SHERLOCK

Sherlock Security Review For Gamma

Public contest prepared for: **Gamma**
Lead Security Expert: **cergyk**
Date Audited: **October 24 - October 27, 2024**

Introduction

Gamma is a protocol for active liquidity management. This contest focuses on using Brevis' ZK coprocessor to validate and incentivize user liquidity positions.

Scope

Repository: GammaStrategies/GammaRewarder

Branch: master

Audited Commit: 50b9775d9fb5a44ec53638acd3eaf694ed7e7417

Final Commit: 1c76eef54bd2aed2d29b41e26f2198acd4656953

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
1	1

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

0xhuh2005
sammy
irresponsible
MaslarovK
0xAadi
PNS
yixuan
0xpetern
tpiliposian
newspacexyz
safdie
Praise03
Naresh
Greed
X0sauce

dimah7
tmotfl
0xMosh
merlin
joshuajee
056Security
Japy69
Ollam
rzizah
vinica_boy
0xDemon
Breeje
Hunter
0xjarix
Artur

0xNirix
Cayde-6
WildSniper
Atharv
Pro_King
0xlemon
0xSolus
Galturok
spark1
cergyk
dobrevaleri
NoOneWinner
Chonkov
silver_eth
ni8mare

Issue H-1: Claim Restriction Prevents Users from Claiming Multiple Epochs

Source:
https://github.com/sherlock-audit/2024-10-gamma-rewarder-judging/issues/212

Found by

056Security, 0xAadi, 0xDemon, 0xMosh, 0xNirix, 0xSolus, 0xhuh2005, 0xjarix, 0xlemon, Artur, Atharv, Breeje, Cayde-6, Chonkov, Galturok, Hunter, Japy69, MaslarovK, NoOneWinner, Ollam, PNS, Pro_King, WildSniper, cergyk, dobrevaleri, irresponsible, joshuajee, merlin, ni8mare, rzizah, sammy, silver_eth, sparkl, vinica_boy

Summary

After a user makes a claim, they are unable to claim rewards for subsequent epochs due to a check that restricts claims based on prior activity.

Root Cause

The handleProofResult function includes a check that requires the claimed amount to be zero (require(claim.amount == 0, ”Already claimed reward.”);). Once a user successfully claims rewards, this condition prevents them from claiming for any future epochs, as their claim amount is no longer zero.

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

For simplicity, let’s assume the distribution rewards are set to 100 blocks, with a total of 100 rewards to be distributed and blocksPerEpoch set to 20, resulting in 5 epochs. In this scenario, the amountPerEpoch is 20. If Kate is eligible to claim for the entire period and, at block 21, she claims rewards for blocks 0-20, the amount recorded in the CumulativeClaim (in the mapping claimed) for Kate for this token and reward distribution will be 20. If Kate attempts to claim rewards for any other epoch, her request would revert due to the check: require(claim.amount == 0 , ”Already claimed reward.”); as claim.amount would be 20. Kate should be eligible to claim for other epochs, but she is not. Github Link

Impact

Users are unable to claim rewards for multiple epochs after their initial claim, resulting in missed opportunities to receive rewards. This restriction diminishes user engagement and overall satisfaction with the protocol.

PoC

No response

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/GammaStrategies/GammaRewarder/pull/2

Issue M-1: Division precision loss in `createDistribution` function lead to incorrect distribution of rewards in `GammaRewarder.sol`

Source: https://github.com/sherlock-audit/2024-10-gamma-rewarder-judging/issues/21

Found by

0xAadi, 0xhuh2005, 0xpetern, Greed, MaslarovK, Naresh, PNS, Praise03, X0sauce, dimah7, irresponsible, newspacexyz, safdie, sammy, tmotfl, tpiliposian, yixuan

Summary

The `createDistribution` function in the contract contains a potential vulnerability related to precision loss during division calculations, which could lead to incorrect distribution of rewards. This occurs because the distribution logic divides `_amount` by fixed values without taking adequate measures to handle precision loss.

