



# SMART CONTRACT AUDIT REPORT

for

## TOKENLON



Prepared By: Shuxiao Wang

Hangzhou, China  
December 20, 2020

## Document Properties

Client	Tokenlon
Title	Smart Contract Audit Report
Target	TokenlonV5
Version	1.0
Author	Xuxian Jiang
Auditors	Xudong Shao, Huaguo Shi, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

## Version Info

Version	Date	Author(s)	Description
1.0	December 20, 2020	Xuxian Jiang	Final Release
1.0-rc	December 18, 2020	Xuxian Jiang	Release Candidate #1
0.3	December 12, 2020	Xuxian Jiang	Additional Findings #2
0.2	December 11, 2020	Xuxian Jiang	Additional Findings #1
0.1	December 9, 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

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	About TokenlonV5 . . . . .	4
1.2	About PeckShield . . . . .	5
1.3	Methodology . . . . .	5
1.4	Disclaimer . . . . .	6
<b>2</b>	<b>Findings</b>	<b>9</b>
2.1	Summary . . . . .	9
2.2	Key Findings . . . . .	10
<b>3</b>	<b>Detailed Results</b>	<b>11</b>
3.1	Inappropriate Subsidy Collection . . . . .	11
3.2	No ETH Support in emergencyWithdraw() . . . . .	14
3.3	Removal of Unused Code . . . . .	15
3.4	Suggested Improvement on stakeWithPermit() . . . . .	18
3.5	Trader-Controllable feeFactor, Not Protocol . . . . .	19
3.6	Potential DoS in AMMWrapper::transactionHash Generation . . . . .	20
3.7	Inconsistency Between Document and Implementation . . . . .	22
3.8	Suggested Reservation of 0 Index in curveTokenIndexes . . . . .	23
3.9	Trust Issue of Admin Keys Behind AllowanceTarget And Spender . . . . .	25
3.10	Other Suggestions . . . . .	27
<b>4</b>	<b>Conclusion</b>	<b>28</b>
	<b>References</b>	<b>29</b>

# 1 | Introduction

Given the opportunity to review the **TokenlonV5** 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 TokenlonV5

`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. `TokenlonV5` further advances earlier versions by re-architecting the protocol to seamlessly support external AMM offerings, such as `UniswapV2` and `Curve`.

The basic information of `TokenlonV5` is as follows:

Table 1.1: Basic Information of `TokenlonV5`

Item	Description
Issuer	Tokenlon
Website	<a href="https://tokenlon.im/">https://tokenlon.im/</a>
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 20, 2020

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

- <https://github.com/consenlabs/tokenlon-v5-sandbox/tree/master> (16af25a)
- <https://github.com/consenlabs/lon-token/tree/audit1> (bd94fd6)

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-v5-sandbox/tree/master> (519186b)
- <https://github.com/consenlabs/lon-token/tree/audit1> (2608502)

## 1.2 About PeckShield

PeckShield Inc. [12] 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](mailto:contact@peckshield.com)).

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [11]:

- 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) [10], 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 audit 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 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
<b>Configuration</b>	Weaknesses in this category are typically introduced during the configuration of the software.
<b>Data Processing Issues</b>	Weaknesses in this category are typically found in functionality that processes data.
<b>Numeric Errors</b>	Weaknesses in this category are related to improper calculation or conversion of numbers.
<b>Security Features</b>	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
<b>Time and State</b>	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.
<b>Error Conditions, Return Values, Status Codes</b>	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.
<b>Resource Management</b>	Weaknesses in this category are related to improper management of system resources.
<b>Behavioral Issues</b>	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
<b>Business Logics</b>	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.
<b>Initialization and Cleanup</b>	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
<b>Arguments and Parameters</b>	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
<b>Expression Issues</b>	Weaknesses in this category are related to incorrectly written expressions within code.
<b>Coding Practices</b>	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 TokenlonV5 implementation. During the first phase of our audit, we studied the smart contract source code and ran 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	2	
Low	2	
Informational	5	
Total	9	

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 2 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 5 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Inappropriate Subsidy Collection	Business Logic	Fixed
PVE-002	Informational	No ETH Support in emergencyWithdraw()	Business Logic	Fixed
PVE-003	Informational	Removal of Unused Code	Coding Practices	Fixed
PVE-004	Informational	Suggested Improvement on stakeWithPermit()	Business Logic	Confirmed
PVE-005	Medium	Trader-Controllable feeFactor, Not Protocol	Security Features	Fixed
PVE-006	Low	Potential DoS in AMMWrapper::transactionHash Generation	Coding Practices	Fixed
PVE-007	Informational	Inconsistency Between Document and Implementation	Coding Practices	Fixed
PVE-008	Informational	Suggested Reservation of 0 Index in curve-TokenIndexes	Coding Practices	Confirmed
PVE-009	Low	Trust Issue of Admin Keys Behind AllowanceTarget And Spender	Security Features	Mitigated

