SMART CONTRACT AUDIT REPORT

for

Lode

Prepared By: Xiaomi Huang

PeckShield

December 21, 2024

Document Properties

Client	Lode
Title	Smart Contract Audit Report
Target	Lode
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	December 21, 2024	Xuxian Jiang	Final Release
1.0-rc	September 22, 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 smart contract source code of the Lode 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 Lode

The Lode Funding Rate Farming protocol is an automated platform designed for users to farm positive funding rates on tradable assets while maintaining a delta-neutral position. By utilizing spot longs and leveraged shorts, the system allows users to profit from funding rate arbitrage without exposure to market direction. Key features include automated position management, real-time monitoring of funding rates and collateral, and rebalancing to mitigate liquidation risk. The protocol provides tools for users to deposit stablecoins, open positions, adjust short positions, and execute trades with built-in protections like stop losses and principal preservation. The basic information of Lode is as follows:

Table 1.1: Basic Information of Lode

Item	Description
Target	Lode
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	December 21, 2024

In the following, we show the Git repository of reviewed files and the commit hash values used in this audit. Note the given repo has a number of contracts and this audit only covers the following contracts[1]: `Account.sol`, `AccountsCenter.sol`, and `BaseAccount.sol`.

- https://github.com/Intent-X/sf-core-contracts.git (6ecfebc)

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 classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

[1] With that, this audit is considered as partial audit and does not cover the integration of external protocols.

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

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.

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 `Lode` implementation. 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 logic, 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 that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability and 2 low-severity vulnerabilities.

Table 2.1: Key Lode Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Validation of Function Arguments in closeSymmioPosition()	Coding Practices	Resolved
PVE-002	Low	Strengthened State Transition Condition in withdrawFromSubAccount()	Business Logic	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 Improved Validation of Function Arguments in closeSymmioPosition()

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

Description

The Lode protocol has a core Account contract that serves as the base for account management. It is also used to centrally manage various sub-accounts and their positions. In the process of examining the position-closing logic, we notice the given input arguments can be better validated.

In the following, we show the implementation of the related closeSymmioPosition() routine. As the name indicates, this routine is used to close a position within the Symmio system for a specific sub-account. With that, there is a need to validate the given quoteId is indeed associated with the given sub-account. To remedy, we can enforce the following requirement, i.e., require(currentSubAccountPositionId[subAccount_] == closeRequestParams_.quoteId).

```
    function closeSymmioPosition(
        bool shouldRefund_,
        address subAccount_,
        CloseRequestPositionParams calldata closeRequestParams_
    ) external gasRefund(shouldRefund_) onlyOwnerOrKeeper {
        _multiAccount()._call(
            subAccount_,
            _toArrayWithOneElement(
                abi.encodeWithSelector(
                    ISymmio.requestToClosePosition.selector,
                    closeRequestParams_.quoteId,
                    closeRequestParams_.closePrice,
                    closeRequestParams_.quantityToClose,
```

```
                    closeRequestParams_.orderType,
                    closeRequestParams_.deadline
                )
            )
        );
    }
```

Listing 3.1: Account::closeSymmioPosition()

Recommendation Improve the above routine by validating the given `quoteId` is indeed associated with the given sub-account. Note another routine `forceCloseSymmioPosition()` can be similarly improved.

Status The issue has been resolved. The team confirms that there is no such need as `Symmio` has its own checks to validate that the specified `quoteId` belongs to the specified `subAccount`, and it will not allow unauthorized or invalid calls.

3.2 Strengthened State Transition Condition in withdrawFromSubAccount()

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

Description

As mentioned in Section `subsec:pve001`, the `Account` contract keeps track of various sub-accounts and their positions. For each sub-account, it maintains a state-transition machine to guard the sub-account operation. While examining the state-transition from `POSITION_WAITING_WITHDRAW` to `CLOSED`, we notice the need of enforcing the `_WITHDRAW_DELAY` parameter. And current implementation can be improved by restricting the `_WITHDRAW_DELAY` enforcement only during the specific state transition.

