SMART CONTRACTS REVIEW

zokyo

June 12th 2024 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that this smart contract passed a security audit.

ZOKYO AUDIT SCORING LEGION OTC

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

HYPOTHETICAL SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:
- 0 Critical issues: 0 points deducted
- 2 High issues: 2 resolved = 0 points deducted
- 3 Medium issues: 2 resolved and 1 acknowledged = - 3 points deducted
- 5 Low issues: 5 resolved = 0 points deducted
- 6 Informational issues: 6 resolved = 0 points deducted

Thus, 100 - 3 = 97

TECHNICAL SUMMARY

This document outlines the overall security of the Legion OTC smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Legion OTC smart contract/s codebase for quality, security, and correctness.

Contract Status

There were 0 critical issues found during the review. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Legion OTC team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Legion OTC repository:
Repo: https://github.com/Graffino/LegionVentures-Contracts/tree/feature/escrow

Last commit -00bf5b31ce1b7930297f1ffe974c10406647c44d

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- PreMarketEscrow.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Legion OTC smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01 Due diligence in assessing the overall code quality of the codebase.

02 Cross-comparison with other, similar smart contract/s by industry leaders.

03 Thorough manual review of the codebase line by line.

Executive Summary

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Legion OTC team and the Legion OTC team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

- **Critical**

  The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

- **High**

  The issue affects the ability of the contract to compile or operate in a significant way.

- **Medium**

  The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

- **Low**

  The issue has minimal impact on the contract’s ability to operate.

- **Informational**

  The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Arbitrary walletAddress leads to potentially locked assets	High	Resolved
2	Method revertEscrow() doesn't work as expected	High	Resolved
3	Possible loss of funds due to destroyEscrow() method	Medium	Resolved
4	Method confirmFailureToTransfer() doesn't check for seller funds	Medium	Resolved
5	Centralization risk	Medium	Acknowledged
6	Missing events	Low	Resolved
7	CEI pattern not followed	Low	Resolved
8	Escrow amount not validated for > 0	Low	Resolved
9	Approved tokens not validated for address(0)	Low	Resolved
10	Missing disable initializer	Low	Resolved
11	Unnecessary Safe Math is being utilized	Informational	Resolved
12	Gas Consumption Concerns Due to Extended Error Messages	Informational	Resolved
13	Redundant check	Informational	Resolved
14	Floating Pragma Version in Solidity Files	Informational	Resolved
15	Use ENUM for seller/buyer roles	Informational	Resolved
16	Remove unused imports	Informational	Resolved

Arbitrary walletAddress leads to potentially locked assets

Function `createEscrow` initiates an asset vault that buyer and seller deposit assets to. A risk arises due to the fact that `walletAddress` is being arbitrarily decided as an argument to the function by the caller. An incompatible `walletAddress` can lead to several risks:

- One of the parties buyer/seller have control on that address and drain the funds before the settlement is reached.
- The `walletAddress` refers to a Smart Contract that is not considered a compatible ERC20 holder, hence it is not capable of approving PreMarketEscrow to transfer assets leading to assets getting locked in the Smart Contract.

Recommendation:

Several measures can be taken to mitigate this issue:

- Create a smart contract to resemble a vault for that `walletAddress` that is controllable by the trusted entities (i.e. `PreMarketEscrow`).
- Receive the funds into the `PreMarketEscrow` itself.

Method revertEscrow() doesn't work as expected

Method revertEscrow() has the following logic:

```
function revertEscrow(string calldata refId) external onlyOwner {
    Escrow storage escrow = escrows[refId];

    require(escrow.seller != address(0) || escrow.buyer != address(0),
"Escrow does not exist.");

    bool isSeller = msg.sender == escrow.seller;
    bool isBuyer = msg.sender == escrow.buyer;

    require(isSeller || isBuyer, "Only the escrow creator can cancel before
funding.");
_ }
```

Here. this method is supposed to be called only by owner but later on it checked that msg.sender is either seller or buyer which is contradicting as msg.sender can not owner and seller/buyer.

This leads to that issue where seller/buyers wont be able to revert their escrow if it was not accepted.

Recommendation:
Allow only owner to revert the escrow in case sellers/buyers dont find a matching buyer/seller.

