SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: RealWagmi
Prepared by: Sherlock
Lead Security Expert: stopthecap
Dates Audited: June 20 - June 25, 2023
Prepared on: August 22, 2023

Introduction

Swap earn and provide liquidity on the leading decentralized protocol built on zkSync. Experience the future of decentralized finance with Wagmi.

Scope

Repository: RealWagmi/concentrator

Branch: main

Commit: dcff15564d079f5ff32686ad738873f274de48fd

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
4	5

Security experts who found valid issues

n1punp
tsvetanovv
josephdara
mahdiRostami
twcctop
kutugu
shealtielanz
tank
stopthecap
duc
ginlee
Bauchibred
n33k
bitsurfer
shogoki
BugBusters
ash
ast3ros
crimson-rat-reach
Phantasmagoria
OxZ00mer
sashik_eth
qpzm
ni8mare
lil.eth
toshii
Avci
Jaraxxus
rogue-lion-0619
0xdice91

Issue H-1: Wrong calculation of `tickCumulatives` due to hardcoded pool fees

Source: https://github.com/sherlock-audit/2023-06-real-wagmi-judging/issues/48

Found by

Bauchibred, OxZ00mer, ast3ros, bitsurfer, crimson-rat-reach, duc, josephdara, mahdiRostami, n1punp, n33k, shogoki, stopthecap

Summary

Wrong calculation of `tickCumulatives` due to hardcoded pool fees

Vulnerability Detail

Real Wagmi is using a hardcoded `500` fee to calculate the `amountOut` to check for slippage and revert if it was to high, or got less funds back than expected.

There are several problems with the hardcoding of the `500` as the fee.

- Not all tokens have `500` fee pools
- The swapping takes place in pools that don't have a `500` fee
- The `500` pool fee is not the optimal to fetch the `tickCumulatives` due to low volume

Specially as they are deploying in so many secondary chains like Kava, this will be a big problem pretty much in every transaction over there.

If any of those scenarios is given, `tickCumulatives` will be incorrectly calculated and it will set an incorrect slippage return.

Impact

Incorrect slippage calculation will increase the risk of `rebalanceAll()` rebalance getting rekt.

Code Snippet

https://github.com/sherlock-audit/2023-06-real-wagmi/blob/main/concentrator/contracts/Multipool.sol#L816-L838 https://github.com/sherlock-audit/2023-06-real-wagmi/blob/main/concentrator/contracts/Multipool.sol#L823

Tool used

Manual Review

Recommendation

Consider allowing the fees as an input and consider not even picking low TVL pools with no transations

Discussion

ctf-sec

There are pools that are support not only 500 fee tiers

because the impact result in Incorrect slippage calculation

the severity is still high

fann95

fixed , quotePoolAddress can be changed
https://github.com/RealWagmi/concentrator/blob/fbccf1caf28edbb23db8c7e0409d0c40c3a56461/contracts/Multipool.sol#L848C61-L848C77

0xffff11

While the fix allows to change the address of the pool, it is not the best fix possible. The best would either store the pools in a mapping with their fees as a key, or pass directly the address as a parameter.

Issue H-2: No slippage protection when withdrawing and providing liquidity in rebalanceAll

Source: https://github.com/sherlock-audit/2023-06-real-wagmi-judging/issues/94

Found by

ast3ros, n33k

Summary

When `rebalanceAll` is called, the liquidity is first withdrawn from the pools and then deposited to new positions. However, there is no slippage protection for these operations.

Vulnerability Detail

In the `rebalanceAll` function, it first withdraws all liquidity from the pools and deposits all liquidity to new positions.

```
_withdraw(_totalSupply, _totalSupply);
```

https://github.com/sherlock-audit/2023-06-real-wagmi/blob/main/concentrator/contracts/Multipool.sol#L853-L853

```
_deposit(reserve0, reserve1, _totalSupply, slots);
```

https://github.com/sherlock-audit/2023-06-real-wagmi/blob/main/concentrator/contracts/Multipool.sol#L885-L885

However, there are no parameters for `amount0Min` and `amount1Min`, which are used to prevent slippage. These parameters should be checked to create slippage protections.

https://docs.uniswap.org/contracts/v3/guides/providing-liquidity/decrease-liquidity
https://docs.uniswap.org/contracts/v3/guides/providing-liquidity/increase-liquidity

Actually, they are implemented in the `deposit` and `withdraw` functions, but just not in the rebalanceAll function.

https://github.com/sherlock-audit/2023-06-real-wagmi/blob/main/concentrator/contracts/Multipool.sol#L437-L438 https://github.com/sherlock-audit/2023-06-real-wagmi/blob/main/concentrator/contracts/Multipool.sol#L559-L560

Impact

The withdraw and provide liquidity operations in rebalanceAll are exposed to high slippage and could result in a loss for LPs of multipool.

Code Snippet

https://github.com/sherlock-audit/2023-06-real-wagmi/blob/main/concentrator/contracts/Multipool.sol#L853-L853 https://github.com/sherlock-audit/2023-06-real-wagmi/blob/main/concentrator/contracts/Multipool.sol#L885-L885

Tool used

Manual Review

Recommendation

Implement slippage protection in rebalanceAll as suggested to avoid loss to the protocol.

