SmartDec

Fysical Smart Contracts Security Analysis

This report is public.

Published: February 28, 2018

Abstract

In this report we consider the security of the Fysical project. Our task is to find and describe security issues in the Fysical smart contract.

Procedure

In our analysis we consider Fysical smart contract code (Fysical_source_2018-01-28.tar, sha1sum 03f96192c53cf95f13b2da76094d9e81e233f7a4) and documentation (Fysical_Technical_Detail.txt, sha1sum ed10ce05f1e82fb687cbdbb2711673450df5b580).
The project has been analyzed in a shortened procedure:

- automated analysis
    - we scan the smart contract with our own Solidity static code analyzer SmartCheck
    - we scan the smart contract with several publicly available automated Solidity analysis tools such as Remix, Solhint, and Securify (beta version since full version was unavailable at the moment this report was made)
    - we manually verify (reject or confirm) the issues found by tools
- manual audit (manual audit was more concise than its full form because of the shortened procedure)
    - we manually analyze the smart contract for security vulnerabilities
    - we check the smart contract logic and compare it with the one described in the documentation
- report
    - we report all the issues found to the developer during the audit process
    - we check the issues fixed by the developer
    - we reflect all the gathered information in the report

The latest version of the code

We have performed the check of the fixed vulnerabilities in the latest version of code — Fysical.tar.gz, sha1sum 588b1c027fcbcbc89884cf54ea454d6bedc969c9, which contains git-repository (version on commit 3d57d5c46ebfa3f0f416ce0569d598a78c441738).

Disclaimer

The audit does not give any warranties on the security of the code. One audit cannot be considered enough. We always recommend proceeding to several independent audits and a public bug bounty program to ensure the security of the smart contracts. Besides, security audit is not an investment advice.

About The Project

Project Architecture

The smart contracts code consists of the Fysical.sol file (1034 lines of code without imports) with imports from OpenZeppelin library (the version given for analysis as part of the project).
Fysical contract inherits from the StandardToken contract from OpenZeppelin library. The contract uses the SafeMath library from OpenZeppelin library for calculations.
No tests or deploy scripts have been provided for the audit.
The project compiles successfully.

Code logic

Fysical contract implements the ERC20 token with the following parameters:

- totalSupply: 1000000000
- decimals: 9
- symbol: FYS
- name: Fysical

The main functionality of the contract is the logic described in the comments in the code and documentation: it is the platform for exchange of offchain resources, which have on-chain references. Fysical contract allows to store references to off-chain resources (possibly encrypted) and trade these resources for FYS tokens (with encryption keys transfer).

Checked vulnerabilities

We have scanned the Fysical smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered (the full list includes them but is not limited to them):

- Reentrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with (Unexpected) Throw
- DOS with (Unexpected) revert
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC20 API violation
- Malicious libraries
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level
- Address hardcoded
- Using delete for arrays
- Integer overflow/underflow
- Locked money
- Private modifier
- Revert/require functions
- Using var
- Visibility
- Using blockhash
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Automated Analysis

We used several publicly available automated Solidity analysis tools.
Securify and Remix do not support 0.4.18 compiler version; the specified version was changed in the code to 0.4.16 for these tools.
Here are the combined results of SmartCheck, Securify, and Remix.
All the issues found by tools were manually checked (rejected or confirmed).

Tool	Vulnerability	False positives	True positives
SmartCheck	Address Hardcoded		7
	Dos With Revert	21	
	Erc20 Approve		1
	Gas Limit And Loops		1
	No Payable Fallback		5
	Pragmas Version		6
	Reentrancy External Call	15	
	Unchecked Math		1
	Visibility		1
Total SmartCheck		36	22
Remix	Gas requirement of function	28	8
	Use of `assert(x)`	1	
	Variables have very similar names	12	
Total Remix		41	8
Solhint	Compiler version must be fixed		6

	Event and function names must be different	1	
	Explicitly mark visibility of state		1
Total Solhint		1	7
Overall Total		**78**	**37**

Securify* — beta version, full version is unavailable.

Cases when these issues lead to actual bugs or vulnerabilities are described in the next section.

Manual Analysis

The contract was manually analyzed in a shortened procedure, its logic was checked and compared with the one described in the comments and documentation. Besides, the results of the automated analysis were manually verified. All confirmed issues are described below.

Critical issues

Critical issues seriously endanger smart contracts security. We highly recommend fixing them.

The audit showed no critical issues.

Medium severity issues

Medium issues can influence smart contracts operation in current implementation. We highly recommend addressing them.

No tests

The provided code does not contain tests. Testing is crucial for code security. We highly recommend not only to cover the code with tests but also to make sure that the coverage is sufficient.

Modularity

It is undesirable to implement both token and non-trivial logic in one contract. If issues are found in logic after deploy and the developer has to update the contract, he/she might encounter migration problems with FYS token.
We highly recommend to make the system more modular. In this particular case, we recommend implementing two separate contracts: one of them should implement the token, and the other should implement the platform logic.

