SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type:	Public
Prepared for:	Gamma
Prepared by:	Sherlock
Lead Security Expert:	pkqs90
Dates Audited:	May 17 - May 20, 2024
Prepared on:	June 14, 2024

Introduction

The locked staking contract allows users to lock an ERC-20 token at various intervals and multipliers and receive fee distributions according to their amounts staked and multipliers

Scope

Repository: GammaStrategies/StakingV2

Branch: Sherlock-Audit

Commit: 0fd03768dda3b15db32cc04269e1110aeb7f07cb

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
2	0

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

pkqs90
joicygiore
0xreadyplayer1
emrekocak
yamato
zraxx
denzi_
KupiaSec
petro1912
T_F_E
HChang26
no
0xRajkumar
h2134
kennedy1030
EgisSecurity
samuraii77

Issue M-1: `earlyExitById()` and `exitLateById()` calls near the end of `lockPeriod` are vulnerable to attacks.

Source:
https://github.com/sherlock-audit/2024-05-gamma-staking-judging/issues/65

Found by

0xRajkumar, 0xreadyplayer1, EgisSecurity, HChang26, KupiaSec, T_F_E, denzi_, emrekocak, h2134, kennedy1030, no, petro1912, pkqs90, samuraii77, yamato, zraxx

Summary

`earlyExitById()` and `exitLateById()` calls near the end of `lockPeriod` are vulnerable to attacks.

Vulnerability Detail

Locks are automatically re-locked at the end of the `lockPeriod` for the lesser of `lockPeriod` or `defaultRelockTime`.

The protocol offers 2 methods for stakers to unstake, `earlyExitById()` and `exitLateById()`. Both methods use `calcRemainUnlockPeriod()` to calculate `unlockTime`.

```
function calcRemainUnlockPeriod(LockedBalance memory userLock) public view
↪   returns (uint256) {
    uint256 lockTime = userLock.lockTime;
    uint256 lockPeriod = userLock.lockPeriod;

    if (lockPeriod <= defaultRelockTime || (block.timestamp - lockTime) <
↪   lockPeriod) {
        return lockPeriod - (block.timestamp - lockTime) % lockPeriod;
    } else {
        return defaultRelockTime - (block.timestamp - lockTime) %
↪   defaultRelockTime;
    }
}
```

`exitLateById()` is the default method, where the lock `id` stops accumulating rewards immediately. The `unlockTime`/cool-down period is calculated by `calcRemainUnlockPeriod()`. Funds are only available for withdrawal after `unlockTime`.

For example: The `unlockTime` is dictated by modulo logic in function `calcRemainUnlockPeriod()`. So if the lock time were 30 days, and the user staked

for 50 days, he would have been deemed to lock for a full cycle of 30 days, followed by 20 days into the second cycle of 30 days, and thus will have 10 days left before he can withdraw his funds.

Based on this design, it is in stakers' best interest to invoke `exitLateById()` towards the end of a `lockPeriod` when there are only a few seconds left before it auto re-locks for another 30 days. This maximizes rewards and minimizes the cool-down period.

If a staker invokes `exitLateById()` 1 minute before auto re-lock, two scenarios can occur depending on when the transaction is mined:

1. If the transaction is mined before the auto re-lock, the staker's funds become available after a 1-minute cool-down period.
2. If the transaction is mined after the auto re-lock, the staker's funds will not be available for another 30 days.

Attackers can grief honest stakers by front-running `exitLateById()` with a series of dummy transactions to fill up the block. If `exitLateById()` is mined after the auto re-lock, the staker must wait another 30 days. The only way to avoid this is to call `exitLateById()` well before the end of the `lockPeriod` when block stuffing is not feasible, reducing the potential reward earned by the staker.

```
function exitLateById(uint256 id) external {
    _updateReward(msg.sender);

    LockedBalance memory lockedBalance = locklist.getLockById(msg.sender, id);
↪   // Retrieves the lock details from the lock list as a storage reference to
↪   modify.

 ->uint256 coolDownSecs = calcRemainUnlockPeriod(lockedBalance);
    locklist.updateUnlockTime(msg.sender, id, block.timestamp + coolDownSecs);

    uint256 multiplierBalance = lockedBalance.amount * lockedBalance.multiplier;
    lockedSupplyWithMultiplier -= multiplierBalance;
    lockedSupply -= lockedBalance.amount;
    Balances storage bal = balances[msg.sender];
    bal.lockedWithMultiplier -= multiplierBalance;
    bal.locked -= lockedBalance.amount;

    locklist.setExitedLateToTrue(msg.sender, id);

    _updateRewardDebt(msg.sender); // Recalculates reward debt after changing
↪   the locked balance.

    emit ExitLateById(id, msg.sender, lockedBalance.amount); // Emits an event
↪   logging the details of the late exit.
```

```
}
```

