SMART CONTRACT AUDIT REPORT

for

Cross Swap

Prepared By: Xiaomi Huang

PeckShield

February 22, 2023

Document Properties

Client	Rhinofi
Title	Smart Contract Audit Report
Target	Cross Swap
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 22, 2023	Jing Wang	Final Release
1.0-rc	February 2, 2023	Jing Wang	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 `Cross Swap` contract, 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 Cross Swap

The `Cross Swap` contract is a smart-wallet that holds funds on behalf of multiple users. This smart-wallet will be deployed on multiple chains, including `Polygon`, `Arbitrum`, and `BSC`. The smart-wallet allows meta transactions to be broadcasted to complete certain generic swap actions on users' behalf when they provide a valid signature. The architecture is flexible, allowing any number of swaps to be completed involving one or multiple input and output tokens. The basic information of `Cross Swap` is as follows:

Table 1.1: Basic Information of Cross Swap

Item	Description
Name	Rhinofi
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 22, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that this audit does not cover the `weiroll` library:

- https://github.com/rhinofi/contracts_public (7b94f37e)

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

- https://github.com/rhinofi/contracts_public (74751d31)

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 the 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 checklist of 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

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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

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) [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. 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 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 `Cross Swap` 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	1	■
Low	1	■
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 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 1 high-severity vulnerability, 1 medium-severity vulnerability and 1 low-severity vulnerability.

Table 2.1: Key Audit Findings of Cross Swap

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Bypassed withdrawalDelay For Emergency Withdrawal	Business Logic	Mitigated
PVE-002	High	Missing Sanity Check For performSwap()	Time and State	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 Possible Bypassed withdrawalDelay For Emergency Withdrawal

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: UserWallet
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In the Cross Swap protocol, users can deposit their assets into the contract and withdraw by signature or directly calling the settleEmergencyWithdrawal() routine. However, the second approach requires the users to first call the requestEmergencyWithdrawal() routine, and after the withdrawalDelay time period passed, they can then retrieve the funds. While reviewing this related implementation, we notice the withdrawalDelay time period could be bypassed. To elaborate, we show the related routines from the UserWallet contract.

```
    function requestEmergencyWithdrawal(address _token) external {
        emergencyWithdrawalRequests[msg.sender][_token] = block.timestamp;
        emit LogEmergencyWithdrawalRequested(msg.sender, _token);
    }

    function settleEmergencyWithdrawal(address _token) external {
        address sender = msg.sender;
        {
          uint256 requestTimestamp = emergencyWithdrawalRequests[sender][_token];
          emergencyWithdrawalRequests[sender][_token] = 0;
          require(requestTimestamp > 0, "EMERGENCY_WITHDRAWAL_NOT_REQUESTED");
          require(requestTimestamp + withdrawalDelay < block.timestamp, "
              EMERGENCY_WITHDRAWAL_STILL_IN_PROGRESS");
        }
        {
          uint256 balance = userBalances[sender][_token];
```

```
                _withdraw(sender, _token, balance, sender);
            }
            emit LogEmergencyWithdrawalSettled(sender, _token);
        }
```

Listing 3.1: UserWallet::requestEmergencyWithdrawal()and settleEmergencyWithdrawal

In particular, the current implementation does not handle withdrawalDelay properly when users deposit, withdraw and transfer the funds. As a result, the current logic is vulnerable and a bad actor can prepare accounts which are already qualified for settleEmergencyWithdrawal() by calling the requestEmergencyWithdrawal() routine even without any funds in the contract. After the withdrawalDelay time period is passed, the prepared accounts are able to deposit and withdraw in the same block.

Recommendation To mitigate, add necessary logic to handle withdrawalDelay.

Status The issue has been mitigated by the team. And the team adds clarifies that they had prevented users from sending tokens to an already started emergency withdrawal account.

