SMART CONTRACT AUDIT REPORT

for

PSTAKE

Prepared By: Yiqun Chen

PeckShield

November 18, 2021

Document Properties

Client	Persistence
Title	Smart Contract Audit Report
Target	pStake
Version	1.0
Author	Xuxian Jiang
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 18, 2021	Xuxian Jiang	Final Release
1.0-rc1	October 19, 2021	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Yiqun Chen
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 pStake 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 pStake

pStake is a liquid staking solution which helps in unlocking the liquidity of the staked assets. The pStake solution is built up using ERC20 contracts over Ethereum blockchain. It is designed as an inter-chain DeFi product of Persistence to enable enable liquid staking of PoS chains. In particular, it works with pBridge to allow for transfer of value between multiple disparate blockchains like Ethereum, Cosmos , Persistence etc. Unlike other bridges available in the blockchain ecosystem which only facilitates creating peg tokens, the bridge solution can also perform inter-chain transactions pertaining to PoS staking and unstaking, which are fulfilled at the protocol level in the native chain.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of pStake

Item	Description
Target	pStake
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 18, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the audited repository contains a number of sub-directories (e.g., `holder`, `interfaces`, and `libraries`) and this audit covers only `StakeLP`, `WhitelistedPTokenEmission`, and `WhitelistedRewardEmission` contracts.

- https://github.com/persistenceOne/pStakeSmartContracts.git (285a5ff)

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

- https://github.com/persistenceOne/pStakeSmartContracts.git (8f631a1)

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 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 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

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

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.

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.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of `pStake`. 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	3	
Low	2	
Informational	0	
Total	5	

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 3 medium-severity vulnerabilities and 2 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Inconsistency Natspec Comments With Implementation	Coding Practices	Fixed
PVE-002	Medium	Improved Logic of _removeToken-ContractForRewards()	Business Logic	Fixed
PVE-003	Medium	Incorrect sToken Whitelisting in setWhitelistedAddress()	Business Logic	Fixed
PVE-004	Low	Improved Reentrancy Prevention in StakeLP	Time And State	Fixed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

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 Inconsistency Natspec Comments With Implementation

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

The pStake protocol is well-documented with the extensive use of NatSpec comments to provide rich documentation for functions, return variables and others. In the process of analyzing current NatSpec comments, we notice the presence of numerous inconsistency with the code implementation.

To elaborate, we show below the _setHolderAddressForRewards() function from the WhitelistedReward-Emission contract. This function is designed to configure reward tokens for a given holder contract. However, the preceding function summary shows it is used to "calculate liquidity and reward tokens and disburse to user", which is very misleading!

```
    /*
     * @dev calculate liquidity and reward tokens and disburse to user
     * @param lpToken: lp token contract address
     * @param amount: token amount
     */
    function _setHolderAddressForRewards(
        address holderContractAddress,
        address[] memory rewardTokenContractAddresses
    ) internal returns (bool success) {
        // add the Holder Contract address if it isn't already available
        if (!_holderContractList.contains(holderContractAddress)) {
            _holderContractList.add(holderContractAddress);
        }

        uint256 i;
        uint256 _rewardTokenContractAddressesLength = rewardTokenContractAddresses
```

```
                .length;
        for (i = 0; i < _rewardTokenContractAddressesLength; i = i.add(1)) {
            // add the Token Contract addresss to the reward tokens list for the Holder
                Contract
            if (rewardTokenContractAddresses[i] != address(0)) {
                // search if the reward token contract is already part of list
                if (
                    _rewardTokenListIndex[holderContractAddress][
                        rewardTokenContractAddresses[i]
                    ] == 0
                ) {
                    _rewardTokenList[holderContractAddress].push(
                        rewardTokenContractAddresses[i]
                    );
                    _rewardTokenListIndex[holderContractAddress][
                        rewardTokenContractAddresses[i]
                    ] = _rewardTokenList[holderContractAddress].length;
                }
            }
        }
        success = true;
        return success;
    }
```

Listing 3.1: `WhitelistedRewardEmission::_setHolderAddressForRewards()`

Note the three audited contracts share the same issue with numerous inconsistent `NatSpec` comments.

