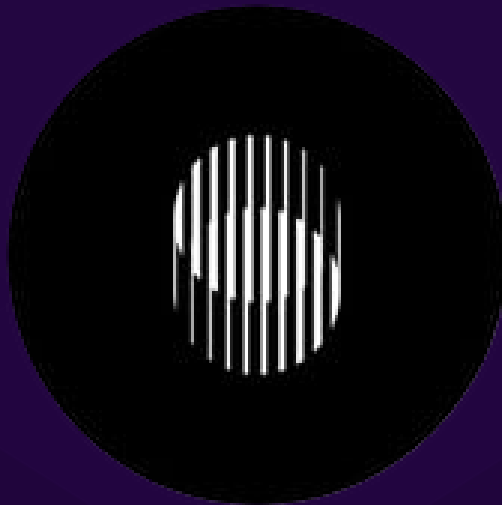




SHERLOCK

SHERLOCK SECURITY REVIEW FOR



Contest type:	Private
Prepared for:	SuperBoring
Prepared by:	Sherlock
Lead Security Expert:	<u>pkqs90</u>
Dates Audited:	July 11 - July 17, 2024
Prepared on:	August 12, 2024



Introduction

SuperBoring, powered by the powerful streaming DEX system TOREX, adds incentive program such as a utility token (\$BORING), staking referral, and distribution fees.

Scope

Repository: superfluid-finance/averagex-contracts-cloned

Branch: master

Commit: 4128cc3b8186aacdf4fdaa020eddc97d8292b09c

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
3	4

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues



pkqs90
y4y

KupiaSec
samuraii77

alexzoid



Issue H-1: Referrer can get less referral reward due to `userData.referrer` being overwritten in `SuperBoring::onFlowChanged()`

Source: <https://github.com/sherlock-audit/2024-06-superboring-judging/issues/6>

Found by

pkqs90, y4y

Summary

Referrers can get 5% extra BORING token for their referral, however a trader's referrer info can be overwritten in `onFlowChanged`, and cause original referrer to lose rewards.

Vulnerability Detail

In `onFlowChanged`, if there is a previous flow record for trader, `userData.distributor` and `userData.referrer` will be set to previous ones:

```
if (prevFlowRate > 0) {
    userData = InFlowUserData({
        distributor:
    ↪ DistributionFeeDIP.getCurrentDistributor(ITorex(msg.sender), trader),
        referrer : QuadraticEmissionTIP.getCurrentReferrer(ITorex(msg.sender),
    ↪ trader)
    });
}
```

And this `userData.referrer` is used somewhere else later, to get the SleepPod of such address:

```
userData.referrer =
    ↪ address(InitialStakingTIP.getOrCreateSleepPod(sleepPodBeacon,
    ↪ userData.referrer));
```

The logic for `getOrCreateSleepPod` is to create a pod for address, if the address has no pod registered in storage. After this, at the end of the function, emission info is updated:

```
QuadraticEmissionTIP.onInFlowChanged(emissionTreasury, ITorex(msg.sender),
                                     trader, userData.referrer,
```



```
prevFlowRate, newFlowRate);
```

And in this function, we see:

```
        if (oldRefData.referrer == referrer) { // this branch is a small
↪ optimization
            // invariant: oldRefData.referrer != referrer != address(0)
            emissionTreasury.updateMemberEmissionUnits(address(torex),
↪ referrer,

↪ emissionPool.getUnits(referrer)

                                                    + newReferralUnits -
↪ oldRefData.referrerUnits);
        } else {
            if (referrer != address(0)) {
                emissionTreasury.updateMemberEmissionUnits(address(torex),
↪ referrer,

↪ emissionPool.getUnits(referrer)

                                                    +
↪ newReferralUnits);
            }
            if (oldRefData.referrer != address(0)) {
                emissionTreasury.updateMemberEmissionUnits(address(torex),
↪ oldRefData.referrer,

↪ emissionPool.getUnits(oldRefData.referrer)

                                                    -
↪ oldRefData.referrerUnits);
            }
        }
        $.referrals[torex][trader] = ReferralData(referrer,
↪ toUint96(newReferralUnits));
        emit ReferrerUpdated(torex, trader, newTraderUnits, referrer,
↪ newReferralUnits);
    }
}
```

Which compares the referrer to the old one, and updates the record eventually.

Let's assume Bob, who created a flow for the first time, and set Alice as his referrer, as his prevFlow is 0, this will not fetch his existing referrer and distributor info. Continue down the function, a pod for both Bob and Alice will be created, assuming both of them are new to the protocol.

Then, emission will be updated, where the trader will be Bob's pod address, while



referrer will be Alice's pod address. This is all good, until Bob made a change to the flow, and triggers `onFlowChanged` again. Now, as Bob has a previous flow in record, `userData.referrer` will be set to his existing referrer address, which is Alice's pod address, this makes sense, as Alice's rewards are sent to her pod. However, a few lines later into the function, when getting pod for Bob and his referrer, Bob's pod will be correctly fetched, but not referrer's. Because `userData.referrer`, which is used for getting pod, is actually a pod itself. Since there is no way a pod to have its own pod, a new pod will be created, and returned for the function.

Now, Alice's pod's pod will be `userData.referrer`, and passed into updating emission units, makes this address the new referrer, until next time Bob makes some change, this pod's address will be fetched for referrer, and when getting pod for this address, it will be none, as this address is also a pod, so a new pod will be created for it, and the cycle continues.

Impact

Original referrer will get less BORING reward.

Code Snippet

```
if (prevFlowRate > 0) {
    userData = InFlowUserData({
        distributor:
        ↪ DistributionFeeDIP.getCurrentDistributor(ITorex(msg.sender), trader),
        referrer : QuadraticEmissionTIP.getCurrentReferrer(ITorex(msg.sender),
        ↪ trader)
    });
}
```

Tool used

Manual Review

Recommendation

Move the else branch down to the pod address part, as the referrer will be a pod's address already.

Discussion

hellwolf



I will have to review tomorrow on this one.

hellwolf

First of all,

I must admit, it's hard for me to answer in verbal text, but my hunch is that:

```
userData.referrer =  
↳ address(InitialStakingTIP.getOrCreateSleepPod(sleepPodBeacon,  
↳ userData.referrer));
```

My intention here is that this line ensures that the referrer is "normalized" to its pod address.

I'd like further examine this in test cases that ensure the described behavior from the watson is not happening.

For now, I will mark it as "in dispute." If you think I missed something, please provide a test case. In any case, I will provide one very soon.

foufrix

request poc

sherlock-admin4

PoC requested from @brandonshiyay

Requests remaining: 1

brandonshiyay

request poc

I will take a deeper look, but as my full time job is kinda busy recently, so I may not have time until the weekends, is it alright?

hellwolf

Hi,

Yes, I can confirm.

The getCurrentReferrer logic is faulty, and it will be updated.

hellwolf

```
diff --git a/packages/evm-contracts/src/BoringPrograms/SleepPod.sol  
↳ b/packages/evm-contracts/src/BoringPrograms/SleepPod.sol  
index 88e8cf1..531aa8a 100644  
--- a/packages/evm-contracts/src/BoringPrograms/SleepPod.sol  
+++ b/packages/evm-contracts/src/BoringPrograms/SleepPod.sol
```



```

@@ -70,6 +70,7 @@ function createSleepPodBeacon(address admin, ISuperToken
↳ boringToken) returns (U
    event SleepPodCreated(address indexed staker, SleepPod indexed pod);

    function createSleepPod(UpgradeableBeacon sleepPodBeacon, address staker)
↳ returns (SleepPod pod) {
+   assert(staker != address(0)); // use-site be aware!
    bytes memory initCall =
↳   abi.encodeWithSelector(SleepPod.initialize.selector, staker);
    pod = SleepPod(address(new BeaconProxy(address(sleepPodBeacon), initCall)));
    emit SleepPodCreated(staker, pod);
diff --git a/packages/evm-contracts/src/SuperBoring.sol
↳ b/packages/evm-contracts/src/SuperBoring.sol
index 4e35b38..34d59e7 100644
--- a/packages/evm-contracts/src/SuperBoring.sol
+++ b/packages/evm-contracts/src/SuperBoring.sol
@@ -221,8 +221,14 @@ contract SuperBoring is UUPSProxiabile, Ownable,
↳ ITorexController, IDistributorSt
    // This may revert, and it will make create/update flow fail.
    userData = abi.decode(userDataRaw, (InFlowUserData));
    if (trader == userData.referrer) revert NO_SELF_REFERRAL();

+
+   if (userData.referrer != address(0)) {
+   // modifying these value in-place, to workaround solidity
↳ stack too deep issue
+   userData.referrer =
↳ address(InitialStakingTIP.getOrCreateSleepPod(sleepPodBeacon,
+
+   userData.referrer));
+   }
    } else {
-   // during update flow, we keep same distributor and referrer
+   // during update flow, we keep same distributor and referrer
↳ (they are sleep pods)
    if (prevFlowRate > 0) {
        userData = InFlowUserData({
            distributor: DistributionFeeDIP.getCurrentDistributor(I
↳ Torex(msg.sender), trader),
@@ -232,6 +238,9 @@ contract SuperBoring is UUPSProxiabile, Ownable,
↳ ITorexController, IDistributorSt
    }
}

+   // This is for being compatible with the initial staking program's pod
↳ system.
+   trader = address(InitialStakingTIP.getOrCreateSleepPod(sleepPodBeacon,
↳ trader));

```




```

+
    // Update global distribution stats, this provides the stats necessary
    ↪ for ranking distributors
    {

@@ -241,10 +250,6 @@ contract SuperBoring is UUPSProxiable, Ownable,
    ↪ ITorexController, IDistributorSt
                                prevFlowRate,
                                ↪ newFlowRate);
    }

-    // To support sleep pod, modifying these value in-place, to workaround
    ↪ solidity stack too deep issue
-    trader = address(InitialStakingTIP.getOrCreateSleepPod(sleepPodBeacon,
    ↪ trader));
-    userData.referrer =
    ↪ address(InitialStakingTIP.getOrCreateSleepPod(sleepPodBeacon,
    ↪ userData.referrer));
-
    // Updating quadratic emission weights.
    //
    // NOTE:

```

Without more test code yet, this is the fix.

pkqs90

Escalate

This should be high sev. The referrer will always lose referral bonus, thus falling in the definition of high sev issue.

Definite loss of funds without (extensive) limitations of external conditions.

sherlock-admin3

Escalate

This should be high sev. The referrer will always lose referral bonus, thus falling in the definition of high sev issue.

Definite loss of funds without (extensive) limitations of external conditions.

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.

hellwolf

Thanks for the update. But, no, it's not. Because BORING is nontransferable, and the distribution incentive program is not critical to the system.

hellwolf

Hi,

To reduce my workload, I will make the same comments for #6 (valid) and #44 #15 #54 (duplicated or similar):

- Validity: Referral program has bugs, it is confirmed per #6. There will be fix.
- Severity: Referral program uses BORING token, it is currently an incentive token, non-transferrable, and BORING mistakenly emitted so far is not what we consider crucial to the project. Further more, we can still recover them in the future, if we so desire. Hence I don't believe it is "Definite loss of funds", due to first it does not have direct value and it is recoverable.

I still think it is a worthy "medium" level, to be fair to all the reporters.

brandonshiyay

Escalate I agree with the dev on severity and root cause, but I'd like to argue that if #15, #44, and #54 is the same as mine (#6), and #14. As the dev claimed above, that when referrers are not set, there will be no loss for any users, because referral rewards are not taken from user's reward, and they are added separately. As a result, in the case of no referrer, it is true that some referral rewards will be sent to an address that is not zero address, but it does not cause any loss of fund/reward for any party who participates in the protocol. The finding is certainly valid, but as it causes no harm, I would say it's different.

