# SMART CONTRACT AUDIT REPORT

for

# QILIN FINANCE

Prepared By: Shuxiao Wang

PeckShield

April 1, 2021

## Document Properties

| | |
|---|---|
| Client | QiLin Finance |
| Title | Smart Contract Audit Report |
| Target | QiLin Finance |
| Version | 1.0 |
| Author | Xuxian Jiang |
| Auditors | Xuxian Jiang, Huaguo Shi, Jeff Liu |
| Reviewed by | Jeff Liu |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | April 1, 2021 | Xuxian Jiang | Final Release |
| 1.0-rc1 | April 1, 2021 | Xuxian Jiang | Release Candidate #1 |
| 0.3 | March 28, 2021 | Xuxian Jiang | Add More Findings #2 |
| 0.2 | March 26, 2021 | Xuxian Jiang | Add More Findings #1 |
| 0.1 | March 22, 2021 | Xuxian Jiang | Initial Draft |

## Contact

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



| | |
|---|---|
| Name | Shuxiao Wang |
| Phone | +86 173 6454 5338 |
| 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 | April 1, 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 (ebf65e7)

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 (f464a34)

## 1.2 About PeckShield

PeckShield Inc. [12] 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 [11]:

- 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) [10], 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 | 1 | ■ |
| Medium | 2 | ■ ■ |
| Low | 3 | ■ ■ ■ |
| Informational | 1 | ■ |
| Total | 8 | |

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 high-severity vulnerability, 2 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key QiLin Finance Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | High | Improved Business Logic In addDeposit() | Business Logic | Fixed |
| PVE-002 | Low | Accommodation of ERC20-Non-Compliant BaseCurrency | Business Logic | Fixed |
| PVE-003 | Informational | Inconsistent Handling in deleteCurrencyKey() | Coding Practices | Fixed |
| PVE-004 | Low | Improved Sanity Checks For System Parameters | Coding Practices | Fixed |
| PVE-005 | Low | Potential Reentrancy Risk In initialFunding() And fundLiquidity() | Time And State | Fixed |
| PVE-006 | Medium | Trust Issue of Admin Keys | Security Features | Mitigated |
| PVE-007 | Medium | Inappropriate Business Logic in Depot::closePosition() | Business Logic Fixed | Fixed |

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.

PeckShield

# 3 | Detailed Results

## 3.1 Improved Business Logic In addDeposit()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `Depot`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

### 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 collateral-adding logic in `Depot`, we notice a flawed handling that may be exploited to drain the pool funds.

To elaborate, we show below the `addDeposit()` function. This function is designed to add margin into a specified position. The logic is rather straightforward in transferring funds from the trading user to the `Depot` contract and updating internal bookkeeping states.

```
    function addDeposit(
        address account,
        uint32 positionId,
        uint margin) external override onlyPower {
        Position memory p = _positions[positionId];

        require(account == p.account, "Position Not Match");

        IERC20 baseCurrencyContract = baseCurrency();

        require(
            baseCurrencyContract.allowance(account, address(this)) >= margin,
            "BaseCurrency Approved To Exchange Is Not Enough");
        baseCurrencyContract.transfer(account, margin);
```

```
        _positions[positionId].margin = p.margin.add(margin);
        if (p.direction == 1) {
            _totalMarginLong = _totalMarginLong.add(margin);
        } else {
            _totalMarginShort = _totalMarginShort.add(margin);
        }
    }
```

Listing 3.1: Depot::addDeposit()

However, our analysis with the above function shows that the funds are not transferred from the trading user to `Depot`, but from `Depot` to the trading user! This is certainly not the intended design. Otherwise, the funds may be simply drained by adding collateral into a position.

**Recommendation** Properly revise the `addDeposit()` logic to transfer funds from trading users into `Depot`, not the other way around.

**Status** This issue has been fixed in this commit: `2c82188`.

## 3.2 Accommodation of ERC20-Non-Compliant BaseCurrency

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Depot`
- Category: Business Logic [8]
- CWE subcategory: N/A

### 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 this section, we examine the `transferFrom()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. Specifically, the `transferFrom()` routine does not have a return value defined and implemented. However, the `IERC20` interface has defined the `transferFrom()` interface with a `bool` return value. As a result, the call to `transferFrom()` may expect a return value. With the lack of return value of `USDT`'s `transferFrom()`, the call will be unfortunately reverted.

```
    function transferFrom(address _from, address _to, uint _value) public
        onlyPayloadSize(3 * 32) {
        var _allowance = allowed[_from][msg.sender];

        // Check is not needed because sub(_allowance, _value) will already throw if
            this condition is not met
        // if (_value > _allowance) throw;
```

```
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        if (_allowance < MAX_UINT) {
            allowed[_from][msg.sender] = _allowance.sub(_value);
        }
        uint sendAmount = _value.sub(fee);
        balances[_from] = balances[_from].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(_from, owner, fee);
        }
        Transfer(_from, _to, sendAmount);
    }
```

Listing 3.2: USDT Token **Contract**

Because of that, a normal call to `transferFrom()` is suggested to use the safe version, i.e., `safeTransferFrom()`, 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()/transfer()` as well, i.e., `safeApprove()/safeTransfer()`.

