



# SMART CONTRACT AUDIT REPORT

for

## SafeStake



Prepared By: Xiaomi Huang

PeckShield  
December 15, 2023

## Document Properties

Client	SafeStake
Title	Smart Contract Audit Report
Target	SafeStake
Version	1.0
Author	Xuxian Jiang
Auditors	Jonathan Zhao, Colin Zhong, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

## Version Info

Version	Date	Author(s)	Description
1.0	December 15, 2023	Xuxian Jiang	Final Release
1.0-rc1	November 30, 2023	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

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	About SafeStake . . . . .	4
1.2	About PeckShield . . . . .	5
1.3	Methodology . . . . .	5
1.4	Disclaimer . . . . .	7
<b>2</b>	<b>Findings</b>	<b>9</b>
2.1	Summary . . . . .	9
2.2	Key Findings . . . . .	10
<b>3</b>	<b>Detailed Results</b>	<b>11</b>
3.1	Public Exposure of Privileged Functions . . . . .	11
3.2	Revisited removeInitiator() Logic in SafeStakeRegistryV2 . . . . .	12
3.3	Incorrect Validator Enabling And Disabling Logic . . . . .	13
3.4	Timely Network Earnings/Fee Update Before New Validator Update . . . . .	14
3.5	Trust Issue of Admin Keys . . . . .	15
<b>4</b>	<b>Conclusion</b>	<b>17</b>
	<b>References</b>	<b>18</b>

# 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	<a href="https://safestake.xyz/">https://safestake.xyz/</a>
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 15, 2023

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> (7925dc3)

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> (df13753)

## 1.2 About PeckShield

PeckShield Inc. [7] 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](mailto:contact@peckshield.com)).

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [6]:

- 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
Additional Recommendations	Holistic Risk Management
	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) [5], 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
<b>Configuration</b>	Weaknesses in this category are typically introduced during the configuration of the software.
<b>Data Processing Issues</b>	Weaknesses in this category are typically found in functionality that processes data.
<b>Numeric Errors</b>	Weaknesses in this category are related to improper calculation or conversion of numbers.
<b>Security Features</b>	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
<b>Time and State</b>	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.
<b>Error Conditions, Return Values, Status Codes</b>	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.
<b>Resource Management</b>	Weaknesses in this category are related to improper management of system resources.
<b>Behavioral Issues</b>	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
<b>Business Logics</b>	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.
<b>Initialization and Cleanup</b>	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
<b>Arguments and Parameters</b>	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
<b>Expression Issues</b>	Weaknesses in this category are related to incorrectly written expressions within code.
<b>Coding Practices</b>	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	3	
Low	0	
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 and 3 medium-severity vulnerabilities.

Table 2.1: Key SafeStake Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Public Exposure of Privileged Functions in SafeStakeAccessControl	Security Features	Resolved
PVE-002	Medium	Revisited removeInitiator() Logic in SafeStakeRegistryV2	Business Logic	Resolved
PVE-003	Medium	Incorrect Validator Enabling And Disabling Logic	Business Logic	Resolved
PVE-004	High	Timely Network Earnings/Fee Update Before New Validator Update	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 Public Exposure of Privileged Functions

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: Multiple Contracts
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

#### Description

The `SafeStake` protocol is a unique decentralized staking framework and protocol that maximizes staker rewards. To facilitate the staking management, the protocol has a number of privileged functions. While examining these privileged functions, we notice some of them are publicly exposed without caller verification.