Discussion

fann95

FIXED. The _checkPriceDeviation function checks deviations from the weighted average price for each pool before _withdraw & _deposit https://github.com/RealWagmi/concentrator/blob/fbccf1caf28edbb23db8c7e0409d0c40c3a56461/contracts/Multipool.sol#L984C14-L984C34

fann95

slippage is also checked during exchange https://github.com/RealWagmi/concentrator/blob/fbccf1caf28edbb23db8c7e0409d0c40c3a56461/contracts/Multipool.sol#L902

0xffff11

Fixed by checking the deviation for withdrawing and depositing while rebalancing.

Issue H-3: Usage of `slot0` is extremely easy to manipulate

Source: https://github.com/sherlock-audit/2023-06-real-wagmi-judging/issues/97

Found by

BugBusters, Jaraxxus, ash, bitsurfer, kutugu, ni8mare, rogue-lion-0619, sashik_eth, shealtielanz, stopthecap, toshii, tsvetanovv

Summary

Usage of `slot0` is extremely easy to manipulate

Vulnerability Detail

Real Wagmi is using `slot0` to calculate several variables in their codebase:

https://github.com/sherlock-audit/2023-06-real-wagmi/blob/main/concentrator/contracts/Multipool.sol#L589-L596

slot0 is the most recent data point and is therefore extremely easy to manipulate.

Multipool directly uses the token values returned by `getAmountsForLiquidity`

to calculate the reserves.

Which they are used to calculate the `lpAmount` to mint from the pool. This allows a malicious user to manipulate the amount of the minted by a user.
https://github.com/sherlock-audit/2023-06-real-wagmi/blob/main/concentrator/contracts/Multipool.sol#L483

https://github.com/sherlock-audit/2023-06-real-wagmi/blob/main/concentrator/contracts/Multipool.sol#L458

Impact

Pool lp value can be manipulated and cause other users to receive less lp tokens.

Code Snippet

https://github.com/sherlock-audit/2023-06-real-wagmi/blob/main/concentrator/contracts/Multipool.sol#L589-L596 https://github.com/sherlock-audit/2023-06-real-wagmi/blob/main/concentrator/contracts/Multipool.sol#L754-L755 https://github.com/sherlock-audit/2023-06-real-wagmi/blob/main/concentrator/contracts/Multipool.sol#L749 https://github.com/sherlock-audit/2023-06-real-wagmi/blob/main/concentrator/contracts/Multipool.sol#L289

Tool used

Manual Review

Recommendation

To make any calculation use a TWAP instead of slot0.

Discussion

SergeKireev

Escalate

Usage of slot0 is correct in this context. Using anything else as a price source would misrepresent current token liquidity in UniV3 pools and break the calculations used during minting (and could actually result in the user receiving more or less shares than due).

The possibility of price manipulation is correctly mitigated by slippage parameters in the deposit function

sherlock-admin

> Escalate
>
> Usage of slot0 is correct in this context. Using anything else as a price source would misrepresent current token liquidity in UniV3 pools and break the calculations used during minting (and could actually result in the user receiving more or less shares than due).
>
> The possibility of price manipulation is correctly mitigated by slippage parameters in the deposit function

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.

fann95

FIXED. The _checkPriceDeviation function checks deviations from the weighted average price for each pool https://github.com/RealWagmi/concentrator/blob/fbccf1caf28edbb23db8c7e0409d0c40c3a56461/contracts/Multipool.sol#L984C14-L984C34

hrishibhat

@ctf-sec

hrishibhat

Result: High Has duplicates Although in general, the `slot0` manipulation issue may need further discussion on its validity,severity and practicality of the issue, currently keeping this issue as a valid high based on historical decisions.

This will be further discussed and with additional information, the rulebook will be updated accordingly

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- crimson-rat-reach: rejected

0xffff11

Fixed by checking deviations `uint256 deviation = currentTick > avarageTick` from TWAP values instead of direct usage of `slot0()`

Issue H-4: Deposit transactions lose funds to front-running when multiple fee tiers are available

Source: https://github.com/sherlock-audit/2023-06-real-wagmi-judging/issues/105

Found by

crimson-rat-reach

Summary

Deposit transactions lose funds to front-running when multiple fee tiers are available

Vulnerability Detail

The deposit transaction takes in minimum parameters for amount0 and amount1 of tokens that the user wishes to deposit, but no parameter for the minimum number of LP tokens the user expects to receive. A malicious actor can limit the number of LP tokens that the user receives in the following way:

A user Alice submits a transaction to deposit tokens into Multipool where (amount0Desired, amount0Min) > (amount1Desired, amount1Min)

A malicious actor Bob can front-run this transaction if there are multiple feeTiers:

- by first moving the price of feeTier1 to make tokenA very cheap (lots of tokenA in the pool)
- then moving the price of feeTier2 in opposite direction to make tokenB very cheap (lots of tokenB in the pool)

This results in reserves being balanced accross feeTiers, and the amounts resulting from `_optimizeAmounts` are balanced as well: https://github.com/sherlock-audit/2023-06-real-wagmi/blob/82a234a5c2c1fc1921c63265a9349b71d84675c4/concentrator/contracts/Multipool.sol#L780-L808

So the minimum amounts checks pass and but results as less LP tokens minted, because even though the reserves are balanced, they are also overinflated due to the large swap, and the ratio: https://github.com/sherlock-audit/2023-06-real-wagmi/blob/82a234a5c2c1fc1921c63265a9349b71d84675c4/concentrator/contracts/Multipool.sol#L468-L470

becomes a lot smaller than before the large swap

Impact

The user loses funds as a result of maximum slippage.