DOS

If after the `transferTokensToEscrow` function call the owner of the `tokenTransfer.source` address (one of the `TokenTransfer` for the current proposal) sets the value of `allowed` for `proposalCreator` to the maximum possible value that fits into `uint256`, then `add` at line 971 will always throw and the tokens will not get back to `tokenTransfer.source` (not to any of `source`'s since the token transfer happens in one cycle). We highly recommend fixing this bug.

The issue has been fixed and is not present in the latest version of the code. However, a new one appears: allowed value is not recovered to the previous value. We can recommend two options to fix it:

- *checking if overflow hasn't occurred. Possible variant:* `allowed[tokenTransfer.source][proposalCreator] >= allowed[tokenTransfer.source][proposalCreator] + tokenTransfer.tokenCount`;
- *or overriding ERC20 functions to ensure that allowed can't be more than total supply.*

The developer explained the new issue:
“After examining the issues you presented in the last report, and seeing confusion surrounds the safety of approvals and ERC20 in the community, we decided our escrow concept should not attempt to restore the previously allowed value. Regardless of which behavior we choose, someone's expectations will likely not be met. Since this approval value could be set explicitly through ERC20 while tokens are in escrow, we can't be sure that an account holder would want this value to be restored anyway.

If we were to check for an overflow on withdrawal, we would need to choose between allowing the transaction to succeed while explicitly setting the approval value to the maximum uint256 or failing the transaction. In the latter case, the account owner would not be able to withdraw the proposal until manually reducing the approval value.

If our chosen behavior turns out to not match an account owner's expectations, the downside is the failed execution of a transaction that could be re-executed following an approval update. But if we had chosen to restore the approval value escrow against the account owner's expectations, the downside would be an irrevocable transaction that the account owner did not desire.”

Low severity issues

Low severity issues can influence smart contracts operation in future versions of code. We recommend to take them into account.

General consideration on the contract capabilities

We want to draw attention to the fact that the contract cannot ensure security (i.e. integrity, confidentiality, and availability) of the whole platform since the platform accepts the data from the real world (off-chain).
In absence of external control over participants of the platform (e.g., based on reputation system), the following unfair behaviour scenarios are theoretically possible:

1. Decrypted resources might not met buyer's expectations after the proposal is accepted.
2. When `acceptProposal` function is called, `ResourceSet`’s creator can transfer the keys that cannot decrypt the resources, wherein the tokens are already transferred.
3. One can create `resourceSet` from another's resources and get tokens for them (including encrypted resources, transferring unsuitable keys).

4. URI standard compliance is not checked at the contract level.
The comment was added in the latest version of the code (Fysical.sol, lines 91-93):
```
"// The risk of preventing support for a future addition to the URI
syntax outweighs the benefit of validating URI
// values within this immutable smart contract, so readers of Uri
values should expect values that do not conform
// to the formal syntax of a URI."
```

5. If symmetric encryption is used, all the data will be compromised.
6. The comment states: "`The 'resourceByteCount' indicates the number of bytes contained in the resource.`" Thus, this restriction can be used only for resources whose size in bytes is less than the maximum number that fits into `uint256`.
7. Data integrity check by its size (`Checksum.resourceByteCount`) can be easily bypassed, thus we recommend considering advisability of this check.
8. Etcetera.

Out-of-gas possible

In several places of code, there are loops over arrays whose length might be pretty big, thus such loops may lead to out-of-gas denial of service.
`getEncryptedResourceDecryptionKey` function, that `buyer` calls after the proposal is accepted to get keys to decrypt the resources, contains the loop over resources from the proposal. Thus, before making a proposal buyer needs to find out whether the keys getting transaction fits into the block. Otherwise, the `resourceSet` creator will be able to set such amount of resources that `buyer` will not be able to get encryption keys after the deal is closed. However, though `buyer` will not be able to call `getEncryptedResourceDecryptionKey`, still the keys will be stored in the blockchain and it will be possible to get them by other ways.
Besides, out-of-gas is possible in `storeEncryptedDecryptionKeys` function, since it has the loop over resources and over the array of keys, which are passed as parameters. However, out-of-gas is unprofitable for the one who sets the resources and keys since he/she will not be able to get tokens for the resources.
Similarly, out-of-gas is possible in the functions that transfer tokens from/to `escrow (0 address)`. However, these functions are called by the `proposal`'s creator, who specifies the number of iterations, thus, the attack is also unprofitable.
The similar situation may occur in `validateIdSet`, but since it is called with parameters passed by the users, the attack is also unprofitable.
The comments were added in the latest version of the code:

- *Fysical.sol, lines 64-67*
```
"// Several structs in Fysical contain an unbounded array of items.
Creators of these objects should take care to not
// include array lengths that would strain the practical block sizes
and gas costs of Ethereum. Note that instead of
// referencing a large set of resources, a Resource Set creator has
the option to create an archive of these resources
// and reference the archive on Fysical."
```
- *Fysical.sol, lines 162-166*

```
" // Creators should be careful to not include so many resources that
an Ethereum transaction to accept a proposal
// might run out of gas while storing the corresponding encrypted
decryption keys.
//
// While developing reasonable filters for un-useful data in this
collection, developers should choose a practical
// maximum depth of traversal through the meta-resources, since an
infinite loop is possible."
```

Redundant field

The comment specifies, what `minimumBlockNumberForWithdrawal` field is used for:

```
Without this mechanism, the resource set creator may have had
concerns that the offer could have been withdrawn after the
acceptance operation was submitted to the Ethereum network.
```

However, the transactions are processed successively. Thus, the situation, in which `withdrawProposal` is called after `acceptProposal` for the same proposal, is impossible. Thereby, we consider `minimumBlockNumberForWithdrawal` field to be redundant and recommend removing it.

The comment was added in the latest version of the code (Fysical.sol, lines 91-93):

```
" // By including a 'minimumBlockNumberForWithdrawal' value later than the
current Ethereum block, the proposal
// creator can give the resource set creator a rough sense of how long the
proposal will remain certainly
// acceptable. This is particularly useful because the execution of an
Ethereum transaction to accept a proposal
// exposes the encrypted decryption keys to the Ethereum network regardless
of whether the transaction succeeds.
// Within the time frame that a proposal acceptance transaction will
certainly succeed, the resource creator need
// not be concerned with the possibility that an acceptance transaction
might execute after a proposal withdrawal
// submitted to the Ethereum network at approximately the same time."
```

Input validation

In some places of the code input validation is presumably missing:

- `createChecksum` function does not check that `resourceByteCount > 0`
  The issue has been fixed and is not present in the latest version of the code.
- `createProposal` function does not check that `minimumBlockNumberForWithdrawal` is later than the current block
- `createTokenTransfer` function does not check that `tokenCount > 0`, though later at token transfer at lines 935, 964, 990 it is checked with `assert` that `tokenCount > 0` (i.e. it is expected to be true)
  The issue has been fixed and is not present in the latest version of the code.

require instead of **assert**

At line 968, there should be `assert()` instead of `require()`. According to the code logic, `returnTokensFromEscrow` function is to be called only after `transferTokensToEscrow` function for the same set of `tokenTransfer`, thus the condition `tokenTransfer.tokenCount <= balances[address(0)]` must be met.
The issue has been fixed and is not present in the latest version of the code.

ERC20 approve issue

There is ERC20 approve issue. We recommend explicitly warning users not to use `approve` directly and to use `increaseApproval`/`decreaseApproval` functions (or to change the approved amount to 0 and then to the desired value) instead - link.
The comment was added in the latest version of the code (Fysical.sol, lines 69-71):

```
"// Please note that ERC20 has a well-known issue surrounding
approvals (See
// https://github.com/ethereum/EIPs/issues/20#issuecomment-
263524729). Instead of executing "approve", please
// use "increaseApproval" and "decreaseApproval" to modify approval
amounts.”
```

Redundant checks

The following check is redundant since `msg.sender` cannot be zero:

```
        require(address(0) != msg.sender);
```

Such checks can be found at lines 297, 520, 705, 801, and 833.
Besides, we recommend removing all `assert()` calls from the production version of code in order to save gas. From our viewpoint, in the contract `assert()` is used correctly and thus will not be triggered if the code does not contain bugs.
The issue has been fixed and is not present in the latest version of the code.

Pragmas version

Solidity source files indicate the versions of the compiler they can be compiled with.
Example:

```
        pragma solidity ^0.4.19; // bad: compiles w 0.4.19 and above
        pragma solidity 0.4.19; // good : compiles w 0.4.19 only
```

We recommend following the latter example, as future compiler versions may handle certain language constructions in a way the developer did not foresee. Besides, we recommend using the latest compiler version (0.4.20 at the moment).
The issue has been fixed and is not present in the latest version of the code.

OpenZeppelin Version

The contract uses unknown version of OpenZeppelin library. We highly recommend using the latest versions of frameworks and following the updates.

The comment was added in the latest version of the code (Fysical.sol, line 3):

```
"// These import statements point at unmodified files from
OpenZeppelin v1.6.0 (See https://github.com/OpenZeppelin/zeppelin-
solidity/releases/tag/v1.6.0)"
```

Conclusion

In this report we have considered the security of the Fysical smart contract. We performed our audit according to the shortened procedure described above.
The audit showed high code quality and security of the project. No serious vulnerabilities were found. However, few medium and low severity issues were found and reported to the developer. In the latest version of the code most of them (and all of the important ones) were fixed.

This analysis was performed by SmartDec.

COO Sergey Pavlin [illegible]

February 28, 2018