SMART CONTRACT AUDIT REPORT

for

Orcus Finance

Prepared By: Xiaomi Huang

PeckShield

May 22, 2022

Document Properties

Client	Orcus Finance
Title	Smart Contract Audit Report
Target	Orcus
Version	1.0
Author	Xuxian Jiang
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 22, 2022	Xuxian Jiang	Final Release
1.0-rc1	May 20, 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 `Orcus Finance` 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 Orcus Finance

`Orcus Finance` aims to deliver the first `Fractional-Algorithmic Stablecoin` pegged to the `United States Dollar` which is built on the `Astar Network`. Since `Astar Network` is a gateway to the multi-chain environment, the protocol allows for expansion to as many networks as possible in the near future. `Orcus Finance` is an entirely decentralized and autonomous protocol with the native governance token which aims to be the first leading `Fractional-Algorithmic Stablecoin` issuer on the `Astar network` and to implement multiple useful financial tools and other synthetic assets in the ecosystem. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Orcus

Item	Description
Issuer	Orcus Finance
Website	https://orcusfinance.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 22, 2022

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

- https://github.com/OrcusFinance/Orcus-Finance.git (6aaf797)

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

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 [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.

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 contract is considered safe regarding the check item. For any discovered issue, we might further

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

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.

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 Logics	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 `Orcus Finance` protocol implementation. 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 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 3 medium-severity vulnerabilities and 2 low-severity vulnerabilities.

Table 2.1: Key Orcus Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incorrect Logic in unclaimedTeamFund()	Business Logic	Resolved
PVE-002	Medium	Improper Logic in Farm::_calcAccOruToAdd()	Business Logic	Resolved
PVE-003	Low	Revisited Logic in ORUSale::sendCarryOveredORU()	Coding Practices	Resolved
PVE-004	Low	Improved Logic to swap() in Arbitrager	Business Logic	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

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 Incorrect Logic in unclaimedTeamFund()

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `ORU`, `OrcusV1Distributor`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Orcus` protocol has the built-in tokenomics that distribute the protocol tokens `ORU` to various ecosystem members. While analyzing the token distribution logic, we notice the current implementation needs to be improved.

To elaborate, we show below this helper routine from the `ORU` token contract. This `unclaimedTeamFund()` routine is designed to compute the unclaimed fund that can be claimed by the team. It comes to our attention that the routine always makes use of the current `emissionRate`, without considering the possibility where the elapsed time may cross multiple years, which require the use of respective `emissionRate` on each year!

```
    function unclaimedTeamFund() public view returns (uint256) {
        uint256 _now = block.timestamp;

        if (_now <= teamVesting.lastClaimed) {
            return 0;
        }

        uint256 _fromEpoch = _now - teamVesting.startTime;
        uint256 _years = _fromEpoch / ONE_YEAR;

        uint256 _emissionRate = TEAM_FUND_EMISSION_RATE;
        for (uint256 i = 0; i < _years; i++) {
            _emissionRate =
                _emissionRate -
                ((_emissionRate * VESTING_DECREASING_RATIO) / RATIO_PRECISION);
```

```
        }

        uint256 _timeElapsed = _now - teamVesting.lastClaimed;
        uint256 _available = Math.min(
            _timeElapsed * _emissionRate,
            TEAM_FUND_ALLOCATION - teamVesting.vestedAmount
        );

        return _available;
    }
```

Listing 3.1: `ORU::unclaimedTeamFund()`

Recommendation Revise the above `unclaimedTeamFund()` routine to properly compute the funds that can be claimed by the team. Note the same issue is also applicable to another contract `OrcusV1Distributor`.

Status This issue has been resolved as the team intends to timely claim the funds without crossing the year boundary.

3.2 Improper Logic in Farm::_calcAccOruToAdd()

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

Description

The `Orcus` protocol has a `Farm` contract to incentivize the protocol adoption among the community. This `Farm` follows the popular `MasterChef` approach to specify the pools for farming. While examining the current logic to calculate the pool-specific `accOruPerShare` state, we notice the implementation should be improved.