Whereas #6 and #14 have explained in the situation where when referrers are set, due to the faulty logic, the rewards which are intended to send to referral's pod is not, causing referrers to lose rewards. In conclusion, I believe #15, #44, and #54 is not a duplicate of #6 and #14, and should be counted as invalid due to low severity.

sherlock-admin3

Escalate I agree with the dev on severity and root cause, but I'd like to argue that if #15, #44, and #54 is the same as mine (#6), and #14. As the dev claimed above, that when referrers are not set, there will be no loss for any users, because referral rewards are not taken from user's reward, and they are added separately. As a result, in the case of no referrer, it is true that some referral rewards will be sent to an address that is not zero address, but it does not cause any loss of fund/reward for any party who



participates in the protocol. The finding is certainly valid, but as it causes no harm, I would say it's different.

Whereas #6 and #14 have explained in the situation where when referrers are set, due to the faulty logic, the rewards which are intended to send to referral's pod is not, causing referrers to lose rewards. In conclusion, I believe #15, #44, and #54 is not a duplicate of #6 and #14, and should be counted as invalid due to low severity.

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.

hellwolf

Fair enough to consider such grouping

1. #15, #44, #54 is for address(0) referral bug,
2. #6, #14 is for updated referrer resulting inaccessible pods.

As mentioned in above comments, due to the nature of BORING and emission reward program, it is still not consider a loss of funds for the user, since BORING is created for the user. Instead, it is indeed a medium level of report since it does create faulty core logic deviating from the intention.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:

<https://github.com/superfluid-finance/averagex-contracts-cloned/commit/dfd93f13de4509018b295c8a6802a97fc4bbc07c>

WangSecurity

As I understand, this issue occurs anytime `onFlowChanged` is called, correct? and with each call and more pods created, the initial referrer receives fewer and fewer rewards, correct?

About the @brandonshiyay escalation, the same escalation about the de-duplication of #15 and #54 was already raised [here](#). Hence, I'll have to reject your escalation, since it wouldn't impact reward distribution after the escalation on #15.

pkqs90

As I understand, this issue occurs anytime `onFlowChanged` is called, correct? and with each call and more pods created, the initial referrer receives fewer and fewer rewards, correct?



For Q1, yes. For Q2, to be precise, the user would lose future \$Boring tokens the second time `onFlowChanged` is called.

My initial escalation that is should be high sev is because this will always lead to loss of \$Boring tokens sent to user's `sleepPod`. Though \$Boring can't be taken out of `sleepPods`, users can stake the \$Boring to earn fees, which I think is equivalent to losing tokens.

hellwolf

My stance remains the same: The so-called "Q1" (#6, #14) should be "medium." "Q2" (#15, #44, #54) is informational.

And, I understood that my stance on the severity is from our product perspective, which considers Referral module rather "marketing" instrument built-in. Hence, the loss are rather fixable and "small". But it is certainly arguable that should the contest also use the same judging criteria.

I certainly commend everyone's effort trying to help out and find bugs, I will leave the ultimate judgement to the judges for whom should deserve more prizes.

WangSecurity

Since this issue would occur every time the `onFlowChanged` is called, I believe high severity is indeed more appropriate cause it leads to a loss of funds after every `onFlowChanged` call.

Hence, planning to accept @pkqs90 escalation and increase the severity to high.

The decision remains the same for @brandonshiyay escalation to reject the escalation.

WangSecurity

Result: High Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- pkqs90: accepted
- brandonshiyay: rejected

sherlock-admin2

The Lead Senior Watson signed off on the fix.



Issue H-2: DistributionFeeDIP does not update previous distributor address.

Source: <https://github.com/sherlock-audit/2024-06-superboring-judging/issues/17>

Found by

pkqs90

Summary

DistributionFeeDIP does not update previous distributor address.

Vulnerability Detail

When a user updates a Torex flow and sets a `userData.distributor`, the distributor would receive some inTokens as fees. The stats for distributors are stored in `DistributionFeeDIP` and updated using `updateDistributionStats()`. These fees are later claimed in `DistributionFeeManager`.

The issue is that inside `updateDistributionStats()`, the code does NOT update the current distributor for the trader. This means the old distributor is always `address(0)`, and the new distributor's flowRate increases with each flow update.

For example:

1. User A creates a flowRate of 10 with distributor B. Distributor B now has a flowRate of 10, distributor `address(0)` has a flowRate of -10.
2. User A updates the flowRate to 20. Distributor B now has a flowRate of 30, distributor `address(0)` has a flowRate of -30.

This causes the entire distribution stats to be inaccurate.

DistributionFeeDIP.sol

```
function updateDistributionStats(ITorex torex, address trader, address
↳ distributor,
                                int96 prevFlowRate, int96 newFlowRate) internal
↳ {
    Storage storage $ = _getStorage();
    Time tnow = Time.wrap(uint32(block.timestamp));

    address prevDistributor = $.distributors[torex][trader];
    DistributionStats storage curStats =
↳ $.distributionStats[torex][prevDistributor];
```



```

DistributionStats storage totalityStats =
↳ $.distributionStats[torex][_PSEUDO_DISTRIBUTOR_FOR_TOTALITY_STATS];
    if (prevDistributor == distributor) {
        (curStats.particle, totalityStats.particle) =
↳ curStats.particle.shift_flow2b
            (totalityStats.particle, FlowRate.wrap(newFlowRate - prevFlowRate),
↳ tnow);
    } else {
        DistributionStats storage newStats =
↳ $.distributionStats[torex][distributor];
        (curStats.particle, totalityStats.particle) =
↳ curStats.particle.shift_flow2b
            (totalityStats.particle, FlowRate.wrap(-prevFlowRate), tnow);
        (newStats.particle, totalityStats.particle) =
↳ newStats.particle.shift_flow2b
            (totalityStats.particle, FlowRate.wrap(newFlowRate), tnow);
    }
    // @bug: $.distributors[torex][trader] is not updated.

    emit DistributorUpdated(torex, trader, distributor, prevDistributor);
}

```

SuperBoring.sol

```

function onInFlowChanged(address trader,
                           int96 prevFlowRate, int96 /*preFeeFlowRate*/, uint256
↳ /*last Updated*/,
                           int96 newFlowRate, uint256 /*now*/,
                           bytes calldata userDataRaw) external override
    onlyRegisteredTorex(msg.sender)
    returns (int96 newFeeFlowRate)
{
    ...
    // Note, we track trader's instead of trader's pod here
    DistributionFeeDIP.updateDistributionStats(ITorex(msg.sender),
                                              trader, userData.distributor,
                                              prevFlowRate, newFlowRate);
    ...
}

```

Impact

The distributionStats accounting would be completely incorrect, and anyone can increase distributor's flowRate to a large value. Also, the flowRate for address(0) could underflow if it reaches -int128.max, which would DoS the entire protocol.



Code Snippet

- <https://github.com/sherlock-audit/2024-06-superboring/blob/main/averagex-contracts-cloned/packages/evm-contracts/src/BoringPrograms/DistributionFeeDIP.sol#L57-L77>

Tool used

Manual Review

Recommendation

Add `$.distributors[torex][trader] = distributor` in `updateDistributionStats`.

Discussion

hellwolf

I believe this report is incorrect.

While I am in the process of adding a few more definitive test cases for the distribution fee program, until then, here is my further commented code of `updateDistributionStats` for this dispute:

```
/// Update distribution stats where a new distributor may replace the previous
↳ distributor
function updateDistributionStats(ITorex torex, address trader, address
↳ distributor,
                                int96 prevFlowRate, int96 newFlowRate) internal
                                ↳ {
    Storage storage $ = _getStorage();
    Time tnow = Time.wrap(uint32(block.timestamp));

    address prevDistributor = $.distributors[torex][trader];
    // stats for the previous distributor (current)
    DistributionStats storage curStats =
    ↳ $.distributionStats[torex][prevDistributor];
    // totality stats
    DistributionStats storage totalityStats =
    ↳ $.distributionStats[torex][_PSEUDO_DISTRIBUTOR_FOR_TOTALITY_STATS];
    if (prevDistributor == distributor) {
        // update stats of the same distributor
        (curStats.particle, totalityStats.particle) =
        ↳ curStats.particle.shift_flow2b
            (totalityStats.particle, FlowRate.wrap(newFlowRate - prevFlowRate),
            ↳ tnow);
    } else {
```



```

        // stats for the new distributor
        DistributionStats storage newStats =
        ↪ $.distributionStats[torex][distributor];
        // update stats for the previous distributor
        (curStats.particle, totalityStats.particle) =
        ↪ curStats.particle.shift_flow2b
            (totalityStats.particle, FlowRate.wrap(-prevFlowRate), tnow);
        // update stats for the new distributor
        (newStats.particle, totalityStats.particle) =
        ↪ newStats.particle.shift_flow2b
            (totalityStats.particle, FlowRate.wrap(newFlowRate), tnow);
    }

    emit DistributorUpdated(torex, trader, distributor, prevDistributor);
}

```

Hence, I think, this issue is invalid.

If such argument is not enough yet, I will have the test case soon, anyways.

pkqs90

Escalate

I'm not sure I fully understood the sponsor's reasoning. This issue is saying that `$.distributors[torex][trader]` is never updated to the current distributor, and it would always be `address(0)`.

sherlock-admin3

Escalate

I'm not sure I fully understood the sponsor's reasoning. This issue is saying that `$.distributors[torex][trader]` is never updated to the current distributor, and it would always be `address(0)`.

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.

hellwolf

Yes, I am sorry for the misunderstanding. It is indeed confirmed and quite a silly one.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:



<https://github.com/superfluid-finance/averagex-contracts-cloned/commit/1a5570224a8fa852fe149412d411124afdc3e113>

WangSecurity

Agree with the escalation, planning to accept and validate with high severity.

WangSecurity

@foufrix @pkqs90 are there any duplicates?

WangSecurity

Result: High Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- pkqs90: accepted

foufrix

@WangSecurity this one is unique

sherlock-admin2

The Lead Senior Watson signed off on the fix.



Issue H-3: `Torex` flow created with zero address referrer will lock its rewards in a `SleepPod` forever

Source:

<https://github.com/sherlock-audit/2024-06-superboring-judging/issues/44>

The protocol has acknowledged this issue.

Found by

alexzoid, pkqs90, samurii77

Summary

When flow is created with zero address referrer, the `SleepPod` is created for staker with zero address. All referrer rewards will be accumulated on this `SleepPod` and will be locked forever as the zero address cannot access or utilize these rewards.

Root Cause

In <https://github.com/sherlock-audit/2024-06-superboring/blob/main/averagex-contracts-cloned/packages/evm-contracts/src/BoringPrograms/InitialStakingTIP.sol#L40> the code does not check if the staker is the zero address before creating a `SleepPod`:

- `p = $.sleepPods[staker];`
- `if (address(p) == address(0)) { should be changed to if (address(p) == address(0) && staker != address(0)) {`

Internal pre-conditions

1. `userData.referrer` is provided as zero address when the flow is created

External pre-conditions

1. None

Attack Path

1. User creates a flow for `Torex` with zero address referrer. Then `Torex.onFlowCreated()` is executed, followed by a callback `SuperBoring.onInFlowChanged()`.



2. Here, at line <https://github.com/sherlock-audit/2024-06-superboring/blob/main/averagex-contracts-cloned/packages/evm-contracts/src/SuperBoring.sol#L245>, a SleepPod is created for the zero address:

```
userData.referrer =  
↳ address(InitialStakingTIP.getOrCreateSleepPod(sleepPodBeacon,  
↳ userData.referrer));
```

3. All current and subsequent referrer rewards for the zero address will be accumulated in this SleepPod and locked forever.

Impact

When a user creates a flow without a referrer, a SleepPod is created for the zero address staker, locking all referrer rewards forever and causing a loss of earnings for the stakers.

