SMART CONTRACT AUDIT REPORT

for

SafeStake

Prepared By: Xiaomi Huang

PeckShield

April 28, 2024

Document Properties

Client	SafeStake
Title	Smart Contract Audit Report
Target	SafeStake
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 28, 2024	Xuxian Jiang	Final Release
1.0-rc1	April 17, 2024	Xuxian Jiang	Release Candidate #1

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 `SafeStake` 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 SafeStake

`SafeStake` is a decentralized staking framework and protocol that maximizes staker rewards by keeping validators secure and online to perform `Ethereum Proof-of-Stake consensus (ETH2)` duties. It splits a validator key into shares and distributes them over several nodes run by independent operators to achieve high levels of security and fault tolerance. Written in `Rust`, `SafeStake` runs on top of the `ETH2/consensus` client and uses (a `BFT` consensus library) for consensus. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The SafeStake

Item	Description
Name	SafeStake
Website	https://safestake.xyz/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 28, 2024

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

- https://github.com/ParaState/SafeStake-Network-Contract.git (a04b936)

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

- https://github.com/ParaState/SafeStake-Network-Contract.git (20b7c3f)

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

Likelihood

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.

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) [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 implementation of the `SafeStake` 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	2	■■
Medium	1	■■
Low	2	■
Informational	0	
Total	5	

We have so far identified a list of potential issues. 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 high-severity vulnerabilities, 1 medium-severity vulnerability, and 2 low-severity vulnerabilities.

Table 2.1: Key SafeStake Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Constructor/Initialization Logic in SafeStakeWhiteList	Coding Practices	Resolved
PVE-002	Low	Improved Per-Owner Operator Limit Enforcement in SafeStakeRegistryV3	Business Logic	Resolved
PVE-003	High	Improved Validation in SafeStakeNetworkV3::deposit()	Coding Practices	Resolved
PVE-004	High	Incorrect Claimable Fee Calculation in SafeStakeNetworkV3	Business Logic	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Constructor/Initialization Logic in SafeStakeWhiteList

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SafeStakeWhiteList`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

To facilitate possible future upgrade, a number of contracts (including `SafeStakeWhiteList`) is instantiated as a proxy with actual logic contracts in the backend. While examining the related contract construction and initialization logic, we notice current construction can be improved.

In the following, we shows the constructor routine. We notice its constructor has a simple payload in calling `__Ownable_init()`. With that, it can be improved by adding the following statement, i.e., `_disableInitializers();`, though current modifier `initializer` serves a similar purpose. Note this statement is called in the logic contract where the initializer is locked. Therefore any user will not able to call the `initialize()` function in the state of the logic contract and perform any malicious activity. Note that the proxy contract state will still be able to call its own initialize function since the constructor does not effect the state of the proxy contract.

```
contract SafeStakeWhiteList is SafeStakeAccessControl, ISafeStakeWhiteList {

    mapping(address => bool ) public isWhite ;
    bool enable;

    constructor() initializer{
        __Ownable_init();
    }
    ...
```

```
}
```

Listing 3.1: `SafeStakeWhiteList::constructor()`

Moreover, the `initialize()` routine is currently missing and should be improved by calling `__Ownable_init()` instead.

Recommendation Improve the above-mentioned constructor routine in `SafeStakeWhiteList`. Note other contracts share the same issue, including `SafeStakeNetworkV3` and `SafeStakeRegistryV3`.

Status This issue has been resolved as the team confirms it will only be initialized once.

3.2 Improved Per-Owner Operator Limit Enforcement in SafeStakeRegistryV3

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SafeStakeRegistryV3`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `SafeStaking` protocol is no exception. Specifically, if we examine the `SafeStakeRegistryV3` contract, it has defined a number of protocol-wide risk parameters, such as `VALIDATORS_PER_OPERATOR_LIMIT` and `_REGISTERED_OPERATORS_PER_ACCOUNT_LIMIT`. In the following, we show the corresponding routines that make use of these parameters.

```
    function registerOperator(
        string calldata name,
        address ownerAddress,
        bytes calldata publicKey
    ) external override onlyRole(NODE_ROLE) returns (uint32 operatorId) {
        // check the operator amount of the owner
        require(_operatorsByOwnerAddress[ownerAddress].length <=
            REGISTERED_OPERATORS_PER_ACCOUNT_LIMIT,"A0");
        require(_owners[ownerAddress].validators.length == 0,"A1");
        // in current design, one address can't have operator and validator at the same
            time
        require(_operatorsByPublicKeys[publicKey] == 0,"A2");
        ...
    }
```

Listing 3.2: SafeStakeRegistryV3:: registerOperator ()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when enforcing them. Our analysis shows the above `registerOperator()` logic can be improved to properly honor the parameter `REGISTERED_OPERATORS_PER_ACCOUNT_LIMIT`, which defines maximum operators one address can register. Current implementation allows for one extra operator before the maximum number of operators.

Recommendation Improve the above routine to properly enforce these system-wide parameters.

Status The issue has been fixed by the following commits: `a814c75` and `62e39ba`.

3.3 Improved Validation in SafeStakeNetworkV3::deposit()

- ID: PVE-003
- Severity: High
- Likelihood: High
- Impact: High
- Target: `SafeStakeNetworkV3`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `SafeStake` protocol has a key `SafeStakeNetworkV3` contract manages the network-side validators and their operators. While reviewing current logic in allowing for the deposit into intended validators, we observe the logic is flawed.

