Lumia
HyperLaunch IDO Staking

Security
Audit
Report

July 23, 2025

1. INTRODUCTION

1.1 Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only. The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of the Client. If you are not the intended recipient(s) of this document, please note that any disclosure, copying or dissemination of its content is strictly forbidden.

1.2 Security Assessment Methodology

BugBlow utilized a widely adopted approach to performing a security audit. Below is a breakout of how our team was able to identify and exploit vulnerabilities.

1.2.1 Code and architecture review:

- Review the source code.
- Read the project's documentation.
- Review the project’s architecture.

Stage goals

- Build a good understanding of how the application works.

1.2.2 Check code against vulnerability checklist:

- Understand the project’s critical assets, resources, and security requirements.
- Manual check for vulnerabilities based on the auditor's checklist.
- Run static analyzers.

Stage goals

- Identify and eliminate typical vulnerabilities (gas limit, replay, flash-loans attack, etc.)

1.2.3 Threat & Attack Simulation:

- Analyze the project's critical assets, resources and security requirements.
- Exploit the found weaknesses in a safe local environment.
- Document the performed work.

Stage goals

- Identify vulnerabilities that are not listed in static analyzers that would likely be exploited by hackers.
- Develop Proof of Concept (PoC).

1.2.4 Report found vulnerabilities:

- Discuss the found issues with developers
- Show remediations.
- Write an initial audit report.

Stage goals

- Verify that all found issues are relevant.

1.2.5 Fix bugs & re-audit code:

- Help developers fix bugs.
- Re-audit the updated code again.

Stage goals

- Double-check that the found vulnerabilities or bugs were fixed and were fixed correctly.

1.2.4. Deliver final report:

- The Client deploys the re-audited version of the code

- The final report is issued for the Client.

Stage goals

- Provide the Client with the final report that serves as security proof.

Severity classification

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
Critical	Vulnerabilities leading to assets theft, fund access locking, or any other loss of funds.
High	Vulnerabilities that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Vulnerabilities that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss funds.
Low	Vulnerabilities that do not have a significant immediate impact and could be easily fixed.

1.3 Project Overview

The HyperLaunchStaking contract is tier-based ERC-20 staking system. It powers a structured, incentive-driven staking mechanism that rewards long-term commitment and larger stakes. Users are categorized into progressive tiers based on the amount staked, unlocking greater allocation weights for IDOs and other exclusive benefits. The contract supports flexible lock durations (90, 180, or 360 days), with higher multipliers and reduced fees for longer commitments—ensuring strong alignment between user incentives and protocol growth. With built-in fee mechanics, tier upgrades, and administrative controls, the system is designed to maximize user engagement, reward loyalty, and reinforce long-term token stability.

Meanwhile, the Whitelist contract is a robust access control system that enables efficient user management for IDO participation. It allows project teams to register individual IDO events and manage eligible participant lists through both single and batch operations. Built-in role-based access ensures secure delegation, while configurable batch limits offer scalability without compromising performance.