PoC

No response

Mitigation

```
diff --git a/averagex-contracts-cloned/packages/evm-contracts/src/BoringPrograms_ |  
↳ /InitialStakingTIP.sol  
↳ b/averagex-contracts-cloned/packages/evm-contracts/src/BoringPrograms/Initia_ |  
↳ lStakingTIP.sol  
index f127835..044bfbc 100644  
--- a/averagex-contracts-cloned/packages/evm-contracts/src/BoringPrograms/Initia_ |  
↳ lStakingTIP.sol  
+++ b/averagex-contracts-cloned/packages/evm-contracts/src/BoringPrograms/Initia_ |  
↳ lStakingTIP.sol  
@@ -37,7 +37,7 @@ library InitialStakingTIP {  
    {  
        Storage storage $ = _getStorage();  
        p = $.sleepPods[staker];  
-        if (address(p) == address(0)) {  
+        if (address(p) == address(0) && staker != address(0)) { // Don't create  
↳ a SleepPod for zero address staker  
            p = $.sleepPods[staker] = createSleepPod(sleepPodBeacon, staker);  
        }  
    }  
}
```



Discussion

foufrix

User input issue, and not zero address check are low/info

alexzoid-eth

Escalate

Hey @foufrix @hellwolf,

I have to disagree that this issue is related to invalid user input.

When initiating a new flow, an optional parameter `referrer` address can be passed. Optional means that when passing `address(0)`, no referrer should be used. Below I explain why this does not happen.

A user initiated a flow in Torex, and `SuperBoring.onInFlowChanged` is executed. Here, `referrer` is extracted from `userData.referrer`. Then, `userData.referrer` is updated with the created or extracted `SleepPod` address.

[averagex-contracts-cloned/packages/evm-contracts/src/SuperBoring.sol#L245-L246](#)

```
userData.referrer =  
↳ address(InitialStakingTIP.getOrCreateSleepPod(sleepPodBeacon,  
↳ userData.referrer));
```

I want to mention that `userData.referrer` after this step will never be zero, despite the protocol having some checks for zero address referrer later.

[/averagex-contracts-cloned/packages/evm-contracts/src/BoringPrograms/QuadraticEmissionTIP.sol#L139-L141](#)

```
// if either there is a new referrer or there was an existing referrer, then  
↳ we need an update.  
if (oldRefData.referrer != address(0) || referrer != address(0)) {
```

Imagine a user doesn't want to pass any referrer address, keeping `userData.referrer` as `address(0)`. In this case, `getOrCreateSleepPod()` will create a new `SleepPod` for a staker with a zero address, as there are no checks for zero staker addresses anywhere. [/averagex-contracts-cloned/packages/evm-contracts/src/BoringPrograms/SleepPod.sol#L72-L77](#)

```
function createSleepPod(UpgradeableBeacon sleepPodBeacon, address staker)  
↳ returns (SleepPod pod) {  
    bytes memory initCall = abi.encodeWithSelector(SleepPod.initialize.selector,  
↳ staker);  
    pod = SleepPod(address(new BeaconProxy(address(sleepPodBeacon), initCall)));
```



```
emit SleepPodCreated(staker, pod);  
}
```

This `SleepPod` will be created and stored as a valid `userData.referrer` address, which will be used for accumulating referrer rewards for a staker with a zero address. As only the staker can extract rewards from their own `SleepPod`, these tokens will be lost forever.

Basically, any user who doesn't want to use a referrer will use this dead `SleepPod`, creating a loss of 5% of potential rewards. This is why I marked this issue as a **high** severity.

sherlock-admin3

Escalate

Hey @foufrix @hellwolf,

I have to disagree that this issue is related to invalid user input.

When initiating a new flow, an optional parameter `referrer` address can be passed. Optional means that when passing `address(0)`, no referrer should be used. Below I explain why this does not happen.

A user initiated a flow in `Torex`, and `SuperBoring.onInFlowChanged` is executed. Here, `referrer` is extracted from `userData.referrer`. Then, `userData.referrer` is updated with the created or extracted `SleepPod` address.

[averagex-contracts-cloned/packages/evm-contracts/src/SuperBoring.sol#L245-L246](#)

```
userData.referrer =  
↳ address(InitialStakingTIP.getOrCreateSleepPod(sleepPodBeacon,  
↳ userData.referrer));
```

I want to mention that `userData.referrer` after this step will never be zero, despite the protocol having some checks for zero address referrer later. [/averagex-contracts-cloned/packages/evm-contracts/src/BoringPrograms/QuadraticEmissionTIP.sol#L139-L141](#)

```
// if either there is a new referrer or there was an existing referrer,  
↳ then we need an update.  
if (oldRefData.referrer != address(0) || referrer != address(0)) {
```

Imagine a user doesn't want to pass any referrer address, keeping `userData.referrer` as `address(0)`. In this case, `getOrCreateSleepPod()` will create a new `SleepPod` for a staker with a zero address, as there are



no checks for zero staker addresses anywhere.
/averagex-contracts-cloned/packages/evm-
contracts/src/BoringPrograms/SleepPod.sol#L72-L77

```
function createSleepPod(UpgradeableBeacon sleepPodBeacon, address staker)
↳ returns (SleepPod pod) {
    bytes memory initCall =
↳ abi.encodeWithSelector(SleepPod.initialize.selector, staker);
    pod = SleepPod(address(new BeaconProxy(address(sleepPodBeacon),
↳ initCall)));
    emit SleepPodCreated(staker, pod);
}
```

This SleepPod will be created and stored as a valid `userData.referrer` address, which will be used for accumulating referrer rewards for a staker with a zero address. As only the staker can extract rewards from their own SleepPod, these tokens will be lost forever.

Basically, any user who doesn't want to use a referrer will use this dead SleepPod, creating a loss of 5% of potential rewards. This is why I marked this issue as a **high** severity.

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.

alexzoid-eth

#15 and #54 are similar issues

hellwolf

Hi,

To reduce my workload, I will make the same comments for #6 (valid) and #44 #15 #54 (duplicated or similar):

- Validity: Referral program has bugs, it is confirmed per #6. There will be fix.
- Severity: Referral program uses BORING token, it is currently an incentive token, non-transferrable, and BORING mistakenly emitted so far is not what we consider crucial to the project. Further more, we can still recover them in the future, if we so desire. Hence I don't believe it is "Definite loss of funds", due to first it does not have direct value and it is recoverable.

I still think it is a worthy "medium" level, to be fair to all the reporters.

hellwolf



@foufrix dup of #15, #54, please remove

WangSecurity

@alexzoid-eth @hellwolf as I understand, in this case, the referrer rewards are not taken from anyone's balance, i.e. referrer rewards are not subtracted from user rewards, and are emitted and distributed separately, correct?

In that case, there are no loss of funds and just tokens sent to `address(0)` and this issue doesn't impact anything?

hellwolf

No impact. To tokens are *not* sent to `address(0)`, it is merely claimable by `address(0)`, but no one can actually claim for `address(0)`.

samuraii77

Tokens being claimable to `address(0)` means less rewards for legitimate users, no?

hellwolf

but no one can actually claim for `address(0)`.

I have summarized in above comment already.

WangSecurity

I have summarized in above comment already.

I think Watson above means that if the rewards weren't allocated to `address(0)` and weren't allocated to anyone if there are no referrers, then other referrers would receive more rewards. But, as I understand, it doesn't affect the rewards of the other referrers, correct? Excuse me if it's silly to confirm, but still want to make sure I understand everything correctly.

alexzoid-eth

Alright, let's clarify the situation. The tokens do not leak, but the distribution (proportions) is broken.

When a trader sets a referrer, that referrer receives an additional 5% of the trader's units in the emission treasury pool. Although rewards for the referrer are locked as the address is unacceptable (`address(0)`), the overall share for other users in the pool decreases. In other words, they receive fewer tokens.

These units function as shares, determining a member's share of the pool's distributions:

"A pool member's units determine their share of the pool's distributions."
Superfluid Documentation shares



When no referrer is set, the value of other users' shares **should** increase, allowing them to receive more tokens.

The issue worsens because not only one Torex but all connected to the emission pool receive fewer tokens:

"According to the number of shares a TOREX has in a Pool, the TOREX will receive a proportional amount of \$BORING emissions."
Superboring Documentation

hellwolf

#15 #44 #54 #8 are of the same cluster of the same or highly related issues, I have exhausted my comments on them, I will leave to the judges to split the pie.

Thanks everyone!

pkqs90

Hi, as required by the judge, I will provide an example here. Feel free to correct me if I'm wrong.

Background: there is a fixed emission rate that sends X \$BORING/second to the pool. This amount of boring tokens is distributed to the referrer's sleepPod.

Example: Say the rate is 100 \$BORING/second. If there are 10 users set referrer to R0, R1, ..., R9, each should get 10 \$BORING/second (assume all user set the same units). Say there are 90 other users that don't set referrers, we would expect R0, R1, ..., R9 still to receive 10 \$BORING/second, but they are actually receiving 1 \$BORING/second, and sleepPod(0) would receive 90 \$BORING/second.

WangSecurity

I confirmed with the sponsor and indeed @pkqs90 is correct above. Hence, I'm planning to accept the escalation and validate it with high severity, since there are no extensive limitations, only the users have to **not** set a referrer and the loss of funds can be even bigger than 90% in the above example. Duplicates are #15, #54 and #8.

hellwolf

Hi,

I respect the judge's decision.

Though, I will raise a meta issue: to me, there is a difference between loss of opportunity vs loss of funds one had. This is a case of "loss of perceived opportunity".

I think the criteria could clarify those differences a bit more.

WangSecurity



Result: High Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- alexzoid-eth: accepted



Issue H-4: A malicious attacker can manipulate distribution stats by setting `userData.distributor` as `address(1)`

Source:

<https://github.com/sherlock-audit/2024-06-superboring-judging/issues/46>

Found by

KupiaSec, y4y

Summary

In the `SuperBoring.onInFlowChanged` function, `userDataRaw` contains the information of distributor. If a user sets this as `address(1)` that represents a pseudo distributor for totality stats, totality stats are updated incorrectly. As a result, member units of distributors are updated incorrectly and they get incorrect fee.

Vulnerability Detail

In the `SuperBoring.onInFlowChanged` function, `userDataRaw` contains the information of distributor. If user provide the info for distributor, `userData.distributor` will be an address of distributor provided by the user.

```
File: averagex-contracts-cloned\packages\evm-contracts\src\SuperBoring.sol
219:         if (userDataRaw.length > 0) {
220:             // This may revert, and it will make create/update flow
↳ fail.
221:     @>         userData = abi.decode(userDataRaw, (InFlowUserData));
222:             if (trader == userData.referrer) revert NO_SELF_REFERRAL();

238:         DistributionFeeDIP.updateDistributionStats(ITorex(msg.sender),
239:     @>                                     trader,
↳ userData.distributor,
240:                                     prevFlowRate,
↳ newFlowRate);
```

Then, in the `DistributionFeeDIP.updateDistributionStats` function, it will update global distribution stats. If distributor is `address(1)`, `newStats` and `totalityStats` will point the same storage variable from L69 if `prevDistributor` is not `address(1)`.

```
File: averagex-contracts-cloned\packages\evm-contracts\src\BoringPrograms\DistributionFeeDIP.sol
↳
```



```

62:         address prevDistributor = $.distributors[torex][trader];
63:         DistributionStats storage curStats =
↳ $.distributionStats[torex][prevDistributor];
64:         DistributionStats storage totalityStats =
↳ $.distributionStats[torex][_PSEUDO_DISTRIBUTOR_FOR_TOTALITY_STATS];
65:         if (prevDistributor == distributor) {
66:             (curStats.particle, totalityStats.particle) =
↳ curStats.particle.shift_flow2b
67:             (totalityStats.particle, FlowRate.wrap(newFlowRate -
↳ prevFlowRate), tnow);
68:         } else {
69:             DistributionStats storage newStats =
↳ $.distributionStats[torex][distributor];
70:             (curStats.particle, totalityStats.particle) =
↳ curStats.particle.shift_flow2b
71:             (totalityStats.particle, FlowRate.wrap(-prevFlowRate), tnow);
72:             @> (newStats.particle, totalityStats.particle) =
↳ newStats.particle.shift_flow2b
73:             (totalityStats.particle, FlowRate.wrap(newFlowRate), tnow);
74:         }

```

Then, \$.distributionStats[torex][_PSEUDO_DISTRIBUTOR_FOR_TOTALITY_STATS] that contains totality stats will be updated with incorrect value. If totality stats are updated with wrong value, the getTotalityStats function will return wrong value and the sync function will work incorrectly due to wrong information of totality stats from L85.

```

File: averagex-contracts-cloned\packages\evm-contracts\src\BoringPrograms\Distri
↳ butionFeeManager.sol
85: @> (int256 dvol,) = p.getDistributorStats(torex, distributor);
86: (int256 tvol,) = p.getTotalityStats(torex);
87: if (tvoll > 0) {
88: @> units = uint128(SafeCast.toUint256(dvol * INT_100PCT_PM /
↳ tvoll));
89: }
90: }
91: pool.updateMemberUnits(distributor, units);

```

As a result, distributors will get wrong distribution fee.

