SMART CONTRACT AUDIT REPORT

for

TOKENLON

Prepared By: Shuxiao Wang

PeckShield

March 26, 2021

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0-rc	March 26, 2021	Xuxian Jiang	Release Candidate #1
0.4	March 20, 2021	Xuxian Jiang	Additional Findings #3
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

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	https://tokenlon.im/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 26, 2021

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.git (16af25a)
- https://github.com/consenlabs/lon-token.git (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 (519186b)
- https://github.com/consenlabs/lon-token.git (607d625)

1.2 About PeckShield

PeckShield Inc. [17] 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 [16]:

- 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) [15], 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
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 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	4	■■■■
Low	3	■■■
Informational	6	■■■■■■
Total	13	

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 4 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 6 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 curveTokenIndexes	Coding Practices	Confirmed
PVE-009	Low	Trust Issue of Admin Keys Behind AllowanceTarget And Spender	Security Features	Mitigated
PVE-010	Medium	Bypass of Cooldown Restriction in LONStaking	Business Logic	Fixed
PVE-011	Low	Possible Costly LPs From Improper Vault Initialization	Time and State	Confirmed
PVE-012	Medium	Potential Sandwich Attacks To Minimize swappedLonAmount	Time and State	Fixed
PVE-013	Informational	Lack of lastTimeBuyback Initialization in RewardDistributor	Coding Practices	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Inappropriate Subsidy Collection

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `AMMWrapper`
- Category: Business Logic [13]
- CWE subcategory: CWE-841 [9]

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(
                BPS_MAX);
331         if (makerIsUniV2) {
332             source = "Uniswap V2";
333             receivedAmount = _tradeUniswapV2TokenToToken(_takerAssetAddr,
                    _makerAssetAddr, _takerAssetAmount, minAmount, _deadline);
334         } else {
335             int128 fromTokenCurveIndex = permStorage.getCurveTokenIndex(_makerAddr,
                    _takerAssetAddr);
336             int128 toTokenCurveIndex = permStorage.getCurveTokenIndex(_makerAddr,
                    _makerAssetAddr);
337             if (fromTokenCurveIndex != 0  toTokenCurveIndex != 0) {
338                 source = "Curve";
339                 uint256 balanceBeforeTrade = IERC20(_makerAssetAddr).balanceOf(address(
                        this));
340                 _tradeCurveTokenToToken(_makerAddr, fromTokenCurveIndex,
                        toTokenCurveIndex, _takerAssetAmount, minAmount);
341                 uint256 balanceAfterTrade = IERC20(_makerAssetAddr).balanceOf(address(
                        this));
342                 receivedAmount = balanceAfterTrade.sub(balanceBeforeTrade);
343             } else {
344                 revert("AMMWrapper: Unsupported makerAddr");
345             }
346         }
347 
348         // Close allowance
349         IERC20(_takerAssetAddr).safeApprove(_makerAddr, 0);
350     }
```

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 tokenA = 1 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[1].

4. Execute the subsidy-collection branch (lines 381 – 392) by calculating the settlement amount settleAmount = makerAssetAmount = newBalanceOfDAI(AMMWrapper)= receivedAmount + S_{DAI}. In other words, makerAssetAmount = (1+1/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 weth,
359         IERC20 _makerAsset,
360         uint256 _makerAssetAmount,
361         uint256 _receivedAmount,
362         uint256 _feeFactor,
363         uint256 _subsidyFactor,
364         address payable _receiverAddr
365     )
366         internal
367         returns (uint256 settleAmount)
368     {
369         if (_receivedAmount == _makerAssetAmount) {
370             settleAmount = _receivedAmount;
371         } else if (_receivedAmount > _makerAssetAmount) {
372             // shouldCollectFee = ((receivedAmount - makerAssetAmount) / receivedAmount)
                     > (feeFactor / 10000)
373             bool shouldCollectFee = _receivedAmount.sub(_makerAssetAmount).mul(BPS_MAX)
                 > _feeFactor.mul(_receivedAmount);
374             if (shouldCollectFee) {
375                 // settleAmount = receivedAmount * (1 - feeFactor) / 10000
376                 settleAmount = _receivedAmount.mul(BPS_MAX.sub(_feeFactor)).div(BPS_MAX)
                     ;
377             } else {
378                 settleAmount = _makerAssetAmount;
379             }
380         } else {
381             // If fee factor is smaller than subsidy factor, choose fee factor as actual
                  subsidy factor
382             // since we should subsidize less if we charge less.
383             uint256 actualSubsidyFactor = (_subsidyFactor < _feeFactor) ? _subsidyFactor
                  : _feeFactor;
384
385             // inSubsidyRange = ((makerAssetAmount - receivedAmount) / receivedAmount) >
                  (actualSubsidyFactor / 10000)
```