Root Cause

The `createDistribution` function calculates `amountPerEpoch` by dividing `realAmountToDistribute` by the number of epochs. In cases where `_amount` is large, this division may lead to precision loss because Solidity's integer division discards the fractional component, potentially causing small discrepancies in each epoch's reward distribution. These inaccuracies accumulate over multiple epochs, leading to an overall loss in reward accuracy. The vulnerability is found in this segment of the `createDistribution` function: https://github.com/sherlock-audit/2024-10-gamma-rewarder/blob/main/GammaRewarder/contracts/GammaRewarder.sol#L108-L147

```
function createDistribution(
        address _hypervisor,
        address _rewardToken,
        uint256 _amount,
        uint64 _startBlockNum,
        uint64 _endBlockNum
    ) external nonReentrant {
        // Other requirements and checks

        uint256 fee = _amount * protocolFee / BASE_9;
        uint256 realAmountToDistribute = _amount - fee;
        uint256 amountPerEpoch = realAmountToDistribute / ((_endBlockNum -
 ↪  _startBlockNum) / blocksPerEpoch);
```

```
        // Further processing
    }
```

In this code, `amountPerEpoch` is calculated by dividing `realAmountToDistribute` by the number of epochs (which is derived from block range). The result may have a fractional part, but Solidity's integer division will truncate it, leading to precision loss.

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

No response

Impact

While this vulnerability does not entirely prevent the function's operation, it introduces inaccuracies in reward distribution that may lead to small amounts being unrewarded over time. This could be noticeable with large distributions or over long durations, where cumulative precision loss can result in rewards lower than anticipated.

PoC

A Hardhat test can demonstrate the impact of precision loss, especially with large distributions where each division results in a truncated value. Setup:

1. Deploy a mock ERC20 token (`MockRewardToken`) and mint tokens to `msg.sender`.
2. Deploy the distribution contract.
3. Set large values for `_amount` and a reasonable range for epochs to see the effect of precision loss.

Create distribution and validate results: Call `createDistribution` and observe `amountPerEpoch` to confirm whether the amount lost to precision affects each epoch's accuracy.

```
const { expect } = require("chai");
const { ethers } = require("hardhat");

describe("Division Precision Loss in `createDistribution`", function () {
    let rewardToken, distributionContract, owner, addr1;
    const largeAmount = ethers.utils.parseUnits("1000000000", 18); // Large amount for distribution
    const protocolFee = 100; // Simulated fee of 0.01%

    before(async function () {
        [owner, addr1] = await ethers.getSigners();

        // Deploy mock ERC20 reward token
        const MockRewardToken = await ethers.getContractFactory("MockRewardToken");
        rewardToken = await MockRewardToken.deploy("Reward Token", "RTK", 18);
        await rewardToken.deployed();

        // Mint tokens to addr1 for testing
        await rewardToken.mint(addr1.address, largeAmount);

        // Deploy the distribution contract
        const DistributionContract = await ethers.getContractFactory("DistributionContract");
        distributionContract = await DistributionContract.deploy(protocolFee);
        await distributionContract.deployed();
    });

    it("Should exhibit precision loss in `amountPerEpoch` calculation", async function () {
        // Set allowance
        await rewardToken.connect(addr1).approve(distributionContract.address, largeAmount);

        // Calculate expected values manually for comparison
        const epochs = 10; // Example block range divided into 10 epochs
        const realAmountToDistribute = largeAmount.mul(9999).div(10000); // Deduct protocol fee
        const expectedPerEpoch = realAmountToDistribute.div(epochs);

        // Execute createDistribution
        await distributionContract.connect(addr1).createDistribution(
            addr1.address,
            rewardToken.address,
            largeAmount,
            1000,
            2000
        );

        // Fetch distribution info and validate precision loss
        const distribution = await distributionContract.getDistribution(addr1.address);
        expect(distribution.amountPerEpoch).to.be.lt(expectedPerEpoch);
    });
});
```

Running this Hardhat test produces output showing that `amountPerEpoch` is lower than expected due to precision loss.

Mitigation

To mitigate this issue, consider modifying the calculation to use a higher precision mechanism, such as scaling the division before converting it back. For example, consider using a `10**18` multiplier to help retain more precision during the division, or calculate `amountPerEpoch` in a way that evenly distributes any rounding discrepancies over epochs.

```
uint256 amountPerEpoch = realAmountToDistribute * 10**18 / numberOfEpochs; // Multiplied to retain precision
amountPerEpoch = amountPerEpoch / 10**18; // Convert back after scaling
```

Alternatively, use a `mod` function to calculate any remainder and distribute it across epochs to ensure accurate distribution.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/GammaStrategies/GammaRewarder/pull/2

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users’ sole risk and is the users’ responsibility.