

# Mystic Finance - Security Review



mystic<sup>♦</sup>

9 December 2025

Conducted by:

**Kann**, Lead Security Researcher

**Jun Wang**, Lead Security Researcher

**Pindarev**, Intern

## Table of Contents

<b>1 Disclaimer</b>	<b>3</b>
<b>2 Risk classification</b>	<b>3</b>
2.1 Impact . . . . .	3
2.2 Likelihood . . . . .	3
<b>3 Executive summary</b>	<b>4</b>
<b>4 Findings</b>	<b>5</b>
4.1 Medium . . . . .	5
4.1.1 Temporary DoS of User Withdrawals Due to Partial Plume Withdrawals Creating Artificial Shortfall . . . . .	5
4.2 Low . . . . .	6
4.2.1 Inefficient Max-Percentage Check Prevents Using Valid Partial Capacity . . . . .	6
4.2.2 Restake() May Re-Stake Funds Already Reserved for User Withdrawals . . . . .	8
4.3 Informational . . . . .	8
4.3.1 Wrong Amount Check in unstakeGov Uses Total Validator Stake Instead of This Contract's Stake . . . . .	8
4.3.2 ERC20 Transfer Handling in recoverERC20 . . . . .	9

## 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	<a href="https://github.com/mystic-finance/Liquid-Staking">https://github.com/mystic-finance/Liquid-Staking</a>
Commit hash	e7af6a3201546962eedd17229181138ac9a5830a
Resolution	Fixed
Documentation	<a href="https://docs.mysticfinance.xyz/">https://docs.mysticfinance.xyz/</a>
Methods	Manual review

#### Scope

stPlumeMinter.sol
OperatorRegistry.sol
stPlumeRewards.sol
frxETHMinter.sol
frxETH.sol
Periphery/MyPlumeFeed.sol
Utils/OwnedUpgradeable.sol

#### Issues Found

Critical risk	0
High risk	0
Medium risk	1
Low risk	2
Informational	2

## 4 Findings

### 4.1 Medium

#### 4.1.1 Temporary DoS of User Withdrawals Due to Partial Plume Withdrawals Creating Artificial Shortfall

**Severity:** *Medium risk*

**Description:** When multiple users interact with the unstake and withdraw logic at different times, a user can be temporarily DoS'd from withdrawing their own funds because of how `plumeStaking.withdraw()` and the local deficit check are combined.

Consider this flow:

User A calls unstake with 50 ETH. This amount gets unstaked from Plume once either `withdrawalQueueThreshold` or `nextBatchUnstakeTimePerValidator` is reached.

User B calls unstake with a larger amount, for example 350 ETH. It takes at least four validators to satisfy this (assuming each has around 100 ETH capacity). Eventually, the full 350 ETH for User B is also unstaked on Plume. At this point, Plume holds around 400 ETH of unstaked funds belonging to both User A and User B.

User A then calls withdraw. Inside `_withdraw`, the contract executes:

```
if (totalWithdrawable > 0) {  
    uint256 balanceBefore = address(this).balance;  
    plumeStaking.withdraw();  
    uint256 balanceAfter = address(this).balance;  
    withdrawn = balanceAfter - balanceBefore;  
}
```

This `plumeStaking.withdraw()` call pulls all 400 ETH of cooled funds from Plume into the contract's balance. Plume now shows zero withdrawable ETH, but the contract's internal buffers reflect that ETH.

Later, other users (call them User Group M) also unstake their funds and get cooling/parked. Once their cooldowns finish, their amounts become withdrawable on Plume too.

Now User B calls withdraw. The code sees `totalWithdrawable > 0` again, so it calls `plumeStaking.withdraw()` a second time. This time, only the newly cooled funds from User Group M are withdrawn from Plume and assigned to `withdrawn`. This `withdrawn` value is less than the 350 ETH that User B expects.

