SHERLOCK

Security Review For Cork Protocol

Public Best Efforts Contest Prepared For: **Cork Protocol**
Lead Security Expert: **0x73696d616f**
Date Audited: **August 29 - September 10, 2024**

Introduction

Cork is building a protocol to price, hedge and trade risk. In this contest our core contracts will be audited in advance of our mainnet launch.

Scope

Repository: Cork-Technology/Depeg-swap

Branch: refactor/netspec-comments

Audited Commit: d4fcdd524deb5d07a7ccfd6eb49ba5157ed42642

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

High	Medium
11	7

Security Experts Who Found Valid Issues

0x73696d616f
dimulski
sakshamguruji
vinica_boy
0xNirix
KupiaSec
oxelmiguel
Smacaud
nikhil840096
Pheonix
Ace-30
MadSisyphus
Kirkeelee
korok
hunter_w3b
Matrox
steadyman
Abhan1041
4gontuk
alphacipher
ivanonchain
0x6a70
4b
MohammedRizwan
0xjoi
StraawHaat
0xnbvc
boringslav
Trooper
ravikiran.web3

ydlee
yovchev_yoan
Pro_King

tinnohofficial
tmotfl
mladenov

durov
404Notfound
octeezy

Issue H-1: Lack of slippage protection leads to loss of protocol funds

Source: https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/66

This issue was not included in the Fix Review and has not been fixed.

Found by

0x6a70, 0x73696d616f, 0xjoi, 0xnbvc, 4b, 4gontuk, Abhan1041, MohammedRizwan, Pheonix, StraawHaat, alphacipher, boringslav, ivanonchain, sakshamguruji, vinica_boy

Summary

There is no slippage protection while removing liquidity and swap tokens from AMM.

Vulnerability Detail

There are 2 intances where slippage protection is missing which are as below:

1. When LV token holder redeem before expiry `vaultLib::redeemEarly` function is called in which `_liquidateLpPartial` function and in that `_redeemCtDsAndSellExcessCt` is called. In `_redeemCtDsAndSellExcessCt` function CT tokens are swapped for RA tokens in AMM as below:

```
...
ra += ammRouter.swapExactTokensForTokens(ctSellAmount, 0, path, address(this),
↪  block.timestamp)[1];
...
```

As stated above, `swapExactTokensForTokens` function's 2nd parameter is 0 which shows that there is no slippage protection for this swap and also deadline is block.timestamp.

2. In `vaultLib::_liquidateLpPartial` function `__liquidateUnchecked` is called in which liquidity is removed from AMM of RA-CT token pair by burning LP tokens of protocol as below:

```
...
(raReceived, ctReceived) =
            ammRouter.removeLiquidity(raAddress, ctAddress, lp, 0, 0,
↪  address(this), block.timestamp);
...
```

As stated above, `removeLiquidity` function's 4th & 5th parameter is 0 which shows that there is no slippage protection for this swap and also deadline is block.timestamp.

In such cases, an attacker can frontrun the transaction by seeing it in the mempool and manipulate the price such that protocol transaction have to bear heavy slippage which will leads to loss of protocol funds.

Also, there is block.timestamp as deadline so malicious node can prevent transaction to execute temporary and execute the transaction when there is high slippage which will also leads to loss of protocol funds.

Impact

Loss of protocol funds which will reduce the yield of users.

Code Snippet

https://github.com/sherlock-audit/2024-08-cork-protocol/blob/db23bf67e45781b00ee6de5f6f23e621af16bd7e/Depeg-swap/contracts/libraries/VaultLib.sol#L282

https://github.com/sherlock-audit/2024-08-cork-protocol/blob/db23bf67e45781b00ee6de5f6f23e621af16bd7e/Depeg-swap/contracts/libraries/VaultLib.sol#L345

Tool Used

Manual Review

Recommendation

Protocol should implement slippage protection and set deadline while removing liquidity and also swap from AMM.

Discussion

ziankork

Yes this is a valid issue, we've already fixed this prior to our trading competition except for the `deadline` vulnerability.

v-kirilov

#75 is a duplicate of this one!

Issue H-2: FlashSwapRouter::emptyReserve() and FlashSwapROuter::emptyReservePartial() functions return incorrect values

Source: https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/68

Found by

0x73696d616f, 0xNirix, KupiaSec, dimulski, nikhil840096, sakshamguruji, vinica_boy

Summary

The protocol deposits RA and CT tokens to an AMM pair, from fees or when users call the depositLv() function. The CT and DS tokens issued by the protocol have an expiration, after the first DS and CT tokens for a pair have been issued and expired, each next time the protocol tries to issue new DS and CT tokens for an existing pair of RA and PA tokens via calling the issueNewDs() function, the VaultLib::onNewIssuance() function will be called. The VaultLib::onNewIssuance() function will then call the VaultLib::_liquidatedLp() function, which internally calls the FlashSwapRouter::emptyReserve() function which will empty the whole reserve, and then return 0.

```
function emptyReserve(ReserveState storage self, uint256 dsId, address to) internal
↪   returns (uint256 reserve) {
    reserve = emptyReservePartial(self, dsId, self.ds[dsId].reserve, to);
}

function emptyReservePartial(ReserveState storage self, uint256 dsId, uint256
↪   amount, address to)
    internal
    returns (uint256 reserve)
{
    self.ds[dsId].ds.transfer(to, amount);

    self.ds[dsId].reserve -= amount;
    reserve = self.ds[dsId].reserve;
}
```

When we go back to the VaultLib::_liquidatedLp() function

```
function _liquidatedLp(
    State storage self,
    uint256 dsId,
    IUniswapV2Router02 ammRouter,
    IDsFlashSwapCore flashSwapRouter
```

```
) internal {
    ...
     // the following things should happen here(taken directly from the whitepaper)
↪   :
    // 1. The AMM LP is redeemed to receive CT + RA
    // 2. Any excess DS in the LV is paired with CT to redeem RA
    // 3. The excess CT is used to claim RA + PA in the PSM
    // 4. End state: Only RA + redeemed PA remains
    uint256 reservedDs = flashSwapRouter.emptyReserve(self.info.toId(), dsId);

    uint256 redeemAmount = reservedDs >= ctAmm ? ctAmm : reservedDs;
    PsmLibrary.lvRedeemRaWithCtDs(self, redeemAmount, dsId);

    // if the reserved DS is more than the CT that's available from liquidating the
↪   AMM LP
    // then there's no CT we can use to effectively redeem RA + PA from the PSM
    uint256 ctAttributedToPa = reservedDs >= ctAmm ? 0 : ctAmm - reservedDs;

    uint256 psmPa;
    uint256 psmRa;

    if (ctAttributedToPa != 0) {
        (psmPa, psmRa) = PsmLibrary.lvRedeemRaPaWithCt(self, ctAttributedToPa,
↪   dsId);
    }

    psmRa += redeemAmount;

    self.vault.pool.reserve(self.vault.lv.totalIssued(), raAmm + psmRa, psmPa);
}
```

As can be seen from the 2 comment in the function any excess CT and DS tokens should be paired and redeemed for RA, however since the FlashSwapRouter::emptyReserve() function will always return 0, so the PsmLib::lvRedeemRaWithCtDs() function will always redeem 0 RA tokens and not burn the CT and DS tokens. As we can see from the above code snippet we will go directly to PsmLib::lvRedeemRaPaWithCt() function, which will try to redeem RA + PA tokens, with all of the CT tokens that were returned from the UniV2 pair when the LP tokens of the protocol were liquidated.

The second case where a problem occurs is when a user tries to redeem his LV tokens by calling the redeemEarlyLv() function which internally calls the VaultLib::redeemEarly() function and after a couple of other internal calls the VaultLib::_redeemCtDsAndSellExcessCt() function is called where the FlashSwapRouter::emptyReservePartial() function is called:

```
function _redeemCtDsAndSellExcessCt(
    State storage self,
    uint256 dsId,
    IUniswapV2Router02 ammRouter,
```

```
    IDsFlashSwapCore flashSwapRouter,
    uint256 ammCtBalance
) internal returns (uint256 ra) {
    uint256 reservedDs = flashSwapRouter.getLvReserve(self.info.toId(), dsId);

    uint256 redeemAmount = reservedDs >= ammCtBalance ? ammCtBalance : reservedDs;

    reservedDs = flashSwapRouter.emptyReservePartial(self.info.toId(), dsId,
↪   redeemAmount);

    ra += redeemAmount;
    PsmLibrary.lvRedeemRaWithCtDs(self, redeemAmount, dsId);

    uint256 ctSellAmount = reservedDs >= ammCtBalance ? 0 : ammCtBalance -
↪   reservedDs;

    DepegSwap storage ds = self.ds[dsId];
    address[] memory path = new address[](2);
    path[0] = ds.ct;
    path[1] = self.info.pair1;

    ERC20(ds.ct).approve(address(ammRouter), ctSellAmount);

    if (ctSellAmount != 0) {
        // 100% tolerance, to ensure this not fail
        ra += ammRouter.swapExactTokensForTokens(ctSellAmount, 0, path,
↪   address(this), block.timestamp)[1];
    }
}
```

When the last LV tokens are being redeemed the **reservedDs** will be equal or very close to **ammCtBalance**, and when the FlashSwapRouter::emptyReservePartial() function is called, it will return the DS reserve after the redeemAmount has been subtracted, which will be either 0, or much less than **redeemAmount**. For this example consider it is 0. When the **ctSellAmount** is calculated it will be much bigger than the actual reserves of CT token in the contract, and when the function tries to transfer the CT tokens to the AMM in order to swap them for RA tokens, the call will revert, and the user redeeming his LV token won't be able to redeem it and receive RA tokens back, thus locking funds in the contract.

Root Cause

The root cause is that the FlashSwapRouter::emptyReserve() and FlashSwapROuter::emptyReservePartial() functions returns the reserve that is left after the redeemAmount has been subtracted.

Internal pre-conditions

1. Users mint LV tokens via the depositLv() function
2. There are a couple of LV tokens that haven't been redeemed yet, and a user decides to redeem them by calling the VaultLib::redeemEarly() function

External pre-conditions

No response

Attack Path

No response

Impact

When it comes to FlashSwapRouter::emptyReserve(), instead of the excess DS in the LV being paired with CT to redeem RA, all of the CT returned from the liquidation of LP will be used to claim RA + PA in the PSM, this is contrary of what is expected from the function according to the docs, and the comments, and may result in VaultLib::_liquidatedLp() function claiming much more PA tokens than it should, and distributing them to LV holders. In the case of FlashSwapROuter::emptyReservePartial(), the last users to withdraw won't be able to do so. The last user that tries to redeem his LV tokens won't be able to do so, and he won't receive his RA tokens back, locking the RA tokens in the contract.

PoC

Gist

After following the steps in the above mentioned gist add the following test to the `AuditorTests.t.sol` contract:

```
function test_IncorrectEmptyReserveReturnedValue() public {
    vm.startPrank(alice);
    WETH.mint(alice, 10e18);
    WETH.approve(address(moduleCore), type(uint256).max);
    moduleCore.depositLv(id, 10e18);
    Asset(lvAddress).approve(address(moduleCore), type(uint256).max);
    vm.expectRevert(bytes("TransferHelper::transferFrom: transferFrom failed"));
    moduleCore.redeemEarlyLv(id, alice, 10e18);
    vm.stopPrank();
}
```

To run the test use: `forgetest-vvv--mttest_IncorrectEmptyReserveReturnedValue`

Mitigation

A lot of things have to be considered when fixing this problems, simply returning the amount that was redeemed may introduce other problems. Returning the amount that was redeemed seems to be okay when it comes to the FlashSwapRouter::emptyReserve() function.

Discussion

ziankork

this is a valid issue, it should return the how much amount is emptied, we did fix this prior to our trading competition and will provide links later

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Cork-Technology/Depeg-swap/pull/59

Issue H-3: LV token holders receive proportional fees, when they shouldn't

Source: https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/106

The protocol has acknowledged this issue.

Found by

dimulski

Summary

The Corc protocol claims that the LV token holders should accrue fees that the protocol generates, from the docs: *In our design we envision several mechanisms through which the Liquidity Vault will generate revenues. Some of these are from protocol fees that will flow to the Liquidity Vault and accrue to Liquidity Vault tokenholders.* However, the fee accrual is incorrect, a user can deposit just before fees are about to be distributed to LV holders, and still receive the same amount of fees as a user who deposited tokens in the beginning. There is one LV token per RA:PA pair, however the CT and DS tokens expire and there may be several CT and DS tokens issued by the protocol. Users can mint an LV token via the Vault::depositLv() function, by depositing the corresponding RA asset. The first issue is that if User A mints a LV token, and request to redeem it, then some fees from swapping in the AMM pair for the CT and RA token are generated, or the protocol collects fees from users utilizing its functionality, then a User B mints a LV token, and request to redeem it, both users will accrues the same amount of fees generated by the protocol, when this shouldn't be the case. As the first users has had his request for redeem for a much longer period, risking the price of the RA token dropping significantly, while User B may just call the Vault::depositLv() function the last block before the CT and DS tokens expire, and request a redeem immediately. Then in the next block when the CT and DS tokens have already expired he can claim the same amount of fees as User A, this is demonstrated in the first POC. To better illustrate the second problem consider the following example

- User A and User B minted LV tokens while the first issued CT and DS tokens were still not expired
- The UniV2 pair generated some fees
- The protocol collects fees, from users utilizing its functionality
- The issued CT and DS tokens expired
- The protocol issued new CT and DS tokens
- User C mints some LV tokens
- The UniV2 pair generates some more fees,

The fees will be split equally between all the users (of course taking into account the amount of LV tokens they hold).

Root Cause

The protocol doesn't implement any mechanism to track when users minted LV tokens, or when they requested a redemption of their LV tokens, or what rewards were accrued to the current LV holders.

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

No response

Impact

Fees are distrusted incorrectly, users who have held LV tokens since the beginning will receive an equal amount of fees with users who deposit much later, even if no fees are generated by the protocol since the last user deposited. The last users to deposit are effectively stealing fees from the users who deposited earlier.

