SMART CONTRACT AUDIT REPORT

for

ParaSwap DirectSwap

Prepared By: Xiaomi Huang

PeckShield

June 8, 2023

Document Properties

Client	ParaSwap
Title	Smart Contract Audit Report
Target	ParaSwap DirectSwap
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Jing Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 8, 2023	Jing Wang	Final Release
1.0-rc	May 29, 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 ParaSwap DirectSwap protocol design document and related smart contract source code, 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 branch of ParaSwap DirectSwap protocol 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 ParaSwap

ParaSwap is a platform that simplifies user interactions with decentralized finance (DeFi) by aggregating decentralized exchanges and other services into one interface. It abstracts the complexity of swaps and optimizes gas usage through audited contracts include DirectSwap and FeeModel. These contracts are integral to the ParaSwap aggregation protocol. Additionally, the DistributorController enhances the current staking protocol by enabling secure distributions and delegation of distribution to other roles. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of ParaSwap DirectSwap

Item	Description
Name	ParaSwap
Website	https://www.paraswap.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 8, 2023

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

- https://github.com/paraswap/paraswap-staking/pull/23
- https://github.com/paraswap/paraswap-contracts/pull/143

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

- https://github.com/paraswap/paraswap-contracts (80243bf)
- https://github.com/paraswap/paraswap-staking (f891aa4)

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 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 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) [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.

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 1.3: 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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `ParaSwap DirectSwap` protocol implementation. 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	0	
Medium	1	■
Low	3	■ ■ ■
Informational	0	
Total	4	

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 that 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 3 low-severity vulnerabilities.

Table 2.1: Key ParaSwap DirectSwap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Logic Of afterBuy()	Business Logic	Fixed
PVE-002	Low	Implicit Assumption on Fee Version	Business Logic	Confirmed
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-004	Low	Safe-Version Replacement With And safeTransferFrom()	Coding Practices	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 Improved Logic Of afterBuy()

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

Description

As mentioned earlier, the `DirectSwap` contract in `ParaSwap` is designed to improve gas efficient for single route swaps. It implements five new functions to handle single route swaps with better gas efficiency. For each of the swaps, there are common functions, e.g., `afterSell()` and `afterBuy()`. In this section, we examine the `afterBuy()` routine that is designed to process leftovers tokens and deliver the fees. While reviewing the current implementation, we notice the handling of leftover tokens can be improved.

To elaborate, we show below the related code snippet of the `afterBuy()` function. This function supports the fee-related handling in various routines. These routines take the leftover amounts and deliver them to the user and the fee receiver. Since the variable `receivedAmount` is computed as `receivedAmount = Utils.tokenBalance(toToken, address(this))` (lines 427), it is more accurate to use the `receivedAmount` rather than `toAmount` to handle the leftover tokens delivery.

```
417 function afterBuy(
418     address fromToken,
419     address toToken,
420     uint256 fromAmount,
421     uint256 toAmount,
422     address payable beneficiary,
423     uint256 feePercent,
424     address payable partner,
425     uint256 expectedAmount
426 ) private returns (uint256 amountIn, uint256 receivedAmount) {
427     receivedAmount = Utils.tokenBalance(toToken, address(this));
```

```
    require(receivedAmount >= toAmount, "Received amount of tokens are less then expected");
    uint256 remainingAmount = Utils.tokenBalance(fromToken, address(this));
    amountIn = fromAmount.sub(remainingAmount);

    {
        if (
            _getFixedFeeBps(partner, feePercent) != 0 &&
            !_isTakeFeeFromSrcToken(feePercent) &&
            !_isReferral(feePercent)
        ) {
            takeToTokenFeeAndTransfer(toToken, toAmount, beneficiary, partner, feePercent);
            Utils.transferTokens(fromToken, beneficiary, remainingAmount);
        } else {
            Utils.transferTokens(toToken, beneficiary, toAmount);
            if (_getFixedFeeBps(partner, feePercent) != 0 && _isTakeFeeFromSrcToken(feePercent)) {
                // take fee from source token and transfer remaining token back to beneficiary
                takeFromTokenFeeAndTransfer(fromToken, amountIn, remainingAmount, beneficiary, partner, feePercent);
            } else if (amountIn < expectedAmount) {
                takeSlippageAndTransferBuy(
                    fromToken,
                    beneficiary,
                    partner,
                    expectedAmount,
                    amountIn,
                    remainingAmount,
                    feePercent
                );
            } else {
                Utils.transferTokens(fromToken, beneficiary, remainingAmount);
            }
        }
    }
}
```

Listing 3.1: `DirectSwap::afterBuy()`

Recommendation Improved the handling the leftover token amounts more accurate in the `afterBuy()` routine.

