SMART CONTRACT AUDIT REPORT

for

Apollox

Prepared By: Xiaomi Huang

PeckShield

May 10, 2023

Document Properties

Client	Apollox
Title	Smart Contract Audit Report
Target	Apollox
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Patrick Lou, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 10, 2023	Xuxian Jiang	Final Release
1.0-rc1	May 2, 2023	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 **4**

1.1 About Apollox . . . 4
1.2 About PeckShield . . . 5
1.3 Methodology . . . 5
1.4 Disclaimer . . . 7

2 Findings **9**

2.1 Summary . . . 9
2.2 Key Findings . . . 10

3 Detailed Results **11**

3.1 Improved Settlement Logic in TradingCloseFacet . . . 11
3.2 Incorrect Pair Slippage Update Logic in LibPairsManager . . . 12
3.3 Suggested Adherence of Checks-Effects-Interactions . . . 13
3.4 Improved Role Member Management in LibAccessControlEnumerable . . . 15
3.5 Incorrect Fee Total Accounting in LibFeeManager . . . 16
3.6 Trust Issue of Admin Keys . . . 17

4 Conclusion **19**

References **20**

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Apollox 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 Apollox

Apollox is a decentralized derivatives trading platform. The main architecture of Apollox V2 is the fully on-chain liquidity model for more transparent, low slippage trades. Users do not need to register, deposit or withdraw funds. All V2 trades are executed against the ALP pool on BNB Smart Chain and liquidity for all V2 trading pairs is shared via the ALP pool to maximize capital efficiency. Real time price feeds will be taken from both Binance Oracle and Chainlink to ensure the most accurate pricing. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Apollox

Item	Description
Target	Apollox
Website	https://www.apollox.finance
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 10, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note this protocol assumes a trusted external oracle, which is not part of the audit.

- https://github.com/apollox-finance/apollox-perp-contracts.git (a38e3b5)

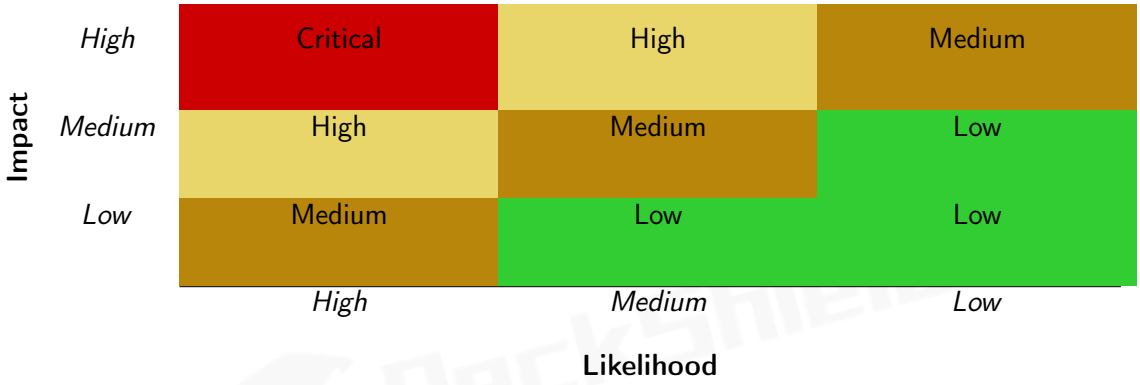
And this is the Git repository and commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/apollox-finance/apollox-perp-contracts.git (e56cc7a)

1.2 About PeckShield

PeckShield Inc. [11] 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



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- 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 1.3: 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) [9], 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 `Apollox` 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	1	■
Medium	2	■■
Low	3	■■■
Informational	0	
Total	6	

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 high-severity vulnerability, 1 medium-severity vulnerability, and 3 low-severity vulnerabilities.

Table 2.1: Key Apollox Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Settlement Logic in Trading-CloseFacet	Business Logic	Resolved
PVE-002	High	Incorrect Pair Slippage Update Logic in LibPairsManager	Business Logic	Resolved
PVE-003	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Resolved
PVE-004	Low	Improved Role Member Management in LibAccessControlEnumerable	Security Features	Resolved
PVE-005	Low	Incorrect Fee Total Accounting in LibFeeManager	Coding Practices	Resolved
PVE-006	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 Settlement Logic in TradingCloseFacet

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `TradingCloseFacet`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned earlier, `Apollox` is a decentralized derivatives trading platform that enables users to open/close trade spot and futures with low fees, deep liquidity and get rewards. While examining the trade-closing logic, we notice the current settlement is not as accurate as expected.