`earlyExitById()` is the second method to unstake. In this function, the staker pays a penalty but can access funds immediately. The penalty consists of a base penalty plus a time penalty, which decreases linearly over time. The current configuration sets the minimum penalty at 15% and the maximum penalty at 50%.

```
function earlyExitById(uint256 lockId) external whenNotPaused {
    if (isEarlyExitDisabled) {
        revert EarlyExitDisabled();
    }
    _updateReward(msg.sender);
    LockedBalance memory lock = locklist.getLockById(msg.sender, lockId);

    if (lock.unlockTime != 0)
        revert InvalidLockId();

    uint256 coolDownSecs = calcRemainUnlockPeriod(lock);
    lock.unlockTime = block.timestamp + coolDownSecs;
    uint256 penaltyAmount = calcPenaltyAmount(lock);
    locklist.removeFromList(msg.sender, lockId);
    Balances storage bal = balances[msg.sender];
    lockedSupplyWithMultiplier -= lock.amount * lock.multiplier;
    lockedSupply -= lock.amount;
    bal.locked -= lock.amount;
    bal.lockedWithMultiplier -= lock.amount * lock.multiplier;
    _updateRewardDebt(msg.sender);

    if (lock.amount > penaltyAmount) {
        IERC20(stakingToken).safeTransfer(msg.sender, lock.amount -
↪   penaltyAmount);
        IERC20(stakingToken).safeTransfer(treasury, penaltyAmount);
        emit EarlyExitById(lockId, msg.sender, lock.amount - penaltyAmount,
↪   penaltyAmount);
    } else {
        IERC20(stakingToken).safeTransfer(treasury, lock.amount);
    emit EarlyExitById(lockId, msg.sender, 0, penaltyAmount);
    }
}
```

The penalty is calculated in `calcPenaltyAmount()`, using `unlockTime` from `calcRemainUnlockPeriod()`.

```
function calcPenaltyAmount(LockedBalance memory userLock) public view returns
↪   (uint256 penaltyAmount) {
```

```
    if (userLock.amount == 0) return 0; // Return zero if there is no amount
↪   locked to avoid unnecessary calculations.
    uint256 unlockTime = userLock.unlockTime;
    uint256 lockPeriod = userLock.lockPeriod;
    uint256 penaltyFactor;


    if (lockPeriod <= defaultRelockTime || (block.timestamp - userLock.lockTime)
↪   < lockPeriod) {

        penaltyFactor = (unlockTime - block.timestamp) * timePenaltyFraction /
↪   lockPeriod + basePenaltyPercentage;
    }
    else {
        penaltyFactor = (unlockTime - block.timestamp) * timePenaltyFraction /
↪   defaultRelockTime + basePenaltyPercentage;
    }

    // Apply the calculated penalty factor to the locked amount.
    penaltyAmount = userLock.amount * penaltyFactor / WHOLE;
}
```

Using the sample example above, if a user invokes `earlyExitById()` on day 50, the penalty is as follows:

Time penalty = (10 / 30) * 35% Base penalty = 15% Total penalty = 26.66%

However, an issue arises when `earlyExitById()` is triggered near the end of the `lockPeriod`. Depending on when the transaction is mined, two scenarios can occur(extreme numbers were used to demonstrate impact):

1. If `earlyExitById()` is mined at 59 days, 23 hours, 59 minutes, and 59 seconds, the time penalty is essentially 0%.
2. If `earlyExitById()` is mined exactly at 60 days, this results in 100% of the time penalty.

The 1-second difference can mean the difference between the minimum penalty and the maximum penalty. An unexpectedly high penalty can occur if the transaction is sent with lower-than-average gas, causing it not to be picked up immediately. Attackers can cause stakers to incur the maximum penalty by block stuffing and delaying their transaction.

