SMART CONTRACT AUDIT REPORT

for

AirSwap Protocol (v4.2)

Prepared By: Xiaomi Huang

PeckShield

January 30, 2024

Document Properties

Client	AirSwap Protocol
Title	Smart Contract Audit Report
Target	AirSwap
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0.1	January 30, 2024	Xuxian Jiang	Post-Final Release #1
1.0	January 25, 2024	Xuxian Jiang	Final Release
1.0-rc	January 23, 2024	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 source code of the `AirSwap(v4.2)` 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 could potentially be improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About AirSwap

`AirSwap` curates a peer-to-peer network for trading digital assets. The protocol is designed to protect traders from counterparty risk, price slippage, and front running. Any market participant can discover others and trade directly peer-to-peer. At the protocol level, each swap is between two parties, a signer and a sender. The signer is the party that creates and cryptographically signs an order, and the sender is the party that sends the order to an EVM-compatible blockchain for settlement. As a decentralized and open project, governance and community activities are also supported by rewards protocols built with on-chain components. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of AirSwap

Item	Description
Name	AirSwap Protocol
Website	https://www.airswap.io/
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	January 30, 2024

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that the repository has a number of contracts and this audit covers the following contracts: `SwapERC20.sol`, `Swap.sol`, `ERC20Adapter.sol`, `ERC721Adapter.sol`, `ERC1155Adapter.sol`, `Staking.sol`, and `Registry.sol` under the `source/` directory.

- https://github.com/airswap/airswap-protocols.git (bcd4c48)

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

- https://github.com/airswap/airswap-protocols.git (6ede424)

1.2 About PeckShield

PeckShield Inc. [5] 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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [4]:

- 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 accordingly classified into four categories, 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 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) [3], 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 1.3: 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

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.

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 design and implementation of the `AirSwap (v4.2)` protocol smart contracts. 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	0	
Low	1	■
Informational	2	■ ■
Total	3	

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 may involve 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 1 low-severity vulnerability and 2 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Repeated available() Calculation Avoidance in Staking	Coding Practices	Resolved
PVE-002	Informational	Revisited Staking Duration Update Logic in Staking	Business Logic	Resolved
PVE-003	Informational	Improved Order Validation Logic in Swap/SwapERC20	Business Logic	Resolved

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 Repeated available() Calculation Avoidance in Staking

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Staking`
- Category: Business Logic [2]
- CWE subcategory: CWE-841 [1]

Description

The `AirSwap` protocol curates a peer-to-peer network for trading digital assets and it is designed to protect traders from counterparty risk, price slippage, and front running. It also has a built-in staking support to facilitate user participation. While examining the current unstaking logic, we notice it makes repeated calculation of `available` staked amount to withdraw for a given account.

In the following, we show below the related implementation. We notice it makes the `available()` twice to query the available (staked) amount to withdraw for a given account. And between the two calls, there are no state changes. With that, we suggest to cache the first result and simply reuse the result instead of making a repeated calculation.

```
function _unstake(address _account, uint256 _amount) private {
  Stake storage _selected = stakes[_account];
  if (_amount > available(_account)) revert AmountInvalid(_amount);
  uint256 nowAvailable = available(_account);
  _selected.balance = _selected.balance - _amount;
  _selected.timestamp = Math.min(
    block.timestamp -
      (((10000 - ((10000 * _amount) / nowAvailable)) *
        (block.timestamp - _selected.timestamp)) / 10000),
    _selected.maturity
  );
  stakingToken.safeTransfer(_account, _amount);
  emit Transfer(_account, address(0), _amount);
}
```

Listing 3.1: `Staking::_unstake()`

Recommendation Improve the above `_unstake()` routine to avoid repeated calls of `available()`.

Status The issue has been fixed with the following commit: `d16d85c`.

3.2 Revisited Staking Duration Update Logic in Staking

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Staking`
- Category: Business Logic [2]
- CWE subcategory: CWE-841 [1]

Description

The `Staking` contract supports the dynamic change of the staking duration. It basically specifies the scheduled duration change time and then updates with the intended staking duration. Our analysis shows this logic can be improved by introducing a new storage state, i.e., `proposedStakingDuration`, to record the intended new staking duration ahead, instead of providing it at the very moment when it is actually used.

