SMART CONTRACT AUDIT REPORT

for

QILIN FINANCE

Prepared By: Yiqun Chen

PeckShield

June 20, 2021

Document Properties

Client	QiLin Finance
Title	Smart Contract Audit Report
Target	QiLin Finance
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 20, 2021	Xuxian Jiang	Final Release
0.3	June 19, 2021	Xuxian Jiang	Add More Findings #2
0.2	June 16, 2021	Xuxian Jiang	Add More Findings #1
0.1	June 12, 2021	Xuxian Jiang	Initial Draft

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **QiLin Finance** design document and related smart contract source code, 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 QiLin Finance

The `QiLin Finance` is a decentralized risk optimizer protocol for crypto derivatives trading on the `Ethereum` blockchain. The protocol contains a number of innovative core features, including an elastic model for liquidity pools, the rebase funding rate mechanism (that greatly reduces the risk of open positions for liquidity during market volatilities), and the dynamic algorithmic slippage mechanism (that incentivizes against position imbalance). The audited system allows for futures position trading such that traders can purchase long and short positions at leverage with guaranteed liquidity.

The basic information of QiLin Finance is as follows:

Table 1.1: Basic Information of QiLin Finance

Item	Description
Issuer	QiLin Finance
Website	https://qilin.fi/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 20, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that QiLin Finance assumes a trusted price oracle with timely market price feeds for supported assets and the oracle itself is not part of this audit.

- https://github.com/CodexDao/QiLin-dev.git (36b732b)

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

- https://github.com/CodexDao/QiLin-dev.git (992fdf5)

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 the 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 Audit Checklist

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

To evaluate the risk, we go through a checklist of 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 Logic	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 implementation of the QiLin Finance protocol. 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	1	■
Informational	1	■
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 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, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key QiLin Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Improved Corner Case Handling In alertBankruptedLiquidation()	Business Logic	Fixed
PVE-002	Low	Improved Sanity Checks For System Parameters	Coding Practices	Confirmed
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Resolved

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 Corner Case Handling In alertBankruptedLiquidation()

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Liquidation`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

At the core of the `QiLin Finance` protocol is the `Depot` contract that manages the positions of all trading users. The trading user can open a new leveraged position and add collateral into an opened position. If the collateral is insufficient to cover an opened position, the position will be considered underwater and may be liquidated. While examining the liquidation-validating logic, we notice the current `alertBankruptedLiquidation()` function can be improved.

To elaborate, we show below the `alertBankruptedLiquidation()` function. This function is designed to raise a signal whether the given `positionID` should be consider underwater and may be liquidated. The logic properly calculates various metrics, i.e., `serviceFee`, `marginLoss`, and `isProfit`. However, when `isProfit` is evaluated to be `true`, the return boolean value needs to computed as `margin.add(value)< serviceFee.add(marginLoss)`, not current `margin.add(value)<= serviceFee.add(marginLoss)` (line 214).

```
194     function alertBankruptedLiquidation(uint32 positionId) external override view
            returns (bool) {
195         IDepot depot = getDepot();

197         (
198             address account,
199             uint share,
200             uint leveragedPosition,
201             uint openPositionPrice,
```

```
            uint32 currencyKeyIdx,
            uint8 direction,
            uint margin,
        ) = depot.position(positionId);

        if (account != address(0)) {
            uint serviceFee = leveragedPosition.mul(systemSetting().positionClosingFee()
                ) / 1e18;
            uint marginLoss = depot.calMarginLoss(leveragedPosition, share, direction);

            (bool isProfit, uint value) = depot.calNetProfit(currencyKeyIdx,
                leveragedPosition, openPositionPrice, direction);

            if (isProfit) {
                return margin.add(value) <= serviceFee.add(marginLoss);
            } else {
                return margin < value.add(serviceFee).add(marginLoss);
            }
        }

        return false;
    }
```

Listing 3.1: `Liquidation::alertBankruptedLiquidation()`

Recommendation Properly handle the corner case in `alertBankruptedLiquidation()` to be consistent with other related functions, i.e., `bankruptedLiquidate()`.

