SMART CONTRACT AUDIT REPORT

for

Stader FTMStaking

Prepared By: Patrick Lou

PeckShield

April 19, 2022

Document Properties

Client	Stader
Title	Smart Contract Audit Report
Target	Stader FTMStaking
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Shulin Bie, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 19, 2022	Xuxian Jiang	Final Release
1.0-rc1	April 1, 2022	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Patrick Lou
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the FTMStaking support in the Stader 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 Stader

Stader aims to build native staking smart contracts across multiple chains including Terra, Solana, among others, and also develop an economic ecosystem to grow and develop solutions like YFI-style farming with rewards, launchpads, gaming with rewards, and liquid staking solutions. The audited FTMStaking support allows protocol users to stake their FTM to get FTMx, which represents the ownership of the staking pool and enables the claim of staking rewards. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The FTMStaking Protocol

Item	Description
Issuer	Stader
Website	https://staderlabs.com
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 19, 2022

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

- https://github.com/stader-labs/stader-ftmx-v0.git (0f47c9f)

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

- https://github.com/stader-labs/stader-ftmx-v0.git (ab6fc3c)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of 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 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 Logics	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 design and implementation of the `FTMStaking` support of the `Stader` protocol. 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	1	■
Low	3	■ ■ ■
Informational	0	
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 1 medium-severity vulnerability and 3 low-severity vulnerabilities.

Table 2.1: Key Stader FTMStaking Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Costly LPs From Improper Pool Initialization	Time and State	Resolved
PVE-002	Low	Generation of Meaningful Events For Setting Changes	Coding Practices	Resolved
PVE-003	Low	Penalty Consistency Between FTMStaking and SFC	Coding Practices	Resolved
PVE-004	Medium	Trust on Admin Keys	Security Features	Resolved

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 Costly LPs From Improper Pool Initialization

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: FTMStaking
- Category: Time and State [5]
- CWE subcategory: CWE-362 [3]

Description

The FTMStaking protocol allows users to deposit supported assets and get in return the share to represent the pool ownership. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the pool share extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the getFTMxAmountForFTM() routine, which is part of deposit logic. This routine is used for participating users to deposit the supported assets and get respective pool shares in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
    function getFTMxAmountForFTM(uint256 ftmAmount)
        public
        view
        returns (uint256)
    {
        uint256 totalFTM = totalFTMWorth();
        uint256 totalFTMx = FTMX.totalSupply();

        if (totalFTM == 0 || totalFTMx == 0) {
            return ftmAmount;
        }
        return (ftmAmount * totalFTMx) / totalFTM;
    }
```

Listing 3.1: FTMStaking::getFTMxAmountForFTM()

Specifically, when the pool is being initialized (line 217), the share value directly takes the value of `ftmAmount` (line 218), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `share = ftmAmount = 1 WEI`. With that, the actor can further deposit a huge amount of the underlying assets with the goal of making the pool share extremely expensive.

An extremely expensive pool share can be very inconvenient to use as a small number of 1 `Wei` may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current deposit logic to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure a guarded launch process that safeguards the first deposit to avoid being manipulated.

Status The issue has been resolved as the team will ensure a guarded launch process that safeguards the first deposit to avoid being manipulated.

3.2 Generation of Meaningful Events For Setting Changes

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `FTMStaking`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `FTMStaking` contract as an example. This contract is designed to allow protocol users to staking the `FTM` asset. While examining the events that reflect the changes of various settings, we notice there is a lack of emitting important events that reflect important setting changes. As an example, when the _withdrawalDelay parameter is updated in FTMStaking::setWithdrawalDelay(), there is no respective event emitted to reflect the withdrawal delay update (line 382).

```
365     function setValidatorPicker(IValidatorPicker picker) external onlyOwner {
366         validatorPicker = picker;
367     }

369     /**
370      * @notice Set epoch duration (onlyOwner)
371      * @param duration the new epoch duration
372      */
373     function setEpochDuration(uint256 duration) external onlyOwner {
374         _epochDuration = duration;
375     }

377     /**
378      * @notice Set withdrawal delay (onlyOwner)
379      * @param delay the new delay
380      */
381     function setWithdrawalDelay(uint256 delay) external onlyOwner {
382         _withdrawalDelay = delay;
383     }
```

Listing 3.2: Example Setters in FTMStaking

Recommendation Properly emit the respective events when the associated settings are updated.

Status This issue has been fixed in the following commit: 9653402.

3.3 Penalty Consistency Between FTMStaking and SFC

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: FTMStaking
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

The FTMStaking contract enforces certain penalty that will be charged when there is a need to undelegate locked assets. When analyzing the logic to compute penalty amount, we notice certain inconsistency in FTMStaking and the underlying Special Fee Contract (SFC) contract.

To elaborate, we show below the two related functions `FTMStaking::calculatePenalty` and `SFC::unlockStake()`. The inconsistency comes from the way to compute the unlock penalty in these two contracts. Specifically, the `FTMStaking` applies the pro-rata penalty share after summing up the penalty for all supported vaults while the `SFC` computes the pro-rata within the vault before the final summing-up. Though the inconsistency might be minimal, it is still helpful to maintain necessary consistency.

