SMART CONTRACT AUDIT REPORT

for

Furucombo Funds

Prepared By: Patrick Lou

PeckShield

May 6, 2022

Document Properties

Client	Furucombo
Title	Smart Contract Audit Report
Target	Furucombo Funds
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Patrick Lou, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 6, 2022	Xuxian Jiang	Final Release
1.0-rc	April 16, 2022	Xuxian Jiang	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 design document and related smart contract source code of the `Furucombo Funds` protocol, we outline in this 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 contract can be further improved due to the presence of several issues. This document outlines our audit results.

1.1 About Furucombo Funds

`Furucombo Funds` is a platform that enables users to create their own fund, and also join other users' funds to maximize their value of knowledge and assets. By creating their own funds, fund managers are able to perform different strategies to gain the best profit on the fund's behalf, earning themselves the fee generated by the fund system. Investors can also join funds that they are interested in by simply purchasing the share of the funds. The strategy execution is empowered by integrated `DeFi` protocols and `Furucombo` system, the leading `DeFi` aggregator, to enable all kinds of leveraging strategies.

Table 1.1: Basic Information of Furucombo Funds

Item	Description
Name	Furucombo
Website	https://furucombo.app/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 6, 2022

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

- https://github.com/dinngodev/furucombo-funds-contract.git (8a17ef1)

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

- https://github.com/dinngodev/furucombo-funds-contract.git (0633b98)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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 OWASP Risk Rating Methodology [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business 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 `Furucombo Funds` 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 logics, 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	3	■■■
Low	2	■■
Informational	0	
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 3 medium-severity vulnerabilities and 2 low-severity vulnerabilities.

Table 2.1: Key Furucombo Funds Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Lack Of Caller Verification in MortgageVault::mortgage()	Security Features	Resolved
PVE-002	Low	Revisited Logic in _set() Helpers	Coding Practices	Resolved
PVE-003	Low	Possible Costly Share Token From Improper Initialization	Time and State	Resolved
PVE-004	Medium	Improved Mortgage Claim In Vault Liquidation/Closure	Business Logic	Resolved
PVE-005	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Lack Of Caller Verification in MortgageVault::mortgage()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: MortgageVault
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

The Furucombo Funds protocol enables users to create their own fund, and also join other users' funds to maximize their value of knowledge and assets. The creation of a user fund requires the deposit of certain corresponding amount of assets to the desired mortgage tier. The assets will be transferred to the mortgage vault when the fund is created. While reviewing the logic of the mortgage vault, we notice the current implementation needs to be improved.

Specifically, we show below the related mortgage() function, which is used to transfer the corresponding amount of mortgage assets when the fund is created. It comes to our attention that this function is permissionless and allows anyone to invoke it. Notice the first parameter of this function represents the sender of the mortgage fund. There is a need to add necessary caller verification to this function to avoid being abused to transfer the funds from approving users without their notice.

```
    function mortgage(
        address sender_,
        address fund_,
        uint256 amount_
    ) external {
        Errors._require(fundAmounts[fund_] == 0, Errors.Code.
            MORTGAGE_VAULT_FUND_MORTGAGED);
        fundAmounts[fund_] += amount_;
        totalAmount += amount_;
        mortgageToken.safeTransferFrom(sender_, address(this), amount_);
        emit Mortgaged(sender_, fund_, amount_);
```

```
    }
```

Listing 3.1: MortgageVault::mortgage()

Recommendation Validate the caller to the above `mortgage()` function to ensure only the trusted caller can be allowed for interaction.

Status The issue has been fixed by this commit: `c310e4d`.

3.2 Revisited Logic in _set() Helpers

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

To facilitate the management of various storage states, the `Furucombo Funds` protocol provides a number of library routines. In particular, the `StorageMap` library manages the mapping-based storage states and the `StorageArray` library facilitates array-based storage states. While analyzing these library routines, we notice a set of `_set()` helpers can be improved.