Status This issue has been fixed in this commit: `2650c7e`.

3.2 Improved Sanity Checks For System Parameters

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SystemSetting`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `QiLin` protocol is no exception. Specifically, if we examine the `SystemSetting` contract, it has defined a number of protocol-wide risk parameters, e.g., `_minInitialMargin` and `_marginRatio`. In the following, we show the corresponding routines that allow for their changes.

```
    function minInitialMargin() external override view returns (uint256) {
        return _minInitialMargin;
```

```
    }

    function minAddDeposit() external override view returns (uint256) {
        return _minAddDeposit;
    }

    function minHoldingPeriod() external override view returns (uint) {
        return _minHoldingPeriod;
    }

    function marginRatio() external override view returns (uint256) {
        return _marginRatio;
    }

    function positionClosingFee() external override view returns (uint256) {
        return _positionClosingFee;
    }
```

Listing 3.2: A Number of Setters in SystemSetting

Our result shows the update logic on these fee parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of a large _positionClosingFee parameter (say more than 100%) will revert the liquidate() operation.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. Also, consider emitting related events for external monitoring and analytics tools.

Status This issue has been confirmed.

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

In the QiLin Finance protocol, the privileged owner account plays a critical role in governing and regulating the system-wide operations. It also has the privilege to control or govern the flow of assets among various components.

To elaborate, we show below the `Exchange::openPosition()` routine. This routine will query for current `Depot` contract and invoke its `newPosition()`. The same occurs to the `closePosition()` functionality. Meanwhile, the `Depot` contract is managed by the `AddressResolver` contract. Note the owner of the `AddressResolver` contract has the privilege to update or modify the current `Depot` contract.

```
    function openPosition(bytes32 currencyKey, uint8 direction, uint16 level, uint position) external override returns (uint32) {
        systemSetting().checkOpenPosition(position, level);

        require(direction == 1 || direction == 2, "Direction Only Can Be 1 Or 2");

        (uint32 currencyKeyIdx, uint openPrice) = exchangeRates().rateForCurrency(currencyKey);
        uint32 index = getDepot().newPosition(msg.sender, openPrice, position, currencyKeyIdx, level, direction);

        emit OpenPosition(msg.sender, index, openPrice, currencyKey, direction, level, position);

        return index;
    }
```

Listing 3.3: `Exchange::openPosition()`

```
contract AddressResolver is Ownable {
    mapping(bytes32 => address) public repository;

    function importAddresses(bytes32[] calldata names, address[] calldata destinations) external onlyOwner {
        require(names.length == destinations.length, "Input lengths must match");

        for (uint i = 0; i < names.length; i++) {
            repository[names[i]] = destinations[i];
        }
    }

    function requireAndGetAddress(bytes32 name, string memory reason) internal view returns (address) {
        address _foundAddress = repository[name];
        require(_foundAddress != address(0), reason);
        return _foundAddress;
    }
}
```

Listing 3.4: The `AddressResolver` Contract

We emphasize that current privilege assignment is necessary and required for proper protocol operation. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure. The discussion with the team has confirmed that the `owner` will be managed by a multi-sig account.

We point out that a compromised `owner` account is capable of modifying current protocol configuaration with adverse consequences, including permanent lock-down of user funds.

Recommendation Promptly transfer the `owner` privilege to the intended `DAO`-like governance contract.

Status This issue has been confirmed and partially mitigated in the deployment script by setting the `Depot`'s `owner` to the contract address.

4 Conclusion

In this audit, we have analyzed the QiLin Finance design and implementation. The system presents a unique, robust offering as a decentralized risk optimizer protocol for crypto derivatives trading. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

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

References

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

[2] MITRE. CWE-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.