Code Snippet

Tool used

Manual Review

Recommendation

Add extra parameters for the minimum number of LP tokens that the user expects, instead of just checking non-zero amount: https://github.com/sherlock-audit/2023-06-real-wagmi/blob/82a234a5c2c1fc1921c63265a9349b71d84675c4/concentrator/contracts/Multipool.sol#L473

Discussion

SergeKireev

Escalate

This should not be a duplicate of #180 but a unique high (see my escalation on #180 on why similar looking issues are invalid)

sherlock-admin

> Escalate
>
> This should not be a duplicate of #180 but a unique high (see my escalation on #180 on why similar looking issues are invalid)

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.

ctf-sec

the issue still talk about the missing slippage when depositing / minting

in fact, the recommendation in this report is the same as the issue 180

duplicate of #180

SergeKireev

> the issue still talk about the missing slippage when depositing / minting
>
> in fact, the recommendation in this report is the same as the issue 180

> duplicate of #180

Respectfully, the recommendation in this issue is absolutely different from the one in #180.

I summarized in my escalation on issue #180 why #180 should be invalid. Slippage protection works correctly unless multiple pools are involved

fann95

FIXED: We simplified the slip check and added the minimum acceptable amount of liquidity tokens to be minted.
https://github.com/RealWagmi/concentrator/blob/fbccf1caf28edbb23db8c7e0409d0c40c3a56461/contracts/Multipool.sol#L415C19-L415C19

hrishibhat

Result: High Unique Considering this issue as a valid high since it opens a user to sandwiching and the funds at loss are unbounded.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- crimson-rat-reach: accepted

huuducsc

I believe this issue should be a low/non-issue, since the attacker can't increase both reserve0 and reserve1 of the Multipool contract through this scenario. This is due to the slippage of the UniswapV3 pool during the attacker's swaps, when you try to increase token0, you need a greater amount of token1 value.

> A malicious actor Bob can front-run this transaction if there are multiple feeTiers:
>
> by first moving the price of feeTier1 to make tokenA very cheap (lots of tokenA in the pool) then moving the price of feeTier2 in opposite direction to make tokenB very cheap (lots of tokenB in the pool)

Therefore, the reserves will not be "overinflated". Also, since attacker can't increase both reserve0 and reserve1, the LP shares of the user will not become a lot smaller than before the large swap.

SergeKireev

> I believe this issue should be a low/non-issue, since the attacker can't increase both reserve0 and reserve1 of the Multipool contract through this scenario. This is due to the slippage of the UniswapV3 pool during the attacker's swaps, when you try to increase token0, you need a greater amount of token1 value.

A malicious actor Bob can front-run this transaction if there are multiple feeTiers:

by first moving the price of feeTier1 to make tokenA very cheap (lots of tokenA in the pool) then moving the price of feeTier2 in opposite direction to make tokenB very cheap (lots of tokenB in the pool)

Therefore, the reserves will not be "overinflated". Also, since attacker can't increase both reserve0 and reserve1, the LP shares of the user will not become a lot smaller than before the large swap.

The attacker can increase both reserves precisely because multiple underlying pools are available (multiple fee tiers), so the attacker adds more of token0 to one fee tier and more of token1 to the other fee tier. This was detailed in the scenario of the original submission.

The slippage during the first swap of the attacker does not matter, as a sandwicher always incurs heavy slippage during a sandwich and rebalances during back run

huuducsc

I know that an attacker can manipulate multiple pools with multiple fee tiers. However, when the attacker adds more token0 to the first fee tier, the token1 amount of the position in this UniswapV3 pool will decrease. Similarly, the token0 amount of the position in the second fee tier will decrease. Moreover, the decreased amounts hold more value than the increased amounts due to the slippage of the UniswapV3 pool.

For example:

1. There are 2 positions in a 2 fee tier UniswapV3 pool. Initially, the first position (first fee tier) represents 1000 USDT-1000 USDC, and the second position (second fee tier) represents 2000 USDT-2000 USDC => Assuming that there is no fee, the initial reserves are 3000 USDT-3000 USDC.
2. The attacker increases USDT in the first pool -> the first position will have 1500 USDT-499 USDC. The attacker increases USDC in the second pool -> the second position will have 1499 USDT-2500 USDC. => The new reserves will be 2999 USDT-2999 USDC (decreased).

huuducsc

The amounts of token0 and token1 in a position are calculated based on the current price and liquidity amount https://github.com/Uniswap/v3-periphery/blob/main/contracts/libraries/LiquidityAmounts.sol#L120-L136. This mechanism is used in concentrated pools to prevent profiting from manipulating price then withdrawing position. If an attacker can increase the reserves without incurring any losses, as in the scenario described above, could anyone else do something similar and profit from their positions?

fann95

For example:

1. There are 2 positions in a 2 fee tier UniswapV3 pool. Initially, the first position (first fee tier) represents 1000 USDT-1000 USDC, and the second position (second fee tier) represents 2000 USDT-2000 USDC => Assuming that there is no fee, the initial reserves are 3000 USDT-3000 USDC.
2. The attacker increases USDT in the first pool -> the first position will have 1500 USDT-499 USDC. The attacker increases USDC in the second pool -> the second position will have 1499 USDT-2500 USDC. => The new reserves will be 2999 USDT-2999 USDC (decreased).

https://github.com/RealWagmi/concentrator/blob/9036fe43b0de2694a0c5e18cfe821d40ce17a9b8/contracts/Multipool.sol#L1013C14-L1013C34 We have protection for a similar scenario of price manipulation in pools..do you think it's not enough? It looks like this issue is invalid.