To elaborate, we show below the related `_set()` function from the `StorageArray` library. Note the value is internally saved as a `bytes32` type, while the current implementation makes the type transformation from `bytes32` to `bytes20` and then to `bytes32` again (line 29), which may unnecessarily bring unwanted information loss.

```
21    function _set(
22        bytes32 slot_,
23        uint256 index_,
24        bytes32 val_
25    ) internal {
26        require(index_ < uint256(_getSlot(slot_).value), "StorageArray: _set invalid
              index");
27        uint256 s = uint256(keccak256(abi.encodePacked(uint256(slot_)))) + index_;
28
29        bytes32 val = bytes32(bytes20(val_));
30        _getSlot(bytes32(s)).value = val;
31    }
```

Listing 3.2: `StorageArray::_set()`

Also, the `_set()` function from the `DealingAsset` library makes use of the `_ASSET_FALSE_FLAG` flag to mark the non-presence of the `key_`. However, it also accepts the same flag as the key value, which may cause confusion and introduce unwanted duplicates in the underlying array `_ASSET_ARR_SLOT`. The same issue is also applicable to the `_set()` function from the `FundQuota` library.

```
    function _set(address key_, bool val_) internal {
        bytes32 key = bytes32(bytes20(key_));
        bytes32 oldVal = _ASSET_MAP_SLOT._get(key);

        if (oldVal == _ASSET_FALSE_FLAG) {
            _ASSET_ARR_SLOT._push(key);
        }

        if (val_) {
            _ASSET_MAP_SLOT._set(key, _ASSET_TRUE_FLAG);
        } else {
            _ASSET_MAP_SLOT._set(key, _ASSET_FALSE_FLAG);
        }
    }
```

Listing 3.3: `DealingAsset::_set()`

Recommendation Revise the above-mentioned `_set()` helper routines for their intended functionalities.

Status The issue has been fixed by this commit: `be4e79c`.

3.3 Possible Costly Share Token From Improper Initialization

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ShareModule`
- Category: Time and State [6]
- CWE subcategory: CWE-362 [3]

Description

The `Furucombo Funds` protocol allows users to join other users' funds by depositing supported assets with the returned share to represent the fund ownership. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the fund share extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the `purchase()` routine, which is used for participating users to deposit the supported assets and get respective fund pool shares in return. The issue occurs when the fund pool is being initialized under the assumption that the current pool is empty.

```
    function purchase(uint256 balance_)
        external
```

```
        virtual
        whenStates(State.Executing, State.Pending)
        nonReentrant
        returns (uint256 share)
    {
        share = _purchase(msg.sender, balance_);
    }

    function _purchase(address user_, uint256 balance_) internal virtual returns (
        uint256 share) {
        uint256 grossAssetValue = _beforePurchase();
        share = _addShare(user_, balance_, grossAssetValue);

        uint256 penalty = _getPendingPenalty();
        uint256 bonus;
        if (state == State.Pending) {
            bonus = (share * (penalty)) / (_FUND_PERCENTAGE_BASE - penalty);
            bonus = currentTotalPendingBonus > bonus ? bonus : currentTotalPendingBonus;
            currentTotalPendingBonus -= bonus;
            shareToken.move(address(this), user_, bonus);
            share += bonus;
        }
        grossAssetValue += balance_;
        denomination.safeTransferFrom(msg.sender, address(vault), balance_);
        _afterPurchase(grossAssetValue);

        emit Purchased(user_, balance_, share, bonus);
    }
```

Listing 3.4: `ShareModule::purchase()`

```
    function _addShare(
        address user_,
        uint256 balance_,
        uint256 grossAssetValue_
    ) internal virtual returns (uint256 share) {
        share = _calculateShare(balance_, grossAssetValue_);
        shareToken.mint(user_, share);
    }
```

Listing 3.5: `ShareModule::_addShare()`

```
    function _calculateShare(uint256 balance_, uint256 grossAssetValue_) internal view
        virtual returns (uint256 share) {
        uint256 shareAmount = shareToken.grossTotalShare();
        if (shareAmount == 0) {
            // Handler initial minting
            share = balance_;
        } else {
            share = (shareAmount * balance_) / grossAssetValue_;
        }
```

```
37      }
```

Listing 3.6: `ShareModule::_calculateShare()`

Specifically, when the pool is being initialized (line 31), the share value directly takes the value of `balance_` (line 33), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `share = balance_ = 1 WEI`. With that, the actor can further donate a huge amount of the underlying assets with the goal of making the pool share extremely expensive.

