SMART CONTRACT AUDIT REPORT

for

Tokenlon (Limit Order)

Prepared By: Patrick Lou

PeckShield

May 3, 2022

Document Properties

Client	Tokenlon
Title	Smart Contract Audit Report
Target	Tokenlon (Limit Order)
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Patrick Lou, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 3, 2022	Xuxian Jiang	Final Release
1.0-rc	April 22, 2022	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Patrick Lou
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 Limit Order support in Tokenlon, 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 Tokenlon

Tokenlon is originally based on the 0x protocol for decentralized atomic currency exchange, which provides users with faster speed, better price decentralized currency exchange services. It is different from other decentralized exchanges in being neither an Automated Market Maker (AMM) nor an order book exchange. Instead, It adopts an exchange methodology called Request For Quotation (RFQ) so that trading on Tokenlon looks like trading with an automated Over-The-Counter (OTC) desk. As a result, Tokenlon achieves extremely low failure of trading transaction execution with competitive, zero-slippage prices. The Limit Order feature supports trading orders that allow to buy or sell at a specified price or better. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Tokenlon (Limit Order)

Item	Description
Name	Tokenlon
Website	https://tokenlon.im/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 3, 2022

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

- https://github.com/consenlabs/tokenlon-contracts.git (63e15ff)

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

- https://github.com/consenlabs/tokenlon-contracts.git (fe62831)

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

Likelihood

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), 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 design and implementation of the `Limit Order` support in `Tokenlon`. 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	1	■
Informational	2	■ ■
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 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, 1 low-severity vulnerability, and 2 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Lack Of Ether Support in Limit Orders	Business Logic	Resolved
PVE-002	Informational	Layout Inconsistency in Order Field Members	Coding Practices	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-004	Informational	Suggested Flexibility in TraderParams And Limit Order Events	Coding Practices	Resolved

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 Lack Of Ether Support in Limit Orders

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

Description

The TokenLon protocol has the UserProxy contract as the main entrance for interaction with various features in supporting low-cost, reliable market making to fulfill user trading requests. The new feature of Limit Order supports to buy or sell at a specified price or better. While examining the current implementation, we notice the dispatcher routine from the UserProxy contract supports the use of Ether, which is currently not supported yet in the LimitOrder feature.

To elaborate, we show below its toLimitOrder() routine, which is invoked to call the actual Limit Order functionality. Note the Limit Order functionality is supported by three public functions: fillLimitOrderByTrader(), fillLimitOrderByProtocol(), and cancelLimitOrder(). None of these three functions has the payable support, which is inconsistent with the following toLimitOrder() function.

```
    function toLimitOrder(bytes calldata _payload) external payable {
        require(isLimitOrderEnabled(), "UserProxy: Limit Order is disabled");
        require(msg.sender == tx.origin, "UserProxy: only EOA");

        (bool callSucceed, ) = limitOrderAddr().call{ value: msg.value }(_payload);
        if (callSucceed == false) {
            // Get the error message returned
            assembly {
                let ptr := mload(0x40)
                let size := returndatasize()
                returndatacopy(ptr, 0, size)
                revert(ptr, size)
            }
```

```
        }
    }
```

Listing 3.1: `UserProxy::toLimitOrder()`

Recommendation If the `ether` is not going to be supported in `Limit Order`, we suggest to remove the `payable` and `msg.value` in the above `toLimitOrder()` function.

Status The issue has been fixed by this PR: `323`.

3.2 Layout Inconsistency in Order Field Members

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `LimitOrderLibEIP712`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

The support of `Limit Order` is facilitates by a number of closely-related data structures, e.g., `Order`, `AllowFill`, `CoordinatorParams`, and `TraderParams`. We notice these data structures have the common fields of `salt` and `expiry` for `EIP712`-based signature validation.

In the following, we show below the definitions of these data structures. For the convenience of order organization and maintenance, we suggest to have the consistent ordering in the layout of common fields. In the current implementation, we notice the `Order` data structure has the `salt` field after `expiry` while the `AllowFill` data structure has the `salt` field before `expiry`!

```
    struct Order {
        IERC20 makerToken;
        IERC20 takerToken;
        uint256 makerTokenAmount;
        uint256 takerTokenAmount;
        address maker;
        address taker;
        uint64 expiry;
        uint256 salt;
    }

    struct AllowFill {
        bytes32 orderHash;
        address executor;
        uint256 fillAmount;
        uint256 salt;
        uint64 expiry;
```

```
    }
```

Listing 3.2: LimitOrderLibEIP712::Order/ AllowFill

```
    struct CoordinatorParams {
        bytes sig;
        uint64 expiry;
        uint256 salt;
    }

    struct TraderParams {
        address taker;
        address recipient;
        uint256 takerTokenAmount;
        uint64 expiry;
        uint256 salt;
        bytes takerSig;
    }
```

Listing 3.3: ILimitOrder :: CoordinatorParams/TraderParams

Recommendation Be consistent in the layout of common field members in the above data structures.

The issue has been fixed by this PR: 324.

3.3 Trust Issue of Admin Keys

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

Description

In Tokenlon,there are privileged accounts (admin and operator) that play a critical role in governing and regulating the system-wide operations (e.g., parameter setting and proxy upgrades). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged accounts need to be scrutinized. In the following, we examine the privileged accounts and their related privileged accesses in current contracts.

```
    function upgradeSpender(address _newSpender) external onlyOperator {
        require(_newSpender != address(0), "LimitOrder: spender can not be zero address"
            );
        spender = ISpender(_newSpender);

```

```
        emit UpgradeSpender(_newSpender);
    }

    function upgradeCoordinator(address _newCoordinator) external onlyOperator {
        require(_newCoordinator != address(0), "LimitOrder: coordinator can not be zero address");
        coordinator = _newCoordinator;

        emit UpgradeCoordinator(_newCoordinator);
    }

    /**
     * @dev approve spender to transfer tokens from this contract. This is used to collect fee.
     */
    function setAllowance(address[] calldata _tokenList, address _spender) external onlyOperator {
        for (uint256 i = 0; i < _tokenList.length; i++) {
            IERC20(_tokenList[i]).safeApprove(_spender, LibConstant.MAX_UINT);

            emit AllowTransfer(_spender);
        }
    }
```

Listing 3.4: Example `Setters` in the `LimitOrder` Contract

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

```
contract TransparentUpgradeableProxyImpl is TransparentUpgradeableProxy {
  constructor(
    address _logic,
    address _admin,
    bytes memory _data
  ) public payable TransparentUpgradeableProxy(_logic, _admin, _data) {}
}
```

Listing 3.5: `TransparentUpgradeableProxyImpl::constructor()`

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 the built-in timelock-based scheme and the multisig-based deployment to regulate the privileges of concern.

3.4 Suggested Flexibility in TraderParams And Limit Order Events

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `LimitOrder`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `LimitOrder` contract as an example. While examining the events that reflect the order dynamics, we notice the current events may be improved for offline indexing and management. To elaborate, we show below the related two events: `_emitLimitOrderFilledByTrader()` and `_emitLimitOrderFilledByProtocol`. When the order is being filled either by a designated trader or by the protocol relayer, it is helpful to have the direct information of the remaining amount that remains to be filled. Otherwise, the current implementation still requires the offchain computation to derive the remaining amount for fulfillment.

```
    function _emitLimitOrderFilledByTrader(LimitOrderFilledByTraderParams memory _params
        ) internal {
        emit LimitOrderFilledByTrader(
            _params.orderHash,
            _params.maker,
            _params.taker,
            _params.allowFillHash,
            _params.recipient,
            _params.makerToken,
            _params.takerToken,
            _params.makerTokenFilledAmount,
            _params.takerTokenFilledAmount,
            _params.makerTokenFee,
            _params.takerTokenFee
        );
```

```
    }
```

Listing 3.6: `LimitOrder::_emitLimitOrderFilledByTrader()`

```
    function _emitLimitOrderFilledByProtocol(LimitOrderFilledByProtocolParams memory
        _params) internal {
        emit LimitOrderFilledByProtocol(
            _params.orderHash,
            _params.maker,
            _params.taker,
            _params.allowFillHash,
            _params.relayer,
            _params.profitRecipient,
            _params.makerToken,
            _params.takerToken,
            _params.makerTokenFilledAmount,
            _params.takerTokenFilledAmount,
            _params.makerTokenFee,
            _params.takerTokenFee,
            _params.takerTokenProfit,
            _params.takerTokenProfitFee,
            _params.takerTokenProfitBackToMaker
        );
    }
```

Listing 3.7: `LimitOrder::_emitLimitOrderFilledByProtocol()`

Moreover, the `TraderParams` data structure supports to specify the recipient after the trade. It is helpful to use the `address(0)` to represent the current taker.

Recommendation Properly extend the above limit order events to include the remaining token amount that remains to be fulfilled. This is very helpful for external analytics and reporting tools.

The issue has been fixed by this PR: `325`.

4 | Conclusion

In this audit, we have analyzed the documentation and implementation of the `Limit Order` support in `Tokenlon`. The audited `Limit Order` feature supports trading orders that allow to buy or sell at a specified price or better. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.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.