0xffff11

Fixed by adding the `lpAmountMin` parameter when depositing

Issue H-5: The `_estimateWithdrawalLp` function might return a very large value, result in users losing significant incentives or being unable to withdraw from the Dispatcher contract

Source:
https://github.com/sherlock-audit/2023-06-real-wagmi-judging/issues/142

Found by

crimson-rat-reach, duc, qpzm

Summary

The `_estimateWithdrawalLp` function might return a very large value, result in users losing significant incentives or being unable to withdraw from the Dispatcher contract

Vulnerability Detail

In Dispatcher contract, `_estimateWithdrawalLp` function returns the value of shares amount based on the average of ratios `amount0 / reserve0` and `amount1 / reserve1`.

From `Dispatcher.withdraw` and `Dispatcher.deposit` function, amount0 and amount1 will be the accumulated fees of users

However, it is important to note that the values of reserve0 and reserve1 can fluctuate significantly. This is because the positions of the Multipool in UniSwapV3 pools (concentrated) are unstable on-chain, and they can change substantially as the state of the pools changes. As a result, the `_estimateWithdrawalLp` function might return a large value even for a small fee amount. This could potentially lead to reverting due to underflow in the deposit function (in cases where `lpAmount > user.shares`), or it could result in withdrawing a larger amount of Multipool LP than initially expected.

Scenario:

1. Total supply of Multipool is 1e18, and Alice has 1e16 (1%) LP amounts which deposited into Dispatcher contract.
2. Alice accrued fees = 200 USDC and 100 USDT
3. The reserves of Multipool are 100,000 USDC and 100,000 USDT, `_estimateWithdrawalLp` of Alice fees will be `(0.2% + 0.1%) / 2 * totalSupply = 0.15% * totalSupply` = 1.5e15 LP amounts

4. However, in some cases, UniSwapV3 pools may experience fluctuations, reserves of Multipool are 10,000 USDC and 190,000 USDT, `_estimateWithdrawalLp` of Alice fees will be `(2% + 0.052%) / 2 * totalSupply = 1.026% * totalSupply` = 1.026e16 LP amounts This result is greater than LP amounts of Alice (1e16), leading to reverting by underflow in deposit/withdraw function of Dispatcher contract.

Impact

Users may face 2 potential issues when interacting with the Dispatcher contract.

1. They might be unable to deposit/withdraw
2. Secondly, users could potentially lose significant incentives when depositing or withdrawing due to unexpected withdrawals of LP amounts for their fees.

Code Snippet

https://github.com/sherlock-audit/2023-06-real-wagmi/blob/main/concentrator/contracts/Dispatcher.sol#L140-L150

Tool used

Manual review

Recommendation

Shouldn't use the average ratio for calculation in `_estimateWithdrawalLp` function

Discussion

ctf-sec

Think can change the severity to medium

huuducsc

Escalate

I believe the severity of this issue is high. There is no reason for downgrading this issue to medium, since it has high impact and high likelihood. The state of the UniswapV3 pool always changes, which causes the reserves of the Multipool contract (Multipool.getReserve()) to fluctuate significantly. The reserves of the Multipool contract are accumulated from all token amounts of each position. (code snippet). It is a fact that if the tick range of a position is short, the token amounts of this position can change significantly by 1 swap, for example, from (1e21, 0) to (0, 1e21)! This is because the amounts of each position depend on the situation of the current `sqrtP` (square root of the price) of this pool within the tick

range of this position. Therefore, using the average calculation for `_estimateWithdrawalLp` function always puts users at risk of significant loss.

sherlock-admin

> Escalate
>
> I believe the severity of this issue is high. There is no reason for downgrading this issue to medium, since it has high impact and high likelihood. The state of the UniswapV3 pool always changes, which causes the reserves of the Multipool contract (Multipool.getReserve()) to fluctuate significantly. The reserves of the Multipool contract are accumulated from all token amounts of each position. (code snippet). It is a fact that if the tick range of a position is short, the token amounts of this position can change significantly by 1 swap, for example, from (1e21, 0) to (0, 1e21)! This is because the amounts of each position depend on the situation of the current `sqrtP` (square root of the price) of this pool within the tick range of this position. Therefore, using the average calculation for `_estimateWithdrawalLp` function always puts users at risk of significant loss.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

fann95

fixed https://github.com/RealWagmi/concentrator/blob/fbccf1caf28edbb23db8c7e0409d0c40c3a56461/contracts/Dispatcher.sol#L268 and https://github.com/RealWagmi/concentrator/blob/fbccf1caf28edbb23db8c7e0409d0c40c3a56461/contracts/Dispatcher.sol#L409C17-L409C17

JeffCX

Based on the impact

> Users may face 2 potential issues when interacting with the Dispatcher contract.

They might be unable to deposit/withdraw Secondly, users could potentially lose significant incentives when depositing or withdrawing due to unexpected withdrawals of LP amounts for their fees.

All impact is medium except when user are not able to withdraw and withdraw revert in underflow,

I think a high severity is ok

hrishibhat

@huuducsc This revert happens only when there is a massive fluctuation in the pool right? These seem to be temporary or extreme cases. I think this should be a medium.

huuducsc

> @huuducsc This revert happens only when there is a massive fluctuation in the pool right? These seem to be temporary or extreme cases. I think this should be a medium.

