SMART CONTRACT AUDIT REPORT

for

Eth2 Liquidity Staking

Prepared By: Xiaomi Huang

PeckShield

May 26, 2022

Document Properties

Client	Persistence
Title	Smart Contract Audit Report
Target	Eth2 Liquidity Staking
Version	1.0
Author	Xuxian Jiang
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 26, 2022	Xuxian Jiang	Final Release
1.0-rc1	May 20, 2022	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 smart contract source code of the `Persistence` protocol on the support of `Eth2 Liquidity Staking`, 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 Persistence

`Persistence` is a cross-chain liquid staking solution which helps in unlocking the liquidity of the staked assets. It is designed as an inter-chain DeFi product to enable the liquid staking of PoS chains. The audited support of `Eth2 Liquidity Staking` allows for customers to participate in `Ethereum 2.0 (ETH2.0)` staking at a low threshold, earn yield from `Ethereum` lock-up rewards, while simultaneously benefiting from additional token rewards, mining revenues, and better liquidity from related DeFi projects. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Eth2 Liquidity Staking

Item	Description
Target	Eth2 Liquidity Staking
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 26, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note this audit covers the `rohit/stketh_exchange_rat` branch.

- https://github.com/persistenceOne/eth2-liquid-staking-contracts.git (7c80b30)

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

- https://github.com/persistenceOne/eth2-liquid-staking-contracts.git (8395d1b)

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 accordingly classified into four categories, 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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 `Persistence` protocol on the support of `Eth2 Liquidity Staking`. 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	2	■ ■
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 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 2 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Potential Slash Prevention in StakingPool::slash()	Time and State	Resolved
PVE-002	Low	Improved Validation in Oracle::pushData()	Coding Practice	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Beside the identified issues, we emphasize that for any user-facing applications and services, 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 should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Potential Slash Prevention in StakingPool::slash()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: StakingPool
- Category: Time and State [6]
- CWE subcategory: CWE-682 [3]

Description

As mentioned earlier, the Eth2 Liquidity Staking support enables the customers to participate in Ethereum 2.0 (ETH2.0) staking at a low threshold, earn yield from Ethereum lock-up rewards, while simultaneously benefiting from additional token rewards, mining revenues, and better liquidity from related DeFi projects. Because of that, there is a constant need of swapping one asset to another.

To elaborate, we show below the slash() function that is designed to slash the given balance of the staking pool. The slashing functionality in essence swaps certain amount of PSTAKE token to stkETH, which will then be burnt.

```
    function slash(uint256 amount) external override {

        require(_msgSender() == core.oracle(), "StakingPool: only oracle can call to
            slash");

        uint256 pstakeBalance = pstake.balanceOf(address(this));

        if(pstakeBalance == 0){
            return;
        }
        uint256 pstakePrice = oracle.price();
        address[] memory path = new address[](3);
        path[0] = address(pstake);
        path[1] = WETH;
        path[2] = address(stkEth);
        uint256[] memory amountsIn = router.getAmountsIn(amount, path);
```

```

        if(!validateDeviation(pstakePrice, amountsIn[0], amount)){
            return;
        }

        if(amountsIn[0] > pstakeBalance){
            pstake.approve(address(router), pstakeBalance);
            router.swapExactTokensForTokens(pstakeBalance, 0, path, address(this), block
                .timestamp + 100);
        }else{
            pstake.approve(address(router), amountsIn[0]);
            router.swapTokensForExactTokens(amount, amountsIn[0], path, address(this),
                block.timestamp + 100);
        }

        stkEth.burn(address(this), stkEth.balanceOf(address(this)));

    }
```

Listing 3.1: StakingPool::slash()

It comes to our attention that the conversion is routed to `UniswapV2` to swap one asset to another. To prevent the swap rate from being manipulated, this `slash()` routine adds the deviation check and reverts if the deviation is larger than the allowed range. Meanwhile, due to the deviation check, it is possible to avoid the slash penalty by intentionally failing the deviation check.

Recommendation This is a design tradeoff in balancing the possible denial-of-service and reduced MEV exposure.

Status This issue has been resolved as this function is not used anywhere as of now.

3.2 Improved Validation in Oracle::pushData()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Oracle`
- Category: Coding Practices [5]
- CWE subcategory: CWE-628 [2]

Description

At the core of `Eth2 Liquidity Staking` is the `Oracle` contract for voting, reward distribution and slashing. While examining one of its functions `pushData()`, we notice it can be benefited from improved validation.

