SMART CONTRACT AUDIT REPORT

for

WOOFi Earn

Prepared By: Xiaomi Huang

PeckShield

August 20, 2022

Document Properties

Client	WOOFi
Title	Smart Contract Audit Report
Target	WOOFi Earn
Version	1.0
Author	Xuxian Jiang
Auditors	Patrick Lou, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 20, 2022	Xuxian Jiang	Final Release
1.0-rc1	August 18, 2022	Xuxian Jiang	Release Candidate #1

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

Given the opportunity to review the design document and related smart contract source code of the WOOFi Earn 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 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 WOOFi Earn

WOOFi Earn provides a hassle-free experience by offering "set-and-forget" yield generating strategies, where users can simply deposit into a vault and let the automated strategies do the rest. Specifically, the audited Supercharger vault further boosts yields for users by allowing liquidity provision in WOOFi 's highly capital-efficient sPMM liquidity pools. Each Supercharger vault adopts a base yield farming strategy in combination with lending assets to the WOOFi liquidity provider. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of WOOFi Earn

Item	Description
Name	WOOFi
Website	https://woo.org/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 20, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the given repository has a number of files and this audit only covers WooLendingManager

.sol and WooSuperChargerVault.sol.

- https://github.com/woonetwork/woofi_swap_smart_contracts.git (0cdd667)

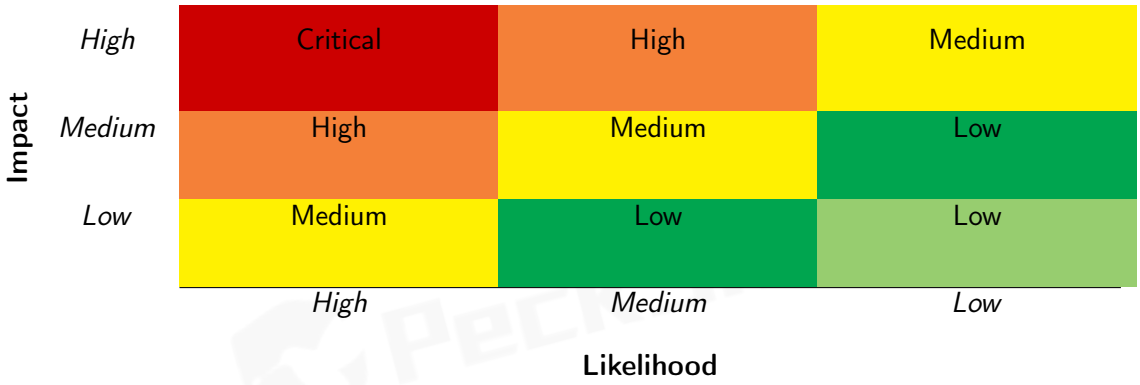
And here is the commit ID after all fixes for the issues found in the audit have been checked in.

- https://github.com/woonetwork/woofi_swap_smart_contracts.git (fc5ca15)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification



1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [8]:

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

Table 1.3: The Full Audit Checklist

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

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

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

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

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered

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

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `WOOFi Earn` 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	1	■
Low	2	■■
Informational	1	■
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 that 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 2 medium-severity vulnerabilities, 1 low-severity vulnerability, and 1 undetermined issue.

Table 2.1: Key Audit Findings of WOOFi Earn Protocol

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Repeated Initialization of WooLendingManager/WooSuperChargerVault	Security Features	Resolved
PVE-002	Low	Improved Weekly Settlement in WooSuperChargerVault	Business Logic	Resolved
PVE-003	Informational	Suggested Event Generation For Parameter Updates	Coding Practices	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Possible Repeated Initialization of WooLendingManager/WooSuperChargerVault

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

At the core of the `WOOFi Earn` protocol is the `WooSuperChargerVault` contract, which adopts a base yield farming strategy in combination with lending assets to the `WOOFi` liquidity provider. While examining the `WooSuperChargerVault` contract, we notice it has a privileged `init()` routine to configure the current `reserveVault`, `lendingManager`, and `withdrawManager`. This initialization routine is guarded with the `onlyOwner` modifier so that only the current owner is allowed to invoke it.

```
118     function init(
119         address _reserveVault,
120         address _lendingManager,
121         address payable _withdrawManager
122     ) external onlyOwner {
123         require(_reserveVault != address(0), 'WooSuperChargerVault: !_reserveVault');
124         require(_lendingManager != address(0), 'WooSuperChargerVault: !_lendingManager')
                ;
125         require(_withdrawManager != address(0), 'WooSuperChargerVault: !_withdrawManager
                ');
126
127         reserveVault = IVaultV2(_reserveVault);
128         require(reserveVault.want() == want);
129         lendingManager = WooLendingManager(_lendingManager);
130         withdrawManager = WooWithdrawManager(_withdrawManager);
131     }
```