Impact

Funds may be locked for longer than expected in `exitLateById()` Penalty may be greater than expected in `earlyExitById()`

Code Snippet

https://github.com/sherlock-audit/2024-05-gamma-staking/blob/main/StakingV2/src/Lock.sol#L313 https://github.com/sherlock-audit/2024-05-gamma-staking/blob/main/StakingV2/src/Lock.sol#L349 https://github.com/sherlock-audit/2024-05-gamma-staking/blob/main/StakingV2/src/Lock.sol#L596 https://github.com/sherlock-audit/2024-05-gamma-staking/blob/main/StakingV2/src/Lock.sol#L569

Tool used

Manual Review

Recommendation

No recommendation for `exitLateById()` since the optimal way to use this function is near the end of a `lockPeriod`.

Consider adding slippage protection to `earlyExitById()`.

```
-   function earlyExitById(uint256 lockId) external whenNotPaused {
+   function earlyExitById(uint256 lockId, uint256 expectedAmount) external
 ↪  whenNotPaused {
        if (isEarlyExitDisabled) {
            revert EarlyExitDisabled();
        }
        _updateReward(msg.sender);
        LockedBalance memory lock = locklist.getLockById(msg.sender, lockId);

        if (lock.unlockTime != 0)
            revert InvalidLockId();

        uint256 coolDownSecs = calcRemainUnlockPeriod(lock);
        lock.unlockTime = block.timestamp + coolDownSecs;
        uint256 penaltyAmount = calcPenaltyAmount(lock);
        locklist.removeFromList(msg.sender, lockId);
        Balances storage bal = balances[msg.sender];
        lockedSupplyWithMultiplier -= lock.amount * lock.multiplier;
        lockedSupply -= lock.amount;
        bal.locked -= lock.amount;
        bal.lockedWithMultiplier -= lock.amount * lock.multiplier;
        _updateRewardDebt(msg.sender);

+       require(expectedAmount >= lock.amount - penaltyAmount);
        if (lock.amount > penaltyAmount) {
            IERC20(stakingToken).safeTransfer(msg.sender, lock.amount -
 ↪  penaltyAmount);
            IERC20(stakingToken).safeTransfer(treasury, penaltyAmount);
```

```
            emit EarlyExitById(lockId, msg.sender, lock.amount - penaltyAmount,
↪   penaltyAmount);
        } else {
            IERC20(stakingToken).safeTransfer(treasury, lock.amount);
        emit EarlyExitById(lockId, msg.sender, 0, penaltyAmount);
        }
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/GammaStrategies/StakingV2/commit/001c056122874fe0e3fa6ece8383f2eafaf12cf5

0xreadyplayer1

Hi @santipu03 , i hope you are good. I've following concerns on the cited dups .

#45 is not a valid duplicate as it takes about user's action of calling exitlatebyId multiple times

#307 is does not show any loss of funds and vaguely states and do not clarify how does the issue actually cause loss of funds

#59 demonstrates the `use of block stuffing to force staker into another cycle` and does not lead to user losing funds - also as you mentioned in my issue `This isn't a feasible attack because the attacker would be losing tens of thousands of dollars each day (gas prices would go to the moon when an address is filling all blocks) just to gain nothing but to grieve others.` although not for days but tons of blocks would be needed to be filled to force the victim into new cycle which is also infeasible additionally since there is `no loss of funds due to getting higher penalty` i don't believe this is a valid duplicate and even valid in the first place

The above mentioned according my reasons above are invalid and should not be duplicates‘

however i would be looking forward to what others think.

omar-ahsan

Escalate

I would like to escalate regarding the grouping of these issues

There are two impacts caused in two different functions by not having a protection parameter.

1. Causes loss of funds for the user through greater penalty amount.
2. Causes locking of funds for the user.

Submissions that identify both impacts

#65 #42 #228 #294 #62 (0xRajKumar) #115 (0xRajKumar) #209 (pkqs90) #212 (pkqs90)

Submissions that only identify one impact

#15 #16 #41 #95 #97 #111 #150 #202

Furthermore I am not sure if #59 and #307 should be placed in any of these groups because I believe they are invalid

I believe it is only fair for the Watsons who have identified both impacts to be treated in a separate group, I do agree that the fix is the same for both functions, but the underlying functions and impacts are different.

sherlock-admin3

> Escalate
>
> I would like to escalate regarding the grouping of these issues
>
> There are two impacts caused in two different functions by not having a protection parameter.
>
> 1. Causes loss of funds for the user through greater penalty amount.
> 2. Causes locking of funds for the user.
>
> Submissions that identify both impacts
>
> #65 #42 #228 #294 #62 (0xRajKumar) #115 (0xRajKumar) #209 (pkqs90) #212 (pkqs90)
>
> Submissions that only identify one impact
>
> #15 #16 #41 #95 #97 #111 #150 #202
>
> Furthermore I am not sure if #59 and #307 should be placed in any of these groups because I believe they are invalid
>
> I believe it is only fair for the Watsons who have identified both impacts to be treated in a separate group, I do agree that the fix is the same for both functions, but the underlying functions and impacts are different.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

0xreadyplayer1

Hi there,

The issue MUST be HIGH @santipu03

because the issue causes `Definite loss of funds`

Additionally due to high likelihood and impact , the issue clearly deems to be a valid high..

looking forward to know what others think.

santipu03

Regarding the escalation of @omar-ahsan:

I considered the root cause of this issue to be the following: **The lack of slippage parameters on exit functions will cause an unexpected loss or a lock of funds**.

I agree that there are two impacts that will be caused by two functions, however, the root cause is still the same. According to the Sherlock guidelines, all issues that identify the same root cause with a valid attack path and impact should be considered duplicates, and that's why I grouped all those issues under this report.