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 Inappropriate Subsidy Collection

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `AMMWrapper`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

#### Description

`Tokenlon` has traditionally provided a unique differentiating feature from other AMM- or orderbook-based exchanges in offering fixed prices for trading users. In other words, the proposed `Request For Quotation` exchange methodology brings a similar user experience with other over-the-counter (OTC) desk in an automated manner. With the new support of external AMM offerings, such as `UniswapV2` and `Curve`, `TokenlonV5` aims to offset intrinsic slippages in these AMM offerings by developing a so-called `subsidy` mechanism. This is an interesting feature with real benefits for trading users. However, our analysis shows the provided `subsidy` may be inappropriately collected by malicious actors.

To elaborate, we show below the related `_swap()` and `_settle()` routines in `AMMWrapper`. The first routine executes the intended swap operation and the second one computes the actual settlement amount for the trading user. The `subsidy` calculation occurs in the second routine.

```
311  /**
312   * @dev internal function of 'trade'.
313   * It executes the swap on chosen AMM.
314   */
315  function _swap(
316      bool makerIsUniV2,
317      address _makerAddr,
318      address _takerAssetAddr,
319      address _makerAssetAddr,
320      uint256 _takerAssetAmount,
321      uint256 _makerAssetAmount,
322      uint256 _deadline,
```

```

323     uint256 _subsidyFactor
324 ) internal returns (string memory source, uint256 receivedAmount) {
325     // Approve
326     IERC20(_takerAssetAddr).safeApprove(_makerAddr, _takerAssetAmount);
327
328     // Swap
329     // minAmount = makerAssetAmount * (10000 - subsidyFactor) / 10000
330     uint256 minAmount = _makerAssetAmount.mul((BPS_MAX.sub(_subsidyFactor)).div(
331         BPS_MAX);
332     if (makerIsUniV2) {
333         source = "Uniswap V2";
334         receivedAmount = _tradeUniswapV2TokenToToken(_takerAssetAddr,
335             _makerAssetAddr, _takerAssetAmount, minAmount, _deadline);
336     } else {
337         int128 fromTokenCurveIndex = permStorage.getCurveTokenIndex(_makerAddr,
338             _takerAssetAddr);
339         int128 toTokenCurveIndex = permStorage.getCurveTokenIndex(_makerAddr,
340             _makerAssetAddr);
341         if (fromTokenCurveIndex != 0 toTokenCurveIndex != 0) {
342             source = "Curve";
343             uint256 balanceBeforeTrade = IERC20(_makerAssetAddr).balanceOf(address(
344                 this));
345             _tradeCurveTokenToToken(_makerAddr, fromTokenCurveIndex,
346                 toTokenCurveIndex, _takerAssetAmount, minAmount);
347             uint256 balanceAfterTrade = IERC20(_makerAssetAddr).balanceOf(address(
348                 this));
349             receivedAmount = balanceAfterTrade.sub(balanceBeforeTrade);
350         } else {
351             revert("AMMWrapper: Unsupported makerAddr");
352         }
353     }
354
355     // Close allowance
356     IERC20(_takerAssetAddr).safeApprove(_makerAddr, 0);
357 }

```

Listing 3.1: AMMWrapper::\_swap()

Suppose the AMMWrapper contract has  $S_{DAI}$  DAI balance gradually collected from earlier trades in terms of trading fees, i.e.,  $DAI.balanceOf(AMMWrapper) = S_{DAI}$ . To collect the  $S_{DAI}$  DAI balance in AMMWrapper, a malicious actor can craft a trading request by setting the `makerAssetAddr` as DAI and `makerAssetAmount` (to be computed later). For simplicity, we assume the system-wide parameter `subsidyFactor`. After that, the following steps are executed:

1. Issue a new ERC20 token – `tokenA`, and create `tokenA/DAI` pair in `UniswapV2`;
2. Initialize the pool by adding the same amount of DAI and `tokenA` so that  $1 \text{ tokenA} = 1 \text{ DAI}$ . Note since it is a new pair, the actor can arbitrarily initialize the price between the two tokens;
3. Compute the required `tokenA` input to exchange for `receivedAmount = makerAssetAmount / (1 +`

subsidyFactor) of DAI – such calculation allows to bypass the `inSubsidyRange` check at line 387; Note this step can be calculated from the `UniswapV2Library::getAmountIn()` helper routine<sup>1</sup>.