Fluctuations in positions usually occur in an UniswapV3 pool, which utilizes a concentrated price mechanism. For example, if the price of ARB is currently 1.25 USDC, and the Multipool contract holds a position in the ARB-USDC pool with a price range of [1.245, 1.255], then the ratio of tokens (token0/token1) in this position is 1:1. However, if the price of ARB increases to 1.253, the ratio of tokens in this position changes to 2:8.

hrishibhat

Result: High Has duplicates Agree with the above comments and the impact mentioned in its duplicates considering this a valid high as the price change can get user funds to be stuck forever.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- huuducsc: accepted

Issue M-1: fee calculations

Source: https://github.com/sherlock-audit/2023-06-real-wagmi-judging/issues/46

Found by

n1punp, tank

Summary

refer to this LOCs: https://github.com/sherlock-audit/2023-06-real-wagmi/blob/main/concentrator/contracts/Multipool.sol#L722-L746, these LOCs don't allow underflow/overflow, but refering to https://github.com/Uniswap/v3-core/issues/573 which allows underflow/overflow. This will lead to some transactions will revert.

Vulnerability Detail

-

Impact

some user transactions will revert.

Code Snippet

```
if (liquidity > 0) {
    (
        uint256 feeGrowthInside0X128Pending,
        uint256 feeGrowthInside1X128Pending
    ) = _getFeeGrowthInside(
            IUniswapV3Pool(position.poolAddress),
            slots[i].tick,
            position.lowerTick,
            position.upperTick
        );
    pendingFee0 += uint128(
        FullMath.mulDiv(
            feeGrowthInside0X128Pending - feeGrowthInside0LastX128,
            liquidity,
            FixedPoint128.Q128
        )
    );
    pendingFee1 += uint128(
        FullMath.mulDiv(
            feeGrowthInside1X128Pending - feeGrowthInside1LastX128,
            liquidity,
```

```
                FixedPoint128.Q128
            )
        );
    }
```

Tool used

-

Manual Review

Recommendation

add unchecked {} when calculate the pending fee

Discussion

huuducsc

Escalate

This issue is not a duplicate of #88, and it is an invalid issue. The above calculation is similar to UniswapV3's implementation for accruing fees, so it is correct. UniswapV3 doesn't allow overflowing/underflowing while updating the position, refer to this code snippet https://github.com/Uniswap/v3-core/blob/main/contracts/libraries/Position.sol#L61-L76 Additionally, the impact of this issue is low.

sherlock-admin

> Escalate
>
> This issue is not a duplicate of #88, and it is an invalid issue. The above calculation is similar to UniswapV3's implementation for accruing fees, so it is correct. UniswapV3 doesn't allow overflowing/underflowing while updating the position, refer to this code snippet https://github.com/Uniswap/v3-core/blob/main/contracts/libraries/Position.sol#L61-L76 Additionally, the impact of this issue is low.

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.

n1punp

@huuducsc This is a valid issue (also dupe of #1) .

Your analysis is incorrect. UniswapV3 does allow overflowing / underflowing. The code snippet you're posting is for Solidity version < 0.8.0, which has not safe math by default

JeffCX

> @huuducsc This is a valid issue (also dupe of #1) .
>
> Your analysis is incorrect. UniswapV3 does allow overflowing / underflowing. The code snippet you're posting is for Solidity version < 0.8.0, which has not safe math by default

Agree,

not a duplicate of #88, seperate valid medium based on the comments;

https://github.com/Uniswap/v3-core/issues/573

the Uniswap code use compiler version before 0.8.0 so the overflow / underflow is implicitly allowed

but in the current real wagmi codebase use compiler version 0.8.18 and does not allow overflow / underflow

based on the issue 573 discussion, the implicit overflow is needed to correctly calculated the fee.

so a valid issue

as for whether the severity is medium or low

https://github.com/Uniswap/v3-core/issues/573#issue-1342090896

I think the situation above is not a uncommon case, so recommend severity is medium

unless there are further comments from both side!

hrishibhat

@huuducsc Do you agree with the above comments?

hrishibhat

@n1punp https://github.com/Jeiwan/uniswapv3-book/issues/45#issuecomment-1445305623 Do you have any comments on this?

n1punp

It's as Jeiwan's commented -- the subtraction should expect underflow (so this will be an issue for solidity >= 0.8.x), so this is a valid issue.

> @n1punp Jeiwan/uniswapv3-book#45 (comment) Do you have any comments on this?

hrishibhat

@n1punp Thanks for clarifying. Yes, that makes sense.
Sorry if I'm missing something. I don't see reverting during a deposit as a major issue. but the revert during withdrawal can be a problem. Do you have a valid example with numbers for the possibility?

Or is this example applicable in this case?
https://github.com/Uniswap/v3-core/issues/573#issue-1342090896

n1punp

@hrishibhat Not sure about the specific case where user can deposit but cannot withdraw. It may require very precise calculations to find it out, which could not be worth the time (since allowing underflow should be the answer anyways), but having the core functionality unavailable should give this issue a valid Medium at least. I'm fine with it being valid Medium (as in #1).

