



SHERLOCK

SHERLOCK SECURITY REVIEW FOR



Prepared for:

Tokemak

Prepared by:

Sherlock

Lead Security Expert: **xiaoming90**

Dates Audited:

July 17 - August 29, 2023

Prepared on:

November 28, 2023

Introduction

Generating sustainable liquidity for the tokenized world. Eliminating inefficiencies and helping LPs to deploy liquidity where it can do the most work is exactly why we are building Tokemak v2!

Scope

Repository: Tokemak/v2-core-audit-2023-07-14

Branch: audit-start

Commit: 62445b8ee3365611534c96aef189642b721693bf

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
19	16

Security experts who found valid issues

[xiaoming90](#)
[0x007](#)
[Kalyan-Singh](#)
[ctf_sec](#)
[berndartmueller](#)
[lemonmon](#)
[0xVolodya](#)
[saidam017](#)

[nobody2018](#)
[Vagner](#)
[Ch_301](#)
[duc](#)
[Aymen0909](#)
[carrotsmugger](#)
[0x73696d616f](#)
[Flora](#)

[ck](#)
[bin2chen](#)
[rvierdiev](#)
[n33k](#)
[talfao](#)
[Bauchibred](#)
[caelumimperium](#)
[0xvj](#)



TangYuanShen
enfrasco
0xWeiss
warRoom
p0wd3r
shaka
pengun
0xdeadbeef
0xGoodess
chaduke
hassan-truscova
lucifero
ADM
pks_
minhtrng
tives
0xbepresent

bitsurfer
BPZ
I3r0ux
wangxx2026
0xSurena
0xJuda
Phantasmagoria
dipp
AuditorPraise
0x70C9
techOptimizzor
KingNFT
ast3ros
MrjoryStewartBaxter
0xTheC0der
lil.eth
Angry_Mustache_Man

VAD37
Nadin
BTK
jecikpo
bulej93
1nc0gn170
0xDjango
0x3b
lodelux
0xmuxyz
Breeje
SaharDevep
harisnabeel
vagrant
asui
0xComfyCat
shogoki



Issue H-1: ETH deposited by the user may be stolen.

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/1>

Found by

0x007, 0x3b, 0xComfyCat, 0xDjango, 0xJuda, 0xSurena, 0xbepresent, 0xdeadbeef, 0xmuxyz, 0xvj, Breeje, Flora, SaharDevep, TangYuanShen, VAD37, asui, berndartmueller, bin2chen, caelumimperium, chaduke, ck, duc, enfrasico, harisnabeel, lemonmon, lodelux, n33k, nobody2018, p0wd3r, pengun, rvierdiiev, saidam017, shogoki, talfao, vagrant, warRoom, xiaoming90 Due to the fact that the WETH obtained through `_processEthIn` belongs to the contract, and `pullToken` transfers assets from `msg.sender`, it is possible for users to transfer excess WETH to the contract, allowing attackers to steal WETH from within the contract using `sweepToken`.

Both `mint` and `deposit` in `LMPVaultRouterBase` have this problem.

Vulnerability Detail

In the `deposit` function, if the user pays with ETH, it will first call `_processEthIn` to wrap it and then call `pullToken` to transfer.

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVaultRouterBase.sol#L43-L57>

```
/// @inheritdoc ILMPVaultRouterBase
function deposit(
    ILMPVault vault,
    address to,
    uint256 amount,
    uint256 minSharesOut
) public payable virtual override returns (uint256 sharesOut) {
    // handle possible eth
    _processEthIn(vault);

    IERC20 vaultAsset = IERC20(vault.asset());
    pullToken(vaultAsset, amount, address(this));

    return _deposit(vault, to, amount, minSharesOut);
}
```

`_processEthIn` will wrap ETH into WETH, and these WETH belong to the contract itself.



<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVaultRouterBase.sol#L111-L122>

```
function _processEthIn(ILMPVault vault) internal {
    // if any eth sent, wrap it first
    if (msg.value > 0) {
        // if asset is not weth, revert
        if (address(vault.asset()) != address(weth9)) {
            revert InvalidAsset();
        }

        // wrap eth
        weth9.deposit{ value: msg.value }();
    }
}
```

However, `pullToken` transfers from `msg.sender` and does not use the WETH obtained in `_processEthIn`.

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/utis/PeripheryPayments.sol#L54-L56>

```
function pullToken(ERC20 token, uint256 amount, address recipient) public
↳ payable {
    token.safeTransferFrom(msg.sender, recipient, amount);
}
```

If the user deposits 10 ETH and approves 10 WETH to the contract, when the deposit amount is 10, all of the user's 20 WETH will be transferred into the contract.

However, due to the `amount` being 10, only 10 WETH will be deposited into the vault, and the remaining 10 WETH can be stolen by the attacker using `sweepToken`.

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/utis/PeripheryPayments.sol#L58-L65>

```
function sweepToken(ERC20 token, uint256 amountMinimum, address recipient)
↳ public payable {
    uint256 balanceToken = token.balanceOf(address(this));
    if (balanceToken < amountMinimum) revert InsufficientToken();

    if (balanceToken > 0) {
        token.safeTransfer(recipient, balanceToken);
    }
}
```

Both `mint` and `deposit` in `LMPVaultRouterBase` have this problem.



Impact

ETH deposited by the user may be stolen.

Code Snippet

- <https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVaultRouterBase.sol#L43-L57>
- <https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/utils/PeripheryPayments.sol#L54-L56>
- <https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/utils/PeripheryPayments.sol#L58-L65>

Tool used

Manual Review

Recommendation

Perform operations based on the size of `msg.value` and `amount`:

1. `msg.value == amount`: transfer WETH from contract not `msg.sender`
2. `msg.value > amount`: transfer WETH from contract not `msg.sender` and refund to `msg.sender`
3. `msg.value < amount`: transfer WETH from contract and transfer remaining from `msg.sender`



Issue H-2: Destination Vault rewards are not added to idleIncrease when info.totalAssetsPulled > info.totalAssetsToPull

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/5>

Found by

0x73696d616f, 0xbepresent, Aymen0909, Ch_301, Kalyan-Singh, TangYuanShen, berndartmueller, bin2chen, bitsurfer, carrotsmugger, duc, lemonmon, nobody2018, p0wd3r, penguin, rvierdiev, saidam017, talfao, warRoom, xiaoming90 Destination Vault rewards are not added to idleIncrease when info.totalAssetsPulled > info.totalAssetsToPull in _withdraw of LMPVault.

This result in rewards not being recorded by LMPVault and ultimately frozen in the contract.

Vulnerability Detail

In the _withdraw function, Destination Vault rewards will be first recorded in info.IdleIncrease by info.idleIncrease += _baseAsset.balanceOf(address(this)) - assetPreBal - assetPulled;.

But when info.totalAssetsPulled > info.totalAssetsToPull, info.idleIncrease is directly assigned as info.totalAssetsPulled - info.totalAssetsToPull, and info.totalAssetsPulled is assetPulled without considering Destination Vault rewards.

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L482-L497>

```
uint256 assetPreBal = _baseAsset.balanceOf(address(this));
uint256 assetPulled = destVault.withdrawBaseAsset(sharesToBurn, address(this));

// Destination Vault rewards will be transferred to us as part of burning out
↳ shares
// Back into what that amount is and make sure it gets into idle
info.idleIncrease += _baseAsset.balanceOf(address(this)) - assetPreBal -
↳ assetPulled;
info.totalAssetsPulled += assetPulled;
info.debtDecrease += totalDebtBurn;

// It's possible we'll get back more assets than we anticipate from a swap
// so if we do, throw it in idle and stop processing. You don't get more than
↳ we've calculated
```



```
if (info.totalAssetsPulled > info.totalAssetsToPull) {
    info.idleIncrease = info.totalAssetsPulled - info.totalAssetsToPull;
    info.totalAssetsPulled = info.totalAssetsToPull;
    break;
}
```

For example,

```
// preBal == 100 pulled == 10 reward == 5 toPull == 6
// idleIncrease = 115 - 100 - 10 == 5
// totalPulled(0) += assetPulled == 10 > toPull
// idleIncrease = totalPulled - toPull == 4 < reward
```

The final `info.idleIncrease` does not record the reward, and these assets are not ultimately recorded by the Vault.

Impact

The final `info.idleIncrease` does not record the reward, and these assets are not ultimately recorded by the Vault.

Meanwhile, due to the `recover` function's inability to extract the `baseAsset`, this will result in no operations being able to handle these Destination Vault rewards, ultimately causing these assets to be frozen within the contract.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L482-L497>

Tool used

Manual Review

Recommendation

```
info.idleIncrease = info.totalAssetsPulled - info.totalAssetsToPull; ->
info.idleIncrease += info.totalAssetsPulled - info.totalAssetsToPull;
```



Issue H-3: Liquidations miss delegate call to swapper

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/205>

Found by

0x007, 0x3b, 0x73696d616f, 0xDjango, 0xJuda, 0xSurena, 0xTheC0der, 0xbepresent, 0xvj, ADM, Angry_Mustache_Man, Ch_301, Kalyan-Singh, MrjoryStewartBaxter, berndartmueller, bin2chen, duc, lil.eth, lodelux, nobody2018, p0wd3r, pengun, rvierdiev, saidam017, shaka, talfao, xiaoming90

LiquidationRow acts as an orchestrator of claiming process. It liquidates tokens across vaults using the **liquidateVaultsForToken** function. This function has a flaw and will revert. Swapper contract is called during the function call, but tokens are not transferred to it nor tokens are transferred back from the swapper to the calling contract. Based on other parts of the codebase the problem is that swapper should be invoked with a low-level delegatecall instead of a normal call.

Vulnerability Detail

The LiquidationRow contract is an orchestrator for the claiming process. It is primarily used to collect rewards for vaults. It has a method called **liquidateVaultsForToken**. Based on docs this method is for: *Conducts the liquidation process for a specific token across a list of vaults, performing the necessary balance adjustments, initiating the swap process via the asyncSwapper, taking a fee from the received amount, and queues the remaining swapped tokens in the MainRewarder associated with each vault.*

```
function liquidateVaultsForToken(
    address fromToken,
    address asyncSwapper,
    IDestinationVault[] memory vaultsToLiquidate,
    SwapParams memory params
) external nonReentrant hasRole(Roles.LIQUIDATOR_ROLE)
↳ onlyWhitelistedSwapper(asyncSwapper) {
    uint256 gasBefore = gasleft();

    (uint256 totalBalanceToLiquidate, uint256[] memory vaultsBalances) =
        _prepareForLiquidation(fromToken, vaultsToLiquidate);
    _performLiquidation(
        gasBefore, fromToken, asyncSwapper, vaultsToLiquidate, params,
↳ totalBalanceToLiquidate, vaultsBalances
    );
}
```



<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/liquidation/LiquidationRow.sol#L167C5-L180C6>

The second part of the function is performing the liquidation by calling **_performLiquidation**. A problem is at the beginning of it. IAsyncSwapper is called to swap tokens.

```
function _performLiquidation(
    uint256 gasBefore,
    address fromToken,
    address asyncSwapper,
    IDestinationVault[] memory vaultsToLiquidate,
    SwapParams memory params,
    uint256 totalBalanceToLiquidate,
    uint256[] memory vaultsBalances
) private {
    uint256 length = vaultsToLiquidate.length;
    // the swapper checks that the amount received is greater or equal than the
    ↪ params.buyAmount
    uint256 amountReceived = IAsyncSwapper(asyncSwapper).swap(params);
    // ...
}
```

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/liquidation/LiquidationRow.sol#L251C8-L251C75>

As you can see the LiquidationRow doesn't transfer the tokens to swapper and swapper doesn't pul them either ([swap function here](#)). Because of this the function reverses.

I noticed that there is no transfer back to LiquidationRow from Swapper either. Tokens can't get in or out.

When I searched the codebase, I found that Swapper is being called on another place using the delegatecall method. This way it can operate with the tokens of the caller. The call can be found here - [LMPVaultRouter.sol:swapAndDepositToVault](#). So I think that instead of missing transfer, the problem is actually in the way how swapper is called.

Impact

Rewards collected through LiquidationRow **claimsVaultRewards** get stuck in the contract. Liquidation can't be called because it reverts when Swapper tries to work with tokens it doesn't possess.



Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/liquidation/LiquidationRow.sol#L167C5-L180C6>

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/liquidation/LiquidationRow.sol#L251C8-L251C75>

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/liquidation/BaseAsyncSwapper.sol#L19C5-L64C6>

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVaultRouter.sol#L53C8-L55C11>

Tool used

Manual Review

Recommendation

Change the async swapper call from the normal function call to the low-level delegatecall function the same way it is done in LMPVaultRouter.sol:swapAndDepositToVault.

I would like to address that AsyncSwapperMock in LiquidationRow.t.sol is a poorly written mock and should be updated to represent how the AsyncSwapper work. It would be nice to update the test suite for LiquidationRow because its current state won't catch this. If you check the LiquidationRow.t.sol tests, the mock swap function only mints tokens, no need to use delegatecall. This is why tests missed this vulnerability.



Issue H-4: When `queueNewRewards` is called, caller could transfer tokens more than it should be

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/379>

Found by

0xVolodya, 0xbepresent, 0xvj, 1nc0gn170, Angry_Mustache_Man, Aymen0909, BPZ, Kalyan-Singh, berndartmueller, bin2chen, bitsurfer, bulej93, caelumimperium, chaduke, duc, l3r0ux, lemonmon, lil.eth, p0wd3r, penguin, saidam017, shaka, wangxx2026, xiaoming90

`queueNewRewards` is used for Queues the specified amount of new rewards for distribution to stakers. However, it used wrong calculated value when pulling token funds from the caller, could make caller transfer tokens more that it should be.

Vulnerability Detail

Inside `queueNewRewards`, irrespective of whether we're near the start or the end of a reward period, if the accrued rewards are too large relative to the new rewards (`queuedRatio` is greater than `newRewardRatio`), the new rewards will be added to the queue (`queuedRewards`) rather than being immediately distributed.

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/rewarders/AbstractRewarder.sol#L235-L261>

```
function queueNewRewards(uint256 newRewards) external onlyWhitelisted {
    uint256 startingQueuedRewards = queuedRewards;
    uint256 startingNewRewards = newRewards;

    newRewards += startingQueuedRewards;

    if (block.number >= periodInBlockFinish) {
        notifyRewardAmount(newRewards);
        queuedRewards = 0;
    } else {
        uint256 elapsedBlock = block.number - (periodInBlockFinish -
        ↪ durationInBlock);
        uint256 currentAtNow = rewardRate * elapsedBlock;
        uint256 queuedRatio = currentAtNow * 1000 / newRewards;

        if (queuedRatio < newRewardRatio) {
            notifyRewardAmount(newRewards);
            queuedRewards = 0;
        } else {
            queuedRewards = newRewards;
        }
    }
}
```



```

    }
}

emit QueuedRewardsUpdated(startingQueuedRewards, startingNewRewards,
↳ queuedRewards);

// Transfer the new rewards from the caller to this contract.
IERC20(rewardToken).safeTransferFrom(msg.sender, address(this), newRewards);
}

```

However, when this function tried to pull funds from sender via `safeTransferFrom`, it used `newRewards` amount, which already added by `startingQueuedRewards`. If previously `queuedRewards` already have value, the processed amount will be wrong.

Impact

There are two possible issue here :

1. If previously `queuedRewards` is not 0, and the caller don't have enough funds or approval, the call will revert due to this logic error.
2. If previously `queuedRewards` is not 0, and the caller have enough funds and approval, the caller funds will be pulled more than it should (`reward param + queuedRewards`)

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/rewarders/AbstractRewarder.sol#L236-L239>

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/rewarders/AbstractRewarder.sol#L260>

Tool used

Manual Review

Recommendation

Update the transfer to use `startingNewRewards` instead of `newRewards` :

```

function queueNewRewards(uint256 newRewards) external onlyWhitelisted {
    uint256 startingQueuedRewards = queuedRewards;
    uint256 startingNewRewards = newRewards;

    newRewards += startingQueuedRewards;
}

```



```

        if (block.number >= periodInBlockFinish) {
            notifyRewardAmount(newRewards);
            queuedRewards = 0;
        } else {
            uint256 elapsedBlock = block.number - (periodInBlockFinish -
↳ durationInBlock);
            uint256 currentAtNow = rewardRate * elapsedBlock;
            uint256 queuedRatio = currentAtNow * 1000 / newRewards;

            if (queuedRatio < newRewardRatio) {
                notifyRewardAmount(newRewards);
                queuedRewards = 0;
            } else {
                queuedRewards = newRewards;
            }
        }

        emit QueuedRewardsUpdated(startingQueuedRewards, startingNewRewards,
↳ queuedRewards);

        // Transfer the new rewards from the caller to this contract.
-        IERC20(rewardToken).safeTransferFrom(msg.sender, address(this),
↳ newRewards);
+        IERC20(rewardToken).safeTransferFrom(msg.sender, address(this),
↳ startingNewRewards);
    }

```



Issue H-5: Curve V2 Vaults can be drained because CurveV2CryptoEthOracle can be reentered with WETH tokens

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/481>

Found by

0x007, 0xVolodya, Kalyan-Singh CurveV2CryptoEthOracle assumes that Curve pools that could be reentered must have

0xEeeeeEeeeEeEeeEeEeEeeEEEEEEEEEEEEEEEEEEEEEEEE token. But this is a wrong assumption cause tokens with WETH token could be reentered too.

Vulnerability Detail

CurveV2CryptoEthOracle.registerPool takes checkReentrancy parameters and this should be True only for pools that have

0xEeeeeEeeeEeEeeEeEeEeeEEEEEEEEEEEEEEEEEEEEEEEE tokens and this is validated [here](#).

```
address public constant ETH = 0xEeeeeEeeeEeEeeEeEeEeeEEEEEEEEEEEEEEEEEEEEEEEE;

...

// Only need ability to check for read-only reentrancy for pools containing
↳ native Eth.
if (checkReentrancy) {
    if (tokens[0] != ETH && tokens[1] != ETH) revert MustHaveEthForReentrancy();
}
```

This Oracle is meant for Curve V2 pools and the ones I've seen so far use WETH address instead of 0xEeeeeEeeeEeEeeEeEeEeeEEEEEEEEEEEEEEEEEEEEEEEE (like Curve V1) and this applies to all pools listed by Tokemak.

For illustration, I'll use the same pool used to [test proper registration](#). The test is for CRV_ETH_CURVE_V2_POOL but this applies to other V2 pools including [rETH/ETH](#). The pool address for CRV_ETH_CURVE_V2_POOL is 0x8301AE4fc9c624d1D396cbDAa1ed877821D7C511 while token address is 0xEEd4064f376cB8d68F770FB1Ff088a3d0F3FF5c4d.

If you interact with the pool, the coins are: 0 - WETH - 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2 1 - CRV - 0xD533a949740bb3306d119CC777fa900bA034cd52

So how can WETH be reentered?! Because Curve can accept ETH for WETH pools.



A look at the pool again shows that Curve uses python kwargs and it includes a variable `use_eth` for exchange, `add_liquidity`, `remove_liquidity` and `remove_liquidity_one_coin`.

When `use_eth` is true, it would take `msg.value` instead of transfer WETH from user. And it would make a raw call instead of transfer WETH to user.

If raw call is sent to user, then they could reenter LMP vault and attack the protocol and it would be successful cause CurveV2CryptoEthOracle would not check for reentrancy in `getPriceInEth`

```
// Checking for read only reentrancy scenario.
if (poolInfo.checkReentrancy == 1) {
    // This will fail in a reentrancy situation.
    cryptoPool.claim_admin_fees();
}
```

A profitable attack that could be used to drain the vault involves

- Deposit shares at fair price
- Remove liquidity on Curve and `updateDebtReporting` in LMPVault with view only reentrancy
- Withdraw shares at unfair price

Impact

The protocol could be attacked with price manipulation using Curve read only reentrancy. The consequence would be fatal because `getPriceInEth` is used for evaluating `debtValue` and this evaluation decides shares and debt that would be burned in a withdrawal. Therefore, an inflated value allows attacker to withdraw too many asset for their shares. This could be abused to drain assets on LMPVault.

The attack is cheap, easy and could be bundled in as a flashloan attack. And it puts the whole protocol at risk cause a large portion of their deposit would be on Curve V2 pools with WETH token.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/oracles/providers/CurveV2CryptoEthOracle.sol#L121-L123>
<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/oracles/providers/CurveV2CryptoEthOracle.sol#L160-L163>
<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/test/oracles/providers/CurveV2CryptoEthOracle.t.sol#L126-L136> <https://etherscan.io/address/0x8301AE4fc9c624d1D396cbDAa1ed877821D7C511#code>



Tool used

Manual Review

Recommendation

If CurveV2CryptoEthOracle is meant for CurveV2 pools with WETH (and no 0xEeeeeEeeeEeEeeEeEeEEEEEEEEEEEEEEEEEEEE), then change the ETH address to weth. As far as I can tell Curve V2 uses WETH address for ETH but this needs to be verified.

```
-   if (tokens[0] != ETH && tokens[1] != ETH) revert MustHaveEthForReentrancy();
+   if (tokens[0] != WETH && tokens[1] != WETH) revert
↪   MustHaveEthForReentrancy();
```



Issue H-6: updateDebtReporting can be front run, putting all the loss on later withdrawals but taking the profit

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/531>

Found by

Kalyan-Singh, berndartmueller, lemonmon

Summary

updateDebtReporting takes in a **user input** of destinations in array whose debt to report, so if a destination vault is incurring loss and is not on the front of withdrawalQueue than a attacker can just update debt for only the destination which are incurring a profit and withdraw in the same txn. He will exit the vault with profit, others who withdraw after the legit updateDebtReporting txn will suffer even more loss than they should have, as some part of the profit which was used to offset the loss was taken by the attacker and protocol fees

Vulnerability Detail

POC-

1. LMPVault has 2000 in deposits 1000 from alice and 1000 from bob
2. Vault has invested that in 1000 in DestinationVault1 & 1000 in DestinationVault2 (no idle for simple calculations)
3. Now Dv1 gain a profit of 5%(+50 base asset) while Dv2 is in 10% loss(-100 base asset)
4. So vault has net loss of 50. Now alice does a updateDebtReporting([Dv1]) and not including Dv2 in the input array.
5. Now she withdraws her money, protocol now falsely believes there is a profit, it also take 20% profit fees(assumed) and mints 10 shares for itself and alice walks away with roughly 1020 assets, incurring no loss
6. Now a legit updateDebtReporting txn comes and bob has to account in for the loss

Test for POC - Add it to LMPVaultMintingTests contract in LMPVault-Withdraw.t.sol file under path test/vault. run it via the command

```
forge test --match-path test/vault/LMPVault-Withdraw.t.sol --match-test  
↳ test_AvoidTheLoss -vv
```



```

function test_AvoidTheLoss() public {

// for simplicity sake, i'll be assuming vault keeps nothing idle

// as it does not affect the attack vector in any ways

_accessController.grantRole(Roles.SOLVER_ROLE, address(this));

_accessController.grantRole(Roles.LMP_FEE_SETTER_ROLE, address(this));

address feeSink = vm.addr(555);

_lmpVault.setFeeSink(feeSink);

_lmpVault.setPerformanceFeeBps(2000); // 20%

address alice = address(789);

uint initialBalanceAlice = 1000;

// User is going to deposit 1000 asset

_asset.mint(address(this), 1000);

_asset.approve(address(_lmpVault), 1000);

uint shareBalUser = _lmpVault.deposit(1000, address(this));

_underlyerOne.mint(address(this),500);

_underlyerOne.approve(address(_lmpVault),500);

_lmpVault.rebalance(

address(_destVaultOne),

address(_underlyerOne),

500,

address(0),

address(_asset),

1000

```



```

);

_asset.mint(alice,initialBalanceAlice);

vm.startPrank(alice);

_asset.approve(address(_lmpVault),initialBalanceAlice);

uint shareBalAlice = _lmpVault.deposit(initialBalanceAlice,alice);

vm.stopPrank();

// rebalance to 2nd vault

_underlyerTwo.mint(address(this), 1000);

_underlyerTwo.approve(address(_lmpVault),1000);

_lmpVault.rebalance(

address(_destVaultTwo),

address(_underlyerTwo),

1000,

address(0),

address(_asset),

1000

);

// the second destVault incurs loss, 10%

_mockRootPrice(address(_underlyerTwo), 0.9 ether);


// the first vault incurs some profit, 5%

// so lmpVault is in netLoss of 50 baseAsset

_mockRootPrice(address(_underlyerOne), 2.1 ether);

```



```

// malicious updateDebtReporting by alice

address[] memory alteredDestinations = new address[](1);

alteredDestinations[0] = address(_destVaultOne);

vm.prank(alice);

_lmpVault.updateDebtReporting(alteredDestinations);


// alice withdraws first

vm.prank(alice);

_lmpVault.redeem(shareBalAlice , alice,alice);

uint finalBalanceAlice = _asset.balanceOf(alice);

emit log_named_uint("final Balance of alice ", finalBalanceAlice);

// protocol also collects its fees

// further wrecking the remaining LPs

emit log_named_uint("Fees shares give to feeSink ",
↳ _lmpVault.balanceOf(feeSink));

assertGt( finalBalanceAlice, initialBalanceAlice);

assertGt(_lmpVault.balanceOf(feeSink), 0);

// now updateDebtReporting again but for all DVs

_lmpVault.updateDebtReporting(_destinations);


emit log_named_uint("Remaining LPs can only get
↳ ",_lmpVault.maxWithdraw(address(this)));

emit log_named_uint("Protocol falsely earned(in base asset)",
↳ _lmpVault.maxWithdraw(feeSink));

emit log_named_uint("Vault totalAssets" , _lmpVault.totalAssets());

```



```

emit log_named_uint("Effective loss take by LPs", 1000 -
↳ _lmpVault.maxWithdraw(address(this)));

emit log_named_uint("Profit for Alice", _asset.balanceOf(alice) -
↳ initialBalanceAlice);

}

```

Logs: final Balance of alice : 1019 Fees shares give to feeSink : 10 Remaining LPs can only get : 920 Protocol falsely earned(in base asset): 9 Vault totalAssets: 930 Effective loss take by LPs: 80 Profit for Alice: 19

Impact

Theft of user funds. Submitting as high as attacker only needs to frontrun a updateDebtReporting txn with malicious input and withdraw his funds.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/5d8e902ce33981a6506b1b5fb979a084602c6c9a/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L628-L630C6>

```

function updateDebtReporting(address[] calldata _destinations) external
↳ nonReentrant trackNavOps { // @audit < user controlled input

_updateDebtReporting(_destinations);

}

```

Tool used

Manual Review

Recommendation

updateDebtReporting should not have any input param, should by default update for all added destination vaults



Issue H-7: Stat calculator returns incorrect report for swETH

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/587>

Found by

xiaoming90

Stat calculator returns incorrect reports for swETH, causing multiple implications that could lead to losses to the protocol,

Vulnerability Detail

The purpose of the in-scope SwEthEthOracle contract is to act as a price oracle specifically for swETH (Swell ETH) per the comment in the contract below and the codebase's [README](#)

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/oracles/providers/SwEthEthOracle.sol#L16>

```
File: SwEthEthOracle.sol
12: /**
13:  * @notice Price oracle specifically for swEth (Swell Eth).
14:  * @dev getPriceEth is not a view fn to support reentrancy checks. Does not
   ↳ actually change state.
15:  */
16: contract SwEthEthOracle is SystemComponent, IPriceOracle {
```

Per the codebase in the contest repository, the price oracle for the swETH is understood to be configured to the SwEthEthOracle contract at Line 252 below.