4. Execute the subsidy-collection branch (lines 381 – 392) by calculating the settlement amount  $\text{settleAmount} = \text{makerAssetAmount} = \text{newBalanceOfDAI}(\text{AMMWrapper}) = \text{receivedAmount} + S_{DAI}$ . In other words,  $\text{makerAssetAmount} = (1 + 1/\text{subsidyFactor}) * S_{DAI}$ .

```

352  /**
353   * @dev internal function of 'trade'.
354   * It collects fee from the trade or compensates the trade based on the actual
      amount swapped.
355   */
356   function _settle(
357       bool _toEth,
358       IWETH with,
359       IERC20 _makerAsset,
360       uint256 _makerAssetAmount,
361       uint256 _receivedAmount,
362       uint256 _feeFactor,
363       uint256 _subsidyFactor,
364       address payable _receiverAddr
365   )
366   {
367       internal
368       returns (uint256 settleAmount)
369   {
370       if (_receivedAmount == _makerAssetAmount) {
371           settleAmount = _receivedAmount;
372       } else if (_receivedAmount > _makerAssetAmount) {
373           // shouldCollectFee = ((receivedAmount - makerAssetAmount) / receivedAmount)
374           > (feeFactor / 10000)
375           bool shouldCollectFee = _receivedAmount.sub(_makerAssetAmount).mul(BPS_MAX)
376           > _feeFactor.mul(_receivedAmount);
377           if (shouldCollectFee) {
378               // settleAmount = receivedAmount * (1 - feeFactor) / 10000
379               settleAmount = _receivedAmount.mul(BPS_MAX.sub(_feeFactor)).div(BPS_MAX)
380           }
381       } else {
382           settleAmount = _makerAssetAmount;
383       }
384   } else {
385       // If fee factor is smaller than subsidy factor, choose fee factor as actual
386       subsidy factor
387       // since we should subsidize less if we charge less.
388       uint256 actualSubsidyFactor = (_subsidyFactor < _feeFactor) ? _subsidyFactor
389       : _feeFactor;
390       // inSubsidyRange = ((makerAssetAmount - receivedAmount) / receivedAmount) >
391       (actualSubsidyFactor / 10000)

```

<sup>1</sup>The required input  $\Delta X$  can be computed as  $(X \cdot \Delta Y) / (Y - \Delta Y)$  where  $\Delta Y = \text{receivedAmount}$ .

```

386     bool inSubsidyRange = _makerAssetAmount.sub(_receivedAmount).mul(BPS_MAX) <=
        actualSubsidyFactor.mul(_receivedAmount);
387     require(inSubsidyRange, "AMMWrapper: amount difference larger than subsidy
        amount");
388
389     bool hasEnoughToSubsidize = (_makerAsset.balanceOf(address(this)) >=
        _makerAssetAmount);
390     require(hasEnoughToSubsidize, "AMMWrapper: not enough savings to subsidize")
        ;
391
392     settleAmount = _makerAssetAmount;
393 }
394
395 // Transfer token/Eth to receiver
396 if (_toEth) {
397     weth.withdraw(settleAmount);
398     _receiverAddr.transfer(settleAmount);
399 } else {
400     _makerAsset.safeTransfer(_receiverAddr, settleAmount);
401 }
402 }

```

Listing 3.2: AMMWrapper::\_settle()

After executing the above steps, the final `settleAmount` will simply transfer the full DAI balance in `AMMWrapper` to the malicious actor as subsidy. This is certainly unintended and should be blocked.

**Recommendation** Revisit the `subsidy` mechanism so that it will not be inappropriately collected by attackers.

**Status** The issue has been fixed in this commit: [e4448e5](#).

## 3.2 No ETH Support in `emergencyWithdraw()`

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

### Description

To mitigate real-world scenarios where users may accidentally send tokens to various contracts, `TokenlonV5` has been thoughtfully designed in allowing for `emergencyWithdraw()` to a designated recipient. This recipient can later recover the accidentally-transferred funds back to the user.

