SMART CONTRACT AUDIT REPORT

for

Eth2 Staking

Prepared By: Xiaomi Huang

PeckShield

September 15, 2022

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	September 15, 2022	Xuxian Jiang	Final Release
1.0-rc	September 15, 2022	Xuxian Jiang	Release Candidate

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

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 the audited protocol 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	September 15, 2022

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 (0e30095f)

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 (3d764fa1)

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	0	
Medium	2	■ ■
Low	1	■
Informational	1	■
Total	4	

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 2 medium-severity vulnerabilities, 1 low-severity vulnerability, and and 1 informational recommendation.

Table 2.1: Key Audit Findings of Eth2 Staking Protocol

ID	Severity	Title	Category	Status
PVE-001	Medium	Caller Validation in StafiLightNode::stake()	Business Logic	Resolved
PVE-002	Informational	Improved Gas Efficiency With Repeated Calls Avoidance	Coding Practices	Resolved
PVE-003	Low	Possible Reverts From Math Operations in AddressQueueStorage	Numeric Errors	Confirmed
PVE-004	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.

PeckShield

3 Detailed Results

3.1 Caller Validation in StafiLightNode::stake()

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

Description

The `StaFi`'s Eth2 Staking supports two types of nodes: light node and super node. Both types support user deposits and stakes, though they differ in the accepted `Ether` amount. While examining the staking logic in light node, we notice the current implementation needs to be improved to ensure only intended callers are accepted.

To elaborate, we show below the related staking logic of the light node. It takes three sets of arguments, i.e., `_validatorPubkeys`, `_validatorSignatures`, and `_depositDataRoots`. Accordingly, it delegates the actual staking logic to an internal `_stake()` routine.

```
    function stake(bytes[] calldata _validatorPubkeys, bytes[] calldata _validatorSignatures, bytes32[] calldata _depositDataRoots) override external onlyLatestContract("stafiLightNode", address(this)) {
        require(_validatorPubkeys.length == _validatorSignatures.length && _validatorPubkeys.length == _depositDataRoots.length);
        // Load contracts
        IStafiUserDeposit stafiUserDeposit = IStafiUserDeposit(getContractAddress("stafiUserDeposit"));
        stafiUserDeposit.withdrawExcessBalanceForLightNode(_validatorPubkeys.length.mul(uint256(32 ether).sub(getCurrentNodeDepositAmount())));

        for (uint256 i = 0; i < _validatorPubkeys.length; i++) {
            _stake(_validatorPubkeys[i], _validatorSignatures[i], _depositDataRoots[i]);
        }
    }
```

Listing 3.1: `StafiLightNode::stake()`

The internal routine validates the given `_validatorPubkey` by checking its status as `PUBKEY_STATUS_MATCH` and then updating it to be `PUBKEY_STATUS_STAKING`. However, it comes to our attention that it does not validate the caller is the owner of the given `_validatorPubkey`! As a result, the current implementation allows any one to invoke the staking logic to bind to other validator's public keys.

```
    function _stake(bytes calldata _validatorPubkey, bytes calldata _validatorSignature,
         bytes32 _depositDataRoot) private {
        setAndCheckNodePubkeyInStake(_validatorPubkey);
        // Send staking deposit to casper
        EthDeposit().deposit{value: uint256(32 ether).sub(getCurrentNodeDepositAmount())
            }(_validatorPubkey, StafiNetworkSettings().getWithdrawalCredentials(),
            _validatorSignature, _depositDataRoot);

        emit Staked(msg.sender, _validatorPubkey);
    }
```

Listing 3.2: `StafiLightNode::_stake()`

```
    // Set and check a node's validator pubkey
    function setAndCheckNodePubkeyInStake(bytes calldata _pubkey) private {
        // check status
        require(getLightNodePubkeyStatus(_pubkey) == PUBKEY_STATUS_MATCH, "pubkey status
             unmatch");
        // set pubkey status
        _setLightNodePubkeyStatus(_pubkey, PUBKEY_STATUS_STAKING);
    }
```

Listing 3.3: `StafiLightNode::setAndCheckNodePubkeyInStake()`

Recommendation Validate the caller to the intended validator owner.

Status This issue has been resolved in the following commit: 472301e.

3.2 Improved Gas Efficiency With Repeated Calls Avoidance

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `StafiNetworkWithdrawal`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

The `StaFi` protocol has a `StafiNetworkWithdrawal` contract that is designed to handle the withdrawals from network validators. While reviewing the current withdrawal logic, we notice the current implementation can be improved for gas efficiency.