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/test/oracles/RootOracleIntegrationTest.t.sol#L252>

```
File: RootOracleIntegrationTest.t.sol
162:         swEthOracle = new SwEthEthOracle(systemRegistry,
   ↳ IswETH(SWETH_MAINNET));
...SNIP...
249:         // Lst special pricing case setup
250:         // priceOracle.registerMapping(SFRXETH_MAINNET,
   ↳ IPriceOracle(address(sfrxEthOracle)));
251:         priceOracle.registerMapping(WSTETH_MAINNET,
   ↳ IPriceOracle(address(wstEthOracle)));
252:         priceOracle.registerMapping(SWETH_MAINNET,
   ↳ IPriceOracle(address(swEthOracle)));
```



Thus, in the context of this audit, the price oracle for the swETH is mapped to the SwEthEthOracle contract.

Both the swETH oracle and calculator use the same built-in `swEth.swETHToETHRate` function to retrieve the price of swETH in ETH.

```
() LST      Oracle                                Calculator
()
swETH  SwEthEthOracle - swEth.swETHToETHRate()  SwethLSTCalculator - IswETH(1stTokenAdd
())
```

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/oracles/providers/SwEthEthOracle.sol#L26>

```
File: SwEthEthOracle.sol
25:     /// @inheritdoc IPriceOracle
26:     function getPriceInEth(address token) external view returns (uint256
    ↪ price) {
    ..SNIP..
30:         // Returns in 1e18 precision.
31:         price = swEth.swETHToETHRate();
32:     }
```

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/stats/calculators/SwethLSTCalculator.sol#L12>

```
File: SwethLSTCalculator.sol
12:     function calculateEthPerToken() public view override returns (uint256) {
13:         return IswETH(1stTokenAddress).swETHToETHRate();
14:     }
```

Within the `LSTCalculatorBase.current` function, assume that the `swEth.swETHToETHRate` function returns x when called. In this case, the price at Line 203 below and backing in Line 210 below will be set to x since the `getPriceInEth` and `calculateEthPerToken` functions depend on the same `swEth.swETHToETHRate` function internally. Thus, `priceToBacking` will always be $1e18$:

$$\begin{aligned} priceToBacking &= \frac{price \times 1e18}{backing} \\ &= \frac{x \times 1e18}{x} \\ &= 1e18 \end{aligned}$$



Since `priceToBacking` is always `1e18`, the premium will always be zero:

$$\begin{aligned} \text{premium} &= \text{priceToBacking} - 1e18 \\ &= 1e18 - 1e18 \\ &= 0 \end{aligned}$$

As a result, the calculator for swETH will always report the wrong statistic report for swETH. If there is a premium or discount, the calculator will wrongly report none.

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/stats/calculators/base/LSTCalculatorBase.sol#L189>

```
File: LSTCalculatorBase.sol
189:     function current() external returns (LSTStatsData memory) {
    ..SNIP..
202:         IRootPriceOracle pricer = systemRegistry.rootPriceOracle();
203:         uint256 price = pricer.getPriceInEth(lstTokenAddress);
    ..SNIP..
210:         uint256 backing = calculateEthPerToken();
211:         // price is always 1e18 and backing is in eth, which is 1e18
212:         priceToBacking = price * 1e18 / backing;
213:     }
214:
215:     // positive value is a premium; negative value is a discount
216:     int256 premium = int256(priceToBacking) - 1e18;
217:
218:     return LSTStatsData({
219:         lastSnapshotTimestamp: lastSnapshotTimestamp,
220:         baseApr: baseApr,
221:         premium: premium,
222:         slashingCosts: slashingCosts,
223:         slashingTimestamps: slashingTimestamps
224:     });
225: }
```

Impact

The purpose of the stats/calculators contracts is to store, augment, and clean data relevant to the LMPs. When the solver proposes a rebalance, the strategy uses the stats contracts to calculate a composite return (score) for the proposed destinations. Using that composite return, it determines if the swap is beneficial for the vault.

If a stat calculator provides inaccurate information, it can cause multiple implications that lead to losses to the protocol, such as false signals allowing the



unprofitable rebalance to be executed.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/stats/calculators/base/LSTCalculatorBase.sol#L189>

Tool used

Manual Review

Recommendation

When handling the swETH within the `LSTCalculatorBase.current` function, consider other methods of obtaining the fair market price of swETH that do not rely on the `swEth.swETHToETHRate` function such as external 3rd-party price oracle.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

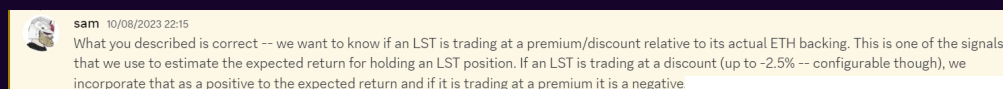
Trumpero commented:

invalid, I think it's intended to let premium = 0 in case of swETH. Also I don't see any further vulnerability if premium = 0

xiaoming9090

Escalate

After my discussion with the protocol team during the audit period (as shown below), the purpose of the `premium` variable is to determine if an LST is trading at a premium/discount relative to its actual ETH backing. This is one of the signals the protocol team uses to estimate the expected return for holding an LST position.



It is incorrect that the premium for swETH is intended to be always zero, as mentioned in the judging comment. If the premium always returns zero, the stat calculator for swETH is effectively broken, which is a serious issue. The judging comment is also incorrect in stating there is no vulnerability if the premium is zero. The stat calculator exists for a reason, and an incorrect stat calculator ultimately leads to losses to the protocol, as mentioned in the "Impact" section of my report:



The purpose of the stats/calculators contracts is to store, augment, and clean data relevant to the LMPs. When the solver proposes a rebalance, the strategy uses the stats contracts to calculate a composite return (score) for the proposed destinations. Using that composite return, it determines if the swap is beneficial for the vault.

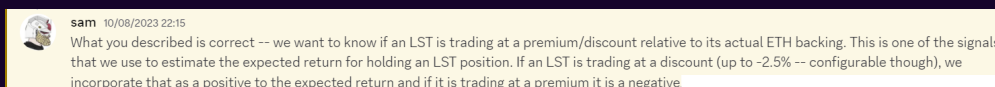
If a stat calculator provides inaccurate information, it can cause multiple implications that lead to losses to the protocol, such as false signals allowing the unprofitable rebalance to be executed.

Thus, this is a valid High issue.

sherlock-admin2

Escalate

After my discussion with the protocol team during the audit period (as shown below), the purpose of the `premium` variable is to determine if an LST is trading at a premium/discount relative to its actual ETH backing. This is one of the signals the protocol team uses to estimate the expected return for holding an LST position.



It is incorrect that the premium for swETH is intended to be always zero, as mentioned in the judging comment. If the premium always returns zero, the stat calculator for swETH is effectively broken, which is a serious issue. The judging comment is also incorrect in stating there is no vulnerability if the premium is zero. The stat calculator exists for a reason, and an incorrect stat calculator ultimately leads to losses to the protocol, as mentioned in the "Impact" section of my report:

The purpose of the stats/calculators contracts is to store, augment, and clean data relevant to the LMPs. When the solver proposes a rebalance, the strategy uses the stats contracts to calculate a composite return (score) for the proposed destinations. Using that composite return, it determines if the swap is beneficial for the vault.

If a stat calculator provides inaccurate information, it can cause multiple implications that lead to losses to the protocol, such as false signals allowing the unprofitable rebalance to be executed.

Thus, this is a valid High issue.

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.

Trumpero

Can u take a look at this issue @codenutt ?

codenutt

Can u take a look at this issue @codenutt ?

Yup definitely an issue. Its more an issue with the oracle itself than the calculator, but an issue none the less.

Evert0x

Planning to accept escalation and make issue high.

Evert0x

Result: High Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- xiaoming9090: accepted



Issue H-8: Incorrect approach to tracking the PnL of a DV

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/589>

Found by

xiaoming90

A DV might be incorrectly marked as not sitting in a loss, thus allowing users to burn all the DV shares, locking in all the loss of the DV and the vault shareholders.

Vulnerability Detail

Let DV_A be a certain destination vault.

Assume that at T_0 , the current debt value (`currentDvDebtValue`) of DV_A is 95 WETH, and the last debt value (`updatedDebtBasis`) is 100 WETH. Since the current debt value has become smaller than the last debt value, the vault is making a loss of 5 WETH since the last rebalancing, so DV_A is sitting at a loss, and users can only burn a limited amount of `DestinationVault_A`'s shares.

Assume that at T_1 , there is some slight rebalancing performed on DV_A , and a few additional LP tokens are deposited to it. Thus, its current debt value increased to 98 WETH. At the same time, the `destInfo.debtBasis` and `destInfo.ownedShares` will be updated to the current value.

Immediately after the rebalancing, DV_A will not be considered sitting in a loss since the `currentDvDebtValue` and `updatedDebtBasis` should be equal now. As a result, users could now burn all the DV_A shares of the `LMPVault` during withdrawal.

DV_A suddenly becomes not sitting at a loss even though the fact is that it is still sitting at a loss of 5 WETH. The loss has been written off.

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/libs/LMPDebt.sol#L275>

```
File: LMPDebt.sol
274:         // Neither of these numbers include rewards from the DV
275:         if (currentDvDebtValue < updatedDebtBasis) {
276:             // We are currently sitting at a loss. Limit the value we can
    ↪ pull from
277:             // the destination vault
278:             currentDvDebtValue = currentDvDebtValue.mulDiv(userShares,
    ↪ totalVaultShares, Math.Rounding.Down);
279:             currentDvShares = currentDvShares.mulDiv(userShares,
    ↪ totalVaultShares, Math.Rounding.Down);
280:         }
```



Impact

A DV might be incorrectly marked as not sitting in a loss, thus allowing users to burn all the DV shares, locking in all the loss of the DV and the vault shareholders.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/libs/LMPDebt.sol#L275>

Tool used

Manual Review

Recommendation

Consider a more sophisticated approach to track a DV's Profit and Loss (PnL).

In our example, DV_A should only be considered not making a loss if the price of the LP tokens starts to appreciate and cover the loss of 5 WETH.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

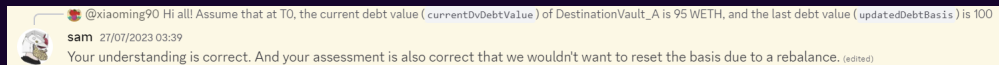
Trumpero commented:

invalid, it is intended that sitting at a loss is considered by comparing the current debt to the latest recorded debt. Additionally, when it considers not sitting at a loss, it doesn't limit the value can pull but it only pulls enough tokens to withdraw. These withdrawn tokens will not be locked

xiaoming9090

Escalate

The main point of this report is to highlight the issues with the current algorithm/approach of tracking the PnL of a DV and marking it as sitting on a loss or not, which is obviously incorrect, as shown in my report. I have discussed this issue with the protocol team during the audit period, and the impact is undesirable, as shown below.

@xiaoming90 Hi all! Assume that at T0, the current debt value (currentDvDebtValue) of DestinationVault_A is 95 WETH, and the last debt value (updatedDebtBasis) is 100
sam 21/07/2023 03:39
Your understanding is correct. And your assessment is also correct that we wouldn't want to reset the basis due to a rebalance. (edited)

It should not be confused with how it is intended to be used in other parts of the protocol, which is unrelated. In addition, the intended approach does not mean that



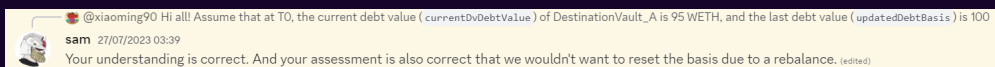
those issues/risks are acceptable and thus considered invalid. If the approach is incorrect, those issues must be flagged during the audit.

Thus, this is a valid High issue.

sherlock-admin2

Escalate

The main point of this report is to highlight the issues with the current algorithm/approach of tracking the PnL of a DV and marking it as sitting on a loss or not, which is obviously incorrect, as shown in my report. I have discussed this issue with the protocol team during the audit period, and the impact is undesirable, as shown below.



It should not be confused with how it is intended to be used in other parts of the protocol, which is unrelated. In addition, the intended approach does not mean that those issues/risks are acceptable and thus considered invalid. If the approach is incorrect, those issues must be flagged during the audit.

Thus, this is a valid High issue.

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.

Trumpero

Tks for the clarification @xiaoming9090 Please review this issue as well @codenutt

codenutt

Tks for the clarification @xiaoming9090 Please review this issue as well @codenutt

Yup agree with @xiaoming9090 here.

Evert0x

Planning to accept escalation and make issue high severity

@Trumpero

Trumpero

Agree with the high severity



Evert0x

Result: High Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- xiaoming9090: accepted



Issue H-9: Inflated price due to unnecessary precision scaling

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/600>

Found by

0xVolodya, Aymen0909, bin2chen, enfrasico, nobody2018, saidam017, talfao, xiaoming90

The price returned by the stat calculators will be excessively inflated, which could lead to multiple implications that lead to losses to the protocol.

Vulnerability Detail

The price at Line 137 below is denominated in 18 decimals as the `getPriceInEth` function always returns the price in 18 decimals precision.

There is no need to scale the accumulated price by $1e18$.

- It will cause the average price (`existing._initAcc`) to be inflated significantly
- The numerator will almost always be larger than the denominator (`INIT_SAMPLE_COUNT = 18`). There is no risk of it rounding to zero, so any scaling is unnecessary.

Assume that throughout the initialization process, the `getPriceInEth(XYZ)` always returns 2 ETH ($2e18$). After 18 rounds (`INIT_SAMPLE_COUNT == 18`) of initialization, `existing._initAcc` will equal 36 ETH ($36e18$). As such, the `averagePrice` will be as follows:

```
averagePrice = existing._initAcc * 1e18 / INIT_SAMPLE_COUNT;
averagePrice = 36e18 * 1e18 / 18
averagePrice = 36e36 / 18
averagePrice = 2e36
```

`existing.fastFilterPrice` and `existing.slowFilterPrice` will be set to $2e36$ at Lines 157 and 158 below.

In the post-init phase, the `getPriceInEth` function return 3 ETH ($3e18$). Thus, the following code will be executed at Line 144s and 155 below:

```
existing.slowFilterPrice = Stats.getFilteredValue(SLOW_ALPHA,
↳ existing.slowFilterPrice, price);
existing.fastFilterPrice = Stats.getFilteredValue(FAST_ALPHA,
↳ existing.fastFilterPrice, price);
```



```
existing.slowFilterPrice = Stats.getFilteredValue(SLOW_ALPHA, 2e36, 3e18); //
↳ SLOW_ALPHA = 645e14; // 0.0645
existing.fastFilterPrice = Stats.getFilteredValue(FAST_ALPHA, 2e36, 3e18); //
↳ FAST_ALPHA = 33e16; // 0.33
```

As shown above, the existing filter prices are significantly inflated by the scale of 1e18, which results in the prices being extremely skewed.

Using the formula of fast filter, the final fast filter price computed will be as follows:

```
((priorValue * (1e18 - alpha)) + (currentValue * alpha)) / 1e18
((priorValue * (1e18 - 33e16)) + (currentValue * 33e16)) / 1e18
((priorValue * 67e16) + (currentValue * 33e16)) / 1e18
((2e36 * 67e16) + (3e18 * 33e16)) / 1e18
1.34e36 (1340000000000000000 ETH)
```

The token is supposed only to be worth around 3 ETH. However, the fast filter price wrongly determine that it is worth around 1340000000000000000 ETH

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/stats/calculators/IncentivePricingStats.sol#L125>

```
File: IncentivePricingStats.sol
125:     function updatePricingInfo(IRootPriceOracle pricer, address token)
↳ internal {
..SNIP..
137:     uint256 price = pricer.getPriceInEth(token);
138:
139:     // update the timestamp no matter what phase we're in
140:     existing.lastSnapshot = uint40(block.timestamp);
141:
142:     if (existing._initComplete) {
143:         // post-init phase, just update the filter values
144:         existing.slowFilterPrice = Stats.getFilteredValue(SLOW_ALPHA,
↳ existing.slowFilterPrice, price);
145:         existing.fastFilterPrice = Stats.getFilteredValue(FAST_ALPHA,
↳ existing.fastFilterPrice, price);
146:     } else {
147:         // still the initialization phase
148:         existing._initCount += 1;
149:         existing._initAcc += price;
150:
151:         // snapshot count is tracked internally and cannot be
↳ manipulated
152:         // slither-disable-next-line incorrect-equality
153:         if (existing._initCount == INIT_SAMPLE_COUNT) { // @audit-info
↳ INIT_SAMPLE_COUNT = 18;
```



```

154:                // if this sample hits the target number, then complete
↳ initialize and set the filters
155:                existing._initComplete = true;
156:                uint256 averagePrice = existing._initAcc * 1e18 /
↳ INIT_SAMPLE_COUNT;
157:                existing.fastFilterPrice = averagePrice;
158:                existing.slowFilterPrice = averagePrice;
159:            }
160:        }

```

Impact

The price returned by the stat calculators will be excessively inflated. The purpose of the stats/calculators contracts is to store, augment, and clean data relevant to the LMPs. When the solver proposes a rebalance, the strategy uses the stats contracts to calculate a composite return (score) for the proposed destinations. Using that composite return, it determines if the swap is beneficial for the vault.

If a stat calculator provides incorrect and inflated pricing, it can cause multiple implications that lead to losses to the protocol, such as false signals allowing the unprofitable rebalance to be executed.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/stats/calculators/IncentivePricingStats.sol#L125>

Tool used

Manual Review

Recommendation

Remove the 1e18 scaling.

```

if (existing._initCount == INIT_SAMPLE_COUNT) {
    // if this sample hits the target number, then complete initialize and set
↳ the filters
    existing._initComplete = true;
-    uint256 averagePrice = existing._initAcc * 1e18 / INIT_SAMPLE_COUNT;
+    uint256 averagePrice = existing._initAcc / INIT_SAMPLE_COUNT;
    existing.fastFilterPrice = averagePrice;
    existing.slowFilterPrice = averagePrice;
}

```



Issue H-10: Immediately start getting rewards belonging to others after staking

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/603>

Found by

0x73696d616f, 0xGoodess, 0xJuda, 0xTheC0der, 0xdeadbeef, 0xvj, Ch_301, Kalyan-Singh, MrjoryStewartBaxter, VAD37, berndartmueller, bin2chen, caelumimperium, carrotsmuggler, jecikpo, l3r0ux, lemonmon, pengun, saidam017, talfao, wangxx2026, xiaoming90

Malicious users could abuse the accounting error to immediately start getting rewards belonging to others after staking, leading to a loss of reward tokens.

Vulnerability Detail

Note This issue affects both LMPVault and DV since they use the same underlying reward contract.

Assume a new user called Bob mints 100 LMPVault or DV shares. The ERC20's `_mint` function will be called, which will first increase Bob's balance at Line 267 and then trigger the `_afterTokenTransfer` hook at Line 271.

<https://github.com/OpenZeppelin/openzeppelin-contracts/blob/0457042d93d9dfd760dbaa06a4d2f1216fdb297/contracts/token/ERC20/ERC20.sol#L259>

```
File: ERC20.sol
259:     function _mint(address account, uint256 amount) internal virtual {
    ..SNIP..
262:         _beforeTokenTransfer(address(0), account, amount);
263:
264:         _totalSupply += amount;
265:         unchecked {
266:             // Overflow not possible: balance + amount is at most
    ↪ totalSupply + amount, which is checked above.
267:             _balances[account] += amount;
268:         }
    ..SNIP..
271:         _afterTokenTransfer(address(0), account, amount);
272:     }
```

The `_afterTokenTransfer` hook will automatically stake the newly minted shares to the rewarder contracts on behalf of Bob.



<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L854>

```
File: LMPVault.sol
854:     function _afterTokenTransfer(address from, address to, uint256 amount)
    ↪ internal virtual override {
    ..SNIP..
862:         if (to != address(0)) {
863:             rewarder.stake(to, amount);
864:         }
865:     }
```

Within the `MainRewarder.stake` function, it will first call the `_updateReward` function at Line 87 to take a snapshot of accumulated rewards. Since Bob is a new user, his accumulated rewards should be zero. However, this turned out to be false due to the bug described in this report.

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/rewarders/MainRewarder.sol#L86>

```
File: MainRewarder.sol
86:     function stake(address account, uint256 amount) public onlyStakeTracker {
87:         _updateReward(account);
88:         _stake(account, amount);
89:
90:         for (uint256 i = 0; i < extraRewards.length; ++i) {
91:             IExtraRewarder(extraRewards[i]).stake(account, amount);
92:         }
93:     }
```

When the `_updateReward` function is executed, it will compute Bob's earned rewards. It is important to note that at this point, Bob's balance has already been updated to 100 shares in the `stakeTracker` contract, and `userRewardPerTokenPaid[Bob]` is zero.