(if anyone can somehow prove that it can deposit but cannot withdraw, then it could be a valid High).

hrishibhat

Result: Medium Has duplicates Considering this a valid medium based on the above comments

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- huuducsc: accepted

Issue M-2: The fees are incorrectly updated in the `_deposit` and `_withdraw` functions, which allows the attacker to break the protocol fees

Source: https://github.com/sherlock-audit/2023-06-real-wagmi-judging/issues/52

Found by

duc

Summary

In `_deposit` and `_withdraw` function of Multipool contract, `fee0` and `fee1` are calculated from the accrued fee amounts to update feeGrowth of contract and send the protocol fee to the owner. However, these calculations are incorrect and do not align with the collected fees from the positions. It allows attacker break the protocol fees.

Vulnerability Detail

In function `withdraw`, all accrued fees from Uniswap V3 pool will be collected (`tokensOwed0After` and `tokensOwed1After`). However, `fee0` and `fee1` are updated by `tokensOwed0After - tokensOwed0Before` and `tokensOwed1After - tokensOwed1Before`.

It is incorrect because in cases where `tokensOwed0Before > 0` or `tokensOwed1Before > 0`, `fee0` and `fee1` end up being smaller than the collected fees. Consequently, when `_upFeesGrowth` is executed, it will send smaller protocol fees than expected.

Therefore, an attacker can exploit this vulnerability and undermine the protocol fees by minting minuscule amounts to all positions within the Multipool contract. This is possible since anyone can mint for any positions in UniSwapV3 pools (the owner of position is recipient). This causes the fees of each position to be accrued beforehand, resulting in `tokensOwed0Before` and `tokensOwed1Before` becoming close to `tokensOwed0After` and `tokensOwed1After`. Then the majority of protocol fees will be lost (solvent in the reserve).

Scenario:

1. Attacker mints minuscule amounts to all positions within the Multipool contract to accrue fees for these positions.
2. The attacker then calls the `earn()` function to collect and update fees. In this scenario, `tokensOwed0Before` and `tokensOwed1Before` become equal to `tokensOwed0After` and `tokensOwed1After` (withdrawing 0 liquidity). As a result, `fee0` and `fee1` become 0, leading to the loss of these protocol fees.

Similar to `_deposit` function.

Impact

Protocol will lose its fees.

Code Snippet

https://github.com/sherlock-audit/2023-06-real-wagmi/blob/main/concentrator/contracts/Multipool.sol#L509-L536 https://github.com/sherlock-audit/2023-06-real-wagmi/blob/main/concentrator/contracts/Multipool.sol#L381-L406

Tool used

Manual review

Recommendation

`fee0` and `fee1` in `_deposit` and `_withdraw` function should be equal to the collected fee amounts (`tokensOwed0After` and `tokensOwed1After`)

Discussion

huuducsc

Escalate

This issue pertains to the incorrect updating of fees, resulting in a significant loss of the protocol fee. The explanation is clear enough if you have knowledge about UniswapV3. Here are my detailed explanations:

1. The fee from the UniswapV3 pool is accumulated into the data of a position when that position is updated (by adding liquidity or removing liquidity ...). This allows anyone to mint minuscule amounts for the positions of Multipool to accrue the fee for them into position data. https://github.com/Uniswap/v3-core/blob/main/contracts/UniswapV3Pool.sol#L442 https://github.com/Uniswap/v3-core/blob/main/contracts/libraries/Position.sol#L79-L85
2. In the function _withdraw of the Multipool contract, `tokensOwed0Before` and `tokensOwed1Before` are the current collectible amounts of this position (the current accrued fee). `amount0` and `amount1` are the amount from burning liquidity. Therefore, `(tokensOwed0After - amount0) - tokensOwed0Before` is only equal to the fee from the burn call (which is very small) and doesn't include the accrued fee mentioned above. However, the contract has collected all the collectible amounts of each position, which means the accrued fee has been collected but is not accumulated in `fee0` and `fee1`.

3. The `_withdraw` function will trigger _upFeesGrowth function with incorrect `fee0` and `fee1`. They don't include the accrued fee of this position before. This will result in the protocol losing a significant protocol fee.

sherlock-admin

> Escalate
>
> This issue pertains to the incorrect updating of fees, resulting in a significant loss of the protocol fee. The explanation is clear enough if you have knowledge about UniswapV3. Here are my detailed explanations:
>
> 1. The fee from the UniswapV3 pool is accumulated into the data of a position when that position is updated (by adding liquidity or removing liquidity ...). This allows anyone to mint minuscule amounts for the positions of Multipool to accrue the fee for them into position data. https://github.com/Uniswap/v3-core/blob/main/contracts/UniswapV3Pool.sol#L442 https://github.com/Uniswap/v3-core/blob/main/contracts/libraries/Position.sol#L79-L85
> 2. In the function _withdraw of the Multipool contract, `tokensOwed0Before` and `tokensOwed1Before` are the current collectible amounts of this position (the current accrued fee). `amount0` and `amount1` are the amount from burning liquidity. Therefore, `(tokensOwed0After - amount0) - tokensOwed0Before` is only equal to the fee from the burn call (which is very small) and doesn't include the accrued fee mentioned above. However, the contract has collected all the collectible amounts of each position, which means the accrued fee has been collected but is not accumulated in `fee0` and `fee1`.
> 3. The `_withdraw` function will trigger _upFeesGrowth function with incorrect `fee0` and `fee1`. They don't include the accrued fee of this position before. This will result in the protocol losing a significant protocol fee.

