SMART CONTRACT AUDIT REPORT

for

STAFI PROTOCOL

Prepared By: Shuxiao Wang

Hangzhou, China

January 15, 2021

Document Properties

Client	Stafi Protocol
Title	Smart Contract Audit Report
Target	Eth2 Staking
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Xudong Shao
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 15, 2021	Xuxian Jiang	Final Release
1.0-rc	January 12, 2021	Xuxian Jiang	Release Candidate
0.3	January 10, 2021	Xuxian Jiang	Add More Findings #2
0.2	January 5, 2021	Xuxian Jiang	Add More Findings #1
0.1	December 31, 2020	Xuxian Jiang	Initial Draft

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction 4
- 1.1 About StaFi's Eth2 Staking . . . 4
- 1.2 About PeckShield . . . 5
- 1.3 Methodology . . . 5
- 1.4 Disclaimer . . . 6

2 Findings 9
- 2.1 Summary . . . 9
- 2.2 Key Findings . . . 10

3 Detailed Results 11
- 3.1 Constant/Immutable States If Fixed Or Set at Constructor() . . . 11
- 3.2 Possible Open Window For Storage Initialization And Manipulation . . . 12
- 3.3 Business Logic in StafiUpgrade::addStafiUpgradeContract() . . . 15
- 3.4 Lack Of Sanity Checks For System Parameters . . . 17
- 3.5 Improved Precision By Multiplication And Division Reordering . . . 18
- 3.6 Trust Issue of Admin Keys Behind SuperUser . . . 20

4 Conclusion 22

References 23

1 Introduction

Given the opportunity to review the design document and related source code of the **Eth2 Staking** in the StaFi protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About StaFi's Eth2 Staking

The consensus transition of Ethereum requires ETH staking and the liquidity loss of staked ETHs may deter user participation, hence calling for an immediate solution. The audited protocol aims to address the liquidity issue of staked assets by proposing a wrapper-based rETH solution. Moreover, it provides a marketplace that allows users to participate in ETH staking with any amount at his own discretion. In the meantime, it supports validators that actually runs and maintains the validator nodes by dynamically providing the required assets for staking. The yielding rewards are distributed back to staking users in proportion to the staked amount from users.

The basic information of StaFi's Eth2 Staking is as follows:

Table 1.1: Basic Information of StaFi's Eth2 Staking

Item	Description
Name	Stafi Protocol
Website	https://stafi.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 15, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit:

- https://github.com/stafiprotocol/eth2-staking (e42e858)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/stafiprotocol/eth2-staking (70e983c)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `StaFi`'s Eth2 Staking implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	🟦
Medium	1	🟦
Low	3	🟦🟦🟦🟦
Informational	1	🟦
Total	6	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 1 medium-severity vulnerability, 3 low-severity vulnerabilities, and and 1 informational recommendation.

Table 2.1: Key Audit Findings of Eth2 Staking Protocol

ID	Severity	Title	Category	Status
PVE-001	Informational	Constant/Immutable States If Fixed Or Set at Constructor()	Coding Practices	Fixed
PVE-002	High	Possible Open Window For Storage Initialization And Manipulation	Business Logic	Confirmed
PVE-003	Low	Business Logic in StafiUpgrade::addStafiUpgradeContract()	Business Logic	Fixed
PVE-004	Low	Lack Of Sanity Checks For System Parameters	Coding Practices	Fixed
PVE-005	Low	Improved Precision By Multiplication And Division Reordering	Numeric Errors	Fixed
PVE-006	Medium	Trust Issue of Admin Keys Behind SuperUser	Security Features	Confirmed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Constant/Immutable States If Fixed Or Set at Constructor()

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `StafiBase`, `AddressQueueStorage`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

In the following, we show the key state variables defined in `StafiBase`. If there is no need to dynamically update these key state variables, e.g., `version` and `stafiStorage`, they can be declared as `immutable` for gas efficiency.

```
abstract contract StafiBase {

    // Version of the contract
    uint8 public version;

    // The main storage contract where primary persistant storage is maintained
    IStafiStorage stafiStorage = IStafiStorage(0);

    ...
}
```

