SECURITY AUDIT REPORT

for

STAFI PROTOCOL

Prepared By: Shuxiao Wang

Hangzhou, China

January 15, 2021

Document Properties

Client	Stafi Protocol
Title	Security Audit Report
Target	Stafi-Node
Version	1.0
Author	Xudong Shao
Auditors	Xudong Shao, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 15, 2021	Xudong Shao	Final Release
1.0-rc	January 12, 2021	Xudong Shao	Release Candidate
0.3	January 10, 2021	Xudong Shao	Add More Findings #2
0.2	January 5, 2021	Xudong Shao	Add More Findings #1
0.1	December 31, 2020	Xudong Shao	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 Stafi-Node 4

1.2 About PeckShield 5

1.3 Methodology 5

1.4 Disclaimer 7

2 Findings **9**

2.1 Summary 9

2.2 Key Findings 10

3 Detailed Results **11**

3.1 Improved Overflow Checks In liquidity_bond() 11

4 Conclusion **13**

References **14**

1 Introduction

Given the opportunity to review the design document and related source code of the **Stafi-Node** 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 the issue mentioned in the report. This document outlines our audit results.

1.1 About StaFi's Stafi-Node

Stafi-Node is the node implementation of `StaFi` protocol which is the first decentralized protocol unlocking liquidity of staked assets. `StaFi` aims to solve the contradiction between mainnet security and token liquidity in PoS consensus. The token holders are staking through staking contracts built in `StaFi` protocol, and then get alternative tokens`rToken` (such as `rXTZ`, `rAtom`, `rDot`, etc.). The `rTokens` are tradable and holders can get staking rewards from original chain at the same time.

The basic information of `StaFi`'s Stafi-Node is as follows:

Table 1.1: Basic Information of `StaFi`'s Stafi-Node

Item	Description
Name	Stafi Protocol
Website	https://stafi.io/
Type	Substrate based blockchain & L2
Platform	Rust
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/stafi-node/tree/rfis/node/pallets/rtoken (b102e88)

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

- https://github.com/stafiprotocol/stafi-node/tree/rfis/node/pallets/rtoken (680a3c3)

1.2 About PeckShield

PeckShield Inc. [5] 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 [4]:

- 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

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

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) [3], 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 Stafi-Node 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	0	
Low	1	
Informational	0	
Total	1	

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 low-severity vulnerability.

Table 2.1: Key Audit Findings of Stafi-Node Protocol

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Overflow Checks In liquidity_bond()	Numeric Errors	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 Improved Overflow Checks In liquidity_bond()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `stafi-node::rfis`
- Category: Numeric Errors [2]
- CWE subcategory: CWE-190 [1]

Description

The `stafi-node` is the node for `StaFi` protocol and allows users to bond `FIS` to get `rFIS` and redeem `FIS` with `rFIS` through `liquidity_bond()` and `liquidity_unbond()` functions. The `rFIS`-based tokens can then circulate without the need to unstake, hence solving the liquidity dilemma of staking.

In the following, we show the implementation of the `liquidity_bond()` function. This function transfers `FIS` from the user to the `controller` and mints `rFIS` in return to the depositor.

```
pub fn liquidity_bond(origin, pool: <T::Lookup as StaticLookup>::Source, value: BalanceOf<T>) -> DispatchResult {
        let who = ensure_signed(origin)?;
        ensure!(Self::nominate_switch(), Error::<T>::NominateSwitchClosed);
        ensure!(!value.is_zero(), Error::<T>::LiquidityBondZero);
        ensure!(staking::EraElectionStatus::<T>::get().is_closed(), staking::Error::<T>::CallNotAllowed);
        let controller = T::Lookup::lookup(pool)?;
        ensure!(Self::is_in_pools(&controller), Error::<T>::PoolNotFound);
        let mut ledger = staking::Ledger::<T>::get(&controller).ok_or(Error::<T>::PoolUnbond)?;

        let limit = Self::pool_balance_limit();
        let bonded = Self::bonded_of(&controller) + value;
        ensure!(limit.is_zero()  bonded <= limit, Error::<T>::PoolLimitReached);

        let v = value.saturated_into::<u128>();
        let rbalance = rtoken_rate::Module::<T>::token_to_rtoken(SYMBOL, v);

```

```
592             T::Currency::transfer(&who, &controller, value, AllowDeath)?;
593             T::RCurrency::mint(&who, SYMBOL, rbalance)?;
594
595             Self::bond_extra(&controller, &mut ledger, value);
596
597             Self::deposit_event(RawEvent::LiquidityBond(who, controller, value, rbalance
                    ));
598
599             Ok(())
600         }
```

Listing 3.1: stafi-node::liquidity_unbond()

However, we notice that the arithmetic operation to derive `bonded`, i.e., `let bonded = Self::bonded_of(&controller)+ value` (line 586), does not make use of `checked_add`. And this may unfortunately lead to an integer overflow if the given `value` is extremely large. Fortunately, it is unlikely as the user would have that huge amount of tokens to deposit.

Recommendation Use `checked_add` for the arithmetic operation in line 586.

Status This issue has been fixed in the commit: 680a3c3.

4 Conclusion

In this audit, we have analyzed the design and implementation of `StaFi`'s Stafi-Node, which is the node of `StaFi` protocol. The system is the first decentralized protocol unlocking liquidity of staked assets. 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 CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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