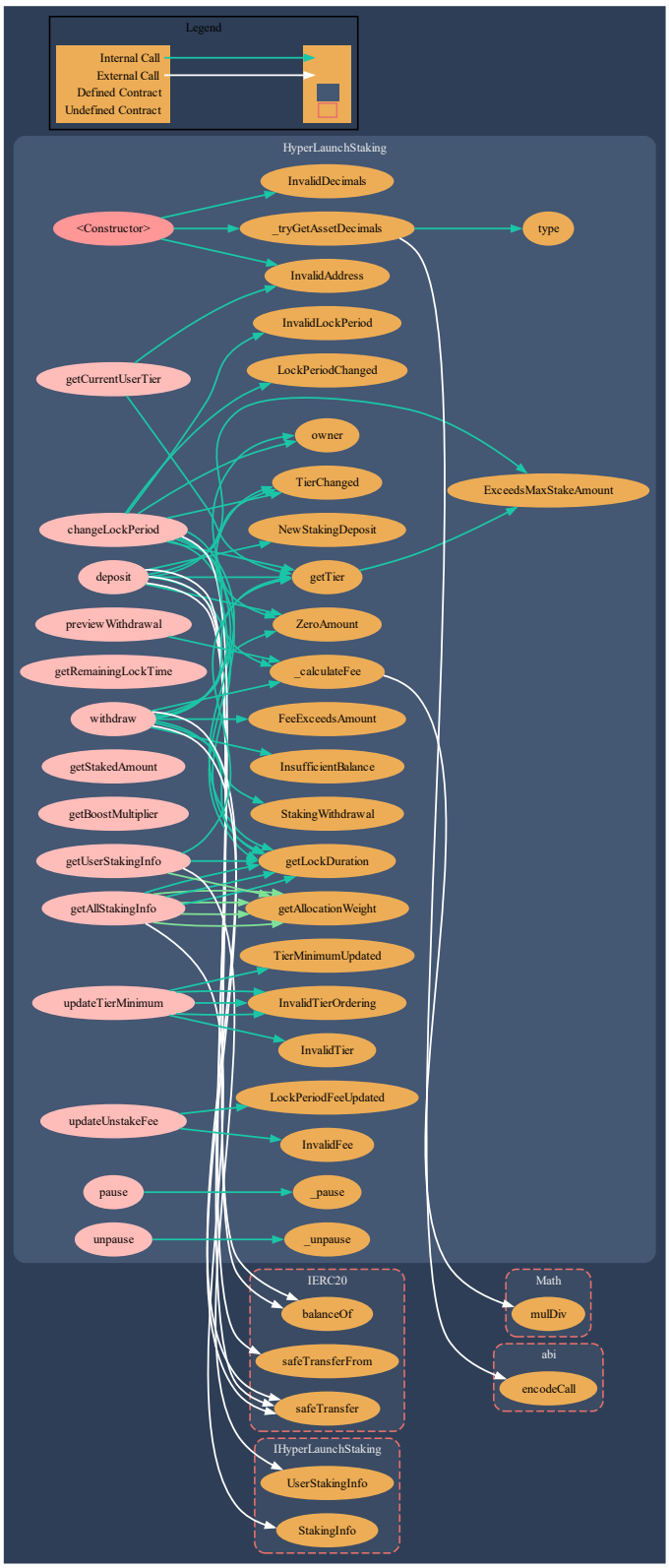

Figure 1. IDO Staking graph flow.

1.4 Project Dashboard

Project Summary

Title	Description
Client name	Lumia
Project name	IDO Staking
Timeline	07.09.2025 - 07.16.2025
Number of Auditors	3

Project Last Log

Date	Commit Hash	Note
24.06.2025	5dfaa8160b797f2a8ce6f5a429e4fb34a3c2d82d	Change Whitelist and Staking contracts

Project Scope

The audit covered the following files:

File name
ido-contracts/src/ido/Whitelist.sol ido-contracts/src/staking/HyperLaunchStaking.sol ido-contracts/src/interfaces/staking/IHyperLaunchStaking.sol

