SMART CONTRACT AUDIT REPORT

for

Arbswap MasterChef

Prepared By: Xiaomi Huang

PeckShield

September 13, 2022

Document Properties

Client	Arbswap Protocol
Title	Smart Contract Audit Report
Target	Arbswap MasterChef
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	September 13, 2022	Luck Hu	Final Release
1.0-rc	September 6, 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

Given the opportunity to review the design document and related smart contract source code of the `Arbswap MasterChef` 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 the identified issues. This document outlines our audit results.

1.1 About Arbswap MasterChef

`Arbswap` is an automated market maker (`AMM`) and decentralized exchange (`DEX`) on `Arbitrum`. The audited `MasterChef` protocol provides the liquidity farming which enables to create either locked liquidity farms or flexible farms while allocating token rewards with a vesting contract. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Arbswap MasterChef

Item	Description
Name	Arbswap Protocol
Website	https://arbswap.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 13, 2022

In the following, we show the Git repositories of reviewed files and the commit hash value used in this audit. Note the audit scope only covers the `contracts/MasterChefV3.sol` and `contracts/VestingMaster.sol`.

- https://github.com/Arbswap-Official/Arbswap-MasterChef.git (5702f19)

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

- https://github.com/Arbswap-Official/Arbswap-MasterChef.git (781b95c)

1.2 About PeckShield

PeckShield Inc. [7] 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 [6]:

- 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) [5], 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 `Arbswap MasterChef` 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
Undetermined	0
Total	3

We have so far identified a list of potential issues. 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 2 low-severity vulnerabilities.

Table 2.1: Key Arbswap MasterChef Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incorrect totalLockedRewards Maintenance in claim()	Coding Practices	Fixed
PVE-002	Low	Improved Validation of Function Arguments	Coding Practices	Fixed
PVE-003	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 Incorrect totalLockedRewards Maintenance in claim()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `VestingMaster`
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

In the `VestingMaster` contract, there is a state variable, i.e., `totalLockedRewards`, that records the total rewards locked in the `VestingMaster` contact. When there are new rewards locked in the contract, the `totalLockedRewards` is increased with the new locked amount. Similarly, when the rewards are claimed from the contract, the `totalLockedRewards` is decreased by the claimed amount.

To elaborate, we show below the code snippet of the `VestingMaster::claim()` routine. As the name indicates, it is used for users to claim the locked rewards. While reviewing the `totalLockedRewards` update in this routine, we notice it is incorrectly increased with the claimed amount (line 106). As expected, the `totalLockedRewards` shall be decreased by the claimed amount.

```
94      function claim() external nonReentrant {
95          LockedReward[] storage lockedRewards = userLockedRewards[msg.sender];
96          uint256 currentTimestamp = block.timestamp;
97          LockedReward memory lockedReward;
98          uint256 claimableAmount;
99          for (uint256 i = 0; i < lockedRewards.length; i++) {
100             lockedReward = lockedRewards[i];
101             if (lockedReward.locked > 0 && currentTimestamp > lockedReward.timestamp) {
102                 claimableAmount += lockedReward.locked;
103                 delete lockedRewards[i];
104             }
105         }
106         totalLockedRewards += claimableAmount;
107         vestingToken.safeTransfer(msg.sender, claimableAmount);
108         emit Claim(msg.sender, claimableAmount);
```

```
    }
```

Listing 3.1: `VestingMaster::claim()`

Recommendation Decrease the `totalLockedRewards` by the claimed amount in the `claim()` routine.

Status The issue has been fixed by this commit: `ce1dd39`.

3.2 Improved Validation of Function Arguments

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `VestingMaster, MasterChefV3`
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

The liquidity farming from the `MasterChefV3` contract is locked into the `VestingMaster` contract which further distributes the rewards to users. The liquidity farming in the `MasterChefV3` is rewarded in `ARBS` token. So in the `VestingMaster` contract, the vesting token should also be `ARBS`.

While reviewing the validation of the input parameters in the `VestingMaster::constructor()` routine, we notice it simply validates `require(_vestingToken != address(0))`. Our analysis shows that it could be improved to `require(_vestingToken == ARBS)`.

```
    constructor(
        address _MasterChef,
        uint256 _period,
        uint256 _lockedPeriodAmount,
        address _vestingToken
    ) {
        require(_vestingToken != address(0), "VestingMaster::constructor: Zero address");
        require(_period > 0, "VestingMaster::constructor: Period zero");
        require(_lockedPeriodAmount > 0, "VestingMaster::constructor: Period amount zero");
        MasterChef = _MasterChef;
        vestingToken = IERC20(_vestingToken);
        period = _period;
        lockedPeriodAmount = _lockedPeriodAmount;
    }
```

