SMART CONTRACT AUDIT REPORT

for

QiLin V2

Prepared By: Patrick Lou

PeckShield

April 13, 2022

Document Properties

Client	QiLin Finance
Title	Smart Contract Audit Report
Target	QiLin V2
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 13, 2022	Xiaotao Wu	Final Release
1.0-rc	April 11, 2022	Xiaotao Wu	Release Candidate

Contact

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

Name	Patrick Lou
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 | Introduction

Given the opportunity to review the `QiLin V2` 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 V2

The `QiLin Finance` is a decentralized risk optimizer protocol for crypto derivatives trading on the `Ethereum` blockchain. Based on the user feedback and development goal, the `QiLin V2` introduces some new features to support permissionless creation and trading for both linear and inverse perpetual contract pairs for any `ERC-20` assets, `Uniswap-V3` oracle price feeding, real-time liquidation, `open-Price` slippage mechanism to further reduce the risk of liquidity providers, and stateless system with much lower gas fee.

The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of QiLin V2

Item	Description
Name	QiLin Finance
Website	https://qilin.fi/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 13, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/CodexDao/qilin-v2.git (f7d2fa6)

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.

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

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

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 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 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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `QiLin V2` 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	1	■
Low	1	■
Informational	1	■
Undetermined	1	■
Total	5	

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, 1 medium-severity vulnerability, 1 low-severity vulnerability, 1 informational recommendation, and 1 undetermined issue.

Table 2.1: Key QiLin V2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Potential Reentrancy Risk In Pool::addLiquidity()	Time and State	Resolved
PVE-002	Undetermined	Possible Price manipulation For Rates::_getPriceV3()/_getPriceV2()	Time and State	Mitigated
PVE-003	Informational	Meaningful Events For Important State Changes	Coding Practices	Resolved
PVE-004	Low	Improved Sanity Checks For System Parameters	Coding Practices	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

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 Potential Reentrancy Risk In Pool::addLiquidity()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `Pool`
- Category: Time and State [9]
- CWE subcategory: CWE-682 [5]

Description

To facilitate the token transfer from users to the protocol, the `Pool` contract provides a helper routine `poolCallback()`. This routine is developed to transfer user funds into this contract via the `Router` contract and will be called in the `addLiquidity()/openPosition()` functions.

To elaborate, we show below the `poolCallback()` routine implementation. This routine allows the `msg.sender` to provide an arbitrary `poolV2Callback()` routine that is used directly in `IPoolCallback(msg.sender).poolV2Callback()` (line 75). However, if the `msg.sender` is a malicious contract, the arbitrary `poolV2Callback()` routine provided by the `msg.sender` can be exploited to reenter the `Pool` contract in a nested manner. Specifically, it first calls the `addLiquidity()` function in the vulnerable contract, but before the first instance of the function call is finished, a second call can be arranged to re-enter the vulnerable contract by invoking the `addLiquidity()` function that should only be executed once. By doing so, the require statement (lines 82-86) can be bypassed for the first execution of the `poolCallback()` routine.

```
73     function poolCallback(address user, uint256 amount) internal {
74         uint256 balanceBefore = IERC20(_poolToken).balanceOf(address(this));
75         IPoolCallback(msg.sender).poolV2Callback(
76             amount,
77             _poolToken,
78             address(_oraclePool),
79             user,
80             _reverse
81         );
```

```
        require(
            IERC20(_poolToken).balanceOf(address(this)) >=
                balanceBefore.add(amount),
            "poolToken is not enough"
        );
    }
```

Listing 3.1: `Pool::poolCallback()`

Note the `openPosition()` routine in the same contract shares the same issue.

Recommendation Add necessary reentrancy guards to prevent unwanted reentrancy risks or only allows the `Router` contract to call the `addLiquidity()/openPosition()` functions.

Status The issue has been fixed by this commit: `5e02bd8`.

3.2 Possible Price manipulation For Rates::_getPriceV3()/_getPriceV2()

- ID: PVE-002
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `Rates`
- Category: Time and State [7]
- CWE subcategory: CWE-362 [3]

Description

The `Rates` contract defines two functions (i.e., `_getPriceV2()` and `_getPriceV3()`) to obtain the current price of the subject matter from `UniswapV3Pool` or `UniswapV2Pair`. During the analysis of these two functions, we notice the price of the subject matter is possible to be manipulated. In the following, we use the `_getPriceV3()` routine as an example.

To elaborate, we show below the related code snippet of the `Rates` contract. Specifically, if we examine the implementation of the `_getPriceV3()`, the `sqrtPriceX96` of the subject matter is derived from `TickMath.getSqrtRatioAtTick(int24(tickCumulatives[1] - tickCumulatives[0])/ int24(OBSERVE_TIME_INTERVAL))` (lines 118-120), where `tickCumulatives[0]` and `tickCumulatives[1]` are the last two seconds tick cumulatives obtained from `UniswapV3Pool`. Because the current observation time for `UniswapV3` oracle is too short and the tick cumulative in the `UniswapV3Pool` can be affected by large trade, the final price of the subject matter may not be trustworthy.

```
113    function _getPriceV3() internal view returns (uint256) {
114        (int56[] memory tickCumulatives, ) = IUniswapV3Pool(_oraclePool).observe(
115            _secondsAgo
116        );
```

```
        uint256 sqrtPriceX96 = uint256(
            TickMath.getSqrtRatioAtTick(
                int24(tickCumulatives[1] - tickCumulatives[0]) /
                    int24(OBSERVE_TIME_INTERVAL)
            )
        );
        uint256 price;
        ...
    }
```

Listing 3.2: Rates::_getPriceV3()

Recommendation Revise current execution logic of _getPriceV3()/_getPriceV2() to defensively detect any manipulation attempts in the subject matter prices.