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.

ctf-sec

This seems like a medium, recommend bring sponsor for review

hrishibhat

@huuducsc Sponsor comment:

> It doesn't make sense and does not matter what small position the user will mint. The Uniswap positions are common for all users. The Uniswap positions are owned by the multipool and fees will be collected on the entire liquidity of the multitool for each withdrawal or deposit.

huuducsc

> @huuducsc Sponsor comment:
>
> > It doesn't make sense and does not matter what small position the user will mint. The Uniswap positions are common for all users. The Uniswap positions are owned by the multipool and fees will be collected on the entire liquidity of the multitool for each withdrawal or deposit.

I agree that fees will be collected on the entire liquidity of the multitool for each withdrawal or deposit, but it is not related to this issue. This issue is about the protocol fee, which is derived from the total fees collected from Multipool's positions, and it may be compromised. The protocol fee is only charged in the `_deposit` and `_withdraw` functions. If the fees from Multipool's positions were to be collected without using `_deposit` and `_withdraw` (for example, by adding a small amount of liquidity directly), the protocol fee would not increase during the next `_deposit` or `_withdraw` action. As a result, an attacker would be able to manipulate the protocol fee by front-running these functions.

fann95

FIXED: https://github.com/RealWagmi/concentrator/commit/9036fe43b0de2694a0c5e18cfe821d40ce17a9b8

hrishibhat

Result: Medium Unique Considering this issue a valid medium

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- huuducsc: accepted

0xffff11

Fixed by deleting the subtraction of `tokensOwed0Before` and accounting for the collected fee amounts

Issue M-3: Possible precision loss in `_checkpositionsRange` function

Source: https://github.com/sherlock-audit/2023-06-real-wagmi-judging/issues/82

Found by

0xdice91, BugBusters, ash, ginlee, josephdara, lil.eth, shealtielanz, tsvetanovv, twcctop

Summary

The `_checkpositionsRange` function contains a potential precision loss issue. The problem arises from the expression ((_range / 2) % _tickSpacing != 0).

Vulnerability Detail

The vulnerability lies in the calculation of `(_range / 2) % _tickSpacing`. The code assumes that _range and _tickSpacing are both int24 variables, representing signed 24-bit integers. However, when `_range` is 9 and `_tickSpacing` is, for example, 3, the division operation (_range / 2) will truncate the fractional part, resulting in 4 instead of 4.5. Subsequently, the modulo operation (4 % _tickSpacing) will evaluate to 1. This can lead to incorrect comparisons and unexpected behavior.

Impact

It can potentially lead to incorrect validation and can also affect the correctness of the positions range check, potentially allowing invalid positions to be considered valid

Code Snippet

```
function _checkpositionsRange(
        int24 _range,
        int24 _tickSpacing,
        int24 tickSpacingOffset
    ) private pure {
        ErrLib.requirement(
            (_range > _tickSpacing) &&
                (_range % _tickSpacing == 0) &&
                (((_range / 2) % _tickSpacing != 0) || tickSpacingOffset != 0),
            ErrLib.ErrorCode.INVALID_POSITIONS_RANGE
        );
    }
```

https://github.com/sherlock-audit/2023-06-real-wagmi/blob/82a234a5c2c1fc1921c63265a9349b71d84675c4/concentrator/contracts/MultiStrategy.sol#L93-L104

Tool used

Manual Review

Recommendation

Discussion

0xffff11

Seems like a valid medium to me. Reference from tickSpacing: https://docs.uniswap.org/contracts/v3/reference/core/libraries/Tick

fann95

FIXED https://github.com/RealWagmi/concentrator/blob/fbccf1caf28edbb23db8c7e0409d0c40c3a56461/contracts/MultiStrategy.sol#L98

0xffff11

> FIXED https://github.com/RealWagmi/concentrator/blob/fbccf1caf28edbb23db8c7e0409d0c40c3a56461/contracts/MultiStrategy.sol#L98

This link seems wrong, it does not bring me to a PR, please check

0xffff11

> FIXED https://github.com/RealWagmi/concentrator/blob/fbccf1caf28edbb23db8c7e0409d0c40c3a56461/contracts/MultiStrategy.sol#L98

Do not completely understand the fix. No comments were made about it, the only thing I could notice by comparing code from different commits, is that `(_range % 2 == 0)` was removed. @fann95

Issue M-4: The deposit - withdraw - trade transaction lack of expiration timestamp check (DeadLine check)

Source: https://github.com/sherlock-audit/2023-06-real-wagmi-judging/issues/163

Found by

Avci, Phantasmagoria, kutugu, sashik_eth, shealtielanz

Summary

The deposit - withdraw - trade transaction lack of expiration timestamp check (DeadLine check)

Vulnerability Detail

the protocol missing the DEADLINE check at all in logic.

