SMART CONTRACT AUDIT REPORT

for

AirSwap Protocol (v4)

Prepared By: Xiaomi Huang

PeckShield

March 3, 2023

Document Properties

Client	AirSwap Protocol
Title	Smart Contract Audit Report
Target	AirSwap
Version	1.0
Author	Xuxian Jiang
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Patrick Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 3, 2023	Xuxian Jiang	Final Release
1.0-rc	February 18, 2023	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 4

1.1 About AirSwap . . . 4

1.2 About PeckShield . . . 5

1.3 Methodology . . . 5

1.4 Disclaimer . . . 7

2 Findings 9

2.1 Summary . . . 9

2.2 Key Findings . . . 10

3 Detailed Results 11

3.1 Improved Order Hash Generation And Verification in Swap . . . 11

3.2 Inconsistent Swap Protocol Fee Collection . . . 12

3.3 Trust Issue of Admin Keys . . . 14

4 Conclusion 16

References 17

1 Introduction

Given the opportunity to review the design document and related source code of the `AirSwap(v4)` 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	March 3, 2023

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

- https://github.com/airswap/airswap-protocols/commit/231cc79cdfa8d4dc4c6cf3152cc9a1c0802bcc86

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 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

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. 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 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 design and implementation of the `AirSwap (v4)` 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	1	■
Low	2	■ ■
Informational	0	
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 medium-severity vulnerability and 2 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Order Hash Generation And Verification in Swap	Coding Practices	Design Choice
PVE-002	Low	Inconsistent Swap Protocol Fee Collection	Business Logic	Design Choice
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 Improved Order Hash Generation And Verification in Swap

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

Description

As mentioned earlier, `AirSwap` 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. Naturally, the swap logic requires proper verification on the user-provided signatures. While examining the current hash generation in the signature verification, we notice the current hash generation can be improved.

In the following, we show below the related hash generation implementation. It comes to our attention that the resulting hash value is computed from a number of fields and one specific field is the current protocol fee (line 467). While it is typically the case that the current `protocolFee` needs to be equal to the order-included `order.protocolFee`, the current validation check however does not explicitly enforce it.

```
456   function _getOrderHash(Order calldata order) internal view returns (bytes32) {
457     return
458       keccak256(
459         abi.encodePacked(
460           "\x19\x01", // EIP191: Indicates EIP712
461           DOMAIN_SEPARATOR,
462           keccak256(
463             abi.encode(
464               ORDER_TYPEHASH,
465               order.nonce,
466               order.expiry,
467               protocolFee,
468               keccak256(abi.encode(PARTY_TYPEHASH, order.signer)),
```

```
                keccak256(abi.encode(PARTY_TYPEHASH, order.sender)),
                order.affiliateWallet,
                order.affiliateAmount
              )
            )
          )
        );
  }
```

Listing 3.1: `Swap::_getOrderHash()`

Recommendation Improve the above hash generation with the order-specific `protocolFee`. Also validate the order to ensure the order-specific `protocolFee` is the same as the protocol-wide `protocolFee`.

Status This issue has been communicated and the team confirms it is not a vulnerability. The contract is the authority for the `protocolFee` and this value needs to be used. The order in itself does not have a fee property, rather, the `protocolFee` is hashed in each order meaning that any change in the fee would lead to an invalid signature.

3.2 Inconsistent Swap Protocol Fee Collection

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Swap`, `SwapERC20`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The `AirSwap` protocol has two swapped-related contracts: `Swap` and `SwapERC20`. By design, the first one is used to facilitate the atomic swap settlement in OTC and the second one facilities the atomic swap settlement for ERC20 tokens. Each swap may incur certain protocol fee. Our analysis shows the protocol fee logic is inconsistent.

