SMART CONTRACT AUDIT REPORT

for

StafiWithdraw Contact

Prepared By: Xiaomi Huang

PeckShield

March 23, 2023

Document Properties

Client	Stafi Protocol
Title	Smart Contract Audit Report
Target	StafiWithdraw
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 23, 2023	Xuxian Jiang	Final Release
1.0-rc	March 17, 2023	Luck Hu	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 `StafiWithdraw` support 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 StafiWithdraw

The `StaFi` protocol provides unique liquid staking derivative (`LSD`) solution on `Ethereum`. With the arrival of `Shanghai` upgrade, the `StafiWithdraw` support enables users to redeem `ETH` with `rETH` in order to ensure seamless transactions. In order to work effectively, it is essential for relevant information to be properly counted through an off-chain service. This information is then used to notify the withdraw contract, especially when a validator exit occurs on the consensus layer (beacon chain). By doing so, the `StaFi` protocol ensures that all transactions are conducted in a secure and efficient manner. The basic information of the audited contract is as follows:

Table 1.1: Basic Information of `StaFi`'s StafiWithdraw

Item	Description
Name	Stafi Protocol
Website	https://stafi.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 23, 2023

In the following, we show the Git repository of reviewed file and the commit hash value used in this audit: Note the audit only covers the `contracts/withdraw/StafiWithdraw.sol` file.

- https://github.com/stafiprotocol/eth2-staking/tree/v3 (4ae6d2d)

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/tree/v3 (TBD)

1.2 About PeckShield

PeckShield Inc. [9] 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 [8]:

- 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.

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

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) [7], 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 1.4: 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 StafiWithdraw 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	0	
Total	3	

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 and 1 low-severity vulnerability.

Table 2.1: Key Audit Findings of StafiWithdraw Protocol

ID	Severity	Title	Category	Status
PVE-001	Medium	Possible Proposal Id Conflicts in StafiWithdraw	Numeric Errors	Fixed
PVE-002	Medium	Trust Issue of Admin Keys Behind SuperUser	Security Features	Confirmed
PVE-003	Low	Suggested Adherence of Checks-Effects-Interactions Pattern	Time and State	Fixed

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 Possible Proposal Id Conflicts in StafiWithdraw

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `StafiWithdraw`
- Category: Numeric Errors [6]
- CWE subcategory: CWE-190 [1]

Description

The `StafiWithdraw` support allows the trusted nodes to vote for proposals to perform operations to the contract, e.g., distribute withdrawals and notify validators exit. Each proposal has an id that is generated from the hash256 of the proposal content. While examining the generation of the proposal ids, we notice the ids may conflict between the proposals to distribute withdrawals and notify validators exit.

To elaborate, we show below the code snippets of the `distributeWithdrawals()/notifyValidatorExit()` routines. In the `distributeWithdrawals()` routine, the proposal id is the hash256 of the five input `uint256` parameters (line 187). In the `notifyValidatorExit()` routine, the proposal id is the hash256 of the input parameters, including two `uint256` and one `uint256` array (line 264).

However, it comes to our attention that, there is no domain separators added to the generation of the proposal ids in both routines. Specially, if the `uint256` array parameter, i.e., `_validatorIndexList`, has three elements and the values of the five input `uint256` parameters for both routines are the same in order, the generated ids from both routines will be the same. As a result, the same proposal id may represent two different proposals.

Based on this, it is suggested to add proper domain separators respectively to the generation of proposal ids in the `distributeWithdrawals()/notifyValidatorExit()` routines.

```
176     function distributeWithdrawals(
177         uint256 _dealedHeight,
178         uint256 _userAmount,
```

```
        uint256 _nodeAmount,
        uint256 _platformAmount,
        uint256 _maxClaimableWithdrawIndex
    ) external override onlyLatestContract("stafiWithdraw", address(this)) onlyTrustedNode(msg.sender) {
        require(_dealedHeight > latestDistributeHeight, "height already dealed");
        require(_maxClaimableWithdrawIndex < nextWithdrawIndex, "withdraw index over");
        require(_userAmount.add(_nodeAmount).add(_platformAmount) <= address(this).balance, "balance not enough");

        bytes32 proposalId = keccak256(
            abi.encodePacked(_dealedHeight, _userAmount, _nodeAmount, _platformAmount, _maxClaimableWithdrawIndex)
        );
        bool needExe = _voteProposal(proposalId);

        // Finalize if Threshold has been reached
        if (needExe) {...}
    }
```

