Mystic Finance - Security Review

06.11.2025

Conducted by:

Kann, Lead Security Researcher

Table of Contents

1 Disclaimer

Audits are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

2 Risk classification

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

2.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

2.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

3 Executive summary

Overview

Project Name	Mystic Finance
Repository	https://github.com/mystic-finance/Liquid-Staking/tree/staked-plume
Commit hash	2e6d7e59def340c63f98808b19c9650d8f63917b
Resolution	Fixed
Documentation	https://docs.frax.finance/frax-ether/overview
Methods	Manual review

Scope

/src/stPlumeMinter.sol /src/frxETH.sol

Issues Found

Critical risk	1
High risk	0
Medium risk	1
Low risk	0
Informational	0

4 Findings

4.1 Critical Risk

4.1.1 Miscalculation in rewards will give too much rewards to users

Severity: *Critical risk*

Description: In `syncRewards`, the rewards are calculated by substracting the last rewards to the total yield. This is inaccurate because total yield keep increasing, and last rewards are just the last amount rewarded.

This means that over time, rewards will become much more inflated than their actual values, giving away too many yield to the users. The first users to withdraw will effectively steal from others.

Proof of Concept:

Let's assume that we get 10 ether of rewards per cycle, for simplicity.

The key parts of function `syncRewards` work as follow:

Cycle 1:

`yieldEth` has increased to 10 ether. `lastRewardAmount` is 0 (first run).

```
uint256 nextRewards = yieldEth - lastRewardAmount uint256 nextRewards = 10 ether - 0 = 10 ether
```

`lastRewardAmount` is set to `nextRewards`, 10 ether.

Cycle 2:

`yieldEth` has increased to 20 ether. `lastRewardAmount` is 10.

```
uint256 nextRewards = 20 ether - 10 ether = 10 ether
```

`lastRewardAmount` is set to `nextRewards`, 10 ether.

Cycle 3, where things start to break:

`yieldEth` has increased to 30 ether. `lastRewardAmount` is 10.

```
uint256 nextRewards = 30 ether - 10 ether = 20 ether
```

Next rewards are calculated as twice the normal amount.

Resolution: Fixed

4.2 Medium Risk

4.2.1 DoS Vulnerability in withdraw() Due to Overcommitment of currentWithheldETH in Concurrent unstake() Calls

Severity: *Medium risk*

Description: The protocol maintains a global variable currentWithheldETH to represent the amount of ETH reserved for user withdrawals after they call unstake(). However, this variable is only decreased

when withdraw() is called — not when unstake() is made. This creates a critical race condition: multiple users can initiate unstake() relying on the same currentWithheldETH balance, leading to overcommitment. If one user completes the withdrawal first, the remaining users are blocked from withdrawing, resulting in a denial of service (DoS).

Example Scenario Initial State: currentWithheldETH = 10 ETH

Alice calls unstake(10 ETH) → succeeds (as 10 ETH is available)

Bob calls unstake(8 ETH) → also succeeds (still sees 10 ETH as available)

Both transactions pass the currentWithheldETH >= amount check, since the variable hasn't been decremented yet.

Alice calls withdraw() first:

Receives 10 ETH

currentWithheldETH becomes 0

Bob then calls withdraw():

Fails, as there is no remaining currentWithheldETH

Additionally, Bob cannot call unstake() again due to the presence of an active withdrawal request

If no one deposits or triggers a stake rebalance before Bob's cooldown period ends, Bob is temporary locked out of his funds until admin calls unstakeGov() or rebalance happens.

Resolution: Fixed