```
    function calculatePenalty(uint256 amountToUndelegate)
        public
        view
        returns (uint256)
    {
        uint256 totalStake;
        uint256 totalPenalty;
        uint256 vaultCount = maxVaultCount();
        for (uint256 i = 0; i < vaultCount; i = _uncheckedInc(i)) {
            address vault = _allVaults[i];
            if (vault != address(0)) {
                uint256 toValidatorID = Vault(vault).toValidatorID();
                totalStake += SFC.getStake(vault, toValidatorID);
                if (SFC.isLockedUp(vault, toValidatorID)) {
                    totalPenalty += _getUnlockPenalty(vault, toValidatorID);
                }
            }
        }
        return (amountToUndelegate * totalPenalty) / totalStake;
    }
```

Listing 3.3: `FTMStaking::calculatePenalty()`

```
    function unlockStake(uint256 toValidatorID, uint256 amount) external returns (uint256) {
        address delegator = msg.sender;
        LockedDelegation storage ld = getLockupInfo[delegator][toValidatorID];

        require(amount > 0, "zero amount");
        require(isLockedUp(delegator, toValidatorID), "not locked up");
        require(amount <= ld.lockedStake, "not enough locked stake");
        require(_checkAllowedToWithdraw(delegator, toValidatorID), "outstanding sFTM balance");

        _stashRewards(delegator, toValidatorID);

        uint256 penalty = _popDelegationUnlockPenalty(delegator, toValidatorID, amount, ld.lockedStake);

        ld.lockedStake -= amount;
        _rawUndelegate(delegator, toValidatorID, penalty);

        emit UnlockedStake(delegator, toValidatorID, amount, penalty);
        return penalty;
```

```
    }

    function _popDelegationUnlockPenalty(address delegator, uint256 toValidatorID,
        uint256 unlockAmount, uint256 totalAmount) internal returns (uint256) {
        uint256 lockupExtraRewardShare = getStashedLockupRewards[delegator][
            toValidatorID].lockupExtraReward.mul(unlockAmount).div(totalAmount);
        uint256 lockupBaseRewardShare = getStashedLockupRewards[delegator][toValidatorID
            ].lockupBaseReward.mul(unlockAmount).div(totalAmount);
        uint256 totalPenaltyAmount = lockupExtraRewardShare + lockupBaseRewardShare / 2;
        uint256 penalty = totalPenaltyAmount.mul(unlockAmount).div(totalAmount);
        getStashedLockupRewards[delegator][toValidatorID].lockupExtraReward =
            getStashedLockupRewards[delegator][toValidatorID].lockupExtraReward.sub(
            lockupExtraRewardShare);
        getStashedLockupRewards[delegator][toValidatorID].lockupBaseReward =
            getStashedLockupRewards[delegator][toValidatorID].lockupBaseReward.sub(
            lockupBaseRewardShare);
        if (penalty >= unlockAmount) {
            penalty = unlockAmount;
        }
        return penalty;
    }
```

Listing 3.4: `SFC::unlockStake()`

Recommendation Be consistent in the above penalty computation.

Status The issue has been resolved and the team clarifies that the penalty consistency was not required in the version shared earlier.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `FTMStaking`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `FTMStaking` protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the system-wide operations (e.g., configure various settings, pause/unpause the protocol, as well as update the vault owner). It also has the privilege to control or govern the flow of assets within the protocol contracts. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    function setEpochDuration(uint256 duration) external onlyOwner {
```

```
        _epochDuration = duration;
    }

    /**
     * @notice Set withdrawal delay (onlyOwner)
     * @param delay the new delay
     */
    function setWithdrawalDelay(uint256 delay) external onlyOwner {
        _withdrawalDelay = delay;
    }

    /**
     * @notice Set the owner of an arbitrary input vault (onlyOwner)
     * @param vault the vault address
     * @param newOwner the new owner address
     */
    function updateVaultOwner(address vault, address newOwner)
        external
        onlyOwner
    {
        // Needs to support arbitrary input address to work with expired/matured vaults
        Vault(vault).updateOwner(newOwner);
    }
```

Listing 3.5: Example Privileged Operations in `FTMStaking`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract 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.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changes to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been confirmed by the team. The team clarifies that the admin key has been transferred to a multi-sig account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `FTMStaking` support in the `Stader` protocol. The audited `FTMStaking` allows protocol users to stake their `FTM` to get `FTMx`, which represents the ownership of the staking pool and enables the claim of staking rewards. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that `Solidity`-based 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

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

[5] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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