Bob's earned reward will be as follows, where r is the `rewardPerToken()`:

$$\text{earned}(\text{Bob}) = 100 \text{ shares} \times (r - 0) = 100r$$

Bob immediately accumulated a reward of $100r$ upon staking into the rewarder contract, which is incorrect. Bob could withdraw $100r$ reward tokens that do not belong to him.

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/rewarders/AbstractRewarder.sol#L128>

```
File: AbstractRewarder.sol
```



```

128:     function _updateReward(address account) internal {
129:         uint256 earnedRewards = 0;
130:         rewardPerTokenStored = rewardPerToken();
131:         lastUpdateBlock = lastBlockRewardApplicable();
132:
133:         if (account != address(0)) {
134:             earnedRewards = earned(account);
135:             rewards[account] = earnedRewards;
136:             userRewardPerTokenPaid[account] = rewardPerTokenStored;
137:         }
138:
139:         emit UserRewardUpdated(account, earnedRewards,
↳ rewardPerTokenStored, lastUpdateBlock);
140:     }
..SNIP..
155:     function balanceOf(address account) public view returns (uint256) {
156:         return stakeTracker.balanceOf(account);
157:     }
..SNIP..
204:     function earned(address account) public view returns (uint256) {
205:         return (balanceOf(account) * (rewardPerToken() -
↳ userRewardPerTokenPaid[account]) / 1e18) + rewards[account];
206:     }

```

Impact

Loss of reward tokens for the vault shareholders.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L854>

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/rewarders/MainRewarder.sol#L86>

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/rewarders/AbstractRewarder.sol#L128>

Tool used

Manual Review



Recommendation

Ensure that the balance of the users in the rewarder contract is only incremented after the `_updateReward` function is executed.

One option is to track the balance of the staker and total supply internally within the rewarder contract and avoid reading the states in the `stakeTracker` contract, commonly seen in many reward contracts.

```
File: AbstractRewarder.sol
function balanceOf(address account) public view returns (uint256) {
-   return stakeTracker.balanceOf(account);
+   return _balances[account];
}
```

```
File: AbstractRewarder.sol
function _stake(address account, uint256 amount) internal {
    Errors.verifyNotZero(account, "account");
    Errors.verifyNotZero(amount, "amount");

+   _totalSupply += amount
+   _balances[account] += amount

    emit Staked(account, amount);
}
```



Issue H-11: Differences between actual and cached total assets can be arbitrated

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/611>

Found by

0x007, 0xWeiss, Ch_301, Flora, Kalyan-Singh, caelumimperium, lemonmon, n33k, xiaoming90

The difference between $totalAssets_{cached}$ and $totalAssets_{actual}$ could be arbitrated or exploited by malicious users for their gain, leading to a loss to other vault shareholders.

Vulnerability Detail

The actual total amount of assets that are owned by a LMPVault on-chain can be derived via the following formula:

$$totalAssets_{actual} = \sum_{n=1}^x debtValue(DV_n)$$

When `LMPVault.totalAssets()` function is called, it returns the cached total assets of the LMPVault instead.

$$totalAssets_{cached} = totalIdle + totalDebt$$

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L304>

```
File: LMPVault.sol
304:     function totalAssets() public view override returns (uint256) {
305:         return totalIdle + totalDebt;
306:     }
```

Thus, the $totalAssets_{cached}$ will deviate from $totalAssets_{actual}$. This difference could be arbitrated or exploited by malicious users for their gain.

Certain actions such as `previewDeposit`, `previewMint`, `previewWithdraw`, and `previewRedeem` functions rely on the $totalAssets_{cached}$ value while other actions such as `_withdraw` and `_calcUserWithdrawSharesToBurn` functions rely on $totalAssets_{actual}$ value.

The following shows one example of the issue.



The `previewDeposit(assets)` function computed the number of shares to be received after depositing a specific amount of assets:

$$shareReceived = \frac{assets_{deposited}}{totalAssets_{cached}} \times totalSupply$$

Assume that $totalAssets_{cached} < totalAssets_{actual}$, and the values of the variables are as follows:

- $totalAssets_{cached} = 110 \text{ WETH}$
- $totalAssets_{actual} = 115 \text{ WETH}$
- $totalSupply = 100 \text{ shares}$

Assume Bob deposited 10 WETH when the total assets are 110 WETH (when $totalAssets_{cached} < totalAssets_{actual}$), he would receive:

$$\begin{aligned} shareReceived &= \frac{10ETH}{110ETH} \times 100e18 \text{ shares} \\ &= 9.090909091e18 \text{ shares} \end{aligned}$$

If a user deposited 10 WETH while the total assets are updated to the actual worth of 115 WETH (when $totalAssets_{cached} == totalAssets_{actual}$), they would receive:

$$\begin{aligned} shareReceived &= \frac{10ETH}{115ETH} \times 100e18 \text{ shares} \\ &= 8.695652174e18 \text{ shares} \end{aligned}$$

Therefore, Bob is receiving more shares than expected.

If Bob redeems all his nine (9) shares after the $totalAssets_{cached}$ has been updated to $totalAssets_{actual}$, he will receive 10.417 WETH back.

$$\begin{aligned} assetsReceived &= \frac{9.090909091e18 \text{ shares}}{(100e18 + 9.090909091e18) \text{ shares}} \times (115 + 10) ETH \\ &= \frac{9.090909091e18 \text{ shares}}{109.090909091e18 \text{ shares}} \times 125ETH \\ &= 10.41666667 ETH \end{aligned}$$

Bob profits 0.417 WETH simply by arbitraging the difference between the cached and actual values of the total assets. Bob gains is the loss of other vault shareholders.

The $totalAssets_{cached}$ can be updated to $totalAssets_{actual}$ by calling the permissionless `LMPVault.updateDebtReporting` function. Alternatively, one could also perform a



sandwich attack against the `LMPVault.updateDebtReporting` function by front-run it to take advantage of the lower-than-expected price or NAV/share, and back-run it to sell the shares when the price or NAV/share rises after the update.

One could also reverse the attack order, where an attacker withdraws at a higher-than-expected price or NAV/share, perform an update on the total assets, and deposit at a lower price or NAV/share.

Impact

Loss assets for vault shareholders. Attacker gains are the loss of other vault shareholders.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L304>

Tool used

Manual Review

Recommendation

Consider updating `totalAssetscached` to `totalAssetsactual` before any withdrawal or deposit to mitigate this issue.



Issue H-12: Gain From LMPVault Can Be Stolen

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/620>

Found by

0x007, 0xvj, Ch_301, Flora, TangYuanShen, berndartmueller, saidam017, xiaoming90

An attacker can steal the gain of the LMPVault.

Vulnerability Detail

Assume the following:

- LMPVault called *LV* integrates with three (3) destination vaults of different protocols (DV_{curve} , $DV_{Balancer}$, $DV_{Maverick}$)
- The Tokemak's liquidator had called the `LiquidatorRow.claimsVaultRewards` function against all three DVs, and carried out the necessary liquidation of the reward tokens received from Covex, Aura, and Maverick. After the liquidation, 10 WETH of rewards is queued to each of the DV's MainRewarder contracts.
- If the `LMPVault.updateDebtReporting` function is triggered against the three DVs, *LV* will be able to collect 30 WETH of reward tokens (10 WETH from each DV's MainRewarder), and *LV*'s total assets will increase by 30 WETH.

For simplicity's sake, assume that there are 100 shares and the total assets are 100 ETH. Thus, the NAV per share is 1.0. If the `LMPVault.updateDebtReporting` function is triggered, the total assets will become 130 ETH (100 ETH + 30 ETH), and the NAV per share will increase to 1.3.

If Alice owned all the 100 shares in the *LV* where she invested 100 ETH when the vault first accepted deposits from the public, she should gain a profit of 30 ETH.

However, malicious users could perform the following actions within a single transaction to steal most of the gains from Alice (also other users). Protocol fees collected from gain are ignored for simplicity's sake.

1. Assume that the liquidator has queued the rewards of 30 WETH.
2. Bob, a malicious user, could perform a flash loan to borrow 1,000,000 WETH OR perform this attack without a flash loan if he is well-funded.
3. Bob deposited 1,000,000 WETH and minted around 1,000,000 shares.
4. At this point, the vault has 1,000,100 WETH and 1,000,100 shares. The NAV per share is still 1.0.



5. Bob triggers the `LMPVault.updateDebtReporting` function, and the *LV*'s total assets will increase by 30 WETH to 1,000,130 WETH. The NAV per share is now 1.00002999700029997000299970003.
6. Bob withdrew all his 1,000,000 shares and received back 1000029.997 WETH.
7. If Bob uses a flash-loan earlier, repay the flash-loan of 1,000,000 WETH and flash-loan fee, which is negligible (2 WEI on dydx).
8. Bob gains 29.997 WETH within a single transaction.
9. Alice only gained a profit of 0.003 WETH, significantly less than the 30 WETH she was supposed to get.

Impact

Loss of assets for the users as their gain can be stolen.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L37>

Tool used

Manual Review

Recommendation

Following are the list of root causes of the issue and some recommendation to mitigate them.

- `updateDebtReporting` function is permissionless and can be called by anyone. It is recommended to implement access control to ensure that this function can only be triggered by Tokemak team. Do note that even if the attacker cannot trigger the `updateDebtReporting` function, it is still possible for the attacker to front-run and back-end the `updateDebtReporting` transaction to carry out the attack if they see this transaction in the public mempool. Thus, consider sending the `updateDebtReporting` transaction as a private transaction via Flashbot so that the attacker cannot sandwich the transaction.
- There is no withdrawal fee and/or deposit fee. Therefore, this attack is mostly profitable. It is recommended to impose a fee on the users of the vault. All users should be charged a fee for the use of the vault. This will make the attack less likely to be profitable in most cases.



- Users can enter and exit the vault within the same transaction/block. This allows the attacker to leverage the flash-loan facility to reduce the cost of the attack to almost nothing. It is recommended to prevent users from entering and exiting the vault within the same transaction/block. If the user entered the vault in this block, he/she could only exit at the next block.
- There is no snapshotting to keep track of the deposit to ensure that gains are weighted according to deposit duration. Thus, a whale could deposit right before the `updateDebtReporting` function is triggered and exit the vault afterward and reap most of the gains. Consider implementing snapshotting within the vault.



Issue H-13: Incorrect pricing for CurveV2 LP Token

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/621>

Found by

xiaoming90

The price of the CurveV2 LP Tokens is incorrect as the incorrect quote currency is being used when computing the value, resulting in a loss of assets due to the overvaluing or undervaluing of the assets.

Vulnerability Detail

Using the Curve rETH/frxETH pool (0xe7c6e0a739021cdba7aac21b4b728779eef974d9) to illustrate the issue:

The price of the LP token of Curve rETH/frxETH pool can be obtained via the following `lp_price` function:

<https://etherscan.io/address/0xe7c6e0a739021cdba7aac21b4b728779eef974d9#code#L1308>

```
def lp_price() -> uint256:
    """
    Approximate LP token price
    """
    return 2 * self.virtual_price * self.sqrt_int(self.internal_price_oracle())
    ↪ / 10**18
```

Thus, the formula to obtain the price of the LP token is as follows:

$$price_{LP} = 2 \times virtualPrice \times \sqrt{internalPriceOracle}$$

Information about the *internalPriceOracle* can be obtained from the `pool.price_oracle()` function or from the Curve's Pool page (<https://curve.fi/#/ethereum/pools/factory-crypto-218/swap>). Refer to the Price Data's Price Oracle section.

<https://etherscan.io/address/0xe7c6e0a739021cdba7aac21b4b728779eef974d9#code#L1341>

```
def price_oracle() -> uint256:
    return self.internal_price_oracle()
```



The *internalPriceOracle* is the price of `coins[1]` (frxETH) with `coins[0]` (rETH) as the quote currency, which means how many rETH (quote) are needed to purchase one frxETH (base).

$$base/quote$$

frxETH/rETH

During pool registration, the `poolInfo.tokenToPrice` is always set to the second coin (`coins[1]`) as per Line 131 below. In this example, `poolInfo.tokenToPrice` will be set to frxETH token address (`coins[1]`).

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/oracles/providers/CurveV2CryptoEthOracle.sol#L107>

```
File: CurveV2CryptoEthOracle.sol
107:     function registerPool(address curvePool, address curveLpToken, bool
    ↳ checkReentrancy) external onlyOwner {
    ..SNIP..
125:         /**
126:         * Curve V2 pools always price second token in `coins` array in
    ↳ first token in `coins` array. This means that
127:         * if `coins[0]` is Weth, and `coins[1]` is rEth, the price will
    ↳ be rEth as base and weth as quote. Hence
128:         * to get lp price we will always want to use the second token
    ↳ in the array, priced in eth.
129:         */
130:         lpTokenToPool[lpToken] =
131:         PoolData({ pool: curvePool, checkReentrancy: checkReentrancy ?
    ↳ 1 : 0, tokenToPrice: tokens[1] });
```

Note that `assetPrice` variable below is equivalent to *internalPriceOracle* in the above formula.

When fetching the price of the LP token, Line 166 computes the price of frxETH with ETH as the quote currency ($frxETH/ETH$) via the `getPriceInEth` function, and assigns to the `assetPrice` variable.

However, the *internalPriceOracle* or `assetPrice` should be $frxETH/rETH$ instead of $frxETH/ETH$. Thus, the price of the LP token computed will be incorrect.

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/oracles/providers/CurveV2CryptoEthOracle.sol#L151>

```
File: CurveV2CryptoEthOracle.sol
151:     function getPriceInEth(address token) external returns (uint256 price) {
```



```

152:         Errors.verifyNotZero(token, "token");
153:
154:         PoolData memory poolInfo = lpTokenToPool[token];
155:         if (poolInfo.pool == address(0)) revert NotRegistered(token);
156:
157:         ICryptoSwapPool cryptoPool = ICryptoSwapPool(poolInfo.pool);
158:
159:         // Checking for read only reentrancy scenario.
160:         if (poolInfo.checkReentrancy == 1) {
161:             // This will fail in a reentrancy situation.
162:             cryptoPool.claim_admin_fees();
163:         }
164:
165:         uint256 virtualPrice = cryptoPool.get_virtual_price();
166:         uint256 assetPrice =
↳ systemRegistry.rootPriceOracle().getPriceInEth(poolInfo.tokenToPrice);
167:
168:         return (2 * virtualPrice * sqrt(assetPrice)) / 10 ** 18;
169:     }

```

Impact

The protocol relies on the oracle to provide accurate pricing for many critical operations, such as determining the debt values of DV, calculators/stats used during the rebalancing process, NAV/shares of the LMPVault, and determining how much assets the users should receive during withdrawal.

Incorrect pricing of LP tokens would result in many implications that lead to a loss of assets, such as users withdrawing more or fewer assets than expected due to over/undervalued vaults or strategy allowing an unprofitable rebalance to be executed.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/oracles/providers/CurveV2CryptoEthOracle.sol#L151>

Tool used

Manual Review

Recommendation

Update the `getPriceInEth` function to ensure that the *internalPriceOracle* or `assetPrice` return the price of `coins[1]` with `coins[0]` as the quote currency.



Issue H-14: Incorrect number of shares minted as fee

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/624>

Found by

0x007, xiaoming90

An incorrect number of shares was minted as fees during fee collection, resulting in a loss of fee.

Vulnerability Detail

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L818>

```
File: LMPVault.sol
818:         profit = (currentNavPerShare - effectiveNavPerShareHighMark) *
    ↳ totalSupply;
819:         fees = profit.mulDiv(performanceFeeBps, (MAX_FEE_BPS ** 2),
    ↳ Math.Rounding.Up);
820:         if (fees > 0 && sink != address(0)) {
821:             // Calculated separate from other mints as normal share
    ↳ mint is round down
822:             shares = _convertToShares(fees, Math.Rounding.Up);
823:             _mint(sink, shares);
824:             emit Deposit(address(this), sink, fees, shares);
825:         }
```

Assume that the following states:

- The profit is 100 WETH
- The fee is 20%, so the fees will be 20 WETH.
- totalSupply is 100 shares and totalAssets() is 1000 WETH

Let the number of shares to be minted be $shares_{2mint}$. The current implementation uses the following formula (simplified) to determine $shares_{2mint}$.

$$\begin{aligned} shares_{2mint} &= fees \times \frac{totalSupply}{totalAsset()} \\ &= 20 \text{ WETH} \times \frac{100 \text{ shares}}{1000 \text{ WETH}} \\ &= 2 \text{ shares} \end{aligned}$$

In this case, two (2) shares will be minted to the `sink` address as the fee is taken.



However, the above formula used in the codebase is incorrect. The total cost/value of the newly-minted shares does not correspond to the fee taken. Immediately after the mint, the value of the two (2) shares is worth only 19.60 WETH, which does not correspond to the 20 WETH fee that the `sink` address is entitled to.

$$\begin{aligned} value &= 2 \text{ shares} \times \frac{1000 \text{ WETH}}{100 + 2 \text{ shares}} \\ &= 2 \text{ shares} \times 9.8039 \text{ WETH} \\ &= 19.6078 \text{ WETH} \end{aligned}$$

Impact

Loss of fee. Fee collection is an integral part of the protocol; thus the loss of fee is considered a High issue.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L818>

Tool used

Manual Review

Recommendation

The correct formula to compute the number of shares minted as fee should be as follows:

$$\begin{aligned} shares2mint &= \frac{profit \times performanceFeeBps \times totalSupply}{(totalAsset() \times MAX_FEE_BPS) - (performanceFeeBps \times profit)} \\ &= \frac{100\epsilon \times 2000 \times 100 \text{ shares}}{(1000\epsilon \times 10000) - (2000 \times 100\epsilon)} \\ &= 2.0408163265306122448979591836735 \text{ shares} \end{aligned}$$

The above formula is the same as the one LIDO used (<https://docs.lido.fi/guides/steth-integration-guide/#fees>)

The following is the proof to show that 2.0408163265306122448979591836735 shares are worth 20 WETH after the mint.



$$\begin{aligned}
 value &= 2.0408163265306122448979591836735 \text{ shares} \times \frac{1000 \text{ WETH}}{100 + 2.0408163265306122448979591836735 \text{ share}} \\
 &= 2.0408163265306122448979591836735 \text{ shares} \times 9.8039 \text{ WETH} \\
 &= 20 \text{ WETH}
 \end{aligned}$$

Discussion

sherlock-admin2

Escalate Users deposit assets via `deposit` to get shares, and amount is also calculated by `_convertToShares`. The protocol can transfer WETH directly to the sink as fees, why mint share? Since the protocol chooses share as the fee, for the sake of fairness, the same formula as for normal users should be used, and no specialization should be made. So, I think it's product design. Not a valid issue.

You've deleted an escalation for this issue.

xiaoming9090

Escalate Users deposit assets via `deposit` to get shares, and amount is also calculated by `_convertToShares`. The protocol can transfer WETH directly to the sink as fees, why mint share? Since the protocol chooses share as the fee, for the sake of fairness, the same formula as for normal users should be used, and no specialization should be made. So, I think it's product design. Not a valid issue.

Disagree. This is an valid issue.

- 1) The fee that the protocol team is entitled to is 20 WETH in my example. Thus, 20 WETH worth of shares must be minted to the protocol. Using the old formula, the protocol team would receive less than 20 WETH, as shown in the report, which is incorrect.
- 2) The protocol team has already confirmed this issue. Thus, this is obviously not a product design, as mentioned by the escalator.

securitygrid

It is unfair to use two different formulas for user addresses and fee addresses. Since share is choosed as the fee, sink must be treated as a user address. Why let users use formula with losses?

xiaoming9090



It is unfair to use two different formulas for user addresses and fee addresses. Since share is chosen as the fee, sink must be treated as a user address. Why let users use formula with losses?

If the same formula is used, it is not the users who are on the losing end. Instead, it is the protocol team who are on the losing end.

Assume that a user and protocol team are entitled to 20 WETH shares.

- 1) The user deposited 20 WETH, and the system should mint to the user 20 WETH worth of shares
- 2) The protocol earned 20 WETH of fee, and the system should mint to the user 20 WETH worth of shares

Speaking of the old formula, an important difference is that when minting the users' shares, the total assets and supply increase because the user deposited 20 WETH. Thus, the value of the share remain constant before and after minting the shares. However, when minting the protocol's share, only the total supply increases.

The following shows the user received 20 WETH worth of shares after the minting.

- The system determines by 2 shares should be minted to the users

$$\begin{aligned} shares2mint &= fees \times \frac{totalSupply}{totalAsset()} \\ &= 20\ WETH \times \frac{100\ shares}{1000\ WETH} \\ &= 2\ shares \end{aligned}$$

- Value of the 2 shares after the minting.

$$\begin{aligned} value &= 2\ shares \times \frac{1000\ WETH + 20\ WETH}{100 + 2\ shares} \\ &= 2\ shares \times 10\ WETH \\ &= 20\ WETH \end{aligned}$$

The following shows that the protocol did not receive 20 WETH worth of shares after the minting.

- The system determines that 2 shares should be minted to the protocol based on the old formula:



$$\begin{aligned}
 shares2mint &= fees \times \frac{totalSupply}{totalAsset()} \\
 &= 20 \text{ WETH} \times \frac{100 \text{ shares}}{1000 \text{ WETH}} \\
 &= 2 \text{ shares}
 \end{aligned}$$

- Value of the 2 shares after the minting.

$$\begin{aligned}
 value &= 2 \text{ shares} \times \frac{1000 \text{ WETH}}{100 + 2 \text{ shares}} \\
 &= 2 \text{ shares} \times 9.8039 \text{ WETH} \\
 &= 19.6078 \text{ WETH}
 \end{aligned}$$

securitygrid

Speaking of the old formula, an important difference is that when minting the users' shares, the total assets and supply increase because the user deposited 20 WETH. Thus, the value of the share remain constant before and after minting the shares. However, when minting the protocol's share, only the total supply increases.

Agree it. Thanks for your explanation.



Issue H-15: Maverick oracle can be manipulated

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/635>

Found by

Bauchibred, ctf_sec, duc, lemonmon, rvierdiev, saidam017, xiaoming90 The MavEthOracle.getPriceInEth() function uses the reserves of the Maverick pool to calculate the price of Maverick LP tokens. These reserves can be manipulated, which can lead to incorrect results of Maverick oracle.

Vulnerability Detail

In the MavEthOracle contract, getPriceInEth function utilizes the reserves of the Maverick pool and multiplies them with the external prices of the tokens (obtained from the rootPriceOracle contract) to calculate the total value of the Maverick position.

However, the reserves of a Maverick position can fluctuate when the price of the Maverick pool changes. Therefore, the returned price of this function can be manipulated by swapping a significant amount of tokens into the Maverick pool. An attacker can utilize a flash loan to initiate a swap, thereby changing the price either upwards or downwards, and subsequently swapping back to repay the flash loan.

Attacker can decrease the returned price of MavEthOracle by swapping a large amount of the higher value token for the lower value token, and vice versa.

Here is a test file that demonstrates how the price of the MavEthOracle contract can be manipulated by swapping to change the reserves.

Impact

There are multiple impacts that an attacker can exploit by manipulating the price of MavEthOracle:

- Decreasing the oracle price to lower the totalDebt of LMPVault, in order to receive more LMPVault shares.
- Increasing the oracle price to raise the totalDebt of LMPVault, in order to receive more withdrawn tokens.
- Manipulating the results of the Stats contracts to cause miscalculations for the protocol.



Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/oracles/providers/MavEthOracle.sol#L59-L72>

Tool used

Manual Review Foundry

Recommendation

Use another calculation for Maverick oracle

Discussion

codenutt

The Mav oracle is only meant to price the value of a boosted position. This is further limited by us only supporting mode "both" positions:

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/README.md?plain=1#L14>. The key about mode "both" positions is that they can only contain 1 bin. The maxTotalBinWidth check against the tick spacing, when limited to a single bin, controls the amount of price change we are willing to tolerate. Its default is 50 bps which we are willing to tolerate.