Impact

Distributors will get wrong distribution fee. Furthermore, a malicious distributor can get all distribution fee by manipulating totality stats.



Code Snippet

<https://github.com/sherlock-audit/2024-06-superboring-KupiaSecAdmin/tree/main/averagex-contracts-cloned/packages/evm-contracts/src/SuperBoring.sol#L219-223> <https://github.com/sherlock-audit/2024-06-superboring-KupiaSecAdmin/tree/main/averagex-contracts-cloned/packages/evm-contracts/src/SuperBoring.sol#L238-240> <https://github.com/sherlock-audit/2024-06-superboring-KupiaSecAdmin/tree/main/averagex-contracts-cloned/packages/evm-contracts/src/BoringPrograms/DistributionFeeDIP.sol#L57-74>

Tool used

Manual Review

Recommendation

In the `DistributionFeeDIP.updateDistributionStats` function, if distributor is `address(1)`, it should be reverted.

```
File: averagex-contracts-cloned\packages\evm-contracts\src\BoringPrograms\DistributionFeeDIP.sol
57:     function updateDistributionStats(ITorex torex, address trader, address
    distributor,
58:                                     int96 prevFlowRate, int96 newFlowRate)
    internal {
+       require(distributor != _PSEUDO_DISTRIBUTOR_FOR_TOTALITY_STATS,
    'Invalid Distributor');
59:       Storage storage $ = _getStorage();
60:       Time tnow = Time.wrap(uint32(block.timestamp));
```

Discussion

hellwolf

It is a confirmed issue.

But I would like to understand if its "high" severity designation is appropriate. I will write up a case to clarify my understanding of the (limited or no) impact.

foufrix

@hellwolf To identify high issue : <https://docs.sherlock.xyz/audits/judging/judging#iv.-how-to-identify-a-high-issue> a finite amount of loss of funds + not a lot of external conditions setting a distributor to `address(1)` fit in not a lot of external conditions as anyone can do it easily

sherlock-admin2



The protocol team fixed this issue in the following PRs/commits:
<https://github.com/superfluid-finance/averagex-contracts-cloned/commit/79f6062e1343049a29562f3d7f3f39b87b7a1c3d>

alexzoid-eth

Escalate

The issue should be info/low since this actually has no impact but only describes an unintended behavior.

The report points out that the totally stats can be updated incorrectly. But in fact the correct value will be returned from the `shift_flow2b` function because the function only adjusts the received values and returns them back. In this case the function returns `(newStats.particle, totalityStats.particle)` values which will be equal so the totally states updates two times with the same new value. This way the report statement about the impact is wrong.

```
import "@superfluid-finance/solidity-semantic-money/src/SemanticMoney.sol";

struct DistributionStats {
    /// Using semantic money basic particle to track money stats
    BasicParticle particle;
}

contract TestShiftFlow2B is TorexTest {
    using SemanticMoney for BasicParticle;

    DistributionStats public sharedParticle;

    function testShiftFlow2B() public {

        sharedParticle.particle = BasicParticle({
            _settled_at: Time.wrap(uint32(1620000000)),
            _flow_rate: FlowRate.wrap(100),
            _settled_value: Value.wrap(1000)
        });

        int96 newFlowRate = 150;
        int96 prevFlowRate = 100;
        Time currentTime = Time.wrap(uint32(1620001000));

        DistributionStats storage curStats = sharedParticle;
        DistributionStats storage totalityStats = sharedParticle;

        (curStats.particle, totalityStats.particle) =
        ↪ curStats.particle.shift_flow2b(totalityStats.particle,
        ↪ FlowRate.wrap(newFlowRate - prevFlowRate), currentTime);
```



```

        console2.log(" _settled_at", Time.unwrap(curStats.particle._settled_at));
        console2.log(" _flow_rate",
↳ FlowRate.unwrap(curStats.particle._flow_rate));
        console2.log(" _settled_value",
↳ Value.unwrap(curStats.particle._settled_value));
        console2.log("=====");

        console2.log(" _settled_at",
↳ Time.unwrap(totalityStats.particle._settled_at));
        console2.log(" _flow_rate",
↳ FlowRate.unwrap(totalityStats.particle._flow_rate));
        console2.log(" _settled_value",
↳ Value.unwrap(totalityStats.particle._settled_value));
        console2.log("=====");
    }
}

```

Ran 1 test for test/SuperBoring.t.sol:TestShiftFlow2B

[PASS] testShiftFlow2B() (gas: 65917)

Logs:

```

    _settled_at 1620001000
    _flow_rate 50
    _settled_value 101000
    =====
    _settled_at 1620001000
    _flow_rate 50
    _settled_value 101000
    =====

```

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 10.60ms (202.22s
↳ CPU time)

sherlock-admin3

Escalate

The issue should be info/low since this actually has no impact but only describes an unintended behavior.

The report points out that the totally stats can be updated incorrectly. But in fact the correct value will be returned from the `shift_flow2b` function because the function only adjusts the received values and returns them back. In this case the function returns (`newStats.particle`, `totalityStats.particle`) values which will be equal so the totally states updates two times with the same new value. This way the report statement about the impact is wrong.



```

import "@superfluid-finance/solidity-semantic-money/src/SemanticMoney.sol";

struct DistributionStats {
    /// Using semantic money basic particle to track money stats
    BasicParticle particle;
}

contract TestShiftFlow2B is TorexTest {
    using SemanticMoney for BasicParticle;

    DistributionStats public sharedParticle;

    function testShiftFlow2B() public {

        sharedParticle.particle = BasicParticle({
            _settled_at: Time.wrap(uint32(1620000000)),
            _flow_rate: FlowRate.wrap(100),
            _settled_value: Value.wrap(1000)
        });

        int96 newFlowRate = 150;
        int96 prevFlowRate = 100;
        Time currentTime = Time.wrap(uint32(1620001000));

        DistributionStats storage curStats = sharedParticle;
        DistributionStats storage totalityStats = sharedParticle;

        (curStats.particle, totalityStats.particle) =
↳ curStats.particle.shift_flow2b(totalityStats.particle,
↳ FlowRate.wrap(newFlowRate - prevFlowRate), currentTime);

        console2.log(" _settled_at",
↳ Time.unwrap(curStats.particle._settled_at));
        console2.log(" _flow_rate",
↳ FlowRate.unwrap(curStats.particle._flow_rate));
        console2.log(" _settled_value",
↳ Value.unwrap(curStats.particle._settled_value));
        console2.log("=====");

        console2.log(" _settled_at",
↳ Time.unwrap(totalityStats.particle._settled_at));
        console2.log(" _flow_rate",
↳ FlowRate.unwrap(totalityStats.particle._flow_rate));
        console2.log(" _settled_value",
↳ Value.unwrap(totalityStats.particle._settled_value));

```



```

        console2.log("=====");
    }
}

```

```
Ran 1 test for test/SuperBoring.t.sol:TestShiftFlow2B
```

```
[PASS] testShiftFlow2B() (gas: 65917)
```

```
Logs:
```

```
  _settled_at 1620001000
```

```
  _flow_rate 50
```

```
  _settled_value 101000
```

```
=====
```

```
  _settled_at 1620001000
```

```
  _flow_rate 50
```

```
  _settled_value 101000
```

```
=====
```

```
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 10.60ms
```

```
↳ (202.22s CPU time)
```

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.

hellwolf

Hi, thanks again for your input @alexzoid-eth!

In my assessment, it is a medium impact due to the following reasonin:

1. anyone may call `DistributionFeeManager.sync(address(1))`.
2. but no one can impersonate `address(1)`.
3. the fees from trading flows that is tagged `address(1)` as distributors will have the fees forever lost.

However, it is not high, because to make (3) widely spread, you must be able to create an application that many people use that uses `address(1)` as distributor. Selling a product is not an trivial task!

Hence, I rest my case.

KupiaSecAdmin

@alexzoid-eth @hellwolf @alexzoid-eth's PoC is right. However, even in the test you provided, `totalityStats` is updated with wrong value. After operation, `totalityStats._flow_rate` should 150, not 50.



And you wrote a test for the case(DistributionFeeDIP.sol#L66-L67) of `prevDistributor == distributor`. But The part I mentioned in the report is the case of (DistributionFeeDIP.sol#L69-73) where `prevDistributor != distributor`. I wrote PoC for that case.

```
pragma solidity 0.8.23;

import { console2 } from "forge-std/Test.sol";
import { TorexTest } from "../TestCommon.sol";

import "@superfluid-finance/solidity-semantic-money/src/SemanticMoney.sol";

struct DistributionStats {
    /// Using semantic money basic particle to track money stats
    BasicParticle particle;
}

contract TestShiftFlow2B is TorexTest {
    using SemanticMoney for BasicParticle;

    DistributionStats public PrevDistributorStats;
    DistributionStats public TotalityStats;

    function testPrint() public {

    }

    function testShiftFlow2B() public {

        PrevDistributorStats.particle = BasicParticle({
            _settled_at: Time.wrap(uint32(1620000000)),
            _flow_rate: FlowRate.wrap(100),
            _settled_value: Value.wrap(1000)
        });

        TotalityStats.particle = BasicParticle({
            _settled_at: Time.wrap(uint32(1620000000)),
            _flow_rate: FlowRate.wrap(200),
            _settled_value: Value.wrap(2000)
        });

        int96 newFlowRate = 150;
        int96 prevFlowRate = 100;
        Time tnow = Time.wrap(uint32(1620001000));

        DistributionStats storage curStats = PrevDistributorStats;
        DistributionStats storage totalityStats = TotalityStats;
```



```

        DistributionStats storage newStats = TotalityStats;

        (curStats.particle, totalityStats.particle) =
↳ curStats.particle.shift_flow2b
            (totalityStats.particle, FlowRate.wrap(-prevFlowRate), tnow);

        console2.log(" curStats : after first operation");
        console2.log(" _settled_at : ",
↳ Time.unwrap(curStats.particle._settled_at));
        console2.log(" _flow_rate : ",
↳ FlowRate.unwrap(curStats.particle._flow_rate));
        console2.log(" _settled_value : ",
↳ Value.unwrap(curStats.particle._settled_value));
        console2.log("=====");

        console2.log(" totalityStats : after first operation");
        console2.log(" _settled_at : ",
↳ Time.unwrap(totalityStats.particle._settled_at));
        console2.log(" _flow_rate : ",
↳ FlowRate.unwrap(totalityStats.particle._flow_rate));
        console2.log(" _settled_value : ",
↳ Value.unwrap(totalityStats.particle._settled_value));
        console2.log("=====");

        (newStats.particle, totalityStats.particle) =
↳ newStats.particle.shift_flow2b
            (totalityStats.particle, FlowRate.wrap(newFlowRate), tnow);

        console2.log(" totalityStats : after second operation");
        console2.log(" _settled_at : ",
↳ Time.unwrap(totalityStats.particle._settled_at));
        console2.log(" _flow_rate : ",
↳ FlowRate.unwrap(totalityStats.particle._flow_rate));
        console2.log(" _settled_value : ",
↳ Value.unwrap(totalityStats.particle._settled_value));
        console2.log("=====");
    }
}

```

Outputs:

```

Ran 1 test for test/Distributor.t.sol:TestShiftFlow2B
[PASS] testShiftFlow2B() (gas: 128670)
Logs:
    curStats : after first operation
    _settled_at : 1620001000

```



```

    _flow_rate : 200
    _settled_value : 101000
    =====
    totalityStats : after first operation
    _settled_at : 1620001000
    _flow_rate : 100
    _settled_value : 202000
    =====
    totalityStats : after second operation
    _settled_at : 1620001000
    _flow_rate : -50
    _settled_value : 202000
    =====

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 10.65ms (369.80s
↳ CPU time)

Ran 1 test suite in 11.84ms (10.65ms CPU time): 1 tests passed, 0 failed, 0
↳ skipped (1 total tests)

```

As shown above, `totalityStats._flow_rate` should be 100, not -50. Here, totality flow rate becomes negative instead of positive.