MEDIUM-1 | RESOLVED

Possible loss of funds due to destroyEscrow() method

Method destroyEscrow() deletes the escrow details without considering the tokens passed to the wallet address if a seller or buyer has deposited tokens.
If owner calls the destroyEscrow() method for such escrows then there is possibility of tokens in wallet address to be stuck.

Recommendation:
Consider checking if seller and/or buyer funds are locked before deleting an escrow and proceed accordingly.

MEDIUM-2 | RESOLVED

Method confirmFailureToTransfer() doesn't check for seller funds

Method confirmFailureToTransfer() doesn't check if the seller has locked funds or not but only checks for buyer funds.
There could be a case where seller funds are not locked ans still the buyer gets 2*amount when owner calls this method.
This would be loss of funds for the protocol if enough tokens are present in the wallet address to transfer to the buyer.

Recommendation:
Ensure that both seller and buyer funds are locked before transferring 2X funds to the buyer.

MEDIUM-3 | ACKNOWLEDGED

Centralization risk

The smart contracts employ modifiers (e.g. onlyOwner) in functions responsible for carrying out actions which can conflict with caller's interest (e.g. setFeeReceiver, destroyEscrow, emergency withdraw funds...etc). This approach centralizes control, allowing a single owner to exclusively perform critical actions, that involves asset transfer, posing a risk to decentralization principles.

Risk: Single Point of Failure: Centralized control introduces a single point of failure, where the compromise of the owner's account can lead to unauthorized access and manipulation of critical functions.

Recommendation:
To mitigate the centralization risk, it is recommended to:

Implement Access Control Lists (ACL): Utilize Access Control Lists to assign different roles with specific permissions, allowing for a more granular control structure.

Multi-Sig or Governance Contracts: Consider implementing multi-signature schemes or governance contracts to distribute decision-making authority, reducing the risk of a single point of failure. Multi-Sig can be utilized in the project after deployment without altering the codebase.

LOW-1 | RESOLVED

Missing events for important state updates

Method emergecyWithdrawForEscrow(...) and method emergecyWithdrawForProject(...) allows admin to witthdraw user funds in case contract is under attack or malicious activity is possible. These methods are not emitting events which can help admins later on to distribute emergency withdrawn funds to users since escrow details in contract is deleted.

Recommendation:
Consider adding events for above mentioned methods as well.

CEI pattern is not followed

Most of the methods of the contract follow this pattern: (Check, Interaction, Effects)

```
function methodName(string calldata refId) external onlyOwner {
// Checks
   Escrow storage escrow = escrows[refId];
   require(escrow.sellerFundsLocked && escrow.buyerFundsLocked, "Funds not
locked");

   uint256 feeAmount = (escrow.amount * feePercentage) / 10000;
// Interaction
   IERC20 token = IERC20(escrow.tokenAddress);

   _transferFromEscrow(token, escrow.walletAddress, escrow.seller,
escrow.amount);
   token.safeTransferFrom(feeReceiver, escrow.seller, feeAmount);

   _transferFromEscrow(token, escrow.walletAddress, escrow.buyer,
escrow.amount);
   token.safeTransferFrom(feeReceiver, escrow.buyer, feeAmount);
// Effects
   _removeEscrow(refId);

   emit EscrowCancelled(refId);
 }
```

Here, it is advised to update the state before making external calls such as transferring ERC20 tokens to avoid reentrancy attack.

In case, ERC777 tokens are used here as one of the whitelisted tokens, the reentracncy attack is possible if owner is malicious or owner private is compromised. Although the likelihood is low, it is advised to follow CEI pattern.

Recommendation:
Consider calling _removeEscrow before token transfers.

LOW-3 | RESOLVED

Escrow amount not validated for `0`

Method createEscrow(...) allows a seller/buyer to create an escrow with amount. But this amount can be `0` as it is not checked if amount > `0` or not.

Recommendation:
Add a validation to check if escrow amount > `0` or not.

LOW-4 | RESOLVED

Approved tokens not validated for address(`0`)

Method approveTokens() approves tokens in which sellers/buyers can deposit the tokens but it is not validated for address(`0`).

If accidentally address(`0`) is approved, createEscrow() can be called with tokenAddress as address(`0`) and this can lead to unexpected results.