Issue H-16: Aura/Convex rewards are stuck after DOS

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/738>

Found by

0x007, 0x73696d616f, ADM, bin2chen, caelumimperium, ck, ctf_sec, minhtrng, nobody2018, penguin, pks_, saidam017, tives, xiaoming90

Since `_claimRewards` accounts for rewards with `balanceBefore/After`, and anyone can claim Convex rewards, then attacker can DOS the rewards and make them stuck in the `LiquidationRow` contract.

Vulnerability Detail

Anyone can claim Convex rewards for any account.

<https://etherscan.io/address/0x0A760466E1B4621579a82a39CB56Dda2F4E70f03#code>

```
function getReward(address _account, bool _claimExtras) public
↳ updateReward(_account) returns(bool){
    uint256 reward = earned(_account);
    if (reward > 0) {
        rewards[_account] = 0;
        rewardToken.safeTransfer(_account, reward);
        IDeposit(operator).rewardClaimed(pid, _account, reward);
        emit RewardPaid(_account, reward);
    }

    //also get rewards from linked rewards
    if(_claimExtras){
        for(uint i=0; i < extraRewards.length; i++){
            IRewards(extraRewards[i]).getReward(_account);
        }
    }
    return true;
}
```

In `ConvexRewardsAdapter`, the rewards are accounted for by using `balanceBefore/after`.

```
function _claimRewards(
    address gauge,
    address defaultToken,
    address sendTo
```




```

) internal returns (uint256[] memory amounts, address[] memory tokens) {

    uint256[] memory balancesBefore = new uint256[](totalLength);
    uint256[] memory amountsClaimed = new uint256[](totalLength);
    ...

    for (uint256 i = 0; i < totalLength; ++i) {
        uint256 balance = 0;
        // Same check for "stash tokens"
        if (IERC20(rewardTokens[i]).totalSupply() > 0) {
            balance = IERC20(rewardTokens[i]).balanceOf(account);
        }

        amountsClaimed[i] = balance - balancesBefore[i];

    }

    return (amountsClaimed, rewardTokens);
}

```

Adversary can call the external convex contract's `getReward(tokemakContract)`. After this, the reward tokens are transferred to Tokemak without an accounting hook.

Now, when Tokemak calls `claimRewards`, then no new rewards are transferred, because the attacker already transferred them. `amountsClaimed` will be 0.

Impact

Rewards are stuck in the `LiquidationRow` contract and not queued to the `MainRewarder`.

Code Snippet

```

// get balances after and calculate amounts claimed
for (uint256 i = 0; i < totalLength; ++i) {
    uint256 balance = 0;
    // Same check for "stash tokens"
    if (IERC20(rewardTokens[i]).totalSupply() > 0) {
        balance = IERC20(rewardTokens[i]).balanceOf(account);
    }

    amountsClaimed[i] = balance - balancesBefore[i];

    if (sendTo != address(this) && amountsClaimed[i] > 0) {
        IERC20(rewardTokens[i]).safeTransfer(sendTo, amountsClaimed[i]);
    }
}
}

```



<https://github.com/sherlock-audit/2023-06-tokemak/blob/5d8e902ce33981a6506b1b5fb979a084602c6c9a/v2-core-audit-2023-07-14/src/destinations/adapters/rewards/ConvexRewardsAdapter.sol/#L102>

Tool used

Manual Review

Recommendation

Don't use `balanceBefore/After`. You could consider using `balanceOf(address(this))` after claiming to see the full amount of tokens in the contract. This assumes that only the specific rewards balance is in the contract.



Issue M-1: `getPriceInEth` in `TellorOracle.sol` doesn't use the best practices recommended by Tellor which can cause wrong pricing

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/351>

Found by

Vagner The function `getPriceInEth` it is used to call Tellor oracle to check the prices of different assets but the implementation doesn't respect the best practices recommended by Tellor which can cause wrong pricing.

Vulnerability Detail

Tellor team has a **Development Checklist** <https://docs.tellor.io/tellor/getting-data/user-checklists#development-checklist> that is important to use when you are using Tellor protocol to protect yourself against stale prices and also to limit dispute attacks. The prices in Tellor protocol can be disputed and changed in case of malicious acting and because of that dispute attacks can happen which would make the prices unstable. `getPriceInEth` calls `getDataBefore` <https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/oracles/providers/TellorOracle.sol#L105> and uses the timestamp returned to check if the prices are stale or not using `DEFAULT_PRICING_TIMEOUT` or `tellorStoredTimeout` <https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/oracles/providers/TellorOracle.sol#L107-L112> The value of the `DEFAULT_PRICING_TIMEOUT` is 2 hours and the frame of time where a dispute could happen is around 12 hours as per Tellor documentation, which could make `getPriceInEth` revert multiple times because of possible dispute attacks that could happen. The way the Tellor protocol recommends to limit the possibility of dispute attacks is by caching the most recent values and timestamps as can be seen here in their example <https://github.com/tellor-io/tellor-caller-liquity/blob/3ed91e4b15f0bac737f2966e36805ade3daf8162/contracts/TellorCaller.sol#L9-L11> Basically every time data is returned it is checked against the last stored values and in case of a dispute attack it will return the last stored undisputed and accurate value as can be seen here <https://github.com/tellor-io/tellor-caller-liquity/blob/3ed91e4b15f0bac737f2966e36805ade3daf8162/contracts/TellorCaller.sol#L51-L58> It is also important to note that the Tellor protocol checks for data returned to not be 0 <https://github.com/tellor-io/tellor-caller-liquity/blob/3ed91e4b15f0bac737f2966e36805ade3daf8162/contracts/TellorCaller.sol#L51> which is not done in the `getPriceInEth`, so in the case of a 0 value it will be decoded <https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/oracles/providers/TellorOracle.sol#L114> and then passed into `_denominationPricing` <https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/oracles/providers/TellorOracle.sol#L114>



[t/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/oracles/providers/TellorOracle.sol#L115](https://github.com/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/oracles/providers/TellorOracle.sol#L115) which would return a 0 value that could hurt the protocol.

Impact

Medium of because of wrong pricing or multiple revert in case of dispute attacks

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/oracles/providers/TellorOracle.sol#L101-L116>

Tool used

Manual Review

Recommendation

I recommend to use the Tellor example in <https://github.com/tellor-io/tellor-caller-liquidity/blob/main/contracts/TellorCaller.sol> and implement mappings to cache the values all the time, similar to how Tellor does it to limit as much as possible dispute attacks and also check for 0 value.

Discussion

sherlock-admin2

2 comment(s) were left on this issue during the judging contest.

Trumpero commented:

low, oracle completeness is considered low similar to chainlink round completeness

thekmj commented:

great analysis

VagnerAndrei26

Escalate I don't believe this should be taken as a low because of several factors :

- firstly, sherlock rules states that Chainlink round completeness check are invalid since the new OCR doesn't rely on rounds for reporting the price, it doesn't state that every round completeness is invalid, because the mechanics may differ from oracle to oracle and in some situations it is



important to check for everything

17. Chainlink round completeness check is invalid. The new OCR does not rely on rounds for reporting.

- secondly, because of how Tellor oracle work, anyone can dispute prices and act maliciously by paying the right amount of fees in TRB, that's why the Tellor talks about "dispute attacks" and specify why it is important to protect yourself against them, they acknowledge the risk where bad actors could manipulate the prices for their benefits and if a protocol implementing Tellor oracle does not protect themselves against this, it could hurt it.
- thirdly, it is important when implementing an external protocol, especially an important one like an oracle, to follow their best practices and recommendations since their methods are battle-tested and any mistakes could lead to loss of funds

In the end, I believe that this should not be treated as a simple Chainlink round completeness check, since the mechanics are different and the issue is more complex than that, I believe it should be treated highly since an attack path that could lead to loss of funds by manipulating the price exists, which Tokemak protocol doesn't protect against, as recommender by Tellor.

sherlock-admin2

Escalate I don't believe this should be taken as a low because of several factors :

- firstly, sherlock rules states that Chainlink round completeness check are invalid since the new OCR doesn't rely on rounds for reporting the price, it doesn't state that every round completeness is invalid, because the mechanics may differ from oracle to oracle and in some situations it is important to check for everything

17. Chainlink round completeness check is invalid. The new OCR does not rely on rounds for reporting.

- secondly, because of how Tellor oracle work, anyone can dispute prices and act maliciously by paying the right amount of fees in TRB, that's why the Tellor talks about "dispute attacks" and specify why it is important to protect yourself against them, they acknowledge the risk where bad actors could manipulate the prices for their benefits and if a protocol implementing Tellor oracle does not protect themselves against this, it could hurt it.
- thirdly, it is important when implementing an external protocol, especially an important one like an oracle, to follow their best practices and recommendations since their methods are battle-tested and any mistakes could lead to loss of funds



In the end, I believe that this should not be treated as a simple Chainlink round completeness check, since the mechanics are different and the issue is more complex than that, I believe it should be treated highly since an attack path that could lead to loss of funds by manipulating the price exists, which Tokemak protocol doesn't protect against, as recommender by Tellor.

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.

xiaoming9090

I agree with the escalator that this should be a valid issue. The root cause is that oracle did not cache the most recent values and timestamps per Tellor's recommendation, leading to dispute attacks. This is similar to the issue I mentioned in my report

(<https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/612>).

This issue (<https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/351>) and Issue <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/612> should be an issue of its own, and should not be grouped with the rest. Refer to my escalation (<https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/612#issuecomment-1748029481>) for more details.

Evert0x

Planning to accept escalation and make issue medium

Evert0x

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- [VagnerAndrei26](#): accepted



Issue M-2: Lost rewards when the supply is 0, which always happens if the rewards are queued before anyone has StakeTracker tokens

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/387>

Found by

0x73696d616f, chaduke, hassan-truscova, lucifero, p0wd3r If the supply of StakeTracker tokens is 0, the `rewardPerTokenStored` won't increase, but the `lastUpdateBlock` will, leading to lost rewards.

Vulnerability Detail

The rewards are distributed in a MasterChef style, which takes snapshots of the total accrued rewards over time and whenever someone wants to get the rewards, it subtracts the snapshot of the user from the most updated, global snapshot.

The `rewardsPerToken()` calculation factors the blocks passed times the reward rate by the `totalSupply()`, to get the reward per token in a specific interval (and then accrues to the previous intervals, as stated in the last paragraph). When the `totalSupply()` is 0, there is 0 `rewardPerToken()` increment as there is no supply to factor the rewards by.

The current solution is to maintain the same `rewardsPerToken()` if the `totalSupply()` is 0, but the `lastUpdateBlock` is still updated. This means that, during the interval in which the `totalSupply()` is 0, no rewards are distributed but the block numbers still move forward, leaving the tokens stuck in the `MainRewarder` and `ExtraRewarder` smart contracts.

This will always happen if the rewards are queued before the `totalSupply()` is bigger than 0 (before an initial deposit to either `DestinationVault` or `LMPVault`). It might also happen if users withdraw all their tokens from the vaults, leading to a `totalSupply()` of 0, but this is very unlikely.

Impact

Lost reward tokens. The amount depends on the time during which the `totalSupply()` is 0, but could be significant.

Code Snippet

The `rewardPerToken()` calculation:



```

function rewardPerToken() public view returns (uint256) {
    uint256 total = totalSupply();
    if (total == 0) {
        return rewardPerTokenStored;
    }

    return rewardPerTokenStored + ((lastBlockRewardApplicable() -
↪ lastUpdateBlock) * rewardRate * 1e18 / total);
}

```

The rewardPerTokenStored does not increment when the totalSupply() is 0.

Tool used

Vscode Foundry Manual Review

Recommendation

The totalSupply() should not realistically be 0 after the initial setup period (unless for some reason everyone decides to withdraw from the vaults, but this should be handled separately). It should be enough to only allow queueing rewards if the totalSupply() is bigger than 0. For this, only a new check needs to be added:

```

function queueNewRewards(uint256 newRewards) external onlyWhitelisted {
    if (totalSupply() == 0) revert ZeroTotalSupply();
    ...
}

```



Issue M-3: LMPVault._withdraw() can revert due to an arithmetic underflow

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/519>

Found by

Flora, berndartmueller, nobody2018, shaka, xiaoming90 LMPVault._withdraw() can revert due to an arithmetic underflow.

Vulnerability Detail

Inside the `_withdraw()` function, the `maxAssetsToPull` argument value of `_calcUserWithdrawSharesToBurn()` is calculated to be equal to `info.totalAssetsToPull - Math.max(info.debtDecrease, info.totalAssetsPulled)`. However, the `_withdraw()` function only halts its loop when `info.totalAssetsPulled >= info.totalAssetsToPull`. This can lead to a situation where `info.debtDecrease >= info.totalAssetsToPull`. Consequently, when calculating `info.totalAssetsToPull - Math.max(info.debtDecrease, info.totalAssetsPulled)` for the next destination vault in the loop, an underflow occurs and triggers a contract revert.

To illustrate this vulnerability, consider the following scenario:

```
function test_revert_underflow() public {
    _accessController.grantRole(Roles.SOLVER_ROLE, address(this));
    _accessController.grantRole(Roles.LMP_FEE_SETTER_ROLE, address(this));

    // User is going to deposit 1500 asset
    _asset.mint(address(this), 1500);
    _asset.approve(address(_lmpVault), 1500);
    _lmpVault.deposit(1500, address(this));

    // Deployed 700 asset to DV1
    _underlyerOne.mint(address(this), 700);
    _underlyerOne.approve(address(_lmpVault), 700);
    _lmpVault.rebalance(
        address(_destVaultOne),
        address(_underlyerOne), // tokenIn
        700,
        address(0), // destinationOut, none when sending out baseAsset
        address(_asset), // baseAsset, tokenOut
        700
    );
}
```



```

// Deploy 600 asset to DV2
_underlyerTwo.mint(address(this), 600);
_underlyerTwo.approve(address(_lmpVault), 600);
_lmpVault.rebalance(
    address(_destVaultTwo),
    address(_underlyerTwo), // tokenIn
    600,
    address(0), // destinationOut, none when sending out baseAsset
    address(_asset), // baseAsset, tokenOut
    600
);

// Deployed 200 asset to DV3
_underlyerThree.mint(address(this), 200);
_underlyerThree.approve(address(_lmpVault), 200);
_lmpVault.rebalance(
    address(_destVaultThree),
    address(_underlyerThree), // tokenIn
    200,
    address(0), // destinationOut, none when sending out baseAsset
    address(_asset), // baseAsset, tokenOut
    200
);

// Drop the price of DV2 to 70% of original, so that 600 we transferred out
↪ is now only worth 420
_mockRootPrice(address(_underlyerTwo), 7e17);

// Revert because of an arithmetic underflow
vm.expectRevert();
uint256 assets = _lmpVault.redeem(1000, address(this), address(this));
}

```

Impact

The vulnerability can result in the contract reverting due to an underflow, disrupting the functionality of the contract. Users who try to withdraw assets from the LMPVault may encounter transaction failures and be unable to withdraw their assets.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L475> <https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L493-L504>



Tool used

Manual Review

Recommendation

To mitigate this vulnerability, it is recommended to break the loop within the `_withdraw()` function if `Math.max(info.debtDecrease, info.totalAssetsPulled) >= info.totalAssetsToPull`

```
if (
  Math.max(info.debtDecrease, info.totalAssetsPulled) >
  info.totalAssetsToPull
) {
  info.idleIncrease =
    Math.max(info.debtDecrease, info.totalAssetsPulled) -
    info.totalAssetsToPull;
  if (info.totalAssetsPulled >= info.debtDecrease) {
    info.totalAssetsPulled = info.totalAssetsToPull;
  }
  break;
}

// No need to keep going if we have the amount we're looking for
// Any overage is accounted for above. Anything lower and we need to keep going
// slither-disable-next-line incorrect-equality
if (
  Math.max(info.debtDecrease, info.totalAssetsPulled) ==
  info.totalAssetsToPull
) {
  break;
}
```



Issue M-4: Unable to withdraw extra rewards

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/565>

Found by

0xGoodess, 0xdeadbeef, Ch_301, berndartmueller, xiaoming90

Users are unable to withdraw extra rewards due to staking of TOKE that is less than MIN_STAKE_AMOUNT, resulting in them being stuck in the contracts.

Vulnerability Detail

Suppose Bob only has 9999 Wei TOKE tokens as main rewards and 100e18 DAI as extra rewards in this account.

When attempting to get the rewards, the code will always get the main rewards, followed by the extra rewards, as shown below.

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/rewarders/MainRewarder.sol#L108>

```
File: MainRewarder.sol
108:     function _processRewards(address account, bool claimExtras) internal {
109:         _getReward(account);
110:
111:         //also get rewards from linked rewards
112:         if (claimExtras) {
113:             for (uint256 i = 0; i < extraRewards.length; ++i) {
114:                 IExtraRewarder(extraRewards[i]).getReward(account);
115:             }
116:         }
117:     }
```

If the main reward is TOKE, they will be staked to the GPToke at Line 376 below.

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/rewarders/AbstractRewarder.sol#L354>

```
File: AbstractRewarder.sol
354:     function _getReward(address account) internal {
355:         Errors.verifyNotZero(account, "account");
356:
357:         uint256 reward = earned(account);
358:         (IGPToke gpToke, address tokenAddress) = (systemRegistry.gpToke(),
    ↪ address(systemRegistry.token()));
359:
```



```

360:         // slither-disable-next-line incorrect-equality
361:         if (reward == 0) return;
362:
363:         rewards[account] = 0;
364:         emit RewardPaid(account, reward);
365:
366:         // if NOT token, or staking is turned off (by duration = 0), just
        ↪ send reward back
367:         if (rewardToken != tokenAddress || tokenLockDuration == 0) {
368:             IERC20(rewardToken).safeTransfer(account, reward);
369:         } else {
370:             // authorize gpToken to get our reward Token
371:             // slither-disable-next-line unused-return
372:             IERC20(address(tokenAddress)).approve(address(gpToken), reward);
373:
374:             // stake Token
375:             gpToken.stake(reward, tokenLockDuration, account);
376:         }
377:     }

```

However, if the staked amount is less than the minimum stake amount (MIN_STAKE_AMOUNT), the function will revert.

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/staking/GPToken.sol#L98>

```

File: GPToken.sol
32:     uint256 public constant MIN_STAKE_AMOUNT = 10_000;
..SNIP..
098:     function _stake(uint256 amount, uint256 duration, address to) internal
        ↪ whenNotPaused {
099:         //
100:         // validation checks
101:         //
102:         if (to == address(0)) revert ZeroAddress();
103:         if (amount < MIN_STAKE_AMOUNT) revert StakingAmountInsufficient();
104:         if (amount > MAX_STAKE_AMOUNT) revert StakingAmountExceeded();

```

In this case, Bob will not be able to redeem his 100 DAI reward when processing the reward. The code will always attempt to stake 9999 Wei Token and revert because it fails to meet the minimum stake amount.

Impact

There is no guarantee that the users' TOKE rewards will always be larger than MIN_STAKE_AMOUNT as it depends on various factors such as the following:



- The number of vault shares they hold. If they hold little shares, their TOKE reward will be insignificant
- If their holding in the vault is small compared to the others and the entire vault, the TOKE reward they received will be insignificant
- The timing they join the vault. If they join after the reward is distributed, they will not be entitled to it.

As such, the affected users will not be able to withdraw their extra rewards, and they will be stuck in the contract.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/rewarders/MainRewarder.sol#L108>

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/rewarders/AbstractRewarder.sol#L354>

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/staking/GPToke.sol#L98>

Tool used

Manual Review

Recommendation

To remediate the issue, consider collecting TOKE and staking it to the GPToke contract only if it meets the minimum stake amount.

```
function _getReward(address account) internal {
    Errors.verifyNotZero(account, "account");

    uint256 reward = earned(account);
    (IGPToke gpToke, address tokenAddress) = (systemRegistry.gpToke(),
    ↪ address(systemRegistry.token()));

    // slither-disable-next-line incorrect-equality
    if (reward == 0) return;

    - rewards[account] = 0;
    - emit RewardPaid(account, reward);

    // if NOT token, or staking is turned off (by duration = 0), just send reward
    ↪ back
    if (rewardToken != tokenAddress || tokenLockDuration == 0) {
```



```

+         rewards[account] = 0;
+         emit RewardPaid(account, reward);
+         IERC20(rewardToken).safeTransfer(account, reward);
    } else {
+         if (reward >= MIN_STAKE_AMOUNT) {
+             rewards[account] = 0;
+             emit RewardPaid(account, reward);
+
+             // authorize gpToke to get our reward Toke
+             // slither-disable-next-line unused-return
+             IERC20(address(tokenAddress)).approve(address(gpToke), reward);
+
+             // stake Toke
+             gpToke.stake(reward, tokenLockDuration, account);
+         }
    }
}

```



Issue M-5: Malicious or compromised admin of certain LSTs could manipulate the price

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/570>

Found by

ctf_sec, xiaoming90

Malicious or compromised admin of certain LSTs could manipulate the price of the LSTs.

Vulnerability Detail

Important Per the [contest detail page](#), admins of the external protocols are marked as "Restricted" (Not Trusted). This means that any potential issues arising from the external protocol's admin actions (maliciously or accidentally) are considered valid in the context of this audit.

Q: Are the admins of the protocols your contracts integrate with (if any) TRUSTED or RESTRICTED?

RESTRICTED

Note This issue also applies to other support Liquid Staking Tokens (LSTs) where the admin could upgrade the token contract code. Those examples are omitted for brevity, as the write-up and mitigation are the same and would duplicate this issue.

Per the [contest detail page](#), the protocol will hold and interact with the Swell ETH (swETH).

Liquid Staking Tokens

- swETH: 0xf951E335afb289353dc249e82926178EaC7DEd78

Upon inspection of the swETH on-chain contract, it was found that it is a Transparent Upgradeable Proxy. This means that the admin of Swell protocol could upgrade the contracts.

Tokemak relies on the `swEth.swETHToETHRate()` function to determine the price of the swETH LST within the protocol. Thus, a malicious or compromised admin of Swell could upgrade the contract to have the `swETHToETHRate` function return an extremely high to manipulate the total values of the vaults, resulting in users being able to withdraw more assets than expected, thus draining the LMPVault.

```
File: SwEthEthOracle.sol
```




```
26:     function getPriceInEth(address token) external view returns (uint256
↳ price) {
27:         // Prevents incorrect config at root level.
28:         if (token != address(swEth)) revert Errors.InvalidToken(token);
29:
30:         // Returns in 1e18 precision.
31:         price = swEth.swETHToETHRate();
32:     }
```

Impact

Loss of assets in the scenario as described above.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/oracles/providers/SwEthEthOracle.sol#L26>

Tool used

Manual Review

Recommendation

The protocol team should be aware of the above-mentioned risks and consider implementing additional controls to reduce the risks.

Review each of the supported LSTs and determine how much power the Liquid staking protocol team/admin has over its tokens.

For LSTs that are more centralized (e.g., Liquid staking protocol team could update the token contracts or have the ability to update the exchange rate/price to an arbitrary value without any limit), those LSTs should be subjected to additional controls or monitoring, such as implementing some form of circuit breakers if the price deviates beyond a reasonable percentage to reduce the negative impact to Tokemak if it happens.



Issue M-6: `previewRedeem` and `redeem` functions deviate from the ERC4626 specification

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/577>

Found by

0x73696d616f, 0xSurena, BPZ, BTK, Flora, Kalyan-Singh, Nadin, bin2chen, enfrasico, shaka, talfao, xiaoming90

The `previewRedeem` and `redeem` functions deviate from the ERC4626 specification. As a result, the caller (internal or external) of the `previewRedeem` function might receive incorrect information, leading to the wrong decision being executed.

Vulnerability Detail

Important The contest page explicitly mentioned that the `LMPVault` must conform with the ERC4626. Thus, issues related to EIP compliance should be considered valid in the context of this audit.

Q: Is the code/contract expected to comply with any EIPs? Are there specific assumptions around adhering to those EIPs that Watsons should be aware of?

`src/vault/LMPVault.sol` should be 4626 compatible

Let the value returned by `previewRedeem` function be $asset_{preview}$ and the actual number of assets obtained from calling the `redeem` function be $asset_{actual}$.

The following specification of `previewRedeem` function is taken from ERC4626 specification:

Allows an on-chain or off-chain user to simulate the effects of their redemption at the current block, given current on-chain conditions.

