Good Entry
Findings & Analysis Report

2023-10-02

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Good Entry smart contract system written in Solidity. The audit took place between August 1—August 7 2023.

Following the C4 audit, 3 wardens (3docSec, xuwinnie and kutugu) reviewed the mitigations for all identified issues; the mitigation review report is appended below the audit report.

Wardens

82 Wardens contributed reports to the Good Entry:

1. 3docSec
2. xuwinnie
3. kutugu
4. libratus
5. 0xDING99YA
6. Jeiwan
7. said
8. LokiThe5th
9. josephdara
10. T1MOH
11. Sathish9098
12. Vagner
13. R-Nemes
14. HChang26
15. catellatech
16. osmanozdemir1
17. Team_FliBit (14si2o_Flint and Naubit)
18. K42
19. Krace
20. Fulum
21. Limbooo
22. DanielArmstrong
23. radev_sw
24. hassan-truscova
25. Satyam_Sharma
26. auditsea
27. 0xmuxyz

28. nadin
29. n1punp
30. nemveer
31. 0xBeirao
32. Hama
33. n33k
34. Madalad
35. JCK
36. Rolezn
37. oakcobalt
38. 0xAnah
39. Raihan
40. ReyAdmirado
41. digitizeworx
42. 0xSmartContract
43. pep7siup
44. SpicyMeatball
45. jesusrod15
46. giovannidisiena
47. DavidGiladi
48. hpsb
49. ravikiranweb3
50. j4ld1na
51. fatherOfBlocks
52. petrichor
53. wahedtalash77
54. hunter_w3b
55. naman1778
56. 0xhex

57. dharma09
58. SY_S
59. 0xta
60. matrix_0wl
61. SAQ
62. Rageur
63. dd0x7e8
64. UniversalCrypto (amaechieth and tettehnetworks)
65. niser93
66. nonseodion
67. banpaleo5
68. sivanesh_808
69. 8olidity
70. 0x70C9
71. Udsen
72. SanketKogekar
73. parsely
74. shirochan
75. MatricksDeCoder
76. Kaysoft
77. Bughunter101
78. grearlake
79. debo
80. piyushshukla

This audit was judged by gzeon.

Final report assembled by PaperParachute.

Summary

The C4 analysis yielded an aggregated total of 14 unique vulnerabilities. Of these vulnerabilities, 6 received a risk rating in the category of HIGH severity and 8 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 26 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 19 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Good Entry repository, and is composed of 38 smart contracts written in the Solidity programming language and includes 2482 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (6)

[H-01] When price is within position's range, `deposit` at TokenisableRange can cause loss of funds

Submitted by [xuwinnie]

When slot0 price is within the range of tokenized position, function `deposit` needs to be called with both parameters, `n0` and `n1`, greater than zero. However, if price moves outside the range during the transaction, user will be charged an excessive fee.

Proof of Concept

```
if ( fee0+fee1 > 0 && ( n0 > 0 || fee0 == 0) && ( n1 > 0 || fee1
  address pool = V3_FACTORY.getPool(address(TOKEN0.token), addre
  (uint160 sqrtPriceX96,,,,,,)  = IUniswapV3Pool(pool).slot0();
  (uint256 token0Amount, uint256 token1Amount) = LiquidityAmount
  if (token0Amount + fee0 > 0) newFee0 = n0 * fee0 / (token0Amou
  if (token1Amount + fee1 > 0) newFee1 = n1 * fee1 / (token1Amou
  fee0 += newFee0;
  fee1 += newFee1;
  n0   -= newFee0;
  n1   -= newFee1;
}
```

Suppose range is [120, 122] and current price is 121. Alice calls `deposit` with `{n0: 100, n1:100}`, if Price moves to 119 during execution (due to market fluctuations or malicious frontrunning), `getAmountsForLiquidity` will return 0 for `token1Amount`. As a result, `newFee1` will be equal to `n1`, which means all the 100 token1 will be charged as fee.

```
(uint128 newLiquidity, uint256 added0, uint256 added1) = POS_MGR
  INonfungiblePositionManager.IncreaseLiquidityParams({
    tokenId: tokenId,
    amount0Desired: n0,
    amount1Desired: n1,
    amount0Min: n0 * 95 / 100,
    amount1Min: n1 * 95 / 100,
    deadline: block.timestamp
  })
);
```

Then, `increaseLiquidity` will succeed since `amount1Min` is now zero.

Recommended Mitigation Steps

Don't use this to calculate fee:

```
if ( fee0+fee1 > 0 && ( n0 > 0 || fee0 == 0) && ( n1 > 0 || fee1
  address pool = V3_FACTORY.getPool(address(TOKEN0.token), addre
  (uint160 sqrtPriceX96,,,,,,)  = IUniswapV3Pool(pool).slot0();
  (uint256 token0Amount, uint256 token1Amount) = LiquidityAmount
  if (token0Amount + fee0 > 0) newFee0 = n0 * fee0 / (token0Amou
  if (token1Amount + fee1 > 0) newFee1 = n1 * fee1 / (token1Amou
  fee0 += newFee0;
  fee1 += newFee1;
  n0   -= newFee0;
  n1   -= newFee1;
}
```

Always use this:

```
  uint256 TOKEN0_PRICE = ORACLE.getAssetPrice(address(TOKEN0.tok
  uint256 TOKEN1_PRICE = ORACLE.getAssetPrice(address(TOKEN1.tok
  require (TOKEN0_PRICE > 0 && TOKEN1_PRICE > 0, "Invalid Oracle
  // Calculate the equivalent liquidity amount of the non-yet co
  // Assume linearity for liquidity in same tick range; calculat
  feeLiquidity = newLiquidity * ( (fee0 * TOKEN0_PRICE / 10 ** T
                                / ( (added0   * TOKEN0_PRICE / 1
```

Keref (Good Entry) disputed and commented:

> Again this concurrency execution environment stuff. There is no price moving "during" execution.

xuwinnie (Warden) commented:

> Again this concurrency execution environment stuff. There is no price moving "during" execution.

> Hi @Keref, I guess there could be some misunderstanding. Here I mean when price is 121, user will need to submit the tx with {n0: 100, n1:100}, and price could move to 119 when tx gets executed. (something similar to slippage)

Keref (Good Entry) confirmed and commented:

> Hi, sorry I misunderstood the report, accepted.

> See PR#4

Good Entry Mitigated:

> Remove complex fee clawing strategy.
> PR: https://github.com/GoodEntry-io/ge/pull/4

Status: Mitigation confirmed. Full details in reports from kutugu, xuwinnie and 3docSec.

[H-02] Unused funds are not returned and not counted in `GeVault`

Submitted by Jeiwan, *also found by* Jeiwan, LokiThe5th, osmanozdemir1, said, *and* HChang26

Users can lose a portion of their deposited funds if some of their funds haven't been deposited to the underlying Uniswap pools. There's always a chance of such event since Uniswap pools take balanced token amounts when liquidity is added but `GeVault` doesn't pre-compute balanced amounts. As a result, depositing and withdrawing can result in a partial loss of funds.

Proof of Concept

The GeVault.deposit() function is used by users to deposits funds into ticks and underlying Uniswap pools. The function takes funds from the caller and calls `rebalance()` to distribute the funds among the ticks. The GeVault.rebalance() function first removes liquidity from all ticks and then deposits the removed assets plus the user assets back in to the ticks:

```
    function rebalance() public {
        require(poolMatchesOracle(), "GEV: Oracle Error");
        removeFromAllTicks();
        if (isEnabled) deployAssets();
```

```
}
```

The `GeVault.deployAssets()` function calls the GeVault.depositAndStash() function, which actually deposits tokens into a `TokenisableRange` contract by calling the TokenisableRange.deposit(). The function deposits tokens into a Uniswap V3 pool and returns unspent tokens to the caller:

```
(uint128 newLiquidity, uint256 added0, uint256 added1) = POS_MGR
    ...
);

...

_mint(msg.sender, lpAmt);
TOKEN0.token.safeTransfer( msg.sender, n0 - added0);
TOKEN1.token.safeTransfer( msg.sender, n1 - added1);
```

However, the `GeVault.depositAndStash()` function doesn't handle the returned unspent tokens. Since Uniswap V3 pools take balanced token amounts (respective to the current pool price) and since the funds deposited into ticks are not balanced (`deployAssets()` splits token amounts in halves), there's always a chance that the `TokenisableRange.deposit()` function won't consume all specified tokens and will return some of them to the `GeVault` contract. However, `GeVault` won't return the unused tokens to the depositor.