The function then hits the deficit logic:

```
if (withdrawn > 0 && withdrawn < amount) {  
    require(amount - withdrawn < fee, "Insufficient funds to cover deficit");  
}
```

Here, `amount` is User B's requested 350 ETH, and `withdrawn` is the much smaller amount that just came from User Group M. The condition `amount - withdrawn < fee` will almost always be false because the fee is tiny compared to the missing difference. As a result, the withdrawal reverts.

At this point, User B is effectively DOS'ed from withdrawing. Every time they try, they will again pull only the small cooled balance from Plume, fail the same `require(amount - withdrawn < fee)` check, and revert. They remain stuck until either another user with a withdrawal request calls withdraw first and drains the Plume balance in a way that avoids the partial-withdraw branch, or enough new unstake events accumulate so that the next `plumeStaking.withdraw()` call returns at least 350 ETH in one shot.

So even though User B has correctly unstaked and the system as a whole has enough ETH across buffers (`currentWithheldETH`, `totalInstantUnstaked`) and future withdrawals, the specific interaction between `plumeStaking.withdraw()` and the `require(amount - withdrawn < fee)` deficit check can temporarily lock that user out of claiming their funds.

Additionally, User B would normally be allowed to withdraw using `currentWithheldETH` alone if the total withdrawable from plume was 0. However, because `withdrawn` is non-zero, the contract is forced into the deficit branch instead of using local liquidity because `withdrawn < amount`. This guarantees a revert as soon as enters check `withdrawn < amount`, making the DOS unavoidable until conditions change.

**Resolution:** Fixed

## 4.2 Low

### 4.2.1 Inefficient Max-Percentage Check Prevents Using Valid Partial Capacity

**Severity:** *Low risk*

**Description:** In the auto-distribution path of `_depositEther`, the contract decides whether a validator can be used by calling `getNextValidator(remainingAmount, validatorId)` with the full `remainingAmount`:

```
while (remainingAmount > 0 && index < numVals) {
    uint256 depositSize = remainingAmount;
    _validatorId = uint16(validators[index].validatorId);
    (uint256 validatorId, uint256 capacity) = getNextValidator(remainingAmount,
        _validatorId);

    if (capacity > 0 && capacity >= minStakeAmount) {
        if (capacity < depositSize) {
            depositSize = capacity;
        }

        if (depositSize < minStakeAmount) {
            currentWithheldETH += remainingAmount;
            return depositedAmount;
        }

        plumeStaking.stake{value: depositSize}(uint16(validatorId));
        remainingAmount -= depositSize;
        depositedAmount += depositSize;
        emit DepositSent(uint16(validatorId));
    }
    index++;
}
```

Inside `getNextValidator`, the validator's new percentage is computed using `depositAmount` equal to that full `remainingAmount`:

```
function getNextValidator(uint256 depositAmount, uint16 validatorId)
    internal
    view
    returns (uint256 validatorId_, uint256 capacity_)
{
    (bool active, , uint256 stakedAmount, ) = plumeStaking.getValidatorStats(
        validatorId);
    uint256 totalStaked = plumeStaking.totalAmountStaked();
    if (!active) return (validatorId, 0);

    (, capacity_) = _getValidatorInfo(validatorId);

    uint256 percentage =
        ((stakedAmount + depositAmount) * RATIO_PRECISION) /
        (totalStaked + depositAmount);

    if (maxValidatorPercentage[validatorId] > 0 &&
        percentage > maxValidatorPercentage[validatorId])
    {
        return (validatorId, 0); // validator is skipped entirely
    }

    if (capacity_ > 0) {
        return (validatorId, capacity_);
    }
    return (validatorId, 0);
}
```

The critical line is:

```
if (maxValidatorPercentage[validatorId] > 0 && percentage > maxValidatorPercentage[
    validatorId]) {
    return (validatorId, 0);
}
```

If sending the entire `remainingAmount` would push a validator above its `maxValidatorPercentage`, `getNextValidator` returns a capacity of 0, and `_depositEther` skips that validator completely. However, `_depositEther` later caps `depositSize` to capacity only if `capacity > 0`, so this partial capping never happens for validators that are rejected at the percentage check.