Listing 3.1: StafiBase.sol

In addition, we notice a state variable in `AddressQueueStorage`, i.e., `capacity`. This is a constant and we can simply define it as a `constant` to avoid gas cost for the access.

```
// Address queue storage helper
contract AddressQueueStorage is StafiBase, IAddressQueueStorage {

    // Libs
    using SafeMath for uint256;

    // Settings
    uint256 public capacity = 2 ** 255; // max uint256 / 2

    // Construct
    constructor(address _stafiStorageAddress) StafiBase(_stafiStorageAddress) public {
        version = 1;
    }
    ...
}
```

Listing 3.2: AddressQueueStorage.sol

Recommendation Revisit the state variable definition and make good use of `immutable`/`constant` states.

Status This issue has been addressed in the following commit: `70e983c`.

3.2 Possible Open Window For Storage Initialization And Manipulation

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `StafiBase`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `StaFi`'s Eth2 Staking implements an approach that isolated all essential states in a dedicated storage contract, i.e., `StafiStorage`. This storage contract saves states of 7 basic types, i.e., `uint256`, `string`, `address`, `bytes`, `bool`, `int256`, and `bytes32`. Each state has an associated key and the content is stored in a key-value pair, i.e., `uIntStorage`, `stringStorage`, `addressStorage`, `bytesStorage`, `boolStorage`, `intStorage`, and `bytes32Storage`. To illustrate, we show below the related key-value pairs in `StafiStorage`.

```
    // Storage types
    mapping(bytes32 => uint256)     private uIntStorage;
    mapping(bytes32 => string)      private stringStorage;
    mapping(bytes32 => address)     private addressStorage;
    mapping(bytes32 => bytes)       private bytesStorage;
    mapping(bytes32 => bool)        private boolStorage;
    mapping(bytes32 => int256)      private intStorage;
    mapping(bytes32 => bytes32)     private bytes32Storage;
```

Listing 3.3: The 7 Key-Value Pairs in StafiStorage

In the meantime, the `StaFi`'s Eth2 Staking implementation provides related `getters` and `setters` in a rather generic, standard approach, such as `getAddress()/setAddress()/deleteAddress()/`, `getUint()/setUint()/deleteUint()`, `getString()/setString()/deleteString()`, `getBytes()/setBytes()/deleteBytes()`, `getBool()/setBool()/deleteBool()`, `getInt()/setInt()/deleteInt()`, and `getBytes32()/setBytes32()/deleteBytes32()`.

Since the storage contract keeps a number of sensitive states (e.g., `owner`, `admin` and `ethDeposit`), these `setters` are properly gated via the `onlyLatestNetworkContract()` modifier with the following implementation. It comes to our attention that the protection is turned-off when the state `contract.storage.initialised` is not updated as `true`.

```
    /// @dev Only allow access from the latest version of a contract in the network
        after deployment
    modifier onlyLatestNetworkContract() {
        // The owner and other contracts are only allowed to set the storage upon
            deployment to register the initial contracts/settings, afterwards their
            direct access is disabled
        if (boolStorage[keccak256(abi.encodePacked("contract.storage.initialised"))] ==
            true) {
            // Make sure the access is permitted to only contracts in our Dapp
            require(boolStorage[keccak256(abi.encodePacked("contract.exists", msg.sender
                ))], "Invalid or outdated network contract");
        }
        _;
    }
```

