SMART CONTRACT AUDIT REPORT

for

ParaSwap

Prepared By: Xiaomi Huang

PeckShield

Jun 10, 2022

Document Properties

Client	ParaSwap
Title	Smart Contract Audit Report
Target	ParaSwap
Version	1.0
Author	Shulin Bie
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	Jun 10, 2022	Shulin Bie	Final Release
1.0-rc	April 22, 2022	Shulin Bie	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 **4**

1.1 About ParaSwap 4

1.2 About PeckShield 5

1.3 Methodology 6

1.4 Disclaimer 9

2 Findings **10**

2.1 Summary 10

2.2 Key Findings 11

3 Detailed Results **12**

3.1 Improper Logic Of SimpleSwapNFT::performSimpleBuyNFT() 12

3.2 Incompatibility With Deflationary/Rebasing Tokens 14

3.3 Inconsistency Between Implementation And Document 16

4 Conclusion **17**

References **18**

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the ParaSwap 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 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 aggregates decentralized exchanges and other DeFi services in one comprehensive interface to streamline and facilitate users' interactions with decentralized finance (DeFi). In other words, it is a middleware streamlining user's interactions with various DeFi services. Specifically, it gathers liquidity from the main decentralized exchanges together in a convenient interface abstracting most of the swaps' complexity to make it convenient and accessible for end-users. Additionally, ParaSwap introduces a limit order mechanism, which supports the swap between any kind of tokens (e.g., ERC20 <-> ERC20, ERC20 <-> ERC721, ERC20 <-> ERC1155, etc.).

Table 1.1: Basic Information of ParaSwap

Item	Description
Target	ParaSwap
Website	https://www.paraswap.io/
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	Jun 10, 2022

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit. In the first repository, our audit only covers the following contracts: `fee/FeeClaimer.sol`, `fee/FeeModel.sol`, `routers/SimpleSwap.sol`, `routers/ProtectedSimpleSwap.sol`, `routers/MultiPath.sol`, `routers/ProtectedMultiPath.sol`, `routers/SimpleSwapNFT.sol`, `routers/OnERC721Received.sol`, `routers/OnERC1155Received.sol`, `routers/ERC165.sol`, `routers/AugustusRFQRouter.sol`, and `lib/UtilsNFT.sol`.

- https://github.com/paraswap/paraswap-contracts/tree/audit/fee-rfq-nft (8d461db)
- https://github.com/paraswap/paraswap-limit-orders.git (82683d3)

And these are the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/paraswap/paraswap-contracts/tree/audit/fee-rfq-nft (fe58b02)
- https://github.com/paraswap/paraswap-limit-orders.git (82683d3)

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

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 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) [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.

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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `ParaSwap` 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	1	🟦
Medium	0	
Low	1	🟦
Informational	1	🟦
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 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 high-severity vulnerability, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key ParaSwap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Improper Logic Of SimpleSwapNFT::performSimpleBuyNFT()	Business Logic	Fixed
PVE-002	Low	Incompatibility With Deflationary/Rebasing Tokens	Business Logic	Confirmed
PVE-003	Informational	Inconsistency Between Implementation And Document	Coding Practices	Fixed

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 Improper Logic Of SimpleSwapNFT::performSimpleBuyNFT()

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

Description

By gathering liquidity from the main decentralized exchanges together, `ParaSwap` is a middleware streamlining user's interactions with various DeFi services. Within the protocol, there is a constant need of swapping from one token to another. The `SimpleSwapNFT` contract is exactly designed to swap `ERC20` token to `ERC721` or `ERC1155` token. In particular, the `performSimpleBuyNFT()` routine is designed to buy `ERC721` or `ERC1155` token with the incoming `ERC20` token. While examining its logic, we observe there is an improper implementation that needs to be improved.

To elaborate, we show below the related code snippet of the `SimpleSwapNFT` contract. By design, in the `performSimpleBuyNFT()` routine, in order to distinguish the bought token as `ERC721` or `ERC1155`, the low 160 bits (i.e., 0 - 159 bits) of the `details.toToken` are used to represent the token address and the `161th` bit (i.e., 160 bit) is used to represent the token type. However, it comes to our attention that the `21th` bit (i.e., 20 bit) of the `details.toToken` is incorrectly used (line 101) to distinguish the bought token, which directly undermines the assumption of the `ParaSwap` design. Given this, we suggest to correct the implementation as below: `if ((details.toToken & (1 << 160))== 0)` (line 101).

```
68      function performSimpleBuyNFT(
69          address[] memory callees,
70          bytes memory exchangeData,
71          uint256[] memory startIndexes,
72          uint256[] memory values,
73          address fromToken,
        UtilsNFT.ToTokenNFTDetails[] memory toTokenDetails,
        uint256 fromAmount,
        uint256 expectedAmount,
        address payable partner,
        uint256 feePercent,
        bytes memory permit,
        address payable beneficiary
    ) private returns (uint256 remainingAmount) {
        require(msg.value == (fromToken == Utils.ethAddress() ? fromAmount : 0), "Incorrect msg.value");
        require(toTokenDetails.length > 0, "toTokenDetails can't be empty");
        require(callees.length + 1 == startIndexes.length, "Start indexes must be 1 greater then number of callees");
        require(callees.length == values.length, "callees and values must have same length");
        require(_isTakeFeeFromSrcToken(feePercent), "fee on dest token not supported");

        //If source token is not ETH than transfer required amount of tokens
        //from sender to this contract
        transferTokensFromProxy(fromToken, fromAmount, permit);

        performCalls(callees, exchangeData, startIndexes, values);

        // Slippage check is not require. If all the requested ERC721 and ERC1155
        // are transferred correctly the swap should succeed.
        for (uint256 i = 0; i < toTokenDetails.length; i++) {
            UtilsNFT.ToTokenNFTDetails memory details = toTokenDetails[i];
            // toToken is packed
            // 0 - 19 bits: token address
            // 20 bit: tokenType 0 -> ERC721, 1 -> ERC1155
            if ((details.toToken & (1 << 20)) == 0) {
                UtilsNFT.transferTokens721(address(details.toToken), beneficiary, details.toTokenID);
            } else {
                UtilsNFT.transferTokens1155(address(details.toToken), beneficiary, details.toTokenID, details.toAmount);
            }
        }

        // take slippage from src token
        remainingAmount = Utils.tokenBalance(fromToken, address(this));
        takeFromTokenFeeSlippageAndTransfer(
            fromToken,
            fromAmount,
            expectedAmount,
            remainingAmount,
            partner,
            feePercent
        );

        return remainingAmount;
    }
```