In particular, we show below its implementation. This function is designed to allow for trusted oracle members to push data to oracle. While its business logic is rather straightforward, we notice the validation of the internal `currentFrameEpochId` variable can be improved. Specifically, it is currently validated by the following two requirements: `currentFrameEpochId >= _getFrameFirstEpochId(currentFrameEpochId, beaconData)` (lines 413-414) and `currentFrameEpochId <= _getFrameFirstEpochId(currentFrameEpochId, beaconData)+ beaconData.epochsPerTimePeriod` (lines 417-418). However, the second requirement can be improved as `currentFrameEpochId < _getFrameFirstEpochId(currentFrameEpochId, beaconData)+ beaconData.epochsPerTimePeriod` since `currentFrameEpochId` needs to be strictly smaller than the current frame end time.

```
409     function pushData(
410         uint256 latestEthBalance,
411         uint256 latestNonce,
412         uint32 numberOfValidators
413     ) external override {
414         require(isOracle(msg.sender), "Not oracle Member");
415         uint256 currentFrameEpochId = _getCurrentEpochId(beaconData);
416 
417         require(
418             currentFrameEpochId > lastCompletedEpochId,
419             "Cannot push to Epoch less that already commited"
420         );
421         require(
422             currentFrameEpochId >=
423             _getFrameFirstEpochId(currentFrameEpochId, beaconData)
424         );
425         require(
426             currentFrameEpochId <=
427             _getFrameFirstEpochId(currentFrameEpochId, beaconData) +
428             beaconData.epochsPerTimePeriod
429         );
430 
431         require(latestNonce == nonce.current(), "incorrect Nonce");
432         require(
433             activatedValidators <= numberOfValidators,
434             "Invalid numberOfValidators"
435         );
436         latestEthBalance = latestEthBalance * ETH2_DENOMINATION;
437         bytes32 candidateId = keccak256(
438             abi.encode(nonce, latestEthBalance, numberOfValidators)
439         );
440         bytes32 voteId = keccak256(abi.encode(msg.sender, candidateId));
441         require(!submittedVotes[voteId], "Oracles: already voted");
442         ...
443     }
```

Listing 3.2: `Oracle::pushData()`

Recommendation Improve the validation of `currentFrameEpochId` in the above `pushData()`

function.

Status The issue has been resolved by following the above suggestion.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

The `Eth2 Liquidity Staking` support has a privileged `governor` account (with the `GOVERN_ROLE`) that plays a critical role in governing and regulating the protocol-wide operations (e.g., adding various roles, setting token contracts, and configuring various system parameters). In the following, we show representative privileged operations in the protocol's core `Permissions` contract.

```
    function grantMinter(address minter) public override onlyGovernor {
        grantRole(MINTER_ROLE, minter);
    }

    /// @notice grants burner role to address
    /// @param burner new burner
    function grantBurner(address burner) public override onlyGovernor {
        grantRole(BURNER_ROLE, burner);
    }

    /// @notice grants node operator role to address
    /// @param nodeOperator new nodeOperator
    function grantNodeOperator(address nodeOperator) public override onlyGovernor {
        grantRole(NODE_OPERATOR_ROLE, nodeOperator);
    }

    /// @notice grants key admin role to address
    /// @param keyAdmin new keyAdmin
    function grantKeyAdmin(address keyAdmin) public override onlyGovernor {
        grantRole(KEY_ADMIN_ROLE, keyAdmin);
    }

    /// @notice grants governor role to address
    /// @param governor new governor
    function grantGovernor(address governor) public override onlyGovernor {
        grantRole(GOVERN_ROLE, governor);
    }
```

Listing 3.3: Example Privileged Functions in `Permissions`

We emphasize that the privilege assignment may be necessary and consistent with the token design. However, it is worrisome if the `governor` is not governed by a `DAO`-like structure. Note that a compromised `governor` account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the entire PoS bridge design.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed 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 confirmed and partially mitigated by saving the admin keys in cold vault.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Persistence` protocol on the support of `Eth2 Liquidity Staking`. The system allows the customers to participate in `Ethereum 2.0 (ETH2.0)` staking at a low threshold, earn yield from `Ethereum` lock-up rewards, while simultaneously benefiting from additional token rewards, mining revenues, and better liquidity from related DeFi projects. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

[3] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.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: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.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.