QuillAudits

Audit Report

October, 2020

OPEN

Contents

Disclaimer

This audit report presents the findings of a security review of the smart contracts under the scope of the audit. The audit does not give any *warranties on the security of the code*. Using specially designed tools for debugging and using automated tests to verify minor changes and fixes.

We always recommend proceeding to several independent audits and a public bug bounty program to ensure the security of the smart contracts.

Quillhash audit is not a security warranty, investment advice, or an endorsement of the cashbox contract.

Introduction

Audit request

Cashbox is a capital-efficient market where users can deposit any ERC20 token and receive a Pool Token which is an ownership token for your share of the pool. Users can use the cashbox to deposit a STOCK token and receive the ERC20 at the determined rate. When burning the Pool Token for the contents of the pool token, you will always receive Stock Token as the first priority. If there is not enough Stock Token, the remaining balance will be given in the ERC20 token. There is also an Upper Cap which limits the minting of pool tokens when that upper cap is reached. More information can be found on - *https://github.com/opendao-protocols/cashbox.*

In scope

This document is a security audit report performed by QuillAudits Team, where Cashbox smart contract - https://github.com/opendao-protocols/cashbox/tree/d364f2be31164959915bdb54285a23d7e8d421c2 has been reviewed.
The address of the contract on rinkeby network:
0x6271902bf08ce913dbd3c0f11efc896a415b5506

Commit hash: d364f2be31164959915bdb54285a23d7e8d421c2

cashbox.sol

Methodology

Our review methodology

The Quillhash team has performed rigorous testing of the project starting with analysing the code design patterns in which we reviewed the smart contract architecture to ensure it is structured and safe use of third party smart contracts and libraries.

Our team then performed a formal line by line inspection of the Smart Contract to find any potential issue like race conditions, transaction-ordering dependence, timestamp dependence, and denial of service attacks.

In the Unit testing Phase, we coded/conducted custom unit tests written for each function in the contract to verify that each function works as expected.

In Automated Testing, We tested the Smart Contract with our in-house developed tools to identify vulnerabilities and security flaws.

The code was tested in collaboration of our multiple team members and this included -

- Testing the functionality of the Smart Contract to determine proper logic has been followed throughout the whole process.
- Analysing the complexity of the code in-depth and detailed, manual review of the code, line-by-line.
- Deploying the code on testnet using multiple clients to run live tests.
- Analysing failure preparations to check how the Smart Contract performs in case of any bugs and vulnerabilities.
- Checking whether all the libraries used in the code are on the latest version.
- Analysing the security of the on-chain data.

Classification of detected issues

High
High vulnerability can be exploited at any time and cause a loss of customers funds or a complete breach of contract operability. (Example: Parity Multisig hack, a user has exploited a vulnerability and violated the operability of the whole system of smart-contracts (Parity Multisigs). This could be performed regardless of external conditions at any time.)

Medium
Medium vulnerability can be exploited in some specific circumstances and cause a loss of customers funds or a breach of operability of smart-contract (or smart-contract system). (Example: ERC20 bug, a user can exploit a bug (or "undocumented opportunity") of the transfer function and occasionally burn his tokens. A user can not violate someone else's funds or cause a complete breach of the whole contract operability. However, this leads to millions of dollars losses for Ethereum ecosystem and token developers.)

Low
Low vulnerability can not cause a loss of customers funds or a breach of contracts operability. However, it can cause any kind of problems or inconveniences. (Example: Permanent owners of multisig contracts, owners are permanent, thus if it will be necessary to remove a misbehaving "owner" from the owners' list then it will require to redeploy the whole contract and transfer funds to a new one.)

Owner Privileges
The ability of an owner to manipulate contracts, may be risky for investors.

Note
Other code flaws, not security-related issues.

The severity is calculated according to the OWASP risk rating model based on Impact and Likelihood:

Overall Risk Severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	Note	Low	Medium
		LOW	MEDIUM	HIGH
	Likelihood			

Summary of Findings

In total, 5 issues were reported, and all are closed

- 0 high severity issues.
- 1 medium severity issues.
- 2 low severity issues.
- 2 notes.
- 0 owner privileges (optional).