Using the `Lon` contract as an example, we show below the `emergencyWithdraw()` routine. And the recipient has been implemented as a dedicated contract, i.e., `EmergencyRecipient`.

```
71 function emergencyWithdraw(IERC20 token) external override {
72     token.transfer(emergencyRecipient, token.balanceOf(address(this)));
73 }
```

Listing 3.3: `Lon::emergencyWithdraw()`

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.6.0 <0.8.0;

4 import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
5 import "../Ownable.sol";

7 contract EmergencyRecipient is Ownable {

9     constructor(address _owner) Ownable(_owner) {
10    }

12     function sendToken(IERC20 token, address recipient, uint256 amount) external
        onlyOwner {
13         token.transfer(recipient, amount);
14     }
15 }
```

Listing 3.4: `EmergencyRecipient::sendToken()`

This is certainly a nice feature that will be welcomed by the community. In the meantime, we notice that it only supports various ERC20-compliant tokens. It is also helpful to support the recovery of accidentally-sent ETH as well.

**Recommendation** Add the ETH support for emergency recovery.

**Status** The issue has been fixed in this commit: [21ea26f](#).

### 3.3 Removal of Unused Code

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [5]

#### Description

`TokenlonV5` makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `SafeMath`, `TransparentUpgradeableProxy`, and `Ownable`, to facilitate its code implementation and organization. For

example, the `AMMWrapper` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `TreasuryVesterFactory` contract, it is defined as an `Ownable` contract (line 7). However, the `owner` account is not used throughout the contract and therefore can be removed.

```

1  // SPDX-License-Identifier: MIT
2  pragma solidity >=0.6.0 <0.8.0;
3
4  import "../Ownable.sol";
5  import "../TreasuryVester.sol";
6
7  contract TreasuryVesterFactory is Ownable {
8      IERC20 public lon;
9
10     event VesterCreated(address indexed vester, address indexed recipient, uint256
        vestingAmount);
11
12     constructor(address _owner, IERC20 _lon) Ownable(_owner) {
13         lon = _lon;
14     }
15
16     function createVester(
17         address recipient,
18         uint256 vestingAmount,
19         uint256 vestingBegin,
20         uint256 vestingCliff,
21         uint256 vestingEnd) external returns(address) {
22         require(vestingAmount > 0, "vesting amount is zero");
23
24         address vester = address(new TreasuryVester(address(lon), recipient,
            vestingAmount, vestingBegin, vestingCliff, vestingEnd));
25
26         lon.transferFrom(msg.sender, vester, vestingAmount);
27
28         emit VesterCreated(vester, recipient, vestingAmount);
29
30         return vester;
31     }
32 }

```

Listing 3.5: `TreasuryVesterFactory.sol`

In the same vein, we notice `Ownable` is not used either in the `StakingRewards` contract.

Moreover, if we examine the `UserProxy` contract, several constants and imports are not needed. For example, the constants `ETH_ADDRESS` and `ZERO_ADDRESS` as well as the state `permStorage` can be safely removed. The imports of `IERC20`, `SafeERC20`, `IPMM`, and `UserProxyStorage` are not necessary either.



```

1 // SPDX-License-Identifier: MIT
2
3 pragma solidity ^0.6.5;
4
5 import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
6 import "@openzeppelin/contracts/token/ERC20/SafeERC20.sol";
7 import "../interface/IPMM.sol";
8 import "../interface/IPermanentStorage.sol";
9 import "../utils/lib_storage/UserProxyStorage.sol";
10
11 /**
12  * @dev UserProxy contract
13  */
14 contract UserProxy {
15     using SafeERC20 for IERC20;
16
17     // Constants do not have storage slot.
18     address private constant ETH_ADDRESS = 0xEeeeeEeeeEeEeeEeEeEeEeEeEeEeEeEeE;
19     address private constant ZERO_ADDRESS = address(0);
20
21     // Below are the variables which consume storage slots.
22     address public operator;
23     string public version; // Current version of the contract
24     IPermanentStorage public permStorage;
25
26     ...
27 }

```

Listing 3.6: UserProxy.sol

Similarly, the constant ETH\_ADDRESS is not used either in PMM (line 35) and the computed EIP712\_DOMAIN\_HASH (lines 104 – 114) can be removed as well.

**Recommendation** Consider the removal of the unused code and the unused constants.

**Status** The issue has been fixed in the following related commits: 21ea26f and c994dc5.

### 3.4 Suggested Improvement on stakeWithPermit()

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: StakingRewards
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

#### Description

The audited `lon-token` repository contains a `StakingRewards` contract that allows users to stake supported assets for `LOX` rewards. While it implements a rather standard functionality, it makes a step further in simplifying the steps required for staking users. Specifically, a staking user typically requires calling `approve()` to specify the spending allowance of the `StakingRewards` contract before calling the `stake()` function. This implies two steps from the user perspective: `approve()` and `stake()`.

To make the staking process as smooth as possible, a new function called `stakeWithPermit()` is introduced. It effectively combines the `approve()` step and the `stake()` step into one by making use of the `permit()` helper routine. For illustration, we show below the `stakeWithPermit()` routine.

