



# SMART CONTRACT AUDIT REPORT

for

## pStake



Prepared By: Yiqun Chen

PeckShield  
January 7, 2022

## Document Properties

Client	Persistence
Title	Smart Contract Audit Report
Target	pStake
Version	1.0
Author	Patrick Liu
Auditors	Patrick Liu, Xuxian Jiang
Reviewed by	Patrick Liu
Approved by	Xuxian Jiang
Classification	Public

## Version Info

Version	Date	Author	Description
1.0	January 7, 2022	Patrick Liu	Final Release
1.0-rc	December 28, 2021	Patrick Liu	Release Candidate

## 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

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	About pStake . . . . .	4
1.2	About PeckShield . . . . .	5
1.3	Methodology . . . . .	5
1.4	Disclaimer . . . . .	7
<b>2</b>	<b>Findings</b>	<b>8</b>
2.1	Summary . . . . .	8
2.2	Key Findings . . . . .	9
<b>3</b>	<b>ERC20 Compliance Checks</b>	<b>10</b>
<b>4</b>	<b>Detailed Results</b>	<b>13</b>
4.1	Possible Multiple Instances Of pStake Token . . . . .	13
4.2	Trust Issue Of Admin Keys . . . . .	14
<b>5</b>	<b>Conclusion</b>	<b>16</b>
	<b>References</b>	<b>17</b>

# 1 | Introduction

Given the opportunity to review the design document and related source code of the `pStake` protocol, we outline in the report our systematic method to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistency between smart contract code and the documentation, 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 liquid staking of PoS chains. This audit aims to check its ERC20-compliance as well as associated business logic.

The basic information of `pStake` is as follows:

Table 1.1: Basic Information of `pStake`

Item	Description
Issuer	Persistence
Website	<a href="https://pstake.finance/">https://pstake.finance/</a>
Type	Ethereum ERC20 Token Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	January 7, 2022

In the following, we show the git repository and the commit hash value used in this audit:

- <https://github.com/persistenceOne/ERC20/tree/vesting> (b6a9e03)

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

- <https://github.com/persistenceOne/ERC20/tree/vesting> (1568cb5)

## 1.2 About PeckShield

PeckShield Inc. [6] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystem 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)).

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [5]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk;

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.2: Vulnerability Severity Classification

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

We perform the audit according to the following procedures:

- 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.

- ERC20 Compliance Checks: We then manually check whether the implementation logic of the audited smart contract(s) follows the standard ERC20 specification and other best practices.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

Table 1.3: The Full List of Check Items

Category	Check Item
<b>Basic Coding Bugs</b>	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
	Approve / TransferFrom Race Condition
<b>ERC20 Compliance Checks</b>	Compliance Checks (Section 3)
<b>Additional Recommendations</b>	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool 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.

## 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.



## 2 | Findings

### 2.1 Summary

Here is a summary of our findings after analyzing the `pStake` token contract. 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 ERC20-related aspects under scrutiny to uncover possible pitfalls and/or bugs

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	
Low	0	
Informational	0	
Total	2	

Moreover, we explicitly evaluate whether the given contracts follow the standard ERC20 specification and other known best practices, and validate its compatibility with other similar ERC20 tokens and current DeFi protocols. The detailed ERC20 compliance checks are reported in Section 3. After that, we examine a few identified issues of varying severities that need to be brought up and paid more attention to. (The findings are categorized in the above table.) Additional information can be found in the next subsection, and the detailed discussions are in Section 4.

### 2.2 Key Findings



Overall, no ERC20 compliance issue was found and our detailed checklist can be found in Section 3. Also, though current smart contracts are well-designed and engineered, the implementation and deployment can be further improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities.

Table 2.1: Key pStake Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Possible Multiple Instances Of pStake Token	Coding Practices	Fixed
PVE-002	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate these issues, we 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 our detailed compliance checks and Section 4 for elaboration of reported issues.



### 3 | ERC20 Compliance Checks

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as the first step of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Table 3.1: Basic `view-only` Functions Defined in The ERC20 Specification

Item	Description	Status
<b>name()</b>	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
<b>symbol()</b>	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
<b>decimals()</b>	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
<b>totalSupply()</b>	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
<b>balanceOf()</b>	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
<b>allowance()</b>	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

Our analysis shows that there is no ERC20 inconsistency or incompatibility issue found in the audited pStake. In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-adopted ERC20

Table 3.2: Key State-Changing Functions Defined in The ERC20 Specification

Item	Description	Status
<b>transfer()</b>	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
<b>transferFrom()</b>	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	✓
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
<b>approve()</b>	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	—
<b>Transfer() event</b>	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to <i>address(0x0)</i> when new tokens are generated	✓
<b>Approval() event</b>	Is emitted on any successful call to approve()	✓

specification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777/ERC2222), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Table 3.3: Additional `opt-in` Features Examined in Our Audit

Feature	Description	Opt-in
<b>Deflationary</b>	Part of the tokens are burned or transferred as fee while on <code>transfer()/transferFrom()</code> calls	—
<b>Rebasing</b>	The <code>balanceOf()</code> function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
<b>Pausable</b>	The token contract allows the owner or privileged users to pause the token transfers and other operations	—
<b>Blacklistable</b>	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	—
<b>Mintable</b>	The token contract allows the owner or privileged users to mint tokens to a specific address	✓
<b>Burnable</b>	The token contract allows the owner or privileged users to burn tokens of a specific address	—

## 4 | Detailed Results

### 4.1 Possible Multiple Instances Of pStake Token

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Orchestrator
- Category: Coding Practices [4]
- CWE subcategory: CWE-1109 [1]

#### Description

At the core of the pStake protocol is the `Orchestrator` contract, which plays a key role on the pStake token deployment and the integration of various token vesting strategies. While reviewing the logic of this `Orchestrator` contract, we notice a key function needs to be improved.