3.2 Missing Sanity Check For performSwap()

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: Swap
- Category: Time and State [5]
- CWE subcategory: CWE-362 [2]

Description

As mentioned before, the Cross Swap contract is designed to complete generic swap actions on behalf of a user by allowing any number of swaps to be completed that may involve one or multiple input and output tokens. In order to achieve the goal, the contract uses a helper contract that will take care of the transactions involved to transition from the in tokens to the out tokens. The Swap contract only checks the delta balance changes of in tokens and the out tokens after the swap meets the list of constraints from the user. To elaborate, we show below the performSwap() routine.

```
    function performSwap(
        address user,
        address tokenFrom,
        address tokenTo,
        uint256 amountFrom,
        uint256 minAmountTo,
        uint256 maxFeeTo,
        bytes calldata data
```

```
257     ) private returns(uint256 tokenFromAmount, uint256 amountToUser, uint256 amountToFee ) {
258         tokenFromAmount = _contractBalance(IERC20Upgradeable(tokenFrom));
259         // Using amountToUser name all the way in order to use a single variable
260         amountToUser = _contractBalance(IERC20Upgradeable(tokenTo));
261
262         // Only approve one token for the max amount
263         IERC20Upgradeable(tokenFrom).safeApprove(paraswapTransferProxy, amountFrom);
264         // Do swap
265         // Arbitary call, must validate the state after
266         safeExecuteOnParaswap(data);
267
268         // After swap, reuse variables to save stack space
269         tokenFromAmount = tokenFromAmount - _contractBalance(IERC20Upgradeable(tokenFrom ));
270         amountToUser = _contractBalance(IERC20Upgradeable(tokenTo)) - amountToUser;
271
272         require(tokenFromAmount <= amountFrom, "HIGHER_THAN_AMOUNT_FROM");
273         require(amountToUser >= minAmountTo, "LOWER_THAN_MIN_AMOUNT_TO");
274
275         ...
276     }
```

Listing 3.2: `Swap::performSwap()`

We notice that there is a check `require(amountToUser >= minAmountTo, "LOWER_THAN_MIN_AMOUNT_TO")` (line 273), which is used to guarantee the balance of `tokenTo` in this contract is increased. However, due to the missing protection of the `non-reentrancy` guard, a malicious actor could reentry into the `deposit()` routine to increase the `tokenTo` balance to bypass this check. With that, we suggest adding necessary sanity checks to ensure `_contractBalance(tokenTo)>= tokenReservers[tokenTo]` or adding `non-reentrancy` guard to the related routines.

Recommendation Add necessary sanity checks to the above `performSwap()` routine or add the `non-reentrancy` guard to the related routines.

Status The issue has been fixed by this commit: `74751d3`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: High
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the Cross Swap protocol, there is a special administrative account, i.e., admin. This admin account plays a critical role in governing and regulating the protocol-wide operations (e.g., roles and parameters configurations). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged admin account and its related privileged accesses in current contract.

To elaborate, we show the related routine from the UserWallet contract. This routine allows the OPERATOR_ROLE account, which could be configured by the admin account, to set parameter withdrawalDelay, which play important role for withdrawing funds.

```
    function setEmergencyWithdrawalDelay(uint256 delay) external onlyRole(OPERATOR_ROLE)
        {
        require(delay <= MAX_WITHDRAWAL_DELAY, 'WITHDRAWAL_DELAY_OVER_MAX');
        withdrawalDelay = delay;
    }
```

Listing 3.3: UserWallet::setEmergencyWithdrawalDelay()

We understand the need of the privileged functions for contract maintenance, but it is worrisome if the privileged admin account is a plain EOA account. 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.

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 mitigated. The team have changed the role of the function setEmergencyWithdrawalDelay to DEFAULT_ADMIN_ROLE, that way the change would go through the TimeLockController.

4 Conclusion

In this audit, we have analyzed the `Cross Swap` design and implementation. The `Cross Swap` contract is a smart-wallet that holds funds on behalf of multiple users and allows meta transactions to be broadcast to complete certain generic swap actions from users. 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-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.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: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.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.