SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	Copra Finance
Prepared by:	Sherlock
Lead Security Expert:	hash
Dates Audited:	March 18 - March 21, 2024
Prepared on:	April 29, 2024

Introduction

Copra Finance

Scope

Repository: copra-finance/copra-v3-audit-1

Branch: main

Commit: e24b1069139e181f2ca0c225b13e53e03e8665e3

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
11	2

Security experts who found valid issues

hash	alexzoid	Arabadzhiev
TurnipBoy	0xStalin	

Issue H-1: Underlying LP value can be manipulated to drain teahouse and orange warehouses

Source:
https://github.com/sherlock-audit/2023-12-copra-finance-judging/issues/5

Found by

TurnipBoy, hash

Summary

Both the orange and teahouse utilizes slot0 to determine the value of the vault when depositing or withdrawing. This allows manipulation of the pool value to withdraw more expected.

Vulnerability Detail

OrangeDopexV2LPAutomator.sol#L241-L272

```
function totalAssets() public view returns (uint256) {

    ...

    (uint160 _sqrtRatioX96, , , , , , ) = pool.slot0();

        for (uint256 i = 0; i < _length; ) {
            _lt = int24(uint24(activeTicks.at(i)));
            _ut = _lt + poolTickSpacing;
            _tid = handler.tokenId(address(pool), handlerHook, _lt, _ut);


            _liquidity =
            ↪   handler.convertToAssets((handler.balanceOf(address(this),
            ↪   _tid)).toUint128(), _tid);


            (_a0, _a1) = LiquidityAmounts.getAmountsForLiquidity(
                _sqrtRatioX96,
                _lt.getSqrtRatioAtTick(),
                _ut.getSqrtRatioAtTick(),
                _liquidity
            );
            _sum0 += _a0;
            _sum1 += _a1;


            unchecked {
                i++;
            }
        }
```

First to show that the value of liquidity is determined using slot0 which is easily manipulated. Above is shown the Orange vault which slot0 to determine the price to calculate underlying assets.

TeaVaultV3Pair.sol#L673-L676

```
function estimatedValueInToken0() external override view returns (uint256
 ↪  value0) {
    (uint256 _amount0, uint256 _amount1) = vaultAllUnderlyingAssets();
    value0 = VaultUtils.estimatedValueInToken0(pool, _amount0, _amount1);
}
```

VaultUtils.sol#L131-L143

```
function estimatedValueInToken0(
    IUniswapV3Pool pool,
    uint256 _amount0,
    uint256 _amount1
) external view returns (uint256 value0) {
    (uint160 sqrtPriceX96, , , , , , ) = pool.slot0();


    value0 = _amount0 + FullMath.mulDiv(
        _amount1,
        FixedPoint96.Q96,
        FullMath.mulDiv(sqrtPriceX96, sqrtPriceX96, FixedPoint96.Q96)
    );
}
```

We see the same thing for teahouse, using slot0 to calculate the price of the second asset.

Next is to show how value can be manipulated. Assume we have a pool of 2 assets each worth $1 containing 100 of each asset. At rest our pool is worth: $1 * 100 + $1 * 100 = $200. Now swap 100 asset0 to manipulate the price. This leaves the pool with 200 token0 and 50 token1. Recalculate our value: $1 * 200 + 50 * $1. The pool is now valued at $250.

This alone is not enough to exploit the pool. You see if you were to withdraw at this manipulated value then the attacker would lose the same amount of value that they would gain since they must restore the pool to it's previous price.

The final trick to make this work is to manipulate a single underlying pool and then withdraw from a pool that is not manipulated. Assume we utilize two of the pools like the one described and there is 400 LP. The total value of the pools are $400 and each LP should be worth $1. Now we manipulate one pool and it is worth $250. This brings the total value to $450 and the contract thinks each LP is worth $450/400 = $1.125. Now we can withdraw from the other pool that is not manipulated. Since that pool is not manipulated the attack will be profitable.

The attack would flow as shown:

1. Assume there are two underlying pools
2. Manipulate the price of one pool to increase LP price
3. Withdraw from the other vault that isn't manipulated
4. Restore the first pool to the market price
5. Profit

Impact

Orange and teahouse warehouses can drained by manipulating underlying pools

Code Snippet

TeahouseLiquidityWarehouse.sol#L67-L74

OrangeLiquidityWarehouse.sol#L42-L45

Tool used

Manual Review

Recommendation

Use chainlink oracles to calculate share value instead of using the underlying vaults.

Discussion

cds95

Hi apologies if I'm missing something obvious but I am having trouble understanding the finding. Currently the Liquidity Warehouse is the contract that will withdraw from whitelisted Orange/Teahouse pools. Are you saying that an attacker would:

1. Swap assets on a Teahouse/Orange pool
2. Call `withdrawLender/withdrawBorrower` on the `LiquidityWarehouse` and pass in the address of the manipulated and not manipulated pool?

Also it looks like swaps can only be performed by permissioned roles for both Orange/Teahouse. Does this mean that only these addresses can perform such attacks?

IAmTurnipBoy

Escalate

I should clarify. Teahouse and orange only allow authorized addresses to rebalance their vaults. In this case I mean that the underlying Uniswap V3 pools is what the attacker would manipulate. These are permissionless and anyone can swap

sherlock-admin2

> Escalate
>
> I should clarify. Teahouse and orange only allow authorized addresses to rebalance their vaults. In this case I mean that the underlying Uniswap V3 pools is what the attacker would manipulate. These are permissionless and anyone can swap

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.

cds95

So are you saying that the attacker could manipulate the Uniswap pools, which would impact the LP token value of the Teahouse/Orange pools and that this could make the LiquidityWarehouse's LP tokens worthless?

IAmTurnipBoy

The LP values of the underlying pool can be manipulated so that the estimatedValueInToken0 will return at a higher value. This will cause the shares to be overvalued and the warehouse will pay out too much to the user who is withdrawing

10xhash

> So are you saying that the attacker could manipulate the Uniswap pools, which would impact the LP token value of the Teahouse/Orange pools and that this could make the LiquidityWarehouse's LP tokens worthless?

This is also possible. See the duplicated issue #19. This report mentions the opposite (ie. increasing the share value by making a swap in uniswap) which an attacker can make use of by withdrawing the shares from another un-manipulated pool

cvetanovv

I think the report may be valid. In short, both reports say that the protocol fully trusts the price provided by the external protocol (uniswap). But these pools(in uniswap) can be manipulated and the recommendation is to integrate Oracle or twap to check the price. But because there is no such protection, a pool can indeed be manipulated.

cds95

Noted. Just to clarify then would the proposed solution be to read the share values in the base/counter assets and convert them to the base asset using CL oracles?

I.e something like this for Teahouse

```
function _convertToTeahouseAssets(address withdrawTarget, uint256 shareAmt)
 ↪  internal view returns (uint256) {
      ITeaVaultV3Pair teahouseVault = ITeaVaultV3Pair(withdrawTarget);
      (uint256 amount0, uint256 amount1) =
       ↪  teahouseVault.vaultAllUnderlyingAssets();
      address baseAsset = address(s_terms.asset);

      uint256 baseAssetAmt;
      uint256 counterAssetAmt;
      address counterAsset;

      if (baseAsset == teahouseVault.assetToken0()) {
          baseAssetAmount = amount0;
          counterAssetAmt = amount1;
          counterAsset = teahouseVault.assetToken1();
      } else {
          baseAssetAmount = amount1;
          counterAssetAmt = amount0;
          counterAsset = teahouseVault.assetToken0();
      }

      // Convert using CL Oracles
      baseAssetAmount += _convertCounterAsset(baseAsset, counterAsset,
       ↪  counterAssetAmt);
      return shareAmt * baseAssetAmount / IERC20(withdrawTarget).totalSupply();
  }
```

Also any suggestions on what to do if there isn't a CL Oracle available for the given asset? The reason I ask is because we are also interacting with a USDC/USDC.e pair.

Czar102

I believe this is a valid issue. Planning to accept the escalation and make this and #19 valid duplicates.

cds95

Update on this issue. We're planning to convert between the base and counter assets using TWAPs using Teahouse's asset oracles here.

Czar102

Result: High Has Duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- IAmTurnipBoy: accepted

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits: https://github.com/copra-finance/copra-v3-audit-1/pull/4

Issue H-2: Malicious Data Injection via `TeahouseLiquidityWarehouse._withdrawFromTarget`

Source:
https://github.com/sherlock-audit/2023-12-copra-finance-judging/issues/32

Found by

0xStalin, TurnipBoy, alexzoid, hash

Summary

`deactivate()` and `liquidate()` functions as well as `withdrawLender` and `withdrawBorrower` functions allow attacker to inject malicious `bytes calldata data`. This specifically pertains to the execution of `multicall` in the Teahouse Liquidity Warehouse extension, where malicious swap data can lead to adverse swaps via `uniswapV3Swap`, resulting in a drain of assets.

Vulnerability Detail

The `TeahouseLiquidityWarehouse` contract facilitates interactions through `multicall`, which includes swap operations. Malicious actors can exploit the `deactivate()` or `liquidate()` functions by passing in crafted `swapData` that triggers detrimental swaps, draining assets without the protocol receiving the intended swap outcomes.

This vulnerability stems from the lack of validation on the `swapData` passed into `multicall`, which can include calls to `uniswapV3Swap` with specifically crafted parameters.

Impact

This vulnerability will lead to significant financial losses by enabling attackers to execute swaps that deplete the protocol's assets without receiving fair value in return.