Issue #59 I considered valid because it identifies the root cause and describes a valid attack path and impact. Issue #307 also describes a correct scenario where a user may have his transaction delayed and pay extra penalties.

Regarding @0xreadyplayer1 comment, please don't try to chop definitions so they fit in your argument, the definition of a high-severity issue includes that it must not require extensive requirements or external states. For this issue to be triggered, a user must exit a lock within the last minutes of their period, which is already an unlikely scenario overall.

Moreover, most of Ethereum nodes discard a transaction when it has been in the mempool for some time and is still not confirmed, if I remember correctly it was about 2 hours for Geth. This means that if a transaction hasn't been confirmed for 2 hours, it won't get executed at all. Taking this into account, this issue will only be triggered when a user tries to exit a lock with a low-gas transaction less than 2 hours before the lock period finishes, which I consider an extensive external condition.

For this reason, this issue warrants medium severity.

0xreadyplayer1

Agree @santipu03 will take care next time sir

samuraii77

@santipu03 While I wouldn't necessarily argue this issue is definitely a high, I disagree with your argument. You said that a user exiting a lock within the last minutes of the period is unlikely but that is not the case. As a matter of fact, especially for exiting late, every user would try to exit as late as possible, even in the last second if they could in order to not miss out on new potential rewards.

omar-ahsan

@santipu03 I agree the root cause is the same that there exists a lack of slippage parameters on both functions

There is also a statement in the rules guideline which states

> In case the same vulnerability appears across multiple places in different contracts, they can be considered duplicates. The exception to this would be if underlying code implementations, impact, and the fixes are different, then they can be treated separately.

My escalation stems from the recent judgement made in Zivoe Content on this particular group of issues where the above statement was used to group certain issues

The head agreed that the `core issue` is the same, the function containing the vulnerability is the same but the the impact and fixes are different hence the issues were regrouped accordingly. I believe that not all 3 are required i.e. `code implementations`, `impact`, and the `fixes` to be different. In this case the functions and impacts are different while the fix is the same for both functions.

I would like to hear the opinion of the head on this and accept the decision which will be made by them.

petro1912

@omar-ahsan Have you looked at my issue carefully? Two impacts of early and late exit were mentioned. “User may not withdraw at expectation time, or pay more penalty than expected.” It involves executing a transaction almost at the nearly unlock time. Paying more penalty means losing user's funds. I don't understand why my issue is not valid.

omar-ahsan

@petro1912 Your submission targets the `calcRemainUnlockPeriod()` function but does not explain through which functions the user can face these impacts, so the path is not defined, only the function through which the time is calculated.

santipu03

> In case the same vulnerability appears across multiple places in different contracts, they can be considered duplicates. The exception to this

> would be if underlying code implementations, impact, and the fixes are different, then they can be treated separately.

@omar-ahsan In this case, the impact is different but the fixes are the same, so following the above rules, all reports should be considered duplicates. The rules declare an exception, but this issue here doesn't meet the requirements for that exception.

omar-ahsan

@santipu03 like I mentioned, the head made a ruling in Zivoe when the function was same, impact was different and the fixes were different, this leads me to believe that not all 3 are required to be different. Here in this case the functions and impacts are different but fixes are same.

nevillehuang

I believe medium severity is appropriate here, since early exits will be an uncommon occurence and users executing an early exit for a lock within the last minutes.seconds of their period, further reduces likelihood.

@omar-ahsan I also agree with lead judge comments here regarding duplication, since it is clear within sherlock rules for duplication of issues with same root cause and fixes.

guhu95

@nevillehuang if #45 is a user mistake, why is this one not a user mistake? Submitting a transaction minutes or seconds before expiry is done knowingly, so the user is accepting the risk - all users know that blockhain transactions aren't immediate, and in this case they know their lock will be re-locked if not included in time.

There is no vulnerability here, the contract works as intended, and all that happened is that the user took a reckless risk for likely 0 reward.

They gain 0 reward form this because a reward distribution must happen in the few seconds / minutes before the lock expiry (because rewards are immediately distributed) - making this behavior not only a user mistake, but also a highly unlikely one due to lack of incentive.

nevillehuang

@guhu95 This issue describes a scenario where the user is calling the functions correctly before lock expires but is forced to suffer a higher penalty from early exits (a difference as large as 45% penalty). There is a loss of fund here and there is a fix that could prevent this from occuring.

In issue #45, as long as user waits for finality and not call late exits twice, their funds will never be locked, and relocks will be performed as intended per code

