SMART CONTRACT AUDIT REPORT

for

Arbswap Protocol

Prepared By: Xiaomi Huang

PeckShield

May 31, 2022

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	May 31, 2022	Xuxian Jiang	Final Release
1.0-rc	May 31, 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 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 Protocol

Arbswap is an automated market maker (AMM) and decentralized exchange (DEX) on Arbitrum that will increase adoption of the Ethereum's second-layer protocol by providing native liquidity and DeFi options for all users. As the first native AMM and DEX built solely for the Arbitrum network, Arbswap aims to become the central focal point of growth for the young blockchain, which is still in the onboarding phase of adopting new projects and DApps. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Arbswap Protocol

Item	Description
Name	Arbswap Protocol
Website	https://arbswap.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 31, 2022

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

- https://github.com/Arbswap-Official/Audit-Contracts.git (f066157)

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/Audit-Contracts.git (1eae8b4)

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

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) [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` 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	1	■
Undetermined	1	■
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, 1 low-severity vulnerability, and 1 undetermined issue.

Table 2.1: Key Arbswap Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Timely mint() in setArbsPerSecond()/setMintable()	Business Logic	Fixed
PVE-002	Low	Incorrect endTime Set in stopReward()	Business Logic	Fixed
PVE-003	Medium	Timely massUpdatePools() in MasterChef	Business Logic	Confirmed
PVE-004	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-005	Undetermined	Incompatibility With Deflationary/Rebasing Tokens	Business Logic	Confirmed

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 Timely mint() in setArbsPerSecond()/setMintable()

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: MirrorARBS
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The MirrorARBS contract is an ERC-20 token contract (xARBS) which provides an incentive mechanism that rewards the staking of ARBS with more ARBS. The rewards are carried out by allocating xARBS shares to users (for their depositing of ARBS) and distributing in a certain speed of arbsPerSecond. And the rewards for stakers are proportional to their share of xARBS tokens.

The arbsPerSecond can be dynamically updated via the setArbsPerSecond() routine. While analyzing the arbsPerSecond update in setArbsPerSecond(), we notice the need of timely invoking mint() to update the reward distribution before the new arbsPerSecond becomes effective.

```
    /**
     * @notice Sets withdraw fee
     * @dev Only callable by the contract admin.
     */
    function setArbsPerSecond(uint256 _arbsPerSecond) external onlyAdmin {
        require(_arbsPerSecond <= MaxArbsPerSecond, "arbsPerSecond cannot be more than
            MaxArbsPerSecond");
        arbsPerSecond = _arbsPerSecond;
    }
```

Listing 3.1: MirrorARBS::setArbsPerSecond()

If the call to mint() is not immediately invoked before updating the arbsPerSecond, the amount of the ARBS rewards will be incorrect when the mint() is invoked next time. The reason is that the ARBS rewards shall be distributed per the old arbsPerSecond before it is updated.

Note the Mintable can also be updated dynamically via the setMintable() routine. Therefore, there's also a need to timely invoking mint() before the new Mintable becomes effective.

Recommendation Timely invoke mint() in the above mentioned setArbsPerSecond()/setMintable () routines before the new configuration becomes effective.

Status The issue has been fixed by this commit: 044070b.

3.2 Incorrect endTime Set in stopReward()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: XARBSPool
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The XARBSPool contract provides an incentive mechanism that rewards the staking of xARBS with the configured rewards tokens. The rewards are carried out by designating a number of staking pools into which xARBS can be staked. Each pool defines a dedicated reward token and the start/end time of the reward period. The staking users are rewarded in proportional to their share of xARBS tokens in the reward pool.

To elaborate, we show below the code snippets of the stopReward() routine. As the name indicates, the routine is designed to stop the reward pool given by the _pid parameter. The reward pool is stopped by updating the endTime of the pool. While analyzing the endTime update in this routine, we notice it sets the endTime to current block.number, NOT the block.timestamp. As a result, the rewards that should have been distributed before the pool is stopped cannot be accumulated to the pool any more.

```
    function stopReward(uint256 _pid) external onlyOwner {
        poolInfo[_pid].endTime = block.number;
    }
```

Listing 3.2: XARBSPool::stopReward()

With that, it's suggested to stop the pool by setting its endTime to current block.timestamp.

Recommendation Revise the above mentioned stopReward() routine to update the endTime to current block.timestamp.

Status The issue has been fixed by this commit: 044070b.

