SMART CONTRACT AUDIT REPORT

for

Preon Finance (AMO)

Prepared By: Xiaomi Huang

PeckShield

November 10, 2024

Document Properties

Client	Preon
Title	Smart Contract Audit Report
Target	Preon
Version	1.0
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 10, 2024	Xuxian Jiang	Final Release
1.0-rc	November 8, 2024	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 | Introduction

Given the opportunity to review the design document and related source code of the AMO support in Preon, 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 Preon

Preon is a decentralized protocol that allows Ether or liquid staking derivatives (LSDs) holders to obtain maximum liquidity against their collateral without paying interest. It took inspiration from Gravita and is designed to be multi-collateral. Each position can have only one collateral type and it is linked to a specific stability pool. Also, the protocol allows cross-collateral positions, linked to a single stability pool. Note the protocol-wide debt token is called STAR, while the governance token is PREON. This audit covers its sub-product on the AMO vault and associated strategy. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Preon

Item	Description
Name	Preon
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	November 10, 2024

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the given repository has a number of contract file and directories and this audit only covers the following two contracts: AMOVault.sol and AerodromeAMOStrategy.sol (under

`PSM/amo/aerodrome`).

- https://github.com/PreonMoney/preon-contracts.git (d8f7ea3)

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

- https://github.com/PreonMoney/preon-contracts.git (8047182)

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.

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

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 1.4: 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 the AMO support in Preon. 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	3	■ ■ ■
Informational	0	
Total	4	

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 3 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Constructor Logic in AMOVault/AerodromeAMOStrategy	Coding Practices	Resolved
PVE-002	Low	Accommodation of Non-ERC2-Compliant Tokens	Coding Practices	Resolved
PVE-003	Low	Revisited Rebalance Logic in AerodromeAMOStrategy	Business Logic	Resolved
PVE-004	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 Improved Constructor Logic in AMOVault/AerodromeAMOStrategy

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

Description

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

In the following, we show its initialization routine. We notice its constructor does not have any payload. With that, it can be improved by adding the following statement, i.e., _disableInitializers();. Note this statement is called in the logic contract where the initializer is locked. Therefore any user will not able to call the initialize() function in the state of the logic contract and perform any malicious activity. Note that the proxy contract state will still be able to call this function since the constructor does not effect the state of the proxy contract.

```
    function initialize(
        address _star,
        address _strategy,
        address _stable,
        uint256 _swapFee,
        address _feeRecipient
    ) external override initializer {
        __ReentrancyGuard_init();
        __Ownable_init();

        require(_star != address(0), "!star");
        require(_stable != address(0), "!stable");
```

```
        require(_strategy != address(0), "!strategy");
        star = IStarToken(_star);
        strategy = _strategy;
        stable = IERC20Upgradeable(_stable);
        DECIMAL_CONVERSION = 10 ** (18 - IERC20Override(_stable).decimals());
        swapFeeCompliment = SWAP_FEE_DENOMINATOR - _swapFee;
        feeRecipient = _feeRecipient;
        swapFee = _swapFee;
        vaultBuffer = 1 ether;
    }
```

Listing 3.1: `AMOVault::initialize()`

Recommendation Improve the above-mentioned constructor routine in the `AMOVault` contract. Note the associated strategy contract can be similarly improved.

Status This issue has been fixed in the following commit: `8088ecb`.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
```

```
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.2: ZRX::transfer()/transferFrom()

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `safeApproveAllTokens()` routine in the `AerodromeAMOStrategy` contract. If the `USDT` token is supported as `WETH`, the unsafe version of `IERC20(WETH).approve(address(swapRouter), 0)` (line 976) may revert as there is no return value in the `USDT` token contract's `approve()` implementation (but the `IERC20` interface expects a return value)!

```
965     function safeApproveAllTokens() external onlyOwner nonReentrant {
966         // to add liquidity to the clPool
967         IERC20(OETHb).approve(address(positionManager), type(uint256).max);
968         // to be able to rebalance using the swapRouter
969         IERC20(OETHb).approve(address(swapRouter), type(uint256).max);
970
971         /* the behaviour of this strategy has slightly changed and WETH could be
972          * present on the contract between the transactions. For that reason we are
973          * un-approving WETH to the swapRouter & positionManager and only approving
974          * the required amount before a transaction
975          */
976         IERC20(WETH).approve(address(swapRouter), 0);
977         IERC20(WETH).approve(address(positionManager), 0);
978     }
```

Listing 3.3: AerodromeAMOStrategy::safeApproveAllTokens()

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

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

3.3 Revisited Rebalance Logic in AerodromeAMOStrategy

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

Description

As mentioned earlier, the `AMO` strategy in `Preon` brings extra rewards to protocol users by depositing the liquidity into the `Aerodrome DEX` pairs. In the process of rebalancing the liquidity, we notice current rebalance implementation can be improved.