Status This issue has been fixed.

3.2 Implicit Assumption on Fee Version

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LeftoversExtension`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, the `FeeModel` contract in `ParaSwap` handles the fee collection logic. In this section, we examine the `FeeModel` for fee and slippage distribution and notice the fee version handling may be improved.

To elaborate, we show below the related code snippet of the `_calcSlippageFees()` function. This function supports the handling of the so-called slippage fee, which takes the `feePercent` and computes `feeBps` as `feePercent & 0x3FFF`. Since the variable `feePercent` has two versions, it is better to ensure it is not version 0. In other words, we can improve the current computation as follows: `feeBps = ( feePercent >> 248 == 0 )? feePercent : feePercent & 0x3FFF`.

```
    function _calcSlippageFees(uint256 slippage, uint256 feePercent)
    private
    view
    returns (uint256 partnerShare, uint256 paraswapShare)
    {
        uint256 feeBps = feePercent & 0x3FFF;
        require(feeBps + paraswapReferralShare <= 10000, "Invalid fee percent");
        paraswapShare = slippage.mul(paraswapReferralShare).div(10000);
        partnerShare = slippage.mul(feeBps).div(10000);
    }
```

Listing 3.2: `FeeModel::_calcSlippageFees()`

Recommendation Improve the above `_calcSlippageFees()` routine to ensure the fee version is not 0.

Status This issue has been confirmed. The team clarifies the check is intentionally removed, it is not needed anymore.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Staking`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `DistributorController` protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the protocol-wide operations (e.g., configure protocol parameters and withdraw funds in emergency). 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 `owner` account and its related privileged accesses in current contract.

To elaborate, we show below the related function. The `emergencyWithdraw()` routine allows the owner to take the rewards in emergency.

```
    function emergencyWithdraw() external onlyOwner {
        distributor.withdrawTokens(msg.sender, type(uint256).max);
        distributor.withdrawNative(msg.sender, type(uint256).max);

        emit EmergencyWithdrawal();
    }
```

Listing 3.3: `DistributorController::emergencyWithdraw()`

We understand the need of the privileged functions for contract maintenance, but it is worrisome if the privileged `owner` 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 The issue has been confirmed by the team. The team clarifies the admin will be a multi-sig controlled by a group of `DAO` members elected publicly.

3.4 Safe-Version Replacement With And safeTransferFrom()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: DistributorController
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transferFrom()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. Specifically, the `transferFrom()` routine does not have a return value defined and implemented. However, the `IERC20` interface has defined the `transferFrom()` interface with a `bool` return value. As a result, the call to `transferFrom()` may expect a return value. With the lack of return value of `USDT`'s `transferFrom()`, the call will be unfortunately reverted.

```
    function transferFrom(address _from, address _to, uint _value) public
        onlyPayloadSize(3 * 32) {
        var _allowance = allowed[_from][msg.sender];

        // Check is not needed because sub(_allowance, _value) will already throw if
            this condition is not met
        // if (_value > _allowance) throw;

        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        if (_allowance < MAX_UINT) {
            allowed[_from][msg.sender] = _allowance.sub(_value);
        }
        uint sendAmount = _value.sub(fee);
        balances[_from] = balances[_from].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(_from, owner, fee);
        }
        Transfer(_from, _to, sendAmount);
    }
```

Listing 3.4: USDT Token **Contract**

Because of that, a normal call to `transferFrom()` is suggested to use the safe version, i.e., `safeTransferFrom()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful.

In the following, we show the `transferRewardsOnly()` routine in the `DistributorController` contract. If `USDT` is given as `rewardToken`, the unsafe version of `rewardToken.transferFrom(rewardsSupplier, address(distributor), totalRewards)` (line 64) may revert as there is no return value in the `USDT` token contract's `transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
63      function transferRewardsOnly(uint256 totalRewards) external onlyDistributorAdmin {
64          rewardToken.transferFrom(rewardsSupplier, address(distributor), totalRewards);
65      }
```

Listing 3.5: `DistributorController::transferRewardsOnly()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `transferFrom()`.

Status This issue has been confirmed. The team clarifies the distributorController's rewards token are bound to underlying distributor's reward token which are standard erc20 tokens.

4 Conclusion

In this audit, we have analyzed the `ParaSwap DirectSwap` and `FeeModel` protocol design and implementation. These contracts are integral to the `ParaSwap` aggregation protocol. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.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: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.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.