Published in SmartDec Cybersecurity Blog

Artyom Orlov Follow

May 23, 2018 · 7 min read · Listen

BCShop Smart Contracts Security Analysis

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

Disclaimer

The audit does not give any warranties on the security of the code. One audit can not be considered enough. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts

Summary

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

The audit showed several issues of different severity level. We highly recommend addressing them.

General recommendations

The contracts code is of good code quality. However, the project have minor compilation issues (see Compilation output). Besides, we recommend fixing Pragmas version. Also, if the developer decides to improve the code, we recommend following Solidity Style Guide (see Visibility and Constant functions).

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

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).
3. Whether the code meets best practices in efficient use of gas, code readability, etc.

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 publicly available automated Solidity


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
- we check ERC20 compliance

Report

- we reflect all the gathered information in the report

Checked vulnerabilities

We have scanned BCshop Product 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
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send


- 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

Project overview

Project description

In our analysis we consider BCshop Product smart contracts code (Git repository, commit 3023172) and documentation (contracts.pdf, sha1sum 9c141f619affbe7b41038a39f19c903a5915704e).

The latest version of the code

We have performed the check of the fixed vulnerabilities in the latest version of code (

Git repository, version on commit 96678fc5ca807103623cc0e83e59bb1920f77e87).

Project architecture

For the audit, we have been provided with the following set of files:

- product/DiscountPolicy
- product/FeePolicy
- product/ProductMaker
- product/ProductPayment
- product/ProductStorage
- shop/EtherFund
- shop/ProxyFund

The project also contains tests, deploy scripts, and several files that are beyond the scope of the audit.

The total volume of the Solidity code that has been audited is 1024 lines of code.


Contract inherits Owned contact

- implementation of ownable contract
- default owner is the contract creator (deployer address)

Contract inherits SafeMath contract

Contract construction sets two ETH receivers with their shares

Contract functionality:

function `migrate` :

- owner can transfer all ETH to any new address

function `copyStateFor` :

- owner can set parameters of any address from other EtherFund (or similar contract)
- side effect — internal variables `lastBalance` and `sumDeposits` are also copied

function `withdraw` , function `withdrawTo` :

- allows sender to withdraw own share of ETH to own address or any other

function `changeReceiver` :

- allows owner to change share from one receiver to another

function `changeShares` :

- allows owner to change share distribution of two receivers
- allows to split one receiver share into one additional address

function `changeShares3` :


ProxyFund contract is contract that allows manager to withdraw ETH from other base fund.

Contract inherits Owned contract

- implementation of ownable contract
- default owner is the contract creator (deployer address)

Contract inherits Manageable contract

- allows owner to add managers

Contract functionality:

function `setBaseFund` :

- allow owner to change baseFund

function `withdraw` :

- allows manager to withdraw ether from baseFund to his address

function `withdrawTo` :

- allows manager to withdraw ether from baseFund to any address

ProductMaker, **ProductStorage**, **ProductPayment** contracts are the system that allows to create “products”, set different parameters of sale, and allows customers to buy product and pay for it with ETH or BCS tokens. Customers are able to have discount via DiscountPolicy. Also, system charges fee for transaction processing.

DiscountPolicy contract keeps information about discounts available for users and process cashbacks.

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.

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

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

Inconsistent state

1.Use of `copyStateFor` function (lines 52–59) of EtherFund contract can lead to inconsistent state of Fund with total share not equal 1000 promille. This can happen because information of each user is copied independently and possibly can be changed on another fund, or not copied at all, or copied from two different funds. This may lead to lock of money on fund, or receivers may not be able to withdraw their ETH.

We recommend implementing copying all receivers with their shares in one transaction.

> *The issue has been fixed by the developer and is not present in the latest version of the code.*

2. Usage of `migrate` function (lines 47–49) of EtherFund contract can lead to. inconsistent state of the current contract and a new one. So receivers of old contract will not be able to withdraw their ETH. Also, their state possibly will not be copied into a new contract.