Recommendation:
Add a validation to check approvedTokens() are not address(`0`).

LOW-5 | RESOLVED

Missing disable initializer

Contract PreMarketEscrow.sol implements the initialize(...) method with the initializer modifier without disabling the initializers for the implementation contract as recommended by OpenZeppelin here.

Recommendation:
Disable the initializers for the implementation method as suggested by OpenZeppelin here.

Unnecessary Safe Math is being utilized

The default safe math operation in solidity versions ^0.8.x incurs extra gas overhead due to it requiring more computation. The following operation, that is being carried out on the iterator of the for-loop, can be more gas-efficient by using the `unchecked` statement.

In function approveTokens(), we have:

```
 function approveTokens(address[] calldata tokenAddresses) external
onlyOwner {
    for (uint256 i = 0; i < tokenAddresses.length; i++) {
      approvedTokens[tokenAddresses[i]] = true;
      emit TokenApproved(tokenAddresses[i]);
    }
  }

As well in function revokeTokens().
 function revokeTokens(address[] calldata tokenAddresses) external
onlyOwner {
    for (uint256 i = 0; i < tokenAddresses.length; i++) {
      approvedTokens[tokenAddresses[i]] = false;
      emit TokenRevoked(tokenAddresses[i]);
    }
  }
```

While the code snippet correctly ensures that the addition operation will not result in an overflow, the unnecessary default safe math usage can be optimized for gas efficiency. Wrapping the operation in an unchecked statement is a recommended practice for situations where the developer can guarantee that overflows/underflows will not occur. This enhancement contributes to more efficient gas utilization without compromising safety.

Recommendation:

Wrap Operation in `unchecked` Statement, given that the condition preceding the operation ensures there is no risk of overflow. It is a common pattern in for-loops since 0.8.x to be as follow:

```
for (uint256 i = 0; i < length;) {
    ...
    unchecked {
        i++;
    }
}
```

INFORMATIONAL-2 | RESOLVED

Gas Consumption Concerns Due to Extended Error Messages

The smart contract employs detailed error messages within require statements, offering developers and users specific insights into the conditions that trigger a requirement failure. However, the extensive length of these error messages result in heightened gas consumption.

Recommendation:

- Employ shorter concise error messages.
- Use Custom Error objects available from solidity 0.8.0.

Redundant check

In line 69, `require(approvedTokens[tokenAddress], "Token not approved")` is a redundant and unnecessary check. Modifier `onlyApprovedToken(tokenAddress)` already carries out that check.

```
60  function createEscrow(
61  address tokenAddress,
62  uint256 amount,
63  string calldata refId,
64  address walletAddress,
65  string calldata creatorRole
66  ) external onlyApprovedToken(tokenAddress) {
67    require(escrows[refId].seller == address(0) && escrows[refId].buyer
== address(0), "Escrow already exists!");
68    require(walletAddress != address(0), "Invalid wallet address");
69    require(approvedTokens[tokenAddress], "Token not approved");
```

Recommendation:
No need to carry out the require statement checking `approvedTokens[tokenAddress]`.

INFORMATIONAL-4 | RESOLVED

Floating Pragma Version in Solidity Files

The Solidity files in this codebase contain pragma statements specifying the compiler version in a floating manner (e.g., ^0.8.0). Floating pragmas can introduce unpredictability in contract behavior, as they allow automatic adoption of newer compiler versions that may introduce breaking changes.

Recommendation:
To ensure stability and predictability in contract behavior, it is recommended to:

Specify Fixed Compiler Version: Instead of using a floating pragma, specify a fixed compiler version to ensure consistency across different deployments and prevent automatic adoption of potentially incompatible compiler versions.

INFORMATIONAL-5 | RESOLVED

Use ENUM for seller/buyer roles

Method createEscrow() has a parameter `createrRole` to check the role of the msg.sender. String uses a lot more gas in this case.

Recommendation:
Use ENUM for the roles.

INFORMATIONAL-6 | RESOLVED

Remove unused imports

Following import is not used. Consider to remove it.

```
import "hardhat/console.sol";
```

Recommendation:
Consider removing unused imports.

	PreMarketEscrow.sol
Reentrance	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

We are grateful for the opportunity to work with the Legion OTC team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Legion OTC team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo