SMART CONTRACT AUDIT REPORT

for

GinFinance

Prepared By: Xiaomi Huang

PeckShield

May 5, 2022

Document Properties

Client	GinFinance
Title	Smart Contract Audit Report
Target	GinFinance
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 5, 2022	Luck Hu	Final Release
1.0-rc	May 4, 2022	Luck Hu	Release Candidate

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 GinFinance 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 Implicit Assumption Enforcement In addLiquidity() 11
 3.2 Fork-Resistant Domain Separator In GinFinanceERC20 13
 3.3 Trust Issue Of Admin Keys 14
 3.4 Suggested Immutable Usages For Gas Efficiency 15

4 Conclusion 17

References 18

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `GinFinance` 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 is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

1.1 About GinFinance

`GinFinance` is an open-source protocol which aims to be the ultimate `DeFi` solution for asset liquidity and exchange on `BOBA` Network. With its efficient swap and staking model, it offers a wide range of decentralized applications to `DeFi` users at convenience. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of GinFinance

Item	Description
Name	GinFinance
Website	https://gin.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 5, 2022

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

- https://github.com/ginfidev/GinFinance-Core.git (eaa4a2c)

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

- https://github.com/ginfidev/GinFinance-Core.git (cf7e778)

1.2 About PeckShield

PeckShield Inc. [10] 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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [9]:

- 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) [8], 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 `GinFinance` smart contracts. 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	3	■ ■ ■
Informational	0	
Total	4	

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 and 3 low-severity vulnerabilities

Table 2.1: Key GinFinance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Implicit Assumption Enforcement In addLiquidity()	Coding Practices	Fixed
PVE-002	Low	Fork-Resistant Domain Separator In GinFinanceERC20	Business Logic	Fixed
PVE-003	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-004	Low	Suggested Immutable Usages For Gas Efficiency	Coding Practices	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 Implicit Assumption Enforcement In addLiquidity()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: GinFinanceRouter02
- Category: Coding Practices [6]
- CWE subcategory: CWE-628 [3]

Description

In the `GinFinanceRouter02` contract, it provides the `addLiquidity()` routine for liquidity providers to add `amountADesired` amount of `tokenA` and `amountBDesired` amount of `tokenB` into the pool as liquidity. To elaborate, we show below the code snippets of the `addLiquidity()` routine and the `_addLiquidity()` routine.

```
    function addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) external virtual override ensure(deadline) returns (uint amountA, uint amountB,
        uint liquidity) {
        (amountA, amountB) = _addLiquidity(tokenA, tokenB, amountADesired,
            amountBDesired, amountAMin, amountBMin);
        address pair = GinFinanceLibrary.pairFor(factory, tokenA, tokenB);
        TransferHelper.safeTransferFrom(tokenA, msg.sender, pair, amountA);
        TransferHelper.safeTransferFrom(tokenB, msg.sender, pair, amountB);
        liquidity = IGinFinancePair(pair).mint(to);
    }
```

Listing 3.1: GinFinanceRouter02::addLiquidity()

```
    function _addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin
    ) internal virtual returns (uint amountA, uint amountB) {
        // create the pair if it doesn't exist yet
        if (IGinFinanceFactory(factory).getPair(tokenA, tokenB) == address(0)) {
            IGinFinanceFactory(factory).createPair(tokenA, tokenB);
        }
        (uint reserveA, uint reserveB) = GinFinanceLibrary.getReserves(factory, tokenA,
            tokenB);
        if (reserveA == 0 && reserveB == 0) {
            (amountA, amountB) = (amountADesired, amountBDesired);
        } else {
            uint amountBOptimal = GinFinanceLibrary.quote(amountADesired, reserveA,
                reserveB);
            if (amountBOptimal <= amountBDesired) {
                require(amountBOptimal >= amountBMin, 'GinFinanceRouter:
                    INSUFFICIENT_B_AMOUNT');
                (amountA, amountB) = (amountADesired, amountBOptimal);
            } else {
                uint amountAOptimal = GinFinanceLibrary.quote(amountBDesired, reserveB,
                    reserveA);
                assert(amountAOptimal <= amountADesired);
                require(amountAOptimal >= amountAMin, 'GinFinanceRouter:
                    INSUFFICIENT_A_AMOUNT');
                (amountA, amountB) = (amountAOptimal, amountBDesired);
            }
        }
    }
```

Listing 3.2: `GinFinanceRouter02::_addLiquidity()`

It comes to our attention that the `GinFinanceRouter02` has implicit assumptions on the `_addLiquidity()` routine. The above routine takes two amounts: `amountXDesired` and `amountXMin`. The first amount `amountXDesired` determines the desired amount for adding liquidity to the pool and the second amount `amountXMin` determines the minimum amount of used assets. There are two implicit conditions, i.e., `amountADesired >= amountAMin` and `amountBDesired >= amountBMin`. However, if these two conditions are not met, current logic will not trigger reverts because the code above performs asymmetric checks for these amounts. Hence, without stating these assumptions, slippage control for some trades on `GinFinanceRouter02` may not be checked and may not be taken into account at all in certain scenarios.

Recommendation Make the requirement of `amountADesired >= amountAMin` and `amountBDesired >= amountBMin` explicit in the `addLiquidity()` routine.

Status This issue has been fixed in the following commit: b420e65.