To elaborate, we show below the implementation of the related `withdrawFromSubAccount()` routine. As the name indicates, this routine is designed to withdraw funds from a sub-account for a specific position. And the withdrawal delay enforcement only occurs when the position state is `POSITION_WAITING_WITHDRAW`, not `QUOTE_WAITING_WITHDRAW`.

```
    function withdrawFromSubAccount(
        bool shouldRefund_,
        uint256 id_,
        FeeDiscountSignature memory signature_
```

```
    ) external gasRefund(shouldRefund_) onlyOwnerOrKeeper {
        DeltaNeutralPosition memory position = positionsInfo[id_];
        if (
            position.status !=
            DeltaNeutralPositionStatus.POSITION_WAITING_WITHDRAW &&
            position.status != DeltaNeutralPositionStatus.QUOTE_WAITING_WITHDRAW
        ) {
            revert InvalidDeltaNeutralPositionStatus();
        }
        if (
            position.startWithdrawTimestamp + _WITHDRAW_DELAY > block.timestamp
        ) {
            revert WithdrawDelayWindow();
        }
        ...
    }
```

Listing 3.2: `Account::withdrawFromSubAccount()`

Recommendation Revise the above logic to enforce `_WITHDRAW_DELAY` only when current position status is `POSITION_WAITING_WITHDRAW`.

Status The issue has been resolved. The team clarifies that `_WITHDRAW_DELAY` is a duplicate `Symmio` of the deallocate delay, which is applied to all allocated `symmio` balances, so both `POSITION_WAITING_WITHDRAW` and `QUOTE_WAITING_WITHDRAW` will have a 12 hour delay for withdraw Therefore, there is no point in applying it for one state out of 2, since it will be applied in any case, and in case of a change in the case, this duplicate delay will most likely be removed in the future.

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 [2]

Description

The `Lode` protocol has a privileged account (with the `DEFAULT_ADMIN_ROLE` privilege) that plays a critical role in governing and regulating the protocol-wide operations (e.g., assign roles, configure parameters, pause/unpause the protocol, and upgrade proxies). It also has the privilege to control or govern the flow of assets among various protocol components. In the following, we examine the privileged account and related privileged accesses in current contracts.

```
    function setCollateral(
        address collateral_
    )
        external
        virtual
        override
        onlyRole(DEFAULT_ADMIN_ROLE)
        notZeroAddress(collateral_)
    {
        collateral = collateral_;
        emit SetCollateral(collateral_);
    }
    ...
    function setSymmioAddress(
        address symmioAddress_
    )
        external
        virtual
        override
        onlyRole(DEFAULT_ADMIN_ROLE)
        notZeroAddress(symmioAddress_)
    {
        symmioAddress = symmioAddress_;
        emit SetSymmioAddress(symmioAddress_);
    }
    ...
    function setMultiAccount(
        address multiAccount_
    )
        external
        virtual
        override
        onlyRole(DEFAULT_ADMIN_ROLE)
        notZeroAddress(multiAccount_)
    {
        multiAccount = multiAccount_;
        emit SetMultiAccount(multiAccount_);
    }
    ...
    function setSwapRouterV3(
        address swapRouterV3_
    )
        external
        virtual
        override
        onlyRole(DEFAULT_ADMIN_ROLE)
        notZeroAddress(swapRouterV3_)
    {
        swapRouterV3 = swapRouterV3_;
        emit SetSwapRouterV3(swapRouterV3_);
    }
    ...
```

```
    function setTresuary(
        address tresuary_
    )
        external
        virtual
        override
        onlyRole(DEFAULT_ADMIN_ROLE)
        notZeroAddress(tresuary_)
    {
        tresuary = tresuary_;
        emit SetTresuary(tresuary_);
    }
    ...
    function upgradeTo(
        address implementation_
    )
        external
        virtual
        override
        onlyRole(DEFAULT_ADMIN_ROLE)
        notZeroAddress(implementation_)
    {
        implementation = implementation_;
        emit Upgraded(implementation_);
    }
    ...
    function pause() external virtual override onlyRole(DEFAULT_ADMIN_ROLE) {
        _pause();
    }
    ...
    function unpause() external virtual override onlyRole(DEFAULT_ADMIN_ROLE) {
        _unpause();
    }
```

Listing 3.3: Example Privileged Operations in `AccountsCenter`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Moreover, it should be noted that current contracts 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 `owner` privilege to the intended `DAO`-like governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated with the use of a multisig as the admin.

4 Conclusion

In this audit, we have analyzed the design and implementation of `Lode` protocol, which is an automated platform designed for users to farm positive funding rates on tradable assets while maintaining a delta-neutral position. By utilizing spot longs and leveraged shorts, the system allows users to profit from funding rate arbitrage without exposure to market direction. Key features include automated position management, real-time monitoring of funding rates and collateral, and rebalancing to mitigate liquidation risk. The protocol provides tools for users to deposit stablecoins, open positions, adjust short positions, and execute trades with built-in protections like stop losses and principal preservation. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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.