This means that if a validator still has some allowed headroom under both capacity and max-percentage, but not enough to take the full `remainingAmount`, the contract refuses to use it at all. It does not attempt a partial deposit that would keep the validator within limits.

For example, a validator with max capacity 10 ETH and current stake 8 ETH still has 2 ETH of safe room. If `remainingAmount` is 5 ETH, the percentage is calculated as if 5 ETH were going to this validator. If that hypothetical 5 ETH would violate the max percentage, the validator is excluded, even though putting just 2 ETH there would be perfectly valid. As a result, those 2 ETH are never used for this validator; the loop just moves to the next one with the full 5 ETH still in `remainingAmount`.

Validators that still have valid headroom (for example, 1–2 ETH) can be skipped entirely if the full `remainingAmount` would break the percentage limit.

This leads to under-utilization of individual validator capacities and reduces how evenly stake is spread, since the allocator refuses to use small remaining slots on validators that are close to their limit.

In practice, more ETH may end up routed to fewer validators.

**Resolution:** Acknowledged

#### 4.2.2 Restake() May Re-Stake Funds Already Reserved for User Withdrawals

**Severity:** *Low risk*

**Description:** The `restake()` function can reuse funds that users have already unstaked and are waiting to withdraw. When users call `unstake()`, their ETH enters cooling/parked state under the contract, but `restake()` does not check whether those funds are already committed to pending withdrawal requests. It simply calls:

```
plumeStaking.restake(validatorId, amount);
```

and consumes whatever cooling balance exists.

This allows the rebalancer to restake ETH that users are expecting to withdraw after their cooldown period. If this happens, users may reach the end of their cooldown and find that their withdrawal cannot be fulfilled because their cooled funds were re-staked before they could withdraw them. This can lead to temporary withdrawal delays or failed withdrawals until the protocol unstake cycle produces new cooled funds again.

**Resolution:** Acknowledged

### 4.3 Informational

#### 4.3.1 Wrong Amount Check in unstakeGov Uses Total Validator Stake Instead of This Contract's Stake

**Severity:** *Informational*

**Description:** The `unstakeGov` function currently does:

```
function unstakeGov(uint16 validatorId, uint256 amount)
    external
    nonReentrant
    onlyByOwnGov
    returns (uint256 amountRestaked)
{
    _rebalance();
    (bool active, , uint256 stakedAmount, ) =
        plumeStaking.getValidatorStats(uint16(validatorId)); // total stake on this
        validator

    if (active && stakedAmount > 0 && stakedAmount >= amount) {
        amountRestaked = plumeStaking.unstake(uint16(validatorId), amount);
    }
}
```

Here, `stakedAmount` is the total amount staked on the validator by all users, not the amount staked by this contract. The function should instead compare amount against this contract's own stake on that validator, using:

```
uint256 validatorStakedAmount =  
    plumeStaking.getUserValidatorStake(address(this), validatorId);
```

so the precondition matches what this contract is actually allowed to unstake.

In practice, if amount is larger than this contract's stake, `plumeStaking.unstake` will revert internally, so funds are not at risk. However, the local check is misleading: it may pass when this contract's own stake is smaller, and the revert will only happen deeper in the staking contract. This makes the API less clear and can cause unnecessary reverts instead of failing fast with an accurate condition.

The suggested fix is to keep active from `getValidatorStats`, but replace the `stakedAmount` comparison with a comparison against `getUserValidatorStake(address(this), validatorId)`.

**Resolution:** Acknowledged

#### 4.3.2 ERC20 Transfer Handling in `recoverERC20`

**Severity:** *Informational*

**Description:** The function uses:

```
require(IERC20(tokenAddress).transfer(to, tokenAmount), "recoverERC20: Transfer  
    failed");
```

This assumes every ERC20 token returns a bool. Many tokens do not return a boolean at all, which will cause this call to revert even when the transfer would succeed.

Because of this, the contract may be unable to recover certain ERC20 tokens sent to it by mistake.

**Recommendation**

Use `SafeERC20.safeTransfer` instead. Safe transfer helpers correctly handle tokens that do not return a boolean.

**Resolution:** Acknowledged