this is actually how uniswap implemented the **Deadline**, this protocol also need deadline check like this logic

https://github.com/Uniswap/v2-periphery/blob/0335e8f7e1bd1e8d8329fd300aea2ef2f36dd19f/contracts/UniswapV2Router02.sol#L61

```
// **** ADD LIQUIDITY ****
function _addLiquidity(
    address tokenA,
    address tokenB,
    uint amountADesired,
    uint amountBDesired,
    uint amountAMin,
    uint amountBMin
) internal virtual returns (uint amountA, uint amountB) {
    // create the pair if it doesn't exist yet
    if (IUniswapV2Factory(factory).getPair(tokenA, tokenB) == address(0)) {
        IUniswapV2Factory(factory).createPair(tokenA, tokenB);
    }
    (uint reserveA, uint reserveB) = UniswapV2Library.getReserves(factory,
↪   tokenA, tokenB);
    if (reserveA == 0 && reserveB == 0) {
        (amountA, amountB) = (amountADesired, amountBDesired);
    } else {
        uint amountBOptimal = UniswapV2Library.quote(amountADesired, reserveA,
↪   reserveB);
        if (amountBOptimal <= amountBDesired) {
```

```
            require(amountBOptimal >= amountBMin, 'UniswapV2Router:
↪   INSUFFICIENT_B_AMOUNT');
            (amountA, amountB) = (amountADesired, amountBOptimal);
        } else {
            uint amountAOptimal = UniswapV2Library.quote(amountBDesired,
↪   reserveB, reserveA);
            assert(amountAOptimal <= amountADesired);
            require(amountAOptimal >= amountAMin, 'UniswapV2Router:
↪   INSUFFICIENT_A_AMOUNT');
            (amountA, amountB) = (amountAOptimal, amountBDesired);
        }
    }
}

function addLiquidity(
    address tokenA,
    address tokenB,
    uint amountADesired,
    uint amountBDesired,
    uint amountAMin,
    uint amountBMin,
    address to,
    uint deadline
) external virtual override ensure(deadline) returns (uint amountA, uint
↪   amountB, uint liquidity) {
    (amountA, amountB) = _addLiquidity(tokenA, tokenB, amountADesired,
↪   amountBDesired, amountAMin, amountBMin);
    address pair = UniswapV2Library.pairFor(factory, tokenA, tokenB);
    TransferHelper.safeTransferFrom(tokenA, msg.sender, pair, amountA);
    TransferHelper.safeTransferFrom(tokenB, msg.sender, pair, amountB);
    liquidity = IUniswapV2Pair(pair).mint(to);
}
```

the point is the deadline check

```
modifier ensure(uint deadline) {
    require(deadline >= block.timestamp, 'UniswapV2Router: EXPIRED');
    _;
}
```

The deadline check ensure that the transaction can be executed on time and the expired transaction revert.

Impact

The transaction can be pending in mempool for a long and the trading activity is very time senstive. Without deadline check, the trade transaction can be executed

in a long time after the user submit the transaction, at that time, the trade can be done in a sub-optimal price, which harms user's position.

The deadline check ensure that the transaction can be executed on time and the expired transaction revert.

Code Snippet

```
function _deposit(
       uint256 amount0Desired,
       uint256 amount1Desired,
       uint256 _totalSupply,
       SlotOData[] memory slots
   ) private {
       uint128 liquidity;
       uint256 fee0;
       uint256 fee1;
       uint256 posNum = multiPosition.length;
       PositionInfo memory position;
       for (uint256 i = 0; i < posNum; ) {
           position = multiPosition[i];

           liquidity = _calcLiquidityAmountToDeposit(
               slots[i].currentSqrtRatioX96,
               position,
               amount0Desired,
               amount1Desired
           );
           if (liquidity > 0) {
               (, , , uint128 tokensOwed0Before, uint128 tokensOwed1Before) =
↪  IUniswapV3Pool(
                   position.poolAddress
               ).positions(position.positionKey);

               IUniswapV3Pool(position.poolAddress).mint(
                   address(this), //recipient
                   position.lowerTick,
                   position.upperTick,
                   liquidity,
                   abi.encode(position.poolFeeAmt)
               );

               (, , , uint128 tokensOwed0After, uint128 tokensOwed1After) =
↪  IUniswapV3Pool(
                   position.poolAddress
               ).positions(position.positionKey);
```

```
                fee0 += tokensOwed0After - tokensOwed0Before;
                fee1 += tokensOwed1After - tokensOwed1Before;

                IUniswapV3Pool(position.poolAddress).collect(
                    address(this),
                    position.lowerTick,
                    position.upperTick,
                    type(uint128).max,
                    type(uint128).max
                );
            }
            unchecked {
                ++i;
            }
        }
```

https://github.com/sherlock-audit/2023-06-real-wagmi/blob/main/concentrator/contracts/Multipool.sol#L360 https://github.com/sherlock-audit/2023-06-real-wagmi/blob/main/concentrator/contracts/Multipool.sol#L433 https://github.com/sherlock-audit/2023-06-real-wagmi/blob/main/concentrator/contracts/Multipool.sol#L557 https://github.com/sherlock-audit/2023-06-real-wagmi/blob/main/concentrator/contracts/Dispatcher.sol#L180

Tool used

Manual Review

Recommendation

- consider adding deadline check like in the functions like withdraw and deposit and all operations the point is the deadline check

```
modifier ensure(uint deadline) {
    require(deadline >= block.timestamp, 'UniswapV2Router: EXPIRED');
    _;
}
```

Discussion

ctf-sec

Seems like the sponsor mark https://github.com/sherlock-audit/2023-06-real-wagmi-judging/issues/102 as medium and dispute #163

#116 is a duplicate of #102 as well,

I understand the sponsor is mainly concerning with rebalance action.

Recommend still leave this issue as a medium

0xffff11

Agree to keep a med here

fann95

Our team does not consider such a check necessary and allows for a delay in deposit and withdrawals subject to the conditions of slippage protection.The deadline check will be included in swapData if needed