Recommendation Remove the inconsistency among the identified misleading `NatSpec` comments.

Status The issue has been fixed by this commit: `8f631a1`.

3.2 Improved Logic of _removeTokenContractForRewards()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `WhitelistedRewardEmission`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, the `pStake` protocol is a liquid staking solution which helps in unlocking the liquidity of the staked assets. It has a helper `WhitelistedRewardEmission` contract to manage the related reward rate and cumulative amount. While examining the logic for reward token removal, we notice a flawed handling.

To elaborate, we show below the related `_removeTokenContractForRewards()` function from this contract. This function has a straightforward logic in removing a set of reward tokens from the given holder contract. However, the actual removal logic efficiently swaps the removed reward token with the last one in the `_rewardTokenList` array, but fails to properly update the reverse lookup index on the replacement reward token (lines 1316-1324).

```
1291     /*
1292      * @dev calculate liquidity and reward tokens and disburse to user
1293      * @param lpToken: lp token contract address
1294      * @param amount: token amount
1295      */
1296     function _removeTokenContractForRewards(
1297         address holderContractAddress,
1298         address[] memory rewardTokenContractAddresses
1299     ) internal returns (bool success) {
1300         uint256 i;
1301         uint256 _rewardTokenContractAddressesLength = rewardTokenContractAddresses
1302                 .length;
1303         for (i = 0; i < _rewardTokenContractAddressesLength; i = i.add(1)) {
1304             if (rewardTokenContractAddresses[i] != address(0)) {
1305                 // remove the token address from the list
1306                 uint256 rewardTokenListIndexLocal = _rewardTokenListIndex[
1307                     holderContractAddress
1308                 ][rewardTokenContractAddresses[i]];
1309                 if (rewardTokenListIndexLocal > 0) {
1310                     if (
1311                         rewardTokenListIndexLocal ==
1312                         _rewardTokenList[holderContractAddress].length
1313                     ) {
1314                         _rewardTokenList[holderContractAddress].pop();
1315                     } else {
1316                         _rewardTokenList[holderContractAddress][
1317                             rewardTokenListIndexLocal.sub(1)
1318                         ] = _rewardTokenList[holderContractAddress][
1319                             _rewardTokenList[holderContractAddress].length.sub(
1320                                 1
1321                             )
1322                         ];
1323                         _rewardTokenList[holderContractAddress].pop();
1324                     }

1326                     // delete the index value
1327                     delete _rewardTokenListIndex[holderContractAddress][
1328                         rewardTokenContractAddresses[i]
1329                     ];
1330                 }
1331             }
1332         }
```

```
        success = true;
        return success;
    }
```

Listing 3.2: `WhitelistedRewardEmission::_removeTokenContractForRewards()`

Recommendation Revise the `_removeTokenContractForRewards()` logic to properly update the reverse lookup index of the replacement reward token.

Status The issue has been fixed by this commit: `8f631a1`.

3.3 Incorrect sToken Whitelisting in setWhitelistedAddress()

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `WhitelistedPTokenEmission`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

To efficiently manage reward tokens, the `pStake` protocol has a helper `WhitelistedPTokenEmission` contract. This contract has a permissioned function `setWhitelistedAddress()` that is used to whitelist the supported `sToken`. While examining its whitelisting logic, we notice a flawed implementation.