logic (since their late exit would apply for an expired lock), so they will not lose their funds, and in fact would have the possibility to gain more rewards from the reward while retaining multiplier. After discussions with sherlocks internal judge, we conclude it is invalid/low severity based on user error.

guhu95

@nevillehuang

> .. but is forced to suffer a higher penalty from early exits

They are not forced, they take the risk knowingly by submitting the tx seconds before the re-lock. They know they will be relocked if the transaction executes late.

Furthermore, there's no incentive to do so as explained above. The only chance of any reward is if a distribution by the admin happens in these few seconds, and if e.g., they will happen every two weeks it means there - so if they submit 10 seconds before - there's a 0.0008% chance a distribution will happen in these 10 seconds.

> There is a loss of fund here and there is a fix that could prevent this from occuring.

There is no loss of funds since a user exiting seconds before will not call "early" exit, they will call "late" exit. So this is exactly the same situation - they will just get relocked.

guhu95

The reason a user exiting seconds before will not call "early" exit, is exactly because they lose some of their stake (a tiny amount) for no gain. That's because calling "late" will lose nothing, and will result in all of the stake returned immediately (because it's seconds before expiry).

If the user is so sensitive to a minuscule chance of a reward distribution happening in the last few seconds, they will also avoid losing part of their stake to the penalty, so will call "late".

This issue is entirely a user mistake, that the user commits with full knowledge, for no gain, and then just gets relocked for 30 days. Everything works as intended.

nevillehuang

At the point when user calls an early exit, the state at which he expects it to be included is correct, that is he should not be penalized for an early exit/should not be penalized so heavily for an early exit, so he is using the function correctly. This is not user mistake to me, so I believe medium severity is appropriate.

However, for issue #45 to be true, the user must have called late exit twice, which goes against the intended use case of the functionality. So my final decision would still be to reject escalation and keep this issue a valid medium severity.

guhu95

> At the point when user calls an early exit, the state at which he expects it to be included is correct, that is he should not be penalized for an early exit/should not be penalized so heavily for an early exit, so he is using the function correctly. This is not user mistake to me, so I believe medium severity is appropriate.

A user will not call early exit in this issue, please have another look at https://github.com/sherlock-audit/2024-05-gamma-staking-judging/issues/65#issuecomment-2143623709 . The "early exit", with penalty, is an incorrect scenario in this case. The user will only call "late exit".

> the state at which he expects it to be included is correct

Why is that correct? The users knows that they will be auto-relocked, and they know how penalties work, and they know blockchains execution happens with a delay. The user knows they have a very high chance of being relocked if calling seconds before, so they are aware.

joicygiore

The project itself has a complete exit mechanism (of course, we can think that the exit conditions are unfriendly, but you have accepted this setting when you choose to enter.). Users have the right to choose to maximize their benefits to execute the exit, and they should also bear the risk of failure, which is normal. We can assume that adding exit protection will still result in the following two situations:

1. The same operation of maximizing benefits will eventually be rolled back and continued to deposit, the difference is that there is a return, so the question is, since you can accept continuous deposits, why do you want to exit?
2. If you want to exit, it is impossible to maximize your profits. Execute early and exit successfully. I don't think anyone would pay off their credit card at the last minute, as they will also be fined.

nevillehuang

I still stand by my comments here, at the point where user calls an early exit, they should expect to not be penalized, so this is not considered them not utilizing the protocol inappropriately. Still planning to reject escalation and leave issue as it is.

WangSecurity

Result: Medium Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- omar-ahsan: rejected

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-2: Integer overflow when calculating rewards

Source: https://github.com/sherlock-audit/2024-05-gamma-staking-judging/issues/207

Found by

joicygiore, pkqs90

Summary

The protocol uses a Sushi Masterchef like method for maintaining rewards for each user. The `cumulatedReward` variable is used for keeping track of the accumulated amount of assets per staking token. However, since `cumulatedReward` is scaled by 1e36, it may lead to overflow during reward calculation.

Vulnerability Detail

Let's see how the reward is calculated:

1. `cumulatedReward` is maintained as the cumulative asset/share ratio scaled by 1e36.
2. Earned rewards and rewardDebt are calculated by `cumulatedReward` times the amount of shared token (with multiplier)

The issue here is `cumulatedReward` is scaled by 1e36, which is too large, and is susceptible to grief attacks. An example is:

1. At the beginning of the Lock contract, attacker initially stakes 1 wei of staking token, and deposits 1e18 reward token.
2. Attacker calls `notifyUnseenReward()` to accumulate the reward. The `cumulatedReward` is now 1e18 * 1e36 / 1 = 1e54.
3. Then, when calculating the earned rewards, if anyone has 1e18 staking tokens, the reward calculation (and the reward debt) would be up to 1e54 * 1e18 = 1e72.

Note that uint256.max is around 1e78, and the limit would be easily hit if the tokens in step 1 and step 3 is 1000e18 instead of 1e18.