Code Snippet

https://github.com/sherlock-audit/2023-12-copra-finance/blob/7c445528b104dcca2ad3feb5dad6f46b3e2f6daa/copra-v3-audit-1/src/LiquidityWarehouse.sol#L114-L122

```
/// @inheritdoc ILiquidityWarehouse
function deactivate(address[] calldata withdrawTargets, bytes calldata data)
↪  external {
    if (_isLiquidationThresholdFulfilled()) revert
↪   LiquidationThresholdNotBreached();
    _compoundInterest();
    s_isActive = false;
    _withdrawAllFromTargets(withdrawTargets, data);
    emit Deactivated();
}
```

https://github.com/sherlock-audit/2023-12-copra-finance/blob/7c445528b104dcca2ad3feb5dad6f46b3e2f6daa/copra-v3-audit-1/src/TeahouseLiquidityWarehouse.sol#L51-L54

```
data[1] = swapData; // Vulnerable to injection of malicious data
ITeaVaultV3PairHelper(withdrawTarget).multicall(ITeaVaultV3Pair(teahouseVaultAdd ⌋
↪   r), 0, 0, data);
```

https://github.com/TeahouseFinance/TeaVaultV3Pair/blob/f7797f9f6266f847aa997641cd0f7601b9ee54e3/contracts/TeaVaultV3PairHelper.sol#L38-L51

```
/// @inheritdoc ITeaVaultV3PairHelper
function multicall(
    ITeaVaultV3Pair _vault,
    uint256 _amount0,
    uint256 _amount1,
    bytes[] calldata _data
) external payable returns (bytes[] memory results) {
    if (address(vault) != address(0x1)) {
        revert NestedMulticall();
    }

    vault = _vault;
    IERC20 token0 = IERC20(_vault.assetToken0());
    IERC20 token1 = IERC20(_vault.assetToken1());
```

https://github.com/TeahouseFinance/TeaVaultV3Pair/blob/f7797f9f6266f847aa997641cd0f7601b9ee54e3/contracts/TeaVaultV3PairHelper.sol#L189-L205

```
/// @inheritdoc IGenericRouter1Inch
function uniswapV3Swap(
    uint256 amount,
    uint256 minReturn,
    uint256[] calldata pools
) external payable override onlyInMulticall returns(uint256 returnAmount) {
    uint256 poolData = pools[0];
    bool zeroForOne = poolData & (1 << 255) == 0;
    IUniswapV3Pool swapPool = IUniswapV3Pool(address(uint160(poolData)));
    address srcToken = zeroForOne? swapPool.token0(): swapPool.token1();

    IERC20 token = IERC20(srcToken);
    if (token.allowance(address(this), address(router1Inch)) < amount) {
        token.approve(address(router1Inch), type(uint256).max);
    }
    returnAmount = router1Inch.uniswapV3Swap(amount, minReturn, pools);
}
```

Tool used

Manual Review

Recommendation

It is recommended to implement robust validation mechanisms for `swapData` before its execution within `multicall`. This could include checks on the legitimacy of swap paths, slippage rates, and the authenticity of pool addresses to ensure they align with expected parameters and are not detrimental to the protocol.

Discussion

cds95

Acknowledged

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/copra-finance/copra-v3-audit-1/pull/4

10xhash

> The protocol team fixed this issue in PR/commit copra-finance/copra-v3-audit-1#4.

Fixed Now user's can only pass the deadline and minAmountOut associated with the swap. The swapping pool is also checked for manipulation

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-1: withdrawAmount is incorrect when processing withdrawals from TeahouseLiquidityWarehouse

Source:
https://github.com/sherlock-audit/2023-12-copra-finance-judging/issues/3

Found by

TurnipBoy, hash

Summary

Withdraws from TeahouseLiquidityWarehouse, loop through all teahouse vaults when withdrawing. Withdraw amount will be withdrawn for EVERY liquidity vault instead of only a single one. This will lead to excess withdrawals. Since teahouse vaults withdrawals are subject to withdrawal fees it will cause loss to the vault through excess withdrawal fees.

Vulnerability Detail

LiquidityWarehouse.sol#L552-L562

```
while (missingWithdrawableAmount > 0 && idx < withdrawTargets.length) {
    address withdrawTarget = withdrawTargets[idx];


    if (!s_withdrawTargets.contains(withdrawTarget)) revert
     ↪  InvalidWithdrawTarget(withdrawTarget);


    _withdrawFromTarget(missingWithdrawableAmount, withdrawTarget, data);


    currentAssetBalance = s_terms.asset.balanceOf(address(this));
    missingWithdrawableAmount = currentAssetBalance > withdrawAmount ? 0 :
     ↪  withdrawAmount - currentAssetBalance;
    ++idx;
}
```

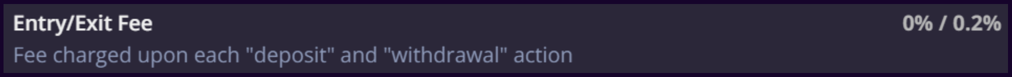
_withdrawFromTargets is structured to withdraw one at a time from each target until the full amount has been withdrawn to cover the withdrawal.

TeahouseLiquidityWarehouse.sol#L37-L54

```
for (uint256 i; i < s_teahouseVaults.length(); ++i) {
    address teahouseVaultAddr = s_teahouseVaults.at(i);
    bytes[] memory data = new bytes[](2);


    // Calculate number of shares to burn which is the minimum of the amount of
    ↪  shares required to withdraw
    // the withdraw amount and the LP balance of the liquidity warehouse
    uint256 sharesAmt = Math.min(
        _convertToTeahouseShares(teahouseVaultAddr, withdrawAmount),
        IERC20(withdrawTarget).balanceOf(address(this))
    );


    // 1) Action to withdraw
    data[0] = abi.encodeWithSelector(ITeaVaultV3Pair.withdraw.selector,
    ↪  sharesAmt, 0, 0);


    // 2) Action to swap
    data[1] = swapData; // TODO:  Can be improved later but this is for
    ↪  illustration purposes
    ITeaVaultV3PairHelper(withdrawTarget).multicall(ITeaVaultV3Pair(teahouseVaul ⌋
    ↪  tAddr), 0, 0, data);
}
```

TeahouseLiquidityWarehouse is structured in an incompatible way. It attempts to withdraw `withdrawAmount` from each teahouse vault. Assume 100 USDC is needed to cover a withdraw and there are 3 teahouse vaults. Instead of withdrawing only 100 USDC it will withdraw 300 USDC.

https://vault.teahouse.finance/arbitrum/0xB38e48B8Bc33CD65551BdaC8d954801D56625eeC/


Entry/Exit Fee 0% / 0.2%
Fee charged upon each "deposit" and "withdrawal" action

Looking at the teahouse vault we can see it has 0.2% withdrawal fee. The excess funds that are withdrawn will need to be deposited again into the vault. This causes the vault to lose funds to this withdrawal fee.

Impact

Vault will lose funds because of excess withdrawals

Code Snippet

TeahouseLiquidityWarehouse.sol#L32-L55

Tool used

Manual Review

Recommendation

_withdrawFromTarget should be restructured so that it will check the balance of the asset after each withdraw so that it does not withdraw too much.

Discussion

cds95

Acknowledged

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/copra-finance/copra-v3-audit-1/pull/4/files#diff-bf073d21765d60024e7f9fbd0f273359fe8c6586751c9bc5f23b549cbcbac2b6

10xhash

> The protocol team fixed this issue in PR/commit https://github.com/copra-finance/copra-v3-audit-1/pull/4/files#diff-bf073d21765d60024e7f9fbd0f273359fe8c6586751c9bc5f23b549cbcbac2b6.

Fixed The interaction with teavault helper is now eliminated and withdrawals are attempted from a vault only if missingWithdrawableAmount is greater than 0

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-2: Impossible to execute whitelisted actions and send the native token within the same execution.

Source: https://github.com/sherlock-audit/2023-12-copra-finance-judging/issues/6

Found by

0xStalin, alexzoid, hash

Summary

Impossible to execute whitelisted actions that require sending the native token.

Vulnerability Detail

The LiquidityWarehouse.execute() function internally calls the LiquidityWarehouseAccessControl._execute() which is on charge to allow the execution of whitelisted actions.

- **The problem is that the external call is capable of sending the native token, but the LiquidityWarehouse.execute() function is not a payable function.** Also, the receive() or fallback() functions are not implemented on the contract, so, the only means the contract could have to receive the native token to send it to the target contract would be by receiving the native within the same call to the LiquidityWarehouse.execute() function, or in other words, if the exernal call to the target required to send native, the only way that native could be send to the target would be by sending it as part of the call to the LiquidityWarehouse.execute() function.
- **But, because the LiquidityWarehouse.execute() function is not a payable function,** if the `executeActions.value` != 0, the whole execution will revert because the LiquidityWarehouse won't have any native to send to the target contract.

```
function execute(LiquidityWarehouseAccessControl.ExecuteAction[] calldata
↪  executeActions)
    external
    override
    nonReentrant
{
    //@audit-issue => The execute() is not a payable function, thus, it does not
↪   allow the senders to send native when calling it!
    _execute(executeActions, msg.sender == owner());
}
```

```
function _execute(ExecuteAction[] calldata executeActions, bool isOwner)
↪   internal {
    for (uint256 i; i < executeActions.length; ++i) {
        ...

        //@audit-issue => If `executeActions.value` != 0, execution will revert
↪   because the contract has no native tokens to send to the target!
        (bool success,) = executeAction.target.call{value:
↪   executeAction.value}(executeAction.data);
        if (!success) revert ExecutionFailed(executeAction.target, msg.sender,
↪   fnSelector);
    }
}
```

Impact

Impossible to execute whitelisted actions that require sending the native token.