```

85     function stakeWithPermit(uint256 amount, uint deadline, uint8 v, bytes32 r, bytes32
      s) external nonReentrant updateReward(msg.sender) {
86         require(amount > 0, "cannot stake 0");
87         _totalSupply = _totalSupply.add(amount);
88         _balances[msg.sender] = _balances[msg.sender].add(amount);
89
90         // permit
91         IEIP2612(address(stakingToken)).permit(msg.sender, address(this), amount,
          deadline, v, r, s);
92
93         stakingToken.safeTransferFrom(msg.sender, address(this), amount);
94         emit Staked(msg.sender, amount);
95     }

```

Listing 3.7: `StakingRewards::stakeWithPermit()`

Its execution logic is rather straightforward in using `permit()` to specify the allowance before actually `transferFrom()` for staking. We notice that the `permit()` call uses `msg.sender` as the first argument, it is worthwhile to explore the possibility to make the staking as generic as possible.

**Recommendation** Suggest to allow for more broader applicability of `stakeWithPermit()` in not restricting `msg.sender` only.

```

85     function stakeWithPermit(address owner, uint256 amount, uint deadline, uint8 v,
      bytes32 r, bytes32 s) external nonReentrant updateReward(msg.sender) {
86         require(amount > 0, "cannot stake 0");
87         _totalSupply = _totalSupply.add(amount);

```

```

88     _balances[msg.sender] = _balances[owner].add(amount);
89
90     // permit
91     IEIP2612(address(stakingToken)).permit(owner, address(this), amount, deadline, v
        , r, s);
92
93     stakingToken.safeTransferFrom(owner, address(this), amount);
94     emit Staked(owner, amount);
95 }

```

Listing 3.8: Revised StakingRewards::stakeWithPermit()

**Status** This issue has been extensively discussed with the team. With the current goal of only making the `stake()` call for `msg.sender` within one single step, the broader applicability is not part of the design goal.

### 3.5 Trader-Controllable feeFactor, Not Protocol

- ID: PVE-005
- Severity: Medium
- Likelihood: High
- Impact: Low
- Targets: AMMWrapper, PMM
- Category: Security Features [7]
- CWE subcategory: CWE-282 [3]

#### Description

According to the TokenlonV5 design, the trading fees are collected in `AMMWrapper` and `PMM` contracts and can be collected by authorized accounts. When examining the way the trading fees are collected, we notice a protocol-wide risk parameter, i.e., `feeFactor`.

In the following, we show the code snippet of the `PMM::_settle()` routine. This routine is designed to calculate the final settlement after completing the swap operation with market makers. The settlement calculation takes into account the `feeFactor` risk parameter.

```

278 // settle
279 function _settle(IWETH with, address receiver, address makerAssetAddr, uint256
    makerAssetAmount, uint16 feeFactor) internal returns(uint256) {
280     uint256 settleAmount = makerAssetAmount;
281     if (feeFactor > 0) {
282         // settleAmount = settleAmount * (10000 - feeFactor) / 10000
283         settleAmount = settleAmount.mul((BPS_MAX).sub(feeFactor)).div(BPS_MAX);
284     }
285
286     if (makerAssetAddr == address(with)){
287         with.withdraw(settleAmount);
288         payable(receiver).transfer(settleAmount);

```

```

289     } else {
290         IERC20(makerAssetAddr).safeTransfer(receiver, settleAmount);
291     }

293     return settleAmount;
294 }

```

Listing 3.9: PMM::\_settle()

Our analysis shows this `feeFactor` in effect is not specified by the protocol. Instead, it is given as part of (untrusted) input arguments. To avoid the fee payment, a trading user is at the liberty of using 0 as the `feeFactor`, undermining the purpose of collecting trading fees for `LON` buy-back.

The same issue is also applicable to `AMMWrapper`.

**Recommendation** Enforce the protocol-wide `feeFactor` by disallowing trading users to specify the parameter.

**Status** The issue has been fixed in this commit: [e4448e5](#).

## 3.6 Potential DoS in `AMMWrapper::transactionHash` Generation

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Targets: `AMMWrapper`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1116 [2]

### Description

With the seamless integration of AMM-based exchange systems (e.g., `UniswapV2` and `Curve`), `TokenlonV5` transparently supports two types of market makers, i.e., `AMMWrapper` and `PMM`. The first type wraps the new AMM-based exchange systems while the second type supports the traditional (professional) market makers.

While examining the full support of new AMM-based exchange systems, we notice the anti-replay mechanism can be better improved. Specifically, the anti-replay mechanism firstly calculates a transaction hash for trade verification and prevents a previous transaction hash from being re-used or replayed. In the following, we show the `_prepare()` routine that calculates and validates the transaction hash.

