SEDL Security Review

Prepared By Pelz

January 2025

Introduction

A time-boxed security review of the **SEDL** protocol was done by Pelz, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About SEDL

The SEDL protocol is an MLM (Multi-Level Marketing) utility token system deployed on the Polygon network. It implements several functionalities for tier-based reward distribution, user activation, and account management, ensuring secure, scalable, and upgradeable operations.The SEDL contract is an MLM (Multi-Level Marketing) utility token system deployed on the Polygon network. It implements several functionalities for tier-based reward distribution, user activation, and account management, ensuring secure, scalable, and upgradeable operations.

Severity Classification

Severity	Impact:High	Impact:Medium	Impact:Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- High - Leads to a significant loss of assets in the protocol or significantly harm a group of users
- Medium - Only a small amount of funds is lost or core contract functionality is broken or affected
- Low - Can lead to any kind of unexpected behaviour with no major impact

Likelihood

- High - Attack path is possible with reasonable assumptions that mimic on chain conditions and the cost of the attack is relatively low compared to the value lost or stolen
- Medium - Only a conditionally incentivised attack vector but still likely
- Low - Has too many or too unlikely assumptions

Actions Required For Severity Levels

- High - Must fix (before deployment, if not already deployed)

- Medium - Should fix
- Low - Could fix

Security Assessment Summary

review commit hash:

- e1f2ab91b399f3caf5a7f1d1a93674489585dace

The following number of issues were found, categorised by their severity:

Critical & High: 2 issues

Medium: 2 issues

Low & Info: 5 issues

Findings Summary

ID	Title	Severity	Status
[H-01]	Lack of User Verification in `activateAccount` Allows Multiple Activations	High	Resolved
[H-02]	Users Can Bypass Minimum Balance Threshold In `upgradeTier` Via Flashloan Attacks	High	Acknowledged
[M-01]	One-Way Logic in `updateMinimumTierBalance` Prevents Reducing Minimum Tier Balance	Medium	Resolved
[M-02]	Precision Loss in Reward Calculations	Medium	Resolved
[L-01]	Unused Pausability Feature Reduces Credibility of the Token	Low	Resolved
[L-02]	Unused `totalRewardsDistributed` Variable in `SEDL.sol`	Low	Resolved
[I-01]	Unused Constants in `SEDL.sol`	Informational	Resolved
[I-02]	Critical Functions Could Emit Events	Informational	Resolved

[I-03]	Use of Floating Pragma	Informational	Resolved

[H-01] Lack of User Verification in `activateAccount` Allows Multiple Activations

Severity

Impact: Medium

Likelihood: High

Description

The `activateAccount` function in `SEDL.sol` allows users to activate accounts using a provided `_userID`. However, the implementation lacks a robust verification mechanism to ensure that the `_userID` has not already been activated by another user. Specifically, the `user.userAddress` field is not checked to verify if it has been previously set. This oversight allows a malicious or careless user to activate the same `_userID` multiple times, which can disrupt the system's logic and lead to inconsistencies.

Code Segment

```
// @audit-issue the same userId can be activated multiple ti
mes cause the user info isn't verified.
function activateAccount(
    bytes32 _userID
) external whenNotPaused validBytesLength(_userID) {
    require(member[_userID], "Not a member");
    require(!users[msg.sender], "Already registered");

    MLM_USER storage user = userInfo[_userID];
    user.tier = MLM_LEVEL.Tier1;
    user.userAddress = msg.sender;
    users[msg.sender] = true;

    emit AccountActivated(msg.sender, _userID, block.timesta
```

```
mp);
}
```

Recommendation

Introduce a condition to ensure that the `user.userAddress` field has not already been set:

```
require(user.userAddress == address(0), "Account already activated");
```

[H-02] Users Can Bypass Minimum Balance Threshold In `upgradeTier` Via Flashloan Attacks

Severity

Impact: Medium

Likelihood: High

Description

The `upgradeTier` function in `SEDL.sol` allows users to upgrade their membership tier if they meet the required token balance for the next tier. However, the function only checks the user's current token balance at the time of the upgrade. This check is susceptible to flashloan exploits, where users can temporarily acquire the necessary token balance to pass the requirement and upgrade their tier without genuinely holding the tokens. This undermines the intended design of the tier system and allows users to bypass the minimum balance constraints.