In the following, we show the full implementation of the related `processWithdrawal()` function. It implements a rather straightforward logic in querying the related contracts, retrieving the reward amounts for the node and the user, then updating the internal accounting, and finally sending out the rewards. We notice the current implementation makes repeated calls to the same contracts `stafiNetworkSettings/stakingPool` for the same results of `platform/node fees` and `node/user deposit balances`. For gas efficiency, we can avoid repeated calls by caching and reusing the first call result.

```
    function processWithdrawal(address _stakingPoolAddress, uint256 _stakingStartBalance
        , uint256 _stakingEndBalance) private {
        // Load contracts
        IStafiNetworkSettings stafiNetworkSettings = IStafiNetworkSettings(
            getContractAddress("stafiNetworkSettings"));
        IStafiUserDeposit stafiUserDeposit = IStafiUserDeposit(getContractAddress("
            stafiUserDeposit"));
        IStafiStakingPoolManager stafiStakingPoolManager = IStafiStakingPoolManager(
            getContractAddress("stafiStakingPoolManager"));
        IStafiEther stafiEther = IStafiEther(getContractAddress("stafiEther"));
        IStafiStakingPool stakingPool = IStafiStakingPool(_stakingPoolAddress);

        uint256 nodeAmount = getStakingPoolNodeRewardAmount(
            stafiNetworkSettings.getPlatformFee(),
            stafiNetworkSettings.getNodeFee(),
            stakingPool.getNodeDepositBalance(),
            stakingPool.getUserDepositBalance(),
            _stakingStartBalance,
            _stakingEndBalance
        );
        uint256 userAmount = getStakingPoolUserRewardAmount(
            stafiNetworkSettings.getPlatformFee(),
            stafiNetworkSettings.getNodeFee(),
            stakingPool.getNodeDepositBalance(),
            stakingPool.getUserDepositBalance(),
            _stakingStartBalance,
            _stakingEndBalance
        );

    ...
}
```

Listing 3.4: `StafiNetworkWithdrawal::processWithdrawal()`

Recommendation Improve the gas efficiency by avoiding the repeated calls in the above `processWithdrawal()` routine.

Status This issue has been confirmed.

3.3 Possible Reverts From Math Operations in AddressQueueStorage

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `AddressQueueStorage`
- 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 examining its uses in current contracts, we notice it may exhibit certain corner cases to bring unintended execution reverts.

To elaborate, we show below the `getLength()/getItem()` routines in the `AddressQueueStorage` contract. As the names indicate, these two routines are used to return the number of items in the queue and query a specific item respectively. In particular, within the `getLength()` routine, the `if`-statement (line 27) has an implicit assumption of not reverting the execution in the calculation of `end = end .add(capacity)`. Similarly, within the `getItem()` routine, it implicitly assumes the given `_index` will not cause overflow in the computation of `getUint(keccak256(abi.encodePacked(_key, ".start"))).add (_index)` (line 33). A possible improvement is to remove the above implicit assumption by avoiding the introduction of unintended reverts.

```
23      // The number of items in a queue
24      function getLength(bytes32 _key) override public view returns (uint256) {
25          uint256 start = getUint(keccak256(abi.encodePacked(_key, ".start")));
26          uint256 end = getUint(keccak256(abi.encodePacked(_key, ".end")));
27          if (end < start) { end = end.add(capacity); }
28          return end.sub(start);
29      }

31      // The item in a queue by index
32      function getItem(bytes32 _key, uint256 _index) override external view returns (
            address) {
33          uint256 index = getUint(keccak256(abi.encodePacked(_key, ".start"))).add(_index)
                ;
34          if (index >= capacity) { index = index.sub(capacity); }
35          return getAddress(keccak256(abi.encodePacked(_key, ".item", index)));
36      }
```

Listing 3.5: `StafiUpgrade::getLength()/getItem()`

Recommendation Improve the above two routines to remove the implicit assumption.

Status This issue has been confirmed.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- 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 admin user, i.e., `SuperUser`, 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 `contracts`, 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 3.6: `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 3.7: `StafiBase::roleHas()`

Notice that the privilege assignment is necessary and consistent with the protocol design. In the meantime, the extra power to the owner may also be a counter-party risk to the protocol users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Moreover, it should be noted that if current contracts are planned to deploy behind a proxy, there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Making the above privileges explicit among protocol users.

Status This issue has been confirmed and the team clarifies the plan to gradually open up to community governance in the future.

PeckShield

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.

PeckShield

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.

PeckShield