Security Audit Report for ChainCheque

Date: Oct 13, 2021

Version: 1.0

Contact: contact@blocksecteam.com

Contents

Report Manifest

Item	Description
Client	Matrixport Technologies Ltd.
Target	ChainCheque

Version History

Version	Date	Description
1.0	Oct 13, 2021	First Release

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high impact security incidents. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

The target contract is used to write a check by the drawer to the payee. The check can append encrypted memo with the payee's public key. It can also append a hash tag on the check or a hash value of a shared secret between the drawer and the payee. The payee can only accept (withdraw) the check if he/she provides the correct secret. The check also has a deadline, after when the check cannot be accepted or rejected by the payee. However, the drawer can revoke the check after the deadline.

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The files that are audited in this report include the following ones.

Repo Name	Github URL
write-a-check	https://github.com/write-a-check/write-a-check

The commit hash before the audit is 255bc49653bad7f885dd1db994ee64394bbf4d59.

1.2 Disclaimer

This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts. Besides, this report does not constitute personal investment advice or a personal recommendation.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data Flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Access control
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [1] and Common Weakness Enumeration [2].

[1] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[2] https://cwe.mitre.org/

Accordingly, the severity measured in this report are classified into four categories: **High**, **Medium**, **Low** and **Undetermined**.

Chapter 2 Findings

In total, we did not find issues or vulnerabilities in the smart contract. However, we have **two recommendations**, as follows:

- High Risk: 0
- Medium Risk: 0
- Low Risk: 0
- Recommendations: 2

ID	Severity	Description	Category
1	-	*No authentication of revokeCheque*	Recommendation
2	-	*Possible collision of the top byte of a hash value and #*	Recommendation

The details are provided in the following sections.

2.1 Additional Recommendation

2.1.1 No authentication of `revokeCheque`

Status Acknowledged.

Description The function `revokeCheque` can be used to revoke a check. However, this function does not have any authentication so that a user can revoke another user's check.

```
152   function revokeCheque(uint id) public {
153     Cheque memory cheque = getCheque(id);
154     require(cheque.deadline != 0, "cheque-not-exists");
155     require(cheque.deadline < block.timestamp, "still-before-deadline");
156     deleteCheque(id);
157     safeTransfer(cheque.coinType, cheque.drawer, uint(cheque.amount));
158     emit RevokeCheque(address(uint160(id>>96)), id, cheque.drawer);
159   }
```

Impact A user can revoke another user's check.

Suggestion Add an authentication mechanism.

Feedback from the project Since the revoke of a check will always return back the tokens to the sender (drawer), not the one who invokes this function, it will not cause the loss of tokens to the users. This is a design that there is no authentication to the `revokeCheque`.

2.1.2 Possible collision of the top byte of a hash value and

Status Acknowledged.

Description In the contract, the check issuer can put a passphrase or hashtag into the `passphraseOrHashtag`. To distinguish between them, it checks the highest byte of this value to determine whether it's a passphrase (hash value of a secret) or a hashtag. If this value is # then it's a hashtag (line 191), else it's a passphrase.

However, there exist a case that the highest byte of the hash value of a secret is also #. This makes it become a hashtag, which is not the intention of the check issuer. As a result, the receiver does not need the secret value to receive the check.

```
183  function receiveCheque(uint id, bool accept, bytes memory passphrase) internal {
184    require(uint(uint160(bytes20(msg.sender))) == (id>>96), "not-payee");
185    Cheque memory cheque = getCheque(id);
186    require(cheque.deadline != 0, "cheque-not-exists");
187    require(block.timestamp <= cheque.deadline, "after-deadline");
188    address receiver = msg.sender;
189    if(accept) {
190      //check passphrase if it is not a hashtag and not zero, note ascii of '#' is 35
191      if(cheque.passphraseOrHashtag != 0 && (cheque.passphraseOrHashtag>>248) != 35) {
192        bytes32 hash = keccak256(passphrase);
193        require((uint(hash)<<8) == (cheque.passphraseOrHashtag<<8), "wrong-passphrase");
194      }
195      emit AcceptCheque(msg.sender, id, cheque.drawer);
196    } else {
197      receiver = cheque.drawer;
198      emit RefuseCheque(msg.sender, id, cheque.drawer);
199    }
200    deleteCheque(id);
201    safeTransfer(cheque.coinType, receiver, uint(cheque.amount));
202  }
```

Impact The receiver does not need the secret value to receive the check.

Suggestion NA

Feedback from the project This is not an issue in practice, since when using a passphrase, the client of this smart contract must change the highest byte of `phassphraseOrHashtag` to be a value that's different from #. In another word, the user needs to ensure the case previously described never happens.