Audit Findings

High Severity Issues

No high severity issues were identified in the smart contract

Medium severity issue

1. **Payable Fallback**.
   StockLiquiditator contract contains a fallback function that allows ether deposit to the contract, but no other function inside the contract is implemented to withdraw ether from it.

   Code snippet
   https://github.com/opendao-protocols/cashbox/blob/dd363f587dfa798897aec258b67a60f460cd8fb2/cashbox.sol#L478

   Line 478

```
function () external payable {  //fallback function

}
```

Status: Closed

Low severity issue

1. Required check for zero address

It is possible to remain out of contract control by accidentally calling changeOwner function with zero address.

Code snippet

https://github.com/opendao-protocols/cashbox/blob/dd363f587dfa798897aec258b67a60f460cd8fb2/cashbox.sol#L517

Line 517

```
function changeOwner(address payable newOwner) external onlyOwner {
        owner=newOwner;
        emit OwnerChanged(newOwner);

}
```

Recommendation

Use condition like:

```
require(newOwner != address(0), "Owner is the zero address");
```

Status: Closed

2. Required check for zero _stockToCashRate

There is a need to check the input value _stockToCashRate of the constructor for zero-value. In case of the argument is not specified, the redeemStockToken function will not work correctly, and the user can not trade Stock token for the ERC20.

Code snippet

https://github.com/opendao-protocols/cashbox/blob/dd363f587dfa798897aec258b67a60f460cd8fb2/cashbox.sol#L499

https://github.com/opendao-protocols/cashbox/blob/dd363f587dfa798897aec258b67a60f460cd8fb2/cashbox.sol#L609-L621

```
constructor (address cashAddress,address stockTokenAddress,uint256 _stockToCashRate,uint256 cashCap,
string memory name,string memory symbol,string memory _url)public ERC20Detailed( name, symbol, 18) {
        owner = msg.sender;
        require(stockTokenAddress != address(0), "stockToken is the zero address");
        require(cashAddress != address(0), "cash is the zero address");
        cash = ERC20Detailed(cashAddress);
        stockToken = ERC20Detailed(stockTokenAddress);
        cashDecimals = cash.decimals();
        stockTokenMultiplier = (10**uint256(stockToken.decimals()));
        stockToCashRate = (10**(cashDecimals)).mul(_stockToCashRate);
        updatePoolRate();
        updateCashValuationCap(cashCap);
        updateURL(_url);
    }
```

```
function redeemStockToken(uint256 stockTokenAmount) external{
        address sender= msg.sender;
        _preValidateData(sender,stockTokenAmount);
        stockToken.transferFrom(sender,address(this),stockTokenAmount);

        // calculate Cash amount to be return
        uint256 outputCashAmount=(stockTokenAmount.mul(stockToCashRate)).div(stockTokenMultiplier);
        uint256 balanceBeforeTransfer = cash.balanceOf(sender);
        cash.transfer(sender,outputCashAmount);
        uint256 balanceAfterTransfer = cash.balanceOf(sender);
        require(balanceAfterTransfer == balanceBeforeTransfer.add(outputCashAmount),
        "Sent & Received Amount mismatched");
        emit StockTokensRedeemed(sender,stockTokenAmount,outputCashAmount);
}
```

Status: Closed

Notes

1. Consider removing the destruction mechanism

Consider removing the self-destruct functionality unless it is absolutely required. If there is a valid use-case, it is recommended to implement a multisig scheme so that multiple parties must approve the self-destruct action.

Code snippet

https://github.com/opendao-protocols/cashbox/blob/dd363f587dfa798897aec258b67a60f460cd8fb2/cashbox.sol#L623

Line 623

```
function kill() external onlyOwner {
        selfdestruct(owner);
    }
```

Status: Closed

2. Known vulnerabilities of ERC-20 token

Lack of transaction handling mechanism issue. This is a very common issue, and it already caused millions of dollars losses for lots of token users! More information here.

Conclusion

In the end, We should mention the high code quality of the project, using safe OpenZeppelin contracts. The audited smart contract is safe to deploy. No high severity issues were found during the audit.

QuillAudits

India, Singapore, United Kingdom, Canada

audits.quillhash.com

hello@quillhash.com