Moreover, the contract won't include them in the TVL calculation:

1. The GeVault.getTVL() function computes the total LP token balance of the contract (`getTickBalance(k)`) and the price of each LP token (`t.latestAnswer()`), to compute the total value of the vault.
2. The GeVault.getTickBalance() function won't count the unused tokens because it only returns the amount of LP tokens deposited into the lending pool. I.e. only the liquidity deposited to Uniswap pools is counted.
3. The TokenisableRange.latestAnswer() function computes the total value (TokenisableRange.sol#L355) of the liquidity deposited into the Uniswap pool (TokenisableRange.sol#L338). Thus, the unused tokens won't be counted here as well.

4. The GeVault.getTVL() function is used to compute the amount of tokens to return to the depositor during withdrawal.

Thus, the unused tokens will be locked in the contract until they're deposited into ticks. However, rebalancing and depositing of tokens can result in new unused tokens that won't be counted in the TVL.

Recommended Mitigation Steps

In the `GeVault.deposit()` function, consider returning unspent tokens to the depositor. Extra testing is needed to guarantee that rebalancing doesn't result in unspent tokens, or, alternatively, such tokens could be counted in a storage variable and excluded from the balance of unspent tokens during depositing.

Alternatively, consider counting `GeVault`'s token balances in the `getTVL()` function. This won't require returning unspent tokens during depositing and will allow depositors to withdraw their entire funds.

Keref (Good Entry) confirmed and commented:

> See update.

Good Entry Mitigated:

> Take unused funds into account for TVL.
> PR: https://github.com/GoodEntry-io/ge/commit/a8ba6492b19154c72596086f5531f6821b4a46a2

Status: Mitigation confirmed. Full details in reports from kutugu, xuwinnie and 3docSec.

[H-03] Overflow can still happen when calculating `priceX8` inside `poolMatchesOracle` operation

Submitted by said, *also found by* radev_sw, Satyam_Sharma, 0xmuxyz, LokiThe5th, Team_FliBit, *and* T1MOH

`poolMatchesOracle` is used to compare price calculated from uniswap v3 pool and chainlink oracle and decide whether rebalance should happened or not. `priceX8` will be holding price information calculated using `sqrtPriceX96` and when operations is performed, it will try to scale down using `2 ** 12`. However, the scale down is not enough and overflow can still happened.

Proof of Concept

Consider this scenario, The GeVault is using WBTC for `token0` and WETH for `token1`.

These are information for the WBTC/WETH from uniswap v3 pool (0x4585FE77225b41b697C938B018E2Ac67Ac5a20c0):

slot0 data (at current time) :

```
    sqrtPriceX96   uint160 :  3152014155488119708324720447996114
```

`token0` (WBTC) decimals is 8 and `token1` (WETH) decimals is 18.

Using these information, try to reproduce the `priceX8` calculation :

```
        function testOraclePrice() public {
            uint160 sqrtPriceX96 = 3152014155488119708324720447996114
            // decimals0 is 8
            uint priceX8 = 10 ** 8;
            // Overflow if dont scale down the sqrtPrice before div
            // @audit - the overflow still possible
            priceX8 =
                (priceX8 * uint(sqrtPriceX96 / 2 ** 12) ** 2 * 1e8)
                2 ** 168;
            // decimals1 is 18
            priceX8 = priceX8 / 10 ** 18;
            assertEq(true, true);
        }
```

the test result in overflow :

```
[FAIL. Reason: Arithmetic over/underflow] testOraclePrice()
```

This will cause calculation still overflow, even using the widely used WBTC/WETH pair

Recommended Mitigation Steps

Consider to change the scale down using the recommended value from uniswap v3 library:

https://github.com/Uniswap/v3-periphery/blob/main/contracts/libraries/OracleLibrary.sol#L49-L69

or change the scale down similar to the one used inside library

```
    function poolMatchesOracle() public view returns (bool matches
      (uint160 sqrtPriceX96,,,,,,) = uniswapPool.slot0();

      uint decimals0 = token0.decimals();
      uint decimals1 = token1.decimals();
      uint priceX8 = 10**decimals0;
      // Overflow if dont scale down the sqrtPrice before div 2*19
-      priceX8 = priceX8 * uint(sqrtPriceX96 / 2 ** 12) ** 2 * 1e8
+      priceX8 = priceX8 * (uint(sqrtPriceX96) ** 2 / 2 ** 64) * 1
      priceX8 = priceX8 / 10**decimals1;
      uint oraclePrice = 1e8 * oracle.getAssetPrice(address(token0
      if (oraclePrice < priceX8 * 101 / 100 && oraclePrice > price
    }
```

Keref (Good Entry) confirmed and commented:

> See PR#3.

Good Entry Mitigated:

> Scale down `sqrtPriceX96` to prevent overflow.
> PR: https://github.com/GoodEntry-io/ge/pull/3

Status: Mitigation confirmed. Full details in reports from kutugu, xuwinnie and 3docSec.

[H-04] TokenisableRange's incorrect accounting of non-reinvested fees in "deposit" exposes the fees to a flash-loan attack

Submitted by 3docSec

https://github.com/code-423n4/2023-08-goodentry/blob/71c0c0eca8af957202ccdbf5ce2f2a514ffe2e24/contracts/TokenisableRange.sol#L190

https://github.com/code-423n4/2023-08-goodentry/blob/71c0c0eca8af957202ccdbf5ce2f2a514ffe2e24/contracts/TokenisableRange.sol#L268

Vulnerability details

The `TokenisableRange` is designed to always collect trading fees from the Uniswap V3 pool, whenever there is a liquidity event (`deposit` or `withdraw`). These fees may be reinvested in the pool, or may be held in form of `fee0` and `fee1` ERC-20 balance held by the TokenisableRange contract.

When a user deposits liquidity in the range, they pay asset tokens, and receive back liquidity tokens, which give them a share of the TokenisableRange assets (liquidity locked in Unisvap V3, plus fee0, and fee1).

To prevent users from stealing fees, there are several mechanisms in place:

1. fees are, as said, always collected whenever liquidity is added or removed, and whenever they exceed 1% of the liquidity in the pool, they are re-invested in Uniswap V3. The intention of this check seems to be limiting the value locked in these fees
2. whenever a user deposits liquidity to the range, the LP tokens given to them are scaled down by the value of the fees, so the participation in fees "is not given away for free"

Both of these mechanisms can however be worked around:

1. the 1% check is done on the `fee0` and `fee1` **amounts** compared to the theoretical pool amounts, and **not on the total value of the fees** as compared to the total value locked in the pool. This means that when the price changes significantly from when fees were accumulated, the combined value of the fees can exceed, potentially by much, the 1% intended cap, without the reinvestment happening before liquidity events. A malicious user can then monitor and act in such market conditions.
2. the downscaling of the LP tokens minted to the user happens only if none of the provided liquidity is added to the pool fees instead of the Uniswap V3 position. The user can send just a few wei's of tokens to short-circuit the downscaling, and have a share of fees "for free".

Impact

Given a TokenisableRange contract in the right state (high value locked in fees, but still no reinvestment happening) a user can maliciously craft a `deposit` and `withdraw` sequence (why not, with flash-loaned assets) to steal most of the fees (`fee0` , `fee1`) held by the pool before distribution.