Listing 3.1: `SimpleSwapNFT::performSimpleBuyNFT()`

Recommendation Correct the implementation of the `performSimpleBuyNFT()` routine as above-mentioned.

Status The issue has been addressed by the following commit: `592a96c`.

3.2 Incompatibility With Deflationary/Rebasing Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [2]
- CWE subcategory: CWE-841 [1]

Description

By design, the `AugustusRFQRouter` contract is one of the main entries for interaction with users, which allows the user to swap `ERC20` token to any kind of tokens (`ERC20`, `ERC721`, or `ERC1155`) via choosing a limit order. The `swapOnAugustusRFQ()` routine is one of the representative routines. While examining its logic, we observe the incoming token (i.e., `order.takerAsset`) is transferred to the `AugustusRFQRouter` contract and then transferred to the maker of the limit order. This is reasonable under the assumption that these transfers will always result in full transfer. Otherwise, the transaction will be reverted.

```
    function swapOnAugustusRFQ(
        IAugustusRFQ.Order calldata order,
        bytes calldata signature,
        uint8 wrapETH // set 0 bit to wrap src, and 1 bit to wrap dst
    ) external payable {
        address userAddress = address(uint160(order.nonceAndMeta));
        require(userAddress == address(0)  userAddress == msg.sender, "unauthorized user
            ");

        uint256 fromAmount = order.takerAmount;
        if (wrapETH & 1 != 0) {
            require(msg.value == fromAmount, "Incorrect msg.value");
            IWETH(weth).deposit{ value: fromAmount }();
        } else {
            require(msg.value == 0, "Incorrect msg.value");
            tokenTransferProxy.transferFrom(order.takerAsset, msg.sender, address(this),
                 fromAmount);
        }
        Utils.approve(exchange, order.takerAsset, fromAmount);

        if (wrapETH & 2 != 0) {
            IAugustusRFQ(exchange).fillOrder(order, signature);
            uint256 receivedAmount = Utils.tokenBalance(order.makerAsset, address(this))
                ;
            IWETH(weth).withdraw(receivedAmount);
            Utils.transferETH(msg.sender, receivedAmount);
        } else {
            IAugustusRFQ(exchange).fillOrderWithTarget(order, signature, msg.sender);
        }
    }
```

Listing 3.2: `AugustusRFQRouter::swapOnAugustusRFQ()`

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 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 routines related to token transfer.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming 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. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into `ParaSwap`. In `ParaSwap`, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., `USDT`) that may have control switches that can be dynamically exercised to suddenly become one.

Recommendation If current codebase needs to support possible deflationary tokens, it is better 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 The issue has been confirmed by the team.

3.3 Inconsistency Between Implementation And Document

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

Description

In the ParaSwap protocol, the AugustusRFQ contract is one of the main entries for interaction with users, which implements a limit order mechanism to support the swap between any kind of tokens. In particular, the struct OrderNFT is used to record one limit order information. While examining its logic, we notice there is a misleading comment embedded above its definition, which brings unnecessary hurdles to understand and/or maintain the software.

To elaborate, we show below the related code snippet of the AugustusRFQ contract. By design, in the struct OrderNFT, the low 160 bits (i.e., 0 - 159 bits) of the makerAsset and takerAsset represent the asset address and the rest represent the token type. However, we notice the comments (lines 28 - 29) are "0 - 19 bits are address; 20 - 21 bits are tokenType (0 ERC20, 1 ERC1155, 2 ERC721)". It will bring unnecessary hurdles to understand the design.

```
27      // makerAsset and takerAsset are Packed structures
28      // 0 - 19 bits are address
29      // 20 - 21 bits are tokenType (0 ERC20, 1 ERC1155, 2 ERC721)
30      struct OrderNFT {
31          uint256 nonceAndMeta; // Nonce and taker specific metadata
32          uint128 expiry;
33          uint256 makerAsset;
34          uint256 makerAssetId; // simply ignored in case of ERC20s
35          uint256 takerAsset;
36          uint256 takerAssetId; // simply ignored in case of ERC20s
37          address maker;
38          address taker;  // zero address on orders executable by anyone
39          uint256 makerAmount;
40          uint256 takerAmount;
41      }
```

Listing 3.3: AugustusRFQ

Recommendation Ensure the consistency between documents (including embedded comments) and implementation.

Status The issue has been addressed by the following commit: fe58b02.

4 Conclusion

In this audit, we have analyzed the `ParaSwap` design and implementation. `ParaSwap` aggregates decentralized exchanges and other DeFi services in one comprehensive interface to streamline and facilitate users' interactions with decentralized finance (`DeFi`). In other words, it is a middleware streamlining user's interactions with various DeFi services. Specifically, it gathers liquidity from the main decentralized exchanges together in a convenient interface abstracting most of the swaps' complexity to make it convenient and accessible for end-users. Additionally, `ParaSwap` introduces a limit order mechanism, which supports the swap between any kind of tokens. 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-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.