Code Snippet

https://github.com/sherlock-audit/2023-12-copra-finance/blob/main/copra-v3-audit-1/src/LiquidityWarehouse.sol#L248-L254

https://github.com/sherlock-audit/2023-12-copra-finance/blob/main/copra-v3-audit-1/src/utils/LiquidityWarehouseAccessControl.sol#L113

Tool used

Manual Review

Recommendation

Make payable the LiquidityWarehouse.execute() function.

```
function execute(LiquidityWarehouseAccessControl.ExecuteAction[] calldata
↪   executeActions)
    external
+   payable
    override
    nonReentrant
{
    _execute(executeActions, msg.sender == owner());
}
```

Discussion

cds95

Hey thanks for the review. Currently we are thinking of adding a `receive` function and to make this function payable.

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/copra-finance/copra-v3-audit-1/pull/3.

10xhash

Fixed. Added `receive()` function to receive ether and marked the `execute` function as payable

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-3: While doing a deposit, lenders can reenter the withdrawLender() while their LENDER_LP_TOKENS are been minted to force the LiquidityWarehouse to withdraw the liquidity from the withdrawTargets

Source:
https://github.com/sherlock-audit/2023-12-copra-finance-judging/issues/9

Found by

0xStalin, hash

Summary

Malicious users can trick the LiquidityWarehouses into withdrawing liquidity from the borrower's vaults as if the LW contract had been deactivated.

Vulnerability Detail

When a lender makes a deposit using the LiquidityWarehouse.depositLender() function, the LiquidityWarehouse contract mints an equivalent amount LENDER_LP_TOKENS to the depositor for the amount of asset been deposited. When the LP_TOKENS are been minted, the ERC1155._mint() function runs a couple of checks to validate that the LP_TOKENS were received, one of those checks is to **call the onERC1155Received() function** on the depositor's account if the depositor is a contract.

- When the execution is forwarded to the caller, the caller contract can re-enter the LiquidityWarehouse.withdrawLender() function. **The problem here is that the amount of assets been deposited has not been transferred into the LW contract (the assets to be deposited are pulled after the LP_TOKENS are minted)**. So, when the caller contract re-enters the LiquidityWarehouse.withdrawLender() function, the LW will be forced to withdraw the missing liquidity from the specified withdrawTargets, thus, pulling the liquidity from the borrower's vaults, thus, stopping the yield accrual process for all the amount of withdrawn liquidity.

I coded a PoC using the SteadefiIntegrationTest.t.sol test file, help me to add the below changes on this test file:

```
...

+ import {console2} from "forge-std/Test.sol";
```

```
...

contract SteadefiIntegrationTest_WhenLiquidityDeployed is
↪  SteadefiIntegrationTest {

  ...

  function test_ReenterWithdrawLenderWhileDepositingPoC() public {
        //@audit-info => LENDER_ONE has already deposited on the LW, and that
↪  liquidity has been distributed to the borrower's vaults!

        uint256 assetValueDeployedBeforeOnETH_USDC =
↪  s_usdcLw.getDeployedAssetValue(ETH_USDC_USDC_LENDING_VAULT);

        address maliciousContract = address(new
↪  AttackerContract(address(s_usdcLw)));

        changePrank(USDC_WHALE);
        s_usdc.transfer(maliciousContract, LENDER_USDC_DEPOSIT_AMOUNT);

        uint256 attackerLoanTokenBalanceBefore =
↪  s_usdc.balanceOf(address(maliciousContract));

        changePrank(maliciousContract);
        s_usdc.approve(address(s_usdcLw), LENDER_USDC_DEPOSIT_AMOUNT);
        s_usdcLw.depositLender(LENDER_USDC_DEPOSIT_AMOUNT / 2);

        uint256 attackerLoanTokenBalanceAfter =
↪  s_usdc.balanceOf(address(maliciousContract));
        assert(s_usdcLw.balanceOf(maliciousContract,
↪  s_usdcLw.LENDER_LP_TOKEN_ID()) == 0);

        uint256 assetValueDeployedAfterOnETH_USDC =
↪  s_usdcLw.getDeployedAssetValue(ETH_USDC_USDC_LENDING_VAULT);

        console2.log("assetValueDeployedBeforeOnETH_USDC: ",
↪  assetValueDeployedBeforeOnETH_USDC);
        console2.log("assetValueDeployedAfterOnETH_USDC : ",
↪  assetValueDeployedAfterOnETH_USDC);

        assert(assetValueDeployedAfterOnETH_USDC <
↪  assetValueDeployedBeforeOnETH_USDC);
  }

}

//@audit-info => Add the below lines of code at the end in the same file!
```

```
import {IntegrationTestConstants} from "./IntegrationTestConstants.t.sol";

interface IERC1155Receiver {
    function onERC1155Received(
        address operator,
        address from,
        uint256 id,
        uint256 value,
        bytes calldata data
    ) external returns (bytes4);
}
contract AttackerContract is IntegrationTestConstants {
    SteadefiLiquidityWarehouse internal s_usdcLw;

    constructor(address liquidityWharehouse) {
        s_usdcLw = SteadefiLiquidityWarehouse(liquidityWharehouse);
    }

    function onERC1155Received(address, address, uint256, uint256, bytes
↪   calldata) external returns (bytes4) {
        if(msg.sender == address(s_usdcLw)) {
            uint256 LENDER_LP_TOKEN_ID = 0;
            uint256 totalShares = s_usdcLw.balanceOf(address(this),
↪   LENDER_LP_TOKEN_ID);

            address[] memory withdrawTargets = new address[](1);
            withdrawTargets[0] = ETH_USDC_USDC_LENDING_VAULT;
            s_usdcLw.withdrawLender(totalShares,withdrawTargets,bytes(""));
        }
        return IERC1155Receiver.onERC1155Received.selector;
    }
}
```

Before running the PoC, help me to add this helper function on the LiquidityWarehouse contract, it is used on the PoC to query the exact liquidity that is deployed on a specific vault

```
abstract contract LiquidityWarehouse ... {
  ...
  ...
  ...

  //@audit-info => Add this helper function before running the PoC!
  function getDeployedAssetValue(address withdrawTarget) external view virtual
↪   returns (uint256) {
```

SHERLOCK

```
        return _getDeployedAssetValue(withdrawTarget);
    }
}
```

Run the test with the following command: `forge test --match-test test_ReenterWithdrawLenderWhileDepositingPoC -vvv`

- Expected output after running the PoC

```
Ran 1 test for test/integration/SteadefiIntegrationTest.t.sol:SteadefiIntegratio
↪  nTest_WhenLiquidityDeployed
[PASS] test_ReenterWithdrawLenderWhileDepositingPoC() (gas: 720694)
Logs:
  assetValueDeployedBeforeOnETH_USDC:  2999997933
  assetValueDeployedAfterOnETH_USDC :  1894734775

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 838.45ms (2.58ms
↪  CPU time)

Ran 1 test suite in 2.56s (838.45ms CPU time): 1 tests passed, 0 failed, 0
↪  skipped (1 total tests)
```

Impact

By withdrawing liquidity from the borrower's vaults, the accrual yield process is interrupted because the liquidity is pulled out from the vaults where is generating yield and is left in the LW where is doing nothing, this causes the borrower to earn less yield because the interests on the lender's deposits are still accruing even though the liquidity is not actively generating yield (it could also cause that the borrower needs to cover the generated interests on the lender deposits if the interests to pay are greater than the generated yield on the vaults).

Code Snippet

https://github.com/sherlock-audit/2023-12-copra-finance/blob/main/copra-v3-audit-1/src/LiquidityWarehouse.sol#L143-L148

https://github.com/sherlock-audit/2023-12-copra-finance/blob/main/copra-v3-audit-1/src/LiquidityWarehouse.sol#L179-L201

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/281550b71c3df9a83e6b80ceefc700852c287570/contracts/token/ERC1155/ERC1155.sol#L447-L466

Tool used

Manual Review & Foundry to code the PoC

Recommendation

The fix for this issue is to add the `nonReentrant()` modifier to the LiquidityWarehouse.withdrawLender() function. In this way, it will be impossible to reenter the function while the LENDER_LP_TOKENS are been minted for a lender's deposit. Also, I'd recommend following the CEI pattern when doing deposits, first pull the assets from the depositor and then mint the LP_TOKENS.

```
+ function withdrawLender(uint256 shareAmount, address[] calldata
↪   withdrawTargets, bytes calldata data) external nonReentrant {
      ...
  }


  function depositLender(uint256 depositAmount) external whenNotPaused
↪   nonReentrant {
      ...

+     s_terms.asset.safeTransferFrom(msg.sender, address(this), depositAmount);

      _mint(
          msg.sender,
          LENDER_LP_TOKEN_ID,
          _convertToShares(totalSupply(LENDER_LP_TOKEN_ID), totalAssetAmount,
↪   depositAmount),
          bytes("")
      );
-     s_terms.asset.safeTransferFrom(msg.sender, address(this), depositAmount);

      ...
  }
```

Discussion

cds95

Valid finding. Will fix.

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/copra-finance/copra-v3-audit-1/pull/2.