3.3 Timely massUpdatePools() in MasterChef

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: MasterChef
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The MasterChef contract provides an incentive mechanism that rewards the staking of supported assets (LP tokens) with the ARBS tokens. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The reward pools can be dynamically added via add() and the weights of supported pools can be adjusted via set(). When analyzing the pool weight update routine set(), we notice the need of timely invoking massUpdatePools() to update the reward distribution before the new pool weight becomes effective.

```
    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;
    }
```

Listing 3.3: MasterChef::set()

If the call to massUpdatePools() is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, these interfaces are restricted to the owner (via the onlyOwner modifier), which greatly alleviates the concern.

Similarly, the reward rate (arbsPerSecond) can also be dynamically updated via the setArbsPerSecond () routine. When analyzing the reward rate update in setArbsPerSecond(), we notice the need of timely invoking massUpdatePools() to update the reward distribution before the new reward rate becomes effective.

```
    // 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: MasterChef::setArbsPerSecond()

Recommendation Timely invoke `massUpdatePools()` when any pool's weight or the reward rate has been updated. In fact, the `_withUpdate` parameter to the `set()`, `add()` and `setArbsPerSecond()` routines can be simply ignored or removed.

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

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

Listing 3.5: Revised MasterChef::set()

Status This issue has been confirmed. The `Arbswap` team clarified that they will keep the parameter `_withUpdate`, and set it to `true` when it's needed to add or update.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the Arbswap protocol, there is a privileged owner account that plays a critical role in governing and regulating the system-wide operations (e.g., set Miner who can mint ARBS tokens). In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
    /**
    * @notice Mint by Miner
    * @param _to: _to
    * @param _amount: _amount
    * @dev Only callable by the Miner.
    */
   function mintByMiner(address _to, uint256 _amount) external onlyMiner {
       require(minerTotalSupply < MinerMaxSupply, "Exceeded maximum supply");
       uint256 mintAmount = (minerTotalSupply + _amount) < MinerMaxSupply
           ? _amount
           : (MinerMaxSupply - minerTotalSupply);
       minerTotalSupply += mintAmount;
       _mint(_to, mintAmount);
       emit MintByMiner(msg.sender, _to, mintAmount);
   }

    /**
    * @notice Sets Miner
    * @param _miner: _miner
    * @dev Only callable by the contract owner.
    */
   function setMiner(address _miner) external onlyOwner {
       require(_miner != address(0), "Cannot be zero address");
       Miner = _miner;
       emit NewMiner(_miner);
   }
```

Listing 3.6: Example Privileged Operations in ArbswapToken.sol

```
    /**
    * @notice Sets admin address
    * @dev Only callable by the contract owner.
    */
   function setAdmin(address _admin) external onlyOwner {
       require(_admin != address(0), "Cannot be zero address");
```

```
    admin = _admin;
}

/**
 * @notice Sets treasury address
 * @dev Only callable by the contract owner.
 */
function setTreasury(address _treasury) external onlyOwner {
    require(_treasury != address(0), "Cannot be zero address");
    treasury = _treasury;
}
```

Listing 3.7: Example Privileged Operations in `MirrorARBS.sol`

There are still other privileged routines not listed here. And 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.

Recommendation Making the above privileges explicit among protocol users.

Status This issue has been confirmed by the team. And the team clarifies that they will use timelock and multi-sig wallet to control the owner role.

3.5 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-005
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `MasterChef`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the `Arbswap` protocol, the `MasterChef` contract rewards users depositing of the supported assets with `ARBS` tokens. In particular, one entry routine, i.e., `deposit()`, accepts user deposits of supported assets (e.g., `pool.lpToken`). Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract. In the following, we show the `deposit()` routine that is used to deposit `pool.lpToken` to the `MasterChef` contract.

```
    // Deposit LP tokens to MasterChef for ARBS allocation.
```

```
    function deposit(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];

        updatePool(_pid);

        uint256 pending = (user.amount * pool.accArbsPerShare) / 1e12 - user.rewardDebt;

        user.amount += _amount;
        user.rewardDebt = (user.amount * pool.accArbsPerShare) / 1e12;

        if (pending > 0) {
            safeArbsTransfer(msg.sender, pending);
        }
        pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);

        emit Deposit(msg.sender, _pid, _amount);
    }
```

Listing 3.8: `MasterChef::deposit()`

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these asset-transferring routines. In other words, the above operations, such as `deposit()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the contract before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into Arbswap Protocol for depositing. In fact, `Arbswap` is indeed in the position to effectively regulate the set of assets that can be listed. Meanwhile, there exist certain assets that may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation If current codebase needs to support deflationary tokens, it is necessary to check the balance before and after the `transfer()`/`transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted `USDT`.

Status This issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Arbswap` protocol, which is an automated market maker (`AMM`) and decentralized exchange (`DEX`) on `Arbitrum` that will increase adoption of the `Ethereum` second-layer protocol by providing native liquidity and `DeFi` options for all users. 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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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