In the following, we show the code snippet from the related `TradingCloseFacet::_settleForCloseTrade()` helper routine. As the name indicates, this routine is designed to settle the trade when it is requested to close. The settlement has an invariant to maintain, i.e., `openTradeReceive + closeFee + userReceive + lpReceive` (line 65). It comes to our attention that in the last `else`-branch (line 77), the current `lpReceive` is assigned as `lpReceive = - openTradeReceive`. However, it forgot to reset `closeFee = 0`. As a result, the current invariant might not hold further.

```
61      function _settleForCloseTrade(
62          LibTrading.TradingStorage storage ts, ITrading.OpenTrade memory ot,
63          bytes32 tradeHash, int256 pnl, int256 fundingFee, uint256 closeFee
64      ) internal {
65          // openTradeReceive + closeFee + userReceive + lpReceive == 0
66          // closeFee >= 0 && userReceive >= 0
67          int256 openTradeReceive = - int256(uint256(ot.margin)) - fundingFee;
68          uint256 userReceive;
69          int256 lpReceive;
70          if (- openTradeReceive + pnl >= int256(closeFee)) {
71              userReceive = uint256(- openTradeReceive + pnl) - closeFee;
72              lpReceive = - pnl;
73          } else if (- openTradeReceive + pnl > 0 && - openTradeReceive + pnl < int256(
                closeFee)) {
```

```
74              closeFee = uint256(- openTradeReceive + pnl);
75              lpReceive = - pnl;
76          } else {
77              lpReceive = - openTradeReceive;
78          }
79
80          _settleAsset(ts, SettleAssetTuple(ot, tradeHash, openTradeReceive, closeFee,
                userReceive, lpReceive));
81      }
```

Listing 3.1: `TradingCloseFacet::_settleForCloseTrade()`

In addition, since the `closeFee` state may be changed within this helper routine, there is a need to propagate the resulting `closeFee` back to its caller `_closeTrade()`, which will include the `closeFee` as part of the `CloseInfo` in `OrderAndTradeHistory`.

Recommendation Revise the above routine to properly adjust `closeFee` in all possible execution paths and accordingly record the resulting `closeFee`. Note another routine `_settleForLiqTrade()` shares the same issue.

Status The issue has been fixed by this commit: `0e0fffb7`.

3.2 Incorrect Pair Slippage Update Logic in LibPairsManager

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `LibPairsManager`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Apollox` protocol has a library contract `LibPairsManager` to manage the set of pairs supported for spot and future trading. For each pair, the contract allows for the adjustment of associated `openFee` and `closeFee`. While reviewing the fee-adjusting logic, we notice the current implementation needs to be revised.

To elaborate, we show below the related `updatePairFee()` routine. This routine takes two input arguments: `base` and `feeConfigIndex`. The first argument indicates the pair to update while the second argument indicates the new `feeConfigIndex` to locate `openFee` and `closeFee`. For gas efficiency, the current logic properly locates the last element in `oldFeePairs` and replaces the old one. However, it does not reflect the `feePosition` member in the affected pair (pointed to by the last element in `oldFeePairs`). As a result, the affected pair may be using the wrong fee parameters to open and close trades. The same issue also occurs when the pair's `slippage` is adjusted.

```
328     function updatePairFee(address base, uint16 feeConfigIndex) internal {
329         PairsManagerStorage storage pms = pairsManagerStorage();
330         Pair storage pair = pms.pairs[base];
331         require(pair.base != address(0), "LibPairsManager: Pair does not exist");
332         (LibFeeManager.FeeConfig memory feeConfig, address[] storage feePairs) =
                LibFeeManager.getFeeConfigByIndex(feeConfigIndex);
333         require(feeConfig.enable, "LibPairsManager: Fee configuration is not available")
                ;

335         uint16 oldFeeConfigIndex = pair.feeConfigIndex;
336         (, address[] storage oldFeePairs) = LibFeeManager.getFeeConfigByIndex(
                oldFeeConfigIndex);
337         uint lastPositionFee = oldFeePairs.length - 1;
338         uint oldFeePosition = pair.feePosition;
339         if (oldFeePosition != lastPositionFee) {
340             oldFeePairs[oldFeePosition] = oldFeePairs[lastPositionFee];
341         }
342         oldFeePairs.pop();

344         pair.feeConfigIndex = feeConfigIndex;
345         pair.feePosition = uint16(feePairs.length);
346         feePairs.push(base);
347         emit UpdatePairFee(base, oldFeeConfigIndex, feeConfigIndex);
348     }
```

Listing 3.2: `LibPairsManager::updatePairFee()`

Recommendation Revise the above-mentioned routines to properly adjust the the pair's fee and slippage.