PoC

Gist After following the steps in the above mentioned gist add the following tests to the `AuditorTests.t.sol` file:

```
function test_IncorrectFeeAccrural() public {
    vm.startPrank(alice);
    WETH.mint(alice, 1e18);
    WETH.approve(address(moduleCore), type(uint256).max);
    moduleCore.depositLv(id, 1e18);
    Asset(lvAddress).approve(address(moduleCore), type(uint256).max);
    moduleCore.requestRedemption(id, 1e18);
    vm.stopPrank();

    /// @notice add 1e18 WETH to the amm pair, imagine this is generated from fees
    IUniswapV2Pair univ2Pair = flashSwapRouter.getUniV2pair(id, 1);
```

```
    WETH.mint(address(univ2Pair), 1e18);

    vm.startPrank(bob);
    WETH.mint(bob, 1e18);
    WETH.approve(address(moduleCore), type(uint256).max);
    moduleCore.depositLv(id, 1e18);
    Asset(lvAddress).approve(address(moduleCore), type(uint256).max);
    moduleCore.requestRedemption(id, 1e18);
    vm.stopPrank();

    /// @notice skip 1100 seconds so the CT and DS tokens expire
    skip(1100);

    vm.startPrank(alice);
    console2.log("WETH balance of alice before she redeems: ",
↪   WETH.balanceOf(alice));
    moduleCore.redeemExpiredLv(id, alice, 1e18);
    console2.log("WETH balance of alice after she has redeemed: ",
↪   WETH.balanceOf(alice));
    vm.stopPrank();

    vm.startPrank(bob);
    console2.log("WETH balance of bob before he redeems: ", WETH.balanceOf(bob));
    moduleCore.redeemExpiredLv(id, bob, 1e18);
    console2.log("WETH balance of bob after he has redeemed: ",
↪   WETH.balanceOf(bob));
    assertEq(WETH.balanceOf(alice), WETH.balanceOf(bob));
    vm.stopPrank();
}
```

```
Logs:
  WETH balance of alice before she redeems:  0
  WETH balance of alice after she has redeemed:  1499999999999998497
  WETH balance of bob before he redeems:  0
  WETH balance of bob after he has redeemed:  1499999999999998497
```

To run the test use: `forgetest-vvv--mttest_IncorrectFeeAccrural`

```
function test_IncorrectFeeAccruralBetweenDSIssuings() public {
    vm.startPrank(alice);
    WETH.mint(alice, 1e18);
    WETH.approve(address(moduleCore), type(uint256).max);
    moduleCore.depositLv(id, 1e18);
    Asset(lvAddress).approve(address(moduleCore), type(uint256).max);
    moduleCore.requestRedemption(id, 1e18);
    vm.stopPrank();

    /// @notice add 1e18 WETH to the amm pair, imagine this is generated from fees
    IUniswapV2Pair univ2Pair = flashSwapRouter.getUniV2pair(id, 1);
```

```
    WETH.mint(address(univ2Pair), 1e18);

    vm.startPrank(bob);
    WETH.mint(bob, 1e18);
    WETH.approve(address(moduleCore), type(uint256).max);
    moduleCore.depositLv(id, 1e18);
    Asset(lvAddress).approve(address(moduleCore), type(uint256).max);
    moduleCore.requestRedemption(id, 1e18);
    vm.stopPrank();

    vm.startPrank(owner);
    /// @notice the first issuance of DS and Ct tokens expires
    skip(1100);
    corkConfig.issueNewDs(id, block.timestamp + expiry, 1e18, 5e18);
    vm.stopPrank();

    vm.startPrank(tom);
    WETH.mint(tom, 1e18);
    WETH.approve(address(moduleCore), type(uint256).max);
    moduleCore.depositLv(id, 1e18);
    Asset(lvAddress).approve(address(moduleCore), type(uint256).max);
    moduleCore.requestRedemption(id, 1e18);
    vm.stopPrank();

    /// @notice add 1e18 WETH to the amm pair, imagine this is generated from fees
    WETH.mint(address(univ2Pair), 0.3e18);
    skip(1100);

    vm.startPrank(alice);
    console2.log("WETH balance of alice before she redeems: ",
↪   WETH.balanceOf(alice));
    moduleCore.redeemExpiredLv(id, alice, 1e18);
    console2.log("WETH balance of alice after she has redeemed: ",
↪   WETH.balanceOf(alice));
    vm.stopPrank();

    vm.startPrank(bob);
    console2.log("WETH balance of bob before he redeems: ", WETH.balanceOf(bob));
    moduleCore.redeemExpiredLv(id, bob, 1e18);
    console2.log("WETH balance of bob after he has redeemed: ",
↪   WETH.balanceOf(bob));
    assertEq(WETH.balanceOf(alice), WETH.balanceOf(bob));
    vm.stopPrank();

    vm.startPrank(tom);
    console2.log("WETH balance of tom before he redeems: ", WETH.balanceOf(tom));
    moduleCore.redeemExpiredLv(id, tom, 1e18);
    console2.log("WETH balance of tom after he has redeemed: ",
↪   WETH.balanceOf(tom));
    assertEq(WETH.balanceOf(alice), WETH.balanceOf(tom));
```

```
    vm.stopPrank();
}
```

```
Logs:
  WETH balance of alice before she redeems:  0
  WETH balance of alice after she has redeemed:  1333333333333331663
  WETH balance of bob before he redeems:  0
  WETH balance of bob after he has redeemed:  1333333333333331663
  WETH balance of tom before he redeems:  0
  WETH balance of tom after he has redeemed:  1333333333333331663
```

To run the test use: `forgetest-vvv--mttest_IncorrectFeeAccruralBetweenDSIssuings`

Mitigation

No response

Discussion

ziankork

This is fixed, by implementing an exchange rate for the LV (will add the link to it later)

cvetanovv

No impact. The protocol works as intended.

AtanasDimulski

Escalate, I don't agree that there is no impact. As I have mentioned in the impact section of the report, the last users who mint LV tokens via deposits will be stealing funds from the first users. This is not how rewarding contracts work. One user can mint LV tokens in the first round, and if 10 rounds pass and then someone else mints LV tokens, they will receive the same rewards. This is clearly stealing rewards from users. The protocol has confirmed and fixed it, so any claim that this is a design decision is incorrect.

sherlock-admin3

> Escalate, I don't agree that there is no impact. As I have mentioned in the impact section of the report, the last users who mint LV tokens via deposits will be stealing funds from the first users. This is not how rewarding contracts work. One user can mint LV tokens in the first round, and if 10 rounds pass and then someone else mints LV tokens, they will receive the same rewards. This is clearly stealing rewards from users. The protocol has confirmed and fixed it, so any claim that this is a design decision is incorrect.

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.

0xsimao

This is a design choice of the protocol, they do not depend on time to accrue fees, which is fine.

AtanasDimulski

@0xsimao I just explained why this is not a design decision. Secondly, reward distributing functionality doesn't work like that. If the HoJ requires more arguments as to why and how this results in users losing funds I will provide them. Let's continue the discussion after the HoJ provides his POV.

0xsimao

The root cause in the report is extremely vague

> The protocol doesn't implement any mechanism to track when users minted LV tokens, or when they requested a redemption of their LV tokens, or what rewards were accrued to the current LV holders.

The current design of the protocol gives fees to users regardless of the time they spent there, which is fine according to the code and readme, which are the sources of truth.

The readme says

> there will likely be external arbitrage bots operating around our system but they are not within scope and act as any market participant

What is happening in these scenarios is that the exchange rate is better than the deposit, but this would never happen due to arbitrage, which is expected as per the readme. So users deposit 1 RA, get 1 LV and redeem 1 LV for more RA.

What would happen with arbitrage is that whenever the pool accrues fees, lps (the lp tokens from the univ2 pool) will be worth more and so will lvs (the liquidity token from the vault). As lvs are minted at 1:1 with the RA deposit, whenever the lv value increases (due to lp fees), it becomes profitable to deposit 1 RA, get 1 LV and `redeemEarly()`, receiving a part of the fees. In the process of depositing and redeeming early, the exchange rate (lv:ra) will get dilluted back to 1:1, as users redeeming early are getting more ra than what they burn of lv. This process will keep the exchange rate close to 1:1 (minus the feePrecentage of the protocol).

This is a byproduct of minting lv at 1:1 with ra when depositing. If current lv holders want to receive their fees, they can redeem early.

AtanasDimulski

@0xsimao so by your logic if I want to provide liquidity by minting LV tokens, for a certain period of time, let's say 10 blocks, and I want to get the maximum amount of fees, I would have to burn and mint LV tokens every single block(if the pair has generated some fees) in order to get the fees that I am owed, this makes no sense. Protocols that distribute rewards based on deposits usually use some variations of this algorithm. I don't see how

the possibility of the protocol utilizing arbitrage bots has anything to do with the issue. If it is because of the first POC I have provided, I don't intend to simulate 10 thousand swaps in order to simulate the pair generating fees. The fact that I can deposit exactly after the protocol is deployed and then another user deposits 10 years after that. In the next block, if we both withdraw our LV tokens, we will receive the same amount of fees. This is a definitive loss of funds for the first user, it is not a protocol design (the sponsor has confirmed that this is not the intended behavior), and it is not logical at all. In the readme only the different fee rates are described, nothing else.

0xsimao

> This is a definitive loss of funds for the first user

But the user can redeem the rewards before to prevent it. This design is suboptimal but it is not loss of funds.

AtanasDimulski

It is clearly a loss of funds, let's not continue with the wild speculation that the user is supposed to be frontrunning every other deposit of LV tokens, in order to get the fees he is owed. I don't intend to continue discussing the imaginary scenarios you are presenting that perfectly suit your agenda and are completely incorrect. I will refrain from further comments unless requested by the HoJ, I recommend you do the same, so we don't waste our time.

WangSecurity

Firstly, the fact that the sponsor set labels ”Sponsor confirmed” and ”Will fix” doesn't mean it's not a design decision and the current version of the protocol and may very well mean they just liked the design proposed in the report, i.e. accrue fees based on time.

Secondly, I've got a couple of questions about the report:

> The protocol collects fees, from users utilizing its functionality

So, the fees from swaps do not accrue right after the swap, but the protocol has to separately claim them, correct? In that sense, the users depositing into the protocol to claim fees do not provide any value to the protocol, i.e. it doesn't affect the liquidity of the pool and the price impact, correct?

0xsimao

> So, the fees from swaps do not accrue right after the swap, but the protocol has to separately claim them, correct?

The fees accrue on the lps of the uni v2 pool, which are held by lv holders. To capture these fees, lv holders need to redeem their lvs to withdraw the corresponding lps from the pool. If the pool had swaps, the lps will be worth more so when lv holders redeem they also get more.

> In that sense, the users depositing into the protocol to claim fees do not provide any value to the protocol, i.e. it doesn't affect the liquidity of the pool and the price impact, correct?

If users want to just claim the fees they can deposit and withdraw from the vault, which would slightly reduce the protocol's liquidity (the fees are liquidity on uni v2 pools, there is no distinction between fees or liquidity added, it just redeems pro-rata to the lps burned).

However, in this process if they call `VaultLib::redeemEarly()`, they also pay a fee. They may not pay a fee if they call VaultLib::redeemExpired(), but the exchange rate may be worse or there may not be liquidity from the previous issuance.

Essentially this design is weird, as the lv value will remain more or less constant over time and users can claim the same revenue indepedently of how long they have been staking.

AtanasDimulski

> Firstly, the fact that the sponsor set labels ”Sponsor confirmed” and ”Will fix” doesn't mean it's not a design decision and the current version of the protocol and may very well mean they just liked the design proposed in the report, i.e. accrue fees based on time.

I believe the sponsor confirmed tag and their comment

> This is fixed, by implementing an exchange rate for the LV (will add the link to it later)

Clearly indicates that they believe something was wrong with the codebase, and they fixed it, they didn't say we decided to improve it a bit.

As to your question

> So, the fees from swaps do not accrue right after the swap, but the protocol has to separately claim them, correct?

I believe @0xsimao has explained it well.

The second question I don't understand clearly and I may provide a bit of a wrong answer, feel free to correct me

> In that sense, the users depositing into the protocol to claim fees do not provide any value to the protocol, i.e. it doesn't affect the liquidity of the pool and the price impact, correct?

The purpose of the LV token and why users are incentivized to deposit it, it so that there is a liquidity for a RA:CT UniV2 pair, and there can be trading. UniV2 pairs are extremally dependent on the liquidity within the contract, the less liquidity the bigger the slippage. Let's take a step back and consider how a UniV2 pair is supposed to work on its own. First liquidity has to be provided for the two tokens if there is one LP provider he will be accruing all the fees. Some time passes the pool is doing well, it is generating fees, a second LP provider comes and deposits liquidity(let's say an equal amount to the first provider so fees should be split 50/50 between them). If they both decide to withdraw at the same time, the first LP provider would get all the fees for the first period where he was the only LP + 50% of the fees for the second period. The second LP will get only 50% of the fees for the second period. For simplicity lets consider in the first period the fees that were generate were 10 tokenX and 10 tokenY, same for the second period. Now the first LP provider will receive 15 tokenX and 15 tokenY, the second LP provider will receive 5

tokenX and 5 tokenY. The first provider had his collateral deposited for a longer period of time where he was taking more risk, for example the price of the token may have dropped, he may have staked it somewhere else and receive more rewards for example ETH may be staked for stEHT. The possibility of impermanent loss should also be considered. I don't believe this should be considered as an opportunity loss, because the problem of misdistribution of fees comes from the logic of the Cork implementation.

However in the case of the Cork protocol if we have the same scenario as described above both of the LPs (people who mint the LV token) will receive 10 tokenX and 10 tokenY. This is clearly a theft of funds from the first LP provider. If for example I have been providing liquidity for 30 days and accruing fees, someone deposits the same amount of LV tokens, and then in the next block we both redeem our LV tokens we will both receive the same amount of fees. The second depositor have risked his collateral for a total of 2 blocks, and receives the same amount of fees as the first who risked his collateral for 30 days and did a great favor to the protocol by providing liquidity to the pair.

Secondly the LV token is the same for a RA:PA pair, but the DS and CT tokens have an expiration, and multiple DS and CT tokens can be minted, thus different RA:CT uniV2 pairs can exist. When DS and CT tokens expire, the LP owned by the protocol in the pair is liquidated, the CT is converted to RA, then a new uniV2 pair is created for the newly issued CT token, the RA is split between RA and the new CT token and deposited in the new uniV2 pair. The protocol generates fees from the redeemRaWithDS() and repurchase() functions as well, the fee is split between RA and CT and deposited in the uniV2 pair. Now if we have the same case as above someone mints LV at the first block, users utilize the protocol and protocol fees are generated, lets say 10 RA and another user mints LV at the last block, when the first DS and CT tokens expire and a new RA:CT2 pair is created all those fees will be mashed together, and again the second user will profit tremendously.

WangSecurity

Thank you for these great explanations. I'll re-iterate to confirm I understand everything correctly:

When we deposit into the pool, we basically add liquidity for this RA/PA pair on UniV2 which improves the slippage. So let's observe the attack in this report with an example: One user wants to initiate a very large swap, let's say 1M USDC volume. The attacker sees it and deposits into Cork. They get a portion of the fees, but they also contributed to that swap and improved slippage, i.e. e.g. without the attacker depositing the user would lose 50K due to slippage and price impact, but with the attacker deposit, they've ended with only 1K slippage and received 999K USDC in the end.

So while the attacker received the portion of the fee, they also contributed to the pool and made slippage better. Is the example above correct? As I understand it wouldn't be possible to also amplify flash loans, since the attacker would be able to withdraw only in the next block?

AtanasDimulski

The uniV2 pair that the protocol creates is for RA for example ETH, and a CT token that the protocol creates and controls the mining of. The PA can be stETH for example which

is supposed to be pegged to RA. The CT token has a different purpose within the protocol than the PA token. Users who mint LV tokens deposit RA token into the Cork protocol, which is split between RA and CT and is fully deposited into the uniV2 pair, we can consider them as LPs. If someone frontruns a big swap and deposits to the pair he should also receive fees from that big swap. Problem is if there is 1 big swap and lets say only user A has provided liquidity, the pair generates 1k fees, after the swap user B provides liquidity by minting LV tokens, then both user A and user B redeem their LV tokens they will both receive 500 in fees. The vulnerability I tried to describe is not about a depositor frontruning big swaps and generating the same fees, but more like depositing after the big swap, not providing any liquidity that helps with slippage for said big swap, and receiving the same fees as the depositor who provided the only liquidity before the big swap. If only User A has provided liquidity before the big swap that generates 1k in fees, he should receive the whole fee, however this is not the case.

WangSecurity

Oh, I see, so it returns a bit to my comment before it. The fees are generated **not in the same transaction** as the swap. The fees are then claimed separately, and the attacker can deposit just to receive the fees without providing liquidity for that big swap, for example, correct?

AtanasDimulski

> the attacker can deposit just to receive the fees without providing liquidity

This part is correct, and it doesn't have to be an attacker, just a user depositing at some random time. This is simply how the protocol works. A bit of clarification on the fee generation part. The fees are accrued to the LP tokens in the same transaction as the swap. In the case of uniV2 LP tokens are like shares. UniV2 is like ERC4626 vault. Users have to burn their LP tokens in order to get back their liquidity + fees. Say there is 100 liquidity and 10 LP shares, after the swap there is 110 liquidity and still 10 LP shares, now if the LP burns his shares he gets 110 liquidity - 100 the original liquidity he deposited + 10 from the generated fees. 1 LP share is now worth 11 liquidity. The redemption in this case will be (10 * 110) / 10 = 110. If the first user hasn't redeemed, the swap has happened and then another user comes in and deposits another 100 liquidity his shares will be calculated in the following way (100 * 10) / 110 ~ 9, now if they both redeem at the same time, with no other fees user A will get for his 10 shares (10 * 210) / 19 ~ 110, user B for his 9 shares will get (9 * 100) / 9 = 100. However, since the Cork protocol owns the LP tokens or the shares in the above example. The same steps happen as above, user A and user B now own 1 LV token each. The protocol owns 19 LP shares, but each 1 LV token is worth 19/2 = 9,5 shares. Thus when the LV are redeem user A and user B will receive the same amount of fees.

WangSecurity

Thank you very much for this explanation. I think I've got the very last question to make the decision: For this attack, the attacker doesn't depend on large swaps, and they can wait for lots of small swaps to happen, so Cork protocol accrues fees and then executes the attack, correct?

AtanasDimulski

Yes UniV2 swap fees are 0.3% for each swap, the attacker can wait for a month for example for thousands of small fees to occur, and then provide liquidity, the result will be the same.

WangSecurity

Thank you again for the clarification. I agree this is a valid finding, the attacker doesn't contribute to the protocol and would receive the fees even for swaps they didn't provide liquidity for. This can be viewed as a design decision, but this is a loss of funds (in this case, fees), so valid finding.

About the constraints, the one I see here is paying the early redemption fee, but the attacker can just wait until the fees stack up and exceed the early redemption fee. The second one is the attacker would need a relatively large capital to get a large part of the fee. I don't see them as extensive limitations. Hence, high severity should be appropriate.

Planning to accept the escalation and validate with high severity. Are there any duplicates?

AtanasDimulski

Hey, @WangSecurity I haven't checked extensively, but @cvetanovv didn't group this finding with anything else + I haven't seen something similar being escalated.

WangSecurity

Result: High Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- AtanasDimulski: accepted

Issue H-4: Incorrect redeemAmount Is Accounted Due To Not Accounting For The Exchange Rate

Source: https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/119

Found by

0x73696d616f, 0xNirix, Ace-30, KupiaSec, Pheonix, korok, oxelmiguel, sakshamguruji, vinica_boy

Summary

When Liquidating LP , DS and CT are paired , then that amount is used to redeem RA . But the accounting for RA has been done incorrectly since it does not account for exchange rate.

Vulnerability Detail

1.) Inside Liquidate LP we empty out the DS reserve and pair it up with the CT amount returned from the AMM ->

https://github.com/sherlock-audit/2024-08-cork-protocol/blob/main/Depeg-swap/contracts/libraries/VaultLib.sol#L376

This is the amount of CtDs being redeemed.

2.) This same amount has been accounted for the increment in RA ->

https://github.com/sherlock-audit/2024-08-cork-protocol/blob/main/Depeg-swap/contracts/libraries/VaultLib.sol#L390

But this is incorrect , this is because `redeemAmount` is an amount in Ct/Ds not in RA , to make it into RA we need to apply the exchange rate over it(exchange rate is how many Redemption Assets you need to deposit to receive 1 Cover Token + 1 Depeg Swap and how many Redemption Assets you receive when redeeming 1 Pegged Asset + 1 Depeg Swap , read more in the Dealing with non-rebasing Pegged Assets section -> https://corkfi.notion.site/Cork-Protocol-Litepaper-f21a57d5c19d48209dfa0f0c2ab776c4).

3.) Therefore incorrect RA amount has been accounted and incorrect amount of RA would be reserved ->

https://github.com/sherlock-audit/2024-08-cork-protocol/blob/main/Depeg-swap/contracts/libraries/VaultLib.sol#L392

meaning , incorrect amount of RA attributed to be withdrawn/redeemed.

4.) This inconsistency is found at multiple places , and instead of making separate reports im listing them here ->

a.) https://github.com/sherlock-audit/2024-08-cork-protocol/blob/main/Depeg-swap/contracts/libraries/PsmLib.sol#L122

The amount here is in RA and we are issuing DS/CT with it.

b.) https://github.com/sherlock-audit/2024-08-cork-protocol/blob/main/Depeg-swap/contracts/core/flash-swaps/FlashSwapRouter.sol#L368

dsAttributed is in DS and we are depositing RA.

c.) https://github.com/sherlock-audit/2024-08-cork-protocol/blob/main/Depeg-swap/contracts/libraries/VaultLib.sol#L331

redeemAmount is in Ct/Ds here

Impact

Code Snippet

https://github.com/sherlock-audit/2024-08-cork-protocol/blob/main/Depeg-swap/contracts/libraries/VaultLib.sol#L390

Tool Used

Manual Review

Recommendation

Account for the exchange rate correctly.

Discussion

ziankork

This is valid, will fix

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Cork-Technology/Depeg-swap/pull/81

Issue H-5: Incoming Redemption Assets not being tracked when repurchase is called

Source: https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/126

Found by

0x73696d616f, 0xNirix, Ace-30, KupiaSec, Pheonix, dimulski, sakshamguruji, steadyman, vinica_boy

Summary

`repurchase()` function take redemption asset and gives back depeg swap along with pegged. However, the incoming redemption asset is not being tracked via `lockFrom` but there's a direct transfer of ra via `lockUnchecked()` causing mismatch in ra accounting.

Vulnerability Detail

Users deposit redemption asset via deposit to get back Cover Token + DepegSwap which can be redeemed back using various combinations.

1. Redeem with CoverToken + DepegSwap (only before expiry of DS)
2. Redeem with DepegSwap + Pegged Asset (only before expiry of DS)
3. Redeem with CoverToken (only after expiry)

Is users use 2. as a means to redeem deposited RA, they can repurchase DepegSwap + Pegged Asset back using `repurchase()` The `repurchase()` function is designed for getting back a combination of DepegSwap + Pegged Asset by giving Redemption Asset.

Incoming and outgoing RA is tracked via PsmRedemptionAssetManager struct attached with State struct. Consider the system before expiry. Let's look at all the places where RA can come and go and how internal tracking is changing respectively.

1. deposit() : ra increased
2. redeem with DS + PA : ra decreased
3. redeem with CT + DS : ra decreased Everytime there is incoming of ra, the contracts increases internal state variable (`locked`) with that amount.

```
struct PsmRedemptionAssetManager {
    address _address;
    uint256 locked;
```

```
    uint256 free;
}
```

But this is not increased when ra comes with `repurchase()` . This can be problematic after time of expiry. After expiry, when users redeems with ct by calling `redeemWithCt()` it internally invokes `_separateLiquidity()`.

```
function _separateLiquidity(State storage self, uint256 prevIdx) internal {
    if (self.psm.liquiditySeparated.get(prevIdx)) {
        return;
    }


    DepegSwap storage ds = self.ds[prevIdx];
    Guard.safeAfterExpired(ds);


    uint256 availableRa = self.psm.balances.ra.convertAllToFree();
    uint256 availablePa = self.psm.balances.paBalance;


    self.psm.poolArchive[prevIdx] = PsmPoolArchive(availableRa, availablePa,
↪   IERC20(ds.ct).totalSupply());


    // reset current balances
    self.psm.balances.ra.reset();
    self.psm.balances.paBalance = 0;


    self.psm.liquiditySeparated.set(prevIdx);
}
```

In this function all the accumulated ra goes to pool archive. This is achieved by zeroing out the variable that was tracking incoming and outgoing ra (by `reset()`) and storing it in PoolArchive.

But since repurchase does not change this internal variable, at the time of `_separateLiquidity()` , actual ra in the system will be much more than what is expected.

Impact

Since protocol's internal tracking assumes less ra in the system than the actual amount, the remaining amount will be stuck in the contract causing direct loss of funds.

Code Snippet

https://github.com/sherlock-audit/2024-08-cork-protocol/blob/main/Depeg-swap/contracts/libraries/PsmLib.sol#L293-L322

Tool Used

Manual Review

Recommendation

Increase locked ra amount whenever repurchase is called by calling `lockFrom` instead of `lockUnchecked()`

Discussion

ziankork

this is valid, although we already fixed this for the trading competition. will provide links later

ziankork

the fix recommended will cause an issue if there's a fee as #155 described. the fix can also be seen there

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Cork-Technology/Depeg-swap/pull/77

Issue H-6: Users will steal excess funds from the Vault due to `VaultPoolLib::redeem()` not always decreasing `self.withdrawalPool.raBalance` and `self.withdrawalPool.paBalance`

Source: https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/144

The protocol has acknowledged this issue.

Found by

0x73696d616f, sakshamguruji

Summary

Vault::redeemExpiredLv() calls VaultLib::redeemExpired(), which allows users to withdraw funds after expiry, even if they have not requested a redemption. This redemption happens in VaultPoolLib::redeem(), when userEligible < amount, it calls internally __redeemExcessFromAmmPool(), where only self.ammLiquidityPool.balance is reduced, but not `self.withdrawalPool.raBalance` and `self.withdrawalPool.paBalance`. As such, when calculing the withdrawal pool balance in the next issuance on VaultPoolLibrary::reserve(), it will double count all the already withdraw self.withdrawalPool.raBalance and self.withdrawalPool.paBalance, allowing users to withdraw the same funds twice.

Root Cause

In `VaultPoolLib::__redeemExcessFromAmmPool()`, `self.withdrawalPool.raBalance` and `self.withdrawalPool.paBalance` are not decreased, but `ra` and `pa` are also withdrawn from the withdrawal pool when the user has partially requested redemption.

Internal pre-conditions

1. User requests redemption of an amount smaller than the total withdrawn in `VaultLib::redeemExpired()`, that is, `userEligible<amount`.

External pre-conditions

None.

Attack Path

1. User calls `Vault::redeemExpiredLv()`, withdrawing from the withdrawal pool, but `self.withdrawalPool.raBalance` and `self.withdrawalPool.paBalance` are not decreased.
2. A new issuance starts, and in `VaultPoolLib::reserve()`, the funds are double counted as not all withdrawals were reduced.
3. As such, `self.withdrawalPool.raExchangeRate` and `self.withdrawalPool.paExchangeRate` will be inflated by double the funds and users will redeem more funds than they should, leading to the insolvency of the Vault.

Impact

Users steal funds while unaware users will not be able to withdraw.

PoC

`__tryRedeemExcessFromAmmPool()` does not decrease the withdrawn `self.withdrawalPool.raBalance` and `self.withdrawalPool.paBalance`.

```
function __tryRedeemExcessFromAmmPool(VaultPool storage self, uint256
↪  amountAttributed, uint256 excessAmount)
    internal
    view
    returns (uint256 ra, uint256 pa, uint256 withdrawnFromAmm)
{
    (ra, pa) = __tryRedeemfromWithdrawalPool(self, amountAttributed);

    withdrawnFromAmm =
        MathHelper.calculateRedeemAmountWithExchangeRate(excessAmount,
↪  self.withdrawalPool.raExchangeRate); //@audit PA is ignored here

    ra += withdrawnFromAmm;
}
```

Mitigation

Replace `__tryRedeemfromWithdrawalPool()` with `__redeemfromWithdrawalPool()`.

```
function __tryRedeemExcessFromAmmPool(VaultPool storage self, uint256
↪  amountAttributed, uint256 excessAmount)
    internal
    view
    returns (uint256 ra, uint256 pa, uint256 withdrawnFromAmm)
{
```

```
    (ra, pa) = __redeemfromWithdrawalPool(self, amountAttributed);

    withdrawnFromAmm =
        MathHelper.calculateRedeemAmountWithExchangeRate(excessAmount,
↪   self.withdrawalPool.raExchangeRate); //@audit PA is ignored here

    ra += withdrawnFromAmm;
}
```

Discussion

ziankork

This is a valid issue, though we removed the functionality of expiry redemption on the LV, so this would have no impact on the updated version of the protocol. still valid nonetheless

Issue H-7: Wrong accounting of locked RA when repurchasing DS+PA with RA

Source: https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/155

Found by

0x73696d616f, dimulski, vinica_boy

Summary

Users have the option to repurchase DS + PA by providing RA to the PSM. A portion of the RA provided is taken as a fee, and this fee is used to mint CT + DS for providing liquidity to the AMM pair.

Vulnerability Detail

NOTE: Currently `PsmLib.sol` incorrectly uses `lockUnchecked()` for the amount of RA provided by the user. As discussed with sponsor it should be `lockFrom()` in order to account for the RA provided.

After initially locking the RA provided, part of this amount is used to provide liquidity to the AMM via `VaultLibrary.provideLiquidityWithFee()`

```
function repurchase(
        State storage self,
        address buyer,
        uint256 amount,
        IDsFlashSwapCore flashSwapRouter,
        IUniswapV2Router02 ammRouter
    ) internal returns (uint256 dsId, uint256 received, uint256 feePrecentage,
↪   uint256 fee, uint256 exchangeRates) {
        DepegSwap storage ds;

        (dsId, received, feePrecentage, fee, exchangeRates, ds) =
↪   previewRepurchase(self, amount);

        // decrease PSM balance
        // we also include the fee here to separate the accumulated fee from the
↪   repurchase
        self.psm.balances.paBalance -= (received);
        self.psm.balances.dsBalance -= (received);

        // transfer user RA to the PSM/LV
```

```
        // @audit-issue shouldnt it be lock checked, deposit -> redeemWithDs ->
↪  repurchase - locked would be 0
        self.psm.balances.ra.lockUnchecked(amount, buyer);

        // transfer user attrubuted DS + PA
        // PA
        (, address pa) = self.info.underlyingAsset();
        IERC20(pa).safeTransfer(buyer, received);

        // DS
        IERC20(ds._address).transfer(buyer, received);

        // Provide liquidity
        VaultLibrary.provideLiquidityWithFee(self, fee, flashSwapRouter, ammRouter);
    }
```

`provideLiqudityWithFee` internally uses `__provideLiquidityWithRatio()` which calculates the amount of RA that should be used to mint CT+DS in order to be able to provide liquidity.

```
function __provideLiquidityWithRatio(
        State storage self,
        uint256 amount,
        IDsFlashSwapCore flashSwapRouter,
        address ctAddress,
        IUniswapV2Router02 ammRouter
    ) internal returns (uint256 ra, uint256 ct) {
        uint256 dsId = self.globalAssetIdx;

        uint256 ctRatio = __getAmmCtPriceRatio(self, flashSwapRouter, dsId);

        (ra, ct) = MathHelper.calculateProvideLiquidityAmountBasedOnCtPrice(amount,
↪  ctRatio);

        __provideLiquidity(self, ra, ct, flashSwapRouter, ctAddress, ammRouter,
↪  dsId);
    }
```

`__provideLiquidity()` uses `PsmLibrary.unsafeIssueToLv()` to account the RA locked.

```
function __provideLiquidity(
        State storage self,
        uint256 raAmount,
        uint256 ctAmount,
        IDsFlashSwapCore flashSwapRouter,
        address ctAddress,
        IUniswapV2Router02 ammRouter,
        uint256 dsId
    ) internal {
```

```
        // no need to provide liquidity if the amount is 0
        if (raAmount == 0 && ctAmount == 0) {
            return;
        }

        PsmLibrary.unsafeIssueToLv(self, ctAmount);

        __addLiquidityToAmmUnchecked(self, raAmount, ctAmount,
↪   self.info.redemptionAsset(), ctAddress, ammRouter);

        _addFlashSwapReserve(self, flashSwapRouter, self.ds[dsId], ctAmount);
    }
```

RA locked is incremented with the amount of CT tokens minted.

```
function unsafeIssueToLv(State storage self, uint256 amount) internal {
        uint256 dsId = self.globalAssetIdx;

        DepegSwap storage ds = self.ds[dsId];

        self.psm.balances.ra.incLocked(amount);

        ds.issue(address(this), amount);
    }
```

Consider the following scenario: For simplicity the minted values in the example may not be accurate, but the idea is to show the wrong accounting of locked RA.

1. PSM has 1000 RA locked.
2. Alice repurchase 100 DS+PA with providing 100 RA and we have fee=5% making the fee = 5
3. PSM will have 1100 RA locked and ra.locked would be 1100 also.
4. In `__provideLiquidity()` let say 3 of those 5 RA are used to mint CT+DS.
5. `PsmLibrary.unsafeIssueToLv()` would add 3 RA to the locked amount, making the psm.balances.ra.locked = 1103 while the real balance would still be 1100.

Impact

Wrong accounting of locked RA would lead to over-distribution of rewards for users + after time last users to redeem might not be able to redeem as there wont be enough RA in the contract due to previous over-distribution. This breaks a core functionality of the protocol and the likelihood of this happening is very high, making the overall severity High.

Code Snippet

`PsmLib.repurchase()`:
https://github.com/sherlock-audit/2024-08-cork-protocol/blob/db23bf67e45781b00ee6de5f6f23e621af16bd7e/Depeg-swap/contracts/libraries/PsmLib.sol#L293

Tool Used

Manual Review

Recommendation

Consider either not accounting for the fee used to mint CT+DS as it is already accounted or do not initially account for the fee when users provide RA.

Discussion

ziankork

This is a valid issue, will fix by not accounting the fee when users provide RA

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Cork-Technology/Depeg-swap/pull/77

spdimov

All the selected duplicates describe different issues: #57, #153, #179 describe the issue in the redeem RA with DS case #171 describes the issue regarding return amounts when providing liquidity to the AMM #219 is a duplicate of #119 #40 is describing something completely different

cvetanovv

Escalate above comment

sherlock-admin2

> Escalate above comment

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.

0xsimao

#57, #153, #179 are valid dups. They all mention the fee used to provide liquidity is not decreased from the balance of the psm. #171 is a dup of #240. #219 is a dup of #119. #40 is invalid, explanation is in #40 itself.

spdimov

I will agree with the previous comment. #57, #153, #179 really describe the same root cause issue and based on other groupings of issues in this contest, maybe they have to be counted as one with the issue described here. I thought that when two similar issues happen in different places in the code, they should be separated as fixing one would not fix the other.

AtanasDimulski

I know this is not part of the original escalation, but since the issue is escalated, I believe it should be of high severity, as it essentially results in the first users that withdraw stealing funds from the last users that withdraw.

WangSecurity

I agree with the duplication explained in this comment and plan to apply it. About the severity, as I understand, there are indeed no limitations on this issue and a couple of last users could end up with no rewards at all. Hence, planning to accept the escalation, apply the duplication explained here and increase the severity to high.

WangSecurity

Result: High Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- cvetanovv: accepted

Issue H-8: Admin new issuance or user calling `Vault::redeemExpiredLv()` after `Psm::redeemWithCt()` will lead to stuck funds when trying to withdraw

Source: https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/156

This issue was not included in the Fix Review and has not been fixed.

Found by

0x73696d616f

Summary

VaultLib::_liquidatedLp() calls PsmLib::lvRedeemRaWithCtDs(), which redeems `ra` with `ct` and `ds`. However, if PsmLib::_separateLiquidity() has already been called, this will lead to an incorrect tracking of funds as `PsmLib::_separateLiquidity()` checkpointed the total supply of `ct`, but the `Vault` will redeem some of it using `PsmLib::lvRedeemRaWithCtDs()`, leading to some `pa` that will never be withdrawn and the vault withdraws too many `Ra`, which means users will not be able to redeem their `ct` for `ra` and `pa` as `ra` has been withdrawn already and it reverts.

Root Cause

In `VaultLib.sol:377`, it calls `PsmLib::lvRedeemRaWithCtDs()` even if `PsmLib::_separateLiquidity()` has already been called. It should skip it in this case.

Internal pre-conditions

1. `PsmLib::_separateLiquidity()` needs to be called before `VaultLib::_liquidatedLp()`, which may be done on a new issuance when the admin calls ModuleCore::issueNewDs() or by users calling `Psm::redeemWithCT()` before `Vault::redeemExpiredLv()`.

External pre-conditions

None.

Attack Path

1. Admin calls `ModuleCore::issueNewDs()`. Or users call `Psm::redeemWithCT()` before `Vault::redeemExpiredLv()`.

Impact

As the `Vault` withdraws `Ra` after the checkpoint and burns the corresponding `Ct` tokens, it will withdraw too many `ra` and not withdraw the `pa` it was entitled to.

PoC

`ModuleCore::issueNewDs()` calls `PsmLib::onNewIssuance()` before `VaultLib::onNewIssuance()` always triggering this bug.

```
function issueNewDs(Id id, uint256 expiry, uint256 exchangeRates, uint256
↪  repurchaseFeePrecentage)
    external
    override
    onlyConfig
    onlyInitialized(id)
{
    ...
    PsmLibrary.onNewIssuance(state, ct, ds, ammPair, idx, prevIdx,
↪  repurchaseFeePrecentage);

    getRouterCore().onNewIssuance(id, idx, ds, ammPair, 0, ra, ct);

    VaultLibrary.onNewIssuance(state, prevIdx, getRouterCore(), getAmmRouter());
    ...
}
```

`PsmLib::_separateLiquidity()` checkpoints `ra` and `pa` based on `ct` supply:

```
function _separateLiquidity(State storage self, uint256 prevIdx) internal {
    ...
    self.psm.poolArchive[prevIdx] = PsmPoolArchive(availableRa, availablePa,
↪  IERC20(ds.ct).totalSupply());
    ...
}
```

`VaultLib::_liquidatedLp()` redeems `ra` with `ct` and `ds` when it should have skipped it has liquidity has already been checkpointed in `PsmLib::_separateLiquidity()`.

```
function _liquidatedLp(
    State storage self,
    uint256 dsId,
```

```
    IUniswapV2Router02 ammRouter,
    IDsFlashSwapCore flashSwapRouter
) internal {
    ...
    PsmLibrary.lvRedeemRaWithCtDs(self, redeemAmount, dsId);

    // if the reserved DS is more than the CT that's available from liquidating the
↪   AMM LP
    // then there's no CT we can use to effectively redeem RA + PA from the PSM
    uint256 ctAttributedToPa = reservedDs >= ctAmm ? 0 : ctAmm - reservedDs;

    uint256 psmPa;
    uint256 psmRa;

    if (ctAttributedToPa != 0) {
        (psmPa, psmRa) = PsmLibrary.lvRedeemRaPaWithCt(self, ctAttributedToPa,
↪   dsId);
    }

    psmRa += redeemAmount;

    self.vault.pool.reserve(self.vault.lv.totalIssued(), raAmm + psmRa, psmPa);
}
```

Mitigation

If the liquidity has been separated, skip redeeming `ra` for `ct` and `ds`.

```
function _liquidatedLp(
    State storage self,
    uint256 dsId,
    IUniswapV2Router02 ammRouter,
    IDsFlashSwapCore flashSwapRouter
) internal {
    ...
    if (!self.psm.liquiditySeparated.get(prevIdx)) {
        PsmLibrary.lvRedeemRaWithCtDs(self, redeemAmount, dsId);
    }
    ...
}
```

Discussion

ziankork

related to #44, the function that's problematic here is `PsmLibrary::lvRedeemRaWithCtDs`. will fix

0xsimao

escalate

This issue is high severity because it has no external dependencies, it will happen whenver new DS tokens are issued.

sherlock-admin2

> escalate
>
> This issue is high severity because it has no external dependencies, it will happen whenver new DS tokens are issued.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

sherlock-admin4

> Escalate, This issue is invalid

You've deleted an escalation for this issue.

0xsimao

The problem is not in `PsmLibrary::lvRedeemRaWithCtDs`, but in `VaultLib::allowbreak _liquidatedLp()`. Even if it was, just because it is the same function does not mean they are dups.

cvetanovv

The reason why this issue is Medium is because it has an external condition. This issue can only happen under certain conditions mentioned in the issue.

> `PsmLib::allowbreak _separateLiquidity()` needs to be called before `VaultLib::allowbreak _liquidatedLp()`, which may be done on a new issuance when the admin calls ModuleCore::issueNewDs() or by users calling `Psm::redeemWithCT()` before `Vault::redeemExpiredLv()`.

0xsimao

@cvetanovv that is not an external condition, it's within the protocol (same contract to be more precise). There are 2 ways for this to happen:

1. Any user can force this order of events at no cost by calling `Psm::redeemWithCT()` before `Vault::redeemExpiredLv()`.
2. `ModuleCore::issueNewDs()` is called, in which it **always** calls `PsmLib::allowbreak _separateLiquidity()` before `VaultLib::allowbreak _liquidatedLp()`.

So 2 means it always happens and 1 means anyone may trigger this. Which means this has internal conditions but they are not extensive, can be easily triggered by 1 and will always be triggered by 2.

davies0212

Skipping `lvRedeemRaWithCtDs()` is not a correct mitigation. Instead, the `raAccrued` and `paAccrued` values of the `self.psm.poolArchive[dsId]` struct should be reduced.

This issue arises from improper tracking of RA and PA amounts and should be considered a duplicate of #166, #126, or #155.

0xsimao

> Skipping lvRedeemRaWithCtDs() is not a correct mitigation. Instead, the raAccrued and paAccrued values of the self.psm.poolArchive[dsId] struct should be reduced.

There may be several mitigations, it's easier to verify the pr. But these linked issues are different than this one.

davies0212

> There may be several mitigations, it's easier to verify the pr. But these linked issues are different than this one.

For this issue, there are not two mitigations. Reducing the `raAccrued` and `paAccrued` values of `self.psm.poolArchive[dsId]` is the correct approach.

Additionally, mitigating in a different area does not mean that the issues are separate; instead, we should focus on the root cause.

0xsimao

> For this issue, there are not two mitigations. Reducing the raAccrued and paAccrued values of self.psm.poolArchive[dsId] is the correct approach.

I would have to check the pr to confirm.

> For this issue, there are not two mitigations. Reducing the raAccrued and paAccrued values of self.psm.poolArchive[dsId] is the correct approach.

Sherlock duplication rules require the root cause and attack path to be the same, which they are not. There are many different ways to make the accounting wrong. Even then, there are several assets in the protocol, ra, pa and ct.

This issue is related to ct accounting specifically, not ra or pa, as the ct attributed to the vault would not be used to redeem via `lvRedeemRaPaWithCt()`, but using `lvRedeemRaWithCtDs()` instead. This means a portion of `self.psm.poolArchive[prevIdx]` would never be redeemed, being stuck, as the ct that was supposed to redeem it pro-rata would be burned in `lvRedeemRaWithCtDs()`.

davies0212

> Sherlock duplication rules require the root cause and attack path to be the same, which they are not. There are many different ways to make the

> accounting wrong. Even then, there are several assets in the protocol, ra, pa and ct.

Issues related to accounting for the exchange rate as `1:1` were all grouped together. If you assert as stated above, they should also be separated.

> This issue is related to ct accounting specifically, not ra or pa, as the ct attributed to the vault would not be used to redeem via `lvRedeemRaPaWithCt()`, but using `lvRedeemRaWithCtDs()` instead. This means a portion of `self.psm.poolArchive[prevIdx]` would never be redeemed, being stuck, as the ct that was supposed to redeem it pro-rata would be burned in `lvRedeemRaWithCtDs()`.

No. The ct attributed to the vault should be used to redeem ra. The ct attributed to the vault belongs to the lv holders (who contributed to the Uniswap V2 pool). They sent ra to the vault, and some of that ra was exchanged for (ct + ds). The generated ct was provided to the Uniswap V2 pool along with the remaining ra, and the generated ds were sent to the FlashSwapRouter.

Therefore, when issuing new ds, the ct liquidated from the Uniswap V2 pool and the ds from the FlashSwapRouter should be reexchanged for ra. If they are not reexchanged, it will result in a loss of funds for the lv holders.

Thus, skipping `lvRedeemRaWithCtDs()` is not a correct mitigation. Instead, the `raAccrued` value of `self.psm.poolArchive[prevIdx]` should be reduced.

0xsimao

> Issues related to accounting for the exchange rate as 1:1 were all grouped together. If you assert as stated above, they should also be separated.

But that is always the same fix, taking the exchange rate into account.

> No. The ct attributed to the vault should be used to redeem ra. The ct attributed to the vault belongs to the lv holders (who contributed to the Uniswap V2 pool).

Lv holders just hold lv, the lps are held by the vault. Users do not hold all ct at this point, some of it is in the univ2 pool, a part of it owned by the vault and the corresponding lps.

> They sent ra to the vault, and some of that ra was exchanged for (ct + ds). The generated ct was provided to the Uniswap V2 pool along with the remaining ra, and the generated ds were sent to the FlashSwapRouter.

Exactly, the vault holds the lps, whose ct is in the uni v2 pool.

> Therefore, when issuing new ds, the ct liquidated from the Uniswap V2 pool and the ds from the FlashSwapRouter should be reexchanged for ra. If they are not reexchanged, it will result in a loss of funds for the lv holders. Thus, skipping lvRedeemRaWithCtDs() is not a correct mitigation. Instead, the raAccrued value of self.psm.poolArchive[prevIdx] should be reduced.

Again, I am not discussing the mitigation, but the bug is in the ct amount held by the vault. Look at the order of events

```
PsmLibrary.onNewIssuance(state, ct, ds, ammPair, idx, prevIdx,
↪  repurchaseFeePrecentage);

getRouterCore().onNewIssuance(id, idx, ds, ammPair, 0, ra, ct);

VaultLibrary.onNewIssuance(state, prevIdx, getRouterCore(), getAmmRouter());
```

Firstly, `PsmLibrary.onNewIssuance()` takes a checkpoint of the ct balances `self.psm.poolArchive[prevIdx]=PsmPoolArchive(availableRa,availablePa,IERC20(ds.ct).totalSupply());`. However, the vault was not liquidated yet, so some of the ct is still in the univ2 pool, and some of that is owned by the vault, that has lps. But the checkpoint is using the full ct `totalSupply()`. So `IERC20(ds.ct).totalSupply());` includes ct owned by the vault (as LP). Let's say that the total ct supply is 100 and the vault's lp is equivalent to 20 ct.

Secondly, `VaultLibrary.onNewIssuance()` is called, which calls `allowbreak _liquidatedLp()`. It burns the lps and gets the corresponding ct and ra via `allowbreak _ allowbreak _liquidateUnchecked()`. Assume it received the mentioned 20 ct. Then it calls `PsmLibrary.lvRedeemRaWithCtDs()`, which burns the 20 ct. So the total supply of ct is 80.

Lastly, as there was enough reserve, no ct is left to redeem with `lvRedeemRaPaWithCt()`.

If we look into `PsmLibrary.lvRedeemRaPaWithCt()`, the amounts to redeem are calculated as (it boils down to `MathHelper::calculateAccrued()`:

```
accrued = (amount * (available * 1e18) / totalCtIssued) / 1e18;
```

`totalCtIssued`, is the checkpointed `ct` before the vault was liquidated, 100. But, there is only 80 ct in circulation due to the vault having burned it. Thus, 20% of this will be stuck.

WangSecurity

About the severity, @0xsimao could you elaborate on the impact:

> As the Vault withdraws Ra after the checkpoint and burns the corresponding Ct tokens, it will withdraw too many ra and not withdraw the pa it was entitled to.

What would be the approximate loss here?

About the duplication, #166, #126 and #155 have different root causes, and this report shouldn't be a duplicate of any of these. Please correct me if it's wrong.

0xsimao

> What would be the approximate loss here?

Whatever the `allowbreak %` the vault has of the ct total supply, as it will never be redeemed. Should be a high % given that users are incentivized to provide liquidity through the vault which

accrues additional fees. Liquidity is provided by sending ra and ct, so by providing liquidity the vault also holds more ct when it burns the lps in the end.

WangSecurity

Thank you for this clarification. Since the loss can indeed be high and the constraints in the report are not extensive, planning to accept the escalation and upgrade the severity to high.

davies0212

In the `PsmLibrary.lvRedeemRaPaWithCt()` function, it calls `allowbreak _beforeCtRedeem()` function

```
    function lvRedeemRaPaWithCt(State storage self, uint256 amount, uint256 dsId)
        internal
        returns (uint256 accruedPa, uint256 accruedRa)
    {
        // we separate the liquidity here, that means, LP liquidation on the LV
        ↪  also triggers
        \_separateLiquidity(self, dsId);

        uint256 totalCtIssued = self.psm.poolArchive[dsId].ctAttributed;
        PsmPoolArchive storage archive = self.psm.poolArchive[dsId];

        (accruedPa, accruedRa) = \_calcRedeemAmount(amount, totalCtIssued,
        ↪  archive.raAccrued, archive.paAccrued);

@>      \_beforeCtRedeem(self, self.ds[dsId], dsId, amount, accruedPa, accruedRa);
    }
```

and in the `allowbreak _beforeCtRedeem()` function, it reduces `ctAttributed` (which is the `totalCtIssued`).

```
    function \_beforeCtRedeem(
        State storage self,
        DepegSwap storage ds,
        uint256 dsId,
        uint256 amount,
        uint256 accruedPa,
        uint256 accruedRa
    ) internal {
        ds.ctRedeemed += amount;
@>      self.psm.poolArchive[dsId].ctAttributed -= amount;
        self.psm.poolArchive[dsId].paAccrued -= accruedPa;
        self.psm.poolArchive[dsId].raAccrued -= accruedRa;
    }
```

So, no stuck amount.

0xsimao

The stuck amount is significant because all the ct the vault held will not be able to redeem the ra and pa tokens.

It is stuck because the checkpoint is taken before the vault burns its ct. So the decrease mentioned here will never fully happen because the vault has burned its ct. So ra and pa will be stuck.

The checkpoint is taken before, on `PsmLibrary.onNewIssuance()`, which calls `PsmLib::allowbreak _separateLiquidity()`, taking the checkpoint.

Then, it calls `VaultLib::onNewIssuance()`, calling `VaultLib::allowbreak _liquidatedLp()`, which calls first `PsmLib::lvRedeemRaWithCtDs()`, calling `ds.burnBothFromSelf()`, which burns the vault's ct amount.

As such, the checkpointed total ct will have a component that was burned and will never be redeemed.

davies0212

@0xsimao, you are correct that `ctAttributed` does not fully decrease. However, I disagree with the assertion that the severity is high.

To summarize, both `raAccrued` and `ctAttributed` do not fully decrease.

As a result, we cannot conclude that `ra` will always be stuck, since both `raAccrued` and `ctAttributed` remain unchanged.

Regarding `pa`, the undecreased `ct` amount depends on the `ds` amount from the `FlashSwapRouter`. If the `ds` amount from the `FlashSwapRouter` is small, the stuck amount will also be small. In fact, if the `ds` amount is zero, there will be no stuck amount at all.

Therefore, the severity of this issue should be considered medium.

0xsimao

Both ra and pa accrued will never decrease, as there is not enough ct to redeem them. There is a high chance there will be reserve so it is not a significant condition. Every time liquidity is deposited in the vault, it adds the ra amount not sent to the univ2 pool to the reserve. So unless everyone stops depositing to the vault and a lot of swaps start occuring that drain the reserve, this issue will occur.

WangSecurity

My decision remains unchanged, even though there are some constraints to trigger the issue. They're not extensive, and the issue could easily happen, leading to completely stuck tokens. Hence, the decision remains: accept the escalation and upgrade severity to high.

WangSecurity

Result: High Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- 0xsimao: accepted

Issue H-9: Attackers will steal the reserve from the `Vault` by receiving `ra` in `FlashSwapRouter::__swapDsforRa()`

Source: https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/161

Found by

0x73696d616f, KupiaSec, Smacaud, oxelmiguel

Summary

FlashSwapRouter::__swapDsforRa() is called as part of FlashSwapRouter::_swapRaforDs() whenever the reserve is sold and the resulting `Ra` is used to provide liquidity to the `Vault` by calling Vault::provideLiquidityWithFlashSwapFee().

However, if we follow the code, `FlashSwapRouter::__swapDsforRa()` calls FlashSwapRouter::__flashSwap(), which calls UniswapV2Pair::swap(). Then, the Uniswap pair calls uniswapV2Call(), which calls FlashSwapRouter::__afterFlashswapSell() and sends the `Ra` to the caller.

Thus, the vault is providing a fee, but does not use the acquired funds to fund it, but the already existing `Ra` in the contract. The `Ra` meant for the liquidity fee is sent to the user calling `FlashSwapRouter::_swapRaforDs()`.

Root Cause

In `FlashSwapRouter.sol:124`, `FlashSwapRouter::__swapDsforRa()` is called, which ultimately sends the `Ra` to the `msg.sender`.

Internal pre-conditions

None.

External pre-conditions

None.

Attack Path

1. User calls `swapRaforDs()` and ends up receiving `Ra` which was intended for the `Vault`.

Impact

Users can steal all reserve from the `Vault`.

PoC

The following code snippets show how the `Ra` ends up in the `caller`, that is, the user that calls `swapRaforDs()`.

```
function __flashSwap(...) internal {
    ...
    bytes memory data = abi.encode(reserveId, dsId, buyDs, msg.sender, extraData);

    univ2Pair.swap(amount0out, amount1out, address(this), data);
}

function uniswapV2Call(address sender, uint256 amount0, uint256 amount1, bytes
↪  calldata data) external {
    (Id reserveId, uint256 dsId, bool buyDs, address caller, uint256 extraData) =
        abi.decode(data, (Id, uint256, bool, address, uint256));
    ...
    if (buyDs) {
        ...
    } else {
        uint256 amount = amount0 == 0 ? amount1 : amount0;

        __afterFlashswapSell(self, amount, reserveId, dsId, caller, extraData);
    }
}

function __afterFlashswapSell(...) internal {
    ...
    IERC20(ra).safeTransfer(caller, raAttributed);
    ...
}
```

Mitigation

The `FlashSwapRouter::__flashSwap()` has to be modified to accept a caller argument, where it accepts either the `msg.sender` or `owner()`. In the flow of selling the reserve, it should send the `Ra` to the `owner`, which is the `Vault`.

Discussion

ziankork

This is a valid issue, though already fixed this prior to our trading competition. will provide links later

SakshamGuruji3

Escalate

According to README this should be a low ->

`AnypotentialmisbehaviourregardingDS/CT/FlashSwapmechanismthatisnotanissuerightnow butcouldposeriskinfutureintegrationsshouldbecounted,butreportedasLowissues.`

sherlock-admin3

> Escalate
>
> According to README this should be a low ->
>
> `AnypotentialmisbehaviourregardingDS/CT/FlashSwapmechanismthatisnotaniss uerightnowbutcouldposeriskinfutureintegrationsshouldbecounted,butreport edasLowissues.`

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.

0xsimao

> that is not an issue right now

This is an issue right now, it does not apply to this.

SakshamGuruji3

I think there is a confusion due to english grammar , pretty sure the sponsor meant that it is not an issue right now , let the sponsor clear on his intent by what he meant

0xsimao

There is nothing to clear, the statement could not be more obvious, they are talking about future issues, as in, 'that is not an issue right now'. Present issues are fine to report.

SakshamGuruji3

@0xsimao If you see my original escalation comment you can see the sponsor has given a thumbs - up reaction , I think it is indeed what he meant and escalation should be valid , athough it will be ideal if he can confirm with a comment

0xsimao

Does not matter, the readme is the source of truth. It's clear only future issues are out of scope. This is in the rules.

cvetanovv

@0xsimao is right here. Readme is the source of truth. There is no way to take into account that someone got the grammar wrong long after the contest is over.

Pheonix244001

@cvetanovv I think there are two ways to interpret what's written in the readme and @SakshamGuruji3 might be right here because the meaning seems to be that issues regarding flashswaprouter will only be issues in future integrations hence everything regarding those is a low, its not justified to incline over one side without confirming what the sponsor actually meant (the words in the readme can be twisted both ways)

0xsimao

@Pheonix244001 firstly, that is incorrect, the readme is very clear. It means issues that could go wrong in the future, but are correct now.

Secondly, look at the question itself:

> Should potential issues, like broken assumptions about function behavior, be reported if they could pose risks in future integrations, even if they might not be an issue in the context of the scope? If yes, can you elaborate on properties/invariants that should hold?

It is asking about issues that are not a problem right now, but could be in the future. The sponsor responds that these issues can be reported, but they are low.

The issue here is a **present** issue, so it does not apply here.

If the sponsor wanted to exclude the `FlashSwapRouter` as out of scope, that was not the section to do so.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Cork-Technology/Depeg-swap/pull/63

WangSecurity

I agree with @cvetanovv and @0xsimao and the README is clear that the issues related to DS/CT/FlashSwap mechanism should be low only if they pose issues in the future integration. Here, the issue poses the losses right now, without the Future integrations. Planning to reject the escalation and leave the issue as it is.

WangSecurity

Result: High Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- SakshamGuruji3: rejected

Issue H-10: Users redeeming early will withdraw `Ra` without decreasing the amount locked, which will lead to stolen funds when withdrawing after expiry

Source: https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/166

This issue was not included in the Fix Review and has not been fixed.

Found by

0x73696d616f, 0xNirix, KupiaSec, Matrox, dimulski, hunter_w3b, nikhil840096, oxelmiguel, vinica_boy

Summary

VaultLib::redeemEarly() is called when users redeem early via Vault::redeemEarlyLv(), which allows users to redeem `Lv` for `Ra` and pay a fee.

In the process, the `Vault` burns `Ct` and `Ds` in VaultLib::_redeemCtDsAndSellExcessCt() for `Ra`, by calling PsmLib::PsmLibrary.lvRedeemRaWithCtDs(). However, it never calls RedemptionAssetManagerLib::decLocked() to decrease the tracked locked `Ra`, but the `Ra` leaves the `Vault` for the user redeeming.

This means that when a new `Ds` is issued in the `PsmLib` or users call PsmLib::redeemWithCt(), PsmLib::_separateLiquidity() will be called and it will calculated the exchange rate to withdraw `Ra` and `Pa` as if the `Ra` amount withdrawn earlier was still there. When it calls self.psm.balances.ra.convertAllToFree(), it converts the locked amount to free and assumes these funds are available, when in reality they have been withdrawn earlier. As such, the `Ra` and `Pa` checkpoint will be incorrect and users will redeem more `Ra` than they should, such that the last users will not be able to withdraw and the first ones will profit.

Root Cause

In `PsmLib.sol:125`, `self.psm.balances.ra.decLocked(amount);` is not called.

Internal pre-conditions

None.

External pre-conditions

None.

Attack Path

1. User calls `Vault::redeemEarlyLv()` or `ModuleCore::issueNewDs()` is called by the admin.

Impact

Users withdraw more funds then they should via `PsmLib::redeemWithCt()` meaning the last users can not withdraw.

PoC

`PsmLib::lvRedeemRaWithCtDs()` does not reduce the amount of `Ra` locked.

```
function lvRedeemRaWithCtDs(State storage self, uint256 amount, uint256 dsId)
↪   internal {
    DepegSwap storage ds = self.ds[dsId];
    ds.burnBothforSelf(amount);
}
```

Mitigation

`PsmLib::lvRedeemRaWithCtDs()` must reduce the amount of `Ra` locked.

```
function lvRedeemRaWithCtDs(State storage self, uint256 amount, uint256 dsId)
↪   internal {
    self.psm.balances.ra.decLocked(amount);
    DepegSwap storage ds = self.ds[dsId];
    ds.burnBothforSelf(amount);
}
```

Discussion

ziankork

dup of #44 and related #156 . will fix

SakshamGuruji3

Escalate

It should be dupe of https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/126 , Even though this report revolves around lvRedeemRaWithCtDs, the root cause concerns incorrect state updates for redemption assets. There are reports that discuss the same incorrect state updates for ra but for different parts of the protocol, involving different functions. I believe we need to group all of these together. For example, the protocol misses exchange range accounting in multiple areas, and all of these instances are grouped in a single report (https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/119). We should do the same with this issue.

sherlock-admin3

> Escalate
>
> It should be dupe of https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/126 , Even though this report revolves around lvRedeemRaWithCtDs, the root cause concerns incorrect state updates for redemption assets. There are reports that discuss the same incorrect state updates for ra but for different parts of the protocol, involving different functions. I believe we need to group all of these together. For example, the protocol misses exchange range accounting in multiple areas, and all of these instances are grouped in a single report (https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/119). We should do the same with this issue.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

0xsimao

The fix here is different in both issues, adding `self.psm.balances.ra.decLocked(amount);` here and `self.psm.balances.ra.lockfrom(amount,buyer);` instead of `self.psm.balances.ra.lockUnchecked(amount,buyer);` in the other.

#119 uses the same fix for everything, fetching the exchange rate and multiplying/dividing by it.

SakshamGuruji3

But in the end it falls under the same thing ”locked/Unlocked accounting” just like multiplying/dividing the exchange rate , isnt that the same thing

0xsimao

There are multiple ways to do incorrect ra tracking but not taking the exchange rate into account is always the same bug.

KupiaSecAdmin

I agree with @SakshamGuruji3.

Issues #126, #155, and #166 (this one) all share the same root cause. They should be consolidated into a single issue.

0xsimao

”Wrong accounting” is not a root cause. The fixes are all different. Both in the code and conceptually.

AtanasDimulski

Agree with @0xsimao wrong accounting can't be considered as the same root cause. Moreover, if you don't agree with how #119 and its dups are grouped you should have escalated each one of them. However not taking the exchange rate into account is the same as lack of slippage protection.

SakshamGuruji3

The argument is still not convincing , first of all 119 has been reported correctly we can not submit 5 vulnerabilities separately for the same root cause (incorrect exchange rate accounting) , secondly the root cause is here the same ”not accounting for locked/unlocked RA correctly” and should be grouped together

0xsimao

The fixes are all different, in the code and conceptually.

https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/126 calls the wrong function, lockUnchecked() instead of lockFrom(). https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/155 receives an amount of ra and increases the ra locked by this full amount, without discounting the fee. https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/166 it's missing a call to self.psm.balances.ra.decLocked(amount); when redeeming ra.

The exchange rate issues are all missing taking into account the exchange rate and the fix is the exact same for all.

These 3 mentioned issues have different fixes because we may need to add more or less ra depending on the function (the wrong amount is not always the same) whereas taking into account the exchange rate is a fixed amount and always the same fix.

WangSecurity

From my understanding, reports #126 and #155 lead to a similar impact, but the underlying reason and root causes are different in all situations. I also agree with @0xsimao and @AtanasDimulski that `wrongaccounting` is not a correct root cause and is too broad. Hence, all the issues have to remain separate, because while the reports lead to similar impacts, they happen due to different root causes, planning to reject the escalation and leave the issue as it is.

WangSecurity

Result: High Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- SakshamGuruji3: rejected

Issue H-11: `VaultPoolLib::reserve()` will store the `Pa` not attributed to user withdrawals incorrectly and leave in untracked once it expires again

Source: https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/191

The protocol has acknowledged this issue.

Found by

0x73696d616f, 0xNirix, dimulski, sakshamguruji

Summary

VaultPoolLib::reserve() stores the `Pa` attributed to withdrawals in self.withdrawalPool.stagnatedPaBalance instead of storing the amount `attributedToAmm`. Additionally, this amount of `Pa`, the one attributed to the `Amm` is never dealt with and leads to stuck `PA`.

The comment in the code mentions

```
// FIXME : this is only temporary, for now
// we trate PA the same as RA, thus we also separate PA
// the difference is the PA here isn't being used as anything
// and for now will just sit there until rationed again at next expiry.
```

But it is incorrect as it is never rationed again, just forgotten. The VaultPoolLib::rationedToAmm() function only uses the `Ra` balance, not the `Pa`, which is effectively left untracked.