```
    function _notifyReward(address _rewardToken, uint256 reward) internal {
        if (lockedSupplyWithMultiplier == 0)
            return; // If there is no locked supply with multiplier, exit
↪   without adding rewards (prevents division by zero).
```

```
        Reward storage r = rewardData[_rewardToken]; // Accesses the reward
↪   structure for the specified token.
>       uint256 newReward = reward * 1e36 / lockedSupplyWithMultiplier; //
↪   Calculates the reward per token, scaled up for precision.
>       r.cumulatedReward += newReward; // Updates the cumulative reward for the
↪   token.
        r.lastUpdateTime = block.timestamp; // Sets the last update time to now.
        r.balance += reward; // Increments the balance of the token by the new
↪   reward amount.
    }

    ...

    function _earned(
        address _user,
        address _rewardToken
    ) internal view returns (uint256 earnings) {
        Reward memory rewardInfo = rewardData[_rewardToken]; // Retrieves reward
↪   data for the specified token.
        Balances memory balance = balances[_user]; // Retrieves balance
↪   information for the user.
>       earnings = rewardInfo.cumulatedReward * balance.lockedWithMultiplier -
↪   rewardDebt[_user][_rewardToken]; // Calculates earnings by considering the
↪   accumulated reward and the reward debt.
    }
```

Impact

Integer overflow during reward calculation.

Code Snippet

- https://github.com/sherlock-audit/2024-05-gamma-staking/blob/main/StakingV2/src/Lock.sol#L493-L494
- https://github.com/sherlock-audit/2024-05-gamma-staking/blob/main/StakingV2/src/Lock.sol#L623

Tool used

Manual review

Recommendation

Couple of ways for mitigation:

1. Use 1e12 as scale factor (like in Masterchef) instead of 1e36.
2. Since the staking token is always GAMMA (which has 18 decimals) and is currently priced at $0.117, the `lockedSupplyWithMultiplier` can be initially set to 1e18 to avoid `cumulatedReward` to be too large. The loss is be negligible for initial stakers.
3. Add a minimum staking amount limit (e.g. 1e18).

Discussion

santipu03

This issue has been marked as invalid for the following reasons:

1. Before users start staking, the admins have to set the staking token.
2. Before anyone can send rewards to the contract, the admins have to set the reward tokens.
3. For the reasons above, the admins can set the staking token first, wait for some stakers, and later set the reward tokens and send some to the contract.
4. Even in the improbable case that this attack is successfully executed, because the only staker is the attacker, the admins can simply redeploy a new contract and stake a minimum amount first to avoid this issue again. If some users have started staking between the attack and the admin's action, they can simply withdraw their locks.

In conclusion, there won't be any loss of funds and the contract can simply be redeployed.

santipu03

@bjp333 What do you think about this issue?

bjp333

I think this issue is a bit of a stretch. We will definitely have more than 1 wei staked prior to any distributions taking place.

pkqs90

Escalate

The overflow scenario does not **only** occur during the beginning of the lock contract. As stated in the original issue, the beginning of the contract is only given as an example, since it is where this is likely to happen.

This issue may also occur when the contract has been active for a while. The reward tokens are already set, and due to users staking/unstaking, if there exists a moment where the `lockedSupplyWithMultiplier` is too little, this overflow issue will still occur.

Also, the original code itself handles the case where `lockedSupplyWithMultiplier` is zero, so it only makes sense that it should also handle where `lockedSupplyWithMultiplier` is 1 wei or 1e3 wei or something.