Proof of Concept

Below is a working PoC that shows under real market conditions how most of the fees (>3% of the pool assets) can be s stolen risk-free by simply depositing and withdrawing a large quantity of liquidity:

```
function testStolenFeesPoc() public {
    vm.createSelectFork(
        "mainnet",
        17811921
    );

    vm.prank(tokenWhale);
    USDC.transfer(alice, 100_000e6);

    vm.startPrank(alice);
    TokenisableRange tr = new TokenisableRange();

    // out of range: WETH is more valuable than that (about
    // the pool will hold 0 WETH
    tr.initProxy(AaveOracle, USDC, WETH, 500e10, 1000e10, "T
```

```
USDC.approve(address(tr), 100_000e6);
tr.init(100_000e6, 0);

// time passes, and the pool trades in range, accumulati
uint256 fee0 = 1_000e6;
uint256 fee1 = 2e18;

vm.mockCall(address(UniswapV3UsdcNFPositionManager),
    abi.encodeWithSelector(INonfungiblePositionManager.c
    abi.encode(fee0, fee1));

vm.stopPrank();
vm.startPrank(tokenWhale);
USDC.transfer(address(tr), fee0);
WETH.transfer(address(tr), fee1);

// now the price is back to 1870 USDC,
// the undistributed fees are 1k USDC and 2 WETH,
// in total about $5k or 5% of the pool value
// (the percentage can be higher with bigger price swing
// but still, they are not reinvested
tr.claimFee();
vm.clearMockedCalls();
require(tr.fee0() != 0);
require(tr.fee1() != 0);

// an attacker now can flashloan & deposit an amount tha
// the majority of the pool liquidity, then withdraw for
uint256 usdcBalanceBefore = USDC.balanceOf(tokenWhale);
uint256 wethBalanceBefore = WETH.balanceOf(tokenWhale);
uint256 poolSharesBefore = tr.balanceOf(tokenWhale);

USDC.approve(address(tr), 10_000_000e6);
// this is the hack: we add just a tiny little bit of WE
// count the value locked in fees in assigning the LP to
WETH.approve(address(tr), 1000);
uint256 deposited = tr.deposit(10_000_000e6, 1000);
tr.withdraw(deposited, 0, 0);

// the profit here is
// 1 wei of USDC lost, probably to rounding
console2.log(int(USDC.balanceOf(tokenWhale)) - int(usdcB
// 1.58 WETH of profit, which is most of the fees,
// and definitely more than 1% of the pool. Yay!
console2.log(int(WETH.balanceOf(tokenWhale)) - int(wethB
require(poolSharesBefore ==  tr.balanceOf(tokenWhale));
```

```
    }
```

It is important to note that since the WETH oracle price at the forked block (17811921) is at 1870, above the 500-1000 range, the above PoC works only after fixing my other finding titled:

> Incorrect Solidity version in FullMath.sol can cause permanent freezing of assets for arithmetic underflow-induced revert

Recommended Mitigation Steps

- Factor in also the token prices when calculating whether the accrued fees are indeed 1% of the pool
- When minting TokenisableRange tokens, **always** downscale the minted fees by the relative value of non-distributed fees in the pool:

```
    // Stack too deep, so localising some variables for feeLiqui
-    // If we already clawed back fees earlier, do nothing, else
-    if ( newFee0 == 0 && newFee1 == 0 ){
+    {
     uint256 TOKEN0_PRICE = ORACLE.getAssetPrice(address(TOKEN0
     uint256 TOKEN1_PRICE = ORACLE.getAssetPrice(address(TOKEN1
     require (TOKEN0_PRICE > 0 && TOKEN1_PRICE > 0, "Invalid Ora
     // Calculate the equivalent liquidity amount of the non-ye
     // Assume linearity for liquidity in same tick range; calc
     feeLiquidity = newLiquidity * ( (fee0 * TOKEN0_PRICE / 10
                                   / ( (added0   * TOKEN0_PRICE
    }
```

[Keref (Good Entry) confirmed and commented]:

> See **PR#4**.

[Good Entry Mitigated]:

> Remove complex fee clawing strategy.
> PR: https://github.com/GoodEntry-io/ge/pull/4

Status: Mitigation confirmed. Full details in reports from kutugu, xuwinnie and 3docSec.

[H-05] V3Proxy swapTokensForExactETH does not send back to the caller the unused input tokens

Submitted by 3docSec, also found by Fulum, Limbooo, DanielArmstrong, T1MOH, and Krace

The `V3Proxy swapTokensForExactETH` function swaps an unspecified amount of a given ERC-20 for a specified amount of the native currency. After the swap happens, however, the difference between the amount taken from the caller (`amountInMax`) and the actual swapped amount (`amounts[0]`) is not given back to the caller and remains locked in the contract.

Impact

Any user of the `swapTokensForExactETH` will always pay `amountInMax` for swaps even if part of it was not used for the swap. This part is lost, locked in the `V3Proxy` contract.

Proof of Concept

- Call `swapTokensForExactETH` with an excessively high `amountInMax`
- Check that any extra input tokens are sent back - this check will fail

```
function testV3ProxyKeepsTheChange() public {
    IQuoter q = IQuoter(0xb27308f9F90D607463bb33eA1BeBb41C27
    ISwapRouter r = ISwapRouter(0xE592427A0AEce92De3Edee1F18

    V3Proxy v3proxy = new V3Proxy(r, q, 500);
    vm.label(address(v3proxy), "V3Proxy");

    address[] memory path = new address[](2);
    path[0] = address(USDC);
    path[1] = address(WETH);

    address[] memory path2 = new address[](2);
    path2[0] = address(WETH);
```

```
        path2[1] = address(USDC);


        // fund Alice
        vm.prank(tokenWhale);
        USDC.transfer(alice, 1870e6);

        // Alice initiates a swap
        uint256[] memory amounts;
        uint256 balanceUsdcBefore = USDC.balanceOf(alice);
        uint256 balanceBefore = alice.balance;
        vm.startPrank(alice);
        USDC.approve(address(v3proxy), 1870e6);
        amounts = v3proxy.swapTokensForExactETH(1e18, 1870e6, pa

        // we check if the swap was done well
        require(amounts[0] < 1870e6);
        require(amounts[1] == 1e18);
        require(alice.balance == balanceBefore + amounts[1]);
        // the following check fails, but would pass if swapToke
        // sent back the excess tokens
        require(USDC.balanceOf(alice) == balanceUsdcBefore - amo
            "Unused input tokens were not sent back!");
    }
```

Recommended Mitigation Steps

Send back the excess tokens:

```
    function swapTokensForExactETH(uint amountOut, uint amountIn
        require(path.length == 2, "Direct swap only");
        require(path[1] == ROUTER.WETH9(), "Invalid path");
        ERC20 ogInAsset = ERC20(path[0]);
        ogInAsset.safeTransferFrom(msg.sender, address(this), am
        ogInAsset.safeApprove(address(ROUTER), amountInMax);
        amounts = new uint[](2);
        amounts[0] = ROUTER.exactOutputSingle(ISwapRouter.ExactO
        amounts[1] = amountOut;
        ogInAsset.safeApprove(address(ROUTER), 0);
        IWETH9 weth = IWETH9(ROUTER.WETH9());
        acceptPayable = true;
        weth.withdraw(amountOut);
        acceptPayable = false;
        payable(msg.sender).call{value: amountOut}("");
```

```
+           ogInAsset.safeTransfer(msg.sender, amountInMax - amount
            emit Swap(msg.sender, path[0], path[1], amounts[0], amou
        }
```

Keref (Good Entry) confirmed and commented:

> See PR#2.

Good Entry Mitigated:

> Send back unused funds to user.
> PR: https://github.com/GoodEntry-io/ge/pull/2

Status: Mitigation confirmed. Full details in reports from kutugu, xuwinnie and 3docSec.