Root Cause

In `VaultPoolLib:170`, the leftover non attributed `Pa` is not dealt with.

Internal pre-conditions

None.

External pre-conditions

None.

Attack Path

1. `VaultPoolLib::reserve()` is called when liquidating the lp position of the `Vault` via `VaultLib::_liquidatedLP()`, triggered by users when redeeming expired liquidity vault shares or on the admin trigerring a new issuance.

Impact

The `Pa` in the `Vault` is stuck.

PoC

`VaultPoolLib::rationedToAmm()` does not deal with the `Pa`.

```
function rationedToAmm(VaultPool storage self, uint256 ratio) internal view returns
↪  (uint256 ra, uint256 ct) {
    uint256 amount = self.ammLiquidityPool.balance;

    (ra, ct) = MathHelper.calculateProvideLiquidityAmountBasedOnCtPrice(amount,
↪  ratio);
}
```

Mitigation

Distributed the `Pa` to users based on their `LV` shares or redeem the `Pa` for `Ra` and add liquidity to the new issued `Ds` or similar.

Discussion

ziankork

valid. will fix

ziankork

This is originally has a feature planned for that, but we're still working on it and the feature still needs some time for us to solidify all aspect of it. that's why we won't fix this. Since the feature are considered non-trivial

Issue M-1: The UUPS proxie standard is implemented incorrectly, making the protocol not upgradeable

Source: https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/47

The protocol has acknowledged this issue.

Found by

dimulski

Summary

Both the `AssetFactory.sol` and `FlashSwapRouter.sol` contracts inherit the UUPSUpgradeable contract from Openzepelin, indicating that the devs of the protocol want to have the possibility of upgrading the above mentioned contracts at some point in the future. Both of the contracts also implement the OwnableUpgradeable contract, and the *_authorizeUpgrade()* function in both contracts has the **onlyOwner** modifer. This function is used to check whether the person who tries to update the implementation contract in the Proxy has the required access. However in both contracts the initialize function sets the owner of the contract to the `ModuleCore.sol` contract as can be seen in the AssetFactory::initialize() and FlashSwapRouter::initialize() functions. The function from the UUPSUpgradeable contract that is used to upgrade the implementation contract in the proxy is the *upgradeToAndCall()* function, however this function can't be called from the `ModuleCore.sol` contract, as the functionality is not implemented. The `ModuleCore.sol` contract is the owner of both the contracts, and there isn't any functionality to transfer the ownership either. Thus `AssetFactory.sol` and `FlashSwapRouter.sol` contracts are effectively not upgradable, breaking a very important functionality of the protocol.

Root Cause

The *upgradeToAndCall()* function can't be called from the `ModuleCore.sol` contract and upgrade the `AssetFactory.sol` and `FlashSwapRouter.sol` contracts.

Internal pre-conditions

1. All the contracts in the protocol are deployed
2. The protocol team decides the want to update the `AssetFactory.sol` and/or the `FlashSwapRouter.sol` contracts

External pre-conditions

No response

Attack Path

No response

Impact

Contracts that are expected to be upgradable, can't be upgraded due to missing functionality in the `ModuleCore.sol` contract.

PoC

No response

Mitigation

Implement a call to the *upgradeToAndCall()* function in the `ModuleCore.sol` contract

Discussion

AtanasDimulski

Escalate, I don't agree that this is a dup of #185, as It describes a completely different way that will prevent contracts from being upgraded. Missign the upgrade upgradeToAndCall() function. Even if the issue described in #185 is fixed, the contracts won't be upgradeable.

sherlock-admin4

> Escalate, I don't agree that this is a dup of #185, as It describes a completely different way that will prevent contracts from being upgraded. Missign the upgrade upgradeToAndCall() function. Even if the issue described in #185 is fixed, the contracts won't be upgradeable.

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.

0xsimao

The lead judge grouped them because they fall under the umbrella of upgradeable issues. The comment here also has a point.

WangSecurity

I see how they can be groupped by the issue of incorrect upgradeability, but I agree with the escalation that this issue should be de-duplicated as it shows a completely different problem and ”upgradeability issues” are too broad of a root cause. Planning to accept the escalation and de-dup this issue with medium severity.

@cvetanovv @AtanasDimulski are there other duplicates?

AtanasDimulski

Hey @WangSecurity all issues that are grouped under #185 and submitted by MadSisyphus talk about the notDelegated modifier, as far as I can see there are no other dups.

WangSecurity

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- AtanasDimulski: accepted

ziankork

This issue is valid and have been fixed, but, we don't intend to make those contract upgradeable on mainnet(due to legal reasons) so we plan to remove the upgradeability of the contract later. The reason we provide a way to upgrade was initially due to the trading competition.

conclusion : valid, but won't fix

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Cork-Technology/Depeg-swap/pull/69/commits/2325f6faf12467fc1e899ef7e95269d6f617b8b7

Issue M-2: Admin will not be able to only pause deposits in the `Vault` due to incorrect check leading to DoSed withdrawals

Source: https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/182

Found by

0x73696d616f, 404Notfound, 4gontuk, Abhan1041, KupiaSec, Pro_King, Smacaud, Trooper, durov, hunter_w3b, korok, mladenov, octeezy, ravikiran.web3, tinnohofficial, tmotfl, ydlee, yovchev_yoan

Summary

The modifier LVDepositNotPaused in Vault::depositLv() checks states[id].vault.config.isWithdrawalPaused instead of states[id].vault.config.isDepositPaused, which means deposits will only be paused if withdrawals are paused, DoSing withdrawals.

Root Cause

In `ModuleState:109`, it checks `states[id].vault.config.isWithdrawalPaused` when it should check `states[id].vault.config.isDepositPaused`.

Internal pre-conditions

1. Admin pauses deposits.

External pre-conditions

None.

Attack Path

1. Admin sets deposits paused, but deposits are not actually paused due to the incorrect modifier.
2. Admin either leaves deposits unpaused or pauses both deposits and withdrawals, DoSing withdrawals.

Impact

Admin is not able to pause deposits alone which would lead to loss of funds as this is an emergency mechanism. If the admin wants to pause deposits, withdrawals would also have to be paused, DoSing withdrawals.

PoC

`ModuleState::LVDepositNotPaused()` is incorrect:

```
modifier LVDepositNotPaused(Id id) {
    if (states[id].vault.config.isWithdrawalPaused) { //@audit isDepositPaused
        revert LVDepositPaused();
    }
    _;
}
```

Mitigation

`ModuleState::LVDepositNotPaused()` should be:

```
modifier LVDepositNotPaused(Id id) {
    if (states[id].vault.config.isDepositPaused) {
        revert LVDepositPaused();
    }
    _;
}
```

Discussion

ziankork

will fix

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Cork-Technology/Depeg-swap/pull/78

Issue M-3: Admin will not be able to upgrade the smart contracts, breaking core functionality and rendering the upgradeable contracts useless

Source: https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/185

Found by

0x73696d616f, MadSisyphus, dimulski

Summary

The AssetFactory and FlashSwapRouter inherit the `UUPSUpgradeable` contract in order to be upgradeable. However, AssetFactory::initialize(), FlashSwapRouter::initialize(), AssetFactory::_authorizeUpgrade() and FlashSwapRouter::_authorizeUpgrade() have the `notDelegated`, which means they can not be called in the context of a proxy, hence they can not be upgradeable.

This renders the inherited `UUPSUpgradeable` useless and the 2 contracts will not be upgradeable. Additionally, the AssetFactory and FlashSwapRouter contracts are not deployed behind proxies, meaning that this problem would be noticed when trying to upgrade and failing.

Root Cause

In `AssetFactory.sol:48`, `AssetFactory.sol:195`, `FlashSwapRouter.sol:32` and `FlashSwapRouter.sol:41` the `notDelegated` modifiers are used.

Internal pre-conditions

None.

External pre-conditions

None.

Attack Path

Admin tries to upgrade the `AssetFactory` and `FlashSwapRouter` contracts but fails.

Impact

The `UUPSUpgradeable` contract is rendered useless, which means the `AssetFactory` and `FlashSwapRouter` contracts can not be upgraded. This leads to breaking major functionality as well as the possibility of stuck/lost funds.

PoC

`FlashSwapRouter`

```
contract RouterState is IDsFlashSwapUtility, IDsFlashSwapCore, OwnableUpgradeable,
↪  UUPSUpgradeable, IUniswapV2Callee {
    ...
    function initialize(address moduleCore, address _univ2Router) external
↪  initializer notDelegated {
        __Ownable_init(moduleCore);
        __UUPSUpgradeable_init();

        univ2Router = IUniswapV2Router02(_univ2Router);
    }
    ...
    function _authorizeUpgrade(address newImplementation) internal override
↪  onlyOwner notDelegated {}
}
```

`AssetFactory`

```
contract AssetFactory is IAssetFactory, OwnableUpgradeable, UUPSUpgradeable {
    ...
    function initialize(address moduleCore) external initializer notDelegated {
        __Ownable_init(moduleCore);
        __UUPSUpgradeable_init();
    }
    ...
    function _authorizeUpgrade(address newImplementation) internal override
↪  onlyOwner notDelegated {}
}
```

Mitigation

Remove the `notDelegated` modifiers.

Discussion

ziankork

yes this is valid, we already removed this prior to our trading competition. will provide links later

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Cork-Technology/Depeg-swap/pull/60

Issue M-4: Attacker Can Decide The Initialization Ratio Of The AMM Pair

Source: https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/186

The protocol has acknowledged this issue.

Found by

0xNirix, sakshamguruji, vinica_boy

Summary

When the RA/CT is deposited in the AMM pool it must follow a pre defined ratio , but the attacker can see the pair has been deployed on uniV2 and make the first deposit (adding liquidity) and distrupt the intended ratio of the CT/RA .

Vulnerability Detail

1.) When depositing , the vault decides the ratio of the RA/CT to be deposited in the AMM pool ->

https://github.com/sherlock-audit/2024-08-cork-protocol/blob/main/Depeg-swap/contracts/libraries/VaultLib.sol#L204

```
function __provideLiquidityWithRatio(
       State storage self,
       uint256 amount,
       IDsFlashSwapCore flashSwapRouter,
       address ctAddress,
       IUniswapV2Router02 ammRouter
   ) internal returns (uint256 ra, uint256 ct) {
       uint256 dsId = self.globalAssetIdx;

       uint256 ctRatio = __getAmmCtPriceRatio(self, flashSwapRouter, dsId);

       (ra, ct) = MathHelper.calculateProvideLiquidityAmountBasedOnCtPrice(amount,
↪  ctRatio);

       __provideLiquidity(self, ra, ct, flashSwapRouter, ctAddress, ammRouter,
↪  dsId);
   }
```

```
function __getAmmCtPriceRatio(State storage self, IDsFlashSwapCore flashSwapRouter,
↪  uint256 dsId)
```

```
        internal
        view
        returns (uint256 ratio)
    {
        // This basically means that if the reserve is empty, then we use the
↪   default ratio supplied at deployment
        ratio = self.ds[dsId].exchangeRate() - self.vault.initialDsPrice;

        // will always fail for the first deposit
        try flashSwapRouter.getCurrentPriceRatio(self.info.toId(), dsId) returns
↪   (uint256, uint256 _ctRatio) {
            ratio = _ctRatio;
        } catch {}
    }
```

If the pool is empty (it's the first liquidity addition) the ratio should follow the default ratio decided at development.

2.) But an attacker can frontrun this first deposit in the vault which adds liquidity to the AMM , and manually call _addLiquidity in the UniV2 pool which calls->

https://github.com/Uniswap/v2-periphery/blob/master/contracts/UniswapV2Router02.sol#L33

and since the pool is empty the attacker can decide the initial ratio of the pool and distrupt the ratio (can be as extreme as possible) and the amount of subsequent deposits of CT/RA in the AMM would follow this initial ratio.

Impact

The attacker can decide the initial ratio of the AMM pool and distrupt subsequent CT/RA deposits in the pool and incorrect DS mints, offload that initialization on the config contract so that only the config contract owner can initialize the vault.

Code Snippet

https://github.com/sherlock-audit/2024-08-cork-protocol/blob/main/Depeg-swap/contracts/libraries/VaultLib.sol#L204

Tool Used

Manual Review

Recommendation

offload that initialization on the config contract so that only the config contract owner can initialize the vault.

Discussion

ziankork

This is valid. will fix

cvetanovv

Actually, this is invalid. The condition for this to happen is for the pool to be empty. But it never happens because the pool is only created when the new DS tokens are issued.

cvetanovv

Escalate

I escalate on behalf of @0xNirix

sherlock-admin3

> Escalate
>
> I escalate on behalf of @0xNirix

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.

0xNirix

https://github.com/sherlock-audit/2024-08-cork-protocol/blob/db23bf67e45781b00ee6de5f6f23e621af16bd7e/Depeg-swap/contracts/libraries/VaultLib.sol#L92-L109

On first issuance of DS token for a RA/PA, the protocol creates the pair but does not add liquidity as it returns due to `if(prevDsId==0)` condition. This means the pool is empty and open for the initialization attack by a third party.

```
function onNewIssuance(
    State storage self,
    uint256 prevDsId,
    IDsFlashSwapCore flashSwapRouter,
    IUniswapV2Router02 ammRouter
) external {
    // do nothing at first issuance
    if (prevDsId == 0) {
        return;
    }


    if (!self.vault.lpLiquidated.get(prevDsId)) {
        \_liquidatedLp(self, prevDsId, ammRouter, flashSwapRouter);
    }
```

```
    \_\_provideAmmLiquidityFromPool(self, flashSwapRouter,
↪   self.ds[self.globalAssetIdx].ct, ammRouter); //@audit liquidity is only added
↪   if this is not the first issuance
}
```

Therefore the following attack is indeed possible:

1. Immediately after first DS issuance of a new RA/PA, Attacker calls deposit function in the PSM, providing RA.
2. Attacker receives equivalent amounts of CT and DS based on the correct exchangeRate.
3. Attacker immediately calls the AMM's addLiquidity function, providing CT and RA at an arbitrary ratio, setting an incorrect initial price.
4. Attacker deposits RA into the Liquidity Vault (LV) and/or waits for other users to deposit RA into the LV.
5. The protocol adds liquidity to the AMM using the LV deposit, but at the incorrect price set by the attacker.
6. Attacker arbitrages between the PSM (at the correct rate) and the AMM (at the incorrect rate), extracting value from the system.

The first issuance condition is mentioned in the issue https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/169 which is duped to this issue.

0xsimao

This issue is invalid because on the first issuance, the admin may add liquidity itself right after deploying if this is a concern.

0xNirix

Admin will have to perform multiple steps (get CT, add liquidity with CT/ RA) and can still be front-run by an attacker.

0xsimao

Yes but according to the admin rule that is irrelevant.

0xNirix

I thought admin rule says admin is trusted to make the right calls/decisions, but here due to a code deficiency admin cannot guarantee prevention of this attack.

spdimov

With the current code implementation, it is clear that the pool is expected to be empty when the first deposit occurs. I do not think that the admin rule is relevant here.

0xsimao

> can still be front-run by an attacker.

He can send the 2 transactions atomically with a smart contract or use flashbots via private rpc. Issuing new ds tokens (which create the pools) is admin permissioned, so he can create a new issuance and deploy liquidity without attacker interference using the 2 methods mentioned above.

The expected outcome is the initial price to be the one set by the admin. To ensure this happens the admin has to make a small deposit, which he will as he is trusted.

0xNirix

The expected outcome is mentioned in the code - the source of truth.

```
// do nothing at first issuance
if (prevDsId == 0) {
    return;
}
```

Hence this issue is valid.

0xsimao

That is in the context of that function, but there is more outside of it.

WangSecurity

I disagree the admin rules can be applied here. The admin has to just deploy the contracts and there are no mentions if the admin will use bundled transactions, flashbots or fund the pool themselves. Hence, it's completely viable way to only deploy a pool without any additional transactions and wouldn't be a mistake on the admin's end.