1.5 Summary of findings

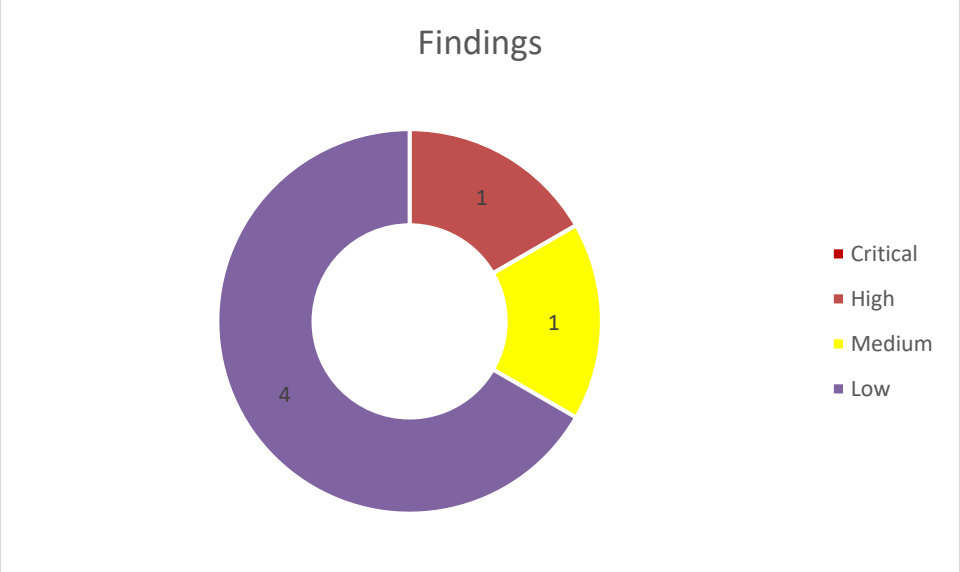

Figure 2. Findings chart

Severity	# of Findings
CRITICAL	0
HIGH	1
MEDIUM	1
LOW	4

2. FINDINGS REPORT

2.1 Critical

Not Found

2.2 High

2.2.1 Uncollected Early Withdrawal Fees Due to Missing Lock Time Initialization

Status: Fixed

Description

When a user stakes transfer-on-fee tokens for the first time, the token may reduce the amount received due to a transfer fee (e.g., 1% fee), and since their tier may stay the same, the contract will fail to set the lockEndTime and currentLockPeriod. This will lead to the withdrawal fee to be set to 0, consequently losing funds.

Why?

The following line doesn't become true.

```
if (tierChanged || lockPeriodChanged || user.totalAmount == amount)
```

1. The tier wouldn't be changed (since the default is Tier 1)
2. Lock period wouldn't be changed
3. The user.totalAmount would be slightly less than the sent amount (because of transfer fees).

Let's consider the following example.

- TokenA is a fee-on-transfer token, it takes 1% of transferred amount as a fee. Alice is a staker

- Alice deposits 20,000 TokenA with lock duration as 90 days by calling the deposit() function.

- As TokenA is a fee-on-transfer token, the actual received amount is 20000 × (1-0.1) = 19800.

- As the staked amount is 19800, the tier of the user becomes tier1.

```
1. enum Tier {
2.   TIER1, // 10,001-100,000 HYPE (allocation weight 1)
3.   TIER2, // 1,001-10,000 HYPE (allocation weight 2)
4.   TIER3, // 1-1,000 HYPE (allocation weight 3)
5.   NONE // 0 HYPE (no allocation)
6. }
```

- tierChanged variable becomes False (As default tier of a user is Tier1 rather than None)

```
1. Tier newTier = getTier(newTotalAmount);
2. Tier oldTier = user.currentTier;
3. bool tierChanged = newTier != oldTier;
```

- lockPeriodChanged variable becomes False (this is the first deposit, user.totalAmount is 0)

```
1. bool lockPeriodChanged = (user.totalAmount > 0) && (lockPeriod != user.currentLockPeriod);
```

- as a result, user.lockEndTime can't be set and remains 0, because the branch wouldn't be executed.

```
1. if (tierChanged || lockPeriodChanged || user.totalAmount == amount) {
2.    user.currentLockPeriod = lockPeriod;
3.    user.lockEndTime = block.timestamp + getLockDuration(lockPeriod);
4.    ...
```

In the withdraw() function, as block.timestamp is bigger than user.lockEndTime which is 0, the fee variable becomes 0.

```
1. uint256 fee;
2. if (block.timestamp < user.lockEndTime) {
3.     fee = _calculateFee(amount, unstakeFeesByLockPeriod[user.currentLockPeriod]);
4. }
5.
```

Finally, the tier1 staker can withdraw their staked tokens without paying any fee.

Remediation

Change the default tier to None by moving it up. This way, the important branch will be executed.

```
1. enum Tier {
2.   NONE // 0 HYPE (no allocation)
3.   TIER1, // 10,001-100,000 HYPE (allocation weight 1)
4.   TIER2, // 1,001-10,000 HYPE (allocation weight 2)
5.   TIER3, // 1-1,000 HYPE (allocation weight 3)
6. }
```

2.3 Medium

2.3.1 Admin Can Set Equal Minimums for Different Tiers

Status: Fixed

Description

The updateTierMinimum() function doesn't strictly enforce a greater-than relationship between tiers. It uses non-strict checks (<, >) like:

```
1. if (newMinimum < tierMinimums[Tier.TIER2])
```

This allows setting equal minimum values for two or more tiers (e.g., TIER1 == TIER2).
As a result, a user staking an amount exactly equal to the threshold could be wrongfully (possibly on purpose) assigned a higher tier, breaking the intended tier hierarchy. It distorts rewards, allocations, and staking incentives.