Specifically, we show below the related routines, i.e., `scheduleDurationChange()` and `setDuration()`. The first routine indicates the duration change time and the second routine actually sets the new duration time. However, the actual duration is not known until the second routine when it is actually enforced. This may be inconvenient for existing stakers.

```
function scheduleDurationChange(uint256 _delay) external onlyOwner {
  if (activeDurationChangeTimestamp != 0) revert TimelockActive();
  if (_delay < minDurationChangeDelay) revert DelayInvalid(_delay);
  activeDurationChangeTimestamp = block.timestamp + _delay;
  emit ScheduleDurationChange(activeDurationChangeTimestamp);
}
```

Listing 3.2: `Staking::scheduleDurationChange()`

```
function setDuration(uint256 _stakingDuration) external onlyOwner {
  if (_stakingDuration == 0) revert DurationInvalid(_stakingDuration);
  if (activeDurationChangeTimestamp == 0) revert TimelockInactive();
  if (block.timestamp < activeDurationChangeTimestamp) revert Timelocked();
  stakingDuration = _stakingDuration;
  delete activeDurationChangeTimestamp;
  emit CompleteDurationChange(_stakingDuration);
}
```

Listing 3.3: `Staking::setDuration()`

Recommendation Revisit the above logic to update the staking duration time to notify ahead current stakers.

Status This issue has been confirmed.

3.3 Improved Order Validation Logic in Swap/SwapERC20

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Swap`, `SwapERC20`
- Category: Business Logic [2]
- CWE subcategory: CWE-841 [1]

Description

In the audited `Swap` contracts, there is a helper routine `check()` that is designed to check the given order and returns list of errors. Our analysis this helper routine can be improved.

To elaborate, we show below the related `check()` routine. By validating the given order, this routine returns a tuple of error count as well as `bytes32[] memory array` of error messages. And the validation can be improved by adjusting the `nonceUsed` check (line 299) as well as the `signatoryMinimumNonce` validation (line 303) as part of the `else`-branch (line 293). The reason is that they only need to be checked when all previous checks fail (within the `else`-branch at line 286).

```
263   function check(
264     address senderWallet,
265     Order calldata order
266   ) external view returns (bytes32[] memory, uint256) {
267     uint256 errCount;
268     bytes32[] memory errors = new bytes32[](MAX_ERROR_COUNT);
269     (address signatory, ) = ECDSA.tryRecover(
270       _getOrderHash(order),
271       order.v,
272       order.r,
273       order.s
274     );

276     if (
277       order.sender.wallet != address(0) && order.sender.wallet != senderWallet
278     ) {
279       errors[errCount] = "SenderInvalid";
280       errCount++;
281     }

283     if (signatory == address(0)) {
284       errors[errCount] = "SignatureInvalid";
285       errCount++;
```

```
    } else {
      if (
        authorized[order.signer.wallet] != address(0) &&
        signatory != authorized[order.signer.wallet]
      ) {
        errors[errCount] = "SignatoryUnauthorized";
        errCount++;
      } else if (
        authorized[order.signer.wallet] == address(0) &&
        signatory != order.signer.wallet
      ) {
        errors[errCount] = "Unauthorized";
        errCount++;
      } else if (nonceUsed(signatory, order.nonce)) {
        errors[errCount] = "NonceAlreadyUsed";
        errCount++;
      }
      if (order.nonce < signatoryMinimumNonce[signatory]) {
        errors[errCount] = "NonceTooLow";
        errCount++;
      }
    }
    ...
  }
```

Listing 3.4: `Swap::check()`

Recommendation Improve the above `check()` logic to adjust the `nonceUsed` and `signatoryMinimumNonce` checks. Note the same improvement can be applied to `SwapERC20` as well.

Status The issue has been fixed with the following commit: `bcd4c48`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `AirSwap (v4.2)` protocol, which curates a peer-to-peer network for trading digital assets. The protocol is designed to protect traders from counterparty risk, price slippage, and front running. Any market participant can discover others and trade directly peer-to-peer. 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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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