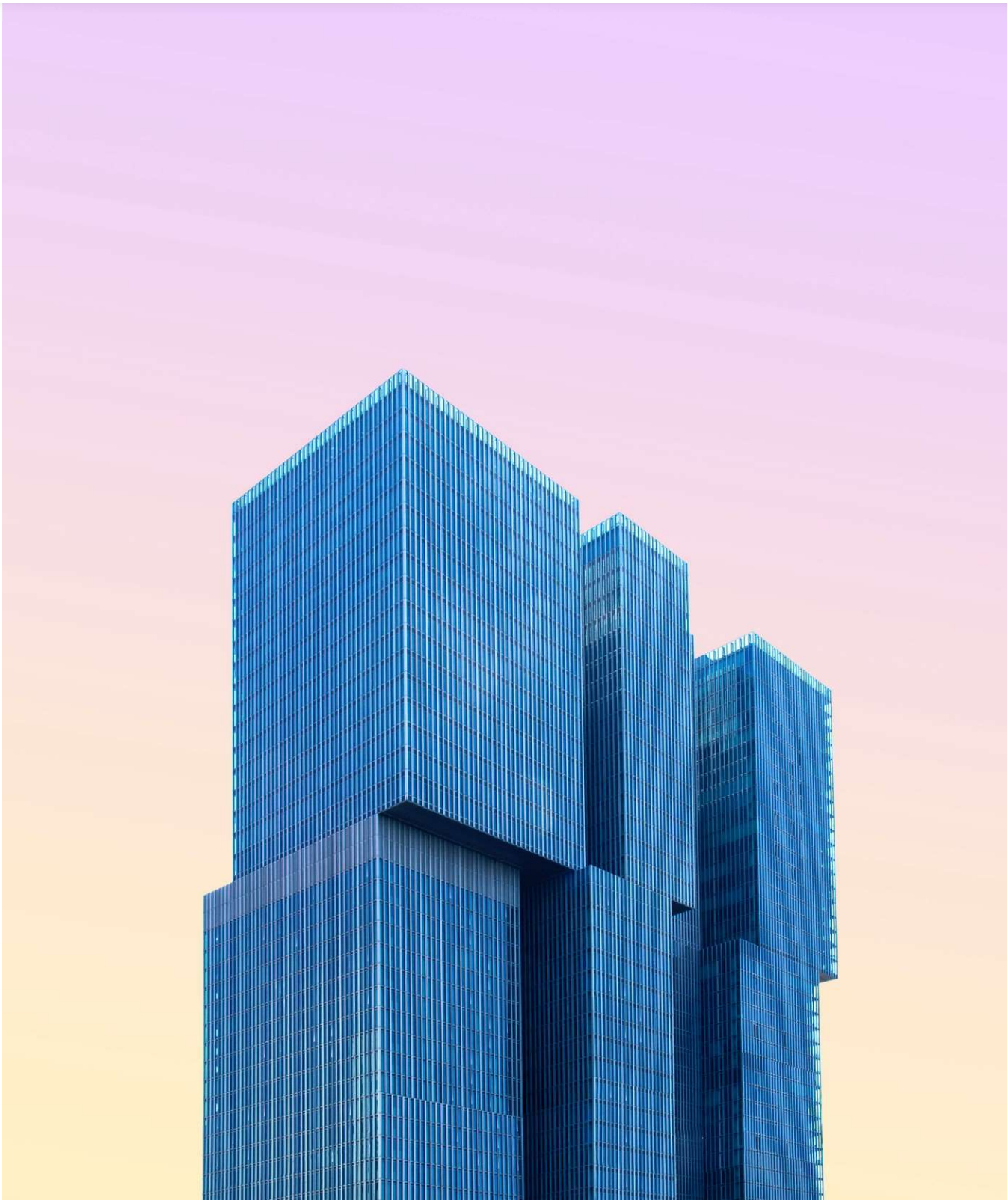
Published in SmartDec Cybersecurity Blog

Alexander Drygin Follow

Aug 30, 2018 · 6 min read · Listen

Boomstarter Smart Contracts Security Review



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

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

Note, that current security review is performed using shortened procedure and only for a certain part of the project and therefore should not be considered a complete audit.

Summary

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

The review did not show any critical issues, although lots of medium and low severity issues were found. Besides, the code logic is complicated and the project appears to be hard to maintain for further development.

General recommendations

We recommend splitting the contract into several separate files to improve the code logic and maintainability of the project, and following best practices for programming design.

Besides, there are some contradictions between the documentation and the code. There is also a list of minor issues that do not affect the code security, but could bother setting up the automated tests.

Procedure

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


We perform our audit according to the following procedure:

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

Report

- we reflect all the gathered information in the report

Checked vulnerabilities

We have scanned Boomstarter 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
- DoS with Block Gas Limit


- 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

Project description

In our analysis we consider Boomstarter technical specification (version on 20.07.2018) and smart contracts code (version on commit ec2d28f).

Files analyzed during the review are listed below:

1. contracts/BoomstarterICO.sol
2. contracts/EthPriceDependentForICO.sol
3. contracts/crowdsale/FundsRegistry.sol

Automated analysis

We used several publicly available automated Solidity analysis tools: SmartCheck, Solhint, and Remix.

The most important issues found by tools were manually checked (rejected or confirmed).

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

The contracts were manually analyzed. Also, the results of the automated analysis were manually verified. All the confirmed important issues are described below.

Critical issues

The review did not reveal any critical issues.

Medium severity issues

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

Split the contract

If some contract implements several chunks of functionality, it's recommended using several contracts instead of one.

Benefits:

- code logic is separated
- deployment costs less
- one can use standard code like Gnosis multisig
- easier to test/maintain/upgrade/audit/…

Multisig feature design

`Muslti sig` should be just a separate contract.
Other contracts should be `Ownable` with ownership transferred to `Multisig` contract.

Token and Sale design

Token contract should not depend on crowdsale or any other contracts.
Sale contract should just receive tokens and sell them.


their tokens. In case if Sale contract doesn't reach the softcap, then investors just withdraw their ether.

Sale and Oraclize design

BoomstarterICO.sol on line 12:

```
contract BoomstarterICO is ArgumentsChecker, ReentrancyGuard,
EthPriceDependentForICO, IICOInfo, IMintableToken {
```

That means that Sale contract inherits from `EthPriceDependent` and `usingOraclize`. As a result of this inheritance:

- constructor for `BoomstarterICO` receives additional parameter `bool _production`;
- constructor for `EthPriceDependent` behaves differently depending on this parameter's value;
- constructor for `BoomstarterICO` is `payable` for no other reason.

That shuffles the code logic. All price logic should be moved to a separate contract.For example, this Price contract can call Sale contract each time it updates price.

Documentation mismatch

- Also, accordingly to the documentation, currency rate USD/ETH should be obtained with a separate contract. The code does not have one.

Address hardcoded

EthPriceDependent.sol at line 27:

```
OAR = OraclizeAddrResolverI(0x6f485C8BF6fc43eA212E93BBF8ce046C7f1cb4
75);
```

This address is Ropsten OAR. Same contract in Mainnet is located on another address.


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

Programming patterns

There are best practices of how to solve standard problems. Consider following them.

Time manipulation for tests

Remove `getTime()` function from code.
Use time-travel features of test frameworks instead. They allow to manipulate `now` value.

Modifier abuse

We don't recommend using modifiers for other purposes other than condition checks:

- modifiers shouldn't change storage of the contract (`ReentrancyGuard` is an exception);
- modifiers should either revert a transaction or execute the function's body.

Breaking these rules is considered bad practice. It leads to code of poor quality, that's hard to read and maintain and is prone to errors.

`multiowned.onlymanyowners` , `BoomstarterICO.timedStateChange` , and `BoomstarterICO.fundsChecker` break these rules.
We recommend refactor those modifiers.

New style constructors

We recommend using new style constructors.

Issues below are grouped by contracts


```
uint public m_ETHPriceLifetime = 60*60*12;
```

should be

```
uint public m_ETHPriceLifetime = 12 hours;
```

FundsRegistry

Incorrect use of assert instead of require.

On lines 55–56:

```
if (State.GATHERING == m_state) { assert(State.REFUNDING ==
_newState || State.SUCCEEDED == _newState); }

else assert(false);
```

should be

```
require(m_state == State.GATHERING); require(_newState ==
State.REFUNDING || _newState == State.SUCCEEDED);
```

On line 73:

```
assert(_investor != m_controller);
```

should be `require` as it checks input parameters.

Visibility

We recommend marking visibility in function explicitly on line 37


EthPriceDependent

Incorrect oraclize integration

On line 63:

```
uint newPrice = parseInt(result).mul(100);
```

Use

```
uint newPrice = parseInt(result, 2);
```

instead. It is more precise and is an intended way to use oraclize.

Price boundary logic

On lines 65–71: if ETH/USD price goes out of specified bounds, then the old price remains. Probably it should reach the bound instead.

BoomstarterICO

Variable is never used

Variable `m_deployer` is never used.

internalBuy

`internalBuy` function is preceded by `timedStateChange` modifier, that guarantees that Sale is in `ACTIVE` state.
It has some test logic in it on lines 202–203:

```
require( !priceExpired() );
 require(m_state == IcoState.ACTIVE || m_state == IcoState.INIT &&
isOwner(client) /* for final test */);
```


Visibility

We recommend marking visibility in function on line 162 explicitly.

> *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

Konstellation Network (DARC)

DarcMatter Events Update Series: Summer 2018

Liti Capital SA

Liti Capital's Wrapped LITI (wLITI) lists on Bitcoin.com Exchange

OAX

Using the openANX Platform

FrogPEPE.finance

The frogPEPE.finance Project

Antier Solutions

Everything you Need to Know About an HD Wallet

Vesta Finance

Whitelist & Treasury Bootstrapping Event Further Details

James

Day of the Dead #01 minted on FTX.us