Mystic Finance - Security Review

06.08.2025

Conducted by:

Kann, Lead Security Researcher

Strapontin, Associate Security Researcher

TradMod, Associate 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	a7a86bdc3d65c5c2512fe6a835c10b4bc2d4315d
Resolution	Fixed
Documentation	https://docs.frax.finance/frax-ether/overview
Methods	Manual review

Scope

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

Issues Found

Critical risk	0
High risk	2
Medium risk	2
Low risk	0
Informational	4

4 Findings

4.1 High Risk

4.1.1 Incorrect Accounting of currentWithheldETH in Instant Withdrawals

Severity: *High risk*

Description: In the withdraw(address recipient) function, when the withdrawal amount is less than or equal to currentWithheldETH, the function enters the instant withdrawal path. In this case, the contract does not call plumeStaking.withdraw(), and the funds are expected to be fully covered by the ETH already held in currentWithheldETH. However, the code still sets withdrawn = amount and then unconditionally adds this value back to currentWithheldETH. Immediately after, it subtracts amount from currentWithheldETH, effectively leaving the value of currentWithheldETH unchanged.

```
} else {
            fee = amount * INSTANT_REDEMPTION_FEE / RATIO_PRECISION;
            withdrawn = amount;
        }
        uint256 cachedWithheldETH = currentWithheldETH;
        currentWithheldETH += withdrawn; //adding the withdraw amount
        withHoldEth += fee;
        currentWithheldETH -= amount;  //deducting the amount
```

This logic is flawed because the ETH used to fulfill the withdrawal is not newly received; it is already present in currentWithheldETH, and should simply be subtracted, not first added and then subtracted. The current logic makes it appear as if no ETH was withdrawn, which results in incorrect accounting of the withheld pool. If multiple such withdrawals occur, the system will consistently over-report the available withheld ETH, potentially leading to over-withdrawals or failed withdrawals in the future when actual funds are not available.

Resolution: Fixed

4.1.2 User Funds Burned With No ETH Returned Due to Missing withdrawalRequests[msg.sender] Assignment After Unstake

Severity: *High risk*

Description: The unstaking process is executed in two steps:

The user initiates unstake().

If currentWithheldETH is insufficient, the contract attempts to unstake from the Plume validator.

However, during the fallback logic that handles validator unstaking, the contract returns amountUnstaked without saving it in the withdrawalRequests[msg.sender] mapping. This causes a critical failure in the withdrawal flow.

Users who unstake in this path will have their frxETH tokens burned but will receive no ETH in return because

No withdrawalRequest is saved.

The withdraw() function depends on withdrawalRequests[msg.sender] being populated.

This effectively results in loss of user funds with no means of recovery through the intended withdraw flow.

Resolution: Fixed

4.2 Medium Risk

4.2.1 Incorrect increasing of end timestamp causes OverExtended delay in syncRewards()

Severity: *Medium risk*

Description: The syncRewards() function contains faulty logic that causes the rewardsCycleEnd to be pushed one full cycle further than necessary. When the function is called late into the current cycle—for example at timestamp 180 with a rewardsCycleLength of 100—it calculates the next rewardsCycleEnd as 200, but then adds an extra 100, setting it to 300. This happens because of the condition:

```
if (end - timestamp < rewardsCycleLength) {
    end += rewardsCycleLength;
}
```

At timestamp 180, the current cycle (ending at 100) is already complete, and only 20 seconds remain to complete the next one. However, instead of moving to 200 as expected, the function pushes the end to 300, delaying the next sync by a full cycle. This causes up to nearly one full cycle of rewards to be skipped and forces users to wait longer than necessary to receive rewards.

Resolution: Fixed

4.2.2 _depositEther Does Not Increment Validator Index, Causing All Deposits to Funnel Into First Validator and Fail Once Full

Severity: *Medium risk*

Description: In the stPlumeMinter contract, the _depositEther() function is responsible for distributing incoming ETH deposits across available validators. This function loops through the validator set to find suitable validators with enough capacity to stake the deposited ETH.

However, there is a critical issue: the index variable, initialized to 0, is never incremented inside the while loop. As a result, the loop evaluates the same validator (validators[0]) in every iteration, preventing the function from ever progressing to the next validator in the list.

When the first validator reaches its staking capacity or becomes inactive, the getNextValidator() function returns a capacity of 0. Since depositSize < minStakeAmount in such cases, the full remainingAmount is stored in the currentWithheldETH variable. This means the user's deposit is held back and never actually staked, despite the user receiving yield-bearing tokens.

Resolution: Fixed

4.3 Informational

4.3.1 submitForValidator() Blocked by Strategic Capacity Manipulation

Severity: *Informational risk*

Description: Intentionally people could leave capacity almost to max - (min stake value + 1) so nobody could call submitForValidator() for this exact validator. Which will result in logic going always on increasing the currentWitheldETH

Resolution: Acknowledged

4.3.2 Missing Length Consistency Check Between amounts and tokens Arrays in claimAll()

Severity: *Informational risk*

Description: In the claimAll() function, the amounts and tokens arrays are expected to correspond to each other by index. However, the code does not validate that both arrays are of the same length before using them in a for loop. If plumeStaking.claimAll() and plumeStaking.getRewardTokens() return arrays of different lengths, this can lead to out-of-bounds access or incorrect logic execution.

To prevent potential errors or misaligned reward handling, a check should be added:

```
require(amounts.length == tokens.length, "Mismatch between amounts and tokens");
```

Resolution: Acknowledged

4.3.3 AllRewardsClaimed is not emitted when all rewards are claimed

Severity: *Informational risk*

Description: In `stPlumeMinter` there is a function called `claimAll` that is used to claim rewards accumulated from all rewards tokens, but here is the case that the event `AllRewardsCLaimed` is not emitted after all rewards have been claimed

Resolution: Acknowledged

4.3.4 `withdrawGov` will always return zero

Severity: *Informational risk*

Description: This function is supposed to return how much owner or timelock(i.e Gov) withdrawed from `plumeStaking` but the return value `amountRestaked` is not updated in the function and as a result it always returns zero.

```
function withdrawGov(uint256 amount) external nonReentrant onlyByOwnGov returns (
    uint256 amountRestaked) {
        _rebalance();
        uint256 balanceBefore = address(this).balance;
        plumeStaking.withdraw();
        uint256 balanceAfter = address(this).balance;
        currentWithheldETH += balanceAfter - balanceBefore;
    }
```

Resolution: Acknowledged