```
    function _notifyReward(address _rewardToken, uint256 reward) internal {
>       if (lockedSupplyWithMultiplier == 0)
>           return; // If there is no locked supply with multiplier, exit
↪   without adding rewards (prevents division by zero).

        Reward storage r = rewardData[_rewardToken]; // Accesses the reward
↪   structure for the specified token.
        uint256 newReward = reward * 1e36 / lockedSupplyWithMultiplier; //
↪   Calculates the reward per token, scaled up for precision.
        r.cumulatedReward += newReward; // Updates the cumulative reward for the
↪   token.
        r.lastUpdateTime = block.timestamp; // Sets the last update time to now.
        r.balance += reward; // Increments the balance of the token by the new
↪   reward amount.
    }
```

Note that if this overflow issue happens, the entire protocol would brick. This is because reward calculation is used for all operations, including staking, early exiting, late exiting, and getting rewards. This means users would not be able to withdraw their staked tokens nor rewards once `cumulatedReward` overflows, which is a disaster.

Quoting the definition of a medium severity issue, this issue would fall under "cause loss of fund" but "requires certain external conditions or specific states", and the loss itself definitely exceeds "small, finite amount of funds".

> Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The losses must exceed small, finite amount of funds, and any amount relevant based on the precision or significance of the loss.

sherlock-admin3

> Escalate
>
> The overflow scenario does not **only** occur during the beginning of the lock contract. As stated in the original issue, the beginning of the contract is only given as an example, since it is where this is likely to happen.

This issue may also occur when the contract has been active for a while. The reward tokens are already set, and due to users staking/unstaking, if there exists a moment where the `lockedSupplyWithMultiplier` is too little, this overflow issue will still occur.

Also, the original code itself handles the case where `lockedSupplyWithMultiplier` is zero, so it only makes sense that it should also handle where `lockedSupplyWithMultiplier` is 1 wei or 1e3 wei or something.

```
    function _notifyReward(address _rewardToken, uint256 reward) internal {
>       if (lockedSupplyWithMultiplier == 0)
>           return; // If there is no locked supply with multiplier, exit
↪   without adding rewards (prevents division by zero).