10xhash

Fixed Now the tokens are transferred before the lp tokens are minted and also a non-reentrant modifier is added to the withdrawLender function

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-4: `LiquidityWarehouse::depositBorrower` is susceptible to inflation attacks

Source:
https://github.com/sherlock-audit/2023-12-copra-finance-judging/issues/10

Found by

Arabadzhiev, hash

Summary

The first deposits made using `LiquidityWarehouse::depositBorrower` will be vulnerable to inflation attacks

Vulnerability Detail

The `LiquidityWarehouse` abstract contract, which is inherited by both the `OrangeLiquidityWarehouse` and `SteadefiLiquidityWarehouse` contracts has a function called `depositBorrower` that currently looks like this:

```
function depositBorrower(uint256 depositAmount) external whenNotPaused
↪   nonReentrant {
    _compoundInterest();

    uint256 totalAssetAmount = getBorrowerNetAssetValue();

    _mint(
        msg.sender,
        BORROWER_LP_TOKEN_ID,
        _convertToShares(totalSupply(BORROWER_LP_TOKEN_ID), totalAssetAmount,
↪   depositAmount),
        bytes("")
    );

    s_terms.asset.safeTransferFrom(msg.sender, address(this), depositAmount);

    if (_isLiquidationThresholdFulfilled()) {
        _activate();
    }
    emit AssetDeposited(depositAmount, true);
}
```

It uses the `LiquidityWarehouse::getBorrowerNetAssetValue` function, in order to fetch the `totalAssetAmount`.

```
function getBorrowerNetAssetValue() public view returns (uint256) {
    return getNetAssetValue() - getLenderNetAssetValue();
}
```

As it can be seen, that function simply returns the difference between the `netAssetValue` and the `lenderNetAssetValue`. Since the `netAssetValue` is based on the total asset balance of the contract, this means that whenever some amount of assets is transferred to it in one way or another, the `borrowerNetAssetValue` will also increase.

Finally, if we look at the `LiquidityWarehouse::_convertToShares` function that is used to determine the amount of shares to be minted, we can see that the amount of shares returned by it is based on the `totalAssetAmount`, which in our case is based on the contract balance of the particular asset, as we mentioned above.

```
function _convertToShares(uint256 totalShareAmount, uint256 totalAssetAmount,
↪   uint256 assetAmount)
    internal
    pure
    returns (uint256)
{
    if (totalShareAmount == 0) return assetAmount; // Handle case when no shares
↪   have been minted yet
    return assetAmount.wadMul(totalShareAmount).wadDiv(totalAssetAmount);
}
```

Taking all of those things into account, we come to the following conclusion - If a given user deposits an amount of assets that is less than the price per share, they will receive 0 shares in return. The problem with this is that it can easily be exploited when the first borrower deposit transaction is sent to the mempool, by performing a sandwich attack to it. And to showcase what is meant by that, let's take a look at an example:

1. Bob sends a transaction to make a borrower deposit of 1000 USDC, which also just so happens to be the first borrower deposit made to that particular contract.
2. Unfortunately for him, Alice has been carefully monitoring the mempool, so after she sees his transaction, she sandwiches it with two transactions of her own - a first one that makes a 1 USDC borrower deposit (which she receives 1 share for) and a second that directly transfers 1000 USDC to the contract (which brings up its borrower share price from 1 USDC to 1001 USDC).
3. The transaction of Bob gets executed - He receives 0 borrower shares for his deposit, since the asset amount of it is less than the price per borrower share

4. Alice executes another transaction that withdraws her 1 borrower share - She receives 2001 USDC minus a withdrawal fee

Impact

The first users that deposit assets as a borrowers have a high likelihood of receiving no shares in return

Code Snippet

LiquidityWarehouse.sol#L158-L176

Tool used

Manual Review

Recommendation

Revert when `totalAssetAmount` is equal to 0. Additionally, you might also want to use virtual shares / mint some amount of shares to the 0 address on the first deposit, in order to protect against partial inflation attacks.

Discussion

cds95

Will fix but shouldn't we be reverting when `_convertToShares` returns zero instead?

cvetanovv

request poc

sherlock-admin3

PoC requested from @Arabadzhiew

Requests remaining: **1**

Arabadzhiew

> Will fix but shouldn't we be reverting when `_convertToShares` returns zero instead?

Yeah, you should revert when `_convertToShares` returns zero instead. Looks like I made a mistake when writing the recommendation here. My apologies.

sherlock-admin2

Escalate I believe there are some incorrect assumptions on how the `depositBorrower()` determines the amount of shares for the amount of deposited assets. During the contest, I thought about this attack vector and I coded a PoC, but thanks to the PoC I realized that the attack is not possible because of how the `getBorrowerNetAssetValue()` determines the total amount of assets before calling the `_mint()`.

In the PoC we can realize that each depositor receives their corresponding amount of shares based on the deposited amounts, and such shares are convertible to the exact amount of assets that were deposited.

- The donated tokens are considered as if they were a reward for the borrowers, all those donated tokens will be distributed among the borrowers. This means that "the attacker" will lose a portion of the donated tokens instead of gaining by stealing from the second depositor.

Add the below test in the file `test/liquidityWarehouseV2/LiquidityWarehouseDepositBorrowerTest.t.sol`

```
...
...
...
+ import {console2} from "forge-std/Test.sol";

contract LiquidityWarehouseDepositBorrowerTest is BaseTest {

   ...

    function test_FirstDepositAttackPoC_BorrowerDeposit() public {
        changePrank(LENDER_ONE);
        s_loanToken.approve(address(s_liquidityWarehouse), 10e18);
        s_liquidityWarehouse.depositBorrower(1);

        //@audit-info => Transfers liquidity to the LW to manipulate the
         ↪  assets/share
        s_loanToken.transfer(address(s_liquidityWarehouse), 10e18-1);

        changePrank(LENDER_TWO);
        s_loanToken.approve(address(s_liquidityWarehouse), 19e18);
        s_liquidityWarehouse.depositBorrower(19e18);

        uint256 lender_one_shares =
         ↪  s_liquidityWarehouse.balanceOf(LENDER_ONE,
         ↪  s_liquidityWarehouse.BORROWER_LP_TOKEN_ID());
        uint256 lender_two_shares =
         ↪  s_liquidityWarehouse.balanceOf(LENDER_TWO,
         ↪  s_liquidityWarehouse.BORROWER_LP_TOKEN_ID());

        uint256 lender_one_assets =
         ↪  s_liquidityWarehouse.convertToAssets(true , lender_one_shares);
        uint256 lender_two_assets =
         ↪  s_liquidityWarehouse.convertToAssets(true , lender_two_shares);

        console2.log("Testing");

        console2.log("lender_one_shares: ", lender_one_shares);
        console2.log("lender_two_shares: ", lender_two_shares);

        console2.log("lender_one_assets: ", lender_one_assets);
        console2.log("lender_two_assets: ", lender_two_assets);


        // assertEq(lender_one_assets,lender_two_assets);

    }

}
```

Run the PoC with the below command:

> forge test --match-test test_FirstDepositAttackPoC_BorrowerDeposit -vvvv

Expected Output:

- The lender1 shares are convertible to `[9.666e18]` of assets
- The lender2 shares are convertible to `[1.933e19])` of assets

That means the lender2 shares are convertible to more assets than the ones that were deposited. lender1 (The Attacker) lost a portion of the donated tokens to perform the so-called attack.

```
[PASS] test_FirstDepositAttackPoC_BorrowerDeposit() (gas: 276320)
Logs:
  Testing
  lender_one_shares:  1
  lender_two_shares:  2
  lender_one_assets:  9666666666666666667
  lender_two_assets:  19333333333333333333
```

Because of the previous comments and results of the PoC, I'm escalating

> the validity of this report and I believe it should be marked as invalid. But please feel free to provide a PoC proving my PoC is wrong and the attack is actually doable.

The escalation could not be created because you are not exceeding the escalation threshold.

You can view the required number of additional valid issues/judging contest payouts in your Profile page, in the Sherlock webapp.

stalinMacias

Escalate

I'd like to escalate this report, the protocol is going to be deployed on Arbitrum, and in Arbitrum it is not possible to front-run transactions since there is no mempool.

> An important characteristic of Arbitrum that is related to MEV opportunities is that there is no mempool for bots to frontrun

- From: MEV on L2 FlashBabies (Ha, Vlachou, Kilbourn, De Michellis) - Page 6.

So, because it is not possible to frontrun on the chain where the protocol is going to be deployed, I think the severity of this report doesn't classify as a high. It could be a good QA in case the protocol plans to deploy on another chain in the future, but for the scope of this audit,
the only chain where the protocol is going to be deployed is Arbitrum.

sherlock-admin2

> Escalate
>
> I'd like to escalate this report, the protocol is going to be deployed on Arbitrum, and in Arbitrum it is not possible to front-run transactions since there is no mempool.
>
> > An important characteristic of Arbitrum that is related to MEV opportunities is that there is no mempool for bots to frontrun
>
> - From: MEV on L2 FlashBabies (Ha, Vlachou, Kilbourn, De Michellis) - Page 6.
>
> So, because it is not possible to frontrun on the chain where the protocol is going to be deployed, I think the severity of this report doesn't classify as a high. It could be a good QA in case the protocol plans to deploy on another chain in the future, but for the scope of this audit,
> the only chain where the protocol is going to be deployed is Arbitrum.

The escalation could not be created because you are not exceeding the escalation threshold.

