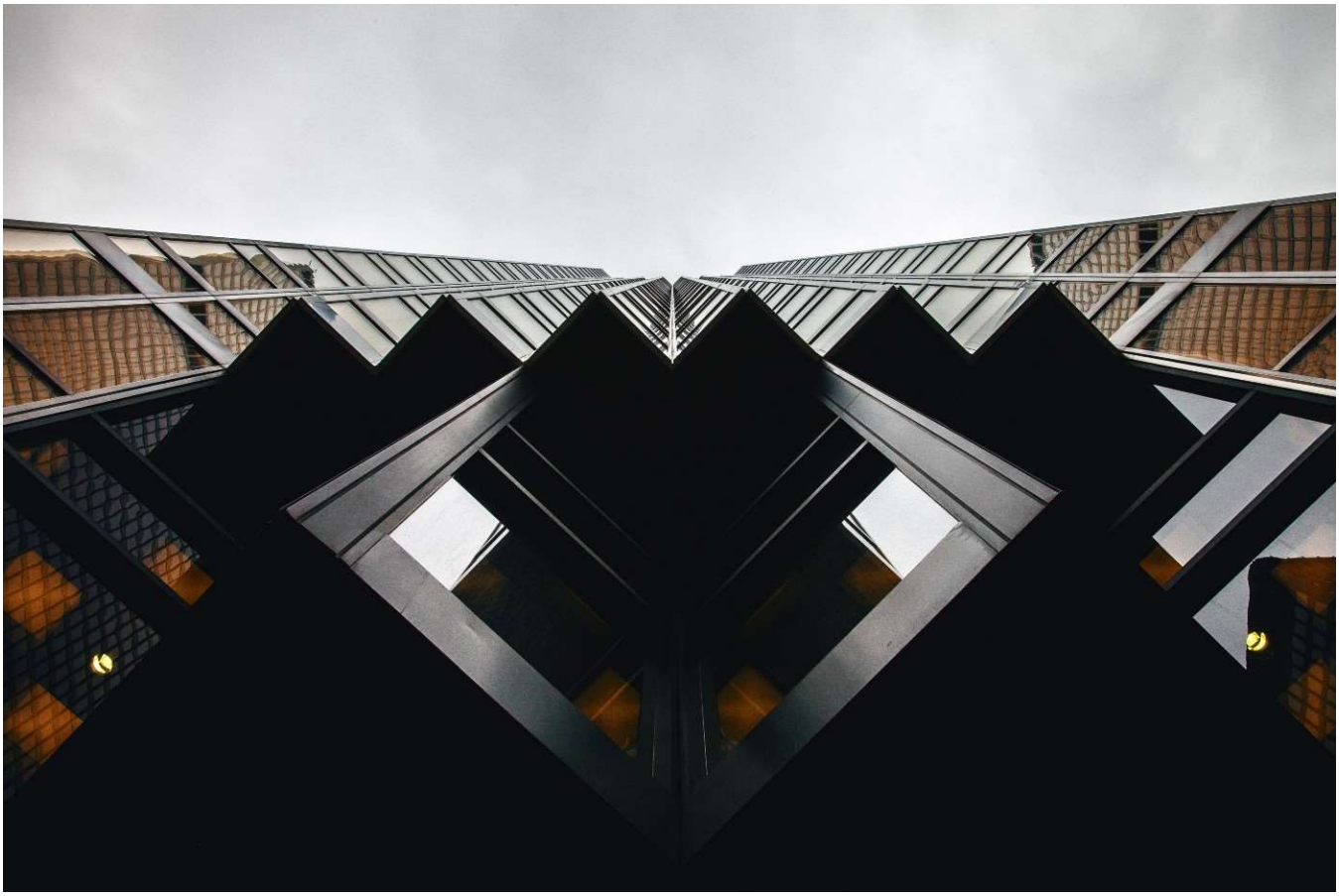
Published in SmartDec Cybersecurity Blog

Pavel Kondr Follow

Aug 21, 2018 · 8 min read · Listen

SONM Smart Contracts Security Analysis



In this report, we consider the security of the SONM project. Our task is to find and describe security issues in the smart contracts of the platform.

Disclaimer

audits and a public bug bounty program to ensure the security of smart contracts. Besides, security audit is not an inv

Summary

In this report, we have considered the security of SONM smart contracts. We performed our audit according to the procedure described below.

The initial audit has shown no critical issues. All of the medium severity issues and low severity issues that could possibly pose a threat were fixed in the latest version of the code. The rest of issues do not endanger project security. Thus, the latest version of the code is ready for the release.

General recommendations

The contracts code is of good code quality and does not contain issues that endanger project security.

Nevertheless, if the developer decides to improve the code, we recommend fixing Locked tokens issue, adding Missing checks, and fixing Costly loops.

However, these are minor issues, which do not influence code operation.

The text below is for technical use; it details the statements made in Summary and General recommendations.