[1]The required input ΔX can be computed as $(X \bullet \Delta Y)/(Y - \Delta Y)$ where ΔY = 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 [13]
- CWE subcategory: CWE-841 [9]

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;
3
4  import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
5  import "./Ownable.sol";
6
7  contract EmergencyRecipient is Ownable {
8
9      constructor(address _owner) Ownable(_owner) {
10     }
11
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 [12]
- CWE subcategory: CWE-563 [6]

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 = 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE;
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 [13]
- CWE subcategory: CWE-841 [9]

Description

The audited `lon-token` repository contains a `StakingRewards` contract that allows users to stake supported assets for `LON` 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);

90          // permit
91          IEIP2612(address(stakingToken)).permit(msg.sender, address(this), amount,
                deadline, v, r, s);

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);

90          // permit
91          IEIP2612(address(stakingToken)).permit(owner, address(this), amount, deadline, v
                , r, s);

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 [10]
- 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 weth, 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         }

286         if (makerAssetAddr == address(weth)){
287             weth.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 [12]
- 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         }
304
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, _makerAssetAddr, _takerAssetAmount, _makerAssetAmount, _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 [12]
- 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 [12]
- CWE subcategory: CWE-563 [6]

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(
                BPS_MAX);
331         if (makerIsUniV2) {
332             source = "Uniswap V2";
333             receivedAmount = _tradeUniswapV2TokenToToken(_takerAssetAddr,
                    _makerAssetAddr, _takerAssetAmount, minAmount, _deadline);
334         } else {
335             int128 fromTokenCurveIndex = permStorage.getCurveTokenIndex(_makerAddr,
                    _takerAssetAddr);
336             int128 toTokenCurveIndex = permStorage.getCurveTokenIndex(_makerAddr,
                    _makerAssetAddr);
337             if (fromTokenCurveIndex != 0 || toTokenCurveIndex != 0) {
338                 source = "Curve";
339                 uint256 balanceBeforeTrade = IERC20(_makerAssetAddr).balanceOf(address(
                        this));
340                 _tradeCurveTokenToToken(_makerAddr, fromTokenCurveIndex,
                        toTokenCurveIndex, _takerAssetAmount, minAmount);
341                 uint256 balanceAfterTrade = IERC20(_makerAssetAddr).balanceOf(address(
                        this));
342                 receivedAmount = balanceAfterTrade.sub(balanceBeforeTrade);
343             } else {
344                 revert("AMMWrapper: Unsupported makerAddr");
345             }
346         }
347 
348         // Close allowance
349         IERC20(_takerAssetAddr).safeApprove(_makerAddr, 0);
350     }
```

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     *                   External functions                     *
119     ***********************************************************/
120     function setCurveTokenIndex(address _makerAddr, address[] calldata _assetAddrs)
            override external isPermitted(curveTokenIndexStorageId, msg.sender) {
121         int128 tokenLength = int128(_assetAddrs.length);
122         for (int128 i = 0 ; i < tokenLength; i++) {
123             address assetAddr = _assetAddrs[uint256(i)];
124             AMMWrapperStorage.getStorage().curveTokenIndexes[_makerAddr][assetAddr] = i;
125         }
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 [10]
- 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
            onlyAuthorized {
64          require(!tokenBlacklist[_tokenAddr], "Spender: token is blacklisted");
65
66          if (_tokenAddr != ETH_ADDRESS && _tokenAddr != ZERO_ADDRESS) {
67
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 Bypass of Cooldown Restriction in LONStaking

- ID: PVE-010
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `LONStaking`
- Category: Business Logic [13]
- CWE subcategory: CWE-841 [9]

Description

To engage protocol users, the `LONStaking` contract has been designed to reward participating users if they stake their tokens to receive pro-rata staking rewards. In order to prevent possible flashloan-assisted front-running attacks that may claim the majority of rewards, the staking logic is designed

to have a cooldown period for staked assets. For each account, the associated cooldown period is recorded internally as `stakersCooldowns[account]`.

When there is a stake operation, the staking user's cooldown timestamp is properly updated. When the pool token is transferred, the receiver's cooldown timestamp will also be updated. The new cooldown timestamp is calculated in the following `_getNextCooldownTimestamp()` routine.