Status The issue has been fixed by this commit: `e56cc7a`.

3.3 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Time and State [8]
- CWE subcategory: CWE-663 [3]

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 [13] exploit, and the Uniswap/Lendf.Me hack [12].

We notice an occasion where the checks-effects-interactions principle is violated. Using the LibLimitOrder as an example, the cancelLimitOrder() function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above re-entrancy.

Apparently, the interaction with the external contract (line 81) starts before effecting the update on internal state (line 82), 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 very same cancelLimitOrder() function. Note that there may be no harm caused to current protocol. However, it is still suggested to follow the known checks-effects-interactions best practice.

```
75      function cancelLimitOrder(bytes32 orderHash) internal {
76          LimitOrderStorage storage los = limitOrderStorage();
77          ILimitOrder.LimitOrder storage order = los.limitOrders[orderHash];
78          check(order);
79
80          _cancelLimitOrder(orderHash, IOrderAndTradeHistory.ActionType.CANCEL_LIMIT);
81          IERC20(order.tokenIn).safeTransfer(order.user, order.amountIn);
82          _removeOrder(los, order, orderHash);
83          emit CancelLimitOrder(msg.sender, orderHash);
84      }
```

Listing 3.3: LibLimitOrder::cancelLimitOrder()

In the meantime, we should mention that the supported tokens in the protocol are expected to implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for re-entrancy.

Recommendation Apply necessary reentrancy prevention by following the checks-effects-interactions best practice. In addition, it is important to ensure the suggested tokens does not allow for hooks for callbacks. Note that the issue is also applicable to another routine, i.e., LibBrokerManager.withdrawCommission().

Status The issue has been fixed by this commit: a1388d7.

3.4 Improved Role Member Management in LibAccessControlEnumerable

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LibAccessControlEnumerable`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

Description

The facilitate role management, the `Apollox` protocol has a `LibAccessControlEnumerable` contract to keep track of the assigned roles and associated bearers. In the process of role management, we notice the current implementation can be improved.

To elaborate, we show below the related two functions: `grantRole()` and `revokeRole()`. The first function is used to grant a role to a user while the second function revokes the role from a user. In the meantime, it also keeps the list of members for a given role in the data structure `acs.roleMembers[role]`. It comes to our attention that the `acs.roleMembers[role]` data structure is always updated regardless whether the given account is already in the member set or not. For revision, we can move the `acs.roleMembers[role]`-updating logic inside the corresponding `if`-branch (line 60 and 69).

```
58      function grantRole(bytes32 role, address account) internal {
59          AccessControlStorage storage acs = accessControlStorage();
60          if (!hasRole(role, account)) {
61              acs.roles[role].members[account] = true;
62              emit RoleGranted(role, account, msg.sender);
63          }
64          acs.roleMembers[role].add(account);
65      }
66
67      function revokeRole(bytes32 role, address account) internal {
68          AccessControlStorage storage acs = accessControlStorage();
69          if (hasRole(role, account)) {
70              acs.roles[role].members[account] = false;
71              emit RoleRevoked(role, account, msg.sender);
72          }
73          acs.roleMembers[role].remove(account);
74      }
```

Listing 3.4: `LibAccessControlEnumerable::grantRole()/revokeRole()`

Recommendation Improve the above routines to properly update the `acs.roleMembers[role]` data structure for the given `role`.

Status This issue has been fixed in the commit: a8d72e4.

3.5 Incorrect Fee Total Accounting in LibFeeManager

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LibFeeManager`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [2]

Description

As mentioned earlier, each trade may be charged for fees, i.e., `openFee` and `closeFee`. While examining the related fee-accounting logic, we notice the current implementation can be improved.