To elaborate, we show below the flawed `setWhitelistedAddress()` function. As the name indicates, this function is designed to whitelist the given `sTokenAddresses`. However, the function can be improved to validate that each given `sTokenAddress` indeed exists. (This specific validation is currently missing.) Moreover, the internal `for`-loop (line 299) uses the wrong `j` index to reference the given `sTokenAddresses` (line 310). The proper index should be the `i` index used in the `for`-loop!

```
249   function setWhitelistedAddress(
250      address whitelistedAddress,
251      address[] memory sTokenAddresses,
252      address holderContractAddress,
253      address lpContractAddress
254   ) public virtual override returns (bool success) {
255      require(hasRole(DEFAULT_ADMIN_ROLE, _msgSender()), "WP2");
256      // lpTokenERC20ContractAddress or sTokenReserveContractAddress can be address(0)
            but not whitelistedAddress
257      // have set holderContract also as non-zero, can allow lpContractAddress to be
            zero for control
258      require(
259         whitelistedAddress != address(0) &&
260            holderContractAddress != address(0) &&
```

```
            sTokenAddresses.length != 0,
            "WP3"
        );

        bool whitelistedAddressExists;
        uint256 j;
        uint256 i;

        // ADD TO _holderWhitelists make sure the same whitelisted address is not added
            before if so revert
        // to add the same whitelisted address, first remove it, then add again
        // add to _holderWhitelists
        for (
            j = 0;
            j < _holderWhitelists[holderContractAddress].length;
            j = j.add(1)
        ) {
            if (
                _holderWhitelists[holderContractAddress][j] ==
                whitelistedAddress
            ) {
                whitelistedAddressExists = true;
                break;
            }
        }

        // if whitelisted contract doesnt already exist then include it in the array else
            revert
        if (!whitelistedAddressExists) {
            // add the whitelistedAddress to the _holderWhitelists array
            _holderWhitelists[holderContractAddress].push(whitelistedAddress);
        } else {
            revert("WP4");
        }

        // ADD TO _holderWhitelists AND _whitelistedSTokenAddresses AND
            _whitelistedAddressHolder AND _holderLPToken
        _whitelistedAddressHolder[whitelistedAddress] = holderContractAddress;
        _holderLPToken[holderContractAddress] = lpContractAddress;

        // add SToken addresses uniquely to _holderSTokens
        for (i = 0; i < sTokenAddresses.length; i = i.add(1)) {
            // ADD TO _whitelistedSTokenAddresses
            // check if sTokenAddress already exists
            // check if all the sTokenAddresses provided are non zero
            require(sTokenAddresses[i] != address(0), "WP5");
            _whitelistedSTokenAddresses[holderContractAddress][
                whitelistedAddress
            ].push(sTokenAddresses[i]);

            // SET WHITELISTING IN STOKEN CONTRACTS
            // for each sTokenAddress, set the whiteliste data
```

```
        ISTokensV2(sTokenAddresses[j]).setWhitelistedAddress(
            whitelistedAddress,
            holderContractAddress,
            lpContractAddress
        );
        // ADD TO _holderWhitelists
        for (
            j = 0;
            j < _holderSTokens[holderContractAddress].length;
            j = j.add(1)
        ) {
            if (
                sTokenAddresses[i] ==
                _holderSTokens[holderContractAddress][j]
            ) {
                break;
            }
        }
        if (j == _holderSTokens[holderContractAddress].length) {
            _holderSTokens[holderContractAddress].push(sTokenAddresses[i]);
        }
    }

    // emit event
    emit SetWhitelistedAddress(
        whitelistedAddress,
        _whitelistedSTokenAddresses[holderContractAddress][
            whitelistedAddress
        ],
        holderContractAddress,
        lpContractAddress,
        block.timestamp
    );

    success = true;
    return success;
}
```

Listing 3.3: `WhitelistedPTokenEmission::setWhitelistedAddress()`

Recommendation Revise the above `setWhitelistedAddress()` logic to use the right index to invoke `ISTokensV2(sTokenAddresses[i]).setWhitelistedAddress()`.

Status The issue has been fixed by this commit: `8f631a1`.

3.4 Improved Reentrancy Prevention in StakeLP

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `StakeLP`
- Category: Time and State [6]
- CWE subcategory: CWE-663 [2]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [11] exploit, and the recent `Uniswap/Lendf.Me` hack [10].