You can view the required number of additional valid issues/judging contest payouts in your Profile page, in the Sherlock webapp.

alexzoid-eth

Escalate

Since the protocol is exclusively deployed on Arbitrum, where front-running transactions are not possible, attackers face significant risks of losing transferred tokens if attacker actions are be after depositor within the same block or followed by subsequent lender actions. I believe the severity should be re-evaluated as a QA.

sherlock-admin2

> Escalate
>
> Since the protocol is exclusively deployed on Arbitrum, where front-running transactions are not possible, attackers face significant risks of losing transferred tokens if attacker actions are be after depositor within the same block or followed by subsequent lender actions. I believe the severity should be re-evaluated as a QA.

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.

Arabadzhiew

> Since the protocol is exclusively deployed on Arbitrum, where front-running transactions are not possible

I can partially agree here. There is indeed no public mempool on Arbitrum, which makes front/back-running more challenging to achieve. However, it does not make it impossible, since there are other ways of getting information about transactions that are about to be executed on-chain, apart from reading them from the mempool. Also, in the case of Arbitrum and other L2 chains, a centralised sequencer has full control over the order at which transactions are going to be executed. Given the fact that in the README of this contest, it is stated that external protocol admins are **restricted**, we can derive that the sequencer is also restricted, meaning that it can easily take advantage of this vulnerability and exploit it in order to extract MEV.

> attackers face significant risks of losing transferred tokens if attacker actions are be after depositor within the same block

This can easily be avoided by performing the exploit from a smart contract that checks whether the `borrowerNetAssetValue` is equal to zero and reverts elsewise.

Even if we close our eyes and forget about all of those things, this vulnerability is still very likely to cause users to receive no / less shares in return for their deposits, in the events where the share price gets increased under totally normal circumstances before their deposits.

Because of all of that, I still believe that this is a valid issue that warrants fixing.

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits: https://github.com/copra-finance/copra-v3-audit-1/pull/7

10xhash

partially fixed

The != 0 check will still allow for inflation attacks since the shares will only be minted on multiples:

Example:

Attacker mints 1 share by depositing 1 token Attacker donates 1e18 - 1 tokens. Now 1 share is worth 1e18 tokens. User deposits 2e18 - 1 tokens. User will only receive 1 share since to receive 2 shares, one must atleast deposit 2e18 tokens. The attacker now gains 1e18/2 amount of tokens. The attack can be carried out with amounts predicting the future user deposits, eg: 0.5e18 + 1, if normally user's deposit 1e18 tokens etc. Increasing the minimum minted shares to 1000 etc. will decrease the loss for user's on such an attack. An ideal general solution would be an initial sacrifice (ie. donating initial 1000 shares to an address so that when inflating is costly for the attacker, similar to what is implemented in uniswap v2) but since the borrower's for copra is specific, there could be whitelist for borrower's

Czar102

Great points made by @Arabadzhiew. Planning to accept the first escalation and consider this issue a valid Medium.

Arabadzhiew

@Czar102 I believe that this issue should remain a high severity one, given the restricted sequencer. If it was not for that, I could have agreed that this issue should be a medium severity one, but given the power of transaction ordering that the sequencer has and the fact that it is restricted makes me consider this issue as one of a high severity. After all, this is pretty much the same as having a public mempool that everyone can take advantage of. Of course, I might be misinterpreting something here, so I am open to having my mind changed.

10xhash

> The protocol team fixed this issue in PR/commit copra-finance/copra-v3-audit-1#7.

Fixed Only trusted address can make the initial deposit preventing this attack

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Czar102

@Arabadzhiew If the admin is restricted and can steal funds, it's a Medium severity issue – not everyone can carry out the exploit. I believe this applies also here, but for a kind of external admin. Hence, I stand by my initial decision.

Arabadzhiew

The point that I was trying to make is that given the restricted nature of this external admin, anyone can go and bribe them so that they execute their transaction X prior to some other transaction Y and vice versa. This will be the same as paying a higher/lower gas price on a blockchain that has a public mempool, it's just that here the mechanism being leveraged is different.

Czar102

@Arabadzhiew there is probably a reason why there hasn't been any frontrunning attack on these networks (the trusted admin hasn't been bribed).

I stand by my previous points. Will apply the decision shortly.

Czar102

Result: Medium Has duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- alexzoid-eth: accepted

Issue M-5: Incorrect decimal handling inside _compoundInterest function

Source:
https://github.com/sherlock-audit/2023-12-copra-finance-judging/issues/17

Found by

hash

Summary

Incorrect decimal handling inside _compoundInterest function makes the protocol loose fees

Vulnerability Detail

```
    function _compoundInterest() internal {
        uint256 lenderBalanceBefore = uint256(s_assetData.lenderBalance);


        if (lenderBalanceBefore > 0 && s_isActive) {
            uint256 compoundedInterest = MathUtils.calculateCompoundedInterest(
                uint256(s_terms.interestRate).wadToRay(),
 ↪  s_assetData.interestLastUpdatedAt, block.timestamp
            ).rayToWad();
            s_assetData.lenderBalance =
 ↪  lenderBalanceBefore.wadMul(compoundedInterest).toUint216();
            uint256 earnedFeeAmount =
                uint256(s_terms.interestFee).wadMul(s_assetData.lenderBalance -
 ↪  lenderBalanceBefore);


=>           uint256 feeShareAmount =
 ↪  totalSupply(LENDER_LP_TOKEN_ID).wadMul(earnedFeeAmount).wadDiv(
                s_assetData.lenderBalance - earnedFeeAmount
            );
```

Here incase the token's decimals are low (usdc,wbtc) then the multiplication `totalSupply(LENDER_LP_TOKEN_ID).wadMul(earnedFeeAmount)` will result in 0 in a lot of ranges (until totalSupply * earnedFeeAmount reaches 1e18 which is dependent on other factors like the interest rate,fee, time between calls etc but in most normal scenarios would result in < 1e18 unless totalSupply,which has the same decimals as the token, is really large). Hence the protocol will loose out on fees

Eg: balance = totalSupply == 1e5 * e6 compoundedInterest = 1.0001e18 // 0.01% hence new balance = 100010000000 protocolFee = 0.01e18 // 1% hence earned fee = 100000

feeShareAmount = (1e11 * (1e5) / 1e18) * .... == 0

with correct ordering = ((1e5 * 1e18 / (100010000000 - 100000)) * (1e11)) / 1e18 == 99990

Impact

Protocol will loose out on fees

Code Snippet

https://github.com/sherlock-audit/2023-12-copra-finance/blob/main/copra-v3-audit-1/src/LiquidityWarehouse.sol#L449-L451

Tool used

Manual Review

Recommendation

Rearrange div and mul as follows:

```
uint256 feeShareAmount = (earnedFeeAmount).wadDiv(
                s_assetData.lenderBalance - earnedFeeAmount
            ).wadMul(totalSupply(LENDER_LP_TOKEN_ID));
```

Discussion

10xhash

Escalate

Can I please know why this was excluded?

sherlock-admin2

> Escalate
>
> Can I please know why this was excluded?

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.

cds95

Hi so IIUC the concern is that this line is going to round down to 0 as `s_assetData.lenderBalance - earnedFeeAmount` is going to be less than 10ˆ18. Wouldn't this only be true if `s_assetData.lenderBalance - earnedFeeAmount` is also 10ˆ18? I guess I'm not understanding why the denominator here would be 10ˆ18 as the `lenderBalance` and `earnedFeeAmount` would both be denominated to the same DP.

```
uint256 feeShareAmount =
↪   totalSupply(LENDER_LP_TOKEN_ID).wadMul(earnedFeeAmount).wadDiv(
               s_assetData.lenderBalance - earnedFeeAmount
);
```

10xhash

> Hi so IIUC the concern is that this line is going to round down to 0 as `s_assetData.lenderBalance - earnedFeeAmount` is going to be less than 10ˆ18. Wouldn't this only be true if `s_assetData.lenderBalance - earnedFeeAmount` is also 10ˆ18? I guess I'm not understanding why the denominator here would be 10ˆ18 as the `lenderBalance` and `earnedFeeAmount` would both be denominated to the same DP.
>
> ```
> uint256 feeShareAmount =
> ↪   totalSupply(LENDER_LP_TOKEN_ID).wadMul(earnedFeeAmount).wadDiv(
>                s_assetData.lenderBalance - earnedFeeAmount
> );
> ```

The issue doesn't occur on `s_assetData.lenderBalance - earnedFeeAmount` but instead occurs on `totalSupply(LENDER_LP_TOKEN_ID).wadMul(earnedFeeAmount)`. If the decimals are say 6, the expansion of this would be for eg: `1e6 x 0.01e6 / 1e18` , with the divisor 1e18 coming due to the usage of wadMul

cds95

I see. Thinking about this again would it not be better to just do this as the decimals will cancel out?

```
uint256 feeShareAmount =
               earnedFeeAmount * totalSupply(LENDER_LP_TOKEN_ID) /
               ↪   (s_assetData.lenderBalance - earnedFeeAmount);
```

SHERLOCK

Thinking of this as the suggested solution

```
uint256 feeShareAmount = (earnedFeeAmount).wadDiv(
                s_assetData.lenderBalance - earnedFeeAmount
            ).wadMul(totalSupply(LENDER_LP_TOKEN_ID));
```

causes the `test_InterestFeesCorrectlyCollected` test to fail.

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/copra-finance/copra-v3-audit-1/pull/5/files#diff-256395475e868890f7000231b0e99669c961877c582bb7e8c0ffdf1a2820c751L420.

