Security Audit Report for Lock Smart Contracts

Date: Dec 06, 2022

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2022.12.06 23:28:08 +08'00'

Contents

Report Manifest

Item	Description
Client	NFEX
Target	Lock Smart Contracts

Version History

Version	Date	Description
1.0	Dec 06, 2022	First Release

Chapter 1 Introduction

1.1 About Target Contracts

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

The audit target is the *Lock smart contract* [1] of NFEX, while those test-purpose files (e.g., under the `test` directory) are out of the audit scope. According to the official document [2], this project is to lock users' assets (including ETH and ERC20 tokens) into the contract. When withdrawing the assets from the contract, a multi-sig verification is performed by the `Maintainer` contract, which also maintains the validators and the multisig threshold. Specifically, the assets can only be unlocked if the number of passed signatures is bigger than the threshold. After a successful verification, the corresponding assets will be transferred to the address specified by the withdrawal.

In terms of contract structure, it is mainly composed of two smart contracts, the asset contract and the maintainer contract. The asset contract is responsible for managing assets, that is, recharge and withdrawal. The maintainer contract mainly handles the review of withdrawals and maintains validators and multi-signature thresholds. Ensure that all withdrawal requests must be verified by the signature verifier. Specifically, the contract of this project mainly consists of two functions, i.e., locking assets and withdrawing assets. Please refer to the official document[2] for more details.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
NFEX Lock	`Version 1`	`a61e631449812cb944c3c31c302a0622507fa12c`
	`Version 2`	`2dcf8042cf28e6360ef428231dfa0b643c12ac5d`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering

[1] https://github.com/NFEX-Coder/NFEX/

[2] https://github.com/NFEX-Coder/NFEX/blob/orange-version2/README.md

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.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

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
- Permission management
- 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 [3] and Common Weakness Enumeration [4]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.

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

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

- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we find **two** potential issues. Besides, we also have **two** recommendations.

- Medium Risk: 1
- Low Risk: 1
- Recommendation: 2

ID	Severity	Description	Category	Status
1	Medium	Problematic check due to the division	Software Security	Fixed
2	Low	The validator can be changed by the owner of the Maintainer contract	Software Security	Acknowledged
3	-	Check the repeated validators in the initialize function	Recommendation	Acknowledged
4	-	Ensure the passed tokens are not elastic tokens	Recommendation	Acknowledged

The details are provided in the following sections.

2.1 Software Security

2.1.1 Problematic check due to the division

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The check at line 78 in the `initialize` function of `Maintainer.sol` is somewhat problematic, i.e., 50% cannot be guaranteed due to the division. For example, if the validators.length is 3, and multisigThreshold is 1, then the check will be bypassed because division rounds towards zero in Solidity.

```
71    function initialize(address[] memory validators, uint8 multisigThreshold)
72    public
73    initializer
74    {
75        ...
76
77        require(
78            multisigThreshold >= SafeMath.div(validators.length, 2),
79            "multisigThreshold is less than 50%"
80        );
81
82        ...
83    }
```

Listing 2.1: Maintainer.sol

Impact The check of the number of passed signatures of the multisig wallet can be bypassed.

Suggestion Change >= to >.

2.1.2 The validator can be changed by the owner of the Maintainer contract

Severity Low

Status Acknowledged

Introduced by `Version 1`

Description According to the contract, the `changeValidators` function can be invoked by the contract owner. However, this violates the security design of the contact. Let's say the maintainer changes the validators to his own two or three addresses, and he/she can unlock the assets using his own signatures.

Impact The maintainer owner can bypass the check of signatures to unlock assets.

Suggestion Enforce a multi-sig style access control, which is similar to unlock the assets.

Feedback from the project The owner of the Maintainer contract will be a multisig wallet, like `Gnosis Safe`.

2.2 Additional Recommendation

2.2.1 Check the repeated validators in the `initialize` function

Status Acknowledged

Introduced by `Version 1`

Description Check whether the validators have repeated addresses in the `initialize` function of the Maintainer contract.

Impact N/A

Suggestion Add the check.

2.2.2 Ensure the passed tokens are not elastic tokens

Status Acknowledged

Introduced by `Version 1`

Description Elastic tokens may lead to imprecise amounts being recorded, e.g., in the `LockToken` function.

```
function LockToken(address token, uint256 amount) public nonReentrant whenNotPaused {
    require(amount > 0, "LOCK: Amount should be greater than 0");
    IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
    emit Locked(token, msg.sender, amount);
}
```

Listing 2.2: Lock.sol

Impact The recorded amount may not be accurate.

Suggestion Check the balance again after the transfer.