```

252     /**
253     * @dev internal function of 'trade'.

```

```

254     * It verifies user signature, transfer tokens from user and store tx hash to
      prevent replay attack.
255     */
256     function _prepare(
257         bool fromEth,
258         IWETH weth,
259         address _makerAddr,
260         address _takerAssetAddr,
261         address _makerAssetAddr,
262         uint256 _takerAssetAmount,
263         uint256 _makerAssetAmount,
264         address _userAddr,
265         address _receiverAddr,
266         uint256 _salt,
267         uint256 _deadline,
268         bytes memory _sig
269     ) internal returns (bytes32 transactionHash) {
270         // Verify user signature
271         // TRADE_WITH_PERMIT_TYPEHASH = keccak256("tradeWithPermit(address makerAddr,
          address takerAssetAddr,address makerAssetAddr,uint256 takerAssetAmount,
          uint256 makerAssetAmount,address receiverAddr,uint256 salt,uint256 deadline)
          ");
272         transactionHash = keccak256(
273             abi.encode(
274                 TRADE_WITH_PERMIT_TYPEHASH,
275                 _makerAddr,
276                 _takerAssetAddr,
277                 _makerAssetAddr,
278                 _takerAssetAmount,
279                 _makerAssetAmount,
280                 _receiverAddr,
281                 _salt,
282                 _deadline
283             )
284         );
285         bytes32 EIP712SignDigest = keccak256(
286             abi.encodePacked(
287                 bytes1(0x19),
288                 bytes1(0x01),
289                 EIP712_DOMAIN_SEPARATOR,
290                 transactionHash
291             )
292         );
293         require(isValidSignature(_userAddr, EIP712SignDigest, bytes(""), _sig), "
          AMMWrapper: invalid user signature");

295         // Transfer asset from user and deposit to weth if needed
296         if (fromEth) {
297             require(msg.value > 0, "AMMWrapper: msg.value is zero");
298             require(_takerAssetAmount == msg.value, "AMMWrapper: msg.value doesn't match
              ");
299             // Deposit ETH to weth

```

```

300         weth.deposit{value: msg.value}();
301     } else {
302         spender.spendFromUser(_userAddr, _takerAssetAddr, _takerAssetAmount);
303     }

305     // Validate that the transaction is not seen before
306     require(! permStorage.isTransactionSeen(transactionHash), "AMMWrapper:
        transaction seen before");
307     // Set transaction as seen
308     permStorage.setTransactionSeen(transactionHash);
309 }

```

Listing 3.10: AMMWrapper::\_prepare()

In particular, the transaction hash is calculated as `transactionHash = keccak256(abi.encode(TRADE_WITH_PERMIT_TYPEHASH, _makerAddr, _takerAssetAddr, _makerAssetAmount, _takerAssetAmount, _receiverAddr, _salt, _deadline))` (lines 272 – 284). It comes to our attention that the `_userAddr` information is not part of the calculated hash. Note the related assets of the transaction are actually transferred out of `_userAddr`.

**Recommendation** Take `_userAddr` into account in the calculation of `transactionHash` for replay prevention.

**Status** The issue has been fixed in this commit: `e23dab1`.

### 3.7 Inconsistency Between Document and Implementation

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [8]
- CWE subcategory: CWE-1041 [1]

#### Description

There are a few inconsistent or misleading descriptions in the given documentations or files, which bring unnecessary hurdles to understand and/or maintain the protocol implementation.

A few example comments can be found in documenting the `UserProxy::initialize()` routine, as well as the `PMM` contract. Specifically, it has been documented that the `UserProxy::initialize()` routine takes two arguments, i.e., `_operator` and `_permStorage`. However, the current implementation only takes one argument, i.e., `_operator`. For comparison, we show the full implementation of `UserProxy::initialize()` below.