```
144     function _getNextCooldownTimestamp(
145         uint256 fromCooldownTimestamp,
146         uint256 amountToReceive,
147         address toAddress,
148         uint256 toBalance
149     ) internal returns (uint256) {
150         uint256 toCooldownTimestamp = stakersCooldowns[toAddress];
151         if (toCooldownTimestamp == 0) {
152             return 0;
153         }
154 
155         if (fromCooldownTimestamp <= toCooldownTimestamp) {
156             return toCooldownTimestamp;
157         } else {
158             toCooldownTimestamp = (
159                 amountToReceive.mul(fromCooldownTimestamp).add(toBalance.mul(
                        toCooldownTimestamp))
160             ).div(amountToReceive.add(toBalance));
161         }
162 
163         return toCooldownTimestamp;
164     }
```

Listing 3.16: LONStaking:_getNextCooldownTimestamp()

If a staking user has not passed the cooldown timestamp, the staked funds will be locked inside the staking contract. It comes to out attention that this above `_getNextCooldownTimestamp()` routine does not properly update the cooldown timestamps of the receiving account. As a result, a malicious actor may simply transfer the pool token to a new account, which will have 0 `stakersCooldowns` and thus bypass the cooldown restriction.

Recommendation Revise the `_getNextCooldownTimestamp()` to properly handle the cooldown timestamp update.

Status The issue has been fixed in this commit: `0fc720c`.

3.11 Possible Costly LPs From Improper Vault Initialization

- ID: PVE-011
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `LONStaking`
- Category: Time and State [11]
- CWE subcategory: CWE-362 [5]

Description

As mentioned in Section 3.10, the `LONStaking` contract aims to provide incentives so that users can stake and lock their funds in a stake pool. The staking users will get their pro-rata share based on their staked amount. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the `xLon` pool token extremely expensive and bring hurdles (or even causes loss) for later stake providers.

To elaborate, we show below the `_stake()` routine. This routine is used for stake providers to deposit supported assets and get respective pool tokens in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
211     function _stake(address _account, uint256 _amount) internal {
212         require(_amount > 0, "cannot stake 0 amount");
213
214         // Mint xLon according to current share and Lon amount
215         uint256 totalLon = lonToken.balanceOf(address(this));
216         uint256 totalShares = totalSupply();
217         uint256 share;
218         if (totalShares == 0  totalLon == 0) {
219             share = _amount;
220         } else {
221             share = _amount.mul(totalShares).div(totalLon);
222         }
223         // Update staker's Cooldown timestamp
224         stakersCooldowns[_account] = _getNextCooldownTimestamp(
225             block.timestamp,
226             share,
227             _account,
228             balanceOf(_account)
229         );
230
231         _mint(_account, share);
232         emit Staked(_account, _amount, share);
233     }
```

Listing 3.17: LONStaking::_stake()

Specifically, when the pool is being initialized, the share value directly takes the value of `_amount` (line 219), which is under control by the malicious actor. As this is the first deposit, the current total

supply equals the calculated `share = _amount.mul(totalShares).div(totalLon)= 1WEI`. With that, the actor can further deposit a huge amount of `lonToken` asset with the goal of making the pool token extremely expensive.

An extremely expensive pool token can be very inconvenient to use as a small number of $1WEI$ may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial stake provider, but this cost is expected to be low and acceptable. Another alternative requires a guarded launch to ensure the pool is always initialized properly.

Recommendation Revise current execution logic of `_stake()` to defensively calculate the share amount when the pool is being initialized.

Status This issue has been confirmed and the team will exercise extra caution in having a guarded launch to ensure the pool will be properly initialized.