10xhash

> I see. Thinking about this again would it not be better to just do this as the decimals will cancel out?
>
> ```
> uint256 feeShareAmount =
>                earnedFeeAmount * totalSupply(LENDER_LP_TOKEN_ID) /
>                ↪  (s_assetData.lenderBalance - earnedFeeAmount);
> ```
>
> Thinking of this as the suggested solution
>
> ```
> uint256 feeShareAmount = (earnedFeeAmount).wadDiv(
>                 s_assetData.lenderBalance - earnedFeeAmount
>             ).wadMul(totalSupply(LENDER_LP_TOKEN_ID));
> ```
>
> causes the `test_InterestFeesCorrectlyCollected` test to fail.

Yes, it would be better to avoid the conversion to e18 entirely. Nothing to be gained with that

cvetanovv

I agree this issue is valid and the reason I exclude it is because I think it is Low severity.

This problem of losing fees, because it rounds to zero, will only be a problem as long as `totalSupply` is a small value. Also if we use the `usdc` token as an example then the loss would be about 10 cents. The temporary loss of fees and the small value that is lost makes me think this is Low severity.

10xhash

The one time loss would be small because that is how the fees will be accumulated ie. sum of extremely small values totalSupply need not be a small value, in the example above I had taken 100k usdc and an extremely high interest rate (large time between calls). I am putting another example scenario below with high totalSupply:

totalSupply = 10 million usdc == 1e7 * 1e6 interest rate = 10% == 0.1e18 fee = 1% == 0.01e18 time between compounding call = 10 secs

hence compoundedInterest = 1000000031709792436 new balance = 10000000317098 earned fee = 3171

here too fee share minted amount will be 0 according to previous calculation (1e13 * 3171 / 1e18) ...

10xhash

> The protocol team fixed this issue in PR/commit https://github.com/copra-finance/copra-v3-audit-1/pull/5/files#diff-256395475e868890f7000231b0e99669c961877c582bb7e8c0ffdf1a2820c751L420.

Fixed the decimals issue but added a problematic computation for feeShares

The decimals issue was fixed by avoiding wadMul and wadDiv and performing native * and / But this PR coupled with PR 6 changed the feeShares calculation which is problematic as explained in PR 6

Czar102

It seems that for $1k in deposits, this rounding can take up to $1k in fees, and in $100k in deposits the number could be as high as $10 per withdrawal.

I believe these amounts exceed dust, so planning to accept this issue as a valid Medium.

10xhash

> > The protocol team fixed this issue in PR/commit https://github.com/copra-finance/copra-v3-audit-1/pull/5/files#diff-256395475e868890f7000231b0e99669c961877c582bb7e8c0ffdf1a2820c751L420.
>
> Fixed the decimals issue but added a problematic computation for feeShares
>
> The decimals issue was fixed by avoiding wadMul and wadDiv and performing native * and / But this PR coupled with PR 6 changed the feeShares calculation which is problematic as explained in PR 6

The PR has been updated to revert back to `s_assetData.lenderBalance` instead of `getLenderNetAssetValue()` for the feeShare computation

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Czar102

Result: Medium Unique

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- 10xhash: accepted

Issue M-6: TeavaultPairHelper is called instead of TeaVaultV3Pair to fetch balance

Source: https://github.com/sherlock-audit/2023-12-copra-finance-judging/issues/24

Found by

hash

Summary

TeavaultPairHelper is called instead of TeaVaultV3Pair to fetch balance

Vulnerability Detail

When withdrawing from Teahouse vaults, the balances are attempted to be fetched from the wtidhrawTarget which would be the address of the `ITeaVaultV3PairHelper` contract and not the vault

```
    function _withdrawFromTarget(uint256 withdrawAmount, address withdrawTarget,
↪   bytes memory swapData)
        internal
        override
    {

        ....

            // @audit should be teahouseVaultAddr instead of withdrawTarget.
↪   withdrawTarget is the helperContract
            uint256 sharesAmt = Math.min(
                _convertToTeahouseShares(teahouseVaultAddr, withdrawAmount),
=>              IERC20(withdrawTarget).balanceOf(address(this))
            );
```

This will cause the `_withdrawFromTarget` function call to fail and user's won't be able to withdraw from Teahouse

Impact

Assets cannot be withdrawn from Teahouse

Code Snippet

https://github.com/sherlock-audit/2023-12-copra-finance/blob/main/copra-v3-audit-1/src/TeahouseLiquidityWarehouse.sol#L32-L47

Tool used

Manual Review

Recommendation

Use TeahouseVaultAddress instead

Discussion

cds95

Will fix

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/copra-finance/copra-v3-audit-1/pull/4/files#diff-bf073d21765d60024e7f9fbd0f273359fe8c6586751c9bc5f23b549cbcbac2b6

10xhash

> The protocol team fixed this issue in PR/commit https://github.com/copra-finance/copra-v3-audit-1/pull/4/files#diff-bf073d21765d60024e7f9fbd0f273359fe8c6586751c9bc5f23b549cbcbac2b6.

Fixed The interaction with TeaVaultV3PairHelper is completely eliminated and now all interactions take place directly with vaults

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-7: Withdrawing all assets from TeaHouse will fail due to overflow

Source:
https://github.com/sherlock-audit/2023-12-copra-finance-judging/issues/26

Found by

hash

Summary

Withdrawing all assets from TeaHouse will fail due to overflow

Vulnerability Detail

When deactivating or liquidating, the internal function `_withdrawAllFromTargets` is called which further invokes `_withdrawFromTarget` with the wildcard number type(uint256).max to indicate that all possible assets must be removed

```
    function _withdrawAllFromTargets(address[] memory withdrawTargets, bytes
 ↪  memory data) internal {
        uint256 numWithdrawTargets = withdrawTargets.length;
        for (uint256 i; i < numWithdrawTargets; ++i) {
            address withdrawTarget = withdrawTargets[i];
            if (!s_withdrawTargets.contains(withdrawTarget)) revert
 ↪  InvalidWithdrawTarget(withdrawTarget);
=>          _withdrawFromTarget(type(uint256).max, withdrawTarget, data);
        }
    }
```

For other integrations, this wildcard number is handled by taking minimum with the total avaible assets in the respective protocols. But for TeaHouse, this is handled incorrectly and a multiplication is attempted with type(uint).max which will inadvertently revert.

```
function _withdrawFromTarget(uint256 withdrawAmount, address withdrawTarget,
 ↪  bytes memory swapData)
    internal
    override
{
    // Other option is to pass in the teahouse vault addresses in swapData
    for (uint256 i; i < s_teahouseVaults.length(); ++i) {
        address teahouseVaultAddr = s_teahouseVaults.at(i);
        bytes[] memory data = new bytes[](2);


        // @audit withdrawAmount == type(uint).max used as is
        uint256 sharesAmt = Math.min(
            _convertToTeahouseShares(teahouseVaultAddr, withdrawAmount),
            IERC20(withdrawTarget).balanceOf(address(this))
        );

    ....
}
```

```
    function _convertToTeahouseShares(address withdrawTarget, uint256 assetAmt)
 ↪  internal view returns (uint256) {
        ITeaVaultV3Pair teahouseVault = ITeaVaultV3Pair(withdrawTarget);
        address token = address(s_terms.asset);
        uint256 estimatedTokenValue = token == teahouseVault.assetToken0()
            ? teahouseVault.estimatedValueInToken0()
            : teahouseVault.estimatedValueInToken1();
=>        return assetAmt * IERC20(withdrawTarget).totalSupply() /
 ↪  estimatedTokenValue;
```

Impact

Calls to deactivate and liquidate will revert and can cause borrower to pay interest till their assets finish. Can be worked around by calling deactivate with 0 targets. It can also allow the withdrawer to enjoy the borrowed amount without interest even after the bond has been deactivated until all the withdrawers manually remove all their assets

Code Snippet

usage of type(uint).max as-is https://github.com/sherlock-audit/2023-12-copra-finance/blob/main/copra-v3-audit-1/src/TeahouseLiquidityWarehouse.sol#L32-L55

will revert when multiplying here
https://github.com/sherlock-audit/2023-12-copra-finance/blob/main/copra-v3-audit-1/src/TeahouseLiquidityWarehouse.sol#L76-L82

Tool used

Manual Review

Recommendation

For the special case, directly assign `sharesAmt` to `IERC20(withdrawTarget).balanceOf(address(this))`

Discussion

cds95

Acknowledged

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits: https://github.com/copra-finance/copra-v3-audit-1/pull/4

10xhash

> The protocol team fixed this issue in PR/commit copra-finance/copra-v3-audit-1#4.

Fixed In case of type(uint).max, the amount of shares to withdraw is set as address(this).balance

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-8: The protocol treasury losses due to receiving and keeping tokens with negative interest rate

Source: https://github.com/sherlock-audit/2023-12-copra-finance-judging/issues/28

The protocol has acknowledged this issue.

Found by

alexzoid

Summary

The protocol treasury faces potential value erosion due to accumulating `BORROWER_LP_TOKEN_ID` tokens, which can diminish in value over time. Furthermore, there's a restriction on redeeming these tokens when the capacity threshold isn't met, leading to potential asset lock-up.

Vulnerability Detail

The treasury receives withdrawal fees in both `LENDER_LP_TOKEN_ID` and `BORROWER_LP_TOKEN_ID` tokens. However, due to the mechanism of interest compounding for lenders, the value of `BORROWER_LP_TOKEN_ID` tokens decreases over time as the borrower's net asset value diminishes. This decrease is a direct consequence of the lender balance increment through `_compoundInterest()`, which effectively reduces the borrower's share in the net asset value.