Listing 3.4: StafiStorage::onlyLatestNetworkContract()

```
    /// @param _key The key for the record
    function setAddress(bytes32 _key, address _value) onlyLatestNetworkContract override
         external {
        addressStorage[_key] = _value;
    }

    /// @param _key The key for the record
    function setUint(bytes32 _key, uint256 _value) onlyLatestNetworkContract override
        external {
        uIntStorage[_key] = _value;
    }

    /// @param _key The key for the record
    function setString(bytes32 _key, string calldata _value) onlyLatestNetworkContract
        override external {
        stringStorage[_key] = _value;
    }

    /// @param _key The key for the record
    function setBytes(bytes32 _key, bytes calldata _value) onlyLatestNetworkContract
        override external {
        bytesStorage[_key] = _value;
    }

    /// @param _key The key for the record
    function setBool(bytes32 _key, bool _value) onlyLatestNetworkContract override
        external {
        boolStorage[_key] = _value;
    }

    /// @param _key The key for the record
    function setInt(bytes32 _key, int256 _value) onlyLatestNetworkContract override
        external {
        intStorage[_key] = _value;
    }

    /// @param _key The key for the record
    function setBytes32(bytes32 _key, bytes32 _value) onlyLatestNetworkContract override
         external {
        bytes32Storage[_key] = _value;
    }
```

Listing 3.5: Various Setter Routines in StafiStorage

In other words, any one may freely update these states until `contract.storage.initialised` is configured to be `true`. However, a malicious actor may take advantage of the time window to whitelist a crafted contract (via `contract.exists`) so that any state update can be performed via this crafted contract. As a result, the sanity checks applied in these `setters` are completely bypassed, hence allowing these protocol-wide sensitive states for manipulation! Note it is also possible for the malicious actor to turn the protection on by writing `true` to `contract.storage.initialised`, which could undermine the stability and/or integrity of the entire protocol as well.

Recommendation Apply a rigorous and sound access control policy that blocks this open time window for state manipulation.

Status This issue has been confirmed. The team decides to exercise extra care when initializing these system paramters. The protocol will not go live to public until all configurations are carefully verified.

3.3 Business Logic in StafiUpgrade::addStafiUpgradeContract()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `StafiUpgrade`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In Section 3.2, we have discussed the dedicated storage contract `StafiStorage`. This storage contract also greatly facilitates the updates of current contracts. Specifically, for an existing contract, the storage always keeps record of the latest version and the access can be guaranteed to use the latest version. This is a rather smooth update process!

To elaborate, we show below the `addStafiUpgradeContract()` routine in `StafiUpgrade`. As the name indicates, this routine implements the update logic by basically updating three states in the storage contract, i.e., `contract.exists`, `contract.name` and `contract.address` (lines 80-82).

```
76      // Add stafi upgrade contract
77      function addStafiUpgradeContract(address _contractAddress) private {
78          string memory name = "stafiUpgrade";
79          bytes32 nameHash = keccak256(abi.encodePacked(name));
80          setBool(keccak256(abi.encodePacked("contract.exists", _contractAddress)), true);
81          setString(keccak256(abi.encodePacked("contract.name", _contractAddress)), name);
82          setAddress(keccak256(abi.encodePacked("contract.address", name)),
                _contractAddress);
83          // Emit contract added event
84          emit ContractAdded(nameHash, _contractAddress, now);
85      }
```

Listing 3.6: StafiUpgrade::addStafiUpgradeContract()

However, it comes to our attention that the old states associated with the updated contracts have not been removed. As a result, it is possible that the old contract can still be properly authenticated through `onlyLatestNetworkContract()` modifier (Section 3.2). This is certainly not desirable if the updates are intended to fix a bug in the old contracts and the bug may be exploited to perform unexpected state updates.

Recommendation Remove old states from the updated contracts to ensure only the latest version is properly maintained.

Status This issue has been addressed in the following commit: `70e983c`.

