Security Audit Report for ETH2 NFT Validator Smart Contracts

Date: Nov 19, 2022

Version: 1.0

Contact: contact@blocksec.com

BlockSec
Audit Team

Digitally signed by BlockSec Audit Team
Date: 2022.11.19 15:32:30 +08'00'

Contents

Report Manifest

Item	Description
Client	Stakefish
Target	ETH2 NFT Validator Smart Contracts

Version History

Version	Date	Description
1.0	Nov 19, 2022	First Release

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec 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 successfully protected digital assets that are worth more than 5 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

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 *eth2-nft-validator-contract* project [1] of Stakefish, while those test-purpose files (e.g., under the `contracts/test` directory) are out of the audit scope.

This project aims to provide an Ethereum staking flow that issues NFTs as staking receipts and proof of validator ownership. Specifically, a user can first deposit (stake) 32 ETH and get a minted NFT token as the receipt. After that, the user will become a validator after the operator confirms this deposit and updates the status accordingly. The user can also request to exit after becoming a validator. Note that the deposited ETH can be withdrawn before the operator confirms the deposit. However, like the staking withdrawals of Eth2 [2], the confirmed deposit cannot be withdrawn at the current stage as it has been transferred to the deposit contract.

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
eth2-nft-validator-contract	`Version 1`	`d87cb3104782a3de40434525d5acfc1c4faf9693` (tag v.0.4.3)
	`Version 2`	`50ac8fe5933fc7312458989f72a50fcd68b9c3f1`

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 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

[1] https://github.com/stakefish/eth2-nft-validator-contract

[2] https://ethereum.org/en/upgrades/merge/

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.
- **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.

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

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

Chapter 2 Findings

In total, we find **one** potential issue. We also have **two** recommendations and **one** note.

- Low Risk: 1
- Recommendation: 2
- Note: 1

ID	Severity	Description	Category	Status
1	Low	Potential access control problem	Software Security	Fixed
2	-	Remove unnecessary check	Recommendation	Fixed
3	-	Check the validity for important variables	Recommendation	Fixed
4	-	The minted NFT tokens are not burnt for users' withdrawal	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Potential access control problem

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `StakefishValidatorBase` contract, there is a modifier named `isNFTMultiCallOrNFTOwner`, which is used to verify the caller. This modifier requires the invocation initiating from either the NFT owner, or the NFT Multicall interface. The latter is further checked by invoking the `isNFTMulticall` function. However, this check (and the modifier) can be bypassed.

```
29    modifier isNFTMultiCallOrNFTOwner() {
30        require(getNFTOwner() == msg.sender || isNFTMulticall(), "not nft owner or multicall");
31        _;
32    }
33
34    function isNFTMulticall() internal view returns (bool) {
35        return StorageSlot.getAddressSlot(_NFT_MANAGER_SLOT).value == msg.sender && getNFTOwner()
              == tx.origin;
36    }
```

Listing 2.1: StakefishValidatorBase.sol

Specifically, suppose there is an attacker who has deployed a malicious contract. The attacker takes a phishing method to make the owner of a validator invoke some function of the contract, which may lead to the following call sequence:

$$Validator\_owner -> malicious\ contract -> NFTManager -> Validator$$

From the validator's perspective, `msg.sender == NFTManager`, and `tx.origin` is the NFT owner (i.e., validator owner). As a result, the check in the `isNFTMulticall()` function can be bypassed.

In the current implementation, there exist two places where the `isNFTMultiCallOrNFTOwner` modifier is used:

- The first one is the `upgradeLatestByNFTOwner` function of the `StakefishValidatorWallet` contract. Though the implementation cannot be controlled by the attacker (is fetched from factory contract), an attacker can make the upgrade of a validator on behalf of the victim (by using the previously mentioned invocation chain).
- The second place is the `migrate` function of the `StakefishValidatorWallet` contract. This should be fine since the new manager address is NOT controlled by the attacker. But again, the attacker can migrate the NFT manager for other validators.

Since the implementation code can be upgraded, be cautious when using this modifier and `tx.origin` in the future versions.

Impact May lead to unexpected results as this modifier can be bypassed.

Suggestion Revise the code.

2.2 Additional Recommendation

2.2.1 Remove unnecessary check

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `initialize` function of the `StakefishValidatorWallet` contract, the check at line 20 is unnecessary, because there exists another check at line 24. Besides, it also means that the first parameter `factory_` is unnecessary as well as the value always comes from `msg.sender`.

```
19    function initialize(address factory_, address nftManager_) external payable {
20        require(factory_ != address(0), "factory may not be null");
21        require(nftManager_ != address(0), "manager may not be null");
22        require(StorageSlot.getAddressSlot(_NFT_MANAGER_SLOT).value == address(0), "initialized
              already");
23        require(StorageSlot.getAddressSlot(_FACTORY_SLOT).value == address(0), "initialized already
              ");
24        require(factory_ == msg.sender, "only factory allowed to initialize.");
25
26        StorageSlot.getAddressSlot(_FACTORY_SLOT).value = factory_;
27        StorageSlot.getAddressSlot(_NFT_MANAGER_SLOT).value = nftManager_;
28        StorageSlot.getAddressSlot(_IMPLEMENTATION_SLOT).value = IStakefishValidatorFactory(
              factory_).latestVersion();
29    }
```

Listing 2.2: StakefishValidatorWallet.sol

Impact N/A

Suggestion Remove the redundant check.

2.2.2 Check the validity for important variables

Status Fixed in `Version 2`

Introduced by Version 1

Description The variable named `protocolFee` (line 32 of the `StakefishValidatorFactory` contract) should not be greater than 10000 (line 98 and line 114 in the `withdraw` function of the `StakefishValidatorV1` contract). It is recommended to add the corresponding check in the `setFee` function.

```
13  contract StakefishValidatorFactory is IStakefishValidatorFactory, Ownable {
14
15      /// @dev prevent front-running - attacker creating with future tokenId and
16      /// break the functionality with the caller (NFTManager)
17      mapping(address => bool) private _approvedDeployers;
18
19      /// @dev instance of wallet contract for EIP-1167 minimal proxy cloning
20      address private _walletClonable;
21
22      /// @dev list of verified contracts
23      address[] public override implementations;
24
25      /// @dev stakefish operator
26      address public override operatorAddress;
27
28      /// @dev migration address
29      address public override migrationAddress;
30
31      /// @dev protocol fee
32      uint256 public override protocolFee = 1500;
```

Listing 2.3: StakefishValidatorFactory.sol

```
93  function withdraw() external override nonReentrant isNFTOwner {
94      uint256 availableBalance = address(this).balance;
95
96      if (withdrawnBalance >= 32 ether) {
97          // all balance need to pay stakefish commission
98          uint256 commission = (availableBalance * getProtocolFee()) / 10000;
99          uint256 userReward = availableBalance - commission;
100         withdrawnBalance += availableBalance;
101         Address.sendValue(payable(StorageSlot.getAddressSlot(_FACTORY_SLOT).value), commission)
                ;
102         Address.sendValue(payable(getNFTOwner()), userReward);
103         emit StakefishValidatorWithdrawn(pubkey, userReward);
104         emit StakefishValidatorCommissionTransferred(pubkey, commission);
105     } else {
106         if (withdrawnBalance + availableBalance <= 32 ether) {
107             // all balance can be withdrawn commission free
108             withdrawnBalance += availableBalance;
109             Address.sendValue(payable(getNFTOwner()), availableBalance);
110             emit StakefishValidatorWithdrawn(pubkey, availableBalance);
111         } else {
112             // a part of the balance can be withdrawn commission free
113             uint256 commissionApplyBalance = availableBalance + withdrawnBalance - 32 ether;
114             uint256 commission = (commissionApplyBalance * getProtocolFee()) / 10000;
115             uint256 userReward = availableBalance - commission;
116             withdrawnBalance += availableBalance;
```

```
117             Address.sendValue(payable(StorageSlot.getAddressSlot(_FACTORY_SLOT).value),
                    commission);
118             Address.sendValue(payable(getNFTOwner()), userReward);
119             emit StakefishValidatorWithdrawn(pubkey, userReward);
120             emit StakefishValidatorCommissionTransferred(pubkey, commission);
121         }
122     }
123 }
```

Listing 2.4: StakefishValidatorV1.sol

```
64  function setFee(uint256 _feePercent) external override onlyOwner() {
65      protocolFee = _feePercent;
66  }
```

Listing 2.5: StakefishValidatorFactory.sol

Impact N/A

Suggestion Add the corresponding check.

2.3 Note

2.3.1 The minted NFT tokens are not burnt for users' withdrawal

Description The minted NFT token is not burnt when a user withdraws the ETH she deposited. For example, the user first deposits 32 ETH and then gets a NFT token. After that, she invokes the `withdraw` function directly before the operator invokes the `makeETH2Deposit` function. As a result, the user still owns the NFT token without staking. Note that in the `claim` function (implemented in `test/TestV2NFTManager.sol`), the `verifyAndBurn` function will be invoked.

Feedback from the Project Yes, it's intentional because we do not want to allow accidental burn and brick the wallet contract (at least right now when ETH is not allowing withdraw yet). It's possible in the future we will implement a BurnContract that implements like V2NFTManager (without the minting, and burn it there).