But, another question I've got is, is there a way to mitigate this issue in real-time, i.e. can the protocol just set a different pool for the same tokens or anything else?

0xsimao

> But, another question I've got is, is there a way to mitigate this issue in real-time, i.e. can the protocol just set a different pool for the same tokens or anything else?

No, it would have to be redeployed, as it maps to a certain ra,pa pair. The contract has no funds yet so this would not be a problem anyway.

Additionally, if the attacker sets a different uni v2 price, users could arbitrage on this pool by depositing and redeeming in the PSM, which uses the correct exchange rate. So the price would naturally go into the correct one due to arbitrage. Assume attacker sets pool at 2 ra for each 1 ct, when it should be 1:1. Deposit in the psm gives 1 ct and ds for each 1 ra. Redeem takes 1 ct, 1 ds and gives 1 ra. So users would deposit 1 ra in the psm to get 1 ds + 1 ct, and trade this ct in the pool, getting more ra, and repeating, until the pool is balanced. If the attacker sets 2 ct for each 1 ra, users can get ct cheap, pair with ds in the market and redeem for ra at a better ratio.

spdimov

The attacker does not need to update the price right after the pool was deployed, it is need to be done just before the first deposit.

I have provided the following scenario in #168

1. New issuance has occured with exchange rate between RA:CT+DS = 1.
2. Attacker mints 10 CT + 10 DS by providing 10 RA
3. Another user decide to deposit 50 RA to the LV
4. It is expected that liquidty sent to the AMM would be ~50% RA ~ 50% CT
5. Attacker frontrun the deposit transaction and deposits 1 RA and 0.1 CT
6. Now the price ration between RA:CT would be much higher than expected.
7. Only small amount of the user's 50 RA would be used to mint CT and almost all of it(RA) are going to be provided to the pair
8. New pair reserves are going to be ~45 RA and 4.5 CT
9. Now attacker can provide his CT tokens in order to get more RA than initially invested.

As it can be seen redeployment does not mitigates the issue.

0xsimao

> The attacker does not need to update the price right after the pool was deployed, it is need to be done just before the first deposit.

This boils down to the admin allowing users depositing in the vault, which deposits into a uni v2 pool with 0 liquidity, which is vulnerable to such attacks - lack of liquidity. This attack is not a problem of the initial price, but most importantly a low liquidity issue, as it can be performed whenever there is low liquidity. Thus it constitutes a systemic risk of using a uni v2 pool. As such, the admin is responsible for preventing it, as it is a general risk of the protocol. It can never be mitigated unless the admin ensures a minimum liquidity in some way.

Even if the pool is enforced to have an initial price of the exchange rate, nothing guarantees that the liquidity reserves stay high. If they decrease, the attack can be performed again. Hence, the only way to guarantee this never happens is not to enforce the initial price, but to ensure the pool has enough liquidity, and this is the job of the admin.

spdimov

As @WangSecurity stated in previous comments, admin rule is not applicable here. We have a code which is designed to expect that the pool will be empty when first deposit happens and potential attacker can take advantage of it.

0xsimao

Except when it is a systemic risk, which is lack of liquidity, as explained here. That is, this is not really fixable without admin intervention. In fact, the root cause of this attack

path is not setting slippage protection. And as explained here, the initial price being different is not a problem.

siddhpurakaran

Can you provide more details about mitigation of this issue

WangSecurity

What I don't understand for now is who loses here. I agree with @0xsimao that this path is about slippage protection and not the issue we discuss here. This issue indeed opens up an arbitrage opportunity, but that's it. The report doesn't show who would lose in this scenario or how the protocol is rendered useless (the price can be arbitraged to the correct one). Hence, this has to remain invalid, planning to reject the escalation.

0xNirix

@WangSecurity, you are absolutely correct that the main issue in this finding is not about slippage protection. The protocol's reliance on the AMM price of RA/CT, which it expects to be fair due to market dynamics, is generally acceptable.

However, this vulnerability allows the initial liquidity provider to set an incorrect price deliberately and maliciously. The protocol suffers losses because it will rely on this incorrect price to provide liquidity, effectively selling a token (RA) at less than fair market price, which the attacker can exploit.

Let's examine the evidence that the protocol fails to add liquidity at the intended price in the code:

1. It is on the first issuance that the protocol fails to provide the initial liquidity (that we have already agreed in this thread but just relisting)

```
// MUST be called on every new DS issuance
    function onNewIssuance(
        State storage self,
        uint256 prevDsId,
        IDsFlashSwapCore flashSwapRouter,
        IUniswapV2Router02 ammRouter
    ) external {
        // do nothing at first issuance
        if (prevDsId == 0) {
            return;
        }

        if (!self.vault.lpLiquidated.get(prevDsId)) {
            \_liquidatedLp(self, prevDsId, ammRouter, flashSwapRouter);
        }

        \_\_provideAmmLiquidityFromPool(self, flashSwapRouter,
↪   self.ds[self.globalAssetIdx].ct, ammRouter); //@audit in all other cases the
↪   new issuance adds the liquidity
    }
```

3. Whenever protocol does provide liquidity, the protocol either relies on the real-time ratio or uses a fixed ratio:

```
function \_\_provideAmmLiquidityFromPool(
       State storage self,
       IDsFlashSwapCore flashSwapRouter,
       address ctAddress,
       IUniswapV2Router02 ammRouter
   ) internal {
       uint256 dsId = self.globalAssetIdx;

       uint256 ctRatio = \_\_getAmmCtPriceRatio(self, flashSwapRouter, dsId);

       (uint256 ra, uint256 ct) = self.vault.pool.rationedToAmm(ctRatio);

       \_\_provideLiquidity(self, ra, ct, flashSwapRouter, ctAddress, ammRouter,
↪  dsId);

       self.vault.pool.resetAmmPool();
   }
```

In the __getAmmCtPriceRatio method

```
function \_\_getAmmCtPriceRatio(State storage self, IDsFlashSwapCore
↪  flashSwapRouter, uint256 dsId)
    internal
    view
    returns (uint256 ratio)
{
    // This basically means that if the reserve is empty, then we use the default
↪  ratio supplied at deployment
    ratio = self.ds[dsId].exchangeRate() - self.vault.initialDsPrice;

    // will always fail for the first deposit
    try flashSwapRouter.getCurrentPriceRatio(self.info.toId(), dsId) returns
↪  (uint256, uint256 \_ctRatio) {
        ratio = \_ctRatio;
    } catch {}
}
```

In the code, we can clearly see that the intention is to use an initial default ratio if the AMM cannot be used to determine the ratio. However, there is a failure to enforce this for the first issuance.

The consequences of this vulnerability are:

1. An attacker can set an artificially low price for RA.

2. The protocol will then use this incorrect price to provide liquidity, effectively selling RA at below market value.
3. The attacker can exploit this mispricing for immediate profit.

While it's true that arbitrage will eventually correct the price, the protocol will have already suffered losses by selling assets at an unfair price. This is not a normal market dynamic but a direct result of the protocol's failure to secure the initial liquidity provision at the intended price.

0xsimao

> The protocol will then use this incorrect price to provide liquidity, effectively selling RA at below market value.

This is a slippage protection issue, same attack path as described here.

SakshamGuruji3

This should not be under the same slippage issue since the root cause is different , see here for reference how this issue was not duped with other https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/166 specially the reasoning by HoJ

0xsimao

The root cause is the provide liquidity function not having slippage. This is just an attack path that allows the ratio of the pool to be more easily manipulated, but not the root cause.

0xNirix

Will try to reiterate - not a slippage protection issue.

As detailed in the comment https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/186#issuecomment-2401372760 with code evidence: The protocol intended to a) set a fixed ratio initially OR b) use AMM ratio for providing liquidity at any point of time after that. Whether it provides these liquidity with or without slippage protection is irrelevant.

The protocol fails to achieve point a) causing loss to the protocol. The initial price can be set by an attacker and protocol will just use that incorrect AMM ratio as expected price. Again whether it uses a slippage against that expected price is irrelevant.

0xsimao

> The protocol intended to a) set a fixed ratio initially OR b) use AMM ratio for providing liquidity at any point of time after that.

No, the protocol uses the fixed ratio if the pool has no liquidity. And as explained, having a different initial price does not lead to medium impact by itself alone.

WangSecurity

Yeah, that's what I think as well, I see how it allows the attacker to profit, but I don't see how it causes losses to other users. Hence, I need a POC (either coded or written) to

prove the loss of funds. If no POC is provided, planning reject the escalation and leave the issue as it is.

0xNirix

umm...if someone is profiting, that should inherently mean there is an equivalent loss.

Here is the written POC, highlighting one of the ways this loss will manifest. Please note it is a complex protocol and loss can manifest in few different ways. Also some of the numbers are approximate but should be close enough

1. Attacker's actions:
    - Provides initial liquidity to AMM at manipulated price: 1,000 RA : 100 CT (10 RA = 1 CT)
2. Protocol's actions:
    - A large deposit comes to LV for 1,000,000 RA.
    - Adds liquidity based on manipulated price: 1,000,000 RA : 100,000 CT
3. AMM state after protocol liquidity addition:
    - Pool: 1,000,000 RA : 100,000 CT
4. Attacker's arbitrage:
    - Swaps 200,000 CT for 700,000 RA in AMM (price impact considered)
5. Final AMM state (back to normal price):
    - Pool: 300,000 RA : 300,000 CT

Impact

1. Massive Liquidity Drain:
    - Initial liquidity: 1,000,000 RA : 100,000 CT
    - Final liquidity: 300,000 RA : 300,000 CT
    - Loss: ~70% of RA liquidity
2. LP Token Devaluation:
    - LP tokens now represent a share of a much smaller pool
    - Value per LP token decreased by approximately 70%

Impact translated to LV Holders (other users):

1. Liquidity Liquidation: When DS expires, the `allowbreak _liquidatedLp` function is called:
    - AMM liquidation yields only 300,000 RA (instead of 1,000,000 RA)
2. Reserve Calculation: The `reserve` function in `VaultPool` is called with drastically reduced amounts:

- Total LV issued: ~1,000,000 LV (unchanged)
- Available RA: ~300,000 (from AMM)

3. Rate Calculation per LV:
    - Before attack: ~1 RA per LV (1,000,000 RA for 1,000,000 LV)
    - After attack: ~0.3 RA per LV (300,000 RA for 1,000,000 LV)
4. Impact on Redemptions: When users redeem their LV tokens:
    - For every 1,000 LV redeemed, users receive approximately:
        - 300 RA (instead of 1,000 RA)
    - This represents a 70% loss in value compared to the original deposit

This vulnerability allows an attacker to manipulate the AMM price, forcing the protocol to provide liquidity at an unfavorable rate. The subsequent arbitrage drains significant RA from the AMM pool.

The impact is severe with:

1. Direct value extraction by the attacker
2. Signifcant devaluation of LV tokens, causing loss for all LV holders.

0xsimao

This is the slippage issue.

0xNirix

Please refer to this comment https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/186#issuecomment-2404534568 to understand why this is not a slippage issue.

0xsimao

The user is depositing to the vault without control over the amount of ra and ct it will get, which means it is a slippage issue. By placing a slippage check this issue disappears. Even if you set the initial ratio of the amm, an attacker can always manipulate the price of the pool.

WangSecurity

@0xsimao by saying this is a slippage issue, do you mean that the problem here is users depositing at this skewed rate (10 RA : 1 CT)?

0xsimao

Yes

WangSecurity

Hmm, but I actually see how the problem here is not just no slippage protection. Even if such a situation happens, and the user deposits 1M Ra and 100k CT with the 10:1 exchange rate, they firstly don't lose anything in value after the deposit, and after

withdrawing (before the swap), they don't lose anything. So, it's not about no slippage protection on the deposit cause adding it wouldn't fix anything.

About the POC:

1. The following numbers look arbitrary (i.e. there's no explanation how you've received those numbers so it looks like just randomly picked numbers. I'm sorry if that's not the case, but it's hard to say, since there's no explanation how we received them):

   Attacker's arbitrage: Swaps 200,000 CT for 700,000 RA in AMM (price impact considered) Final AMM state (back to normal price): Pool: 300,000 RA : 300,000 CT

It may be that the price wouldn't be 300k RA: 300k CT or that the attacker would've got 700k RA.

2. This situation could've been arbitraged easily before anyone deposited into the pool. The attacker deposited 1000 RA and 100 CT, so anyone seeing they can get 10 RA for 1 CT would instantly do that.
3. The depositor can refrain from depositing into the pool if they see an exchange rate like that and wait for the price to stabilise.

Hence, my decision to reject the escalation and leave the issue as it is remains.

0xsimao

It is a slippage issue because the user should not deposit if the exchange rate is so bad (10:1), unless it is frontrunned and the exchange rate gets worse. If a user deposits with an exchange rate 10:1, most of the funds will be arbitraged away. The fix is the user setting slippage so the price stays near the exchange rate.

0xNirix

@WangSecurity response to your queries regarding POC:

> The following numbers look arbitrary (i.e. there's no explanation how you've received those numbers so it looks like just randomly picked numbers. I'm sorry if that's not the case, but it's hard to say, since there's no explanation how we received them): Attacker's arbitrage: Swaps 200,000 CT for 700,000 RA in AMM (price impact considered) Final AMM state (back to normal price): Pool: 300,000 RA : 300,000 CT It may be that the price wouldn't be 300k RA: 300k CT or that the attacker would've got 700k RA.

Yes, like I mentioned these are approximate but roughly correct numbers but omitted some details for brevity. To arrive at above number I used uniswap v2 swap formula. Here is the breakdown:

1. Initial state:
   - RA (Reserve A) = 1,000,000
   - CT (Reserve B) = 100,000

2. Constant product (k): k = RA * CT = 1,000,000 * 100,000 = 100,000,000,000
3. The swap: User wants to swap 200,000 CT for RA
4. New CT balance after swap: CT_new = 100,000 + 200,000 = 300,000
5. Calculate new RA balance using the constant product formula: k = RA_new * CT_new 100,000,000,000 = RA_new * 300,000 RA_new = 100,000,000,000 / 300,000 = 333,333.33 (rounded to 2 decimal places)
6. Amount of RA given to the user: RA_out = 1,000,000 - 333,333.33 = 666,666.67
7. Final pool state: RA remaining = 333,333.33 CT remaining = 300,000

   This situation could've been arbitraged easily before anyone deposited into the pool. The attacker deposited 1000 RA and 100 CT, so anyone seeing they can get 10 RA for 1 CT would instantly do that.

Arbitrage is only expected if it would profit the arbitrageurs. The attacker can set the price by using a very small amount e.g. .01 RA and 0.001 CT and we should not expect someone to be able to make any profit off that.

> The depositor can refrain from depositing into the pool if they see an exchange rate like that and wait for the price to stabilise.

The protocol deposits to pool automatically when a user deposits to its liquidity vault (LV). For this LV vault, the user is concerned only about the RA to LV token exchange rate. The price of RA to LV is determined by a separate exchange rate which is refreshed periodically at protocol determined expiration time. So user will not know about this liquidity drain impact till the expiration time (which can be anything e.g. after several days). This LV functionality where user is depositing RA, has pool deposits as an internal mechanism.

WangSecurity

> It is a slippage issue because the user should not deposit if the exchange rate is so bad (10:1), unless it is frontrunned and the exchange rate gets worse. If a user deposits with an exchange rate 10:1, most of the funds will be arbitraged away. The fix is the user setting slippage so the price stays near the exchange rate

But, the depositor doesn't lose in value just by depositing. Their value before and after the deposit is the same, and they didn't lose anything. Even if there's slippage protection, they deposit without any loss, and a swap to ”drain liquidity” is made after the deposit has occurred, and the slippage check wouldn't catch it.