We notice there are several occasions where the `checks-effects-interactions` principle is violated. Using the `StakeLP` as an example, the `calculateSyncedRewards()` function (see the code snippet below) is provided to calculate rewards for the provided `holderAddress`. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

In particular, the interaction with the external contract inside `_calculateRewards()` (line 383) starts before effecting the update on the internal state, hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
356     function calculateSyncedRewards(address holderAddress)
357         public
358         virtual
359         override
360         whenNotPaused
361         returns (
362             uint256[] memory RewardAmounts,
363             address[] memory RewardTokens,
364             address[] memory sTokenAddresses,
365             address lpTokenAddress
366         )
367     {
368         // check for validity of arguments
369         require(holderAddress != address(0), "LP1");
370 
371         // initiate calculateHolderRewards for all StokenAddress-whitelistedAddress pair
              that
372         // comes under the holder contract
```

```
        IWhitelistedPTokenEmission(_whitelistedPTokenEmissionContract)
            .calculateAllHolderRewards(holderAddress);

        // now initiate the calculate Rewards to distribute to the user
        // calculate liquidity and reward tokens and disburse to user
        (
            RewardAmounts,
            RewardTokens,
            sTokenAddresses,
            lpTokenAddress
        ) = _calculateRewards(holderAddress, _msgSender());

        require(lpTokenAddress != address(0), "LP2");

        emit TriggeredCalculateSyncedRewards(
            holderAddress,
            _msgSender(),
            RewardAmounts,
            RewardTokens,
            sTokenAddresses,
            block.timestamp
        );
    }
```

Listing 3.4: `StakeLP::calculateSyncedRewards()`

Note that the `removeLiquidity()` function has the proper `nonReentrant` modifier to prevent potential `re-entrancy`. It is also suggested to apply the same modifier to other functions, including `addLiquidity()` and `calculateSyncedRewards()`.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to the above-mentioned functions.

Status The issue has been fixed by this commit: `8f631a1`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: High
- Likelihood: High
- Impact: High
- Target: `StakeLP`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The `pStake` protocol has a privileged `admin` account 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 `StakeLP` contract.

```
    function setWhitelistedPTokenEmissionContract(
        address whitelistedPTokenEmissionContract
    ) public virtual override {
        require(hasRole(DEFAULT_ADMIN_ROLE, _msgSender()), "LP4");
        _whitelistedPTokenEmissionContract = whitelistedPTokenEmissionContract;
        emit SetWhitelistedPTokenEmissionContract(
            whitelistedPTokenEmissionContract
        );
    }

    function setWhitelistedRewardEmissionContract(
        address whitelistedRewardEmissionContract
    ) public virtual override {
        require(hasRole(DEFAULT_ADMIN_ROLE, _msgSender()), "LP5");
        _whitelistedRewardEmissionContract = whitelistedRewardEmissionContract;
        emit SetWhitelistedRewardEmissionContract(
            whitelistedRewardEmissionContract
        );
    }
```

Listing 3.5: Example Privileged Functions in `StakeLP`

We emphasize that the privilege assignment may be necessary and consistent with the token design. However, it is worrisome if the `admin` is not governed by a `DAO`-like structure. Note that a compromised `admin` 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. Specifically, the team confirms the strategic roadmap has governance for `pSTAKE` protocol to be implemented and the admin keys being controlled by `MPC` (multi party computation), which the team has already implemented in effect for the bridge component of the protocol.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `pStake` protocol. The system presents a unique offering as a liquid staking solution which helps in unlocking the liquidity of the staked assets. It also works closely with `pBridge` to efficiently transfer assets, including `ERC20`, `ERC721`, `ERC1155` and many other token standards, between multiple disparate blockchains like `Ethereum`, `Cosmos` , `Persistence` etc. 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-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

[6] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.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.

[10] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[11] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.