SMART CONTRACT AUDIT REPORT

for

Orbs TWAP

Prepared By: Xiaomi Huang

PeckShield

September 8, 2022

Document Properties

Client	Orbs LTD.
Title	Smart Contract Audit Report
Target	Orbs TWAP
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	September 8, 2022	Shulin Bie	Final Release
1.0-rc	September 2, 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

Given the opportunity to review the design document and related smart contract source code of the `Orbs TWAP` 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 Orbs TWAP

The `Orbs TWAP` protocol implements a `TWAP Order` mechanism (either `Limit Order` or `Market Order`) with the possibility of partial fills. It aims to minimize a large order's impact on the market or neutralize market volatility over time by dividing trades into smaller quantities and executing them at regular intervals over time. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of Orbs TWAP

Item	Description
Target	Orbs TWAP
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	September 8, 2022

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

- https://github.com/orbs-network/twap.git (f06f4e1)

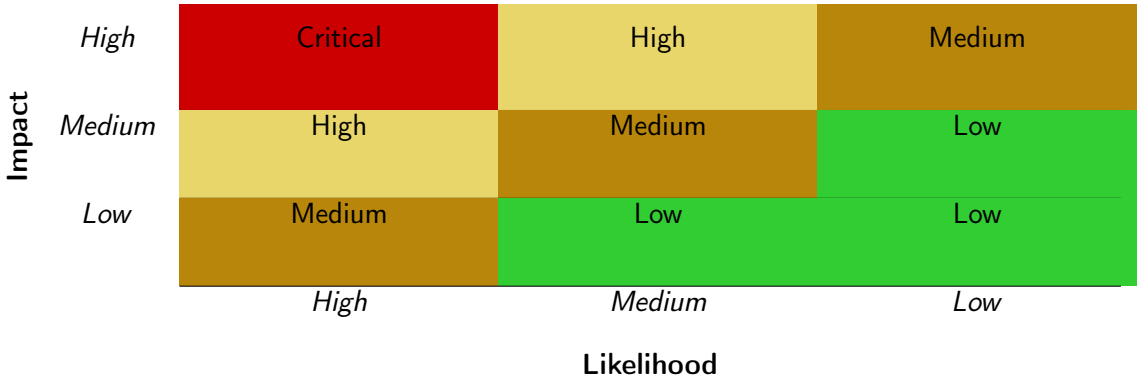
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/orbs-network/twap.git (c129880)

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



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

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

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.

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 Orbs TWAP 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	1	
Informational	0	
Total	2	

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 1 low-severity vulnerability.

Table 2.1: Key Orbs TWAP Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Revisited Logic of TWAP::performFill()	Business Logic	Fixed
PVE-002	Low	Incompatibility with Deflationary/Rebasing Tokens	Business Logic	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 Revisited Logic of TWAP::performFill()

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

Description

The TWAP contract is the main entry for interaction with users. It allows the user (i.e., maker) to create a TWAP order, which breaks a larger order down into smaller trades (or chunks) that will be executed at regular intervals over a period of time. Once created, an English Auction bidding war on each chunk will be enabled. Anyone can serve as a taker by finding the best swap path on any DEX to fill the order for the chunk within the parameters set by the maker. In particular, the fill() routine is called by the winning bidder to fill the order for the chunk. After further analysis, we observe the internal performFill() routine called inside it needs to be improved.

To elaborate, we show below the related code snippet of the TWAP contract. Inside the performFill() routine, the local expectedOut variable indicates the minimum amount of dstToken that the maker of the order should receive. By design, it should be the bid amount of the winning bidder. However, we notice the minimum amount specified by the maker is incorrectly used as expectedOut (line 327). Apparently, it does not meet the design of the English Auction. Given this, we suggest to improve the implementation as below: uint256 expectedOut = o.bid.dstAmount (line 327).

```
306     function fill(uint64 id) external nonReentrant {
307         OrderLib.Order memory o = order(id);
308
309         (address exchange, uint256 srcAmountIn, uint256 dstAmountOut, uint256 dstFee) =
                performFill(o);
310         o.filled(srcAmountIn);
311
312         ...
```

```
313     }
314
315     function performFill(OrderLib.Order memory o) private returns (address exchange,
            uint256 srcAmountIn, uint256 dstAmountOut, uint256 dstFee)
316     {
317         require(msg.sender == o.bid.taker, "taker");
318         require(block.timestamp < o.status, "status"); // deadline, canceled or
                completed
319         require(block.timestamp > o.bid.time + MIN_BID_WINDOW_SECONDS, "pending bid");
320
321         exchange = o.bid.exchange;
322         dstFee = o.bid.dstFee;
323         srcAmountIn = o.srcBidAmountNext();
324         ERC20(o.ask.srcToken).safeTransferFrom(o.ask.maker, address(this), srcAmountIn);
325         ERC20(o.ask.srcToken).safeIncreaseAllowance(exchange, srcAmountIn);
326
327         uint256 expectedOut = o.dstMinAmountNext();
328         dstAmountOut = IExchange(exchange).swap(srcAmountIn, expectedOut + dstFee, o.bid
                .data);
329         dstAmountOut -= dstFee;
330         require(dstAmountOut >= expectedOut, "min out");
331
332         ERC20(o.ask.dstToken).safeTransfer(o.bid.taker, dstFee);
333         ERC20(o.ask.dstToken).safeTransfer(o.ask.maker, dstAmountOut);
334     }
```

Listing 3.1: `TWAP::fill()&&performFill()`

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

Status The issue has been addressed in this commit: `762badc`.

3.2 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `UniswapV2Exchange/TWAP`
- Category: Business Logic [2]
- CWE subcategory: CWE-841 [1]

Description

By design, the `UniswapV2Exchange` contract is an adapter of `DEX`. In particular, one entry routine, i.e., `swap()`, allows the user to swap a kind of `ERC20` token to another. While examining its logic, we observe the current implementation is reasonable under the assumption that the token transfer inside the routine will always result in full transfer. Otherwise, the transaction will be reverted.

```
31      function swap(
32          uint256 amountIn,
33          uint256 amountOutMin,
34          bytes calldata data
35      ) public returns (uint256 amountOut) {
36          address[] memory path = abi.decode(data, (address[]));
37          ERC20 srcToken = ERC20(path[0]);
38          srcToken.safeTransferFrom(msg.sender, address(this), amountIn);
39          srcToken.safeIncreaseAllowance(address(uniswap), amountIn);
40          return
41              uniswap.swapExactTokensForTokens(amountIn, amountOutMin, path, msg.sender,
                    block.timestamp)[
42                  path.length - 1
43              ];
44      }
```

Listing 3.2: `UniswapV2Exchange::swap()`

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.

Note another routine, i.e., `TWAP::performFill()`, shares the same issue.

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 addressed by the following commits: `bac1255` and `762badc`.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Orbs TWAP`, which implements a `TWAP Order` mechanism with the possibility of partial fills. It aims to minimize a large order's impact on the market or neutralize market volatility over time by dividing trades into smaller quantities and executing them at regular intervals over time. 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.