In the following, we show two example privileged routines from the `SafeStakeAccessControl` contract. These two routines are designed to configure the access control for role authorization. However, these two routines are public and their public exposure without any caller authentication will corrupt the protocol integrity or cripple the entire protocol functionality.

```
54     function grantRole(bytes32 role, address account) public virtual {
55         if (!hasRole(ADMIN_ROLE, account)) {
56             _roles[role].members[account] = true;
57         }
58     }
59
60     function revokeRole(bytes32 role, address account) public virtual {
61         if (hasRole(ADMIN_ROLE, account)) {
62             _roles[role].members[account] = false;
63         }
64     }
```

Listing 3.1: `SafeStakeAccessControl::grantRole()/revokeRole()`

**Recommendation** Revisit all public functions and add necessary caller verification. Note this issue affects a few public functions, including `SafeStakeCount::updateAddressNetworkFee()`, `SafeStakeNetworkV2::enableOwnerValidators()`, `SafeStakeNetworkV2::disableOwnerValidators()`, and `SafeStakeRegistryV2::removeInitiator()`.

**Status** This issue has been fixed by the following commits: 25f546e and 83e880.

## 3.2 Revisited `removeInitiator()` Logic in `SafeStakeRegistryV2`

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `SafeStakeRegistryV2`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

### Description

The `SafeStake` protocol has a key `SafeStakeRegistryV2` contract to keep track of current operators, validators, initiators, and their owners. In the process of reviewing the initiator removal logic, we notice current implementation needs to be revisited.

To elaborate, we show below the implementation of the related `removeInitiator()` routine. As the name indicates, this routine is designed to remove an active initiator. While it properly locates the index of the initiator for removal, it does not accordingly update related data structures (such as `_initiatorsByOperator`) for each associated `operatorId`.

```

628     function removeInitiator(address owner, uint32 initiatorId) external override {
629         _onlyInitiatorOwner(owner, initiatorId);
630         Initiator storage initiator = _initiators[initiatorId];
631         require(initiator.active);
632         initiator.active = false;
633         uint32 index = 0;
634         bool found = false;
635         for (; index < _initiatorsByOwnerAddress[owner].length; index++) {
636             if (_initiatorsByOwnerAddress[owner][index] == initiatorId) {
637                 found = true;
638                 break;
639             }
640         }
641         require(found);
642         _initiatorsByOwnerAddress[owner][index] = _initiatorsByOwnerAddress[owner][
            _initiatorsByOwnerAddress[owner].length - 1];
643         _initiatorsByOwnerAddress[owner].pop();
644         emit InitiatorRemoval(initiatorId);
645     }

```

Listing 3.2: `SafeStakeRegistryV2::removeInitiator()`

Moreover, it is also suggested to check whether we need to verify the initiator status before the removal.

**Recommendation** Revise the above routine for proper initiator removal.

**Status** This issue has been fixed by the following commit: 3b6ec14.

### 3.3 Incorrect Validator Enabling And Disabling Logic

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: SafeStakeRegistryV2
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

#### Description

As mentioned earlier, the key SafeStakeRegistryV2 contract is used to keep track of current operators, validators, initiators, and their owners. In the process of reviewing the validator enabling and disabling logic, we observe the logic needs to be revisited.

To elaborate, we show below the implementation of the related `enableOwnerValidators()` and `disableOwnerValidators()` routines. If we examine the former, it is used to enable a given validator. However, it validates the following requirement: `require(!_owners[ownerAddress].validatorsDisabled)` (line 290). This requirement is in conflict with the purpose and should be revised as `require(_owners[ownerAddress].validatorsDisabled)`. The same reasoning is also applicable to the `disableOwnerValidators()` routine.

```

286     function enableOwnerValidators(
287         address ownerAddress
288     ) external override onlyRole(NODE_ROLE) {
289         require(
290             !_owners[ownerAddress].validatorsDisabled
291         );
292         _activeValidatorCount += _owners[ownerAddress].activeValidatorCount;
293         _owners[ownerAddress].validatorsDisabled = false;
294     }
295
296     /**
297      * @dev See {ISafeStakeRegistry-disableOwnerValidators}.
298      */
299     function disableOwnerValidators(
300         address ownerAddress
301     ) external override onlyRole(NODE_ROLE) {
302         require(
303             _owners[ownerAddress].validatorsDisabled

```

```

304     );
305     require(
306         _owners[ownerAddress].activeValidatorCount > 0
307     );
308     _activeValidatorCount -= _owners[ownerAddress].activeValidatorCount;
309     _owners[ownerAddress].validatorsDisabled = true;
310 }

```

Listing 3.3: SafeStakeRegistryV2:enableOwnerValidators()/disableOwnerValidators()

**Recommendation** Revise the above-mentioned routines to properly enable and/or disable the given validator. Note the same issue is also applicable to the same routines in SafeStakeNetworkV2 contract.