In the following, we show the related `chargeOpenFee()` routine. As the name indicates, this routine is designed to charge the open fee for the given trade. The collected fee is updated in the `feeDetails` structure with three members: `total`, `daoAmount`, and `brokerAmount`. We notice the `total` member should be always updated no matter whether other members are updated. However, the current implementation will not update the `total` member unless the second `daoAmount` is updated. The same issue is also present in the `chargeCloseFee()` routine.

```
115     function chargeOpenFee(address token, uint256 feeAmount, uint24 broker) internal
            returns (uint24){
116         FeeManagerStorage storage fms = feeManagerStorage();
117         IFeeManager.FeeDetail storage detail = fms.feeDetails[token];
118
119         uint256 daoShare = feeAmount * fms.daoShareP / 1e4;
120         if (daoShare > 0) {
121             IERC20(token).safeTransfer(fms.daoRepurchase, daoShare);
122             detail.total += feeAmount;
123             detail.daoAmount += daoShare;
124         }
125         (uint256 commission, uint24 brokerId) = LibBrokerManager.updateBrokerCommission(
                token, feeAmount, broker);
126         detail.brokerAmount += commission;
127
128         uint256 lpAmount = feeAmount - daoShare - commission;
129         LibVault.deposit(token, lpAmount);
130         emit OpenFee(token, feeAmount, daoShare, brokerId, commission);
131         return brokerId;
132     }
```

Listing 3.5: `LibFeeManager::chargeOpenFee()`

Recommendation Properly keep track of the `total` member of `feeDetails` when a trade is opened, closed, or liquidated.

Status The issue has been fixed by this commit: `269fc96`.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

Description

In the `Apollox` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure various system parameters, assign other roles, as well as upgrade the proxy). In the following, we show the representative functions potentially affected by the privilege of the account.

```
15      function initFeeManagerFacet(address daoRepurchase, uint16 daoShareP) external {
16          LibAccessControlEnumerable.checkRole(Constants.DEPLOYER_ROLE);
17          require(daoRepurchase != address(0), "FeeManagerFacet: daoRepurchase cannot be 0
                address");
18          LibFeeManager.initialize(daoRepurchase, daoShareP);
19      }
20
21      function addFeeConfig(uint16 index, string calldata name, uint16 openFeeP, uint16
            closeFeeP) external override {
22          LibAccessControlEnumerable.checkRole(Constants.PAIR_OPERATOR_ROLE);
23          require(openFeeP < 1e4 && closeFeeP < 1e4, "FeeManagerFacet: Invalid parameters"
                );
24          LibFeeManager.addFeeConfig(index, name, openFeeP, closeFeeP);
25      }
26
27      function removeFeeConfig(uint16 index) external override {
28          LibAccessControlEnumerable.checkRole(Constants.PAIR_OPERATOR_ROLE);
29          LibFeeManager.removeFeeConfig(index);
30      }
31
32      function updateFeeConfig(uint16 index, uint16 openFeeP, uint16 closeFeeP) external
            override {
33          LibAccessControlEnumerable.checkRole(Constants.PAIR_OPERATOR_ROLE);
34          require(openFeeP < 1e4 && closeFeeP < 1e4, "FeeManagerFacet: Invalid parameters"
                );
35          LibFeeManager.updateFeeConfig(index, openFeeP, closeFeeP);
36      }
```

Listing 3.6: Example Privileged Operations in `FeeManagerFacet`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it would be worrisome if the privileged account is not governed by a `DAO`-like

structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

Moreover, it should be noted that current contracts are to be deployed behind a proxy with the typical `Diamond` implementation. And naturally, there is a need to properly manage the admin privileges as they are capable of upgrading the entire protocol implementation.

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 The issue has been confirmed by the team. The team intends to introduce multi-sig (with multiple role separation) and `timelock` mechanisms to mitigate this issue.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the Apollox protocol, which is a decentralized derivatives trading platform. The main architecture of Apollox V2 is the fully on-chain liquidity model for more transparent, low slippage trades. Users do not need to register, deposit or withdraw funds. All V2 trades are executed against the ALP pool on BNB Smart Chain and liquidity for all V2 trading pairs is shared via the ALP pool to maximize capital efficiency. Real time price feeds will be taken from both Binance Oracle and Chainlink to ensure the most accurate pricing. The current code base is well organized and those identified issues are promptly confirmed and fixed.

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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[3] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

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

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

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

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

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

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

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

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