Status This issue has been confirmed. Below is the feedback from the QiLin team:

We consider that _getPriceV2() and _getPriceV3() in Rates contract can not be attacked by flashloan. Possible price manipulation alleged in the audit report will only appear when large funds flow into the market and the oracle's liquidity is insufficient, and attackers will take huge risks. So we consider that this is a market risk, not a systematic vulnerability of the QiLin contract.

3.3 Meaningful Events For Important State Changes

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: SystemSettings
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [4]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the SystemSettings contract as an example. While examining the events that reflect the SystemSettings dynamics, we notice there is a lack of emitting related events to reflect important state changes. Specifically, when the addLeverage()/deleteLeverage() are being called, there are no corresponding events being emitted to reflect the occurrence of addLeverage()/deleteLeverage().

```
    function addLeverage(uint32 leverage_) external onlyOwner {
        _leverages[leverage_] = true;
    }

    function deleteLeverage(uint32 leverage_) external onlyOwner {
        _leverages[leverage_] = false;
    }
```

Listing 3.3: `SystemSettings::addLeverage()/deleteLeverage()`

Recommendation Properly emit the related event when the above-mentioned functions are being invoked.

Status The issue has been fixed by this commit: `5e02bd8`.

3.4 Improved Sanity Checks For System Parameters

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SystemSettings`
- Category: Coding Practices [8]
- 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 V2` protocol is no exception. Specifically, if we examine the `SystemSettings` contract, it has defined many system-wide risk parameters for the `Pool` contract. In the following, we use the `setProtocolFee()` routine as an example.

```
    function setProtocolFee(uint256 protocolFee_) external onlyOwner {
        _protocolFee = protocolFee_;
        emit SetSystemParam(systemParam.ProtocolFee, protocolFee_);
    }
```

Listing 3.4: `SystemSettings::setProtocolFee()`

Specifically, the `_protocolFee` parameter defines the amount of `ls` token minted for the `_official` and there is a need to exercise extra care when configuring or updating it. Our analysis shows the update logic on this parameter 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 `_protocolFee` may set a huge protocol fee, resulting in the `QiLin V2` users to suffer asset losses when calling the `removeLiquidity()` function. Note the similar issue also exists in other setter functions of the same contract.

Recommendation Validate any change regarding this system-wide parameter to ensure it fall in an appropriate range.

Status The issue has been fixed by this commit: `5e02bd8`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `SystemSettings`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the `QiLin V2` protocol, there are two privileged account, i.e., `owner and suspender`. These accounts play a critical role in governing and regulating the system-wide operations (e.g., suspend/resume the pool, set the owner for a pool, and set the key parameters for the `QiLin V2` protocol, etc.). Our analysis shows that these privileged accounts need to be scrutinized. In the following, we use the `SystemSettings` contract as an example and show the representative functions potentially affected by the privileges of the `owner/suspender` accounts.

```
    function resumeSystem() external override onlySuspender {
        _active = true;
        emit Resume(msg.sender);
    }

    function suspendSystem() external override onlySuspender {
        _active = false;
        emit Suspend(msg.sender);
    }
```

Listing 3.5: `SystemSettings::resumeSystem()/suspendSystem()`

```
    function setProtocolFee(uint256 protocolFee_) external onlyOwner {
        _protocolFee = protocolFee_;
        emit SetSystemParam(systemParam.ProtocolFee, protocolFee_);
    }

    function setLiqProtocolFee(uint256 liqProtocolFee_) external onlyOwner {
        _liqProtocolFee = liqProtocolFee_;
        emit SetSystemParam(systemParam.LiqProtocolFee, liqProtocolFee_);
    }

    ...
```

Listing 3.6: `SystemSettings::setProtocolFee()/setLiqProtocolFee()`

```
    function setPoolOwner(address pool, address newOwner) external onlyOwner {
        if (_poolSettings[pool].owner != address(0)) {
            _poolSettings[pool].owner = newOwner;
        } else {
            _poolSettings[pool] = PoolSetting(
                newOwner,
                _marginRatio,
                _closingFee,
                _liqFeeBase,
                _liqFeeMax,
                _liqFeeCoefficient,
                _liqLsRequire,
                _rebaseCoefficient,
                _imbalanceThreshold,
                _priceDeviationCoefficient,
                _minHoldingPeriod,
                _debtStart,
                _debtAll,
                _minDebtRepay,
                _maxDebtRepay,
                _interestRate,
                _liquidityCoefficient,
                _deviation
            );
            _debtSettings[IPool(pool).debtToken()] = _interestRate;
        }

        emit SetPoolOwner(pool, newOwner);
    }
```

Listing 3.7: `SystemSettings::setPoolOwner()`

If the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the protocol users. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed `DAO`. In the meantime, a timelock-based mechanism can also be considered as mitigation. Moreover, it should be noted if current contracts are to be deployed behind a proxy, there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the QiLin V2 design and implementation. Based on the user feedback and development goal, the QiLin V2 introduces some new features to support permissionless creation and trading for both linear and inverse perpetual contract pairs for any ERC-20 assets, Uniswap-V3 oracle price feeding, real-time liquidation, open-Price slippage mechanism to further reduce the risk of liquidity providers, and stateless system with much lower gas fee. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, 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-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

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

[5] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

[7] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

[9] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.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.