[H-06] Incorrect Solidity version in FullMath.sol can cause permanent freezing of assets for arithmetic underflow-induced revert

Submitted by 3docSec, *also found by Vagner (*1, 2*),* hassan-truscova, *R-Nemes (*1, 2*),* auditsea, nadin, *and* n1punp

https://github.com/code-423n4/2023-08-goodentry/blob/71c0c0eca8af957202ccdbf5ce2f2a514ffe2e24/contracts/TokenisableRange.sol#L227

https://github.com/code-423n4/2023-08-goodentry/blob/71c0c0eca8af957202ccdbf5ce2f2a514ffe2e24/contracts/TokenisableRange.sol#L227

https://github.com/code-423n4/2023-08-goodentry/blob/71c0c0eca8af957202ccdbf5ce2f2a514ffe2e24/contracts/TokenisableRange.sol#L240

https://github.com/code-423n4/2023-08-goodentry/blob/71c0c0eca8af957202ccdbf5ce2f2a514ffe2e24/contracts/TokenisableRange.sol#L187

https://github.com/code-423n4/2023-08-goodentry/blob/71c0c0eca8af957202ccdbf5ce2f2a514ffe2e24/contracts/TokenisableRange.sol#L338

https://github.com/code-423n4/2023-08-goodentry/blob/71c0c0eca8af957202ccdbf5ce2f2a514ffe2e24/contracts/lib/FullMath.sol#L2

Vulnerability details

`TokenisableRange` makes use of the `LiquidityAmounts.getAmountsForLiquidity` helper function in its `returnExpectedBalanceWithoutFees`, `getTokenAmountsExcludingFees` and `deposit` functions to convert UniswapV3 pool liquidity into estimated underlying token amounts.

This function `getAmountsForLiquidity` will trigger an arithmetic underflow whenever `sqrtRatioX96` is smaller than `sqrtRatioAX96`, causing these functions to revert until this ratio comes back in range and the math no longer overflows.

Such oracle price conditions are not only possible but also likely to happen in real market conditions, and they can be permanent (i.e. one asset permanently appreciating over the other one).

Moving up the stack, assuming that `LiquidityAmounts.getAmountsForLiquidity` can revert (which is shown in the below PoC with real-world conditions), both the `returnExpectedBalanceWithoutFees` and `getTokenAmountsExcludingFees` functions can revert. In particular, the former **is called by the** `claimFee()` function, which is always called when **depositing** and **withdrawing** liquidity.

The root cause of this issue is that the FullMath.sol library, **imported from UniswapV3** was **altered to build with solidity v0.8.x**, which has under/overflow protection; the library, however, makes use of these by design, so it won't work properly when compiled in v0.8.0 or later:

```
/// @dev Handles "phantom overflow" i.e., allows multiplication a
library FullMath {
```

Impact

When the fair exchange price of the pool backing the TokenisableRange's falls outside the range (higher side), the `deposit` and `withdraw` will always revert, locking the underlying assets in the pool until the price swings to a different value that does not trigger an under/overflow. If the oracle price stays within this range indefinitely, the funds are permanently locked.

Proof of Concept

I'll prove that permanent freezing can happen in two steps:

- first I'll show one condition where the underflow happens
- then, I'll set up a fuzz test to prove that given an A and B ticker, we cannot find a market price lower than A such that the underflow does not happen

The most simple way to prove the first point is by calling `LiquidityAmounts.getAmountsForLiquidity` in isolation with real-world values:

```
    function testGetAmountsForLiquidityRevert() public {
        // real-world value: it's in fact the value returned by
        // V3_FACTORY.getPool(USDC, WETH, 500).slot0();
        // at block 17811921; it is around 1870 USDC per WETH
        uint160 sqrtRatioX96 = 1834502451234584391374419429242401

        // start price and end corresponding to 1700 to 1800 USDC
        uint160 sqrtRatioAX96 = 1866972058592130739290643700340901
        uint160 sqrtRatioBX96 = 1921904167735311150677430952623401

        vm.expectRevert();
        LiquidityAmounts.getAmountsForLiquidity(sqrtRatioX96, sqr
    }
```

However, a more integrated test that involves PositionManager can also be considered:

```
    function testPocReturnExpectedBalanceUnderflow() public {
        vm.createSelectFork(
            "mainnet",
            17811921
```

```
        );
        vm.startPrank(tokenWhale);
        TokenisableRange tr = new TokenisableRange();
        tr.initProxy(AaveOracle, USDC, WETH, 1700e10, 1800e10, "
        USDC.approve(address(tr), 100_000e6);
        tr.init(100_000e6, 0);
        vm.expectRevert();
        tr.returnExpectedBalance(0, 0);
    }
```

Then, we can prove the second point with a negative fuzz test:

```
    function testFailPermanentFreeze(uint160 sqrtRatioX96) publi
        // start & and price, corresponding to 1700 to 1800 USDC
        uint160 sqrtRatioAX96 = 18669720585921307392906437003409
        uint160 sqrtRatioBX96 = 19219041677353111506774309526234

        // make sure that the market ratio is lower than the lowe
        // that is the range where I first observed the underflo
        // (WETH above 1800 USDC)
        sqrtRatioX96 = sqrtRatioX96 % (sqrtRatioAX96 - 1);

        // expect a revert here
        LiquidityAmounts.getAmountsForLiquidity(sqrtRatioX96, sq
    }
```

Tools Used

IDE, Foundry

Recommended Mitigation Steps

Restore [the original FullMath.sol library] so it compiles with solc versions earlier than 0.8.0.

```
// SPDX-License-Identifier: GPL-3.0
- pragma solidity ^0.8.4;
+ pragma solidity >=0.4.0 <0.8.0;

/// @title Contains 512-bit math functions
/// @notice Facilitates multiplication and division that can have
```

```
/// @dev Handles "phantom overflow" i.e., allows multiplication a
```

Another possible option, which is however not recommended, is to enclose the non-assembly statements of FullMath.sol in an `unchecked` block.

Keref (Good Entry) confirmed and commented:

> There's an error with those lib versions, and we will replace with libs from the 0.8 branch.

Good Entry Mitigated:

> Use correct Uniswap for sol `^0.8` libs.
> PR: https://github.com/GoodEntry-io/ge/commit/8b0feaec0005937c8e6c7ef9bf039a0c2498529a

Status: Mitigation confirmed. Full details in reports from kutugu, xuwinnie and 3docSec.

Medium Risk Findings (8)

[M-01] V3 Proxy does not send funds to the recipient, instead it sends to the msg.sender

Submitted by josephdara, *also found by* 3docSec

The functions above can be used to swap tokens, however the swaps are not sent to the provided address. Instead they are sent to the msg.sender. This could cause issues if the user has been blacklisted on a token. Or if the user has a compromised signature/allowance of the target token and they attempt to swap to the token, the user loses all value even though they provided an destination address.

Proof of Concept

```
//@audit-H does not send tokens to the required address
```

```
    function swapExactTokensForTokens(uint amountIn, uint amount
        require(path.length == 2, "Direct swap only");
        ERC20 ogInAsset = ERC20(path[0]);
        ogInAsset.safeTransferFrom(msg.sender, address(this), am
        ogInAsset.safeApprove(address(ROUTER), amountIn);
        amounts = new uint[](2);
        amounts[0] = amountIn;
//@audit-issue it should be the to address not msg.sender
        amounts[1] = ROUTER.exactInputSingle(ISwapRouter.ExactIn
        ogInAsset.safeApprove(address(ROUTER), 0);
        emit Swap(msg.sender, path[0], path[1], amounts[0], amou
    }
```

Here is one of the many functions with this issue, as we can see after the swap is completed, tokens are sent back to the msg.sender from the router not to the `to` address.

Tools Used

Manual Review.
Uniswap Router: https://github.com/Uniswap/v3-periphery/blob/main/contracts/SwapRouter.sol

Recommended Mitigation Steps

The uniswap router supports inputting of a destination address. Hence the router should be called with the to address not the msg.sender.

Else remove the `address to` from the parameter list.

Keref (Good Entry) confirmed, but disagreed with severity and commented:

> Cannot remove `to` from param list as the stated goal is to be compatible with uniswap v2 interface.

> Although it looks like a severe issue, in our case this was actually on purpose as this is a compatibility module used for OPM only, where it's always `msg.sender == to`, and to avoid potential spam under GE banner (ppl swapping spam tokens to famous addresses).

> So it's not a high risk issue, maybe low only.

gzeon (Judge) decreased severity to Medium and commented:

> Recommend to restrict caller to OPM.