        Reward storage r = rewardData[_rewardToken]; // Accesses the reward
↪   structure for the specified token.
        uint256 newReward = reward * 1e36 / lockedSupplyWithMultiplier; //
↪   Calculates the reward per token, scaled up for precision.
        r.cumulatedReward += newReward; // Updates the cumulative reward
↪   for the token.
        r.lastUpdateTime = block.timestamp; // Sets the last update time to
↪   now.
        r.balance += reward; // Increments the balance of the token by the
↪   new reward amount.
    }
```

Note that if this overflow issue happens, the entire protocol would brick. This is because reward calculation is used for all operations, including staking, early exiting, late exiting, and getting rewards. This means users would not be able to withdraw their staked tokens nor rewards once `cumulatedReward` overflows, which is a disaster.

Quoting the definition of a medium severity issue, this issue would fall under "cause loss of fund" but "requires certain external conditions or specific states", and the loss itself definitely exceeds "small, finite amount of funds".

> Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The losses must exceed small, finite amount of funds, and any amount relevant based on the precision or significance of the loss.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour

escalation window closes. After that, the escalation becomes final.

santipu03

Even though it is true that this issue can be triggered when the contract has been active for a while, it would require that the staked value is almost null, which is highly unlikely. Because the staking token is GAMMA, which is a token currently valued at ~0.14 USD and with 18 decimals, it's **almost impossible** that during the life of the contract, there are less than 1e18 tokens staked in total.

In a similar way, the classic vault inflation attack can always be triggered when `totalSupply` is 0, but the only real risk is when a Vault is deployed because it's the only realistic scenario where the total value deposited is strictly 0.

pkqs90

I agree that this is unlikely to happen. However,

1. When talking about the severity of an issue, we should not talk about how likely it is to happen (some *almost impossible* examples that actually happened: the Luna crash, or USDC depeg).
2. The Sherlock rules also does not talk about the *possibility*, since it is impossible to quantify. The scenario described above is possible (however unlikely) and falls under "certain external conditions or specific states".

   Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The losses must exceed small, finite amount of funds, and any amount relevant based on the precision or significance of the loss.

santipu03

Also, take into account that there will be stakes locked for 720 days, which belong to the team and investors, so the requirements necessary for this bug to appear are not realistic at all.

I don't believe it's fair to shield behind the expression *"requires certain external conditions or specific states"* to defend issues that will never be triggered in reality.

pkqs90

This issue has been marked as invalid for the following reasons:

1. Before users start staking, the admins have to set the staking token.
2. Before anyone can send rewards to the contract, the admins have to set the reward tokens.
3. For the reasons above, the admins can set the staking token first, wait for some stakers, and later set the reward tokens and send some to the contract.

4. Even in the improbable case that this attack is successfully executed, because the only staker is the attacker, the admins can simply redeploy a new contract and stake a minimum amount first to avoid this issue again. If some users have started staking between the attack and the admin's action, they can simply withdraw their locks.

In conclusion, there won't be any loss of funds and the contract can simply be redeployed.

I'd also like to dispute the initial reasons for invalidating this issue.

- For point 3, there would be no incentive for stakers if the reward token is not set. It is very likely that only *after* the admins set the reward token, would the users begin to stake.
- For point 4, in order for the attacker to overflow `cumulatedReward`, he would have to be the first staker. He may even try to frontrun the original first staker. The point here is, after the attacker performs this attack, the staking of the following users would be affected. The admins can definitely just redeploy a new contract to fix this, but this would be unfair to users that has already staked in the original contract, and their tokens would be locked up for nothing.

santipu03

- For point 3, there would be no incentive for stakers if the reward token is not set. It is very likely that only after the admins set the reward token, would the users begin to stake.

However, before users would begin to stake, the developers would have already configured the locks for the team and investors, making it impossible to trigger this issue because many tokens are already staked. Realistically, the only moment an attacker can trigger this bug is at market deployment, but it won't be possible to execute the attack if the developers configure the contract correctly and set the locks for the team and investors before setting any reward tokens.

Even under extreme circumstances where developers fail to configure the "team and investor" locks and deploy the contract, the worst outcome would necessitate redeploying the contract. If users have already staked in a compromised contract, admins could remove penalties, allowing users to withdraw their stakes early without consequences and reinvest in the new contract. Thus, even if this vulnerability were exploited, its actual impact would effectively be negligible.

nevillehuang

This issue seems to be contigent on admin actions before the first notification of rewards. The difference between this and a first depositor inflation attack is the attack cannot immediately happen as long as the admin has taken the precautions

(stake a minimum amount e.g. 1e18 before setting reward tokens), which is the case as well for ERC4626 vaults wherein a admin can deposit/mint a minimum amount of shares in deployment scripts

I also note #72 and #313 as possible duplicates. Will discuss internally with lead judge. Historically based on my judging experience, first inflation attack and its mitigations must be explicitly stated as a known risk, so I am inclined to believe this issue is valid for now, although still considering the extensive admin actions required highlighted by lead judge before notification of rewards is allowed

cc: @WangSecurity

0xRajkumar

Considering all the points @santipu03 said, I would say the likelihood is very low. Therefore, I believe this is informational at best.

nevillehuang

I believe this issue to be valid with low likelihood and high impact because

1. The admin actions to prevent this "inflation attack" is not highlighted in contest details and known risks + public discord channels
2. The scenario of low amount of stakers resulting in extremely low lockedBalance while unlikely, is possible (if many users unstake), and will brick all subsequent stakers from getting rewards

Planning to accept escalation and make issue and duplicates a valid medium severity.

Evert0x

Result: Medium Has Duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- pkqs90: accepted

nevillehuang

Considering issue #313 as the only duplicate of this issue asilssue #72 fails to identify the correct attack path.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/GammaStrategies/StakingV2/pull/4

sherlock-admin2

The Lead Senior Watson signed off on the fix.

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.