MUST return as close to and no more than the exact amount of assets that would be withdrawn in a `redeem` call in the same transaction. I.e. `redeem` should return the same or more assets as `previewRedeem` if called in the same transaction.

It mentioned that the `redeem` should return the same or more assets as `previewRedeem` if called in the same transaction, which means that it must always be $asset_{preview} \leq asset_{actual}$.

However, it is possible that the `redeem` function might return fewer assets than the number of assets previewed by the `previewRedeem` ($asset_{preview} > asset_{actual}$), thus it does not conform to the specification.



<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L422>

```
File: LMPVault.sol
422:     function redeem(
423:         uint256 shares,
424:         address receiver,
425:         address owner
426:     ) public virtual override nonReentrant noNavDecrease ensureNoNavOps
    ↪ returns (uint256 assets) {
427:         uint256 maxShares = maxRedeem(owner);
428:         if (shares > maxShares) {
429:             revert ERC4626ExceededMaxRedeem(owner, shares, maxShares);
430:         }
431:         uint256 possibleAssets = previewRedeem(shares); // @audit-info
    ↪ round down, which is correct because user won't get too many
432:
433:         assets = _withdraw(possibleAssets, shares, receiver, owner);
434:     }
```

Note that the `previewRedeem` function performs its computation based on the cached `totalDebt` and `totalIdle`, which might not have been updated to reflect the actual on-chain market condition. Thus, these cached values might be higher than expected.

Assume that `totalIdle` is zero and all WETH has been invested in the destination vaults. Thus, `totalAssetsToPull` will be set to $asset_{preview}$.

If a DV is making a loss, users could only burn an amount proportional to their ownership of this vault. The code will go through all the DVs in the withdrawal queue (`withdrawalQueueLength`) in an attempt to withdraw as many assets as possible. However, it is possible that the `totalAssetsPulled` to be less than $asset_{preview}$.

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L448>

```
File: LMPVault.sol
448:     function _withdraw(
449:         uint256 assets,
450:         uint256 shares,
451:         address receiver,
452:         address owner
453:     ) internal virtual returns (uint256) {
454:         uint256 idle = totalIdle;
455:         WithdrawInfo memory info = WithdrawInfo({
456:             currentIdle: idle,
```



```

457:         assetsFromIdle: assets >= idle ? idle : assets,
458:         totalAssetsToPull: assets - (assets >= idle ? idle : assets),
459:         totalAssetsPulled: 0,
460:         idleIncrease: 0,
461:         debtDecrease: 0
462:     });
463:
464:     // If not enough funds in idle, then pull what we need from
    ↪ destinations
465:     if (info.totalAssetsToPull > 0) {
466:         uint256 totalVaultShares = totalSupply();
467:
468:         // Using pre-set withdrawalQueue for withdrawal order to help
    ↪ minimize user gas
469:         uint256 withdrawalQueueLength = withdrawalQueue.length;
470:         for (uint256 i = 0; i < withdrawalQueueLength; ++i) {
471:             IDestinationVault destVault =
    ↪ IDestinationVault(withdrawalQueue[i]);
472:             (uint256 sharesToBurn, uint256 totalDebtBurn) =
    ↪ _calcUserWithdrawSharesToBurn(
473:                 destVault,
474:                 shares,
475:                 info.totalAssetsToPull - Math.max(info.debtDecrease,
    ↪ info.totalAssetsPulled),
476:                 totalVaultShares
477:             );
    ..SNIP..
478:
479:         // At this point should have all the funds we need sitting in in
    ↪ the vault
480:         uint256 returnedAssets = info.assetsFromIdle +
    ↪ info.totalAssetsPulled;

```

Impact

It was understood from the protocol team that they anticipate external parties to integrate directly with the LMPVault (e.g., vault shares as collateral). Thus, the LMPVault must be ERC4626 compliance. Otherwise, the caller (internal or external) of the `previewRedeem` function might receive incorrect information, leading to the wrong action being executed.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L422>



Tool used

Manual Review

Recommendation

Ensure that $asset_{preview} \leq asset_{actual}$.

Alternatively, document that the `previewRedeem` and `redeem` functions deviate from the ERC4626 specification in the comments and/or documentation.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

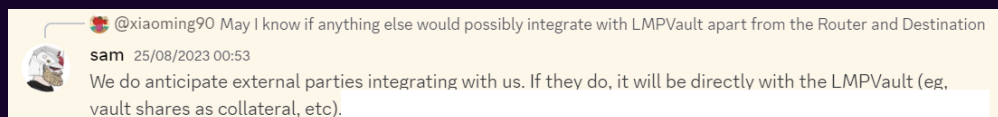
Trumpero commented:

low, not following ERC4626 won't incur risk for the users and protocol To make sure the `assetPreview` \leq `assetActual`, users (integrated protocol) should use router to redeem

xiaoming9090

Escalate

I have confirmed with the protocol team that they anticipate external parties integrating directly with the LMPVault (e.g., vault shares as collateral), as shown below. The router is not applicable in the context of this issue, as ERC4626 is strictly applied to the LMPVault only.



Per the judging docs, the issue will be considered as valid if there is external integrations.

EIP compliance with no integrations: If the protocol does not have external integrations then issues related to code not fully complying with the EIP it is implementing and there are no adverse effects of this, is considered informational

Also, the [contest page](#) explicitly mentioned that the `LMPVault` must conform with the ERC4626. Thus, issues related to EIP compliance is considered valid in the context of this audit.



Q: Is the code/contract expected to comply with any EIPs? Are there specific assumptions around adhering to those EIPs that Watsons should be aware of?

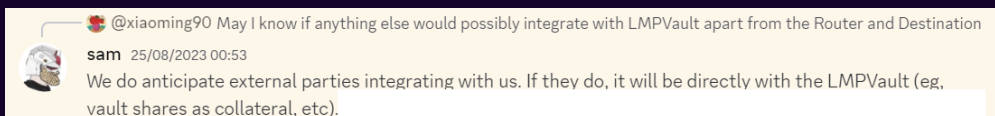
src/vault/LMPVault.sol should be 4626 compatible

In this case, non-conforming to ERC4626 is a valid Medium.

sherlock-admin2

Escalate

I have confirmed with the protocol team that they anticipate external parties integrating directly with the LMPVault (e.g., vault shares as collateral), as shown below. The router is not applicable in the context of this issue, as ERC4626 is strictly applied to the LMPVault only.



Per the judging docs, the issue will be considered as valid if there is external integrations.

EIP compliance with no integrations: If the protocol does not have external integrations then issues related to code not fully complying with the EIP it is implementing and there are no adverse effects of this, is considered informational

Also, the contest page explicitly mentioned that the LMPVault must conform with the ERC4626. Thus, issues related to EIP compliance is considered valid in the context of this audit.

Q: Is the code/contract expected to comply with any EIPs? Are there specific assumptions around adhering to those EIPs that Watsons should be aware of?

src/vault/LMPVault.sol should be 4626 compatible

In this case, non-conforming to ERC4626 is a valid Medium.

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.

midori-fuse

Escalate



If this issue ends up being valid post-escalation, then #452 #441 #319 #288 #202 should be dupes of this.

sherlock-admin2

Escalate

If this issue ends up being valid post-escalation, then #452 #441 #319 #288 #202 should be dupes of this.

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.

JeffCX

I thought the EIP comppliance issue is valid low in sherlock

JeffCX

<https://docs.sherlock.xyz/audits/judging/judging>

EIP compliance with no integrations: If the protocol does not have external integrations then issues related to code not fully complying with the EIP it is implementing and there are no adverse effects of this, are considered informational

Oxbtk

If #577 is considered a medium, then #412 should be a medium too, because as per of the EIP-4626:

It is considered most secure to favor the Vault itself during calculations over its users.

Kalyan-Singh

#656 shows how deposit function reverts under certain conditions due to maxDeposit not being eip compliant, I think that will be a genuine problem for external integrators. I think this escalations result should also be reflected there.

xiaoming9090

I thought the EIP comppliance issue is valid low in sherlock

For this contest, it was explicitly mentioned in the [contest page](#) that the LMPVault must conform with the ERC4626. Thus, issues related to EIP compliance is considered valid Medium in the context of this audit.



Q: Is the code/contract expected to comply with any EIPs? Are there specific assumptions around adhering to those EIPs that Watsons should be aware of?

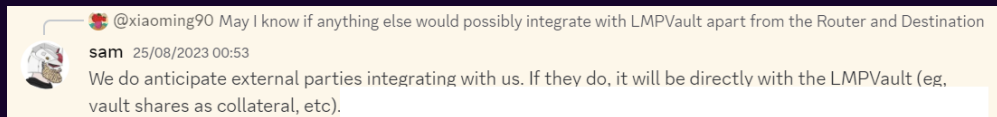
src/vault/LMPVault.sol should be 4626 compatible

xiaoming9090

<https://docs.sherlock.xyz/audits/judging/judging>

EIP compliance with no integrations: If the protocol does not have external integrations then issues related to code not fully complying with the EIP it is implementing and there are no adverse effects of this, are considered informational

I have confirmed with the protocol team that they anticipate external parties integrating directly with the LMPVault (e.g., vault shares as collateral), as shown below. Thus, there are external integrations.



OxSurena

I thought the EIP compliance issue is valid low in sherlock

For this contest, it was explicitly mentioned in the [contest page](#) that the LMPVault must conform with the ERC4626. Thus, issues related to EIP compliance is considered valid Medium in the context of this audit.

Q: Is the code/contract expected to comply with any EIPs? Are there specific assumptions around adhering to those EIPs that Watsons should be aware of? src/vault/LMPVault.sol should be 4626 compatible

Exactly, it was explicitly mentioned in the [contest page](#) that code/contract **expected / should** to comply with 4626.

@JeffCX

<https://docs.sherlock.xyz/audits/judging/judging>

EIP compliance with no integrations: If the protocol does not have external integrations then issues related to code not fully complying with the EIP it is implementing and there are no adverse effects of this, are considered informational

In case of conflict between information in the Contest README vs Sherlock rules, **the README overrides Sherlock rules.**



Trumpero

I believe this issue is a low/info issue evidently. Judging docs of Sherlock clearly stated that:

EIP compliance with no integrations: If the protocol does not have external integrations, then issues related to the code not fully complying with the EIP it is implementing, and there are no adverse effects of this, it is considered informational.

Therefore, it should be an informational issue.

The issue must cause a loss of funds (even if unlikely) to be considered as medium

Medium: There is a viable scenario (even if unlikely) that could cause the protocol to enter a state where a material amount of funds can be lost.

Furthermore, according to Sherlock's judging docs, the external integrations in the future are not considered valid issues:

Future issues: Issues that result out of a future integration/implementation that was not intended (mentioned in the docs/README) or because of a future change in the code (as a fix to another issue) are not valid issues.

Evert0x

Current opinion is to accept escalation and make issue medium because of the following judging rule.

In case of conflict between information in the README vs Sherlock rules, the README overrides Sherlock rules. <https://docs.sherlock.xyz/audits/judging/judging#iii.-some-standards-observed>

Trumpero

I believe this issue doesn't meet the requirements of a medium issue since it doesn't cause any loss, which is an important requirement to be a valid issue in Sherlock. Even without considering the Sherlock docs about EIP compliance, it only has a low impact, so I don't think it is a valid medium. Historically, the potential risk from a view function has never been accepted as a medium in Sherlock. Additionally, there is no potential loss since users should use the router to redeem, which protects users from any loss by using the minimum redeem amount.

midori-fuse

Historically, the potential risk from a view function has never been accepted as a medium in Sherlock

Disputing this point. [Evidence](#)

xiaoming9090



The README explicitly stated that `LMPVault.sol` should be 4626 compatible. README overwrites the Sherlock rules. Thus, any 4626 incompatible in this contest would be classified as Medium.

Evert0x

The core question is, does the README also override the severity classifications? It states that "Sherlock rules for valid issues" are overwritten. But it's unclear if the severity definitions are included in this, especially because the medium/high definitions are stated above this rule.

The intention of this sentence is that the protocol can thrive in the context defined by the protocol team.

In this case, it's clear that the team states that the `LMPVault.sol` should exist in the context of complete ERC4626 compatibility. Making this issue valid.

Planning to make some changes to the Medium definition and Hierarchy of truth language so it will be clear for future contests.

Evert0x

Planning to accept escalation and duplicate with
<https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/452>
<https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/441>
<https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/319>
<https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/288>
<https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/202>

Evert0x

Update: planning to add #412 #577 #656 #487 and categorize as a general "Failing to comply with ERC4626" issue family.

Evert0x

Result: Medium Has Duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- [xiaoming9090](#): accepted
- [midori-fuse](#): accepted

Evert0x

Will add #665 <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/465>, <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/503> and <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/731> as



duplicates as they point out an issue that will make the contract fail to comply with ERC4626.

Because `_maxMint()` has the possibility to return `uint256.max` it can break the ERC4626 specification of `maxDeposit` of "MUST NOT REVERT"

See <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/665#issuecomment-1780762436>



Issue M-7: Losses are not distributed equally

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/591>

Found by

xiaoming90

The losses are not distributed equally, leading to slower users suffering significant losses.

Vulnerability Detail

Assume that three (3) destination vaults (DVs) and the withdrawal queue are arranged in this order: DV_A , DV_B , DV_C .

Assume the following appreciation and depreciation of the price of the underlying LP tokens of the DV:

- Underlying LP Tokens of DV_A appreciate 5% every T period (Vault in Profit)
- Underlying LP Tokens of DV_B depreciate 5% every T period (Vault in Loss)
- Underlying LP Tokens of DV_C depreciate 10% every T period (Vault in Loss)

For simplicity's sake, all three (3) DVs have the same debt value.

In the current design, if someone withdraws the assets, they can burn as many DV_A shares as needed since DV_A is in profit. If DV_A manages to satisfy the withdrawal amount, the loop will stop here. If not, it will move to DV_B and DV_C to withdraw the remaining amount.

However, malicious users (also faster users) can abuse this design. Once they notice that LP tokens of DV_B and DV_C are depreciating, they could quickly withdraw as many shares as possible from the DV_A to minimize their loss. As shown in the chart below, once they withdrew all the assets in DV_A at T_{14} , the rest of the vault users would suffer a much faster rate of depreciation (~6%).

Thus, the loss of the LMPVault is not evenly distributed across all participants. The faster actors will incur less or no loss, while slower users suffer a more significant higher loss.



	DV_A	DV_B	DV_C	Total Debt Value	Rate of Loss OR Profit	
T0	10000	10000	10000	30000	0.00%	
T1	10500	9500	9000	29000	-3.33%	
T2	11025	9025	8100	28150	-2.93%	
T3	11576.25	8573.75	7290	27440	-2.52%	
T4	12155.06	8145.063	6561	26861.125	-2.11%	
T5	12762.82	7737.809	5904.9	26405.525	-1.70%	
T6	13400.96	7350.919	5314.41	26066.28531	-1.28%	
T7	14071	6983.373	4782.969	25837.34619	-0.88%	
T8	14774.55	6634.204	4304.672	25713.43085	-0.48%	
T9	15513.28	6302.494	3874.205	25689.98115	-0.09%	
T10	16288.95	5987.369	3486.784	25763.10006	0.28%	
T11	17103.39	5688.001	3138.106	25929.50046	0.65%	
T12	17958.56	5403.601	2824.295	26186.4595	0.99%	
T13	18856.49	5133.421	2541.866	26531.77808	1.32%	
T14	19799.32	4876.75	2287.679	26963.74503	1.63%	
T15		4632.912	2058.911	6691.823623	-75.18%	Excluded
T16		4401.267	1853.02	6254.286875	-6.54%	
T17		4181.203	1667.718	5848.921522	-6.48%	
T18		3972.143	1500.946	5473.089538	-6.43%	
T19		3773.536	1350.852	5124.387743	-6.37%	
T20		3584.859	1215.767	4800.62577	-6.32%	
T21		3405.616	1094.19	4499.806154	-6.27%	
T22		3235.335	984.7709	4220.106352	-6.22%	
T23		3073.569	886.2938	3959.862489	-6.17%	
T24		2919.89	797.6644	3717.554674	-6.12%	
T25		2773.896	717.898	3491.793719	-6.07%	
T26		2635.201	646.1082	3281.309134	-6.03%	
T27		2503.441	581.4974	3084.938267	-5.98%	
T28		2378.269	523.3476	2901.616486	-5.94%	



Impact

The losses are not distributed equally, leading to slower users suffering significant losses.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L37>

Tool used

Manual Review

Recommendation

Consider burning the shares proportionately across all the DVs during user withdrawal so that loss will be distributed equally among all users regardless of the withdrawal timing.



Issue M-8: Malicious users could lock in the NAV/Share of the DV to cause the loss of fees

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/601>

Found by

0x73696d616f, xiaoming90

Malicious users could lock in the NAV/Share of the destination vaults, resulting in a loss of fees.

Vulnerability Detail

The `_collectFees` function only collects fees whenever the NAV/Share exceeds the last NAV/Share.

During initialization, the `navPerShareHighMark` is set to 1, effectively 1 ETH per share (1:1 ratio). Assume the following:

- It is at the early stage, and only a few shares (0.5 shares) were minted in the LMPVault
- There is a sudden increase in the price of an LP token in a certain DV (Temporarily)
- `performanceFeeBps` is 10%

In this case, the debt value of DV's shares will increase, which will cause LMPVault's debt to increase. This event caused the `currentNavPerShare` to increase to 1.4 temporarily.

Someone calls the `permissionless updateDebtReporting` function. Thus, the profit will be $0.4 \text{ ETH} * 0.5 \text{ Shares} = 0.2 \text{ ETH}$, which is small due to the number of shares (0.5 shares) in the LMPVault at this point. The fee will be 0.02 ETH (~40 USD). Thus, the fee earned is very little and almost negligible.

At the end of the function, the `navPerShareHighMark` will be set to 1.4, and the highest NAV/Share will be locked forever. After some time, the price of the LP tokens fell back to its expected price range, and the `currentNavPerShare` fell to around 1.05. No fee will be collected from this point onwards unless the NAV/Share is raised above 1.4.

It might take a long time to reach the 1.4 threshold, or in the worst case, the spike is temporary, and it will never reach 1.4 again. So, when the NAV/Share of the LMPVault is 1.0 to 1.4, the protocol only collects 0.02 ETH (~40 USD), which is too little.



<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L800>

```
function _collectFees(uint256 idle, uint256 debt, uint256 totalSupply) internal {
    address sink = feeSink;
    uint256 fees = 0;
    uint256 shares = 0;
    uint256 profit = 0;

    // If there's no supply then there should be no assets and so nothing
    // to actually take fees on
    if (totalSupply == 0) {
        return;
    }

    uint256 currentNavPerShare = ((idle + debt) * MAX_FEE_BPS) / totalSupply;
    uint256 effectiveNavPerShareHighMark = navPerShareHighMark;

    if (currentNavPerShare > effectiveNavPerShareHighMark) {
        // Even if we aren't going to take the fee (haven't set a sink)
        // We still want to calculate so we can emit for off-chain analysis
        profit = (currentNavPerShare - effectiveNavPerShareHighMark) *
    ↪ totalSupply;
        fees = profit.mulDiv(performanceFeeBps, (MAX_FEE_BPS ** 2),
    ↪ Math.Rounding.Up);
        if (fees > 0 && sink != address(0)) {
            // Calculated separate from other mints as normal share mint is
    ↪ round down
            shares = _convertToShares(fees, Math.Rounding.Up);
            _mint(sink, shares);
            emit Deposit(address(this), sink, fees, shares);
        }
        // Set our new high water mark, the last nav/share height we took fees
        navPerShareHighMark = currentNavPerShare;
        navPerShareHighMarkTimestamp = block.timestamp;
        emit NewNavHighWatermark(currentNavPerShare, block.timestamp);
    }
    emit FeeCollected(fees, sink, shares, profit, idle, debt);
}
```

Impact

Loss of fee. Fee collection is an integral part of the protocol; thus the loss of fee is considered a High issue.



Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L800>

Tool used

Manual Review

Recommendation

Consider implementing a sophisticated off-chain algorithm to determine the right time to lock the `navPerShareHighMark` and/or restrict the access to the `updateDebtReporting` function to only protocol-owned addresses.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

Trumpero commented:

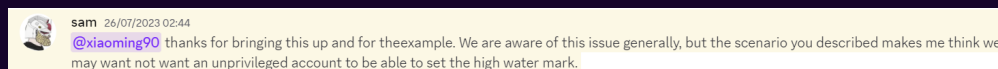
low, collecting fee based on NAV is protocol design, so I think it's fine when the NAV is too large and make the fee can't be accrued for the protocol, it's a risk from protocol's choice. Furthermore, in theory more deposits is equivalent with more rewards, so it's likely when the protocol grows bigger, the NAV should increase and the fee can be active again.

xiaoming9090

Escalate

First of all, a risk from the protocol's choice does not mean that the issue is invalid/low. Any risk arising from the protocol's design/architecture and its implementation should be highlighted during an audit.

I have discussed this with the protocol team during the audit period as shown below, and the impact of this issue is undesirable.



Using the example in my report, the period where it takes for the NAV/Share of the LMPVault to increase from 1.0 to 1.4 after the attack, the protocol only collects 0.02 ETH (~40 USD), which shows that the design of the accounting and collection of fees is fundamentally flawed. The protocol team might not have been aware of this attack/issue when designing this fee collection mechanism and assumed that the



NAV would progressively increase over a period of time, but did not expect this edge case.

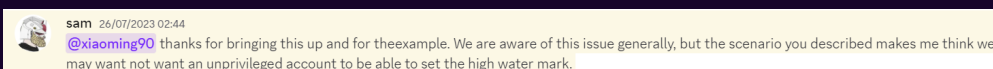
Thus, this is a valid High issue due to a loss of fee.

sherlock-admin2

Escalate

First of all, a risk from the protocol's choice does not mean that the issue is invalid/low. Any risk arising from the protocol's design/architecture and its implementation should be highlighted during an audit.

I have discussed this with the protocol team during the audit period as shown below, and the impact of this issue is undesirable.



Using the example in my report, the period where it takes for the NAV/Share of the LMPVault to increase from 1.0 to 1.4 after the attack, the protocol only collects 0.02 ETH (~40 USD), which shows that the design of the accounting and collection of fees is fundamentally flawed. The protocol team might not have been aware of this attack/issue when designing this fee collection mechanism and assumed that the NAV would progressively increase over a period of time, but did not expect this edge case.

Thus, this is a valid High issue due to a loss of 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.

Trumpero

@codenutt Could you please share your thoughts about this issue? Do you think it is a real issue of the current protocol's fee model which causes a loss of fee, or is it simply another choice of the fee model?

codenutt

@codenutt Could you please share your thoughts about this issue? Do you think it is a real issue of the current protocol's fee model which causes a loss of fee, or is it simply another choice of the fee model?

There are a lot of ways this issue can creep up. It could be something nefarious. It could just be a temporary price spike that would occur normally even if updateDebtReporting was permissioned. We have to do debt reporting fairly



frequenting or stale data starts affecting the strategy. However it happens though, we do generally recognize it as an issue and a change we have planned (something we were still solidifying going into the audit). We'll be implementing some decay logic around the high water mark so if we don't see a rise after say 90 days it starts to lower.

Evert0x

Planning to accept escalation and make issue medium as the issue rests on some assumptions, it's also unclear how significant the potential losses exactly are.

Evert0x

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- xiaoming9090: accepted

Evert0x

Update #469 is a duplicate



Issue M-9: Price returned by Oracle is not verified

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/604>

Found by

xiaoming90

The price returned by the oracle is not adequately verified, leading to incorrect pricing being accepted.

Vulnerability Detail

As per the [example](#) provided by Tellor on how to integrate the Tellor oracle into the system, it has shown the need to check that the price returned by the oracle is not zero.

<https://github.com/tellor-io/tellor-caller-liquity/blob/3ed91e4b15f0bac737f2966e36805ade3daf8162/contracts/TellorCaller.sol#L51C34-L51C34>

```
function getTellorCurrentValue(bytes32 _queryId)
    ..SNIP..
    // retrieve most recent 20+ minute old value for a queryId. the time buffer
    ↪ allows time for a bad value to be disputed
    (, bytes memory data, uint256 timestamp) = tellor.getDataBefore(_queryId,
    ↪ block.timestamp - 20 minutes);
    uint256 _value = abi.decode(data, (uint256));
    if (timestamp == 0 || _value == 0) return (false, _value, timestamp);
```

Thus, the value returned from the `getDataBefore` function should be verified to ensure that the price returned by the oracle is not zero. However, this was not implemented.