Listing 3.2: `VestingMaster::constructor()`

What is more, the `MasterChefV3` contract provides both the flexible and locked liquidity farming functionalities. The `owner` can add new liquidity farms via the `MasterChefV3::add()` routine (showed as below). Specially, if it is a locked liquidity farm (`_lock == true`), the `_maxLockDuration /_minLockDuration` parameters are given to limit the max/min lock durations. However, current implementation does not properly validate the `_maxLockDuration/_minLockDuration` parameters. Our study shows that this routine could be improved by adding new validation `require(_minLockDuration > 0 && _maxLockDuration > _minLockDuration)`. Note the same improvement could also be applied to the `MasterChefV3::set()` routine.

```
    function add(
        uint256 _allocPoint,
        IERC20 _lpToken,
        bool _withUpdate,
        bool _lock,
        uint256 _maxLockDuration,
        uint256 _minLockDuration,
        uint256 _durationFactor,
        uint256 _boostWeight
    ) external onlyOwner {
        require(_allocPoint <= MaxAllocPoint, "add: too many alloc points!!");
        if (_lock) {
            require(
                _boostWeight >= MIN_BOOST_WEIGHT && _boostWeight <= MAX_BOOST_WEIGHT,
                "_boostWeight must be between MIN_BOOST_WEIGHT and MAX_BOOST_WEIGHT"
            );
            require(_durationFactor > 0, "_durationFactor can not be zero");
        }

        // ensure you can not add duplicate pools
        require(!LPTokenAdded[address(_lpToken)], "Pool already exists!!!!");
        LPTokenAdded[address(_lpToken)] = true;

        if (_withUpdate) {
            massUpdatePools();
        }

        uint256 lastRewardTime = block.timestamp > startTime ? block.timestamp :
            startTime;
        totalAllocPoint += _allocPoint;
        poolInfo.push(
            PoolInfo({
                lpToken: _lpToken,
                allocPoint: _allocPoint,
                lastRewardTime: lastRewardTime,
                accArbsPerShare: 0,
                lock: _lock,
                maxLockDuration: _maxLockDuration,
                minLockDuration: _minLockDuration,
                durationFactor: _durationFactor,
                boostWeight: _boostWeight,
```

```
                boostAmount: 0
            })
        );
    }
```

Listing 3.3: `MasterChefV3::add()`

Recommendation Revisit the above mentioned routines to add the proposed validations.

Status The issue has been fixed by this commit: `ce1dd39`. And the team will ensure the `_vestingToken` is exactly the `ARBS` address when they deploy the `VestingMaster`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `MasterChefV3`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `Arbswap MasterChef` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations. To elaborate, we show below the sensitive operations that are related to the `owner` account. Specifically, it has the authority to set the vesting address which is used to distribute users rewards, set the `arbsPerSecond` which is the `ARBS` reward rate to reward all the liquidity farming, add new pool, set pool weight/boostWeight which are used to accumulate the pool rewards, etc.

```
    function setVesting(address _vesting) external onlyOwner {
        // Vesing can be zero address when we do not need vesting.
        Vesting = IVestingMaster(_vesting);
        emit NewVesting(_vesting);
    }

    // Changes arbs token reward per second, with a cap of maxarbs per second
    // Good practice to update pools without messing up the contract
    function setArbsPerSecond(uint256 _arbsPerSecond, bool _withUpdate) external
        onlyOwner {
        require(_arbsPerSecond <= maxArbsPerSecond, "setArbsPerSecond: too many arbs!");

        // This MUST be done or pool rewards will be calculated with new arbs per second
        // This could unfairly punish small pools that dont have frequent deposits/
            withdraws/harvests
        if (_withUpdate) {
            massUpdatePools();
```

```
        }

        arbsPerSecond = _arbsPerSecond;
    }
```

Listing 3.4: Example Privileged Operations in `MasterChefV3.sol`

```
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate
    ) external onlyOwner {
        require(_allocPoint <= MaxAllocPoint, "add: too many alloc points!!");
        if (_withUpdate) {
            massUpdatePools();
        }

        totalAllocPoint = totalAllocPoint - poolInfo[_pid].allocPoint + _allocPoint;
        poolInfo[_pid].allocPoint = _allocPoint;
    }

    function set(
        uint256 _pid,
        bool _withUpdate,
        uint256 _maxLockDuration,
        uint256 _minLockDuration,
        uint256 _durationFactor,
        uint256 _boostWeight
    ) external onlyOwner {
        PoolInfo storage pool = poolInfo[_pid];
        require(pool.lock, "Not lock pool");
        require(
            _boostWeight >= MIN_BOOST_WEIGHT && _boostWeight <= MAX_BOOST_WEIGHT,
            "_boostWeight must be between MIN_BOOST_WEIGHT and MAX_BOOST_WEIGHT"
        );
        require(_durationFactor > 0, "_durationFactor can not be zero");
        if (_withUpdate) {
            massUpdatePools();
        }
        pool.maxLockDuration = _maxLockDuration;
        pool.minLockDuration = _minLockDuration;
        pool.durationFactor = _durationFactor;
        pool.boostWeight = _boostWeight;
    }
```

Listing 3.5: Example Privileged Operations in `MasterChefV3.sol`

It would be worrisome if the `owner` account is a plain EOA account. A multi-sig account could greatly alleviates this concern, though it is 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 for mitigation.

Recommendation Promptly transfer the `owner` privileges of the `MasterChefV3` contact to the intended governance contract. And 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 mitigated as the team confirmed they will use multi-sig wallet to control the owner.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Arbswap MasterChef` protocol. `Arbswap` is an automated market maker (`AMM`) and decentralized exchange (`DEX`) on `Arbitrum`. The audited `MasterChef` protocol provides the liquidity farming which enables to create either locked liquidity farms or flexible farms while allocating token rewards with a vesting contract. 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-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 CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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

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

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

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