**Status** This issue has been fixed by the following commits: b8d240b and c83601c.

### 3.4 Timely Network Earnings/Fee Update Before New Validator Update

- ID: PVE-004
- Severity: High
- Likelihood: High
- Impact: High
- Target: SafeStakeNetworkV2
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

#### Description

By design, the SafeStake protocol may collect network earning fee for the operation. In the process of analyzing the logic to compute available network earning and fee, we notice an issue that may use stale state for the earning calculation.

In the following, we use the `registerValidator()` routine as an example. This routine is used to register a new validator with the bulk logic implemented in an internal helper `_registerValidatorUnsafe()`. However, it comes to our attention that this routine may increase the active validator number, which directly affects the network earning and fee calculation. To fix, there is a need to invoke `_updateNetworkEarnings()` and `_updateAddressNetworkFee()` before calling the above helper `_registerValidatorUnsafe()`.

```

109     function registerValidator(
110         bytes calldata publicKey,
111         uint32[] calldata operatorIds,
112         bytes[] calldata sharesPublicKeys,
113         bytes[] calldata sharesEncrypted,
114         uint256 amount

```

```

115     ) external override {
116         _registerValidatorUnsafe(
117             msg.sender,
118             publicKey,
119             operatorIds,
120             sharesPublicKeys,
121             sharesEncrypted,
122             amount
123         );
124         _updateNetworkEarnings();
125         _updateAddressNetworkFee(msg.sender);
126     }

```

Listing 3.4: SafeStakeNetworkV2::registerValidator()

**Recommendation** Revise the above routine to properly update network earning and owner's network fee before a new validator is registered. Note the same issue affects a number of routines, including `registerValidator()`, `batchRegisterValidator()`, `initiatorStake()`, `disableOwnerValidators()`, and `enableOwnerValidators()`.

**Status** The issue has been fixed by this commit: 475221c.

## 3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

### 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 `SafeStakeRegistryV2` contract as an example and show the representative functions potentially affected by the privileges of the administrative account.

```

135     function certifyOperatorFromDao(
136         uint32 operatorId
137     ) external override onlyRole(SIGNER_ROLE) {
138         Operator storage operator = _operators[operatorId];
139         require(operator.active);
140
141         operator.fromParaStateDao = true;
142     }

```

```

143     emit OperatorCertifiedFromDao(operatorId);
144 }
145
146 /**
147  * @dev See {ISafeStakeRegistry-verifyOperator}.
148  */
149 function verifyOperator(
150     uint32 operatorId
151 ) external override onlyRole(SIGNER_ROLE) {
152     Operator storage operator = _operators[operatorId];
153     require(operator.active);
154
155     operator.verified = true;
156     emit OperatorVerified(operatorId);
157 }
158
159 /**
160  * @dev See {ISafeStakeRegistry-updateOperatorFee}.
161  */
162 function updateOperatorFee(
163     uint64 fee
164 ) external override onlyRole(NODE_ROLE) {
165     _operatorFee = fee;
166 }

```

Listing 3.5: Example Privileged Operations in SafeStakeRegistryV2

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-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [2] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [3] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [4] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [5] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [6] OWASP. Risk Rating Methodology. [https://www.owasp.org/index.php/OWASP\\_Risk\\_Rating\\_Methodology](https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology).
- [7] PeckShield. PeckShield Inc. <https://www.peckshield.com>.