In the following, we show the related `_calcAccOruToAdd()` routine in the `Farm` contract. As the name indicates, this routine is used to compute the new normalized reward index for the given pool (with the `pid`). It comes to our attention the computation is scaled by `10**_lpDecimals` (line 221), which may cause inconsistency when it does not equal to the intended `ORU_PRECISION`! In other words, we should use the `ORU_PRECISION` as the scaling factor, instead of `10**_lpDecimals`.

```
210     /// @dev Calculated amount of accOruPerShare to add to the pool.
211     function _calcAccOruToAdd(uint256 _pid) internal view returns (uint256) {
212         PoolInfo memory pool = poolInfo[_pid];
213         uint256 _lpSupply = lpToken[_pid].balanceOf(address(this));
```

```
        uint256 _lpDecimals = ERC20(address(lpToken[_pid])).decimals();
        uint64 _currentTs = _currentBlockTs();
        if (_currentTs <= pool.lastRewardTime  _lpSupply <= 0) {
            return 0;
        }
        uint256 _time = _currentTs - pool.lastRewardTime;
        uint256 _oruReward = (_time * oruPerSecond * pool.allocPoint) /
            totalAllocPoint;
        return (_oruReward * (10**_lpDecimals)) / _lpSupply;
    }
```

Listing 3.2: `Farm::_calcAccOruToAdd()`

Recommendation Be consistent in the use of the scaling factor for reward index calculation.

Status This issue has been resolved as the team confirms the scaling factor is ensured to be the same as `10**_lpDecimals`.

3.3 Revisited Logic in ORUSale::sendCarryOveredORU()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ORUSale`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The `Orcus` protocol also has a `ORUSale` contract to allow for users to directly purchase `ORU` tokens. The sale logic specifies the use of `USDC` as the payment token and regulates the claims so that once a claim is made, the next claim needs to wait for 2 weeks. In the following, we show the related `claim()` routine.

```
    function claim() public {
        require(!saleStarted, "Sale does not finished yet.");
        require(claims[msg.sender].userExist, "User does not exist on claim list");
        require(claims[msg.sender].oruAmt > 0, "Claims is over for this user");

        uint256 claimTime = block.timestamp;
        uint amtToSend = claims[msg.sender].distributionAmt;

        if (!claims[msg.sender].firstTimeClaimed) {
            oru.safeTransfer(msg.sender, amtToSend);

            claims[msg.sender].lastClaimed = claimTime;
            claims[msg.sender].oruAmt -= amtToSend;
            claims[msg.sender].firstTimeClaimed = true;
```

```
        }

        else {
            require(claims[msg.sender].lastClaimed + INTERVAL_BETWEEN_CLAIMS <=
                claimTime, "Claim time isn't come");

            oru.safeTransfer(msg.sender, amtToSend);
            claims[msg.sender].lastClaimed = claimTime;
            claims[msg.sender].oruAmt -= amtToSend;
        }

    }
```

Listing 3.3: `ORUSale::claim()`

In the meantime, we notice the `ORUSale` contract has another privileged function `sendCarryOveredORU()`, which allows to withdraw all `ORU` tokens from the contract. In other words, the privileged `owner` is able to drain `ORU` tokens even before the buyers are able to claim all purchased tokens.

```
    function sendCarryOveredORU() public onlyOwner {

        require(saleFinished, "Sale isn't finished");
        uint oruBal = oru.balanceOf(address(this));

        oru.safeTransfer(owner(), oruBal);
    }
```

Listing 3.4: `ORUSale::sendCarryOveredORU()`

Recommendation Restrict the privileged `sendCarryOveredORU()` function so that the privileged `owner` can only exercise it with the assurance that the remaining amount is sufficient to cover the claims by buyers!

Status This issue has been resolved as the team has abandoned the use of this contract.

3.4 Improved Logic to swap() in Arbitrager

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Arbitrager`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As a `fractional-algorithmic stablecoin`, the protocol has the constant need of converting one token to another. To facilitate the token swap need, the protocol provides the related `swap()` routines.