Specifically, we show below the protocol fee collection in these two contracts. The `Swap` contract collects the protocol fee from the sender while the `SwapERC20` contract collects the protocol fee from the signer. The inconsistency may bring unnecessary confusion to the trading users.

```
    // Transfer protocol fee from sender if possible
    uint256 protocolFeeAmount = (order.sender.amount * protocolFee) /
      FEE_DIVISOR;
    if (protocolFeeAmount > 0) {
      _transfer(
        order.sender.wallet,
```

```
      protocolFeeWallet,
      protocolFeeAmount,
      order.sender.id,
      order.sender.token,
      order.sender.kind
    );
  }

  // Transfer royalty from sender if supported by signer token
  if (supportsRoyalties(order.signer.token)) {
    address royaltyRecipient;
    uint256 royaltyAmount;
    (royaltyRecipient, royaltyAmount) = IERC2981(order.signer.token)
      .royaltyInfo(order.signer.id, order.sender.amount);
    if (royaltyAmount > 0) {
      if (royaltyAmount > maxRoyalty) revert RoyaltyExceedsMax(royaltyAmount);
      _transfer(
        order.sender.wallet,
        royaltyRecipient,
        royaltyAmount,
        order.sender.id,
        order.sender.token,
        order.sender.kind
      );
    }
  }
```

Listing 3.2: `Swap::swap()`

```
function swap(
  address recipient,
  uint256 nonce,
  uint256 expiry,
  address signerWallet,
  address signerToken,
  uint256 signerAmount,
  address senderToken,
  uint256 senderAmount,
  uint8 v,
  bytes32 r,
  bytes32 s
) external override {
  // Ensure the order is valid for signer and sender
  _check(
    nonce,
    expiry,
    signerWallet,
    signerToken,
    signerAmount,
    msg.sender,
    senderToken,
    senderAmount,
    v,
```

```
        r,
        s
      );

      // Transfer token from sender to signer
      IERC20(senderToken).safeTransferFrom(
        msg.sender,
        signerWallet,
        senderAmount
      );

      // Transfer token from signer to recipient
      IERC20(signerToken).safeTransferFrom(signerWallet, recipient, signerAmount);

      // Calculate and transfer protocol fee and any rebate
      _transferProtocolFee(signerToken, signerWallet, signerAmount);
```

Listing 3.3: `SwapERC20::swap()`

Recommendation Revisit the above protocol fee logic to ensure the consistency between `Swap` and `SwapERC20`.

Status This issue has been communicated and the team confirms it is not a vulnerability. `SwapERC20` and `Swap` have distinct purposes leading to necessary differences in logic. The use case for `SwapERC20` (client-server RFQ) is simpler for signers to be concerned with fees instead of senders. The use case for `Swap (NFT OTC)` is for signers to sell `NFTs`, so `protocolFees` could only actually be assessed on the sender side.

3.3 Trust Issue of Admin Keys

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

Description

In the `AirSwap` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and payee adjustment). It also has the privilege to regulate or govern the flow of assets within the protocol.

With great privilege comes great responsibility. Our analysis shows that the `owner` account is indeed privileged. In the following, we show representative privileged operations in the `Pool` protocol.

```
  function setStakingToken(address _stakingToken) external override onlyOwner {
    require(_stakingToken != address(0), "INVALID_ADDRESS");
    // set allowance on old staking token to zero
    IERC20(stakingToken).safeApprove(stakingContract, 0);
    stakingToken = _stakingToken;
    IERC20(stakingToken).safeApprove(stakingContract, 2 ** 256 - 1);
  }

  /**
   * @notice Admin function to migrate funds
   * @dev Only owner
   * @param tokens address[]
   * @param dest address
   */
  function drainTo(
    address[] calldata tokens,
    address dest
  ) external override onlyOwner {
    for (uint256 i = 0; i < tokens.length; i++) {
      uint256 bal = IERC20(tokens[i]).balanceOf(address(this));
      IERC20(tokens[i]).safeTransfer(dest, bal);
    }
    emit DrainTo(tokens, dest);
  }
```

Listing 3.4: Various Privileged Operations in `Pool`

We emphasize that the privilege assignment with various protocol contracts is necessary and required for proper protocol operations. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure.

We point out that a compromised `owner` account would allow the attacker to invoke the above `drainTo` to steal funds in current protocol, which directly undermines the assumption of the `AirSwap` protocol.

Recommendation Promptly transfer the privileged account 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 and partially mitigated with a multi-sig account to regulate the governance privileges. The multi-sig account is a standard Gnosis Safe wallet, which is controlled by multiple participants, who agree to propose and submit transactions as they relate to the DAO's proposal submission and voting mechanism. This avoids risk of any single compromised EOA as it would require collusion of multiple participants.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `AirSwap` 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-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.