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/oracles/providers/TellorOracle.sol#L101>

```
File: TellorOracle.sol
101:     function getPriceInEth(address tokenToPrice) external returns (uint256)
    ↪ {
102:         TellorInfo memory tellorInfo = _getQueryInfo(tokenToPrice);
103:         uint256 timestamp = block.timestamp;
104:         // Giving time for Tellor network to dispute price
105:         (bytes memory value, uint256 timestampRetrieved) =
    ↪ getDataBefore(tellorInfo.queryId, timestamp - 30 minutes);
106:         uint256 tellorStoredTimeout = uint256(tellorInfo.pricingTimeout);
107:         uint256 tokenPricingTimeout = tellorStoredTimeout == 0 ?
    ↪ DEFAULT_PRICING_TIMEOUT : tellorStoredTimeout;
```



```

108:
109:         // Check that something was returned and freshness of price.
110:         if (timestampRetrieved == 0 || timestamp - timestampRetrieved >
↳ tokenPricingTimeout) {
111:             revert InvalidDataReturned();
112:         }
113:
114:         uint256 price = abi.decode(value, (uint256));
115:         return _denominationPricing(tellorInfo.denomination, price,
↳ tokenToPrice);
116:     }

```

Impact

The protocol relies on the oracle to provide accurate pricing for many critical operations, such as determining the debt values of DV, calculators/stats used during the rebalancing process, NAV/shares of the LMPVault, and determining how much assets the users should receive during withdrawal.

If an incorrect value of zero is returned from Tellor, affected assets within the protocol will be considered worthless.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/oracles/providers/TellorOracle.sol#L101>

Tool used

Manual Review

Recommendation

Update the affected function as follows.

```

function getPriceInEth(address tokenToPrice) external returns (uint256) {
    TellorInfo memory tellorInfo = _getQueryInfo(tokenToPrice);
    uint256 timestamp = block.timestamp;
    // Giving time for Tellor network to dispute price
    (bytes memory value, uint256 timestampRetrieved) =
↳ getDataBefore(tellorInfo.queryId, timestamp - 30 minutes);
    uint256 tellorStoredTimeout = uint256(tellorInfo.pricingTimeout);
    uint256 tokenPricingTimeout = tellorStoredTimeout == 0 ?
↳ DEFAULT_PRICING_TIMEOUT : tellorStoredTimeout;

```



```

    // Check that something was returned and freshness of price.
-   if (timestampRetrieved == 0 || timestamp - timestampRetrieved >
↳   tokenPricingTimeout) {
+   if (timestampRetrieved == 0 || value == 0 || timestamp - timestampRetrieved
↳   > tokenPricingTimeout) {
        revert InvalidDataReturned();
    }

    uint256 price = abi.decode(value, (uint256));
    return _denominationPricing(tellorInfo.denomination, price, tokenToPrice);
}

```

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

Trumpero commented:

low, similar to chainlink round completeness

xiaoming9090

Escalate

The reason why chainlink round completeness is considered invalid/low in Sherlock is that the OCR does not rely on rounds for reporting anymore. Not validating the price returned from the oracle is zero is a much more serious issue and is different from the round completeness issue related to lagged/outdated price.

If an incorrect value of zero is returned from Tellor, affected assets within the protocol will be considered worthless. Thus, it is important that this check must be done. In addition, per the [example](#) provided by Tellor on how to integrate the Tellor oracle into the system, it has shown the need to check that the price returned by the oracle is not zero.

Thus, this is a valid High issue.

sherlock-admin2

Escalate

The reason why chainlink round completeness is considered invalid/low in Sherlock is that the OCR does not rely on rounds for reporting anymore. Not validating the price returned from the oracle is zero is a much more serious issue and is different from the round completeness issue related to lagged/outdated price.



If an incorrect value of zero is returned from Tellor, affected assets within the protocol will be considered worthless. Thus, it is important that this check must be done. In addition, per the example provided by Tellor on how to integrate the Tellor oracle into the system, it has shown the need to check that the price returned by the oracle is not zero.

Thus, this is a valid High issue.

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.

Trumpero

Hello @xiaoming9090, according to the information I found in the Tellor documentation under the section "Solidity Integration," the provided example in the documentation differs from the example you provided, and it lacks a check for the 0-value.

Could you please provide more details about the source of your example?

xiaoming9090

Hey @Trumpero, the repo can be found in the Tellor's User Checklist, which is a guide that the protocol team need to go through before using the Tellor oracle.

One point to make is that Chainlink or Tellor would not provide an example that checks for zero-value in all their examples. The reason is that not all use cases require zero-value checks. Suppose a simple protocol performs a non-price-sensitive operation, such as fetching a price to emit an event for the protocol team's internal reference. In that case, there is no need to check for zero value.

However, for Tokemak and many other protocols involving financial transactions, it is critical that the price of assets cannot be zero due to errors from Oracle. Thus, a zero-values check is consistently implemented on such kind of protocols. Therefore, we need to determine if a zero-values check is required on a case-by-case basis. In this case, Tokemak falls under the latter group.

Trumpero

Tks @xiaoming9090, understand it now. Seem a valid issue for me @codenutt. The severity for me should be medium, since it assumes the tellor oracle returns a 0 price value.

Evert0x

Planning to accept escalation and make issue medium



codenutt

@Trumpero Our goal with the check is to verify that a price was actually found. Based on their contract checking for `timestamp == 0` is sufficient as it returns both 0 and 0 in this state: <https://github.com/tellor-io/tellorFlex/blob/bdefcab6d90d4e86c34253fdc9e1ec778f370c3c/contracts/TellorFlex.sol#L450>

Trumpero

Based on the sponsor's comment, I think this issue is low.

Evert0x

Planning to reject escalation and keep issue state as is @xiaoming9090

xiaoming9090

The sponsor's comment simply explains the purpose/intention of the if-condition is to check that a price is returned from Tellor Oracle. However, this does not necessarily mean that it is fine that the returned price is zero OR there is no risk if the return price is zero.

The protocol uses two (2) oracles (Chainlink and Tellor). In their Chainlink oracle's implementation, the protocol has explicitly checked that the price returned is more than zero. Otherwise, the oracle will revert. Thus, it is obvious that zero price is not accepted to the protocol based on the codebase.

<https://github.com/sherlock-audit/2023-06-tokemak/blob/5d8e902ce33981a6506b1b5fb979a084602c6c9a/v2-core-audit-2023-07-14/src/oracles/providers/ChainlinkOracle.sol#L113>

```
if (
    roundId == 0 || price <= 0 || updatedAt == 0 || updatedAt > timestamp
    || updatedAt < timestamp - tokenPricingTimeout
) revert InvalidDataReturned();
```

However, this was not consistently implemented in their Tellor's oracle, which is highlighted in this report.

In addition, as pointed out in my earlier escalation. for Tokemak and many other protocols involving financial transactions, it is critical that the price of assets cannot be zero. Thus, a zero-values check is consistently implemented on such kind of protocols. Some examples are as follows:

- AAVE - [Source Code](#) and [Documentation](#)
- Compound - [Source Code](#)

Evert0x

Thanks @xiaoming9090



The core issue is that 0 value isn't handled well. There is no counter argument to this in the recent comments.

Planning to accept escalation and make issue medium

Evert0x

@Trumpero let me know if you agree with this.

Trumpero

I agree that this issue should be a medium within the scope of Tokemak contracts.

Evert0x

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- xiaoming9090: accepted



Issue M-10: Malicious users could use back old values

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/612>

Found by

ctf_sec, xiaoming90

Malicious users could use back old values to manipulate the price.

Vulnerability Detail

Per the Teller's User Checklist, it is possible that a potential attacker could go back in time to find a desired value in the event that a Tellor value is disputed. Following is the extract taken from the checklist:

Ensure that functions do not use old Tellor values

In the event where a Tellor value is disputed, the disputed value is removed & previous values remain. Prevent potential attackers from going back in time to find a desired value with a check in your contracts. This repo is a great reference for integrating Tellor.

The current implementation lack measure to guard against such attack.

```
File: TellorOracle.sol
101:     function getPriceInEth(address tokenToPrice) external returns (uint256)
    ↪ {
102:         TellorInfo memory tellorInfo = _getQueryInfo(tokenToPrice);
103:         uint256 timestamp = block.timestamp;
104:         // Giving time for Tellor network to dispute price
105:         (bytes memory value, uint256 timestampRetrieved) =
    ↪ getDataBefore(tellorInfo.queryId, timestamp - 30 minutes);
106:         uint256 tellorStoredTimeout = uint256(tellorInfo.pricingTimeout);
107:         uint256 tokenPricingTimeout = tellorStoredTimeout == 0 ?
    ↪ DEFAULT_PRICING_TIMEOUT : tellorStoredTimeout;
108:
109:         // Check that something was returned and freshness of price.
110:         if (timestampRetrieved == 0 || timestamp - timestampRetrieved >
    ↪ tokenPricingTimeout) {
111:             revert InvalidDataReturned();
112:         }
113:
114:         uint256 price = abi.decode(value, (uint256));
115:         return _denominationPricing(tellorInfo.denomination, price,
    ↪ tokenToPrice);
116:     }
```



Anyone can submit a dispute to Tellor by paying a fee. The disputed values are immediately removed upon submission, and the previous values will remain. The attacks are profitable as long as the economic gains are higher than the dispute fee. For instance, this can be achieved by holding large amounts of vault shares (e.g., obtained using own funds or flash-loan) to amplify the gain before manipulating the assets within it to increase the values.

Impact

Malicious users could manipulate the price returned by the oracle to be higher or lower than expected. The protocol relies on the oracle to provide accurate pricing for many critical operations, such as determining the debt values of DV, calculators/stats used during the rebalancing process, NAV/shares of the LMPVault, and determining how much assets the users should receive during withdrawal.

Incorrect pricing would result in many implications that lead to a loss of assets, such as users withdrawing more or fewer assets than expected due to over/undervalued vaults or strategy allowing an unprofitable rebalance to be executed.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/oracles/providers/TellorOracle.sol#L101>

Tool used

Manual Review

Recommendation

Update the affected function as per the recommendation in [Teller's User Checklist](#).

```
function getPriceInEth(address tokenToPrice) external returns (uint256) {
    TellorInfo memory tellorInfo = _getQueryInfo(tokenToPrice);
    uint256 timestamp = block.timestamp;
    // Giving time for Tellor network to dispute price
    (bytes memory value, uint256 timestampRetrieved) =
    ↪ getDataBefore(tellorInfo.queryId, timestamp - 30 minutes);
    uint256 tellorStoredTimeout = uint256(tellorInfo.pricingTimeout);
    uint256 tokenPricingTimeout = tellorStoredTimeout == 0 ?
    ↪ DEFAULT_PRICING_TIMEOUT : tellorStoredTimeout;

    // Check that something was returned and freshness of price.
    if (timestampRetrieved == 0 || timestamp - timestampRetrieved >
    ↪ tokenPricingTimeout) {
```



```

        revert InvalidDataReturned();
    }

+   if (timestampRetrieved > lastStoredTimestamps[tellorInfo.queryId]) {
+       lastStoredTimestamps[tellorInfo.queryId] = timestampRetrieved;
+       lastStoredPrices[tellorInfo.queryId] = value;
+   } else {
+       value = lastStoredPrices[tellorInfo.queryId]
+   }

    uint256 price = abi.decode(value, (uint256));
    return _denominationPricing(tellorInfo.denomination, price, tokenToPrice);
}

```

Discussion

xiaoming9090

Escalate

This is wrongly duplicated to Issue 744. In Issue 744, it highlights that the 30-minute delay is too large, which is different from the issue highlighted here. This issue is about a different vulnerability highlighted in Tellor checklist where malicious users can re-use back an old value that has nothing to do with the 30-minute delay being too large.

sherlock-admin2

Escalate

This is wrongly duplicated to Issue 744. In Issue 744, it highlights that the 30-minute delay is too large, which is different from the issue highlighted here. This issue is about a different vulnerability highlighted in Tellor checklist where malicious users can re-use back an old value that has nothing to do with the 30-minute delay being too large.

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.

JeffCX

Escalate

Agree with LSW.



#844 is a duplicate of this issue as well, the issue that only highlight the stale price feed should be de-dup together! Thanks!

sherlock-admin2

Escalate

Agree with LSW.

#844 is a duplicate of this issue as well, the issue that only highlight the stale price feed should be de-dup together! Thanks!

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.

Trumpero

Agree with the @xiaoming9090, this issue should not be a duplication of #744 Please check this as well @codenutt

Evert0x

Planning to accept both escalations and make #612 a valid medium with #844 as a duplicate.

codenutt

Agree with the @xiaoming9090, this issue should not be a duplication of #744 Please check this as well @codenutt

Yup agree. Thx!

Evert0x

Result: Medium Has Duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- xiaoming9090: accepted
- JEFFCX: accepted



Issue M-11: Incorrect handling of Stash Tokens within the `ConvexRewardsAdapter._claimRewards()`

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/632>

Found by

duc, nobody2018 The `ConvexRewardsAdapter._claimRewards()` function incorrectly handles Stash tokens, leading to potential vulnerabilities.

Vulnerability Detail

The primary task of the `ConvexRewardAdapter._claimRewards()` function revolves around claiming rewards for Convex/Aura staked LP tokens.

An intriguing aspect of this function's logic lies in its management of "stash tokens" from AURA staking. The check to identify whether `rewardToken[i]` is a stash token involves attempting to invoke `IERC20(rewardTokens[i]).totalSupply()`. If the returned total supply value is 0, the implementation assumes the token is a stash token and bypasses it. However, this check is flawed since the total supply of stash tokens can indeed be non-zero. For instance, at this [address](#), the stash token has `totalSupply = 150467818494283559126567`, which is definitely not zero.

This misstep in checking can potentially lead to a Denial-of-Service (DOS) situation when calling the `claimRewards()` function. This stems from the erroneous attempt to call the `balanceOf` function on stash tokens, which lack the `balanceOf()` method. Consequently, such incorrect calls might incapacitate the destination vault from claiming rewards from AURA, resulting in protocol losses.

Impact

- The `AuraRewardsAdapter.claimRewards()` function could suffer from a Denial-of-Service (DOS) scenario.
- The destination vault's ability to claim rewards from AURA staking might be hampered, leading to protocol losses.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/destinations/adapters/rewards/ConvexRewardsAdapter.sol#L80-L86>

Tool used

Manual Review



Recommendation

To accurately determine whether a token is a stash token, it is advised to perform a low-level `balanceOf()` call to the token and subsequently validate the call's success.

Discussion

sherlock-admin2

Escalate

#649 is a duplicate of this issue.

You've deleted an escalation for this issue.



Issue M-12: navPerShareHighMark not reset to 1.0

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/661>

Found by

berndartmueller, xiaoming90

The navPerShareHighMark was not reset to 1.0 when a vault had been fully exited, leading to a loss of fee.

Vulnerability Detail

The LMPVault will only collect fees if the current NAV (currentNavPerShare) is more than the last NAV (effectiveNavPerShareHighMark).

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L800>

```
File: LMPVault.sol
800:     function _collectFees(uint256 idle, uint256 debt, uint256 totalSupply)
    ↳ internal {
801:         address sink = feeSink;
802:         uint256 fees = 0;
803:         uint256 shares = 0;
804:         uint256 profit = 0;
805:
806:         // If there's no supply then there should be no assets and so
    ↳ nothing
807:         // to actually take fees on
808:         if (totalSupply == 0) {
809:             return;
810:         }
811:
812:         uint256 currentNavPerShare = ((idle + debt) * MAX_FEE_BPS) /
    ↳ totalSupply;
813:         uint256 effectiveNavPerShareHighMark = navPerShareHighMark;
814:
815:         if (currentNavPerShare > effectiveNavPerShareHighMark) {
816:             // Even if we aren't going to take the fee (haven't set a sink)
817:             // We still want to calculate so we can emit for off-chain
    ↳ analysis
818:             profit = (currentNavPerShare - effectiveNavPerShareHighMark) *
    ↳ totalSupply;
```

Assume the current LMPVault state is as follows:



- totalAssets = 15 WETH
- totalSupply = 10 shares
- NAV/share = 1.5
- effectiveNavPerShareHighMark = 1.5

Alice owned all the remaining shares in the vault, and she decided to withdraw all her 10 shares. As a result, the totalAssets and totalSupply become zero. It was found that when all the shares have been exited, the effectiveNavPerShareHighMark is not automatically reset to 1.0.

Assume that at some point later, other users started to deposit into the LMPVault, and the vault invests the deposited WETH to profitable destination vaults, resulting in the real/actual NAV rising from 1.0 to 1.49 over a period of time.

The system is designed to collect fees when there is a rise in NAV due to profitable investment from sound rebalancing strategies. However, since the effectiveNavPerShareHighMark has been set to 1.5 previously, no fee is collected when the NAV rises from 1.0 to 1.49, resulting in a loss of fee.

Impact

Loss of fee. Fee collection is an integral part of the protocol; thus the loss of fee is considered a High issue.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L448>

Tool used

Manual Review

Recommendation

Consider resetting the navPerShareHighMark to 1.0 whenever a vault has been fully exited.

```
function _withdraw(  
    uint256 assets,  
    uint256 shares,  
    address receiver,  
    address owner  
) internal virtual returns (uint256) {  
    ..SNIP..
```



```
        _burn(owner, shares);

+   if (totalSupply() == 0) navPerShareHighMark = MAX_FEE_BPS;

        emit Withdraw(msg.sender, receiver, owner, returnedAssets, shares);

        _baseAsset.safeTransfer(receiver, returnedAssets);

        return returnedAssets;
    }
```

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

Trumpero commented:

invalid, the fee is only collected when navPerShareHighMark increases. In the example, it should collect fee only at the first rising of NAV from 1.0 to 1.5; the second rising (even after withdrawing all) isn't counted.

xiaoming9090

Escalate

Please double-check with the protocol team. The protocol should always collect a fee when a profit is earned from the vaults. The fact that it only collects fees during the first rise of NAV, but not the second rise of NAV is a flaw in the design and leads to loss of protocol fee. Thus, this is a valid High issue.

sherlock-admin2

Escalate

Please double-check with the protocol team. The protocol should always collect a fee when a profit is earned from the vaults. The fact that it only collects fees during the first rise of NAV, but not the second rise of NAV is a flaw in the design and leads to loss of protocol fee. Thus, this is a valid High issue.

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.

Trumpero



@codenutt, I believe this issue revolves around the protocol's intention regarding how to handle the protocol fee. Given that @xiaoming9090 and I hold differing viewpoints on this intention, I would greatly appreciate hearing your thoughts on the matter.

codenutt

Agree with @xiaoming9090 here. We should be resetting the high water mark. Do believe it should only be a Medium, though. We've actually already fixed it as it was brought up as a Low issue in our parallel Halborn audit as well. If there was a complete outflow of funds and the high water mark was high enough that we wouldn't see us over coming it any time soon, there'd be no reason we wouldn't just shut down the vault and spin up a new one.

xiaoming9090

@codenutt Thanks for your response!

@hrishibhat @Trumpero Fee collection is an integral part of the protocol. Based on Sherlock's judging criteria and historical decisions, loss of protocol fee revenue is considered a Medium and above.

Evert0x

Planning to accept escalation and make issue medium.

If @xiaoming9090 can provide a strong argument for high, I will consider assigning high severity. But it's unclear if the losses are material.

Evert0x

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- xiaoming9090: accepted



Issue M-13: Vault cannot be added back into the vault registry

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/674>

Found by

0x70C9, AuditorPraise, Aymen0909, KingNFT, Phantasmagoria, ast3ros, carrotsmugger, dipp, techOptimizor, xiaoming90

The vault registry does not clear the vault type mapping when removing a vault, which prevents the same vault from being added back later.

Vulnerability Detail

When removing a vault from the registry, all states related to the vaults such as the `_vaults`, `_assets`, `_vaultsByAsset` are cleared except the `_vaultsByType` state.

```
function removeVault(address vaultAddress) external onlyUpdater {
    Errors.verifyNotZero(vaultAddress, "vaultAddress");

    // remove from vaults list
    if (!_vaults.remove(vaultAddress)) revert VaultNotFound(vaultAddress);

    address asset = ILMPVault(vaultAddress).asset();

    // remove from assets list if this was the last vault for that asset
    if (_vaultsByAsset[asset].length() == 1) {
        //slither-disable-next-line unused-return
        _assets.remove(asset);
    }

    // remove from vaultsByAsset mapping
    if (!_vaultsByAsset[asset].remove(vaultAddress)) revert
        ↪ VaultNotFound(vaultAddress);

    emit VaultRemoved(asset, vaultAddress);
}
```

<https://github.com/sherlock-audit/2023-06-tokemak/blob/5d8e902ce33981a6506b1b5fb979a084602c6c9a/v2-core-audit-2023-07-14/src/vault/LMPVaultRegistry.sol#L64-L82>

The uncleared `_vaultsByType` state will cause the `addVault` function to revert when trying to add the vault back into the registry even though the vault does not exist in



the registry anymore.

```
if (!_vaultsByType[vaultType].add(vaultAddress)) revert  
↳ VaultAlreadyExists(vaultAddress);
```

<https://github.com/sherlock-audit/2023-06-tokemak/blob/5d8e902ce33981a6506b1b5fb979a084602c6c9a/v2-core-audit-2023-07-14/src/vault/LMPVaultRegistry.sol#L59>

Impact

The addVault function is broken in the edge case when the updater tries to add the vault back into the registry after removing it. It affects all the operations of the protocol that rely on the vault registry.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/5d8e902ce33981a6506b1b5fb979a084602c6c9a/v2-core-audit-2023-07-14/src/vault/LMPVaultRegistry.sol#L64-L82>

Tool used

Manual Review

Recommendation

Clear the _vaultsByType state when removing the vault from the registry.

```
function removeVault(address vaultAddress) external onlyUpdater {  
    Errors.verifyNotZero(vaultAddress, "vaultAddress");  
+    ILMPVault vault = ILMPVault(vaultAddress);  
+    bytes32 vaultType = vault.vaultType();  
  
    // remove from vaults list  
    if (!_vaults.remove(vaultAddress)) revert  
↳ VaultNotFound(vaultAddress);  
  
    address asset = ILMPVault(vaultAddress).asset();  
  
    // remove from assets list if this was the last vault for that asset  
    if (_vaultsByAsset[asset].length() == 1) {  
        //slither-disable-next-line unused-return  
        _assets.remove(asset);  
    }  
}
```



```
        }

        // remove from vaultsByAsset mapping
        if (!_vaultsByAsset[asset].remove(vaultAddress)) revert
↪ VaultNotFound(vaultAddress);
+         if (!_vaultsByType[vaultType].remove(vaultAddress)) revert
↪ VaultNotFound(vaultAddress);

        emit VaultRemoved(asset, vaultAddress);
    }
}
```



Issue M-14: LMPVault.updateDebtReporting could underflow because of subtraction before addition

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/675>

Found by

0x007 debt = totalDebt - prevNTotalDebt + afterNTotalDebt in LMPVault._updateDebtReporting could underflow and breaking a core functionality of the protocol.