Keref (Good Entry) commented:

> We could have several (out of this audit scope) contracts that need to swap, restricting to msg.sender is enough.

gzeon (Judge) commented:

> Right, so maybe checking msg.sender == to ?

Good Entry Mitigated:

> Added explicit require `msg.sender == to`.
> PR: https://github.com/GoodEntry-io/ge/pull/10

Status: Mitigation confirmed. Full details in reports from kutugu, xuwinnie and 3docSec.

[M-02] Incorrect parameters passed to UniV3 may cause funds stuck in the vault

Submitted by libratus

Note: this issue happened on the deployed version of GoodEntry and was discovered when using https://alpha.goodentry.io

Due to incorrect parameters and validation when working with UniV3 LP the vault may enter a state where rebalancing reverts. This means any deposits and withdrawals from the vault become unavailable.

Code walkthrough

When rebalancing a vault, the existing positions need to be removed from Uni. This is done in removeFromTick function.

```
if (aBal > 0){
  lendingPool.withdraw(address(tr), aBal, address(this));
  tr.withdraw(aBal, 0, 0);
}
```

Here, zeros are passed as `amount0Min` and `amount1Min` arguments. The execution continues in [TokenisableRange.withdraw] function. `decreaseLiquidity` is called to remove liquidity from Uni.

```
(removed0, removed1) = POS_MGR.decreaseLiquidity(
  INonfungiblePositionManager.DecreaseLiquidityParams({
    tokenId: tokenId,
    liquidity: uint128(removedLiquidity),
    amount0Min: amount0Min,
    amount1Min: amount1Min,
    deadline: block.timestamp
  })
);
```

Here there is an edge-case that for really small change in liquidity the returned values `removed0` and `removed1` can be 0s (will be explained at the end of a section).

Then, `collect` is called and `removed0` , `removed1` are passed as arguments.

```
POS_MGR.collect(
  INonfungiblePositionManager.CollectParams({
    tokenId: tokenId,
    recipient: msg.sender,
    amount0Max: uint128(removed0),
    amount1Max: uint128(removed1)
  })
);
```

However, `collect` reverts when both of these values are zeros - https://github.com/Uniswap/v3-periphery/blob/main/contracts/NonfungiblePositionManager.sol#L316

As a result, any deposit/withdraw/rebalancing of the vault will revert when it will be attempting to remove existing liquidity.

When can decreaseLiquidity return 0s

This edge-case is possible to achieve as it happened in the currently deployed alpha version of the product. The sponsor confirmed that the code deployed is the same as presented for the audit.

The tick that caused the revert has less than a dollar of liquidity. Additionally, that tick has outstanding debt and so the `aBal` value was small enough to cause the issue. In the scenario that happened on-chain aBal is only *33446*.

```
        uint aBal = ERC20(aTokenAddress).balanceOf(address(this));
        uint sBal = tr.balanceOf(aTokenAddress);

        // if there are less tokens available than the balance (beca
        if (aBal > sBal) aBal = sBal;
        if (aBal > 0){
          lendingPool.withdraw(address(tr), aBal, address(this));
          tr.withdraw(aBal, 0, 0);
        }
```

Proof of Concept

The issue can be demonstrated on the contracts deployed on the arbitrum mainnet. This is the foundry test:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";

interface IGeVault {
    function withdraw(uint liquidity, address token) external re
}

contract GeVaultTest is Test {
    IGeVault public vault;

    function setUp() public {
```

```
        vault = IGeVault(0xdcc16DEfe27cd4c455e5520550123B4054D1b
        // 0xdcc16DEfe27cd4c455e5520550123B4054D1b432 btc vault
    }

    function testWithdraw() public {
        // Account that has a position in the vault
        vm.prank(0x461F5f86026961Ee7098810CC7Ec07874077ACE6);

        // Trying to withdraw 1 USDC
        vault.withdraw(1e6, 0xFF970A61A04b1cA14834A43f5dE4533eBD
    }
}
```

The test can be executed by forking the arbitrum mainnet

```
forge test -vvv --fork-url <your_arb_rpc> --fork-block-number 11
```

The result is an error in UniV3 `NonfungiblePosition.collect` method

```
    |   |   |   ├─ [1916] 0xC36442b4a4522E871399CD717aBDD847Ab11F
    |   |   |   |   └─ ← "EvmError: Revert"
```

Recommended Mitigation Steps

It is unclear why this `collect` call is needed because the fees are already collected a few lines above in `claimFees`. I suggest removing the second `collect` call altogether. If it's needed then perhaps only collect if one of removed0/removed1 is non-zero.

gzeon (Judge) commented:

> The writeup looks correct but it seems to be quite an edge case and therefore the risk is low.
> But @Keref can you review?

Keref (Good Entry) commented:

> Yes, it was confirmed. The auditor also contacted me in private and we solved the bug, checking that it didn't try to collect (0, 0).

gzeon (Judge) commented:

> Cool, I believe severity to be Medium as this clearly affected the availability of the protocol. However, I am not fully convinced that this will make the asset stuck permanently without an upgrade. For example, it seems to be possible to borrow everything from the lendingpool so that aBal is set to 0 and the withdraw will be skipped.

libratus (Warden) commented:

> Yes, it might be correct that there are ways to unstuck the funds without an upgrade.

Good Entry Mitigated:

> Prevent collect from reverting by adding a check that it doesnt try to collect `0`.
> PR: https://github.com/GoodEntry-io/ge/commit/bbbac57c110223f45851494971a34f57c55922c7

Status: Mitigation confirmed. Full details in reports from kutugu, xuwinnie and 3docSec.

[M-03] Incorrect boundaries check in GeVault's `getActiveTickIndex` can temporarily freeze assets due to Index out of bounds error

Submitted by 3docSec

https://github.com/code-423n4/2023-08-goodentry/blob/71c0c0eca8af957202ccdbf5ce2f2a514ffe2e24/contracts/GeVault.sol#L431

https://github.com/code-423n4/2023-08-goodentry/blob/71c0c0eca8af957202ccdbf5ce2f2a514ffe2e24/contracts/GeVault.

sol#L357-L358

`GeVault` stores the `TokenisableRange` instances it operates on in an ordered array:

```
TokenisableRange[] public ticks;
```

Whenever a rebalancing happens, the `GeVault` contract withdraws all its liquidity and redeploys it on at most four ticks, starting from (and including) the one identified by the `getActiveTickIndex` function:

```
function deployAssets() internal {
  uint newTickIndex = getActiveTickIndex();
  // [...]
  uint tick0Index = newTickIndex;
  uint tick1Index = newTickIndex + 2;
  // [...]
    depositAndStash(ticks[tick0Index], availToken0 / 2, 0);
    depositAndStash(ticks[tick0Index+1], availToken0 / 2, 0);
  // [...]
    depositAndStash(ticks[tick1Index], 0, availToken1 / 2);
    depositAndStash(ticks[tick1Index+1], 0, availToken1 / 2);
```

However, the `getActiveTickIndex()`, given that the termination condition of its `for` loop, can return indices up to `ticks.length - 3`, included (**because the increment of `activeTickIndex` that made the boundary check fail is kept**):

```
/// @notice Return first valid tick
function getActiveTickIndex() public view returns (uint active'
  if (ticks.length >= 5){
    // looking for index at which the underlying asset differs
    for (activeTickIndex = 0; activeTickIndex < ticks.length -
      // [...]
      if ( /* ... */ )
        break;
    }
  }
}
```

So the highest value it can possibly return is `ticks.length - 3`. If we take this value and project where `rebalance()` will deploy assets, we'll have the ticks at the four indices: `[ticks.length - 3, ticks.length - 2, ticks.length - 1, ticks.length]`, and the last value will overflow, causing the rebalancing, and the liquidity operation that triggered it (if any) to fail.

Impact

Whenever the market is such that the `getActiveTickIndex` returns the last possible index, the contract will revert on any `rebalance`, `deposit`, and more importantly `withdraw` operations. Despite the impact including locking assets, this finding is reported as medium severity because the protocol governance could resolve the situation by adding extra ticks & rescue the assets without requiring a contract code upgrade.

Proof of Concept

I have a running PoC in my environment, which I will keep aside and will be happy to provide if requested, but I would rather not share it because it's a monstrous setup with a couple of workarounds to not make it even worse.

High level my setup is:

- set up a GeVault with 6 ranges, most of which are mostly at higher prices than the market:
    - range at index 0 at 3001-3250 (implicit, e10)
    - 1 at 2751-3000
    - 2 at 2501-2750
    - 3 at 2251-2500
    - 4 at 2001-2250
    - 5 at 1751-2000
    - with USDC/WETH at 1870 (I forked mainnet at block 17811921)
    - call `getActiveTickIndex()` which will return `3` (higher than 2, which is the max value that would not make the fund deployment go out of range)
    - deposit some WETH - this will revert out of bounds

Tools Used

IDE, foundry

Recommended Mitigation Steps

Decrease by 1 the loop boundaries:

```
    /// @notice Return first valid tick
    function getActiveTickIndex() public view returns (uint active'
      if (ticks.length >= 5){
        // looking for index at which the underlying asset differs
-       for (activeTickIndex = 0; activeTickIndex < ticks.length -
+       for (activeTickIndex = 0; activeTickIndex < ticks.length -

          (uint amt0, uint amt1) = ticks[activeTickIndex+1].getTok
          (uint amt0n, uint amt1n) = ticks[activeTickIndex+2].getT
          if ( (amt0 == 0 && amt0n > 0) || (amt1 == 0 && amt1n > 0
            break;
        }
      }
    }
```

Or use a dedicated loop variable, and explicitly return the correct value.

Keref (Good Entry) disputed and commented:

> `getActiveTickIndex` goal is to return a valid index if it exists, independently of whether the rest of the tx will succeed or fail bc there arent enough ticks.

gzeon (Judge) invalidated the finding

3docSec (Warden) commented:

> Hi,

> The sponsor's comment confirms that `getActiveTickIndex` returns an index that may not have enough ticks after it, which is the root cause of this finding.

> Therefore, I believe both the finding's description and availability impact on GeVault's `rebalance`, `deposit`, and `withdraw` remain valid, so I would ask to kindly reconsider the invalidation of this finding.

> For the sake of accurate reporting and in accordance with sponsor's intention that was now made clear, I would recommend a different mitigation than what was initially reported - protecting the `GeVault.deployAssets()` code that uses the active tick index from the out-of-bounds error reported in this finding:

```
      if (availToken0 > 0){
        depositAndStash(ticks[tick0Index], availToken0 / 2, 0);
        depositAndStash(ticks[tick0Index+1], availToken0 / 2, 0);
      }
      if (availToken1 > 0){
        depositAndStash(ticks[tick1Index], 0, availToken1 / 2);
+      if (tick1Index+1 < ticks.length)
        depositAndStash(ticks[tick1Index+1], 0, availToken1 / 2);
      }
```

Keref (Good Entry) commented:

> I guess it indeed makes sense as QA to either just skip depositing in the tick or to revert with an explicit rather than ouf of range error.

3docsec (Warden) commented:

> Yep, but that would not avoid the following case:
>
> - users successfully deposit when the last ticks are not selected
> - the market changes asset prices towards the last ticks
> - users try to withdraw and fail because of out-of-bounds error; having an explicit error preventing withdrawal would not make them any happier
>
> Happy to provide more context if needed.

Keref (Good Entry) acknowledged and commented:

> Code has been changed (substantially) to avoid those errors. See PR#11.

gzeon (Judge) validated the finding and commented:

> Looks valid, I missed the fact that this would prevent withdrawal in the original judging.

Good Entry Mitigated:

> Reworked `activeTickIndex` as per desc above.
> PR: https://github.com/GoodEntry-io/ge/pull/11

Status: Mitigation error. Full details in reports from **kutugu**, **xuwinnie** and **3docSec**, and also included in the **Mitigation Review** section below.

[M-04] First depositor can break minting of liquidity shares in GeVault

Submitted by **nemveer**, *also found by* **0xBeirao**, **Hama**, **n33k**, *and* **Madalad**

https://github.com/code-423n4/2023-08-goodentry/blob/71c0c0eca8af957202ccdbf5ce2f2a514ffe2e24/contracts/GeVault.sol#L271-L278

https://github.com/code-423n4/2023-08-goodentry/blob/71c0c0eca8af957202ccdbf5ce2f2a514ffe2e24/contracts/GeVault.sol#L420-L424

In **GeVault**, while depositing tokens in the pool, liquidity tokens are minted to the users.

Calculation of liquidity tokens to mint uses `balanceOf(address(this))` which makes it susceptible to first deposit share price manipulation attack.

`deposit` calls `getTVL`, which calls `getTickBalance`.

GeVault.deposit#L271-L278

```
        uint vaultValueX8 = getTVL();
        uint tSupply = totalSupply();
        // initial liquidity at 1e18 token ~ $1
        if (tSupply == 0 || vaultValueX8 == 0)
```

```
    liquidity = valueX8 * 1e10;
  else {
    liquidity = tSupply * valueX8 / vaultValueX8;
  }
```

GeVault.getTVL#L392-L398

```
function getTVL() public view returns (uint valueX8){
  for(uint k=0; k<ticks.length; k++){
    TokenisableRange t = ticks[k];
    uint bal = getTickBalance(k);
    valueX8 += bal * t.latestAnswer() / 1e18;
  }
}
```

GeVault.getTickBalance#L420-L424

```
function getTickBalance(uint index) public view returns (uint
  TokenisableRange t = ticks[index];
  address aTokenAddress = lendingPool.getReserveData(address(t
  liquidity = ERC20(aTokenAddress).balanceOf(address(this));
}
```

Although there is a condition on line `281` that liquidity to be minted must be greater than 0, User's funds can be at risk.

Proof of Concept

When totalSupply is zero, an attacker can go ahead and execute following steps.

1. Calls deposit function with 1 wei amount of underlying as argument. To that, he will be minted some amount of liquidity share depending on the price of underlying.
2. Withdraw all except one wei of shares.
3. Transfer some X amount of underlying directly to pool contract address.

So, now 1 wei of share worths X underlying tokens. Attacker won't have any problem making this X as big as possible. Because he'll always be able to redeem it with 1 wei

of share.

Impact

1. Almost 1/4th of first deposit can be frontrun and stolen.
2. Let's assume there is a first user trying to deposit with z dollars worth of tokens
3. An attacker can see this transaction in mempool and carry out the above-described attack with x = (z/2 + 1).
4. This means the user gets 1 Wei of share which is only worth ~ 3x/4 of tokens.
5. Here, the percentage of the user funds lost depends on how much capital the attacker has. let's say a attacker keeps 2 wei in the share initially instead of 1 (this makes doubles capital requirement), they can get away with 33% of the user's funds.
6. DOS to users who tries to deposit less than X because of this check