Impact of this issue: - Distributors will get wrong distribution fee. - Furthermore, a malicious distributor can get all distribution fee by manipulating totality stats. - Moreover, when the flow rate is negative in some special cases, it can deplete the balance of protocol.

@hellwolf I have an additional question. Who sets `userData.distributor` - a distributor or a user?

WangSecurity

@hellwolf can you assist us on whether `userData.distributor` is set by the user or a distributor?

hellwolf

Distributor, we don't expect user can set it at scale for more people.

KupiaSecAdmin

@WangSecurity @hellwolf In the report, I wrote as a malicious distributor can get all distribution fee by manipulating totality stats. As you said, since distributors can set `userData.distributor` as any value, setting `userData.distributor` to `address(1)` fits in not a lot of external conditions. So I think this issue should be High according to Sherlock rule(<https://docs.sherlock.xyz/audits/judging/judging#iv.-how-to-identify-a-high-issue>).



hellwolf

Okay, I will leave it to the judge to decide.

I do concede that the entire DistributionFeeManager module is not reliable, but it has been fixed. This does make judging anything around it a murky decision.

Thanks!

pkqs90

Hey @KupiaSecAdmin. First of all, I think this is a very nice finding!

But you state that malicious users can use this bug to steal all distribution fee. Can you show a full poc on how to do that? I tried to reproduce the poc but didn't find a way. IMO what the malicious user can do using this bug is manipulate the totalityStats - but how would he benefit from it?

Correct me if I'm wrong, if the totalityStats flowRate is X, the malicious user would need to create a flow that is X and the distributor to address(1) if he wants the totalityStats to be 0. This would be pretty hard considering X is the sum of flowRate of all other honest users. Also, even if he did make totalityStats to near 0 or negative, any other user can also benefit from it since DistributionFeeManager.sync can be called by anyone.

WangSecurity

Furthermore, a malicious distributor can get all distribution fee by manipulating totality stats

As I understand the confusion comes from this line, correct @pkqs90? But I think the problem here is that this would lead to loss of rewards to other users, rather than stealing the reward and getting the tokens, correct @KupiaSecAdmin?

KupiaSecAdmin

@WangSecurity In reality, it will be difficult to steal all distribution fee by manipulating totality stats. Here, the problem is distributors can manipulate totality stats and update their units as incorrect values easily, which makes fee distribution work improperly.

pkqs90

I see, so the attack vector is:

1. Attacker manipulates totalityStats to near 0
2. Attacker call DistributionFeeManager.sync to update member units of some distributor feeDistributionPool to very large
3. Attacker restores totalityStats



No questions from me then. Nice finding!

WangSecurity

In that case, I agree that high severity is appropriate here, it leads to a loss of fees for other distributors (definite loss of funds) and distributors are the ones that could set distributor address to address(1), which I believe is not an extensive limitation.

Planning to reject the escalation and leave the issue as it is.

WangSecurity

Result: High Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- alexzoid-eth: rejected

sherlock-admin2

The Lead Senior Watson signed off on the fix.



Issue M-1: UniswapV3PoolTwapObserver uses the slot0 price when duration is zero, which is manipulable. Attackers can exploit this to buy tokens nearly for free.

Source: <https://github.com/sherlock-audit/2024-06-superboring-judging/issues/21>

Found by

pkqs90

Summary

UniswapV3PoolTwapObserver uses the slot0 price when duration is zero, which is manipulable. Attackers can exploit this to buy tokens nearly for free.

Vulnerability Detail

Background

First, we need to understand how moving liquidity works. Each time a user calls move liquidity, all the inTokens up to the current timestamp need to be swapped. During the move liquidity process, a new checkpoint is created in the UniswapV3PoolTwapObserver to update the timestamp.

The core issue here is that if the duration is zero for the current liquidity movement from the last liquidity movement, the UniswapV3PoolTwapObserver defaults to using the spot price in slot0, which is manipulable.

At first glance, it might seem that if the duration is zero, all inTokens would have already been swapped in the last liquidity movement, leaving no tokens to move for this liquidity movement. However, there is a way to get some tokens.

Notice that the `availableInTokens()` function calculates the amount of inTokens left for this round of liquidity movement by `_inToken.balanceOf(address(this)) - minimumDeposit`. There is a concept of `minimumDeposit`, calculated by `uint256 minimumDeposit = _inToken.getBufferAmountByFlowRate(_requestedFeeDistFlowRate);`. This is the amount of tokens that needs to be in Torex to maintain the outflow for the `feeDistributionPool`. If a user decreases their `flowRate`, the `minimumDeposit` decreases, and an attacker could perform a sandwich attack to buy these tokens at a very cheap price.

The deposit amount can be found in [ConstantFlowAgreementV1.sol](#). It is basically the `flowRate` multiplied by the `liquidationPeriod` (which is 4 hours), so a `flowRate` of `1e18` would mean a deposit amount of `14400e18`.



Attack vector

To sum it up, the attacker would monitor the mempool and find a user with flowRate X closing their flow. The attack vector is:

Before the transaction:

1. The attacker calls `moveLiquidity` for the first time, moving the remaining `inTokens` in Torex.
2. The attacker manipulates the Uniswap slot0 spot price by swapping a large number of tokens.

Transaction is executed, and the user closes their flow. Now, there are 14400X more available tokens in Torex.

After the transaction:

1. The attacker calls `moveLiquidity` again and buys 14400X tokens at a very cheap price, since it uses the Uniswap slot0 spot price as the oracle.
2. The attacker restores the Uniswap spot price.

Additionally, anyone can use this method to buy back their backcharged tokens that were spent as a deposit for the fee flow at a cheap price. In that case, everything can be done in a single transaction. According to the [code comments](#), the backcharged fees are not refunded if the flowRate decreases: `// however, in case of the flow rate going down, there is no refund for back charged fees..` However, using this attack method, users can get their backcharged fees back.

UniswapV3PoolTwapObserver.sol

```
function getTwapSinceLastCheckpoint(uint256 time, uint256 inAmount) public
↳ override view
    returns (uint256 outAmount, uint256 duration)
{
>    duration = getDurationSinceLastCheckpoint(time);

    int24 tick;
    if (duration > 0) {
        // calculating tick of the TWAP
        int56 currentTickCumulative = _getCurrentTickCumulative();
        tick = SafeCast.toInt24((int256(currentTickCumulative) -
↳ int256(_lastTickCumulative))
                                / SafeCast.toInt256(duration));
    } else {
        // special case: when duration is zero, returning the current tick
↳ directly
>    ((*sqrtPriceX96*/, tick, /*obsIdx*/, /*obsCrd*/, /*obsCrdNext*/, /*feeP*/,
↳ /*unlckd*/) = uniPool.slot0());
```



```

    }

    // Note:
    // 1. OracleLibrary.getQuoteAtTick(int24 tick, uint128 baseAmount,
↳ address baseToken, address quoteToken)
    // returns "quoteAmount Amount of quoteToken received for
↳ baseAmount of baseToken."
    // 1.1 TickMath.getSqrtRatioAtTick(int24 tick) returns (uint160
↳ sqrtPriceX96)
    if (inverseOrder == false) {
        outAmount = OracleLibrary.getQuoteAtTick(tick,
↳ SafeCast.toUint128(inAmount),
                                                    uniPool.token0(),
↳ uniPool.token1());
    } else {
        outAmount = OracleLibrary.getQuoteAtTick(tick,
↳ SafeCast.toUint128(inAmount),
                                                    uniPool.token1(),
↳ uniPool.token0());
    }
}
}

```

TorexCore.sol

```

function _moveLiquidity(bytes memory moverData) internal
    nonReentrant
    returns (LiquidityMoveResult memory result)
{
    (result.inAmount, result.minOutAmount, result.durationSinceLastLME,
↳ result.twapSinceLastLME) =
        getLiquidityEstimations();

    // Step 1: Transfer the inAmount of inToken liquidity to the liquidity
↳ mover.
    _inToken.transfer(msg.sender, result.inAmount);

    // Step 2: Ask liquidity mover to provide outAmount of outToken
↳ liquidity in exchange.
    assert((ILiquidityMover(msg.sender))
        .moveLiquidityCallback(_inToken, _outToken, result.inAmount,
↳ result.minOutAmount, moverData));
    // We distribute everything the contract has got from liquidity mover.
    result.outAmount = _outToken.balanceOf(address(this));
    if (result.outAmount < result.minOutAmount) revert
↳ LIQUIDITY_MOVER_SENT_INSUFFICIENT_OUT_TOKENS();

    // Step 3: Create new torex observer check point

```




```

>         _observer.createCheckpoint(block.timestamp);

        // Step 4: Distribute all outToken liquidity to degens.
        // NOTE: This could be more than outAmount.
        result.actualOutAmount = _outToken
            .estimateDistributionActualAmount(address(this),
↳         _outTokenDistributionPool, result.outAmount);
        _outToken.distributeToPool(address(this), _outTokenDistributionPool,
↳         result.actualOutAmount);
    }

    function getLiquidityEstimations() public view
        returns (uint256 inAmount, uint256 minOutAmount, uint256
↳         durationSinceLastLME, uint256 twapSinceLastLME)
    {
>         inAmount = availableInTokens();
        (minOutAmount, durationSinceLastLME, twapSinceLastLME) =
↳         getBenchmarkQuote(inAmount);
    }

```

Torex.sol

```

    function availableInTokens() override internal view returns (uint256 amount)
↳     {
>         uint256 minimumDeposit =
↳         _inToken.getBufferAmountByFlowRate(_requestedFeeDistFlowRate);
        amount = _inToken.balanceOf(address(this));
        if (amount >= minimumDeposit) return amount - minimumDeposit; else
↳         return 0;
    }

```

Impact

1. For a normal user, anyone can use this method to buy back their backcharged tokens that were spent as a deposit for the fee flow at a cheap price.
2. For an attacker, they can perform a sandwich attack to buy tokens at a very cheap price.

Code Snippet

- <https://github.com/sherlock-audit/2024-06-superboring/blob/main/averagex-contracts-cloned/packages/evm-contracts/src/UniswapV3PoolTwapObserver.sol#L78>



Tool used

Manual Review

Recommendation

Always use the TWAP price in `UniswapV3PoolTwapObserver` to avoid price manipulation. Consider using a fixed window (e.g. 30 minutes).