Vulnerability Detail

debt = totalDebt - prevNTotalDebt + afterNTotalDebt where prevNTotalDebt equals (destInfo.currentDebt * originalShares) / Math.max(destInfo.ownedShares, 1) and the key to finding a scenario for underflow starts by noting that each value deducted from totalDebt is calculated as cachedCurrentDebt.mulDiv(sharesToBurn, cachedDvShares, Math.Rounding.Up)

LMPDebt

```
...
L292     totalDebtBurn = cachedCurrentDebt.mulDiv(sharesToBurn, cachedDvShares,
↳ Math.Rounding.Up);
...
L440     uint256 currentDebt = (destInfo.currentDebt * originalShares) /
↳ Math.max(destInfo.ownedShares, 1);
L448     totalDebtDecrease = currentDebt;
```

Let: totalDebt = destInfo.currentDebt = destInfo.debtBasis =
cachedCurrentDebt = cachedDebtBasis = 11 totalSupply = destInfo.ownedShares
= cachedDvShares = 10

That way: cachedCurrentDebt * 1 / cachedDvShares = 1.1 but totalDebtBurn
would be rounded up to 2

sharesToBurn could easily be 1 if there was a loss that changes the ratio from 1:1.1
to 1:1. Therefore currentDvDebtValue = 10 * 1 = 10

```
if (currentDvDebtValue < updatedDebtBasis) {
    // We are currently sitting at a loss. Limit the value we can pull from
    // the destination vault
    currentDvDebtValue = currentDvDebtValue.mulDiv(userShares, totalVaultShares,
↳ Math.Rounding.Down);
    currentDvShares = currentDvShares.mulDiv(userShares, totalVaultShares,
↳ Math.Rounding.Down);
}
```



```
// Shouldn't pull more than we want
// Or, we're not in profit so we limit the pull
if (currentDvDebtValue < maxAssetsToPull) {
    maxAssetsToPull = currentDvDebtValue;
}

// Calculate the portion of shares to burn based on the assets we need to pull
// and the current total debt value. These are destination vault shares.
sharesToBurn = currentDvShares.mulDiv(maxAssetsToPull, currentDvDebtValue,
    ↪ Math.Rounding.Up);
```

Steps

- call redeem 1 share and previewRedeem request 1 `maxAssetsToPull`
- 2 debt would be burn
- Therefore $\text{totalDebt} = 11 - 2 = 9$
- call another redeem 1 share and request another 1 `maxAssetsToPull`
- 2 debts would be burn again and
- totalDebt would be 7, but $\text{prevNTotalDebt} = 11 * 8 // 10 = 8$

Using 1, 10 and 11 are for illustration and the underflow could occur in several other ways. E.g if we had used 100,001, 1,000,010 and 1,000,011 respectively.

Impact

`_updateDebtReporting` could underflow and break a very important core functionality of the protocol. `updateDebtReporting` is so critical that funds could be lost if it doesn't work. Funds could be lost both when the vault is in profit or at loss.

If in profit, users would want to call `updateDebtReporting` so that they get more asset for their shares (based on the profit).

If in loss, the whole vault asset is locked and withdrawals won't be successful because the Net Asset Value is not supposed to reduce by such action (`noNavDecrease` modifier). Net Asset Value has reduced because the loss would reduce `totalDebt`, but the only way to update the `totalDebt` record is by calling `updateDebtReporting`. And those impacted the most are those with large funds. The bigger the fund, the more NAV would decrease by withdrawals.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-202>



[3-07-14/src/vault/LMPVault.sol#L781-L792](https://github.com/sherlock-audit/2023-07-14/src/vault/LMPVault.sol#L781-L792) <https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/libs/LMPDebt.sol#L440-L449> <https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/libs/LMPDebt.sol#L295>

Tool used

Manual Review

Recommendation

Add before subtracting. ETH in circulation is not enough to cause an overflow.

```
- debt = totalDebt - prevNTotalDebt + afterNTotalDebt
+ debt = totalDebt + afterNTotalDebt - prevNTotalDebt
```

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

Trumpero commented:

invalid, let get through each step in the description: 0, we have, totalSupply = 10, totalDebt = 11 1, redeem 1 share --> totalSupply = 10 - 1 = 9 2, 2 debts will be burn -> totalDebt = 11 - 2 = 9 --> invalid here because the withdraw function contains the noNAVDecrease, but as we can see here is newNav = 9 / 9 = 1 < oldNav = 11 / 10 = 1.1

bizzyvinci

Escalate

This issue is valid.

Using 10 and 11 is just for illustration and there are many possibilities as pointed out in the report as well.

Using 1, 10 and 11 are for illustration and the underflow could occur in several other ways. E.g if we had used 100,001, 1,000,010 and 1,000,011 respectively.

And here's a PoC that could be added to [LMPVaultMintingTests](#) which would revert due to arithmetic underflow in updateDebtReporting

```
function test_675() public {
    uint num0 = 1_000_000;
```



```

uint num1 = 1_000_001;
uint num2 = 1_000_002;

// Mock root price of underlyerOne to 1,000,001 (ether-like)
// test debtvalue is correct and where you want it
_mockRootPrice(address(_underlyerOne), num1 * 1e12);
assertEq(num1, _destVaultOne.debtValue(num0));

// deposit
// rebalance
_accessController.grantRole(Roles.SOLVER_ROLE, address(this));
_accessController.grantRole(Roles.LMP_FEE_SETTER_ROLE, address(this));

// User is going to deposit 1,000,010 asset
_asset.mint(address(this), num0);
_asset.approve(address(_lmpVault), num0);
_lmpVault.deposit(num0, address(this));

// Deployed all asset to DV1
_underlyerOne.mint(address(this), num1);
_underlyerOne.approve(address(_lmpVault), num1);
_lmpVault.rebalance(
    address(_destVaultOne),
    address(_underlyerOne), // tokenIn
    num1,
    address(0), // destinationOut, none when sending out baseAsset
    address(_asset), // baseAsset, tokenOut
    num0
);

// totalDebt should equal num2 now
assertEq(num2, _lmpVault.totalDebt());
assertEq(num2, _lmpVault.totalAssets());

// also get mock priceback where we want it
_mockRootPrice(address(_underlyerOne), 1 ether);
assertEq(num0, _destVaultOne.debtValue(num0));

// start attack
_lmpVault.redeem(1, address(this), address(this));
assertEq(num2 - 2, _lmpVault.totalDebt());
assertEq(num2 - 2, _lmpVault.totalAssets());

_lmpVault.redeem(1, address(this), address(this));
assertEq(num2 - 4, _lmpVault.totalDebt());
assertEq(num2 - 4, _lmpVault.totalAssets());

```



```
        _lmpVault.updateDebtReporting(_destinations);  
    }  
}
```

In this case I used 1,000,000 against 1 wei, therefore it's **within NAV change tolerance**. And it illustrates the root cause I'm trying to point out.

```
debt = totalDebt - prevNTotalDebt + afterNTotalDebt
```

totalDebt **could be less than** prevNTotalDebt.

prevNTotalDebt is currentDebt which is debt during rebalance scaled to reflect current supply.

```
uint256 currentDebt = (destInfo.currentDebt * originalShares) /  
    ↪ Math.max(destInfo.ownedShares, 1);
```

However, a value that is rounded up is subtracted from totalDebt.

```
// https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/libs/LMPDebt.sol#L295C9-L295C98  
    ↪ 3-07-14/src/vault/libs/LMPDebt.sol#L295C9-L295C98  
totalDebtBurn = cachedCurrentDebt.mulDiv(sharesToBurn, cachedDvShares,  
    ↪ Math.Rounding.Up);  
  
// https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L489  
    ↪ 3-07-14/src/vault/LMPVault.sol#L489  
info.debtDecrease += totalDebtBurn;  
  
// https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L518C13-L518C44  
    ↪ 3-07-14/src/vault/LMPVault.sol#L518C13-L518C44  
totalDebt -= info.debtDecrease;
```

So, the goal is to get rounding up against totalDebt that is less than the rounding down in prevNTotalDebt.

The PoC achieves that by getting a double rounding up against totalDebt, which would result in just 1 rounding down in prevNTotalDebt. If you notice the attack section, each redemption of 1 share would burn 2 debts (instead of burning 1.000_001 debt because of rounding up).

P.S: Big Kudos to @Trumpero for taking the time to comment on invalid issues

sherlock-admin2

Escalate

This issue is valid.



Using 10 and 11 is just for illustration and there are many possibilities as pointed out in the report as well.

Using 1, 10 and 11 are for illustration and the underflow could occur in several other ways. E.g if we had used 100,001, 1,000,010 and 1,000,011 respectively.

And here's a PoC that could be added to LMPVaultMintingTests which would revert due to arithmetic underflow in updateDebtReporting

```
function test_675() public {
    uint num0 = 1_000_000;
    uint num1 = 1_000_001;
    uint num2 = 1_000_002;

    // Mock root price of underlyerOne to 1,000,001 (ether-like)
    // test debtvalue is correct and where you want it
    _mockRootPrice(address(_underlyerOne), num1 * 1e12);
    assertEq(num1, _destVaultOne.debtValue(num0));

    // deposit
    // rebalance
    _accessController.grantRole(Roles.SOLVER_ROLE, address(this));
    _accessController.grantRole(Roles.LMP_FEE_SETTER_ROLE,
    ↪ address(this));

    // User is going to deposit 1,000,010 asset
    _asset.mint(address(this), num0);
    _asset.approve(address(_lmpVault), num0);
    _lmpVault.deposit(num0, address(this));

    // Deployed all asset to DV1
    _underlyerOne.mint(address(this), num1);
    _underlyerOne.approve(address(_lmpVault), num1);
    _lmpVault.rebalance(
        address(_destVaultOne),
        address(_underlyerOne), // tokenIn
        num1,
        address(0), // destinationOut, none when sending out baseAsset
        address(_asset), // baseAsset, tokenOut
        num0
    );

    // totalDebt should equal num2 now
    assertEq(num2, _lmpVault.totalDebt());
    assertEq(num2, _lmpVault.totalAssets());
}
```



```

        // also get mock price back where we want it
        _mockRootPrice(address(_underlyerOne), 1 ether);
        assertEq(num0, _destVaultOne.debtValue(num0));

        // start attack
        _lmpVault.redeem(1, address(this), address(this));
        assertEq(num2 - 2, _lmpVault.totalDebt());
        assertEq(num2 - 2, _lmpVault.totalAssets());

        _lmpVault.redeem(1, address(this), address(this));
        assertEq(num2 - 4, _lmpVault.totalDebt());
        assertEq(num2 - 4, _lmpVault.totalAssets());

        _lmpVault.updateDebtReporting(_destinations);
    }

```

In this case I used 1,000,000 against 1 wei, therefore it's **within NAV change tolerance**. And it illustrates the root cause I'm trying to point out.

```
debt = totalDebt - prevNTotalDebt + afterNTotalDebt
```

totalDebt **could be less than** prevNTotalDebt.

prevNTotalDebt is currentDebt which is debt during rebalance scaled to reflect current supply.

```

uint256 currentDebt = (destInfo.currentDebt * originalShares) /
↳ Math.max(destInfo.ownedShares, 1);

```

However, a value that is rounded up is subtracted from totalDebt.

```

// https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audi
↳ t-2023-07-14/src/vault/libs/LMPDebt.sol#L295C9-L295C98
totalDebtBurn = cachedCurrentDebt.mulDiv(sharesToBurn, cachedDvShares,
↳ Math.Rounding.Up);

// https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audi
↳ t-2023-07-14/src/vault/LMPVault.sol#L489
info.debtDecrease += totalDebtBurn;

// https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audi
↳ t-2023-07-14/src/vault/LMPVault.sol#L518C13-L518C44
totalDebt -= info.debtDecrease;

```

So, the goal is to get rounding up against totalDebt that is less than the

rounding down in prevNTotalDebt.

The PoC achieves that by getting a double rounding up against totalDebt, which would result in just 1 rounding down in prevNTotalDebt. If you notice the attack section, each redemption of 1 share would burn 2 debts (instead of burning 1.000_001 debt because of rounding up).

P.S: Big Kudos to @Trumpero for taking the time to comment on invalid issues

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.

Trumpero

The POC looks good to me. Could you please review this issue, @codenutt?

codenutt

Yup I'd agree @Trumpero , looks valid. Thanks @bizzyvinci !

Evert0x

Planning to accept escalation and make issue valid.

Evert0x

@Trumpero do you think this deserves the high severity label?

Trumpero

I think it should be medium since this case just happens in some specific cases

Evert0x

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- bizzyvinci: accepted



Issue M-15: LMPVault: DoS when feeSink balance hits perWalletLimit

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/679>

Found by

Ch_301, n33k, warRoom, xiaoming90

The LMPVault token share has a per-wallet limit. LMPVault collects fees as share tokens to the feeSink address. `_collectFees` will revert if it mints shares that make the feeSink balance hit the `perWalletLimit`.

Vulnerability Detail

`_collectFees` mints shares to feeSink.

```
function _collectFees(uint256 idle, uint256 debt, uint256 totalSupply) internal {
    address sink = feeSink;
    ....
    if (fees > 0 && sink != address(0)) {
        // Calculated separate from other mints as normal share mint is round
        ↪ down
        shares = _convertToShares(fees, Math.Rounding.Up);
        _mint(sink, shares);
        emit Deposit(address(this), sink, fees, shares);
    }
    ....
}
```

`_mint` calls `_beforeTokenTransfer` internally to check if the target wallet exceeds `perWalletLimit`.

```
function _beforeTokenTransfer(address from, address to, uint256 amount) internal
    ↪ virtual override whenNotPaused {
    ....
    if (balanceOf(to) + amount > perWalletLimit) {
        revert OverWalletLimit(to);
    }
}
```

`_collectFees` function will revert if `balanceOf(feeSink) + fee shares > perWalletLimit`. `updateDebtReporting`, `rebalance` and `flashRebalance` call `_collectFees` internally so they will be unfunctional.



Impact

updateDebtReporting, rebalance and flashRebalance won't be working if feeSink balance hits perWalletLimit.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L823>

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L849-L851>

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L797>

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L703>

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/LMPVault.sol#L727>

Tool used

Manual Review

Recommendation

Allow feeSink to exceeds perWalletLimit.



Issue M-16: Incorrect amount given as input to `_handleRebalanceIn` when `flashRebalance` is called

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/701>

Found by

Aymen0909, ck

When `flashRebalance` is called, the wrong deposit amount is given to the `_handleRebalanceIn` function as the whole `tokenInBalanceAfter` amount is given as input instead of the delta value `tokenInBalanceAfter - tokenInBalanceBefore`, this will result in an incorrect rebalance operation and can potentially lead to a DOS due to the insufficient amount error.

Vulnerability Detail

The issue occurs in the `flashRebalance` function below :

```
function flashRebalance(
    DestinationInfo storage destInfoOut,
    DestinationInfo storage destInfoIn,
    IERC3156FlashBorrower receiver,
    IStrategy.RebalanceParams memory params,
    FlashRebalanceParams memory flashParams,
    bytes calldata data
) external returns (uint256 idle, uint256 debt) {
    ...

    // Handle increase (shares coming "In", getting underlying from the swapper
    // and trading for new shares)
    ↪ if (params.amountIn > 0) {
        IDestinationVault dvIn = IDestinationVault(params.destinationIn);

        // get "before" counts
        uint256 tokenInBalanceBefore =
    ↪ IERC20(params.tokenIn).balanceOf(address(this));

        // Give control back to the solver so they can make use of the "out"
    ↪ assets
        // and get our "in" asset
        bytes32 flashResult = receiver.onFlashLoan(msg.sender, params.tokenIn,
    ↪ params.amountIn, 0, data);

        // We assume the solver will send us the assets
```



```

        uint256 tokenInBalanceAfter =
↳ IERC20(params.tokenIn).balanceOf(address(this));

        // Make sure the call was successful and verify we have at least the
↳ assets we think
        // we were getting
        if (
            flashResult != keccak256("ERC3156FlashBorrower.onFlashLoan")
            || tokenInBalanceAfter < tokenInBalanceBefore + params.amountIn
        ) {
            revert Errors.FlashLoanFailed(params.tokenIn, params.amountIn);
        }

        if (params.tokenIn != address(flashParams.baseAsset)) {
            // @audit should be `tokenInBalanceAfter - tokenInBalanceBefore`
↳ given to `_handleRebalanceIn`
            (uint256 debtDecreaseIn, uint256 debtIncreaseIn) =
                _handleRebalanceIn(destInfoIn, dvIn, params.tokenIn,
↳ tokenInBalanceAfter);
            idleDebtChange.debtDecrease += debtDecreaseIn;
            idleDebtChange.debtIncrease += debtIncreaseIn;
        } else {
            idleDebtChange.idleIncrease += tokenInBalanceAfter -
↳ tokenInBalanceBefore;
        }
    }
    ...
}

```

As we can see from the code above, the function executes a flashloan in order to receive the tokenIn amount which should be the difference between tokenInBalanceAfter (balance of the contract after the flashloan) and tokenInBalanceBefore (balance of the contract before the flashloan): tokenInBalanceAfter - tokenInBalanceBefore.

But when calling the _handleRebalanceIn function the wrong deposit amount is given as input, as the total balance tokenInBalanceAfter is used instead of the received amount tokenInBalanceAfter - tokenInBalanceBefore.

Because the _handleRebalanceIn function is supposed to deposit the input amount to the destination vault, this error can result in sending a larger amount of funds to DV than what was intended or this error can cause a DOS of the flashRebalance function (due to the insufficient amount error when performing the transfer to DV), all of this will make the rebalance operation fail (or not done correctly) which can have a negative impact on the LMPVault.



Impact

See summary

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/vault/libs/LMPDebt.sol#L185-L215>

Tool used

Manual Review

Recommendation

Use the correct received tokenIn amount tokenInBalanceAfter - tokenInBalanceBefore as input to the _handleRebalanceIn function :

```
function flashRebalance(
    DestinationInfo storage destInfoOut,
    DestinationInfo storage destInfoIn,
    IERC3156FlashBorrower receiver,
    IStrategy.RebalanceParams memory params,
    FlashRebalanceParams memory flashParams,
    bytes calldata data
) external returns (uint256 idle, uint256 debt) {
    ...

    // Handle increase (shares coming "In", getting underlying from the swapper
    ↪ and trading for new shares)
    if (params.amountIn > 0) {
        IDestinationVault dvIn = IDestinationVault(params.destinationIn);

        // get "before" counts
        uint256 tokenInBalanceBefore =
    ↪ IERC20(params.tokenIn).balanceOf(address(this));

        // Give control back to the solver so they can make use of the "out"
    ↪ assets
        // and get our "in" asset
        bytes32 flashResult = receiver.onFlashLoan(msg.sender, params.tokenIn,
    ↪ params.amountIn, 0, data);

        // We assume the solver will send us the assets
        uint256 tokenInBalanceAfter =
    ↪ IERC20(params.tokenIn).balanceOf(address(this));
```



```

        // Make sure the call was successful and verify we have at least the
↪ assets we think
        // we were getting
        if (
            flashResult != keccak256("ERC3156FlashBorrower.onFlashLoan")
            || tokenInBalanceAfter < tokenInBalanceBefore + params.amountIn
        ) {
            revert Errors.FlashLoanFailed(params.tokenIn, params.amountIn);
        }

        if (params.tokenIn != address(flashParams.baseAsset)) {
            // @audit Use `tokenInBalanceAfter - tokenInBalanceBefore` as input
            (uint256 debtDecreaseIn, uint256 debtIncreaseIn) =
                _handleRebalanceIn(destInfoIn, dvIn, params.tokenIn,
↪ tokenInBalanceAfter - tokenInBalanceBefore);
            idleDebtChange.debtDecrease += debtDecreaseIn;
            idleDebtChange.debtIncrease += debtIncreaseIn;
        } else {
            idleDebtChange.idleIncrease += tokenInBalanceAfter -
↪ tokenInBalanceBefore;
        }
    }
    ...
}

```

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

Trumpero commented:



Issue M-17: OOG / unexpected reverts due to incorrect usage of staticcall.

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/822>

Found by

carrotsmugger, ctf_sec

OOG / unexpected reverts due to incorrect usage of staticcall.

Vulnerability Detail

The function `checkReentrancy` in `BalancerUtilities.sol` is used to check if the balancer contract has been re-entered or not. It does this by doing a `staticcall` on the pool contract and checking the return value. According to the solidity docs, if a `staticcall` encounters a state change, it burns up all gas and returns. The `checkReentrancy` tries to call `manageUserBalance` on the vault contract, and returns if it finds a state change.

The issue is that this burns up all the gas sent with the call. According to EIP150, a call gets allocated 63/64 bits of the gas, and the entire 63/64 parts of the gas is burnt up after the `staticcall`, since the `staticcall` will always encounter a storage change. This is also highlighted in the balancer monorepo, which has guidelines on how to check re-entrancy [here](#).

This can also be shown with a simple POC.

```
unction testAttack() public {
    mockRootPrice(WSTETH, 1_123_300_000_000_000_000); //wstETH
    mockRootPrice(CBETH, 1_034_300_000_000_000_000); //cbETH

    IBalancerMetaStablePool pool =
↪ IBalancerMetaStablePool(WSTETH_CBETH_POOL);

    address[] memory assets = new address[] (2);
    assets[0] = WSTETH;
    assets[1] = CBETH;
    uint256[] memory amounts = new uint256[] (2);
    amounts[0] = 10_000 ether;
    amounts[1] = 0;

    IBalancerVault.JoinPoolRequest memory joinRequest =
↪ IBalancerVault.JoinPoolRequest({
        assets: assets,
        maxAmountsIn: amounts, // maxAmountsIn,
```



```

        userData: abi.encode(
            IBalancerVault.JoinKind.EXACT_TOKENS_IN_FOR_BPT_OUT,
            amounts, //maxAmountsIn,
            0
        ),
        fromInternalBalance: false
    });

    IBalancerVault.SingleSwap memory swapRequest =
    ↪ IBalancerVault.SingleSwap({
        poolId:
    ↪ 0x9c6d47ff73e0f5e51be5fd53236e3f595c5793f2000200000000000000000042c,
        kind: IBalancerVault.SwapKind.GIVEN_IN,
        assetIn: WSTETH,
        assetOut: CBETH,
        amount: amounts[0],
        userData: abi.encode(
            IBalancerVault.JoinKind.EXACT_TOKENS_IN_FOR_BPT_OUT,
            amounts, //maxAmountsIn,
            0
        )
    });

    IBalancerVault.FundManagement memory funds =
    ↪ IBalancerVault.FundManagement({
        sender: address(this),
        fromInternalBalance: false,
        recipient: payable(address(this)),
        toInternalBalance: false
    });

    emit log_named_uint("Gas before price1", gasleft());
    uint256 price1 = oracle.getPriceInEth(WSTETH_CBETH_POOL);
    emit log_named_uint("price1", price1);
    emit log_named_uint("Gas after price1 ", gasleft());
}

```

The oracle is called to get a price. This oracle calls the `checkReentrancy` function and burns up the gas. The gas left is checked before and after this call.

The output shows this:

```

[PASS] testAttack() (gas: 9203730962297323943)
Logs:
Gas before price1: 9223372036854745204
price1: 1006294352158612428

```



```
Gas after price1 : 425625349158468958
```

This shows that 96% of the gas sent is burnt up in the oracle call.

Impact

This causes the contract to burn up 63/64 bits of gas in a single check. If there are lots of operations after this call, the call can revert due to running out of gas. This can lead to a DOS of the contract.

Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/libs/BalancerUtilities.sol#L19-L28>

Tool used

Foundry

Recommendation

According to the monorepo [here](#), the staticcall must be allocated a fixed amount of gas. Change the reentrancy check to the following.

```
(, bytes memory revertData) = address(vault).staticcall{ gas: 10_000 }(
    abi.encodeWithSelector(vault.manageUserBalance.selector, 0)
);
```

This ensures gas isn't burnt up without reason.

Discussion

JeffCX

Escalate

politely dispute that the severity is high because the transaction that meant to check the reentrancy burn too much gas and can revert and can block withdraw of fund or at least constantly burn all amount of gas and make user lose money

in a single transaction, the cost burnt can be minimal, but suppose the user send 10000 transaction, the gas burnt lose add up

sherlock-admin2



Escalate

politely dispute that the severity is high because the transaction that meant to check the reentrancy burn too much gas and can revert and can block withdraw of fund or at least constantly burn all amount of gas and make user lose money

in a single transaction, the cost burnt can be minimal, but suppose the user send 10000 transaction, the gas burnt loss add up

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.

sherlock-admin2

Escalate

This issue should be a Low.

The reason is that it will only consume all gas if the staticcall reverts, which means a re-entrancy is detected. For normal usage of the application, there will not be any re-entrancy. Thus, the normal users who use the protocol in the intended manner will not be affected by this issue.

It will only consume all the gas of attackers trying to carry out a re-entrancy attack against the protocol. In this case, the attacker will not get the gas refund when the re-entrancy detection reverts.

