swap function of UniswapModule or SushiSwapModule will cause the trade size to be incorrect. This can cause tokens to get stuck inside of the Slingshot contract if a trade is executed with these parameters, and the input token is worth more than the output token. (For example, if you attempted to swap 1 WETH to DAI like this, you would receive just 1 DAI, with the excess 0.999 ETH left inside the Uniswap/Sushiswap module.)

[L-05] Curve module does not validate `i` and `j` index are associated with the iToken and jToken addresses.
These parameters i and j are not validated to be the ones correlated to addresses iToken and jToken. Some simple input validation could prevent loss of funds from malformed inputs.

[L-06] No access restrictions
Anyone can call **`executeTrades`**, as there is no access control for this function. As very specific input parameters are required, unpredictable situations can occur when everyone can call the function **`executeTrades`**. At the very least the event log could be spammed by calling the function **`executeTrades`** with an empty array for trades.

[L-07] `swap()` payable for no reason
The **`swap()`** function in the module contracts is payable although it does not use the msg.value and operates on WETH. Slingshot contract handles all the conversions ETH <-> WETH so modules should only expect WETH. Declaring a function as payable means it can also receive ETH. However, there is no function to later extract ETH which is accidentally sent to this function, leaving ETH stuck there forever.

[L-08] Useless fallback function

The Slingshot contract has declared a fallback function which is not directly used. The Slingshot contract should not accept ETH using a fallback function as in case someone accidentally sends ETH directly to this contract, it then can be swapped by anyone monitoring the contract.

[L-09] `executeTrades` recipient can be 0x0

The function **`executeTrades`** does not check that the recipient is set. When the recipient is not set, a default address of 0x0 is used. All the swapped tokens will be forwarded to this address. If the recipient is not set (0x0), it means that tokens are burned (sent to an address you can't recover them from). It depends on the intentions. Maybe the user wants to burn their tokens? Then 0x0 as a recipient would make sense.

[L-10] Modules do not check for duplicate tokens

Each module separately accepts and uses the addresses of input and output tokens. However, function **`executeTrades`** expects that **`fromToken`** and **`toToken`** are the same in every module. Parameters for each swap module are provided in the bytes **`encodedCalldata`**. These modules cannot necessarily receive the same values for parameters such as input token and output token. Also, theoretically, they can be different than the ones that are passed to the function **`executeTrades`** in the Slingshot contract. If different output tokens are provided as call data for each module, then the user will only receive the toToken that is declared in function **`executeTrades`**. All other tokens will be left in the contract. This is possible, for instance, when UI forms the incorrect call data for the trades and when **`finalAmountMin`** is very low (e.g. 0) so the recipient would be happy with any amount received.

[L-12] The IWETH interface is incorrect

First, the **`balanceOf`** function on the WETH contract accepts an address parameter, not a uint256 parameter (see line 32 of the WETH9 contract). Second, the function is not declared view, which means calls to the **`balanceOf`** function on any contract wrapped with this interface could potentially change state. Since the Byzantium hardfork, the EVM can enforce that view functions use the **`STATICCALL`** opcode, ensuring that view functions cannot mutate state. (See here for more info.)

Code 423n4

Gas Optimizations

[G-01] Make WETH and ETH_ADDRESS constant

[G-02] `trades.length` **reads from storage multiple times**

[G-03] Trade directly with uniswap pools

[G-04] The LibERC20Token.approveIfBelow function uses an unnecessarily gas-intensive pattern to set a max approval.

[G-05] Remove SafeMath from UniswapModule and SushiSwapModule

[G-06] Make executeTrades and swap external

Code 423n4

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provide code, but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.