In particular, it comes to our attention that the `mintAndTransferTokens()` function is implemented to instantiate a pStake token on each invocation. As a result, if this function is invoked multiple times, it is possible for the protocol to have multiple instances of pStake tokens. To elaborate, we show below the implementation of this `mintAndTransferTokens()` function.

```
42  function mintAndTransferTokens() external onlyOwner returns (address[] memory vestings) {
43
44      token = new pStake(address(this), mainOwner);
45      vestings = new address[](vestingInfos.length);
46      for(uint i=0; i<vestingInfos.length; i++){
47          VestingInfo memory vestingInfo = vestingInfos[i];
48          address vesting = deploy();
49          StepVesting(vesting).initialize(
50              token,
51              vestingInfo.cliffTime,
52              vestingInfo.stepDuration,
53              vestingInfo.cliffAmount,
54              vestingInfo.stepAmount,
55              vestingInfo.numOfSteps,
56              vestingInfo.beneficiary);
```

```

57         token.transfer(vesting, vestingInfo.cliffAmount + vestingInfo.stepAmount*
58             vestingInfo.numOfSteps);
59         vestings[i] = vesting;
60         vestingMapping[vestingInfo.beneficiary] = vesting;
61     }
62 }

```

Listing 4.1: `Orchestrator::mintAndTransferTokens()`

Note that the function is guarded with the `onlyOwner` modifier, indicating it can only be successfully invoked by the privileged owner. However, the presence of possibly multiple instances of `pStake` (line 44) greatly undermines the integrity of the entire protocol. By design, there is a need to ensure that there is only one instance of the `pStake` token for the system to work properly.

**Recommendation** Move the `pStake` instantiation logic to the `Orchestrator::constructor()` to make sure there is only one `pStake` instance in the system.

**Status** The issue has been fixed by this commit: 1568cb5.

## 4.2 Trust Issue Of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `pStake`, `Orchestrator`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [2]

### Description

In the `pStake` contract, there is a privileged `minter` account that allows to mint additional tokens into circulation. In the following, we list the related `setMinter()` and `mint()` functions.

```

91     function setMinter(address minter_) external {
92         require(msg.sender == minter, "pStake: only the minter can change the minter address");
93         emit MinterChanged(minter, minter_);
94         minter = minter_;
95     }
96
97     /**
98      * @notice Mint new tokens
99      * @param dst The address of the destination account
100      * @param rawAmount The number of tokens to be minted
101      */
102     function mint(address dst, uint rawAmount) external {
103         require(msg.sender == minter, "pStake: only the minter can mint");

```

```

104     require(dst != address(0), "pStake: cannot transfer to the zero address");
105
106     // mint the amount
107     uint96 amount = safe96(rawAmount, "pStake: amount exceeds 96 bits");
108     uint96 safeSupply = safe96(totalSupply, "pStake: totalSupply exceeds 96 bits");
109     totalSupply = add96(safeSupply, amount, "pStake: totalSupply exceeds 96 bits");
110
111     // transfer the amount to the recipient
112     balances[dst] = add96(balances[dst], amount, "pStake: transfer amount overflows");
113     emit Transfer(address(0), dst, amount);
114
115     // move delegates
116     _moveDelegates(address(0), delegates[dst], amount);
117 }

```

Listing 4.2: pStake::setMinter()/pStake::mint()

We also notice that in the `Orchestrator` contract, there is a privileged owner account that plays a critical role in governing and regulating the entire operation and maintenance. To elaborate, we list below the `constructor()` function.

```

29     constructor(VestingInfo[] memory _vestingInfos, address _mainOwner) {
30         mainOwner = _mainOwner;
31
32         for(uint i=0; i<_vestingInfos.length; i++){
33             VestingInfo memory vestingInfo = _vestingInfos[i];
34             vestingInfos.push(vestingInfo);
35         }
36
37         StepVesting stepVesting = new StepVesting();
38         vestingImplementation = address(stepVesting);
39     }

```

Listing 4.3: Orchestrator::constructor()

We understand the need of the privileged functions for contract operation, but at the same time the extra power to the admin role may also be a counter-party risk to the contract users. It would be worrisome if the related role account is a plain EOA account. A multi-sig account will greatly alleviate this concern, though it is far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the privileged roles to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered for mitigation.

**Recommendation** Properly transfer the privileged account to the intended DAO-like governance contract. 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. The team mentioned that the `pStake` minter role will be a timelock-based smart contract.

## 5 | Conclusion

In this security audit, we have examined the design and implementation of the `pStake` protocol. During our audit, we first checked all respects related to the compatibility of the ERC20 specification and other known ERC20 pitfalls/vulnerabilities. We then proceeded to examine other areas such as coding practices and business logics. Overall, we identified two issues that were promptly confirmed and addressed by the team. In the meantime, as disclaimed in Section [1.4](#), we appreciate any constructive feedbacks or suggestions about our findings, procedures, audit scope, etc.





## References

- [1] MITRE. CWE-1109: Use of Same Variable for Multiple Purposes. <https://cwe.mitre.org/data/definitions/1109.html>.
- [2] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [3] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [4] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [5] OWASP. Risk Rating Methodology. [https://www.owasp.org/index.php/OWASP\\_Risk\\_Rating\\_Methodology](https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology).
- [6] PeckShield. PeckShield Inc. <https://www.peckshield.com>.