SMART CONTRACT AUDIT REPORT

for

TOKENLON V5.2

Prepared By: Shuxiao Wang

PeckShield

May 23, 2021

Document Properties

Client	Tokenlon
Title	Smart Contract Audit Report
Target	Tokenlon V5.2
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Yiqun Chen, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	May 23, 2021	Xuxian Jiang	Final Release
1.0-rc	May 22, 2021	Xuxian Jiang	Release Candidate #1
0.2	May 20, 2020	Xuxian Jiang	Additional Findings #1
0.1	May 15, 2020	Xuxian Jiang	First Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **Tokenlon V5.2** 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 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 V5.2

`Tokenlon` is originally based on `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. `Tokenlon V5.2` is an upgrade over earlier versions by better supporting external AMM/PMM offerings, such as `UniswapV2` and `Curve`.

The basic information of Tokenlon V5.2 is as follows:

Table 1.1: Basic Information of Tokenlon V5.2

Item	Description
Issuer	Tokenlon
Website	https://tokenlon.im/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 23, 2021

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 (fb58afb)

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 (7fc32f)

1.2 About PeckShield

PeckShield Inc. [10] 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 [9]:

- 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) [8], 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 Tokenlon V5.2 protocol design and 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 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	2	■ ■
Informational	1	■
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, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Adjustment Of AuthorizeSpender Event in authorize()	Coding Practices	Fixed
PVE-002	Low	Better Handling of Privilege Transfers	Security Features	Fixed
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-004	Informational	Incompatibility with Deflationary/Re-basing Tokens	Business Logic	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 Adjustment Of AuthorizeSpender Event in authorize()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Spender
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

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 Spender contract as an example. This contract is designed to authorize the user asset transfers. While examining the events that reflect the token dynamics, we notice the AuthorizeSpender event is not properly emitted when the new spender becomes effective.

To elaborate, we show below its authorize() routine, which is invoked to authorize a new spender. Note that it indeed emits an AuthorizeSpender event. However, the new spender does not become effective until the timelockExpirationTime timer expires. With that, it is suggested to emit the event when the new spender becomes effective.

```
    function authorize(address[] calldata _pendingAuthorized) external onlyOperator {
        require(_pendingAuthorized.length > 0, "Spender: authorize list is empty");
        require(numPendingAuthorized == 0 && timelockExpirationTime == 0, "Spender: an authorize current in progress");

        if (timelockActivated) {
            numPendingAuthorized = _pendingAuthorized.length;
            for (uint256 i = 0; i < _pendingAuthorized.length; i++) {
```

```
                require(_pendingAuthorized[i] != address(0), "Spender: can not authorize zero address");
                pendingAuthorized[i] = _pendingAuthorized[i];

                emit AuthorizeSpender(_pendingAuthorized[i], true);
            }
            timelockExpirationTime = now + TIME_LOCK_DURATION;
        } else {
            for (uint256 i = 0; i < _pendingAuthorized.length; i++) {
                require(_pendingAuthorized[i] != address(0), "Spender: can not authorize zero address");
                authorized[_pendingAuthorized[i]] = true;

                emit AuthorizeSpender(_pendingAuthorized[i], true);
            }
        }
    }
```

Listing 3.1: Spender::authorize()

Recommendation Properly emit the `AuthorizeSpender` event at the very moment when the new spender becomes effective. This is very helpful for external analytics and reporting tools.

Status The issue has been fixed by this commit: `24b5eb9`.

3.2 Better Handling of Privilege Transfers

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-282 [1]

Description

Tokenlon V5.2 implements a rather basic access control mechanism that allows a privileged account, i.e., `operator`, to be granted exclusive access to typically sensitive functions (e.g., `setAllowanceTarget()`). Because of the privileged access and the implications of these sensitive functions, the `operator` account is essential for the protocol-level safety and operation. In the following, we elaborate with the `operator` account.

With the `Spender` contract as an example, a specific function, i.e., `transferOwnership()`, is provided to allow for possible `operator` updates. However, current implementation achieves its goal within a single transaction. This is reasonable under the assumption that the `_newOperator` parameter is always correctly provided. However, in the unlikely situation, when an incorrect `_newOperator` is provided, the contract owner may be forever lost, which might be devastating for Tokenlon V5.2 operation and maintenance.

As a common best practice, instead of achieving the `operator` update within a single transaction, it is suggested to split the operation into two steps. The first step initiates the `operator` update intent and the second step accepts and materializes the update. Both steps should be executed in two separate transactions. By doing so, it can greatly alleviate the concern of accidentally transferring the contract `operator` to an uncontrolled address. In other words, this two-step procedure ensures that an `operator` public key cannot be nominated unless there is an entity that has the corresponding private key. This is explicitly designed to prevent unintentional errors in the `operator` transfer process.

```
    function transferOwnership(address _newOperator) external onlyOperator {
        require(_newOperator != address(0), "Spender: operator can not be zero address")
            ;
        operator = _newOperator;

        emit TransferOwnership(_newOperator);
    }
```

Listing 3.2: Spender::transferOwnership()

Recommendation Implement a two-step approach for the `operator` update (or transfer): `setOperator()` and `acceptOperator()`.

Status The issue has been fixed by this commit: `54b0ca9`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In Tokenlon V5.2, there is a privileged contract, i.e., `AllowanceTarget`, that plays a critical role in receiving allowances from trading users. This contract is designed to greatly facilitate the asset transfers for various trade operations.

In the following, we show the code snippet from the `AllowanceTarget` contract. This contract has a key function, i.e., `executeCall()`, which can only be invoked by the `Spender` contract. Within the `Spender` contract, there is a routine named `spendFromUser()` that is designed to spend tokens on user's behalf. By design, only an authorized entity can successfully invoke it.

```
    /// @dev Execute an arbitrary call. Only an authority can call this.
    /// @param target The call target.
    /// @param callData The call data.
    /// @return resultData The data returned by the call.
    function executeCall(
        address payable target,
        bytes calldata callData
    )
        override
        external
        onlySpender
        returns (bytes memory resultData)
    {
        bool success;
        (success, resultData) = target.call(callData);
        if (!success) {
            // Get the error message returned
            assembly {
                let ptr := mload(0x40)
                let size := returndatasize()
                returndatacopy(ptr, 0, size)
                revert(ptr, size)
            }
        }
    }
```

Listing 3.3: AllowanceTarget::executeCall()

```
    /// @dev Spend tokens on user's behalf. Only an authority can call this.
    /// @param _user The user to spend token from.
    /// @param _tokenAddr The address of the token.
    /// @param _amount Amount to spend.
    function spendFromUser(address _user, address _tokenAddr, uint256 _amount) external
        onlyAuthorized {
        require(! tokenBlacklist[_tokenAddr], "Spender: token is blacklisted");

        if (_tokenAddr != ETH_ADDRESS && _tokenAddr != ZERO_ADDRESS) {

            uint256 balanceBefore = IERC20(_tokenAddr).balanceOf(msg.sender);
            (bool callSucceed, ) = address(allowanceTarget).call(
                abi.encodeWithSelector(
                    IAllowanceTarget.executeCall.selector,
                    _tokenAddr,
                    abi.encodeWithSelector(
                        IERC20.transferFrom.selector,
                        _user,
                        msg.sender,
                        _amount
                    )
                )
            );
            require(callSucceed, "Spender: ERC20 transferFrom failed");
            // Check balance
```

```
            uint256 balanceAfter = IERC20(_tokenAddr).balanceOf(msg.sender);
            require(balanceAfter.sub(balanceBefore) == _amount, "Spender: ERC20
                transferFrom result mismatch");

        }
    }
```

Listing 3.4: Spender::spendFromUser()

As this `spendFromUser()` routine is capable of taking assets from current trading users up to permitted allowances, it is properly guarded with the `onlyAuthorized` modifier, which naturally introduces the trust issue on the authorized accounts.

As a mitigation, instead of having a single EOA as the authorized account, an alternative is to make use of a multi-sig wallet. To further eliminate the administration key concern, it may be required to transfer the role to a community-governed DAO. In the meantime, a timelock-based mechanism might also be applicable for mitigation.

Recommendation Promptly transfer the privilege of authorized accounts to the intended `DAO`-like governance contract. And 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 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `PMM`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In Tokenlon V5.2, there is a core `PMM` contract that supports low-cost, reliable market making to fulfill user trading requests. In particular, one key routine, i.e., `fill()`, accepts asset transfer-in and trades back the requested asset. Naturally, the contract implements a number of low-level helper routines to transfer assets into or out of Tokenlon V5.2. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    function fill(
        uint256 userSalt,
```

```
        bytes memory data,
        bytes memory userSignature
    )
        override
        public
        payable
        onlyUserProxy
        nonReentrant
        returns (uint256)
    {
        // decode & assert
        (LibOrder.Order memory order,
        TradeInfo memory tradeInfo) = _assertTransaction(userSalt, data, userSignature);

        // Deposit to WETH if taker asset is ETH, else transfer from user
        IWETH weth = IWETH(permStorage.wethAddr());
        if (address(weth) == tradeInfo.takerAssetAddr) {
            require(
                msg.value == order.takerAssetAmount,
                "PMM: insufficient ETH"
            );
            weth.deposit{value: msg.value}();
        } else {
            spender.spendFromUser(tradeInfo.user, tradeInfo.takerAssetAddr, order.
                takerAssetAmount);
        }

        IERC20(tradeInfo.takerAssetAddr).safeIncreaseAllowance(zxERC20Proxy, order.
            takerAssetAmount);

        // send tx to 0x
        zeroExchange.executeTransaction(
            userSalt,
            address(this),
            data,
            ""
        );

        // settle token/ETH to user
        uint256 settleAmount = _settle(weth, tradeInfo.receiver, tradeInfo.
            makerAssetAddr, order.makerAssetAmount, tradeInfo.feeFactor);
        IERC20(tradeInfo.takerAssetAddr).safeApprove(zxERC20Proxy, 0);

        emit FillOrder(
            SOURCE,
            tradeInfo.transactionHash,
            tradeInfo.orderHash,
            tradeInfo.user,
            tradeInfo.takerAssetAddr,
            order.takerAssetAmount,
            order.makerAddress,
            tradeInfo.makerAssetAddr,
```

```
            order.makerAssetAmount,
            tradeInfo.receiver,
            settleAmount,
            tradeInfo.feeFactor
        );
        return settleAmount;
    }
```

Listing 3.5: PMM::fill()

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 a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `OHM/YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `deposit()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into Tokenlon V5.2 for borrowing/lending. In fact, Tokenlon V5.2 is indeed in the position to effectively regulate the set of assets that can be listed. Meanwhile, there exist certain assets that may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation If current codebase needs to support deflationary tokens, it is necessary 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 This issue has been acknowledged by the team. And the team has a proper vetting process in place to prevent deflationary/rebasing tokens from being listed in the protocol.

4 Conclusion

In this audit, we have analyzed the Tokenlon V5.2 documentation and implementation. The system presents a unique, robust offering as a decentralized non-custodial atomic currency exchange where end-users can participate as traders. Tokenlon V5.2 improves early versions by providing additional innovative features in seamlessly supporting external AMM integrations. 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-282: Improper Ownership Management. https://cwe.mitre.org/data/definitions/282.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[9] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[10] PeckShield. PeckShield Inc. https://www.peckshield.com.