Procedure

In our audit, we consider the following crucial features of the smart contract code:

1. Whether the code is secure.
2. Whether the code corresponds to the documentation (including whitepaper).

Automated analysis

- we scan project's smart contracts with our own Solidity static code analyzer SmartCheck
- we scan project's smart contracts with several publicly available automated Solidity analysis tools such as Remix and Solhint
- we manually verify (reject or confirm) all the issues found by tools

Manual audit

- we manually analyze smart contracts for security vulnerabilities
- we check smart contracts logic and compare it with the one described in the whitepaper

Report

- we reflect all the gathered information in the report

Checked vulnerabilities

We have scanned SONM smart contracts 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


- 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

- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Project overview

In our analysis we consider smart contracts code (version on commit 8b349d25214c6c600f4724e3e9ed3dbc57f9ae63). Only smart contracts in `directoryblockchain/source/contracts` are audited. Other components of the project, for example, golang-files (*.go) are out of scope of this audit.

Project architecture

For the audit, we have been provided with the truffle project. The project is an npm package, it includes tests and deploy script.

- The project successfully compiles with `truffle compile` command
- The project successfully passes all the tests with `truffle test` command

The project includes the following files containing Solidity code:

- AddressHashMap.sol
- Blacklist.sol
- DeployList.sol
- Market.sol
- MultiSigWallet.sol
- OracleUSD.sol


- SimpleGatekeeperWithLimitLive.sol
- TestnetFaucet.sol

Code logic

The project has the following smart contracts:

- **AddressHashMap.sol —** a simple smart contract that stores mapping of bytes32 values to addresses. The smart contract provides the owner with read and write functions allowing accessing the mapping. The smart contract is not used in the other audited smart contracts and it is probably intended for use in off-chain code.
- **Blacklist.sol —** a smart contract that holds various blacklists. This contract is created in the constructor of `Market` smart contract. This contract is only used by `Marketsmart` contract.
- **DeployList.sol —** a smart contract that holds a list of addresses called deployers. This smart contract is not used in the other audited smart contracts.
- **Market.sol —** this smart contracts implements the marketplace. It keeps track of bid and ask requests, creates deels from matched bid/ask pairs, etc. It imports SNM.sol,Blacklist.sol, OracleUSD.sol, and ProfileRegistry.sol smart contracts.
- **MultiSigWallet.sol —** a multi-signature wallet contract. It is not used in the other audited smart contracts.
- **OracleUSD.sol —** a smart contract that holds the current token/USD exchange rate. This smart contract is used by Market.sol.
- **SNM.sol —** this smart contract is an interface to the ERC20-compatible SNM token. The smart contract is used by Market.sol.
- **SNMMasterchain.sol —** the smart contract implements SNM token. The token is already deployed on the mainnet at address 0x983F6d60db79ea8cA4eB9968C6aFf8cfA04B3c63

- **SimpleGatekeeperWithLimit.sol —** this smart contract accepts any transfers to itself of a token, but only the owner may transfer tokens from this smart contract. This smart contract is an interface between a side-chain and the mainchain. It additionally imposes transfer limits and ability to temporarily freeze transactions.
- **SimpleGatekeeperWithLimitLive.sol —** the same asSimpleGatekeeperWithLimit.sol, but with the following changes:

A. SimpleGatekeeperWithLimitLive.sol supports only **SNMMasterchain** token whereas **SimpleGatekeeperWithLimit.sol** supports any ERC20 token

B. SimpleGatekeeperWithLimitLive.sol does not check for success of token transfers whereas **SimpleGatekeeperWithLimit.sol** do check for success.

- **TestnetFaucet.sol —** this smart contract is intended for debugging
  The smart contracts uses zeppelin-solidity package version 1.9.0 or above. The following smart contracts of zeppelin-solidity package is used:
- **ownership/Ownable.sol —** implements ownership for a smart contract
- **lifecycle/Pausable.sol —** allows pausing and unpausing smart contract operations
- **token/ERC20/StandardToken.sol —** ERC20 token interface

Automated analysis

We used several publicly available automated Solidity analysis tools. Here are the combined results of SmartCheck, Solhint, and Remix.
All the issues found by tools were manually checked (rejected or confirmed).

False positives are constructions that were discovered by the tools as vulnerabilities but do not consist a security threat.

True positives are constructions that were discovered by the tools as vulnerabilities and can actually be exploited by attackers or lead to incorrect contracts operation.


Manual analysis

The contracts were completely manually analyzed, their logic was checked and compared with the one described in the 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 has shown no critical issues.


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

Gas limit and loops

MultisigWallet.sol, lines 329,380:

```
for (i=0; i < transactionCount; i++) ...
```

There is no upper limit on transactionCount, it increments each time a new transaction is registered. Eventually, as the count of transactions increases, gas cost of smart contract calls will raise.

> *The developer is aware of this issue and plans to resolve it in the future version of code.*

Locked tokens

Several contracts (**Market.sol**, **AddressHashMap.sol**, **Blacklist.sol**, **MultisigWallet.sol**, **OracleUSD.sol**, **ProfileRegistry.sol**, **SimpleGatekeeper.sol**) do not support retrieval of tokens sent by mistake. It is recommended to implement an owner-only method allowing transferring tokens (other then SNM for **Market** contract) to a specified address.

> *The developer is aware of this issue and plans to resolve it in the future version of code.*

Missing checks

There are several missing checks:

- **MultisigWallet.sol**, line 372:

```
function getTransactionIds(uint from, uint to, bool pending, bool
executed)
```

`getTransactionIds` function takes `from` variable and `to` variable, but does not check that `to` is greater than `from`


```
function removeDeployer(address _deployer)
```

If `deployers` array is empty, `deployers.length` — 1 causes integer underflow.

- **SNMasterchain.sol**, lines 130, 132, 133. By specifying negative value as `_value` parameter the ICO smart contract may decrease `totalSupply` and balance of `_holder`. However, the ICO is over and this vulnerability cannot be exploited. We recommend implementing all missing checks.

> *The developer is aware of this issue and plans to resolve it in the future version of code.*

ERC20 approve issue

There is ERC20 approve issue in SNM token: changing the approved amount from a nonzero value to another nonzero value allows a double spending with a front-running attack.

We recommend instructing users to follow one of two ways:

- not to use `approve()` function directly and to `useincreaseApproval()/decreaseApproval()` functions instead
- to change the approved amount to 0, wait for the transaction to be mined, and then to change the approved amount to the desired value

> *The developer is aware of this issue and plans to resolve it in the future version of code.*

Low severity issues

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

Pragmas version

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

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 (0.4.24 at the moment).

> *The developer is aware of this issue and plans to resolve it in the future version of code.*

Gas limit and loops

MultisigWallet.sol, line 372:

```
for (i = from; i < to; i++) ...
```

There is no upper limit on the difference between `to` and `from` input variables, this can lead to gas limit issue. In this case, we recommend separating the transaction into several ones.

> *The developer is aware of this issue and plans to resolve it in the future version of code.*

Deprecated constructions

Deprecated syntax is used in several places in the code:

- **MultisigWallet.sol**: invoking events without `emit` prefix is deprecated.
- **MultisigWallet.sol**: defining constructors as functions with the same name as the contract is deprecated. Use `constructor(…) { … }` instead.
- **Market.sol**, lines 197, 643, 688; **SimpleGatekeeper.sol** line 27; **SimpleGatekeeperWithLimit.sol** line 71; **SimpleGatekeeperWithLimitLive.sol** line 71: using `this` keyword as address is deprecated (`address(this)` is preferred).

We highly recommend not to use deprecated constructions.

> *The developer is aware of this issue and plans to resolve it in the future version of code.*

Code quality

- Implicit visibility. There are variables and functions with implicit visibility level in the code: **AddressHashMap.sol** lines 8, 9; **Blacklist.sol** lines 19, 21; **DeployList.sol** lines 8,14; **Market.sol** 113, 123, 125, 127, 130, 133, 139, 141, 143, 145, 147, 149;**OracleUSD.sol** line 9; **ProfileRegistry.sol** lines 41, 47, 49;**SimpleGatekeeperWithLimit.sol** lines 27, 33;**SimpleGatekeeperWithLimitLive.sol** lines 27, 33.
- We recommend specifying visibility levels (`public` , `private` , `external` , `internal`) explicitly and correctly in order to improve code readability.
- Explicit modifier, **MultisigWallet.sol**. The `onlyWallet` modifier gives access only to the `this` contract. It is recommended that function is declared `internal` , then access to them will be restricted automatically.
- Redundant getter **MultisigWallet.sol**, line 337. The `owners` variable is declared `public` , then getter will be generated automatically.
- Redundant check: **SNMMasterChain.sol**, 13. The assertion is always true.
- Redundant getter **ProfileRegistry.sol**, line 71. The `validators` variable is declared `public` , then getter will be generated automatically.
- Missing check **ProfileRegistry.sol**, line 57. The `_level` variable value is notvchecked for validity.
- Misleading comment **Market.sol**, line 196:

```
"this line contains err."
```

However, the next line of code is correct.

We recommend fixing all the issues.

> *The developer is aware of this issue and plans to resolve it in the future version of code.*


This audit was performed by SmartDec, a security team specialized in static code analysis, decompilation and secure development.

Feel free to use SmartCheck, our smart contract security tool for Solidity language, and follow us on Medium. We are also available for smart contract development and auditing work.

SmartDec

More from SmartDec Cybersecurity Blog

Follow

Security tutorials, tools, and ideas

Artyom Orlov · Aug 13, 2018

Soundeon Smart Contracts Security Analysis

Ethereum 8 min read

Igor Sobolev · Aug 6, 2018

YGGDRASH Smart Contracts Security Analysis

Ethereum 8 min read

Alexander Drygin · Jul 2, 2018

Sergio Pavlin · Jul 2, 2018

Infinito Wallet mobile application security analysis

Security 2 min read

Pavel Kondr · Jun 28, 2018

ERC20 approve issue in simple words

Ethereum 5 min read

Read more from SmartDec Cybersecurity Blog