Discussion

pkqs90

Escalate

This issue is slightly different from #52. This issue did consider the case that the `inToken` amount would be 0 between checkpoints (which is the reason sponsor used to dispute #52). However, due to the existence of deposit buffer for outflowing fees, attackers can still buy these token for free.

Considering the code comments `// however, in case of the flow rate going down, there is no refund for back charged fees..` Using this method, users can get their backcharged fees back, which should be unexpected.

One discrepancy is that the original issue didn't calculate the fee percentage (which in UT is 1%), so it should be 144X instead of 14400X, assuming X is the flow rate.

sherlock-admin3

Escalate

This issue is slightly different from #52. This issue did consider the case that the `inToken` amount would be 0 between checkpoints (which is the reason sponsor used to dispute #52). However, due to the existence of deposit buffer for outflowing fees, attackers can still buy these token for free.

Considering the code comments `// however, in case of the flow rate going down, there is no refund for back charged fees..` Using this method, users can get their backcharged fees back, which should be unexpected.

One discrepancy is that the original issue didn't calculate the fee percentage (which in UT is 1%), so it should be 144X instead of 14400X, assuming X is the flow rate.

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.

hellwolf

However, due to the existence of deposit buffer for outflowing fees, attackers can still buy these token for free.

Can you elaborate, or even better, have a PoC?

pkqs90

@hellwolf Please see the PoC. There are 6 steps:

1. Grant Alice a lot of inToken/outToken to initialize testing.
2. Set outToken/inToken price to 1.
3. Alice sets inflow to 1e18.
4. Advance 1000 seconds.
5. Random Liquidity Mover comes and performs moveLiquidity. There are no more available inTokens left.
6. Alice performs the attack: A. Alice deletes her inflow; B. Alice manipulates the uniswap price to 0; C. Alice triggers moveLiquidity.

From the log output for step 6 Alice's liquidity move: inAmount, outAmount = 288000000003619540352 45000154000, we can see that:

1. Alice bought 288000000003619540352 (288e18) inTokens nearly for free. The outAmount 45000154000 is not the amount Alice paid for, but what was left during the previous moveLiquidity. You can see we've set Alice outToken allowance to zero before she conducted moveLiquidity.
2. How is 288000000003619540352 (288e18) calculated? First of all, the deposit buffer is 4 hours * flowRate, which is 14400e18. We set the fee percentage to 1%, so the deposit buffer for fee is 144e18. The reason it's 288e18 is because Alice deposited an extra 144e18 as backAdjustment when she opened her flow.

```
function testGao() external {
  // Step 1: Grant Alice a lot of inToken/outToken to initialize testing.
  {
    vm.startPrank(ADMIN);
    _inToken.transfer(alice, 1e40 ether);
    _outToken.transfer(alice, 1e40 ether);
    vm.stopPrank();
  }

  // Step 2: Set outToken/inToken price to 1.
```



```

    _setPriceScaler(1);

    // Step 3: Alice sets inflow to 1e18.
    _doChangeFlow(alice, 1e18);

    // Step 4: Advance 1000 seconds.
    _doAdvanceTime(1000);

    // Step 5: Random Liquidity Mover comes and performs moveLiquidity. There
    ↪ are no more inTokens left.
    {
        vm.startPrank(LM);
        _outToken.approve(address(_mockLiquidityMover), type(uint256).max);
        (uint256 inAmount, uint256 outAmount,) =
    ↪ _mockLiquidityMover.triggerLiquidityMovement(_torex);
        console2.log("LM's liquidity move: inAmount, outAmount =", inAmount,
    ↪ outAmount);
        // LM's liquidity move: inAmount, outAmount = 846000000000000154000
    ↪ 846000000000000154000
        vm.stopPrank();
    }

    // Step 6: Alice performs the attack:
    //   A. Alice deletes her inflow.
    //   B. Alice manipulates the uniswap price to 0.
    //   C. Alice triggers moveLiquidity.
    _doChangeFlow(alice, 0);
    _setPriceScaler(0);
    {
        vm.startPrank(alice);
        _outToken.approve(address(_mockLiquidityMover), 0); // Set allowance to
    ↪ 0 to make sure Alice does not pay for anything.
        (uint256 inAmount, uint256 outAmount,) =
    ↪ _mockLiquidityMover.triggerLiquidityMovement(_torex);
        console2.log("Alice's liquidity move: inAmount, outAmount =", inAmount,
    ↪ outAmount);
        // Alice's liquidity move: inAmount, outAmount = 288000000003619540352
    ↪ 45000154000
        vm.stopPrank();
    }
}

```

hellwolf

Hi, there are a couple of issues with your PoC:

1. First of all, once Alice change flow to 0 (delete flow), Alice no longer has units



in out token distribution pool

2. Due to (1), you should also check `actualOutAmount` returned by `triggerLiquidityMovement`:

```
function triggerLiquidityMovement(ITorex _torex) external  
    returns (uint256 inAmount, uint256 outAmount, uint256 actualOutAmount)
```

You should notice that `actualOutAmount` will be 0:

```
(uint256 inAmount, uint256 outAmount, uint256 actualOutAmount) =  
    ↪ _mockLiquidityMover.triggerLiquidityMovement(_torex);  
console2.log("Alice's liquidity move: inAmount, outAmount, actualOutAmount  
    ↪ =", inAmount, outAmount, actualOutAmount);
```

->

```
Alice's liquidity move: inAmount, outAmount, actualOutAmount =  
    ↪ 2880000000003619684352 450000000000 0
```

3. Even more damningly, manipulating uniswap slot0 price to zero is no where near a trivial thing in any liquid uniswap pool in the production environment. Note that the PoC uses mock contract, not the production pool.

All in all, I am still not convinced that the watson can produce such an attack in a production environment effectively, let alone a PoC in a test environment.

pkqs90

Hey @hellwolf, please see my reply below:

For point 1 - Alice changes her flow to 0 is exactly the reason there are >0 available inTokens in the Torex. Because Alice deleted her inflow, the fee distribution flow from Torex->feeDistributionPool would decrease, thus the required deposit buffer for it would decrease. This is the amount of tokens that is available (which is the 288e18 in the above PoC) for the next liquidity move.

For point 2 - the `actualOutAmount` is what is distributed to the inflow users, but since Alice removed her flow, this is zero. But this value doesn't mean much. The value we should be looking at is the amount of outTokens Alice moved to the Torex to buy the 288e18 inTokens, which is zero (proved in PoC. Alice set the `outputToken` allowance to zero before liquidity move)

For point 3 - I agree manipulating it to 0 is impossible, but by using a flashLoan it is possible to manipulate it to very close to zero, especially for smaller liquidity pools.

The result of the attack is the user can take back 2 times the backadjustment fees that user deposited for fee distribution buffer nearly for free. I agree this may not



be much, but it is also non-trivial if the inflow rate is high. As for the mediation, I would suggest simply remove the case where duration == 0 case and revert.

WangSecurity

@hellwolf could you share your opinion on the above comment?

hellwolf

For point 1 - Alice changes her flow to 0 is exactly the reason there are >0 available inTokens in the Torex. Because Alice deleted her inflow, the fee distribution flow from Torex->feeDistributionPool would decrease, thus the required deposit buffer for it would decrease. This is the amount of tokens that is available (which is the 288e18 in the above PoC) for the next liquidity move.

Yes, by design, Alice loses that part of the in-token as fee to pay for the buffer of fee flow. (Sorry, tongue twister!)

For point 2 - the actualOutAmount is what is distributed to the inflow users, but since Alice removed her flow, this is zero. But this value doesn't mean much. The value we should be looking at is the amount of outTokens Alice moved to the Torex to buy the 288e18 inTokens, which is zero (proved in PoC. Alice set the outputToken allowance to zero before liquidity move)

To put under the context too,

1e18 flow rate is 2.6M / month:

```
>>> 1e18 * 3600 * 24 * 30.41 / 1e18  
2627424.0
```

That is the context for the amount in-tokens Alice pays (loses): 288 for 2.6M / month flow.

For point 3 - I agree manipulating it to 0 is impossible, but by using a flashLoan it is possible to manipulate it to very close to zero, especially for smaller liquidity pools.

So, your thesis is that, within the same block:

1. Alice, the attacker, creates a large flow to TOREX,
2. Alice do a liquidity movement with regular profit incentive,
3. Alice deletes the flow to TREX,
4. Alice manipulates slot0 that TOREXes uses for the same LM, with a flash loan.



5. Alice recoupe the remaining fees he contributed in TOREX, and exchange for out-tokens with a cheaper price. But remember **Alice must find out-tokens in this transaction himself!**

6. Alice repays the flash loan.

So, at this point, you might as well ask, if Alice could manipulate slot0 and find those out-tokens to give to himself, what is Alice gaining here?

So, I repeat that I don't see a clear economic-viable attack path.

As for the mediation, I would suggest simply remove the case where `duration == 0` case and revert.

But I do agree with this, that we will simply disallow the possibility in the next version of TOREX to do the LM in the same block, so slot0 question will no longer in play.

pkqs90

I want to clarify a bit on the steps:

1. Alice, initially creates a flow to Torex, and she uses the Torex as a normal user.
2. When Alice wants to remove the flow, by design, she should not receive the deposit buffer fee she initially put into Torex.
3. However, Alice can wait for another liquidity mover and backrun his transaction with the above steps (A. remove flow, B. flashloan + manipulate slot0, C. buy back tokens for nearly free, D. recover slot0 + repay flashloan)
4. After performing step 3, Alice removes her flow and also successfully buys back the fees (twice the amount) she deposited in step 2 for a cheap price.

I agree the amount of tokens may be small when compared to the total inflow (e.g. for 1 month, $288 / 2.6M \sim 0.01\%$). But what if Alice wants to delete the flow in a shorter time period? (For example she opens for 1 hour and immediately regrets it and wants to remove it). This is pretty subjective and I will respect the judge's judgement. Just want to make sure the attack steps are clear and will leave the rest to the judge.

hellwolf

Alice removes her flow and also successfully buys back the fees (twice the amount) she deposited in step 2 for a cheap price.

Hmm, I still fail to grasp the economics, could you help me here with exact tokens follow with hypothetical numbers:

1. Say Alice pays \$100 in ETH.
2. Someone did the first LM.



3. (attack hypothesis) After the flash loan, slot0 manipulation, Alice coordinates the second LM which sells that \$100 in ETH for \$1000 in USDC (actually any extraordinary amount of USDC you like, to b make the case of "buying back tokens nearly free"), but who is paying that USDC?

pkqs90

1. Alice pays 0.1 ETH as fee, ETH/USDC is originally 3000, so that is 300 USDC.
2. Someone did the first LM.
3. Alice manipulates ETH/USDC price to 1 (hypothetical number). Then Alice can pay 0.1 USDC to get the 0.1 ETH. Alice is paying the USDC, but at a cheap price, to buy back the ETH that was deposited in step 1.

hellwolf

1. Alice pays 0.1 ETH as fee, ETH/USDC is originally 3000, so that is 300 USDC.
2. Someone did the first LM.
3. Alice manipulates ETH/USDC price to 1 (hypothetical number). Then Alice can pay 0.1 USDC to get the 0.1 ETH. Alice is paying the USDC, but at a cheap price, to buy back the ETH that was deposited in step 1.

But both USDC and ETH are from Alice's pocket, no?

pkqs90

But both USDC and ETH are from Alice's pocket, no?

Yes. The first 0.1ETH is from Alice's pocket. But you can see in the above coded PoC, Alice can actually buy 0.2ETH, because the `availableInTokens()` is `inToken.balanceOf(Torex) - minimumDeposit`.

When Alice flow is opened, `inToken.balanceOf(Torex)` is 0.1ETH, `minimumDeposit` is 0.1ETH, so `availableInTokens()` is 0. When Alice flow is deleted, `inToken.balanceOf(Torex)` is 0.2ETH (deposit of `feeDistributionPool` is returned to `Torex`), `minimumDeposit` is 0, so `availableInTokens()` is 0.2ETH.