To elaborate, we show below the implementation of the related `deposit()` routine. It basically transfers in the given amount of tokens and then updates the given set of validators. However, it does not validate the given amount for token transfer-in is equal to the sum of assigned numbers to the given set of validators. As a result, a malicious user may exploit it to increase the validator's balance and steal the funds from the contract.

```
    function deposit(bytes[] memory publickeys, uint256[] memory eachamounts,uint256
        totalamount) external {
        _deposit( msg.sender, totalamount );
        for(uint32 index=0; index < publickeys.length; ++index) {
            _updateValidatorBalance(publickeys[index], 0);
            _updateValidatorBalance(publickeys[index], eachamounts[index]);
        }
    }
```

Listing 3.3: `SafeStakeNetworkV3:deposit()`

In addition, the `accountClaimFee()` routine can also be improved by validating the recovered signer is not equal to `address(0)`.

Recommendation Revise the above-mentioned routines to properly validate user inputs.

Status The issue has been fixed by this commit: `fe33809`.

3.4 Incorrect Claimable Fee Calculation in SafeStakeNetworkV3

- ID: PVE-004
- Severity: High
- Likelihood: High
- Impact: High
- Target: `SafeStakeNetworkV3`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

By design, the `SafeStake` protocol distributes validator rewards to to its operators. In the process of analyzing the logic to compute claimable fee for related operators, we notice an issue that may use stale state for the fee calculation.

In the following, we show the implementation of the vulnerable `accountClaimFee()` routine. As the name indicates, this routine allows for claiming the operator fee for the given account. We notice the fee calculation is based on the following statement (line 227): `can_claim = (endBlockNumber - detail.startBlockNumber)* detail.lastOperatorFee + op_detail.earnings`, where the `detail.startBlockNumber` state should be replaced as `max(detail.startBlockNumber, op_detail.lastBlockNumber)`. Note other two routines `getOperatorEarningsByPublicKey()` and `_removeValidatorUnsafe()` share the same issue.

```
202     function accountClaimFee(address account, uint32[] calldata operatorIds,
203         uint32[] calldata performances, uint256 nonce, bytes memory signature)
204         external checkClaimNonce(nonce){

206         bytes32 hash = prefixed(
207             keccak256(
208                 abi.encode(account, operatorIds, performances, nonce)
209             )
210         );

212         address signer = recoverSigner(hash, signature);
213         require(hasRole(SIGNER_ROLE, signer),"G1");

215         uint256 claimed = 0 ;
216         uint256 penalty = 0 ;

218         for(uint32 a=0; a < operatorIds.length; ++a) {
219             uint32 operatorId = operatorIds[a];
220             require(performances[a] <= 100,"G2");
221             uint256 earnings = 0;
222             for(uint32 b=0; b < _operatorInDatas[operatorId].length; ++b) {
223                 bytes memory publicKey = _operatorInDatas[operatorId][b].publicKey;
224                 ValidatorData storage detail = _validatorDatas[publicKey];
```

```
                OperatorWorkDetail storage op_detail = _operatorWorkDetail[operatorId][
                    publicKey];
                uint256 endBlockNumber = detail.endBlockNumber <= block.number ? detail.
                    endBlockNumber : block.number;
                uint256 can_claim = (endBlockNumber - detail.startBlockNumber) * detail.
                    lastOperatorFee + op_detail.earnings;
                op_detail.earnings = 0;
                op_detail.lastBlockNumber = block.number;
                earnings += can_claim;
            }
            uint256 true_claim = earnings * performances[a] / 100;
            penalty += earnings - true_claim;
            claimed += true_claim;
        }
        self.networkPenalty += penalty;

        require(_token.transfer(account, claimed), "G3");
        _claimNonce[nonce] = true;
        emit AccountClaim(nonce, account, claimed, penalty, block.number);
    }
```

Listing 3.4: `SafeStakeNetworkV3::accountClaimFee()`

Recommendation Revise the above routine to properly update operator fees.

Status The issue has been fixed by the following commits: `f752675`. and `1c8bd7f2`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In `SafeStake`, there is a privileged administrative account (with authorized roles, e.g., `ADMIN_ROLE`). The administrative account plays a critical role in governing and regulating the protocol-wide operations. Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `SafeStakeRegistryV3` contract as an example and show the representative functions potentially affected by the privileges of the administrative account.

```
    function enableOwnerValidators(
        address ownerAddress
    ) external override onlyRole(NODE_ROLE) {
        require(
```

```
            _owners[ownerAddress].validatorsDisabled, "F1"
        );
        _activeValidatorCount += _owners[ownerAddress].activeValidatorCount;
        _owners[ownerAddress].validatorsDisabled = false;
    }

    /**
     * @dev See {ISafeStakeRegistry-disableOwnerValidators}.
     */
    function disableOwnerValidators(
        address ownerAddress
    ) external override onlyRole(NODE_ROLE) {
        require(
            !_owners[ownerAddress].validatorsDisabled, "G1"
        );
        require(
            _owners[ownerAddress].activeValidatorCount > 0, "G2"
        );
        _activeValidatorCount -= _owners[ownerAddress].activeValidatorCount;
        _owners[ownerAddress].validatorsDisabled = true;
    }
```

Listing 3.5: Example Privileged Operations in `SafeStakeRegistryV3`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the administrative account may also be a counter-party risk to the protocol users. It would be worrisome if the privileged administrative account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

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 This issue has been mitigated with the plan to transfer the privileged account to a `multi-sig` account.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `SafeStake` protocol, which is a decentralized staking framework and protocol that maximizes staker rewards by keeping validators secure and online to perform `Ethereum Proof-of-Stake consensus (ETH2)` duties. It splits a validator key into shares and distributes them over several nodes run by independent operators to achieve high levels of security and fault tolerance. Written in `Rust`, `SafeStake` runs on top of the `ETH2/consensus` client and uses (a `BFT` consensus library) for consensus. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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