Listing 3.1: `WooSuperChargerVault::init()`

To elaborate, we show above the implementation of the `init()` routine. Our analysis shows that this routine can be repeatedly invoked to update or change `reserveVault`, `lendingManager`, and `withdrawManager`. While it requires the `owner` privilege and may be needed for operational convenience, the possibility of repeated initialization by the privileged owner may expose unnecessary risks to protocol users. With that, we suggest to ensure the `init()` routine can only be invoked once.

Recommendation Ensure the `WooSuperChargerVault` contract can only be initialized once. The same issue is also applicable to the `WooLendingManager` contract.

Status The issue has been resolved as the team confirms it is part of the design. In the meantime, the clarifies that the `init()` will not be repeatedly invoked.

3.2 Improved Weekly Settlement in WooSuperChargerVault

- ID: PVE-002
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `WooSuperChargerVault`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, the `WOOFi Earn` protocol has a core `WooSuperChargerVault` contract that adopts a base yield farming strategy in combination with lending assets to the `WOOFi` liquidity provider. As part of the built-in logic, this `WooSuperChargerVault` has a weekly schedule for settlement, which is implemented in two main functions `startWeeklySettle()` and `endWeeklySettle()`. Our analysis shows that the current weekly settlement implementation can be improved.

To elaborate, we show below the related code snippets from these two functions. As the name indicates, the `startWeeklySettle()` function is used to trigger the settlement while the `endWeeklySettle()` function performs the actual settlement. It comes to our attention that the latter requires the fresh calculation of the share price in `getPricePerFullShare()` (line 301). However, the share price may be affected by the accrued interest in `lendingManager`. In other words, we need to ensure the interest is timely accrued within `endWeeklySettle()` before the share price is calculated!

```
290     function startWeeklySettle() external onlyAdmin {
291         require(!isSettling, 'IN_SETTLING');
292         isSettling = true;
293         lendingManager.accureInterest();
294         emit WeeklySettleStarted(msg.sender, requestedTotalShares,
                weeklyNeededAmountForWithdraw());
295     }
296
```

```
297     function endWeeklySettle() public onlyAdmin {
298         require(isSettling, '!SETTLING');
299         require(weeklyNeededAmountForWithdraw() == 0, 'WEEKLY_REPAY_NOT_CLEARED');
300
301         uint256 sharePrice = getPricePerFullShare();
302
303         isSettling = false;
304         uint256 amount = requestedTotalAmount();
305
306         if (amount != 0) {
307             uint256 shares = _sharesUp(amount, reserveVault.getPricePerFullShare());
308             reserveVault.withdraw(shares);
309
310             if (want == weth) {
311                 IWETH(weth).deposit{value: amount}();
312             }
313             require(available() >= amount);
314
315             TransferHelper.safeApprove(want, address(withdrawManager), amount);
316             uint256 length = requestUsers.length();
317             for (uint256 i = 0; i < length; i++) {
318                 address user = requestUsers.at(0);
319
320                 withdrawManager.addWithdrawAmount(user, requestedWithdrawShares[user].
                        mul(sharePrice).div(1e18));
321
322                 requestedWithdrawShares[user] = 0;
323                 requestUsers.remove(user);
324             }
325
326             _burn(address(this), requestedTotalShares);
327             requestedTotalShares = 0;
328         }
329
330         instantWithdrawnAmount = 0;
331
332         lendingManager.accureInterest();
333         uint256 totalBalance = balance();
334         instantWithdrawCap = totalBalance.div(10);
335
336         emit WeeklySettleEnded(msg.sender, totalBalance, lendingBalance(),
                reserveBalance());
337     }
```

Listing 3.2: WooSuperChargerVault::startWeeklySettle()/endWeeklySettle()

Recommendation Timely accrue the interest before calculating the share price in the above `endWeeklySettle()` routine.

Status This issue has been resolved as the team clarifies that the suggested interest accrual in the above `endWeeklySettle()` routine may cause difficulty in the proper settlement.

3.3 Suggested Event Generation For Parameter Updates

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

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.

While examining the events that reflect the protocol dynamics in the WooSuperChargerVault contract, we notice there is a lack of emitting an event to reflect the changes of various important states, including lendingManager, withdrawManager, and treasury. To elaborate, we show below the code snippet of these setter routines.

```
369     function setLendingManager(address _lendingManager) external onlyOwner {
370         lendingManager = WooLendingManager(_lendingManager);
371     }

373     function setWithdrawManager(address payable _withdrawManager) external onlyOwner {
374         withdrawManager = WooWithdrawManager(_withdrawManager);
375     }

377     function setTreasury(address _treasury) external onlyOwner {
378         treasury = _treasury;
379     }

381     function setInstantWithdrawFeeRate(uint256 _feeRate) external onlyOwner {
382         instantWithdrawFeeRate = _feeRate;
383     }
```

Listing 3.3: Various Setters in WooSuperChargerVault

With that, we suggest to add the respective events in the above setter routines. Also, the address-related parameters are better indexed. Specifically, each emitted event is represented as a topic that usually consists of the signature (from a keccak256 hash) of the event name and the types (uint256, string, etc.) of its parameters. Each indexed type will be treated like an additional topic. If an argument is not indexed, it will be attached as data (instead of a separate topic). Considering

that the address-related parameters may be queried, it is better treated as topics, hence the need of being `indexed`.

Recommendation Properly emit the respective events when important protocol parameters are updated. This is very helpful for external analytics and reporting tools.

Status The issue has been fixed by this commit: `fc5ca15`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `WOOFi Earn` contract, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., migrate the vault, recover stuck tokens, etc). Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
339     function migrateReserveVault(address _vault) external onlyOwner {
340         require(_vault != address(0), '!_vault');

342         uint256 preBal = (want == weth) ? address(this).balance : available();
343         reserveVault.withdraw(IERC20(address(reserveVault)).balanceOf(address(this)));
344         uint256 afterBal = (want == weth) ? address(this).balance : available();
345         uint256 reserveAmount = afterBal.sub(preBal);

347         address oldVault = address(reserveVault);
348         reserveVault = IVaultV2(_vault);
349         require(reserveVault.want() == want, 'INVALID_WANT');
350         if (want == weth) {
351             reserveVault.deposit{value: reserveAmount}(reserveAmount);
352         } else {
353             TransferHelper.safeApprove(want, address(reserveVault), reserveAmount);
354             reserveVault.deposit(reserveAmount);
355         }

357         emit ReserveVaultMigrated(msg.sender, oldVault, _vault);
358     }

360     function inCaseTokenGotStuck(address stuckToken) external onlyOwner {
361         if (stuckToken == ETH_PLACEHOLDER_ADDR) {
362             TransferHelper.safeTransferETH(msg.sender, address(this).balance);
```

```
363         } else {
364             uint256 amount = IERC20(stuckToken).balanceOf(address(this));
365             TransferHelper.safeTransfer(stuckToken, msg.sender, amount);
366         }
367     }
```

Listing 3.4: Various Privileged Functions in `WooSuperChargerVault`

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 privileges explicit or raising necessary awareness among protocol users.

Moreover, it should be noted that if current contracts are planned to deploy behind a proxy, there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Making the above privileges explicit among protocol users.

Status This issue has been confirmed and the team clarifies that the admin key is managed by a 3/5 multi-sig wallet using `Gnosis Safe`. In addition, the `supercharger vault` does not hold any want token, but only dispatches 'want' token to either `reserveVault` or `lendingManager`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the WOOFi Earn protocol, which provides a hassle-free experience by offering "set-and-forget" yield generating strategies, where users can simply deposit into a vault and let the automated strategies do the rest. Specifically, the audited Supercharger vault further boosts yields for users by allowing liquidity provision in WOOFi's highly capital-efficient sPMM liquidity pools. Each Supercharger vault adopts a base yield farming strategy in combination with lending assets to the WOOFi liquidity provider. The current code base is well organized and those identified issues are promptly confirmed and fixed.

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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

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

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

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

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

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