Listing 3.1: `StafiWithdraw::distributeWithdrawals()`

```
    function notifyValidatorExit(
        uint256 _withdrawCycle,
        uint256 _ejectedStartCycle,
        uint256[] calldata _validatorIndexList
    ) external override onlyLatestContract("stafiWithdraw", address(this)) onlyTrustedNode(msg.sender) {
        require(_ejectedStartCycle < _withdrawCycle && _withdrawCycle < currentWithdrawCycle(), "cycle not match");

        bytes32 proposalId = keccak256(abi.encodePacked(_withdrawCycle, _ejectedStartCycle, _validatorIndexList));
        bool needExe = _voteProposal(proposalId);

        // Finalize if Threshold has been reached
        if (needExe) {...}
    }
```

Listing 3.2: `StafiWithdraw::notifyValidatorExit()`

Recommendation Revisit the above mentioned routines to add proper domain separators respectively for the generation of proposal ids.

Status This issue has been fixed in the following commit: 1696f1b.

3.2 Trust Issue of Admin Keys Behind SuperUser

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `StafiWithdraw`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In `StafiWithdraw`, there is a privileged admin user, i.e., `SuperUser`, that plays a critical role in governing and regulating the system-wide operations (e.g., set the overall withdraw limit per cycle, set the withdraw limit for each user per cycle). 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 3.3: `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 3.4: `StafiBase::roleHas()`

Notice that the privilege assignment is necessary and consistent with the protocol design. In the meantime, the extra power to the admin user 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.

3.3 Suggested Adherence of Checks-Effects-Interactions Pattern

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `StafiWithdraw`
- Category: Time and State [5]
- CWE subcategory: CWE-663 [3]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [11] exploit, and the recent `Uniswap/Lendf.Me` hack [10].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. In the following, we show the code snippet of the `unstake()` function, which is provided to externally call the `msg.sender` to transfer `ETH`. However, if the `msg.sender` is a contract, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 132) starts before effecting the update on internal states (lines 138 – 139), hence violating the principle. In this particular case, if the external contract has certain hidden logic in its `receive()/fallback()` functions that may be capable of launching `re-entrancy` via the same entry function.

Based on this, it is strongly recommended to adhere to the `checks-effects-interactions` best practice or making use of `nonReentrant` to block possible `re-entrancy`.

```
121 function unstake(uint256 _rEthAmount) external override onlyLatestContract("
     stafiWithdraw", address(this)) {
122     uint256 ethAmount = _processWithdraw(_rEthAmount);
123     IStafiUserDeposit stafiUserDeposit = IStafiUserDeposit(getContractAddress("
         stafiUserDeposit"));
124     uint256 stakePoolBalance = stafiUserDeposit.getBalance();
125 
126     uint256 totalMissingAmount = totalMissingAmountForWithdraw.add(ethAmount);
127     if (stakePoolBalance > 0) {...}
128     totalMissingAmountForWithdraw = totalMissingAmount;
129 
130     bool unstakeInstantly = totalMissingAmountForWithdraw == 0;
131     if (unstakeInstantly) {
132         (bool result, ) = msg.sender.call{value: ethAmount}("");
133         require(result, "Failed to unstake ETH");
```

```
    } else {...}

    emit Unstake(msg.sender, _rEthAmount, ethAmount, nextWithdrawIndex, unstakeInstantly
        );

    withdrawalAtIndex[nextWithdrawIndex] = Withdrawal({_address: msg.sender, _amount:
        ethAmount});
    nextWithdrawIndex = nextWithdrawIndex.add(1);
}
```

Listing 3.5: `StafiWithdraw::unstake()`

Recommendation Adhere to the `checks-effects-interactions` best practice or apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier.

Status This issue has been fixed in the following commit: 1696f1b.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `StafiWithdraw` implementation, which enables users to redeem `ETH` with `rETH` with the the arrival of `Shanghai` upgrade. In order to work effectively, it is essential for relevant information to be properly counted through an off-chain service. This information is then used to notify the withdraw contract. By doing so, the `StaFi` protocol ensures that all transactions are conducted in a secure and efficient manner. 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-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

[3] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

[5] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

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

[10] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[11] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.