In particular, we show below the implementation of the related routine. It has a rather straightforward logic in (1) withdrawing the intended liquidity, (2) swapping to the desired position, and (3) then adding back the liquidity. Our analysis indicates that the first step only withdraws the intended liquidity, which can be improved to draw all strategy liquidity so that the subsequent second-step swap operation can avoid unnecessary impact and influence from its own liquidity.

```
    function _rebalance(
        uint256 _amountToSwap,
        bool _swapWeth,
        uint256 _minTokenReceived
    ) internal {
        /**
         * Would be nice to check if there is any total liquidity in the pool before
            performing this swap
         * but there is no easy way to do that in UniswapV3:
         * - clPool.liquidity() -> only liquidity in the active tick
         * - asset[1&2].balanceOf(address(clPool)) -> will include uncollected tokens of
             LP providers
         *   after their liquidity position has been decreased
         */
        /**
         * When rebalance is called for the first time there is no strategy
         * liquidity in the pool yet. The full liquidity removal is thus skipped.
         * Also execute this function when WETH is required for the swap.
         */
        if (tokenId != 0 && _swapWeth && _amountToSwap > 0) {
            _ensureWETHBalance(_amountToSwap);
        }
        // in some cases we will just want to add liquidity and not issue a swap to move
            the
        // active trading position within the pool
```

```
        if (_amountToSwap > 0) {
            _swapToDesiredPosition(_amountToSwap, _swapWeth, _minTokenReceived);
        }
        ...
    }
```

Listing 3.4: `AerodromeAMOStrategy::_rebalance()`

Recommendation Revise the above logic to reduce the swap cost as much as possible.

Status This issue has been resolved.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Preon` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., configure parameters, manage contracts, and upgrade proxies). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
    function setFee(uint256 _newSwapFee) external override onlyOwner {
        require(_newSwapFee <= MAX_SWAP_FEE, ">MAX_SWAP_FEE");
        swapFee = _newSwapFee;
        swapFeeCompliment = SWAP_FEE_DENOMINATOR - _newSwapFee;
        emit NewFeeSet(_newSwapFee);
    }
    ...
    function setDebtLimit(uint256 _newDebtLimit) external override onlyOwner {
        starDebtLimit = _newDebtLimit;
        emit NewDebtLimitSet(_newDebtLimit);
    }
    ...
    function toggleRedeemPaused(bool _paused) external override onlyOwner {
        redeemPaused = _paused;
        emit RedeemPauseToggle(_paused);
    }
    ...
    function setFeeRecipient(
        address _newFeeRecipient
```

```
    ) external override onlyOwner {
        require(_newFeeRecipient != address(0), "!feeRecipient");
        feeRecipient = _newFeeRecipient;
        emit NewFeeRecipientSet(_newFeeRecipient);
    }
    ...
    function setStrategistAddr(
        address _newStrategistAddr
    ) external override onlyOwner {
        require(_newStrategistAddr != address(0), "!strategistAddr");
        strategistAddr = _newStrategistAddr;
        emit NewStrategistAddrSet(_newStrategistAddr);
    }
```

Listing 3.5: Example Privileged Functions in `AMOVault`

Note that if the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. 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. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts may have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

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 mitigated as the team makes use of a multisig to act as the privileged owner.

4 Conclusion

In this audit, we have analyzed the design and implementation of the AMO support in Preon, which is a decentralized protocol that allows Ether or liquid staking derivatives (LSDs) holders to obtain maximum liquidity against their collateral without paying interest. It took inspiration from Gravita and is designed to be multi-collateral. Each position can have only one collateral type and it is linked to a specific stability pool. Also, the protocol allows cross-collateral positions, linked to a single stability pool. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that Solidity-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

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

[7] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.