Please correct me if I misunderstood how deposit works for distribution pools.
Thanks!

```
function availableInTokens() override internal view returns (uint256 amount) {
    uint256 minimumDeposit =
    ↪ _inToken.getBufferAmountByFlowRate(_requestedFeeDistFlowRate);
    amount = _inToken.balanceOf(address(this));
    if (amount >= minimumDeposit) return amount - minimumDeposit; else return 0;
}
```



sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:

<https://github.com/superfluid-finance/averagex-contracts-cloned/commit/5d8f1afa882630def154ee2d93ace131d9e717e3>

hellwolf

Oh, I see, that's an interesting observation.

But I think the minimum deposit was also part of the "back charge" paid by Alice. It cannot come from anybody else, since it's always in-sync with the in-token flow rate of traders. So Any time Alice wants to increase in-token flow rate, it would necessarily increase the fee flow rate, which increases the minimum buffer requirement, which increases Alice's back charge amount required.

So I still claim that Alice cannot "gain" any money from somebody else.

Fwiw, I have plugged the same-block LME possibility for the future version of TORE anyways:

<https://github.com/superfluid-finance/averagex-contracts-cloned/commit/5d8f1afa882630def154ee2d93ace131d9e717e3>

WangSecurity

Then in this case, all Alice gets is here tokens back, which represent the backcharged fees, correct?

hellwolf

Then in this case, all Alice gets is here tokens back, which represent the backcharged fees, correct?

Yes. More accurately, Alice get those "backcharged fees", which she paid, back.

And note that, while Alice may manipulate slot0, but there is no out-tokens for her to drain, since liquidity mover's job is to source the out-tokens themselves. One TOREX contract objective is to have low amount of liquidity locked, temporarily, namely in-tokens waiting for liquidity movers to swap them for out-tokens.

WangSecurity

In that case, I agree it's a valid issue, since this leads to less fees being distributed and Alice getting the back charged fees back. But, since the amount is quite small, medium is more appropriate.

Planning to accept the escalation and validate with medium severity.

WangSecurity

Result: Medium Unique



sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- pkqs90: accepted

WangSecurity

@foufrix @pkqs90 as I understand there are no duplicates, correct?

pkqs90

@foufrix @pkqs90 as I understand there are no duplicates, correct?

No dups. All other price manipulation issues don't describe how to get available tokens to swap.

foufrix

@WangSecurity Correct, it's the only issue that talks about getting back charged fees.

sherlock-admin2

The Lead Senior Watson signed off on the fix.



Issue M-2: Updating stake or updating flow may DoS if Torex doesn't enable QuadraticEmission in the beginning.

Source:

<https://github.com/sherlock-audit/2024-06-superboring-judging/issues/22>

Found by

pkqs90

Summary

Updating stake or updating flow may DoS if Torex doesn't enable QuadraticEmission in the beginning.

Vulnerability Detail

First, we need to understand that it's entirely possible for a Torex to be created and operate for a while before the admin enables QuadraticEmission (QE) for it. This is because Torex is created in a permissionless way - anyone can call `SuperBoring.createUniV3PoolTwapObserverAndTorex` to create it. The admin needs to explicitly call `SuperBoring.govQEEnableForTorex` to enable QE.

Now, let's see how enabling QE while the Torex has been active for a while could result in a DoS.

If a user updates their stake or flow, `QuadraticEmissionTIP.onStakeUpdated` and `QuadraticEmissionTIP.onInFlowChanged` are called. However, if QE is not enabled, these functions simply skip execution due to `isQEEnabledForTorex(torex)` being false. If QE is later enabled, performing operations like `q0 + Math.sqrt(newStakedAmount) - Math.sqrt(oldStakedAmount);` or `emissionPool.getUnits(trader) + newTraderUnits - prevTraderUnits` could revert due to underflow if the previous stake amount or traderUnits (flowRate) is larger than the new one.

An example is:

1. Torex is created, but QE is not enabled.
2. A user creates a flowRate of 100 units in QE.
3. QE is then enabled.
4. The user tries to reduce the flowRate to 50 units, but `emissionPool.getUnits(trader) + newTraderUnits - prevTraderUnits` reverts because it evaluates to `0 + 50 - 100`.



This situation arises because QuadraticEmissionTIP incorrectly uses trader's units or stake amount when QE was not enabled, leading to incorrect calculations and potential underflows when QE is later enabled.

```
function onStakeUpdated(EmissionTreasury emissionTreasury, ITorex torex,
                        uint256 oldStakedAmount, uint256 newStakedAmount)
↳ internal
{
    if (isQEEEnabledForTorex(torex)) {
        Storage storage $ = _getStorage();

        // Note: what if sqrt implementation outputs different number over
↳ different implementations?
        uint256 q0 = EnumerableMap.get($.torexQs, address(torex));
> uint256 q1 = q0 + Math.sqrt(newStakedAmount) -
↳ Math.sqrt(oldStakedAmount);
        $.qqSum = $.qqSum + q1 * q1 - q0 * q0;
        EnumerableMap.set($.torexQs, address(torex), q1);

        adjustEmission(emissionTreasury, torex);
    }
}

// @dev This hook updates flow updater's emission share.
function onInFlowChanged(EmissionTreasury emissionTreasury, ITorex torex,
                        address trader, address referrer,
                        int96 prevFlowRate, int96 newFlowRate) internal
{
    assert(trader != referrer); // please provide a nicer revert in the
↳ use-site

    if (isQEEEnabledForTorex(torex)) {
        Storage storage $ = _getStorage();
        ISuperfluidPool emissionPool =
↳ emissionTreasury.getEmissionPool(address(torex));

        // update trader's reward
        // Assumption: the scaling factor is fixed; otherwise, we would have
↳ to store prevTraderUnits.
        uint128 newTraderUnits =
↳ scaleInTokenFlowRateToBoringPoolUnits(torex, newFlowRate);
        {
            uint128 prevTraderUnits =
↳ scaleInTokenFlowRateToBoringPoolUnits(torex, prevFlowRate);
            emissionTreasury.updateMemberEmissionUnits(address(torex),
↳ trader,
```



```

>
↪ emissionPool.getUnits(trader)
>                                     + newTraderUnits -
↪ prevTraderUnits);
    }
    ...
}

```

Impact

Updating stake or updating flow may DoS.

Code Snippet

- <https://github.com/sherlock-audit/2024-06-superboring/blob/main/averagex-contracts-cloned/packages/evm-contracts/src/BoringPrograms/QuadraticEmissionTIP.sol#L132-L134>
- <https://github.com/sherlock-audit/2024-06-superboring/blob/main/averagex-contracts-cloned/packages/evm-contracts/src/BoringPrograms/QuadraticEmissionTIP.sol#L108>

Tool used

Manual Review

Recommendation

Maintain the user's flowRate and stakedAmount within QuadraticEmissionTIP and only update it after QE is enabled, and don't rely on passed in previous values.

Discussion

hellwolf

This sounds plausible. Thanks for the report. I will add a test case and refine the code logic.

sherlock-admin3

Escalate

I think this should be high sev.

The issue explains a scenario where if user had created a flow to a torex, but can't reduce it due to the dos in onInFlowChanged(). This means a



user cannot stop his supertokens outflowing to the torex app, which is a loss of funds.

Also, the user cannot reduce his superboring stakes from the torex, which is also loss of funds.

You've deleted an escalation for this issue.

hellwolf

I argue that this is not going to be a high severity, because the condition of this happening requiring the operator of the QE program not aware of such bug existing.

Knowing this bug existing, the QE operator should:

1. Not enable QE for an active TOREX.
2. Request dev to fix this bug; until then QE must not be enabled to TOREX.

Hence I believe that while this is a valid bug, it should fit medium severity bucket instead.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:

<https://github.com/superfluid-finance/averagex-contracts-cloned/commit/d274392762a408f4d2ad33d149de99ca2d60f27d>

pkqs90

@hellwolf I agree with you. Removing the escalation.

sherlock-admin2

The Lead Senior Watson signed off on the fix.



Issue M-3: AdjustmentFlow for feeDistributionPool is streamed to SuperBoring contract, but there is no way to rescue it

Source:

<https://github.com/sherlock-audit/2024-06-superboring-judging/issues/34>

The protocol has acknowledged this issue.

Found by

pkqs90

Summary

AdjustmentFlow for feeDistributionPool is streamed to SuperBoring contract, but there is no way to rescue it.

Vulnerability Detail

From this doc <https://github.com/superfluid-finance/protocol-monorepo/wiki/General-Distribution-Agreement#adjustment-flow>, we know that if a pool's flowRate cannot be fully divided by the total amount of units, the remainder in the flowRate is streamed to the pool admin.

In the case of the feeDistributionPool, the admin is the controller, which is the SuperBoring protocol. However, once the inToken is streamed to the SuperBoring protocol, there is no method to rescue it, so it remains locked inside forever.

This is not a trivial amount. Consider $1e5$ users staking for a Torex, each with $1e5$ units, making the total unit amount $1e10$, which is the upper limit for the flowRate streamed to the SuperBoring protocol. This means $1e10$ tokens are streamed to the SuperBoring protocol per second. If the stream continues for one day, it would amount to $1e10 * 86400 \sim 8e14$ tokens (note that all SuperTokens have 18 decimals). This is a non-trivial for some expensive tokens.

```
>     feeDistributionPool = _inToken.createPool(address(controller),
↪   PoolConfig({
        transferabilityForUnitsOwner: false,
        distributionFromAnyAddress: true
    }));

function _onInFlowChanged(ISuperToken superToken, address sender, int96
↪   prevFlowRate, uint256 lastUpdated,
        bytes memory ctx) internal
    returns (bytes memory newCtx)
{
```



```

    ...

    // update fee distribution flow rate, this requires the buffer cost
    ↪ taken as part of back adjustment.
    > newCtx = _inToken.distributeFlowWithCtx(address(this),
    ↪ feeDistributionPool, _requestedFeeDistFlowRate, newCtx);
    ↪ (, _actualFeeDistFlowRate, _feeDistBuffer) =
    ↪ _inToken.getGDAFlowInfo(address(this), feeDistributionPool);
    }

```

Impact

An non-trivial amount of token may be locked up in SuperBoring contract.

Code Snippet

- <https://github.com/sherlock-audit/2024-06-superboring/blob/main/averagex-contracts-cloned/packages/evm-contracts/src/Torex.sol#L148>
- <https://github.com/sherlock-audit/2024-06-superboring/blob/main/averagex-contracts-cloned/packages/evm-contracts/src/Torex.sol#L292>

Tool used

Manual Review

Recommendation

Add a token rescue function for admins in SuperBoring contract.

Discussion

hellwolf

Hi, this is a confirmed report.

However, due to the amount "leaked" into the SuperBoring is inconsequential, we will not fix it for now.

Note that, should that "leaked" amount become consequential, we could still opt to upgrade SuperBoring to recoup those amounts.