Remediation

Use strict inequality when updating tier minimums:

```
1. if (tier == Tier.TIER1) {
2.     require(newMinimum > tierMinimums[Tier.TIER2], "InvalidTierOrdering");
3. } else if (tier == Tier.TIER2) {
4.     require(newMinimum > tierMinimums[Tier.TIER3] && newMinimum < tierMinimums[Tier.TIER1], "In-
validTierOrdering");
5. } else if (tier == Tier.TIER3) {
6.     require(newMinimum < tierMinimums[Tier.TIER2], "InvalidTierOrdering");
7. }
```

2.4 Low

2.4.1 Redundant Condition in getTier()

Status: Fixed

Description

This line has an unnecessary check:

if (amount >= tierMinimums[Tier.TIER1] && amount <= MAX_STAKE_AMOUNT)

Because just above, the contract already reverts if *amount* > *MAX_STAKE_AMOUNT*.

Remediation

Simplify to:.

```
if (amount >= tierMinimums[Tier.TIER1]) return Tier.TIER1;
```

2.4.2 Redundant lockEndTime Update in changeLockPeriod()

Status: Fixed

Description

This function sets lockEndTime twice with the same values:

```
1. user.currentLockPeriod = newLockPeriod;
2. user.lockEndTime = block.timestamp + getLockDuration(newLockPeriod); // <- already set
3.
4. if (newTier != user.currentTier) {
5.     ...
6.     user.lockEndTime = block.timestamp + getLockDuration(user.currentLockPeriod); // <- redundant
7. }
```

Remediation

Remove the second line inside the if block.

2.4.3 Missing Decimal Validation May Allow Misconfigured Tokens

Status: Fixed

Description

In the constructor of the HyperLaunchStaking contract, the code attempts to read the number of decimals from the staking token:

```
(bool success, uint8 assetDecimals) = _tryGetAssetDecimals(stakingToken_);
```

However, if this call fails, the returned value may be 0, which is invalid or highly suspicious for ERC20 tokens. If not explicitly checked, this could result in misconfigured staking behavior, incorrect tier thresholds, and reward miscalculations.

Remediation

Add a check to explicitly reject 0-decimal tokens:

```
1. (bool success, uint8 assetDecimals) = _tryGetAssetDecimals(stakingToken_);
2. if (assetDecimals == 0) revert InvalidDecimals(); // Add this line
```

2.4.4 Whole Batch Reverts on One Bad Address

Status: Fixed

Description

In batchAddToWhitelist() and batchRemoveFromWhitelist() of the Whitelist contract, one invalid address (e.g., zero address or already-whitelisted address) causes the entire batch to revert and lose funds (gas) spent on execution for the previous addresses. If the list is long enough, it may result in misconfigured staking behavior, incorrect tier thresholds, and reward miscalculations.

Remediation

Skip bad addresses instead of reverting the whole call:

```
1. if (user == address(0)) continue;
2. if (_whitelisted[idoId][user]) continue;
3. // ...
```

CONSCLUSION

This security audit identified one high-severity, one medium-severity, and five low-severity findings in the HyperLaunchStaking and Whitelist contracts.

High:

• Uncollected Early Withdrawal Fees: Fee-on-transfer tokens can bypass lock enforcement due to missing lockEndTime, allowing users to withdraw without penalty and causing protocol fund loss.

Medium:

• Equal Tier Minimums Allowed: The updateTierMinimum function permits overlapping thresholds between tiers, enabling unintended tier assignment and undermining the reward structure.

Low:

• Redundant Condition in getTier: Minor logic duplication affecting clarity.

• Redundant lockEndTime Update: Inefficient code in changeLockPeriod.

• Missing Decimal Validation: Tokens with zero decimals are accepted, risking misconfigured staking behavior.

• Batch Reverts on Invalid Address: A single bad entry in batch whitelist operations causes full transaction failure.

All identified issues have been addressed by the project team.
The applied fixes significantly strengthen the contract logic, improving its robustness, maintainability, and security posture. The protocol is now considered well-structured and secure based on the reviewed scope.