We recommend transfering ETH and copying address in one transaction or implementing other logic of migration with partial share migration of ETH.

> *Comments from developer team: dependent contracts will be paused during migrating.*

Medium severity issues

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

Unchecked math

Solidity is prone to an integer over- and underflow. Overflow leads to unexpected effects and can lead to loss of funds if exploited by malicious account. The values in the following cases are not checked:

```
require(share1 + share2 == 1000);
```

- EtherFund.sol, line 113:

```
require(share1 + share2 == sharePermille[receiver1] +
sharePermille[receiver2]);
```

- EtherFund.sol, line 132:

```
require(share1 + share2 + share3 == sharePermille[receiver1] +
sharePermille[receiver2] + sharePermille[receiver3]);
```

In case the parameters are very large, this may lead to overflow.

We recommend checking the values transferred as deploy parameters so that they do not lead to overflow.

> *The issue has been fixed and is not present in the latest version of the 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. Example:

```
pragma solidity ^0.4.18; // bad: compiles w 0.4.18 and above
```


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 — 0.4.21 at the moment (0.4.18 is used in the contracts).

Unchecked math

Solidity is prone to an integer over- and underflow. Overflow leads to unexpected effects and can lead to loss of funds if exploited by malicious account. The values in the following cases are not checked:

- FeePolicy.sol, line 93:

```
fee = fee + escrowFee;
```

- FeePolicy.sol, line 180:

```
require(_defaultFeePermille + _escrowFeePermille +
_fiatPriceFeePermille <= 1000);
```

In case the parameters are very large, this may lead to overflow.

We recommend checking the values transferred as deploy parameters so that they do not lead to overflow.

> *The issue has been fixed and is not present in the latest version of the code.*

Visibility

There are variables and functions with implicit visibility level in the code:

- EtherFund.sol, lines 31, 42
- FeePolicy.sol, lines 18, 34
- ProductPayment.sol, lines 49, 61


We recommend specifying visibility levels (`public` , `private` , `external` , `internal`) explicitly and correctly in order to improve code readability.

Constant functions

`constant` state modifier is used for functions:

- DiscountPolicy.sol, lines 54, 72, 151
- EtherFund.sol, lines 62, 67, 163
- FeePolicy.sol, lines 82,119, 125, 137
- ProductPayment.sol, lines 89, 105
- ProductStorage.sol, lines 147, 152, 168, 186, 204, 213, 222, 231, 240, 251, 260, 270, 280, 285
- ProxyFund.sol, line 32

We recommend using `view` instead of `constant` , which will be deprecated for functions.

If a function is not supposed to modify the state or read from state, consider declaring it as `pure` .

> *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 · May 2, 2018

MithrilOre Smart Contracts Security Analysis

Ethereum 7 min read

Artyom Orlov · Apr 28, 2018

Joyso Smart Contracts Security Analysis

Ethereum 9 min read

Artyom Orlov · Apr 28, 2018

MinerOne Smart Contracts Security Analysis

Ethereum 7 min read

Artyom Orlov · Apr 19, 2018

Lendingblock Smart Contracts Security Audit

Ethereum 9 min read

Artyom Orlov · Apr 17, 2018

Rate3 Smart Contracts Security Audit

Ethereum 7 min read

Read more from SmartDec Cybersecurity Blog

javier lecca

BASIC GIT COMMANDS

Louis.Z

Encrypting sections of Web.Config through runCommand

TechGig

What is the difference between programming and coding?

<hello,w>

#08. MJ's Thesis—Assumption Mapping

demola malomo in Dev Genius

Build a REST API with Golang and MongoDB—Gorilla/Mux Version

Arvin Fernandez

CSS Media Queries On A High Level

【☆ゆう☆】

Elf Binary Mangling Part 3—Weaponization

Vijaya Kumar Chinthala in Analytics Vidhya

File Handling strip(), rstrip() and lstrip()