Furthermore, the right design could be that the "admin" of the fee distribution pools is the "DistributionFeeManager."

foufrix



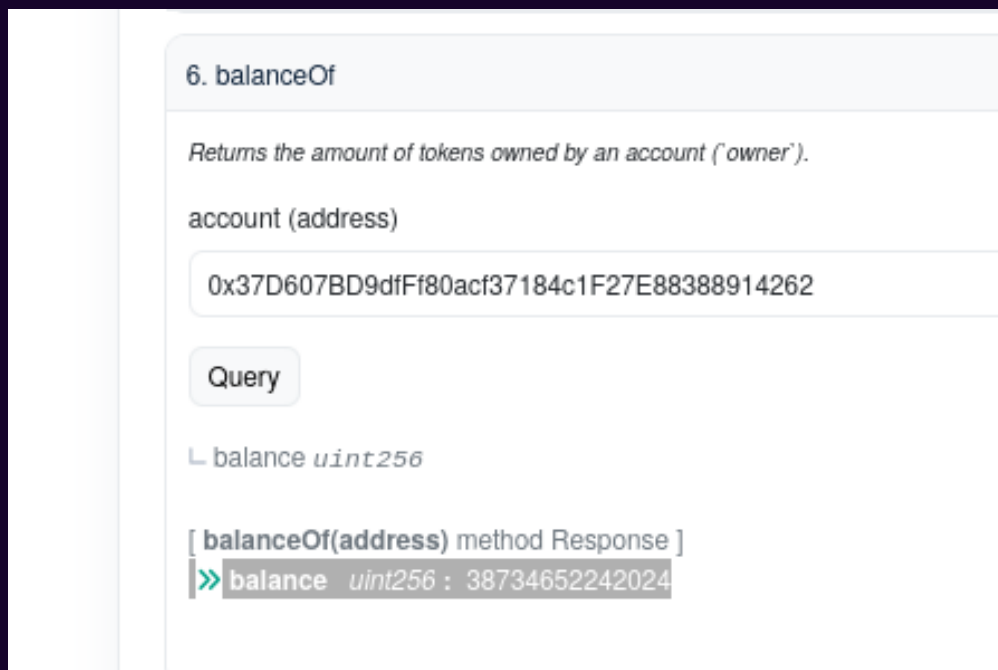
@hellwolf do you have an estimation of this amount?

hellwolf

@hellwolf do you have an estimation of this amount?

Yes, I knew it was small because of the design. But to double confirm, I just did a check of DEGENx (BASE):

- <https://basescan.org/token/0x1eff3dd78f4a14abfa9fa66579bd3ce9e1b30529>
- `balanceOf(0x37D607BD9dFf80acf37184c1F27E88388914262) # SB_ADDRESS=0x37D607BD9dFf80acf37184c1F27E88388914262`
- `result: balance 38734652242024 (3.8734652242024e-05)`



The screenshot shows a web interface for a blockchain explorer. It features a search bar with the address '0x37D607BD9dFf80acf37184c1F27E88388914262' and a 'Query' button. Below the button, the result is displayed as 'balance uint256 : 38734652242024'. The interface is clean and modern, with a light blue header and a white body.

foufrix

And this contract <https://basescan.org/txs?a=0x37d607bd9dfff80acf37184c1f27e88388914262p=37> was deployed 20 days ago Ok so even if it were for ETH this would represent 0.12, *meaningatmostlet's exaggerate0.50/month* Planning to remove this issue as it seems like dust amount.

pkqs90

`Ethx.balanceOf(SuperBoring) = 547240791460216 = 0.00054e18`. This is the amount of Ethx that is streamed from `feeDistributionPool` to `SuperBoring` due to rounding issue.

I'm not sure if there is a more efficient way to calculate the total amount of Ethx



streamed to feeDistributionPool(). I will sum up the total volume of Ethx torex inflow and multiply the fee percentage to acquire the total fee.

Use SuperBoring.getTotalityStats(Torex) to calculate the total Ethx flow. There are 6 torexes, see the numbers in below screenshots. The sum is
 $3826452599224841134 + 2158552628342779578 + 211458692196439160 + 158390334951414130 + 607334136134322566 + 140071995301962554 = 7102260386151759122 = 7.1e18$

Multiply this by 0.5% fee is $7.1e18 * 0.5\% = 0.0355e18$.

This means 0.00054e18 out of 0.0355e18 of fees is lost, which is 1.5%.

@hellwolf @foufrix Can you help check if there is anything wrong with the above calculation? Also, I'm not sure if there is a more precise way to calculate the total volume passed through feeDistributionPool().

[1] Ethx.balanceOf(SuperBoring) <https://basescan.org/address/0x46fd5cfB4c12D87acD3a13e92BAa53240C661D93#readProxyContract>

6. balanceOf

Returns the amount of tokens owned by an account ('owner').

account (address)

0x37d607bd9dfff80acf37184c1f27e88388914262

Query

↳ balance uint256

[balanceOf(address) method Response]

» balance uint256 : 547240791460216

[2] All Torexes related to Ethx. There are 6 torexes that take Ethx as inToken. <https://basescan.org/address/0x37d607bd9dfff80acf37184c1f27e88388914262#readPr>

8. getAllTorexesMetadata

A backward compatible parameter-less getAllTorexesMetadata which hardcodes a query range.

Query

↳ metadataList tuple[]

[getAllTorexesMetadata method Response]

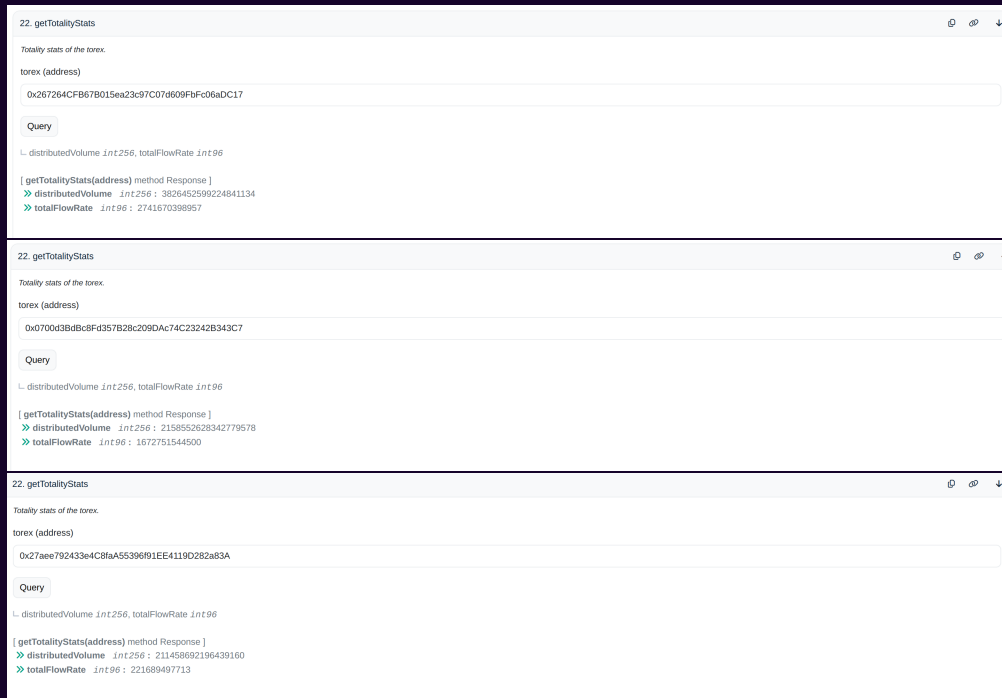
» metadataList tuple[] :

[[0x267264CFB67B015ea23c97C07d609FbF06aDC17, 0x46fd5cfB4c12D87acD3a13e92BAa53240C661D93, 0xD04383398dD2426297da660F9CCA3d439AF9ce1b]
[0x269F9EF6868F70FB20DDF7CfD69Fe1DBFD307dE, 0xD04383398dD2426297da660F9CCA3d439AF9ce1b, 0x46fd5cfB4c12D87acD3a13e92BAa53240C661D93]
[0x0700d3BdBc8Fd357B28c209DAc74C23242B343C7, 0x46fd5cfB4c12D87acD3a13e92BAa53240C661D93, 0x1ef3Dd78F4A14aBfa9Fa66579bD3Ce9E1B30529]
[0x68E5E539374353445b03Ec87D2Abfe2C791dEebc, 0x1ef3Dd78F4A14aBfa9Fa66579bD3Ce9E1B30529, 0x46fd5cfB4c12D87acD3a13e92BAa53240C661D93]
[0x27aee792433e4C8faA55396f91EE4119D282a83A, 0x46fd5cfB4c12D87acD3a13e92BAa53240C661D93, 0x09b1AD979d093377e201d804Fa9aC0a9a07cfB0b]
[0x76BA7a8a4d8320c6E9D4542255Fb05268f1B48BE, 0x09b1AD979d093377e201d804Fa9aC0a9a07cfB0b, 0x46fd5cfB4c12D87acD3a13e92BAa53240C661D93]
[0x6a19Ee195D996B70667894E2dAE9D10EE4d3D969, 0x8414Ab8C70c7b16a46012d49b8111959Baf2fC42, 0x46fd5cfB4c12D87acD3a13e92BAa53240C661D93]
[0xE0C1424108963425FB1Cca1829A7Cb610eecd5, 0x46fd5cfB4c12D87acD3a13e92BAa53240C661D93, 0x8414Ab8C70c7b16a46012d49b8111959Baf2fC42]
[0x9b3E9D6aF3ec387AbC9733c33a113Bb5Ed21ee, 0x5f2Fab273F1F64b6bc6ab8F35314CD21501F35C5, 0x46fd5cfB4c12D87acD3a13e92BAa53240C661D93]
[0x598aF5742B4a6aBd7b66B2aEd3Da17690ab72f2, 0x46fd5cfB4c12D87acD3a13e92BAa53240C661D93, 0x5f2Fab273F1F64b6bc6ab8F35314CD21501F35C5]
[0x16dF7D980198861Ba701C47C7D5E9Cb2D6bf7F8f, 0x46fd5cfB4c12D87acD3a13e92BAa53240C661D93, 0xefbE11336b0008dCE3797C515E6457cC4841645c]
[0x43dc12CA897e6533e78B33b43e6993597D09DD73, 0xefbE11336b0008dCE3797C515E6457cC4841645c, 0x46fd5cfB4c12D87acD3a13e92BAa53240C661D93]]

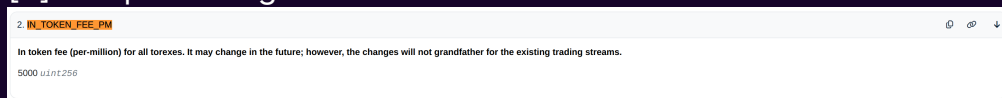
oxyContract

[3] SuperBoring.getTotalityStats(Torex). Only selecting top 3 torexes.





[4] Fee percentage is 0.5%.



pkqs90

Escalate

Please see above comment.

sherlock-admin3

Escalate

Please see above comment.

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.

pkqs90

Also note that the lost of fees is related to 1) the price of inToken, 2) amount of staked tokens. This is because the more expensive inToken is, the smaller the fee volume is likely to be; and the more staked tokens, the larger the member units (denominator), thus a possible large rounding error. This is why Ethx is more likely to have a larger rounding error than other tokens.



hellwolf

Hi, thanks for the escalation.

We have a [dune dashboard](#), so I gathered all the ETHx volume

```
>>> totalETHxVolume = 12772+472+2126+7252+534+712
>>> (562440411980940 / 1e18 * 3000) / (totalETHxVolume * 0.005)
0.01413877355407089
```

Yes, it is 1.4% currently locked in SuperBoring contract. The reason this is happening is because:

1. Volume of ETHx is under our estimation
2. BORING token we are issuing is anticipating a much higher volume.

So, the loss is a few magnitudes higher than planned.

Also, that is not to say they are permanently lost. There is still plan to add SuperBoring logic to recoup those funds back to FeeDistributionManager, which is rather trivial to do.

To be fair to the watson, I am happy to consider it is a medium, and it won't be fixed for now.

foufrix

As it's more than 0.01%, it's a valid medium, I agree with the escalation

WangSecurity

Agree with the escalation and believe the report shows sufficient explanation of the rounding down precision loss here. Planning to accept the escalation and validate with medium severity.

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- [pkqs90](#): accepted

WangSecurity

@foufrix @pkqs90 are there any duplicates?

pkqs90



I didn't find one. @foufrix can help double check.

foufrix

I confirm that this issue is unique



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.