The process exacerbates when considering the protocol's inability to redeem `BORROWER_LP_TOKEN_ID` tokens under certain conditions, such as when the LiquidityWarehouse is inactive or the requested withdrawal amount exceeds the `capacityThreshold`. This creates a scenario where the treasury could be holding depreciating assets with limited options for conversion back to liquid assets.

Impact

Asset losses, temporary assets blocking.

Code Snippet

https://github.com/sherlock-audit/2023-12-copra-finance/blob/7c445528b104dcca2ad3feb5dad6f46b3e2f6daa/copra-v3-audit-1/src/LiquidityWarehouse.sol#L438-L457

```
/// @notice Updates the lender's balance with the interest owed
function _compoundInterest() internal {
    uint256 lenderBalanceBefore = uint256(s_assetData.lenderBalance);

    if (lenderBalanceBefore > 0 && s_isActive) {
        uint256 compoundedInterest = MathUtils.calculateCompoundedInterest(
            uint256(s_terms.interestRate).wadToRay(),
↪   s_assetData.interestLastUpdatedAt, block.timestamp
        ).rayToWad();
        s_assetData.lenderBalance =
↪   lenderBalanceBefore.wadMul(compoundedInterest).toUint216();
        uint256 earnedFeeAmount =
            uint256(s_terms.interestFee).wadMul(s_assetData.lenderBalance -
↪   lenderBalanceBefore);

        uint256 feeShareAmount =
↪   totalSupply(LENDER_LP_TOKEN_ID).wadMul(earnedFeeAmount).wadDiv(
            s_assetData.lenderBalance - earnedFeeAmount
        );

        _mint(s_terms.feeRecipient, LENDER_LP_TOKEN_ID, feeShareAmount,
↪   bytes(""));
    }

    s_assetData.interestLastUpdatedAt = block.timestamp.toUint40();
}
```

https://github.com/sherlock-audit/2023-12-copra-finance/blob/7c445528b104dcca2ad3feb5dad6f46b3e2f6daa/copra-v3-audit-1/src/LiquidityWarehouse.sol#L370-L372

```
function getBorrowerNetAssetValue() public view returns (uint256) {
    return getNetAssetValue() - getLenderNetAssetValue();
}

function getLenderBalance() public view returns (uint256) {
    uint256 lenderBalance = s_assetData.lenderBalance;
    if (!s_isActive) return lenderBalance;
    uint256 compoundedInterest = MathUtils.calculateCompoundedInterest(
        uint256(s_terms.interestRate).wadToRay(),
↪   s_assetData.interestLastUpdatedAt, block.timestamp
    ).rayToWad();
    return lenderBalance.wadMul(compoundedInterest);
}
```

https://github.com/sherlock-audit/2023-12-copra-finance/blob/7c445528b104dcca2ad3feb5dad6f46b3e2f6daa/copra-v3-audit-1/src/LiquidityWarehouse.sol#L227-L229

```
function withdrawBorrower(uint256 shareAmount, address[] calldata
↪   withdrawTargets, bytes calldata data)
    external
    nonReentrant
{

    if (s_isActive && !_isCapacityThresholdFulfilled()) {
        revert CapacityThresholdBreached();
    }
```

https://github.com/sherlock-audit/2023-12-copra-finance/blob/7c445528b104dcca2ad3feb5dad6f46b3e2f6daa/copra-v3-audit-1/src/LiquidityWarehouse.sol#L217-L223

```
uint256 withdrawalFee = uint256(s_terms.withdrawalFee);
uint256 feeAmount = withdrawalFee.wadMul(withdrawableAmount);
uint256 withdrawableAmountAfterFee = withdrawableAmount - feeAmount;

_safeTransferFrom(
    msg.sender, s_terms.feeRecipient, BORROWER_LP_TOKEN_ID,
↪   withdrawalFee.wadMul(shareAmount), bytes("")
);
```

Tool used

Manual Review

Recommendation

The protocol should consider structuring withdrawal fees from borrowers in the underlying asset tokens instead of `BORROWER_LP_TOKEN_ID` tokens. This adjustment would safeguard the treasury from the negative impact of depreciating token values and restrictive withdrawal conditions.

Discussion

cds95

Hi thanks for reviewing this. Wouldn't the total sum of the fees collected from both the borrower and lender LP tokens be the same though as the value of the lender LP tokens will also increase?

cvetanovv

request poc

sherlock-admin3

PoC requested from @alexzoid-eth

Requests remaining: **2**

alexzoid-eth

Assume Alice and Bob have an agreement to invest in a target.

1. Bob as a borrower deposits 150 tokens.
2. Alice as a lender deposits 849 tokens.
3. Then Alice decides for some reasons to withdraw assets despite fees and agreement.
4. Bob can't find a new lender for investment and at the same time doesn't want to pay fees.
5. While Bob was deciding what to do the fee recipient has withdrawn all received lender tokens.
6. So Bob withdraws 150 tokens, and deposits 8.4 tokens as a lender. The capacity threshold doesn't let the receiver withdraw assets.
7. Let's note the fee recipient assets value: 1.5849 ether.
8. Then Bob waits for the liquidation threshold to partially redeem assets as a lender.
9. To prevent the `_deactivate` function successfully invoked, Bob withdraws 2.5 Eth.
10. Let's note that the fee recipient assets value decreased despite lender withdrawal fees: 1.1907 ether
11. The lender and borrower balances are between the capacity threshold. So the fee recipient can't withdraw even 1 wei as a borrower.
12. After another 6 months the fee recipient assets value is 0.8775 ether.
13. To prevent the `_deactivate` function successfully invoked, Bob withdraws 2.0 Eth.
14. Let's note that the fee recipient assets value decreased despite lender withdrawal fees: 0.9027 ether.
15. Bob repeats this while the receiver's balance is zeroed.

The provided example proves the report statements.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.19;

import {console} from "forge-std/console.sol";
import {BaseTest} from "../BaseTest.t.sol";
import {ILiquidityWarehouse} from "../../src/interfaces/ILiquidityWarehouse.sol";

contract LiquidityWarehouseAudit is BaseTest {

    address _feeRecipient;

    function setUp() public override {
        BaseTest.setUp();
        _feeRecipient = s_liquidityWarehouse.getTerms().feeRecipient;
    }

    function testAudit() public {

        // Bob as a borrower deposits 150 tokens
        changePrank(BORROWER_ONE);
        s_loanToken.approve(address(s_liquidityWarehouse), 150 ether);
        s_liquidityWarehouse.depositBorrower(150 ether);

        // Alice as a lender deposits 849 tokens
        changePrank(LENDER_ONE);
        s_loanToken.approve(address(s_liquidityWarehouse), 849 ether);
        s_liquidityWarehouse.depositLender(849 ether);

        // Then Alce decides for some reasons to withdraw assets despite fees
↪   and agreement.
        address[] memory withdrawTargets = new address[](0);
        s_liquidityWarehouse.withdrawLender(849 ether, withdrawTargets,
↪   bytes(""));

        // While Bob was deciding what to do the fee recipient has withdrawn all
↪   received lender tokens
        changePrank(_feeRecipient);

↪   s_liquidityWarehouse.withdrawLender(s_liquidityWarehouse.getLenderBalance(),
↪   withdrawTargets, bytes(""));

        // Then Bob understands how to reduce the fee for withdrawal. So Bob
↪   withdraws 150 tokens
        changePrank(BORROWER_ONE);
        s_liquidityWarehouse.withdrawBorrower(150 ether, withdrawTargets,
↪   bytes(""));
```

```
        // Then Bob instantly deposits 8.48 tokens as a lender
        s_loanToken.approve(address(s_liquidityWarehouse), 8.4 ether);
        s_liquidityWarehouse.depositLender(8.4 ether);

        // Let's note the fee recipient assets value: 1.5849 ether
        _showFeeRecipientAssetBalance();

        // Since the lender and borrower balances near the capacity threshold,
↪   the fee recipient can  neither withdraw all tokens except for a small amount
↪   nor `_deactivate` the contract.
        _feeRecipientWithdrawBorrowerReverts();
        _deactivateReverts();

        // 6 month pass
        vm.warp(block.timestamp + 180 days);

        // To prevent the `_deactivate` function successfully invoked, Bob
↪   withdraws 2.5 Eth.
        changePrank(BORROWER_ONE);
        s_liquidityWarehouse.withdrawLender(2.5 ether, withdrawTargets,
↪   bytes(""));

        // Let's note that the fee recipient assets value decreased despite
↪   lender withdrawal fees: 1.1907 ether
        _showFeeRecipientAssetBalance();

        // The lender and borrower balances are between the capacity threshold.
↪   So the fee recipient can't withdraw even 1 wei as a borrower.
        _feeRecipientWithdrawBorrowerReverts();
        _deactivateReverts();

        // another 6 month pass
        vm.warp(block.timestamp + 180 days);

        // Let's note that the fee recipient assets value decreased again:
↪   0.8775 ether
        _showFeeRecipientAssetBalance();

        // To prevent the `_deactivate` function successfully invoked, Bob
↪   withdraws 2.0 Eth.
        changePrank(BORROWER_ONE);
        s_liquidityWarehouse.withdrawLender(2 ether, withdrawTargets, bytes(""));