In current implementation, if we examine the `Depot::newPosition()` routine that is designed to open a position by pulling intended funds from the trading user. To accommodate the specific idiosyncrasy, there is a need to user `safeTransferFrom()`, instead of `transferFrom()` (line 117).

```
108    function newPosition(
109        address account,
110        uint openPositionPrice,
111        uint margin,
112        uint32 currencyKeyIdx,
113        uint16 level,
114        uint8 direction) external override onlyPower returns (uint32) {
115        require(_initialFundingCompleted, 'Initial Funding Has Not Completed');

117        baseCurrency().transferFrom(account, address(this), margin);

119        uint leveragedPosition = margin.mul(level);
120        uint share = leveragedPosition.mul(1e18) / _netValue(direction);

122        _positionIndex++;
123        _positions[_positionIndex] = Position(
124            share,
125            openPositionPrice,
126            leveragedPosition,
127            margin,
```

```
            account,
            currencyKeyIdx,
            direction);

        if (direction == 1) {
            _totalMarginLong = _totalMarginLong.add(margin);
            _totalLeveragedPositionsLong = _totalLeveragedPositionsLong.add(
                leveragedPosition);
            _totalShareLong = _totalShareLong.add(share);
        } else {
            _totalMarginShort = _totalMarginShort.add(margin);
            _totalLeveragedPositionsShort = _totalLeveragedPositionsShort.add(
                leveragedPosition);
            _totalShareShort = _totalShareShort.add(share);
        }

        return _positionIndex;
    }
```

Listing 3.3: Depot::newPosition()

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

**Status** This issue has been fixed in this commit: `475f460`.

## 3.3 Inconsistent Handling in deleteCurrencyKey()

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `ExchangeRates`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [1]

### Description

In the `QiLin` protocol, there is an `ExchangeRates` contract that maintains critical states about `key`-based currencies as well as their currency rates. In the following, we elaborate three related `key` operations: `addCurrencyKey()`, `updateCurrencyKey()`, and `deleteCurrencyKey()`.

```
    function addCurrencyKey(bytes32 currencyKey_, address aggregator_) external override
        onlyOwner {
        require(address(_aggregators[currencyKey_]) == address(0), "aggregator already
            exist");
        AggregatorV2V3Interface aggregator = AggregatorV2V3Interface(aggregator_);

        require(aggregator.latestRound() >= 0, "Given Aggregator is invalid");
```

```

        _aggregators[currencyKey_] = aggregator_;
        _indexNext++;

        uint32 idx = _indexNext;
        _Indexs2Keys[idx] = currencyKey_;
        _key2Indexs[currencyKey_] = idx;
        _keyAddressIndexs[idx] = aggregator_;
    }

    function updateCurrencyKey(bytes32 currencyKey_, address aggregator_) external
        override onlyOwner {
        address oldAggregator = address(_aggregators[currencyKey_]);
        require(oldAggregator != address(0), "aggregator does not exist");
        AggregatorV2V3Interface aggregator = AggregatorV2V3Interface(aggregator_);
        require(aggregator.latestRound() >= 0, "Given Aggregator is invalid");
        _aggregators[currencyKey_] = aggregator_;
    }

    function deleteCurrencyKey(bytes32 currencyKey_) external override onlyOwner {
        delete _aggregators[currencyKey_];
    }
```

Listing 3.4: Related Key Operations In ExchangeRates

While analyzing the above three routines, we notice that `addCurrencyKey()` properly maintains the mapping between the given `currency key` and the associated index. The `updateCurrencyKey()` routine, as the name indicates, update the given `currency key`. However, the `deleteCurrencyKey()` routine simply deletes the `currency key`-related storage slot, but without changing the mapping with its index.

**Recommendation** When a `currency key` is removed, properly adjust all associated mappings and state variables.

**Status** This issue has been fixed in this commit: `6bade327`.

## 3.4 Improved Sanity Checks For System Parameters

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: SystemSetting
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [2]

### 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 setConstantMarginRatio(uint256 constantMarginRatio_) external onlyOwner {
        _constantMarginRatio = constantMarginRatio_;
    }

    function setMinInitialMargin(uint256 minInitialMargin_) external onlyOwner {
        _minInitialMargin = minInitialMargin_;
    }

    function setMinAddDeposit(uint minAddDeposit_) external onlyOwner {
        _minAddDeposit = minAddDeposit_;
    }

    function setMinHoldingPeriod(uint minHoldingPeriod_) external onlyOwner {
        _minHoldingPeriod = minHoldingPeriod_;
    }

    function setMarginRatio(uint256 marginRatio_) external onlyOwner {
        _marginRatio = marginRatio_;
    }

    function setPositionClosingFee(uint256 positionClosingFee_) external onlyOwner {
        _positionClosingFee = positionClosingFee_;
    }

    function setLiquidationFee(uint256 liquidationFee_) external onlyOwner {
        _liquidationFee = liquidationFee_;
    }
```

Listing 3.5: 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 fixed in this commit: `4a3a2f6`.

## 3.5 Potential Reentrancy Risk In initialFunding() And fundLiquidity()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Fluidity`, `sfluidity`
- Category: Time and State [9]
- CWE subcategory: CWE-663 [4]

### 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` [14] exploit, and the recent `Uniswap/Lendf.Me` hack [13].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `sfluidity` as an example, the `claim()` 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 177) starts before effecting the update on internal states (lines 184−188), 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 `claim()` function.