3.2 Fork-Resistant Domain Separator In GinFinanceERC20

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: GinFinanceERC20
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In the GinFinanceERC20 contract, the state variable DOMAIN_SEPARATOR is immutable because it is only assigned in the constructor() function (lines 30-37).

```
25      constructor() public {
26          uint chainId;
27          assembly {
28              chainId := chainid()
29          }
30          DOMAIN_SEPARATOR = keccak256(
31              abi.encode(
32                  keccak256('EIP712Domain(string name,string version,uint256 chainId,
                        address verifyingContract)'),
33                  keccak256(bytes(name)),
34                  keccak256(bytes('1')),
35                  chainId,
36                  address(this))
37          );
38      }
```

Listing 3.3: GinFinanceERC20::constructor()

The DOMAIN_SEPARATOR is used in the ERC20-extending function, i.e., permit(), which allows for approvals to be made via secp256k1 signatures. When analyzing this permit() routine, we notice the current implementation can be improved by recalculating the value of DOMAIN_SEPARATOR in permit() function. Suppose there is a hard-fork, since DOMAIN_SEPARATOR is immutable, a valid signature for one chain could be replayed on the other.

```
    function permit(address owner, address spender, uint value, uint deadline, uint8 v,
        bytes32 r, bytes32 s)
    external {
            require(deadline >= block.timestamp, 'GinFinanceERC20: EXPIRED');
            bytes32 digest = keccak256(
                abi.encodePacked(
                    '\x19\x01',
                    DOMAIN_SEPARATOR,
```

```
                keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value, nonces[
                    owner]++, deadline))
            )
        );
        address recoveredAddress = ecrecover(digest, v, r, s);
        require(recoveredAddress != address(0) && recoveredAddress == owner, '
            GinFinanceERC20: INVALID_SIGNATURE');
        _approve(owner, spender, value);
    }
```

Listing 3.4: `GinFinanceERC20::permit()`

Recommendation Recalculate the value of `DOMAIN_SEPARATOR` inside the `permit()` routine.

Status This issue has been fixed in the following commit: `5dd181a`.

3.3 Trust Issue Of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `GinFinanceFactory`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `GinFinanceFactory` contract, there is a privileged `feeToSetter` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., set the `pairFee/feeToAlloc/feeTo`). In the following, we show the representative functions potentially affected by the privileged `feeToSetter`.

```
    function setFeeTo(address _feeTo) external override {
        require(msg.sender == feeToSetter, 'GinFinanceFactory: FORBIDDEN');
        feeTo = _feeTo;
    }

    function setFeeToSetter(address _feeToSetter) external override {
        require(msg.sender == feeToSetter, 'GinFinanceFactory: FORBIDDEN');
        feeToSetter = _feeToSetter;
    }

    function setFeeToAlloc(uint _feeToAlloc) external override {
        require(msg.sender == feeToSetter, 'GinFinanceFactory: FORBIDDEN');
        require(_feeToAlloc <= 5, "GinFinanceFactory: FEE_TO_ALLOC_OVERFLOW");
        feeToAlloc = _feeToAlloc;
    }

    // Max is 1%
    function setDefaultFee(uint _defaultFee) external override {
```

```
        require(msg.sender == feeToSetter, 'GinFinanceFactory: FORBIDDEN');
        require(_defaultFee != 0 && _defaultFee <= 100, "GinFinanceFactory:
            FEE_OUT_OF_RANGE");
        defaultFee = _defaultFee;
    }

    // Max is 1%
    function setPairFee(address _pair, uint _fee) external override {
        require(msg.sender == feeToSetter, 'GinFinanceFactory: FORBIDDEN');
        require(_fee != 0 && _fee <= 100, 'GinFinanceFactory: FEE_OUT_OF_RANGE');
        pairFee[_pair] = _fee;
    }
```

Listing 3.5: `GinFinanceFactory.sol`

We emphasize that the privilege assignment is indeed necessary and consistent with the protocol design. However, it is worrisome if the privileged account is a plain EOA account. A multi-sig account could greatly alleviate this concern, though it is still far from perfect. Note that a compromised account would allow the attacker to set the `pairFee/feeToAlloc/feeTo`, etc. arbitrarily, which directly undermines the assumption of the `GinFinance` design.

Recommendation Suggest a multi-sig account plays the privileged `feeToSetter` account to mitigate this issue. Additionally, all changes to privileged operations may need to be mediated with necessary timelocks.

Status This issue has been confirmed by the team.

3.4 Suggested Immutable Usages For Gas Efficiency

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `GinFinancePair`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1099 [1]

Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

In the following, we show the key state variable `factory` in the `GinFinancePair` contract. If there is no need to dynamically update this key variable, it can be declared as either constant or `immutable` for gas efficiency. In particular, the above state variable can be defined as `immutable` as it will not be changed after its initialization in `constructor()`.

```
contract GinFinancePair is GinFinanceERC20 {
  using SafeMath for uint;
  using UQ112x112 for uint224;

  uint public constant MINIMUM_LIQUIDITY = 10**3;
  bytes4 private constant SELECTOR = bytes4(keccak256(bytes('transfer(address,uint256)')));

  address public factory;
  address public token0;
  address public token1;
  ...
  constructor() public {
      factory = msg.sender;
  }
```

Listing 3.6: `GinFinancePair.sol`

Recommendation Revisit the state variable definition and make extensive use of `immutable` states for gas efficiency.

Status This issue has been fixed in the following commit: `a3892a6`.

4 Conclusion

In this audit, we have analyzed the `GinFinance` protocol design and implementation. The protocol is the ultimate `DeFi` solution for asset liquidity and exchange on `BOBA` Network. With its efficient swap and staking model, it offers wide range of decentralized applications to `DeFi` users at convenience. 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.

PeckShield

References

[1] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

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

[3] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.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 VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

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

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

PeckShield