An extremely expensive pool share can be very inconvenient to use as a small number of 1 `Wei` may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current purchase logic to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure a guarded launch process that safeguards the first deposit to avoid being manipulated.

Status The issue has been fixed by this commit: `0db27d6`.

3.4 Improved Mortgage Claim In Vault Liquidation/Closure

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `FundImplementation`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [4]

Description

Each fund created in the `Furucombo Funds` protocol has a lifecycle with the associated state-machine that contains the following six self-evident states: `Initializing`, `Reviewing`, `Executing`, `Pending`, `Liquidating`, and `Closed`. While reviewing the state-transition among these states, we notice the mortgage funds will also be accordingly transferred.

To elaborate, we show below the `liquidate()` function, which triggers the state transition from `Pending` to `Liquidating`. And this specific transition also leads to the transfer of mortgage funds to the protocol owner (`comptroller.owner()` - line 118). This is inconsistent with the design as the liquidation should be transferred to the liquidator, not the protocol owner!

```
140     /// @notice Liquidate the fund by anyone and transfer owner to liquidator.
141     function liquidate() external nonReentrant {
142         Errors._require(pendingStartTime != 0, Errors.Code.
                IMPLEMENTATION_PENDING_NOT_START);
143         Errors._require(
144             block.timestamp >= pendingStartTime + comptroller.pendingExpiration(),
145             Errors.Code.IMPLEMENTATION_PENDING_NOT_EXPIRE
146         );
147 
148         _liquidate();
149 
150         mortgageVault.claim(comptroller.owner());
151         _transferOwnership(comptroller.pendingLiquidator());
152     }
153 
154     /// @notice Close the fund. The pending share will be settled
155     /// without penalty.
156     function close() public override onlyOwner nonReentrant whenStates(State.Executing,
            State.Liquidating) {
157         if (_getResolvePendingShare(false) > 0) {
158             _settlePendingShare(false);
159         }
160 
161         super.close();
162 
163         mortgageVault.claim(msg.sender);
164     }
```

Listing 3.7: `FundImplementation::liquidate()/close()`

Recommendation Revise the above `liquidate()` function with the proper transfer of the mortgage funds to the intended recipient.

Status The issue has been fixed by this commit: `c310e4d`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the Furucombo Funds protocol, there is a privileged owner account that plays a critical role in governing and regulating the system-wide operations (e.g., whitelist/blacklist handlers and configure various parameters). In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    function permitHandlers(
        uint256 level_,
        address[] calldata tos_,
        bytes4[] calldata sigs_
    ) external consistentTosAndSigsLength(tos_, sigs_) onlyOwner {
        for (uint256 i = 0; i < tos_.length; i++) {
            _handlerCallACL._permit(level_, tos_[i], sigs_[i]);
            emit PermitHandler(level_, tos_[i], sigs_[i]);
        }
    }

    function forbidHandlers(
        uint256 level_,
        address[] calldata tos_,
        bytes4[] calldata sigs_
    ) external consistentTosAndSigsLength(tos_, sigs_) onlyOwner {
        for (uint256 i = 0; i < tos_.length; i++) {
            _handlerCallACL._forbid(level_, tos_[i], sigs_[i]);
            emit ForbidHandler(level_, tos_[i], sigs_[i]);
        }
    }
```

Listing 3.8: Example Privileged Operations in ComptrollerImplementation

Notice that the privilege assignment is necessary and consistent with the protocol design. In the meantime, the extra power to the owner may also be a counter-party risk to the protocol users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these onlyOwner privileges explicit or raising necessary awareness among protocol users.

Recommendation Making these onlyOwner privileges explicit among protocol users.

Status This issue has been mitigated and the team has confirmed that these privileged functions should be called by a multi-sig wallet, which for safety will have to permit/forbid handlers.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Furucombo Funds` protocol, which is a platform that enables users to create their own fund, and also join other users' funds to maximize their value of knowledge and assets. By creating their own funds, fund managers are able to perform different strategies to gain the best profit on the fund's behalf, earning themselves the fee generated by the fund system. The current code base is well organized and those identified issues are promptly confirmed and addressed.

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-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.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: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.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 VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

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

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