Code Segment

```
function upgradeTier(
    bytes32 _userID
) external whenNotPaused nonReentrant onlyUsers validBytesLe
ngth(_userID) {
    MLM_USER storage user = userInfo[_userID];
    require(user.userAddress == msg.sender, "Unauthorized");
    require(user.tier != MLM_LEVEL.Tier4, "Max tier reache
d");
```

```
    MLM_LEVEL nextTier = MLM_LEVEL(uint256(user.tier) + 1);
    uint256 requiredBalance = minimumTierBalance[uint256(nex
tTier)];
    require(
        sedlToken.balanceOf(msg.sender) >= requiredBalance,
// @audit-issue users can upgrade tier easily by flashloanin
g tokens
        "Insufficient balance"
    );

    user.tier = nextTier;

    emit TierUpgraded(msg.sender, nextTier, block.timestam
p);
}
```

Recommendation

1. Implement a Staking Mechanism

Require users to lock the required token balance for a specified period before allowing a tier upgrade. This ensures that users genuinely hold the tokens.

Example of Staking Mechanism

```
mapping(bytes32 => uint256) public tierUpgradeTimestamps;

function upgradeTier(
    bytes32 _userID
) external whenNotPaused nonReentrant onlyUsers validBytesLe
ngth(_userID) {
    MLM_USER storage user = userInfo[_userID];
    require(user.userAddress == msg.sender, "Unauthorized");
    require(user.tier != MLM_LEVEL.Tier4, "Max tier reache
d");

    MLM_LEVEL nextTier = MLM_LEVEL(uint256(user.tier) + 1);
    uint256 requiredBalance = minimumTierBalance[uint256(nex
tTier)];
```

```
    require(
        sedlToken.balanceOf(msg.sender) >= requiredBalance,
        "Insufficient balance"
    );

    // Lock tokens for staking
    require(
        block.timestamp >= tierUpgradeTimestamps[_userID] +
7 days,
        "Staking period not met"
    );

    tierUpgradeTimestamps[_userID] = block.timestamp; // Upd
ate lock timestamp
    user.tier = nextTier;

    emit TierUpgraded(msg.sender, nextTier, block.timestam
p);
}
```

2. Use Token Snapshot Mechanisms

Leverage a snapshot mechanism to validate that users consistently hold the required balance over a specific period.

[M-01] One-Way Logic in `updateMinimumTierBalance` Prevents Reducing Minimum Tier Balance

Severity

Impact: Medium

Likelihood: Medium

Description

The `updateMinimumTierBalance` function in `SEDL.sol` is intended to allow the contract owner to update the minimum balance required for each tier. However, the implementation imposes a restriction that prevents reducing the minimum balance for a given tier. Specifically, the second `require` statement enforces that the new balance for a tier must be greater than the balance of the previous tier:

```
require(
    tierIndex==0|| newBalance> minimumTierBalance[tierIndex-
1],
"Invalid balance sequence"
);
```

This logic ensures that the tier sequence remains strictly increasing, but it also means that once a tier's minimum balance is set, it can only be increased, never decreased.

Recommendation

Modify the `require` statement to allow the tier's minimum balance to be reduced while still ensuring a valid balance sequence.

[M-02] Precision Loss in Reward Calculations

Severity

Impact: Medium

Likelihood: Medium

Description

The `distributeRewards` function in `SEDL.sol` calculates reward amounts for multiple tiers using percentage-based arithmetic. The calculations are done in **wei**, which has 18 decimals. However, this approach could result in precision loss, especially when dealing with small amounts, as integer division in Solidity truncates any remainder. This precision loss may lead to inaccurate reward distribution among users.