```

43  /*****

```

```

44      *          Constructor and init functions          *
45      *****/
46      /// @dev Replacing constructor and initialize the contract. This function should
          only be called once.
47      function initialize(address _operator) external {
48          require(_operator != address(0), "UserProxy: _operator should not be 0");
49          require(
50              keccak256(abi.encodePacked(version)) == keccak256(abi.encodePacked("")),
51              "UserProxy: not upgrading from default version"
52          );
53
54          // Set operator
55          operator = _operator;
56          // Upgrade version
57          version = "5.0.0";
58      }

```

Listing 3.11: UserProxy:: initialize ()

Also, in the description of PMM, it indicates a whitelist of all authorized market makers. However, there is no such whitelist in the current implementation.

**Recommendation** Ensure the consistency between documents and implementation.

**Status** This issue has been fixed by updating the corresponding documentations.

### 3.8 Suggested Reservation of 0 Index in curveTokenIndexes

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: PermanentStorage
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [5]

#### Description

As mentioned in Section 3.6, TokenlonV5 transparently supports two types of market makers, i.e., AMMWrapper and PMM. And the integration of AMM-based exchange systems supports both UniswapV2 and Curve. When examining the support of Curve, we notice a possible improvement to better maintain the mapping between the supported Curve pools and their underlying assets.

Specifically, we show below the code snippet of `_swap()` routine in AMMWrapper. This routine is tasked with performing the actual swap with a chosen AMM, i.e., UniswapV2 or Curve. In the case of Curve, the source asset for swap is represented as `fromTokenCurveIndex` (line 335) and the target asset is shown as `toTokenCurveIndex` (line 336). Both are indexes maintained in `permStorage`.

```

311  /**
312   * @dev internal function of 'trade'.
313   * It executes the swap on chosen AMM.
314   */
315  function _swap(
316      bool makerIsUniV2,
317      address _makerAddr,
318      address _takerAssetAddr,
319      address _makerAssetAddr,
320      uint256 _takerAssetAmount,
321      uint256 _makerAssetAmount,
322      uint256 _deadline,
323      uint256 _subsidyFactor
324  ) internal returns (string memory source, uint256 receivedAmount) {
325      // Approve
326      IERC20(_takerAssetAddr).safeApprove(_makerAddr, _takerAssetAmount);
327
328      // Swap
329      // minAmount = makerAssetAmount * (10000 - subsidyFactor) / 10000
330      uint256 minAmount = _makerAssetAmount.mul((BPS_MAX.sub(_subsidyFactor)).div(
331          BPS_MAX);
332      if (makerIsUniV2) {
333          source = "Uniswap V2";
334          receivedAmount = _tradeUniswapV2TokenToToken(_takerAssetAddr,
335              _makerAssetAddr, _takerAssetAmount, minAmount, _deadline);
336      } else {
337          int128 fromTokenCurveIndex = permStorage.getCurveTokenIndex(_makerAddr,
338              _takerAssetAddr);
339          int128 toTokenCurveIndex = permStorage.getCurveTokenIndex(_makerAddr,
340              _makerAssetAddr);
341          if (fromTokenCurveIndex != 0 || toTokenCurveIndex != 0) {
342              source = "Curve";
343              uint256 balanceBeforeTrade = IERC20(_makerAssetAddr).balanceOf(address(
344                  this));
345              _tradeCurveTokenToToken(_makerAddr, fromTokenCurveIndex,
346                  toTokenCurveIndex, _takerAssetAmount, minAmount);
347              uint256 balanceAfterTrade = IERC20(_makerAssetAddr).balanceOf(address(
348                  this));
349              receivedAmount = balanceAfterTrade.sub(balanceBeforeTrade);
350          } else {
351              revert("AMMWrapper: Unsupported makerAddr");
352          }
353      }
354
355      // Close allowance
356      IERC20(_takerAssetAddr).safeApprove(_makerAddr, 0);
357  }

```

Listing 3.12: AMMWrapper::\_swap()

We point out that the index in `permStorage` starts from 0 as the base, which unfortunately is the default value if uninitialized or unmapped. In other words, there is no way to tell whether the



mapping is valid or simply not initialized. To avoid that, we suggest the reservation of 0 index in `curveTokenIndexes` in `permStorage`.

In particular, if we examine the corresponding `setCurveTokenIndex()` routine, the assigned index can be internally incremented by 1 and a non-0 comparison can be used to validate a mapped token index in `Curve`. In other words, the validation at line 337, i.e., `if (fromTokenCurveIndex != 0 || toTokenCurveIndex != 0)` can be tightened as `if (fromTokenCurveIndex != 0 && toTokenCurveIndex != 0)`. When being used to interact with the intended `Curve` pool, we can correspondingly decrement the index by 1.