3.4 Lack Of Sanity Checks For System Parameters

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `StafiNetworkSettings`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The Eth2 Staking protocol is no exception. Specifically, if we examine the `UbiswapFactory` contract, it has defined a number of system-wide risk parameters, e.g., `settings.network.node.fee`, `settings.network.platform.fee`, and `settings.network.node.refund.ratio`. In the following, we show corresponding routines that allow for their changes.

```
    // The node commission rate as a fraction of 1 ether
    function getNodeFee() override public view returns (uint256) {
        return getUintS("settings.network.node.fee");
    }
    function setNodeFee(uint256 _value) public onlySuperUser {
        setUintS("settings.network.node.fee", _value);
    }

    // The platform commission rate as a fraction of 1 ether
    function getPlatformFee() override public view returns (uint256) {
        return getUintS("settings.network.platform.fee");
    }
    function setPlatformFee(uint256 _value) public onlySuperUser {
        setUintS("settings.network.platform.fee", _value);
    }

    // The node refund commission rate as a fraction of 1 ether
    function getNodeRefundRatio() override public view returns (uint256) {
        return getUintS("settings.network.node.refund.ratio");
    }
    function setNodeRefundRatio(uint256 _value) public onlySuperUser {
        setUintS("settings.network.node.refund.ratio", _value);
    }
```

Listing 3.7: Various Getters/Setters in StafiNetworkSettings

This parameter defines an important aspect of the protocol operation and needs to exercise extra care when configuring or updating it. Our analysis shows the update logic on it can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `settings.`

`network.platform.fee` may charge unreasonable tax on the staking operation, hence incurring cost to staking users.

Recommendation Validate any changes regarding the system-wide parameter to ensure the changes fall in an appropriate range. If necessary, also consider emitting relevant events for its changes.

Status This issue has been addressed in the following commit: `70e983c`.

3.5 Improved Precision By Multiplication And Division Reordering

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact:Low
- Target: `StafiNetworkBalances`, `StafiNetworkWithdrawal`
- Category: Numeric Errors [8]
- CWE subcategory: CWE-190 [2]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (`mul`) and division (`div`) are involved.

Specifically, if we examine the following `StafiNetworkWithdrawal::withdrawStakingPool()` routine, we notice the final conditional check (line 81) is performed via `calcBase.mul(submissionCount).div( stafiNodeManager.getTrustedNodeCount())>= stafiNetworkSettings.getNodeConsensusThreshold()`.

```
54      function withdrawStakingPool(address _stakingPoolAddress, uint256
            _stakingStartBalance, uint256 _stakingEndBalance) override external
55      onlyLatestContract("stafiNetworkWithdrawal", address(this)) onlyTrustedNode(msg.
            sender) onlyRegisteredStakingPool(_stakingPoolAddress) {
56          // Load contracts
57          IStafiNetworkSettings stafiNetworkSettings = IStafiNetworkSettings(
                getContractAddress("stafiNetworkSettings"));
58          // Check settings
59          require(stafiNetworkSettings.getProcessWithdrawalsEnabled(), "Processing
                withdrawals is currently disabled");
60          // Check balance
```

```
        require(getBalance() >= _stakingEndBalance, "Insufficient withdrawal pool
            balance");
        // Check withdrawal status
        IStafiStakingPoolManager stafiStakingPoolManager = IStafiStakingPoolManager(
            getContractAddress("stafiStakingPoolManager"));
        require(!stafiStakingPoolManager.getStakingPoolWithdrawalProcessed(
            _stakingPoolAddress), "Withdrawal has already been processed for stakingpool
            ");
        // Check stakingpool status
        IStafiStakingPool stakingPool = IStafiStakingPool(_stakingPoolAddress);
        require(stakingPool.getStatus() == StakingPoolStatus.Staking, "Staking pool can
            only be set as withdrawable while staking");
        // Get submission keys
        bytes32 nodeSubmissionKey = keccak256(abi.encodePacked("stakingpool.withdrawable
            .submitted.node", msg.sender, _stakingPoolAddress, _stakingStartBalance,
            _stakingEndBalance));
        bytes32 submissionCountKey = keccak256(abi.encodePacked("stakingpool.
            withdrawable.submitted.count", _stakingPoolAddress, _stakingStartBalance,
            _stakingEndBalance));
        // Check & update node submission status
        require(!getBool(nodeSubmissionKey), "Duplicate submission from node");
        setBool(nodeSubmissionKey, true);
        setBool(keccak256(abi.encodePacked("stakingpool.withdrawable.submitted.node",
            msg.sender, _stakingPoolAddress)), true);
        // Increment submission count
        uint256 submissionCount = getUint(submissionCountKey).add(1);
        setUint(submissionCountKey, submissionCount);
        // Check submission count & set stakingpool withdrawable
        uint256 calcBase = 1 ether;
        IStafiNodeManager stafiNodeManager = IStafiNodeManager(getContractAddress("
            stafiNodeManager"));
        if (calcBase.mul(submissionCount).div(stafiNodeManager.getTrustedNodeCount()) >=
             stafiNetworkSettings.getNodeConsensusThreshold()) {
            processWithdrawal(_stakingPoolAddress, _stakingStartBalance,
                _stakingEndBalance);
        }
    }
```

Listing 3.8: StafiNetworkWithdrawal::withdrawStakingPool()

A better approach is to perform the multiplication operation before division to avoid introducing unnecessary precision loss. In particular, the above computation $A.mul(B).div(C) >= D$ can be adjusted as $A.mul(B) >= C.mul(D)$. In other words, the final conditional check can be revised as the following: `calcBase.mul(submissionCount)>= stafiNodeManager.getTrustedNodeCount().mul (stafiNetworkSettings.getNodeConsensusThreshold())`. Certainly, the reordering should not introduce any unwanted overflow in the multiplication operations.

Note the `StafiNetworkBalances` contract shares the same issue.

Recommendation Avoid unnecessary precision loss due to the lack of floating support in `Solidity`. If there is no concern in introducing the overflow risk, it is always preferred to perform multiply-before-divide in the computation.

Status This issue has been addressed in the following commit: `70e983c`.

3.6 Trust Issue of Admin Keys Behind SuperUser

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

In Eth2 Staking, there is a privileged contract, i.e., `owner`, that plays a critical role in in governing and regulating the system-wide operations (e.g., enabling `deposits/withdrawals`, adding `trusted nodes` , updating protocol-wide `constracts`, and customizing various parameters).

In the following, we show the `onlySuperUser()` modifier implementation. This modifier validates the `msg.sender` is either `owner` or `admin`. This is necessary to prevent sensitive storage-based states from being manipulated.

```
/**
* @dev Modifier to scope access to admins
*/
modifier onlySuperUser() {
    require(roleHas("owner", msg.sender)  roleHas("admin", msg.sender), "Account is
        not a super user");
    _;
}
```

Listing 3.9: StafiBase :: onlySuperUser()

```
/**
* @dev Check if an address has this role
*/
function roleHas(string memory _role, address _address) internal view returns (bool)
    {
    return getBool(keccak256(abi.encodePacked("access.role", _role, _address)));
}
```

Listing 3.10: StafiBase :: roleHas()

We emphasize that the current privilege assignment may serve the design goal for the time being. However, it is worrisome that `owner` is not governed by a `DAO`-like structure. The discussion with the team has confirmed that the governance will be managed by a multisig account.

We point out that a compromised `owner` account would allow the attacker to mess up current states, disrupt the protocol execution, or even completely undermines the asset safety.

Recommendation Promptly design a trustless, decentralized scheme to reduce the concern on the centralized `owner` privilege. As a mitigation, instead of having a single EOA account as the `owner`, an alternative is to make use of a multi-sig wallet. To further eliminate the administration key concern, it may be required to transfer the role to a community-governed DAO. In the meantime, a timelock-based mechanism might also be applicable for mitigation.

Status This issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the design and implementation of `StaFi`'s Eth2 Staking implementation, which is a timely solution to address the liquidity issue of staked assets during the consensus transition of Ethereum. The system presents a clean and consistent design that makes it a distinctive and valuable addition to current DeFi ecosystem. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[9] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[11] PeckShield. PeckShield Inc. https://www.peckshield.com.