Code Segment

```
uint256 directReferrerReward = (amount *
    tierRewardPercentages[uint256(MLM_LEVEL.Tier1)]) / 100;

uint256 secondUplineReferrerReward = (amount *
    tierRewardPercentages[uint256(MLM_LEVEL.Tier2)]) / 100;

uint256 thirdUplineReferrerReward = (amount *
    tierRewardPercentages[uint256(MLM_LEVEL.Tier3)]) / 100;

uint256 fourthUplineReferrerReward = (amount *
    tierRewardPercentages[uint256(MLM_LEVEL.Tier4)]) / 100;
```

Recommendations

Adopt a fixed-point math library, such as OpenZeppelin's `FixedPointMath` or the `wmul` function from **FixedPointMathLib**, to perform calculations with higher precision. These libraries allow operations with a consistent 18-decimal precision (WAD), minimizing precision loss.

[L-01] Unused Pausability Feature Reduces Credibility of the Token

Severity

Impact: Low

Likelihood: Medium

Description

The `SEDLTOKEN.sol` contract inherits the `Pausable` functionality but does not implement or utilize it meaningfully in its critical operations. The contract includes `pause` and `unpause` functions that allow the owner to toggle the paused state. However, there is no mechanism enforcing or integrating the paused state in any of the token's core functions, such as `transfer`, `transferFrom`, or `approve`.

Additionally, the mere presence of such owner-controlled pausing features without proper implementation raises suspicion among decentralized exchanges

(DEXes) and the broader community. Tokens with unexplained pausing capabilities are often flagged as potential scams, significantly reducing trust and credibility.

Recommendation

1. **Utilize Pausability Effectively:**

   If the intent is to have a pausability feature, integrate it into the core token operations. For example, include the `whenNotPaused` modifier in critical functions like `transfer` and `transferFrom`. This ensures that the feature serves a meaningful purpose.

2. **Consider Removing Pausability:**

   If the pausing capability is not required for the token's use case, it is recommended to remove the inheritance of `Pausable` and the associated functions (`pause` and `unpause`) to prevent suspicion and increase trust.

[L-02] Unused `totalRewardsDistributed` Variable in `SEDL.sol`

Severity

Impact: Low

Likelihood: Medium

Description

The `SEDL.sol` contract declares the following variable:

```
uint256 public totalRewardsDistributed; // @audit-issue neve
r updated
```

This variable is intended to track the total rewards distributed to users. However, it is never updated within the contract, rendering it ineffective for its intended purpose.

Recommendation

Update the `totalRewardsDistributed` variable in functions that distribute rewards to users or remove the Variable if Unnecessary

[I-01] Unused Constants in `SEDL.sol`

Severity

Impact: Informational

Likelihood: Informational

Description

The `SEDL.sol` contract includes two constant variables:

```
uint256 public constant MAX_TIER_COUNT = 5;
uint256 public constant MAX_BYTES_LENGTH = 32;
```

These constants appear to be intended for enforcing limits, such as maximum tiers or maximum byte lengths in certain operations. However, they are never used in the contract's logic or any of its functions.

Recommendation

1. **Implement the Constants if Needed:**
    - If these constants are intended to enforce constraints, integrate them into relevant functions or validations. For example:
        - Use `MAX_TIER_COUNT` in logic where the number of tiers is set or checked.
        - Use `MAX_BYTES_LENGTH` to validate byte array lengths in applicable functions. Example:

```
require(tierCount <= MAX_TIER_COUNT, "Exceeded max tier c
ount");
require(bytes(data).length <= MAX_BYTES_LENGTH, "Exceeded
max bytes length");
```

2. **Remove the Constants if Not Needed:**
    - If there is no valid use case for these constants, remove them from the contract to maintain a clean and minimal codebase.

[I-02] Critical Functions Could Emit Events

Severity

Impact: Informational

Likelihood: Informational

Description

The protocol lacks event emissions for critical functions that update the state. Functions such as `updateMinimumTierBalance` and `updateTierRewardPercentage` modify key variables but do not emit events to notify off-chain systems.

Recommendation

Add events to each state-changing function. These events should include the old and new values for enhanced clarity.

[I-03] Use of Floating Pragma

Severity

Impact: Informational

Likelihood: Informational

Description

The protocol currently uses a floating Solidity pragma (`^0.8.20`), which allows the compiler to use any version of Solidity from `0.8.20` up to but not including `0.9.0`. While this can offer flexibility during development, it can introduce risks in production environments.

Recommendation

Update the Solidity pragma to a fixed version that you have thoroughly tested.