```

117  /*****
118  *                               *
119  *****/
120  function setCurveTokenIndex(address _makerAddr, address[] calldata _assetAddrs)
121      override external isPermitted(curveTokenIndexStorageId, msg.sender) {
122      int128 tokenLength = int128(_assetAddrs.length);
123      for (int128 i = 0 ; i < tokenLength; i++) {
124          address assetAddr = _assetAddrs[uint256(i)];
125          AMMWrapperStorage.getStorage().curveTokenIndexes[_makerAddr][assetAddr] = i;
126      }

```

Listing 3.13: `permStorage::setCurveTokenIndex()`

**Recommendation** Reserve the 0 index in `curveTokenIndexes` and ensure the actual index starts from 1.

**Status** This issue has been discussed with the team. For the consistency with the widely-used index mapping in `Curve`, the team decides to leave it as is.

### 3.9 Trust Issue of Admin Keys Behind AllowanceTarget And Spender

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: AllowanceTarget, Spender
- Category: Security Features [7]
- CWE subcategory: CWE-287 [4]

#### Description

In `TokenlonV5`, 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.

```

59  /// @dev Execute an arbitrary call. Only an authority can call this.
60  /// @param target The call target.
61  /// @param callData The call data.
62  /// @return resultData The data returned by the call.
63  function executeCall(
64      address payable target,
65      bytes calldata callData
66  )
67      override
68      external
69      onlySpender
70      returns (bytes memory resultData)
71  {
72      bool success;
73      (success, resultData) = target.call(callData);
74      if (!success) {
75          // Get the error message returned
76          assembly {
77              let ptr := mload(0x40)
78              let size := returndatasize()
79              returndatacopy(ptr, 0, size)
80              revert(ptr, size)
81          }
82      }
83  }

```

Listing 3.14: `AllowanceTarget::executeCall()`

```

59  /// @dev Spend tokens on user's behalf. Only an authority can call this.
60  /// @param _user The user to spend token from.
61  /// @param _tokenAddr The address of the token.
62  /// @param _amount Amount to spend.
63  function spendFromUser(address _user, address _tokenAddr, uint256 _amount) external
64      onlyAuthorized {
65      require(! tokenBlacklist[_tokenAddr], "Spender: token is blacklisted");
66
67      if (_tokenAddr != ETH_ADDRESS && _tokenAddr != ZERO_ADDRESS) {
68          uint256 balanceBefore = IERC20(_tokenAddr).balanceOf(msg.sender);
69          (bool callSucceed, ) = address(allowanceTarget).call(
70              abi.encodeWithSelector(
71                  IAllowanceTarget.executeCall.selector,
72                  _tokenAddr,
73                  abi.encodeWithSelector(
74                      IERC20.transferFrom.selector,
75                      _user,
76                      msg.sender,

```

```

77         _amount
78     )
79 )
80 );
81 require(callSucceed, "Spender: ERC20 transferFrom failed");
82 // Check balance
83 uint256 balanceAfter = IERC20(_tokenAddr).balanceOf(msg.sender);
84 require(balanceAfter.sub(balanceBefore) == _amount, "Spender: ERC20
    transferFrom result mismatch");
85
86 }
87 }

```

Listing 3.15: 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.

## 3.10 Other Suggestions

Due to the fact that compiler upgrades might bring unexpected compatibility or inter-version inconsistencies, it is always suggested to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.6.0;` instead of `pragma solidity >=0.6.0 <0.8.0.`

Moreover, we strongly suggest not to use experimental Solidity features (e.g., `pragma experimental ABIEncoderV2`) or third-party unaudited libraries. If necessary, refactor current code base to only use stable features or trusted libraries.

## 4 | Conclusion

In this audit, we have analyzed the TokenlonV5 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. TokenlonV5 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-1041: Use of Redundant Code. <https://cwe.mitre.org/data/definitions/1041.html>.
- [2] MITRE. CWE-1116: Inaccurate Comments. <https://cwe.mitre.org/data/definitions/1116.html>.
- [3] MITRE. CWE-282: Improper Ownership Management. <https://cwe.mitre.org/data/definitions/282.html>.
- [4] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [5] MITRE. CWE-563: Assignment to Variable without Use. <https://cwe.mitre.org/data/definitions/563.html>.
- [6] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [7] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [8] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [9] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.

- [10] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [11] OWASP. Risk Rating Methodology. [https://www.owasp.org/index.php/OWASP\\_Risk\\_Rating\\_Methodology](https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology).
- [12] PeckShield. PeckShield Inc. <https://www.peckshield.com>.