```
163     function claim() public {
164         address msgSender = msg.sender;
165         userDeposit memory userInfoData = _userInfo[msgSender];
166
```

```
        require(userInfoData.amount > 0, "deposit is not exist");

        _settlePool();

        uint256 userShare = userInfoData.amount.mul(
            block.number.sub(userInfoData.last_block_num)
        );

        _total_deposit = _total_deposit.sub(userInfoData.amount);

        _poolToken.transfer(msgSender, userInfoData.amount);

        _transferToUserByShare(
            msgSender,
            userInfoData.un_settle_share.add(userShare)
        );

        userInfoData.un_settle_share = 0;
        userInfoData.amount = 0;
        userInfoData.last_block_num = block.number;

        _userInfo[msgSender] = userInfoData;
    }
```

Listing 3.6: sfluidity ::claim()

Note that two other routines `initialFunding()` and `fundLiquidity()` share the same issue. In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`.

**Recommendation** Apply necessary reentrancy prevention by making use of the common `nonReentrant` modifier.

**Status** This issue has been fixed in this commit: `3bfbf16`.

## 3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

### 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);

        //depot.transferIn(msg.sender, position); in newTosition
        emit OpenPosition(msg.sender, index, openPrice, currencyKey, direction, level,
            position);

        return index;
    }
```

Listing 3.7: 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 3.8: 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 with a multi-sig account to regulate the governance/controller privileges.

## 3.7 Inappropriate Business Logic in Depot::closePosition()

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Depot`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

### Description

As mentioned in Section 3.1, at the core of the `QiLin Finance` protocol is the `Depot` contract that manages the positions of all trading users. A trading user can open a new leveraged position and add collateral into an opened position. In the following, we examine the position-closing logic and report a finding on possible discrepancy.

To elaborate, we show below the implementation of the `closePosition()` function. As the name indicates, this routine is designed to close an open position. The close operation requires certain bookkeeping on internal states. For example during `direction == 1`, there is a need to update `_liquidityPool`, `_totalMarginLong`, `_totalLeveragedPositionsLong`, and `_totalShareLong` (lines 183 – 187). Otherwise, when `direction != 1`, there is a need to update `_liquidityPool`, `_totalMarginShort`, `_totalLeveragedPositionsShort`, and `_totalShareShort` (lines 189 – 192).

```
    function closePosition(
        address account,
        uint32 positionId,
        uint8 direction,
        bool isProfit,
        uint margin,
        uint share,
        uint value,
        uint marginLoss,
        uint fee) external override onlyPower {
        uint transferOutValue = isProfit.addOrSub(margin, value).sub(fee).sub(marginLoss);
        if ( isProfit && (_liquidityPool.add(fee) <= value) ){
            transferOutValue = _liquidityPool;
        }
        baseCurrency().transfer(account, transferOutValue);

        uint liquidityPoolVal = (!isProfit).addOrSub2Zero(_liquidityPool.add(fee), value);
        uint detaLeveraged = share.mul(_netValue(direction)) / 1e18;

        if (direction == 1) {
            _liquidityPool = liquidityPoolVal;
            _totalMarginLong = _totalMarginLong.sub(margin.sub2Zero(marginLoss));
            _totalLeveragedPositionsLong = _totalLeveragedPositionsLong.sub(detaLeveraged);
            _totalShareLong = _totalShareLong.sub(share);
        } else {
            _liquidityPool = liquidityPoolVal;
            _totalMarginShort = _totalMarginShort.sub(margin.sub2Zero(marginLoss));
            _totalLeveragedPositionsShort = _totalLeveragedPositionsShort.sub(detaLeveraged);
            _totalShareShort = _totalShareShort.sub(share);
        }

        delete _positions[positionId];
    }
```

Listing 3.9: Depot::closePosition()

We notice the change on the `_totalMarginLong` or `_totalMarginShort` needs to be revisited. Using `_totalMarginLong` as an example, it is updated as follows: `_totalMarginLong = _totalMarginLong.sub(margin.sub2Zero(marginLoss))`. However, there is a corner case that may occur, i.e., `margin < marginLoss`. If this corner case happens, the `_totalMarginLong` state should be updated as follows: `_totalMarginLong = _totalMarginLong.add(marginLoss).sub(margin)`. The same revision should be applicable to `_totalMarginShort` as well.

**Recommendation** Revise the above calculations to properly update `_totalMarginLong` or `_totalMarginShort` by covering all possible corner cases.

**Status** This issue has been fixed in this commit: `1f67a60`.

PeckShield

# 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.

PeckShield

# References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

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

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

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

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

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

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

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

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

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

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

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

PeckShield