About the calculations, thank you very much for elaborating, I appreciate it. But wouldn't this be mitigated in a way expressed here. Even if the RA/CT exchange rate is 10, the users still get 1 CT + 1 DS for 1 RA and can arbitrage the skewed exchange rate themselves, isn't it correct?

0xsimao

@WangSecurity the slippage catches it because if the depositor specifies a min or max exchange rate, depositing with a bad exchange rate will never go through.

Providing liquidity to pools also has slippage checks.

0xNirix

Correct @WangSecurity, anyone can arbitrage the exchange rate back, however that should not be considered as mitigation because:

1. As mentioned in the previous comment, it should not be expected that users will arbitrage the rate back unless it is to their profit. Arbitrage is only expected if it would profit the arbitrageurs. The attacker can set the price by using a very small amount e.g. .01 RA and 0.001 CT and we should not expect someone to be able to make any profit off that.
2. The users at loss specifically here are LV depositors, they are looking to earn yield on their RA tokens, they should not be expected to know about an internal RA:CT pool that LV internally deposits into as the pool's current rate does not matter to them in anyway during the deposit.

cvetanovv

I think this issue is valid, and escalation can be accepted.

I had originally left this issue valid, and the only reason I invalidated it is that the condition for this to happen is for the pool to be empty.

But @0xNirix showed early in the discussion that the pool will always be empty. We even got a comment from the developers that nothing would be done after that:

```
// do nothing at first issuance
if (prevDsId == 0) {
    return;
}
```

Arguments that the admin can prevent this attack are invalid.

A user can front-run his transaction and do the attack before the second function call. And what would be the logic of developers putting `return` once they want to call the function again to add liquidity? There is no logic in this, and it confirms that the pool will be empty, and vulnerability is possible.

Also, the arguments that it is a duplicate of the slippage group are invalid. You can see the duplication rules:
https://docs.sherlock.xyz/audits/real-time-judging/judging#ix.-duplication-rules

Even if the root-cause is the same, the attack path is different: `Onlywhenthe"potentialduplicate"meetsallfourrequirementswillthe"potentialduplicate"beduplicatedwiththe"targetissue"`

WangSecurity

I agree with @cvetanovv, and I believe he explained very well why this issue has to be valid. Planning to accept the escalation and validate the family with medium severity. Plan to apply this decision in a couple of hours.

0xsimao

@WangSecurity what do you think of this? It is clearly a slippage issue

WangSecurity

As I've said a couple of times earlier, this is not a slippage-related issue, and I stand by my previous arguments about this. The decision remains as in my previous message: accept the escalation and validate with medium severity. Planning to apply this decision in a couple of hours.

0xsimao

The issue only happens because someone frontruns the user and sets a different exchange rate. If the user deposits with a bad exchange rate, it's their fault.

It's exactly like providing liquidity to a unsiwap pool, users also set minimum amounts A and B, so they can control the ratio.

If slippage is added then users always deposit with a favorable exchange rate and this is issue is fixed. And slippage is mentioned directly in the rules for groups and the issue can be fixed by adding slippage protection

> Front-running/sandwich/slippage protection:
>
> Can be fixed by slippage protection;

WangSecurity

> The issue only happens because someone frontruns the user and sets a different exchange rate

Front-running is not required here.

> If the user deposits with a bad exchange rate, it's their fault.

I believe the point 2 in this comment explains well, why it's not relevant here.

> It's exactly like providing liquidity to a unsiwap pool, users also set minimum amounts A and B, so they can control the ratio. If slippage is added then users always deposit with a favorable exchange rate and this is issue is fixed. And slippage is mentioned directly in the rules for groups and the issue can be fixed by adding slippage protection

What matters here is the RA to LV exchange rate, and it would be correct. So if we implement slippage protection for this, it won't fix the issue.

Moreover, even if considered a slippage-related issue, this family deserves to be separated based on the following:

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

The decision remains, accept the escalation and validate with medium severity, the decision will be applied tomorrow 10 AM UTC.

0xsimao

> As mentioned in the previous comment, it should not be expected that users will arbitrage the rate back unless it is to their profit. Arbitrage is only expected if it would profit the arbitrageurs. The attacker can set the price by using a very small amount e.g. .01 RA and 0.001 CT and we should not expect someone to be able to make any profit off that.

It is for their profit, they will do it.

> The users at loss specifically here are LV depositors, they are looking to earn yield on their RA tokens, they should not be expected to know about an internal RA:CT pool that LV internally deposits into as the pool's current rate does not matter to them in anyway during the deposit.

Again, it was their choice to deposit with a bad exchange rate. Unless they are frontrunned. Obviously users do not deposit blindly, they should set an exchange rate limit, but they do not, as it is a slippage issue. This is like saying depositing in a uniswap pool should not have slippage control, which makes no sense.

> What matters here is the RA to LV exchange rate, and it would be correct. So if we implement slippage protection for this, it won't fix the issue.

It fixes it because users would never deposit with a bad exchange rate, it would be a mistake.

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

But everything is the same, slippage when interacting with a uniswap pool. The bug is here:

```
(,, uint256 lp) = ammRouter.addLiquidity(
    token0, token1, token0Amount, token1Amount, token0Tolerance, token1Tolerance,
↪   address(this), block.timestamp
);
```

The tolerances are incorrectly calculated on chain, they should be passed as arguments. Then it would be fixed.

WangSecurity

> Again, it was their choice to deposit with a bad exchange rate. Unless they are frontrunned. Obviously users do not deposit blindly, they should set an exchange rate limit, but they do not, as it is a slippage issue. This is like saying depositing in a uniswap pool should not have slippage control, which makes no sense

The exchange rate remains before and after the deposit. Even if you set an exchange rate limit it doesn't matter here because it doesn't change and that slippage check

would be satisfied.

> It fixes it because users would never deposit with a bad exchange rate, it would be a mistake

The RA:LV exchange would be correct, and it wouldn't be a mistake to deposit at the correct RA:LV exchange rate. The problem is in the RA:CT exchange rate and setting slippage protection to RA:LV exchange rate doesn't fix the problem of the RA:CT exchange rate.

My decision to accept the escalation and validate with medium severity remains and it's the final decision. Planning to apply it tomorrow after 10 AM UTC.

WangSecurity

Result: Medium Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- cvetanovv: rejected

Issue M-5: Withdrawing all `lv` before expiry will lead to lost funds in the Vault

Source: https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/211

Found by

0x73696d616f, dimulski

Summary

VaultLib:redeemEarly() redeems users' liquidity vault positions, `lv` for `Ra`, before expiry. After expiry, it is not possible to deposit into the vault or redeem early.

Whenever all users redeem early, when it gets to the expiry date, VaultLib::_liquidatedLp() is called to remove the lp position from the `AMM` into `Ra` and `Ct` (redeem from the `PSM` into more `Ra` and `Pa`) and split among all `lv` holders.

However, as the total supply of `lv` is `0` due to users having redeemed all their positions via `VaultLib::redeemEarly()`, when it gets to VaultPoolLib::reserve(), it reverts due to a division by `0` error, never allowing the `Vault::_liquidatedLp()` call to go through.

As the `Ds` has expired, it is also not possible to deposit into it to increase the `lv` supply, so all funds are forever stuck.

Root Cause

In `MathHelper:134`, the `ratePerLv` reverts due to division by 0. It should calculate the rate after the return guard that checks if the `totalLvIssued==0`.

Internal pre-conditions

1. All users must redeem early.

External pre-conditions

None.

Attack Path

1. All users redeem early via `Vault::redeemEarlyLv()`.
2. The `Ds` expires and there is no `lv` tokens, making all funds stuck.

Impact

All funds are stuck.

PoC

The `MathHelper` separates liquidity by calculating first the `ratePerLv`, which will trigger a division by 0 revert.

```
function separateLiquidity(uint256 totalAmount, uint256 totalLvIssued, uint256
↪  totalLvWithdrawn)
    external
    pure
    returns (uint256 attributedWithdrawal, uint256 attributedAmm, uint256 ratePerLv)
{
    // with 1e18 precision
    ratePerLv = ((totalAmount * 1e18) / totalLvIssued);

    // attribute all to AMM if no lv issued or withdrawn
    if (totalLvIssued == 0 || totalLvWithdrawn == 0) {
        return (0, totalAmount, ratePerLv);
    }
    ...
}
```

Mitigation

The `MathHelper` should place the return guard first:

```
function separateLiquidity(uint256 totalAmount, uint256 totalLvIssued, uint256
↪  totalLvWithdrawn)
    external
    pure
    returns (uint256 attributedWithdrawal, uint256 attributedAmm, uint256 ratePerLv)
{
    // attribute all to AMM if no lv issued or withdrawn
    if (totalLvIssued == 0 || totalLvWithdrawn == 0) {
        return (0, totalAmount, 0);
    }
    ...
    // with 1e18 precision
    ratePerLv = ((totalAmount * 1e18) / totalLvIssued);
}
```

Discussion

sherlock-admin2

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

tsvetanovv commented:

> Borderline Low/Medium

ziankork

This is valid. will fix

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Cork-Technology/Depeg-swap/pull/80

Issue M-6: Rebasing tokens are not supported contrary to the readme and will lead to loss of funds

Source: https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/235

The protocol has acknowledged this issue.

Found by

0x73696d616f, Kirkeelee, dimulski

Summary

The readme states that rebasing tokens are supported

> Rebasing tokens are supported with exchange rate mechanism

However, only non rebasing tokens such as the wrapped version `wsteth` are supposed. If `stETH` is used, it will accrue value in the `Psm` and `Vault` (technically they are the same contract) which will be left untracked as `Ra` and `Pa` deposits are tracked in state variables.

Root Cause

The code does not handle rebasing tokens even though the readme says it does. The exchange rate mechanism only supports non rebasing tokens such as `wsteth`.

Internal pre-conditions

None.

External pre-conditions

None.

Attack Path

Admin creates `steth` pairs using it as `Ra` or `Pa`, whose value will grow in the protocol but left untracked as the quantites are tracked with state variables.

Impact

Stuck yield accruel in the Vault/Psm contracts.

PoC

`State.sol` tracks the balances:

```
struct Balances {
    PsmRedemptionAssetManager ra;
    uint256 dsBalance;
    uint256 ctBalance;
    uint256 paBalance;
}
```

Mitigation

Don't set rebasing tokens are `Ra` or `Pa` or implement a way to sync the balances.

Discussion

ziankork

this is intended as there's no easy way to sync balance between psm and vault since when we sync we have to somehow calculate the virtual reserves of each PSM and Vault from the total shares that the contract holds. so won't fix

Issue M-7: Providing liquidity to the AMM does not check the return value of actually provided tokens leading to locked funds.

Source: https://github.com/sherlock-audit/2024-08-cork-protocol-judging/issues/240

Found by

0x73696d616f, sakshamguruji, vinica_boy

Summary

When providing liquidity to an AMM pair, the protocol specifies both the desired amount of tokens to be provided and a minimum amount to be accepted. Any difference between the two–meaning the amount not used by the AMM–should be properly accounted for within the protocol, as it is not taken by the AMM.

Vulnerability Detail

The `__addLiquidityToAmmUnchecked` function is used to provide liquidity to the RA:CT AMM pair. In the current implementation, `raTolerance` and `ctTolerance` are calculated based on the reserves of the pair during the current transaction with 1% slippage tolerance. The amounts to be provided are determined by the current price ratio in the pair, which ensures that the amounts are almost always exactly what the AMM expects to maintain the `X*Y=K` constant product formula.

```
function __addLiquidityToAmmUnchecked(
        State storage self,
        uint256 raAmount,
        uint256 ctAmount,
        address raAddress,
        address ctAddress,
        IUniswapV2Router02 ammRouter
    ) internal {
        (uint256 raTolerance, uint256 ctTolerance) =
            MathHelper.calculateWithTolerance(raAmount, ctAmount,
↪   MathHelper.UNIV2_STATIC_TOLERANCE);

        ERC20(raAddress).approve(address(ammRouter), raAmount);
        ERC20(ctAddress).approve(address(ammRouter), ctAmount);

        (address token0, address token1, uint256 token0Amount, uint256
↪   token1Amount) =
```

```
            MinimalUniswapV2Library.sortTokensUnsafeWithAmount(raAddress,
↪   ctAddress, raAmount, ctAmount);
        (,, uint256 token0Tolerance, uint256 token1Tolerance) =
            MinimalUniswapV2Library.sortTokensUnsafeWithAmount(raAddress,
↪   ctAddress, raTolerance, ctTolerance);

        (,, uint256 lp) = ammRouter.addLiquidity(
            token0, token1, token0Amount, token1Amount, token0Tolerance,
↪   token1Tolerance, address(this), block.timestamp
        );

        self.vault.config.lpBalance += lp;
    }
```

The current implementation does not check the actual amounts used by the AMM when providing liquidity. As a result, small differences (1-2 wei of the corresponding token) between the provided amount and the actual amount used by the AMM may remain locked in the contract. These differences arise from rounding in the RA:CT price ratio calculations and the corresponding amounts that should be provided. Over time, these small discrepancies could accumulate, leading to higher amount of locked tokens in the contract.

PoC:

Adjust the `__addLiquidityToAmmUnchecked()` function to:

```
function __addLiquidityToAmmUnchecked(
        State storage self,
        uint256 raAmount,
        uint256 ctAmount,
        address raAddress,
        address ctAddress,
        IUniswapV2Router02 ammRouter
    ) internal {
        (uint256 raTolerance, uint256 ctTolerance) =
            MathHelper.calculateWithTolerance(raAmount, ctAmount,
↪   MathHelper.UNIV2_STATIC_TOLERANCE);

        ERC20(raAddress).approve(address(ammRouter), raAmount);
        ERC20(ctAddress).approve(address(ammRouter), ctAmount);

        (address token0, address token1, uint256 token0Amount, uint256
↪   token1Amount) =
            MinimalUniswapV2Library.sortTokensUnsafeWithAmount(raAddress,
↪   ctAddress, raAmount, ctAmount);
        (,, uint256 token0Tolerance, uint256 token1Tolerance) =
            MinimalUniswapV2Library.sortTokensUnsafeWithAmount(raAddress,
↪   ctAddress, raTolerance, ctTolerance);
```

```
        uint256 lp;
        // add one more block to avoid stack too deep errors
        {
            uint256 actual0;
            uint256 actual1;
            (actual0, actual1, lp) = ammRouter.addLiquidity(
                token0, token1, token0Amount, token1Amount, token0Tolerance,
↪   token1Tolerance, address(this), block.timestamp
            );
            if(actual0 != token0Amount || actual1 != token1Amount){
                revert();
            }
        }
        self.vault.config.lpBalance += lp;
    }
```

Running the tests with the following function would result in some tests failing due to this difference in provided and used amounts.

Impact

The impact of these small amounts of locked funds is not significant on their own, but due to the compound effect over time and the high likelihood of this happening with each liquidity provision, the overall severity of the issue should be considered Medium.

Code Snippet

`VaultLib.__addLiquidityToAmmUnchecked()`:
https://github.com/sherlock-audit/2024-08-cork-protocol/blob/db23bf67e45781b00ee6de5f6f23e621af16bd7e/Depeg-swap/contracts/libraries/VaultLib.sol#L55

Tool Used

Manual Review

Recommendation

To handle the small differences between the provided and actual amounts used by the AMM, the return values of the `addLiquidity()` function should be checked, as shown in the adjusted `__addLiquidityToAmmUnchecked()` function. This allows the protocol to detect any discrepancies and take appropriate action.

Depending on the protocol's decision, these leftover funds can either be:

- Returned to users to prevent token loss, ensuring they are not penalized by the rounding differences.

- Accounted for by the protocol and later used for liquidity provision or distributed as rewards, thereby ensuring the funds are not wasted and remain within the protocol's ecosystem.

Discussion

ziankork

yes this is valid and have been fixed, will provide the fix links later

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Cork-Technology/Depeg-swap/pull/71/commits/4944b7dbf56144d389fa578432536bf1c44539c2

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.