The loss of gas refunding for the attacker is not a valid issue in Sherlock since we are protecting the users, not the malicious users attempting to perform a re-entrancy attack or someone using the features in an unintended manner that triggers the re-entrancy revert. In addition, the loss of gas refund is not substantial enough to be considered a Medium since the chance of innocent users triggering a re-entrancy is close to zero in real life.

From the perspective of a smart contract, if an innocent external contract accidentally calls the Balancer protocol that passes the control to the contract and then calls Tokemak, which triggers a re-entrancy revert, such a contract is not operating correctly and should be fixed.

You've deleted an escalation for this issue.

JeffCX



With full respect to senior watson's comment

The reason is that it will only consume all gas if the staticcall reverts,
which means a re-entrancy is detected

I have to dispute the statement above,

Please review the duplicate issue #837 as well

the old balancer reentrancy check version does not cap the staticcall gas limit
but the new version add the 10000 gas cap

<https://github.com/balancer/balancer-v2-monorepo/blob/227683919a7031615c0bc7f144666cdf3883d212/pkg/pool-utils/contracts/lib/VaultReentrancyLib.sol#L43-L55>

and the balancer team already clearly state that the static call always revert even
the reentrancy is not detected

```
// However, use a static call so that it can be a view function (even though the
↳ function is non-view).
// This allows the library to be used more widely, as some functions that need
↳ to be protected might be
// view.
//
// This staticcall always reverts, but we need to make sure it doesn't fail due
↳ to a re-entrancy attack.
// Staticcalls consume all gas forwarded to them on a revert caused by storage
↳ modification.
// By default, almost the entire available gas is forwarded to the staticcall,
// causing the entire call to revert with an 'out of gas' error.
//
// We set the gas limit to 10k for the staticcall to
// avoid wasting gas when it reverts due to storage modification.
// `manageUserBalance` is a non-reentrant function in the Vault, so calling it
↳ invokes `_enterNonReentrant`
// in the `ReentrancyGuard` contract, reproduced here:
//
//     function _enterNonReentrant() private {
//         // If the Vault is actually being reentered, it will revert in the
↳ first line, at the `_require` that
//         // checks the reentrancy flag, with "BAL#400" (corresponding to
↳ Errors.REENTRANCY) in the revertData.
//         // The full revertData will be:
↳ `abi.encodeWithSignature("Error(string)", "BAL#400")`.
//         _require(_status != _ENTERED, Errors.REENTRANCY);
//     }
```



```
//          // If the Vault is not being reentered, the check above will pass: but
↳ it will *still* revert,
//          // because the next line attempts to modify storage during a
↳ staticcall. However, this type of
//          // failure results in empty revertData.
//          _status = _ENTERED;
//      }
```

use not capping the gas limit of static call means the user are constantly waste too much gas (let us say for a single call the user waste and lose 0.01 ETH at gas), after 20000 transaction the loss is cumulatively 200 ETH and has no upper limit of loss

because the reason above

the balancer push a PR fix for this issue specifically, can see the PR

<https://github.com/balancer/balancer-v2-monorepo/pull/2467>

gas is either wasted or transaction revert and block withdraw, which is both really bad for user in long term, so the severity should be high instead of medium

```
// If the Vault is not being reentered, the check above will pass: but it will
↳ *still* revert,
// because the next line attempts to modify storage during a staticcall.
↳ However, this type of
// failure results in empty revertData.
```

carrotsmuggler

The comment by the LSW is wrong. The POC clearly shows 90% of gas is consumed even when no re-entrancy is detected, i.e. for normal usage of the protocol.

When there is reentrancy, the entire transaction reverts. If there is no reentrancy, the static call still reverts due to a state change. There is no reentrancy for the situation given in the POC.

The `manageUserBalance` call always does a state change. When a state change is encountered during a static call, the entire gas is burnt up and the execution reverts. This happens irrespective of reentrancy conditions.

xiaoming9090

Thanks @JeffCX and @carrotsmuggler for your explanation. You are correct that the issue will happen irrespective of reentrancy conditions. The `manageUserBalance` function will trigger the `_enterNonReentrant` modifier. If there is no re-entrancy, this line of code will result in a state change that consume all the gas pass to it. I have deleted the escalation.



Trumpero

Hello JeffCX, I believe that the loss of gas might not qualify as a valid high. According to the guidelines in Sherlock's [documentation](#), the OOG matter will be deemed a medium severity issue. It could be considered high only in cases where it results in a complete blockage of all user funds indefinitely.

JeffCX

Sir, I agree the OOG does not block user withdraw forever

but because the static call always revert and waste 63/64 gas when withdraw, the remaining 1 / 64 gas has to be enough to complete the transaction.

this means user are force to overpaying 100x more gas to complete the withdraw from balancer vault

we can make a analogy:

the protocol open a bank, user put 10K fund into the bank, and the user should only pays for 1 USD transaction fee when withdraw the fund,

but the bank said, sorry because of a bug, everyone has to pay 100 USD to withdraw the fund, and this 100x cost applies every user in every withdrawal transaction, then the result is the withdraw is not really usable and cause consistent loss of fund

this 100x gas applies to all user in every withdrawal transaction that use the balancer vault and the loss of fund from gas has no upper bound, so I think a high severity is still justified.

Trumpero

Hello, @JeffCX,

The Out Of Gas (OOG) situation renders users unable to call a method of the contracts due to insufficient gas. On the other hand, your issue poses a risk where users:

- Couldn't call a method due to insufficient gas
- Users can pay more fees to trigger the function

From what I observe, the impact of your issue appears to be a subset of the impact caused by an OOG. Therefore, if the OOG is considered medium, your issue should be equal to or less than medium in severity. I would appreciate it if you could share your opinion on this.

JeffCX

Yeap, it is a subset of impact caused by OOG,



so Users can pay more fees to trigger the function, but as I already shown, every user needs to pay 100x gas more in every withdrawal transaction for balancer vault, so the lose of fund as gas is cumulatively high :)

Trumpero

I believe that in the scenario of an OOG/DOS, it represents the worst-case scenario for your issue. This is because when an OOG/DOS happens, users will pay a gas fee without any results, resulting in a loss of their gas. Hence, the impact of an OOG can be rephrased as follows: "users pay 100x gas fee but can't use the function". On the other hand, your issue states that "users pay 100x gas fee but sometime it fails". Is that correct?

JeffCX

user can pay 100x gas and use the function as long as the remaining 1/64 gas can complete the executions.

in my original report

<https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/837>

the impact I summarized is:

the function may waste too much gas and result in an out of gas error
and can block function call such as withdraw

emm as for

users pay 100x gas fee but sometime it fails

as long as user pay a lot of gas (which they should not, transaction can be processed), and if they do not pay that amount of gas, transaction fails

and sir, just a friendly reminder

my original escalation is the severity should be high instead of medium based on the impact

Trumpero

Certainly, I comprehend that you are aiming to elevate the severity level of this issue. However, my stance remains that this issue should be classified as medium due to the following rationale:

Let's consider a situation where Alice intends to initiate 2 methods. Method A results in a denial-of-service (DOS) due to an out-of-gas (OOG) scenario, while method B aligns with your described issue.

1. Alice expends 1 ETH as a gas fee but is unable to execute method A. Even when she attempts to allocate 10 ETH for the gas fee, she still cannot trigger method A.



2. Simultaneously, Alice expends 1 ETH as a gas fee but encounters an inability to execute method B. However, when she allocates 10 ETH for the gas fee, she successfully triggers method B.

Consequently, we observe that method A costs Alice 11 ETH as a gas fee without any return, whereas method B costs Alice the same 11 ETH, yet she gains the opportunity to execute it. Hence, we can infer that method A is more susceptible than method B.

JeffCX

Sir, I don't think the method A and method B example applies in the codebase and in this issue

there is only one method for user to withdraw share from the vault

I can add more detail to explain how this impact withdraw using top-down approach

User can withdraw by calling withdraw in LMPVault.sol and triggers [_withdraw](#)

the [_withdraw](#) calls the method [_calcUserWithdrawSharesToBurn](#)

this calls [LMPDebt._calcUserWithdrawSharesToBurn](#)

we need to know the [debt value](#) by calling destVault.debtValue

this calls this [line of code](#)

this calls the [oracle code](#)

```
uint256 price = _systemRegistry.rootPriceOracle().getPriceInEth(_underlying);
```

then if the dest vault is the balancer vault, balancer reentrancy check is triggered to waste 63 / 64 waste in [oracle code](#)

so there is no function A and function B call

as long as user can withdraw and wants to withdraw share from balancer vault, 100x gas overpayment is required

JeffCX

I think we can treat this issue same as "transaction missing slippage protection"

missing slippage protection is consider a high severity finding, but user may not lose million in one single transaction, the loss depends on user's trading amount

the loss amount for individual transaction can be small but there are be a lot of user getting frontrunning and the missing slippage cause consistent leak of value

all the above character applies to this finding as well

can refer back to my first analogy



the protocol open a bank, user put 10K fund into the bank, and the user should only pays for 1 USD transaction fee when withdraw the fund,

but the bank said, sorry because of a bug, everyone has to pay 100 USD to withdraw the fund, and this 100x cost applies every user in every withdrawal transaction, then the result is the withdraw is not really usable and cause consistent loss of fund

this 100x gas applies to all user in every withdrawal transaction that use the balancer vault and the loss of fund from gas has no upper bound, so I think a high severity is still justified.

Evert0x

I think we can treat this issue same as "transaction missing slippage protection"

You are referring to the gas usage here? Putting a limit on the gas is not a task for the protocol, this is a task for the wallet someone is using.

As the escalation comment states

in a single transaction, the cost burnt can be minimal

Impact is not significant enough for a high severity.

Current opinion is to reject escalation and keep issue medium severity.

JeffCX

Putting a limit on the gas is not a task for the protocol

sir, please read the report again, the flawed logic in the code charge user 100x gas in every transaction in every withdrawal

in a single transaction, the cost burnt can be minimal

the most relevant comments is <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/822#issuecomment-1765550141>

and <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/822#issuecomment-1769126560>

idk how do state it more clearly, emm if you put money in the bank, you expect to pay 1 USD for withdrawal transaction fee, but every time you have to pay 100 USD withdrawal fee because of the bug

this cause loss of fund for every user in every transaction for not only you but every user...

Evert0x



@JeffCX what are the exact numbers on the withdrawal costs? E.g. if I want to withdraw \$10k, how much gas can I expect to pay? If this is a significant amount I can see the argument for

How to identify a high issue: Definite loss of funds without limiting external conditions.

But it's not clear how much this will be assuming current mainnet conditions.

JeffCX

I write a simple POC

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "forge-std/console.sol";

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MockERC20 is ERC20 {
    constructor()ERC20("MyToken", "MTK")
    {}

    function mint(address to, uint256 amount) public {
        _mint(to, amount);
    }
}

interface ICheckReentrancy {
    function checkReentrancy() external;
}

contract ReentrancyCheck {

    uint256 state = 10;

    function checkReentrancy() external {
        address(this).staticcall(abi.encodeWithSignature("hihi()"));
    }

    function hihi() public {
        state = 11;
    }
}
```



```

contract Vault {

    address balancerAddr;
    bool checkRentrancy;

    constructor(bool _checkRentrancy, address _balancerAddr) {
        checkRentrancy = _checkRentrancy;
        balancerAddr = _balancerAddr;
    }

    function toggleCheck(bool _state) public {
        checkRentrancy = _state;
    }

    function withdraw(address token, uint256 amount) public {

        if(checkRentrancy) {
            ICheckRentrancy(balancerAddr).checkRentrancy();
        }

        IERC20(token).transfer(msg.sender, amount);

    }

}

contract CounterTest is Test {

    using stdStorage for StdStorage;
    StdStorage stdlib;

    MockERC20 token;
    Vault vault;
    RentrancyCheck reentrancyCheck;

    address user = vm.addr(5201314);

    function setUp() public {

        token = new MockERC20();
        reentrancyCheck = new RentrancyCheck();
        vault = new Vault(false, address(reentrancyCheck));
        token.mint(address(vault), 100000000 ether);

        vm.deal(user, 100 ether);
    }

}

```




```

        // vault.toggleCheck(true);
    }

    function testPOC() public {

        uint256 gas = gasleft();
        uint256 amount = 100 ether;
        vault.withdraw(address(token), amount);
        console.log(gas - gasleft());

    }
}

```

the call is

```

if check reentrancy flag is true

user withdraw ->
check reentrancy staticall revert and consume most of the gas
-> withdraw completed

```

or

```

if check reentrancy flag is false

user withdraw ->
-> withdraw completed

```

note first we do not check the reentrancy

```

// vault.toggleCheck(true);

```

we run

```

forge test -vvv --match-test "testPOC" --fork-url "https://eth.llamarpc.com"
↳ --gas-limit 10000000

```

the gas cost is 42335

```

Running 1 test for test/Counter.t.sol:CounterTest
[PASS] testPOC() (gas: 45438)

```



```
Logs:  
42335
```

then we uncomment the `vault.toggleCheck(true)` and check the reentrancy that revert in staticcall

```
vault.toggleCheck(true);
```

we run the same test again, this is the output, as we can see the gas cost surge

```
Running 1 test for test/Counter.t.sol:CounterTest  
[PASS] testPOC() (gas: 9554791)  
Logs:  
9551688
```

then we can use this python script to estimate how much gas is overpaid as lost of fund

```
regular = 42313  
  
overpaid = 9551666  
  
# gas price: 45 gwei -> 0.000000045  
  
cost = 0.000000045 * (overpaid - regular);  
  
print(cost)
```

the cost is

```
0.427920885 ETH
```

in a single withdraw, assume user lost 0.427 ETH,

if 500 user withdraw 20 times each and the total number of transaction is 10000

the lose on gas is $10000 * 0.427$ ETH

JeffCX

note that the more gas limit user set, the more fund user lose in gas

but we are interested in what the lowest amount of gas limit user that user can set the pay for withdrawal transaction

I did some fuzzing

that number is 1800000 unit of gas



the command to run the test is

```
forge test -vvv --match-test "testPOC" --fork-url "https://eth.llamarpc.com"
↳ --gas-limit 1800000
```

setting gas limit lower than 1800000 unit of gas is likely to revert in out of gas
under this setting, the overpaid transaction cost is 1730089

```
Running 1 test for test/Counter.t.sol:CounterTest
[PASS] testPOC() (gas: 1733192)
Logs:
    1730089
```

in other words,

in each withdrawal for every user, user can lose 0.073 ETH, (1730089 unit of gas * 45 gwei -> 0.000000045 ETH)

assume there are 1000 user, each withdraw 10 times, they make 1000 * 10 = 100_00 transaction

so the total lost is 100_00 * 0.07 = 700 ETH

in reality the gas is more than that because user may use more than 1800000 unit of gas to finalize the withdrawal transaction

Evert0x

@JeffCX thanks for putting in the effort to make this estimation.

But as far as I can see, your estimation doesn't use the actual contracts in scope. But maybe that's irrelevant to make your point.

This seems like the key sentence

in each withdrawal for every user, user can lose 0.073 ETH,

This is an extra 100–150 dollars per withdrawal action.

This is not a very significant amount in my opinion. I assume an optimized withdrawal transaction will cost between 20–50. So the difference is not as big.

JeffCX

Sir, I don't think the method A and method B example applies in the codebase and in this issue

there is only one method for user to withdraw share from the vault

I can add more detail to explain how this impact withdraw using top-down approach



User can withdraw by calling `withdraw` in `LMPVault.sol` and triggers `_withdraw`

the `_withdraw` calls the method `_calcUserWithdrawSharesToBurn`

this calls `LMPDebt._calcUserWithdrawSharesToBurn`

we need to know the debt value by calling `destVault.debtValue`

this calls this line of code

this calls the oracle code

```
uint256 price =  
↳ _systemRegistry.rootPriceOracle().getPriceInEth(_underlying);
```

then if the dest vault is the balancer vault, balancer reentrancy check is triggered to waste 63 / 64 waste in oracle code

so there is no function A and function B call

as long as user can withdraw and wants to withdraw share from balancer vault, 100x gas overpayment is required

the POC is a simplified flow of this

it is ok to disagree sir:)

Evert0x

Result: Medium Has Duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- JEFFCX: rejected



Issue M-18: Slashing during LSTCalculatorBase.sol deployment can show bad apr for months

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/824>

Found by

carrotsmugger, saidam017, xiaoming90

Slashing during LSTCalculatorBase.sol deployment can show bad apr for months

Vulnerability Detail

The contract LSTCalculatorBase.sol has some functions to calculate the rough APR expected from a liquid staking token. The contract is first deployed, and the first snapshot is taken after APR_FILTER_INIT_INTERVAL_IN_SEC, which is 9 days. It then calculates the APR between the deployment and this first snapshot, and uses that to initialize the APR value. It uses the function calculateAnnualizedChangeMinZero to do this calculation.

The issue is that the function calculateAnnualizedChangeMinZero has a floor of 0. So if the backing of the LST decreases over that 9 days due to a slashing event in that interval, this function will return 0, and the initial APR and baseApr will be set to 0.

The calculator is designed to update the APR at regular intervals of 3 days. However, the new apr is given a weight of 10% and the older apr is given a weight of 90% as seen below.

```
return ((priorValue * (1e18 - alpha)) + (currentValue * alpha)) / 1e18;
```

And alpha is hardcoded to 0.1. So if the initial APR starts at 0 due to a slashing event in the initial 9 day period, a large number of updates will be required to bring the APR up to the correct value.

Assuming the correct APR of 6%, and an initial APR of 0%, we can calculate that it takes upto 28 updates to reflect close the correct APR. This translates to 84 days. So the wrong APR can be shown for upto 3 months. The protocol uses these APR values to justify the allocation to the various protocols. Thus a wrong APR for months would mean the protocol would sub optimally allocate funds for months, losing potential yield.

Impact

The protocol can underperform for months due to slashing events messing up APR calculations close to deployment date.



Code Snippet

<https://github.com/sherlock-audit/2023-06-tokemak/blob/main/v2-core-audit-2023-07-14/src/stats/calculators/base/LSTCalculatorBase.sol#L108-L110>

Tool used

Manual Review

Recommendation

It is recommended to initialize the APR with a specified value, rather than calculate it over the initial 9 days. 9 day window is not good enough to get an accurate APR, and can be easily manipulated by a slashing event.

Discussion

codenutt

This behavior is acceptable. If we happen to see a slash > 12 bps over the initial 9 days, yes, we set it to 0. It increases the ramp time for that LST so pools with that LST will be set aside for a while until some APR (incentive, etc) comes up. For larger slashes that are more material (> 25bps), we have a 90 day penalty anyway.



Issue M-19: curve admin can drain pool via reentrancy (equal to execute emergency withdraw and rug tokenmak fund by third party)

Source: <https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/862>

Found by

ctf_sec

curve admin can drain pool via reentrancy (equal to execute emergency withdraw and rug tokenmak fund)

Vulnerability Detail

A few curve liquidity is pool is well in-scope:

Curve Pools

```
Curve stETH/ETH: 0x06325440D014e39736583c165C2963BA99fAf14E
Curve stETH/ETH ng: 0x21E27a5E5513D6e65C4f830167390997aA84843a
Curve stETH/ETH concentrated: 0x828b154032950C8ff7CF8085D841723Db2696056
Curve stETH/frxETH: 0x4d9f9D15101EEC665F77210cB999639f760F831E
Curve rETH/ETH: 0x6c38cE8984a890F5e46e6dF6117C26b3F1EcFC9C
Curve rETH/wstETH: 0x447Ddd4960d9fdBF6af9a790560d0AF76795CB08
Curve rETH/frxETH: 0xbA6c373992AD8ec1f7520E5878E5540Eb36DeBf1
Curve cbETH/ETH: 0x5b6C539b224014A09B3388e51CaAA8e354c959C8
Curve cbETH/frxETH: 0x548E063CE6F3BaC31457E4f5b4e2345286274257
Curve frxETH/ETH: 0xf43211935C781D5ca1a41d2041F397B8A7366C7A
Curve swETH/frxETH: 0xe49AdDc2D1A131c6b8145F0EBa1C946B7198e0BA
```

one of the pool is 0x21E27a5E5513D6e65C4f830167390997aA84843a

<https://etherscan.io/address/0x21E27a5E5513D6e65C4f830167390997aA84843a#code#L1121>

Admin of curve pools can easily drain curve pools via reentrancy or via the withdraw_admin_fees function.

```
@external
def withdraw_admin_fees():
    receiver: address = Factory(self.factory).get_fee_receiver(self)

    amount: uint256 = self.admin_balances[0]
    if amount != 0:
        raw_call(receiver, b"", value=amount)
```



```
amount = self.admin_balances[1]
if amount != 0:
    assert ERC20(self.coins[1]).transfer(receiver, amount,
↪ default_return_value=True)

self.admin_balances = empty(uint256[N_COINS])
```

if admin of the curve can set a receiver to a malicious smart contract and reenter withdraw_admin_fees a 1000 times to drain the pool even the admin_balances is small

the line of code

trigger the reentrancy

This is a problem because as stated by the tokemak team:

In case of external protocol integrations, are the risks of external contracts pausing or executing an emergency withdrawal acceptable? If not, Watsons will submit issues related to these situations that can harm your protocol's functionality.

Pausing or emergency withdrawals are not acceptable for Tokemak.

As you can see above, pausing or emergency withdrawals are not acceptable, and this is possible for curve pools so this is a valid issue according to the protocol and according to the read me

Impact

curve admins can drain pool via reentrancy

Code Snippet

<https://etherscan.io/address/0x21E27a5E5513D6e65C4f830167390997aA84843a#code#L1121>

Tool used

Manual Review

Recommendation

N/A



Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

Trumpero commented:

invalid, as the submission stated: "Admin of curve pools can easily drain curve pools via reentrancy", so no vulnerability for tokemak here

JeffCX

Escalate

as the protocol docs mentioned

<https://audits.sherlock.xyz/contests/101>

In case of external protocol integrations, are the risks of external contracts pausing or executing an emergency withdrawal acceptable? If not, Watsons will submit issues related to these situations that can harm your protocol's functionality.

Pausing or emergency withdrawals are not acceptable for Tokemak.

in the issue got exploit in this report, user from tokenmak lose fund as well

sherlock-admin2

Escalate

as the protocol docs mentioned

<https://audits.sherlock.xyz/contests/101>

In case of external protocol integrations, are the risks of external contracts pausing or executing an emergency withdrawal acceptable? If not, Watsons will submit issues related to these situations that can harm your protocol's functionality.

Pausing or emergency withdrawals are not acceptable for Tokemak.

in the issue got exploit in this report, user from tokenmak lose fund as well

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.



Trumpero

Hi @JeffCX, based on this comment of sponsors in the contest channel, I think this issue should be marked as low/invalid: <https://discord.com/channels/812037309376495636/1130514263522410506/1143588977962647582>

JeffCX

Sponsor said emergency withdrawal or pause is an unacceptable risk.

Did you read it as "acceptable" sir?

JeffCX

Some discussion is happening

<https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/899>

but this is a separate external integration risk than the balancer one that can impact tokemak user :) and don't think this is a known issue

Trumpero

Hello @JeffCX,

Upon further consideration of this matter, I find it to be valid. The potential for the curve admin to exploit the reentrancy-attack and drain the curve pool could have a direct impact on the Tokemak protocol.

I suggest that you review this issue as well, @codenutt.

JeffCX

Hello @JeffCX,

Upon further consideration of this matter, I find it to be valid. The potential for the curve admin to exploit the reentrancy-attack and drain the curve pool could have a direct impact on the Tokemak protocol.

I suggest that you review this issue as well, @codenutt.

Thank you very much!

codenutt

Thanks @Trumpero / @JeffCX! Just to confirm, this is an issue with some Curve pools just in general, correct? Not necessarily with a particular interaction we have with them.

Trumpero

Yes, you are right

Evert0x

Planning to accept escalation and label issue as valid



JeffCX

thanks

Evert0x

@Trumpero would you agree with high severity?

Trumpero

No I think it should be medium since it assume the curve admin become malicious

JeffCX

Agree with medium,
<https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/570> similar finding about external admin turn into malicious risk is marked as medium as well

Evert0x

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- JEFFCX: accepted