While examining these routines, we notice an implicit assumption that may be satisfied in the current deployment, but not necessarily be held in other deployments.

To elaborate, we show below the related `sellOusd()` routine. The current logic properly computes the expected return before making the swap call to the `ousdPair`. However, the swap call has implicitly assumed the deployment address of `oust` is numerically smaller than `usdc`. While this is indeed the case for the current deployment in `Astar Network`, the deployment on other networks may not be the case. To improve the portability, it is helpful to make necessary accommodation to remove the assumption. Note that another routine `buyOusd()` shares the same issue.

```
    function sellOusd() external onlyOwnerOrOperator {
        uint256 _rsvOusd = 0;
        uint256 _rsvUsdc = 0;
        if (address(ousd) <= address(collat)) {
            (_rsvOusd, _rsvUsdc, ) = ousdPair.getReserves();
        } else {
            (_rsvUsdc, _rsvOusd, ) = ousdPair.getReserves();
        }

        uint256 _ousdAmt = _calcOusdAmtToSell(_rsvUsdc, _rsvOusd);
        uint256 _usdcAmt = (_ousdAmt * SWAP_FEE_PRECISION) /
        (SWAP_FEE_PRECISION - swapFee) /
        MISSING_PRECISION;

        ousdPair.swap(
            0,
            _usdcAmt,
            address(this),
            abi.encode(FlashCallbackData({usdcAmt: _usdcAmt, isBuy: true}))
        );
    }
```

Listing 3.5: `Arbitrager::sellOusd()`

Recommendation Remove the above implicit assumption to make the protocol portable to other networks.

Status This issue has been resolved as the team clarifies that the `arbitrager` contract can be readily re-deployed, and reflected in the `bank` contract. Also, `arbitrager` is deployed only for existing pairs, so the team is certain about the order of tokens on the pair.

3.5 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 [2]

Description

In the Orcus protocol, there are some privileged accounts, i.e., owners and operator. These privileged accounts play critical roles in governing and regulating the system-wide operations (e.g., configure protocol parameters, execute privileged operations, etc.). Our analysis shows that these privileged accounts needs to be scrutinized. In the following, we use the BankStates contract as an example and show the representative functions potentially affected by the privileges of the owner account.

```
    function setTcrMovement(uint256 _tcrMovement) external onlyOwnerOrOperator {
        tcrMovement = _tcrMovement;
        emit LogSetTcrMovement(tcrMovement);
    }

    function setUpdatePeriod(uint32 _updatePeriod)
        external
        onlyOwnerOrOperator
    {
        updatePeriod = _updatePeriod;
        emit LogSetUpdatePeriod(updatePeriod);
    }

    function setTcrMin(uint256 _tcrMin) external onlyOwnerOrOperator {
        tcrMin = _tcrMin;
        emit LogSetTcrMin(_tcrMin);
    }

    function setEcrMin(uint256 _ecrMin) external onlyOwnerOrOperator {
        ecrMin = _ecrMin;
        emit LogSetEcrMin(ecrMin);
    }

    function toggleMintPaused() external onlyOwnerOrOperator {
        mintPaused = !mintPaused;
        emit LogToggleMintPaused(mintPaused);
    }

    function toggleRedeemPaused() external onlyOwnerOrOperator {
        redeemPaused = !redeemPaused;
        emit LogToggleRedeemPaused(redeemPaused);
    }
```

```
    function toggleZapMintPaused() external onlyOwnerOrOperator {
        zapMintPaused = !zapMintPaused;
        emit LogToggleZapMintPaused(zapMintPaused);
    }
```

Listing 3.6: Example Privileged Operations in `BankStates`

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 accounts 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 `Orcus Finance` design and implementation. `Orcus Finance` is an entirely decentralized and autonomous protocol with the native governance token which aims to be the first leading `Fractional-Algorithmic Stablecoin` issuer on the `Astar network` and to implement multiple useful financial tools and other synthetic assets in the ecosystem. The current code base is well structured and neatly organized. 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-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.