        // Let's note that the fee recipient assets value decreased despite
↪   lender withdrawal fees: 0.9027 ether
        _showFeeRecipientAssetBalance();
        // Bob can continue acting at the same manner
        _feeRecipientWithdrawBorrowerReverts();
        _deactivateReverts();
    }

    function _showFeeRecipientAssetBalance() private view {
        uint256 borrowerShares = s_liquidityWarehouse.balanceOf(_feeRecipient,
 ↪  s_liquidityWarehouse.BORROWER_LP_TOKEN_ID());
        uint256 lenderShares = s_liquidityWarehouse.balanceOf(_feeRecipient,
 ↪  s_liquidityWarehouse.LENDER_LP_TOKEN_ID());
        uint256 borrowerAssets = s_liquidityWarehouse.convertToAssets(true,
 ↪  borrowerShares);
        uint256 lenderAssets = s_liquidityWarehouse.convertToAssets(false,
 ↪  lenderShares);
        console.log("Fee Recipient Assets", borrowerAssets + lenderAssets);
    }

    function _feeRecipientWithdrawBorrowerReverts() private {
        address[] memory withdrawTargets = new address[](0);
        uint256 withdrawAmount = s_liquidityWarehouse.balanceOf(_feeRecipient,
 ↪  s_liquidityWarehouse.BORROWER_LP_TOKEN_ID());
        changePrank(_feeRecipient);
        vm.expectRevert(ILiquidityWarehouse.CapacityThresholdBreached.selector);
        s_liquidityWarehouse.withdrawBorrower(
            withdrawAmount,
            withdrawTargets,
            bytes("")
            );
    }

    function _deactivateReverts() private {
        address[] memory withdrawTargets = new address[](0);
        vm.expectRevert(ILiquidityWarehouse.LiquidationThresholdNotBreached.sele ⌡
 ↪  ctor);
        s_liquidityWarehouse.deactivate(withdrawTargets, "");
    }
}
```

```
[] Compiling...
No files changed, compilation skipped

Ran 1 test for
 ↪  test/liquidityWarehouseV2/LiquidityWarehouseAudit.sol:LiquidityWarehouseAudit
[PASS] testAudit() (gas: 730808)
Logs:
  Fee Recipient Assets 15849000000000000
  Fee Recipient Assets 11907732438668320
  Fee Recipient Assets 877599536970831560
  Fee Recipient Assets 902783163484157730

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 13.20ms (7.89ms CPU
↪   time)

Ran 1 test suite in 71.67ms (13.20ms CPU time): 1 tests passed, 0 failed, 0
↪   skipped (1 total tests)
```

Issue M-9: Interest will be compounded even when the protocol is inactive

Source:
https://github.com/sherlock-audit/2023-12-copra-finance-judging/issues/29

Found by

alexzoid, hash

Summary

It compounds interest even during periods of inactivity, due to the absence of an interest update mechanism upon reactivation.

Vulnerability Detail

The `activate()` function lacks an update to `s_assetData.interestLastUpdatedAt`. As a result, when the protocol transitions from an inactive to active state, interest accrued during the inactivity is incorrectly compounded.

This scenario unfolds as follows:

1. The `DebtCovenant` contract deactivates the LiquidityWarehouse via `deactivate()`, triggering an update to `interestLastUpdatedAt`.
2. The protocol remains inactive for an extended period without user interactions.
3. The attacker (as a lender) donates tokens to fulfill the liquidation threshold.
4. `DebtCovenant` or attacker reactivates the protocol using `activate()`, but `interestLastUpdatedAt` remains unchanged.
5. Subsequent user interactions trigger interest compounding over the entire inactive period, leading to erroneous interest calculations and borrower losses.

Impact

This issue causes inaccurate interest compounding for the duration of the protocol's inactivity, potentially leading to financial discrepancies towards malicious lenders at the expense of borrowers.

Code Snippet

https://github.com/sherlock-audit/2023-12-copra-finance/blob/7c445528b104dcca2ad3feb5dad6f46b3e2f6daa/copra-v3-audit-1/src/LiquidityWarehouse.sol#L123-L128

```
/// @inheritdoc ILiquidityWarehouse
function activate() external {
    if (!_isLiquidationThresholdFulfilled()) revert
↪   LiquidationThresholdBreached();
    _activate();
}
```

https://github.com/sherlock-audit/2023-12-copra-finance/blob/7c445528b104dcca2ad3feb5dad6f46b3e2f6daa/copra-v3-audit-1/src/LiquidityWarehouse.sol#L437-L458

```
/// @notice Updates the lender's balance with the interest owed
function _compoundInterest() internal {
    uint256 lenderBalanceBefore = uint256(s_assetData.lenderBalance);

    if (lenderBalanceBefore > 0 && s_isActive) {
        uint256 compoundedInterest = MathUtils.calculateCompoundedInterest(
            uint256(s_terms.interestRate).wadToRay(),
↪   s_assetData.interestLastUpdatedAt, block.timestamp
        ).rayToWad();
        s_assetData.lenderBalance =
↪   lenderBalanceBefore.wadMul(compoundedInterest).toUint216();
        uint256 earnedFeeAmount =
            uint256(s_terms.interestFee).wadMul(s_assetData.lenderBalance -
↪   lenderBalanceBefore);

        uint256 feeShareAmount =
↪   totalSupply(LENDER_LP_TOKEN_ID).wadMul(earnedFeeAmount).wadDiv(
            s_assetData.lenderBalance - earnedFeeAmount
        );

        _mint(s_terms.feeRecipient, LENDER_LP_TOKEN_ID, feeShareAmount,
↪   bytes(""));
    }

    s_assetData.interestLastUpdatedAt = block.timestamp.toUint40();
}
```

Tool used

VSCode, Foundry

Recommendation

To ensure interest calculations accurately reflect the protocol's active status, update the `interestLastUpdatedAt` timestamp upon activation:

```
diff --git a/copra-v3-audit-1/src/LiquidityWarehouse.sol
 ↪  b/copra-v3-audit-1/src/LiquidityWarehouse.sol
index ce5c79d..cd2d91f 100644
--- a/copra-v3-audit-1/src/LiquidityWarehouse.sol
+++ b/copra-v3-audit-1/src/LiquidityWarehouse.sol
@@ -123,6 +123,7 @@ abstract contract LiquidityWarehouse is
     /// @inheritdoc ILiquidityWarehouse
     function activate() external {
         if (!_isLiquidationThresholdFulfilled()) revert
 ↪  LiquidationThresholdBreached();
+        _compoundInterest();
         _activate();
     }
```

Discussion

cds95

Great finding. We will fix.

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/copra-finance/copra-v3-audit-1/pull/1.

10xhash

Flawed fix The fix allows any user to avoid interest for the borrower's by calling activate() when the state is already active (ie. the bond has never been deactivated). Check that the current status is not active inside the acitvate function()

10xhash

Fixed The PR has been updated to revert in case the status is already active

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-10: The owner of the Steadefi LendingVaults can permanently prevent the Copra LiquidityWarehouses from withdrawing

Source:
https://github.com/sherlock-audit/2023-12-copra-finance-judging/issues/30

The protocol has acknowledged this issue.

Found by

Arabadzhiev, hash

Summary

The Steadefi admin can permanently prevent Copra from withdrawing their assets by pausing their LendingVaults

Vulnerability Detail

As it can be seen, the Steadefi LendingVault has a function called emergencyShutdown, that is capable of pausing the contract at any given moment. The function has a limited access to only addresses that have the keeper role assigned to them. This role can only be assigned by the contract owner.

Furthermore, what we can also see is that the withdraw function of the same contract is only executable when the contract is not paused. This is the only function that allows the withdrawal of assets from the `LendingVault` contracts.

Taking a look at an arbitrary Steadefi LendingVault deployed on Arbitrum, we can also verify that the implementation details referred to above are indeed present in the contracts that are deployed on-chain.

Finally, according to the contest README, external contract owners should **not** be able to prevent Copra from being able to always execute withdrawals:

> Q: In case of external protocol integrations, are the risks of external contracts pausing or executing an emergency withdrawal acceptable? If not, Watsons will submit issues related to these situations that can harm your protocol's functionality.
>
> Pausing is acceptable as long **as we can always withdraw**

What we can conclude from all of this information is that the admin/owner of the Steadefi LendingVaults has the ability to prevent the Copra protocol users from performing an action for an unbounded period of time, that should otherwise always be possible to perform.

Impact

The users of Copra can be prevented from withdrawing their funds

Code Snippet

Tool used

Manual Review

Recommendation

Reconsider whether you really want integrate with Steadefi, given the power that its admin has

Discussion

cds95

Acknowledged

Issue M-11: Steadefi WETH vault cannot facilitate withdrawals when there is insufficient WETH liquidity

Source:
https://github.com/sherlock-audit/2023-12-copra-finance-judging/issues/34

Found by

We released the Steadefi LiquidityWarehouse privately to a few lenders for testing and found an issue with the WETH Steadefi LW when users tried to withdraw by calling the withdraw function after the assets had been deployed to the relevant Steadefi contracts. The issue was that the warehouse received ETH from the Steadefi ETH-USDC LendingVault instead of WETH when withdrawing. This caused withdrawals to fail as the Liquidity Warehouse thought that there was insufficient liquidity because it didn't have enough WETH.

The current suggested fix is to autowrap native tokens when the liquidity warehouse receives it.

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/copra-finance/copra-v3-audit-1/pull/8

10xhash

> The protocol team fixed this issue in the following PRs/commits:
> copra-finance/copra-v3-audit-1#8

Fixed native ETH sent by steadefi vault is now converted to WETH inside the receive function

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Disclaimers

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

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