3.12 Potential Sandwich Attacks To Minimize swappedLonAmount

- ID: PVE-012
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `RewardDistributor`
- Category: Time and State [14]
- CWE subcategory: CWE-682 [8]

Description

As part of the incentive mechanisms, the `RewardDistributor` contract is designed to distribute the rewards to participating users. Within this contract, there is a core `_swap()` routine that converts the fee token to `LON`. To elaborate, we show below the full `_swap()` implementation.

```
366     function _swap(
367         address _feeTokenAddr,
368         address _exchangeAddr,
369         address[] memory _path,
370         uint256 _amountFeeTokenToSwap,
371         uint256 _minLonAmount
```

```
372     ) internal returns (uint256 swappedLonAmount) {
373         // Approve exchange contract
374         IERC20(_feeTokenAddr).safeApprove(_exchangeAddr, MAX_UINT);
375
376         // Swap fee token for Lon
377         IUniswapRouterV2 router = IUniswapRouterV2(_exchangeAddr);
378
379         uint256[] memory amounts = router.swapExactTokensForTokens(
380             _amountFeeTokenToSwap,
381             _minLonAmount,  // Minimum amount of Lon expected to receive
382             _path,
383             address(this),
384             block.timestamp + 60
385         );
386         swappedLonAmount = amounts[_path.length - 1];
387
388         // Clear allowance for exchange contract
389         IERC20(_feeTokenAddr).safeApprove(_exchangeAddr, 0);
390     }
```

Listing 3.18: RewardDistributor :: _swap()

We notice the conversion is routed to `UniswapV2` in order to swap them to the intended `LON`. And the swap operation does restrict possible slippage with `_minLonAmount`. However, the given slippage control is given from the user argument and is therefore still vulnerable to possible front-running attacks, resulting in a smaller gain for this round of swap.

In the meantime, it is important to notice that the `buyback()` routine has been properly mitigated by the associated modifier `only_EOA_or_Self`. However, the `buyback()` routine can be called from the `batchBuyback()`, which can still be successfully invoked without being restricted to be an EOA account.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user or the buyback operation in our case because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

Status The issue has been fixed in this commit: `bc08821`.

3.13 Lack of lastTimeBuyback Initialization in RewardDistributor

- ID: PVE-013
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `RewardDistributor`
- Category: Coding Practices [12]
- CWE subcategory: CWE-628 [7]

Description

As mentioned in Section 3.12, the `RewardDistributor` contract is designed to distribute the rewards to participating users. This contract has been properly initialized with a number of required parameters. However, there is an important one that has not been properly initialized: i.e., `lastTimeBuyback`. Note this parameter is designed to record the last timestamp of buyback. If this parameter is not initialized, it will take the default value of 0, which is certainly not the intended buyback timestamp.

```
113     constructor(
114         address _owner,
115         address _operator,
116         uint32 _buyBackInterval,
117         uint8 _miningFactor,
118         address _treasury,
119         address _lonStaking,
120         address _miningTreasury,
121         address _feeTokenRecipient
122     ) Ownable(_owner) {
123         operator = _operator;
124 
125         buybackInterval = _buyBackInterval;
126 
127         require(_miningFactor <= 100, "incorrect mining factor");
128         miningFactor = _miningFactor;
129 
130         require(Address.isContract(_lonStaking), "Lon staking is not a contract");
131         treasury = _treasury;
132         lonStaking = _lonStaking;
133         miningTreasury = _miningTreasury;
134         feeTokenRecipient = _feeTokenRecipient;
135 
136         // Hardcode Lon fee token data
137         FeeToken storage feeToken = feeTokens[LON_TOKEN_ADDR];
138         feeToken.RFactor = 40;
139         feeToken.minBuy = 100;
140         feeToken.enable = true;
141     }
```

Listing 3.19: RewardDistributor :: constructor()

Recommendation Properly initialize the `lastTimeBuyback` parameter in the `RewardDistributor::constructor()` function.

Status The issue has been fixed in this commit: `e654a09`.

3.14 Other Suggestions

Due to the fact that compiler upgrades might bring unexpected compatibility or inter-version consistencies, 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-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

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

[7] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

[8] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

[11] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

[14] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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