```
require(liquidity > 0, "GEV: No Liquidity Added");
```

Recommended Mitigation Steps

Burn some MINIMUM_LIQUIDITY during first deposit.

Keref (Judge) confirmed, but disagreed with severity and commented:

> Issue is medium severity as easily preventable and only affects GE team when deploying a new vault.

gzeon (Judge) decreased severity to Medium

[M-05] addDust does not achieve the goal correctly and may overflow revert

Submitted by kutugu

The purpose of addDust is to ensure that both the token0Amount and token1Amount are greater than 100 units. The current implementation is to calculate the value of 100

units 1e18 scales of token0 and token1, take the maximum value as liquidity, and add to the repayAmount.

The calculation does not take into account the actual getTokenAmounts result:

1. The calculated dust amount is based on the oracle price, while the actual amount consumed is based on the lp tick price
2. Even if the price of lp tick is equal to the spot price, which can't guarantee that the token0Amount and token1Amount of getTokenAmounts will be greater than 100 units.

Proof of Concept

▶ Details

Tools Used

Foundry

Recommended Mitigation Steps

Check the amount of token0Amount and token1Amount corresponding to repayAmount instead of adding dust manually

Keref (Good Entry) acknowledged and commented:

> There is a misunderstanding here, in the case of price being below a tick no amount of dust will bring token1 above 0. In that case we want the active token to be above 100.
> The reason is, as currently is, depositing liquidity may return less than the expected liquidity, due to rounding. We want to add dust such that any necessary token is in enough excess that the liquidity recreated repays the debt fully.

> However we agree that we've approached the problem in a wrong way and will modify the TokenisableRange so that it supports depositing an exact liquidity, and not care about that in the OPM.

Good Entry Mitigated:

> Removed `addDust` mechanism, replaced by `depositExactly` in TR.
> PR: https://github.com/GoodEntry-io/ge/pull/8

Status: Mitigation confirmed. Full details in reports from kutugu and 3docSec.

[M-06] User can steal refunded underlying tokens from `initRange` operation inside `RangeManager`

Submitted by said, *also found by* pep7siup, *oakcobalt (*1, 2*)*, Jeiwan, SpicyMeatball, jesusrod15, giovannidisiena, HChang26, *and* 3docSec

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/RangeManager.sol#L95-L102

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/TokenisableRange.sol#L134-L163

After the owner of `RangeManager` create new range via `generateRange` , they can then call `initRange` to init the range and providing the initial underlying tokens for initial uniswap v3 mint amounts. However, after operation the refunded underlying tokens is not send back to the owner, this will allow user to steal this token by triggering `cleanup()` .

Proof of Concept

▶ Details

Recommended Mitigation Steps

Consider to add `cleanup` after the `initRange` call :

```
    function initRange(address tr, uint amount0, uint amount1) ext
      ASSET_0.safeTransferFrom(msg.sender, address(this), amount0)
      ASSET_0.safeIncreaseAllowance(tr, amount0);
      ASSET_1.safeTransferFrom(msg.sender, address(this), amount1)
      ASSET_1.safeIncreaseAllowance(tr, amount1);
      TokenisableRange(tr).init(amount0, amount1);
      ERC20(tr).safeTransfer(msg.sender, TokenisableRange(tr).bala
+     cleanup();
```

```
        }
```

Keref (Good Entry) confirmed, but disagreed with severity and commented:

> The funds deposited are used to create the NFT and to avoid the first depositor attack.
> Because the TR can never have 0 assets or they are considered closed, the funds put there are somehow already at a loss, the dust returned on depositing is just dust.
> But it's a good catch. Would say low but bc some (dust) funds are at risk maybe medium?

Keref (Good Entry) commented:

> See patch pull#1.

gzeon (Judge) decreased severity to Medium and commented:

> Considered downgrade to low given the negligible leak.

Good Entry Mitigated:

> Added return value check.
> PR: https://github.com/GoodEntry-io/ge/pull/3

Status: Mitigation confirmed. Full details in reports from kutugu, xuwinnie and 3docSec.

[M-07] Incorrect calculations in deposit() function in TokenisableRange.sol can make the users suffer from immediate loss

Submitted by 0xDING99YA

Calculations of uncompounded fee in deposit() in TokenisableRange.sol is incorrect, this can make a potential immediate fund loss after a user make a deposit.

Proof of Concept

Generally speaking, functions in a protocol should be designed symmetrically. For example in a DEX, when swap X to Y and then immediately swap Y to X, when excluding fees, the user won't loss any funds. In this TokenisableRange.sol case, when a user deposits some funds, if the user withdraw it immediately, and when there is no market condition change and exclude fees, the amount the user can obtained should be same as the deposit value.

However, this is not the case in deposit() function. In deposit(), a user will input the amount he/she want to deposit, a uncompound fee may subtracted from that amount, and the rest of the amount will be deposited, and LP tokens will be transferred to the user. When withdraw, the user burn the LP token, and obtain the corresponding liquidity and uncompounded fee. As stated above, when no market change and exclude any fees, the liquidity and uncompounded fee a user can get should be the same as paid in deposit(). The issue is in these lines:

```
if (fee0 + fee1 > 0 && (n0 > 0 || fee0 == 0) && (n1 > 0 || fee1 =
        address pool = V3_FACTORY.getPool(
            address(TOKEN0.token),
            address(TOKEN1.token),
            feeTier * 100
        );
        (uint160 sqrtPriceX96, , , , , , ) = IUniswapV3Pool(pool
        (uint256 token0Amount, uint256 token1Amount) = Liquidity
            .getAmountsForLiquidity(
                sqrtPriceX96,
                TickMath.getSqrtRatioAtTick(lowerTick),
                TickMath.getSqrtRatioAtTick(upperTick),
                liquidity
            );
        if (token0Amount + fee0 > 0)
            newFee0 = (n0 * fee0) / (token0Amount + fee0);
        if (token1Amount + fee1 > 0)
            newFee1 = (n1 * fee1) / (token1Amount + fee1);
        fee0 += newFee0;
        fee1 += newFee1;
        n0 -= newFee0;
        n1 -= newFee1;
    }
```

The issue here is, the deducted uncompounded fee newFee0 and newFee1 are computed based on users input n0 and n1, but the user input n0 and n1 may not be as the same ratio correspond to the current market ratio. So in conclusion, the uncompounded fees is computed based on user input n0 and n1, the ratio between that n0 and n1 may not be the current ratio under current market condition, but later the actual deposit amounts are based on the current ratio and the minted LP token also based on that ratio, the current ratio is also used in withdraw, so it is this difference that result in a potential loss in uncompounded fee.

This may not obvious so let's look at an example with solid numbers.

We assume such a condition. User input n0 = 415, n1 = 100, uncompounded fee0 f0 = 20, fee1 f1 = 5, under current market price and tick range token0Amount t0 = 4000, token1Amount t1 = 1000, t0 and t1 correspond to a liquidity L of 1000, and LP token total supply T is 2000.

Since f0 + f1 = 25 > 0 && n0 = 415 > 0 && n1 = 100 > 0, we will enter the first if statement to compute the deducted uncompounded fee. newFee0 = 415*20/(4000+20) = 415/201, newFee1 = 100*5/(1000+5) = 100/201, updated n0 = 415 - 415/201 = 412.9353234, updated n1 = 100 - 100/201 = 20000/201.

We will then make the deposit. Under current assumed market condition we can deposit n1 of 20000/201 and n0 of 80000/201 and obtain a new liquidity newL of 20000/201. The LP token amount we can get is newL/L*T = 40000/201. Note here n1 will all be deposited, deposited n0 is 80000/201 , which is larger than 95% of (n1 - newFee0), which is 95% of 412.9353234. So slippage check is satisfied.

In conclusion, in this deposit(), user specified n0 of 415 and n1 of 100, user have an actual deposit of n0 = 80000/201 and n1 of 20000/201, this corresponding to a liquidity of 20000/201, user also deposit a uncompounded fee, newFee0 is 415/201 and newFee1 is 100/201. User get back 40000/201 LP token. Now updated fee0 = 20 + 415/201 = 1475/67, and updated fee1 = 5 + 100/201 = 1105/201. Updated total liquidity is 1000 + 20000/201 = 1099.502488, and updated LP token supply is 442000/201.

Now user wants to withdraw. He will burn all his LP token, so the removedLiquidity he can get is (40000/201)/(442000/201)*1099.502488 = 20000/201, this is same as the liquidity got in deposit(). For uncompounded fee, obtained fee0 = (40000/201)/(442000/201)*1475/67 = 1.9923, obtained fee1 =

(40000/201)/(442000/201)*1105/201 = 100/201. We can see the fee1 got back is exactly the same, but the fee0 different, deposited 415/201 = 2.0647 but only get back 1.9923, so the user will incur a loss.

As mentioned before, the loss in uncompounded fee here is due to use user input n0 and n1 here, which may not be proportional to the actual deposit amount.

Tools Used

VS Code

Recommended Mitigation Steps

The protocol should first calculate a proportioned n0 and n1 based on user inputs, then compute uncompounded fee based on that. Take the example above, user inputs n0 = 415 and n1 = 100, the protocol should calculate that the proportioned n0 = 400 and n1 = 100 under current condition. Then the fee should be calculated based on the n0 of 400 and n1 of 100.

Keref (Good Entry) acknowledged and commented:

> That portion of code was completely removed as it caused several issues. See PR#8.

Good Entry Mitigated:

> Removed `addDust` mechanism, replaced by `depositExactly` in TR.
> PR: https://github.com/GoodEntry-io/ge/pull/8

Status: Mitigation confirmed. Full details in reports from kutugu, xuwinnie and 3docSec.

[M-08] Return value of low level `call` not checked

Submitted by dd0x7e8, *also found by* Udsen, SanketKogekar, pep7siup, hpsb, parsely, josephdara, Fulum, Sathish9098, shirochan, MatricksDeCoder, Kaysoft, ravikiranweb3, Bughunter101, j4ld1na, T1MOH, fatherOfBlocks, grearlake, debo, *and* piyushshukla

https://github.com/code-423n4/2023-08-goodentry/blob/71c0c0eca8af957202ccdbf5ce2f2a514ffe2e24/contracts/helper/V3Proxy.sol#L156

https://github.com/code-423n4/2023-08-goodentry/blob/71c0c0eca8af957202ccdbf5ce2f2a514ffe2e24/contracts/helper/V3Proxy.sol#L174

https://github.com/code-423n4/2023-08-goodentry/blob/71c0c0eca8af957202ccdbf5ce2f2a514ffe2e24/contracts/helper/V3Proxy.sol#L192

Impact

If the `msg.sender` is a contract and its `receive()` function has the potential to revert, the code `payable(msg.sender).call{value:wad}("")` could potentially return a false result, which is not being verified. As a result, the calling functions may exit without successfully returning ethers to senders.

Proof of Concept

https://github.com/code-423n4/2023-08-goodentry/blob/71c0c0eca8af957202ccdbf5ce2f2a514ffe2e24/contracts/helper/V3Proxy.sol#L147C1-L158C6

https://github.com/code-423n4/2023-08-goodentry/blob/71c0c0eca8af957202ccdbf5ce2f2a514ffe2e24/contracts/helper/V3Proxy.sol#L160C1-L176C6

https://github.com/code-423n4/2023-08-goodentry/blob/71c0c0eca8af957202ccdbf5ce2f2a514ffe2e24/contracts/helper/V3Proxy.sol#L178C1-L194C6

PoC using Foundry:

▶ Details

Recommended Mitigation Steps

It's recommended to check the return value to be true or just use OpenZeppelin `Address` library `sendValue()` function for ether transfer. See https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v4.9.3/contracts/utils/Address.sol#L64 .

Keref (Good Entry) confirmed and commented:

> See PR#3.

Status: Not fully mitigated. Full details in reports from kutugu and 3docSec, and also included in the Mitigation Review section below.

Low Risk and Non-Critical Issues

For this audit, 26 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **Team_FliBit** received the top score from the judge.

The following wardens also submitted reports: Sathish9098, Rolezn, 3docSec, oakcobalt, UniversalCrypto, pep7siup, osmanozdemir1, josephdara, niser93, hpsb, kutugu, nonseodion, LokiThe5th, digitizeworx, banpaleo5, catellatech, j4ld1na, DavidGiladi, sivanesh_808, 8olidity, said, ravikiranweb3, 0x70C9, fatherOfBlocks, *and* Krace.

[L-01] No function to remove a tick

Summary

In the `GeVault.sol` it is impossible to remove a `tick` once it is added to the `ticks[]`.

Vulnerability Details

There are methods to add, shift and modify ticks but none to remove ticks from the ticks array.

But if due to a human error a faulty tick is added, there won't be any way to remove it. It could only be overridden to 0 and moved left to avoid breaking other functions, a

non-ideal solution.

Therefore, it is advisable to implement another function to allow removing a tick.

Impact

It won't allow removing faulty added ticks, being the only solution to override them to 0 and move them to the left, to avoid breaking other functions.

Proof of Concept

- `pushTick` function: https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/contracts/GeVault.sol#L116
- `shiftTick` function: https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/contracts/GeVault.sol#L137
- `modifyTick` function: https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/contracts/GeVault.sol#L167

Recommended Mitigation Steps

Add a function to remove a faulty tick from the ticks array.

[L-02] Transactions calling addDust revert if token has more than 20 decimals

Summary

In the `addDust` function in the `OptionsPositionManager.sol` file, if `token0` or `token1` has more than 20 decimals, the transaction will always revert.

Vulnerability Details

The function executes some code to scale the tokens value:

```
uint scale0 = 10**(20 - ERC20(token0).decimals()) * oracle.getAs
```

```
uint scale1 = 10**(20 - ERC20(token1).decimals()) * oracle.getAs
```

But since they are doing `20 - token0Decimals` and `20 - token1Decimals`, if one of those tokens has more than 20 decimals, the subtraction will underflow causing the entire transaction to revert, making it impossible to run this function for that token pair.

Impact

The token pairs having at least one token with more than 20 decimals won't be able to be used in the `addDust` execution since those parameters will always cause the function to revert.

Proof of Concept

- Affected lines: https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/contracts/PositionManager/OptionsPositionManager.sol#L544-L551

Recommended Mitigation Steps

Since it is already doing some logic to support different token decimals, a conditional to check if the token decimals are higher than 20 could be implemented, and then some logic to support tokens with more than 20 decimals.

[L-03] Math always revert for tokens with 39 or more decimals

Summary

In the `initProxy` function of the `TokenisableRange.sol` contract, if `TOKEN0` or `TOKEN1` has 39 or more decimals, the math will overflow, causing the entire function to revert.

Vulnerability Details

In the `initProxy` function there is some math to calculate the upper and lower ticks used (https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/contracts/TokenisableRange.sol#L99-L100):

```
int24 _upperTick = TickMath.getTickAtSqrtRatio( uint160( 2**48
int24 _lowerTick = TickMath.getTickAtSqrtRatio( uint160( 2**
```

But if `TOKEN0.decimals` or `TOKEN1.decimals` have 39 or more decimals, the math will overflow causing the entire transaction to revert, making it impossible to initialize the contract for that token pair.

Impact

This makes it impossible to use this protocol feature with pairs that have at least one token with 39 or more decimals. Not being able to use some protocol features is a high-severity issue but since there are not many tokens using 39 or more decimals we set the impact to low.

Proof of Concept

- Lines where the issue happens: https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/contracts/TokenisableRange.sol#L99-L100

Just try running in Solidity the following math with the following params:

- `TOKEN1.decimals = 39`
- `TOKEN0.decimals = 18`
- `startX10 = 1e10` - Comments in the code explain that this is a value scaled by 1e10, so we can use 1e10 as a demo value

You can try the following contract:

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.8.2 <0.9.0;

contract TestContract {
    function sqrt(uint x) internal pure returns (uint y) {
      uint z = (x + 1) / 2;
      y = x;
      while (z < y) {
          y = z;
```

```
            z = (x / z + z) / 2;
        }
    }

    function testThatReverts() public pure {
        uint256 TOKEN1Decimals = 39;
        uint256 TOKEN0Decimals = 18;
        uint256 startX10 = 1e10;
        uint160( 2**48 * sqrt( (2 ** 96 * (10 ** TOKEN1Decimals)
    }

    function testThatWorks() public pure {
        uint256 TOKEN1Decimals = 38;
        uint256 TOKEN0Decimals = 18;
        uint256 startX10 = 1e10;
        uint160( 2**48 * sqrt( (2 ** 96 * (10 ** TOKEN1Decimals)
    }
}
```

Tools Used

Foundry fuzzing, and Remix.

Recommended Mitigation Steps

Restrict the decimals supported, change the way the math is done, or use a library like `PBRMath` to support bigger numbers without causing overflow (https://github.com/PaulRBerg/prb-math).

[N-01] Sqrt function does not work in every case

Summary

The `sqrt` function used in the project fails due to overflow in some cases, making it suboptimal to use.

Vulnerability Details

The `sqrt` function defined here and also here (which by the way it should always use the second, instead of having duplicated code in other contracts) does not work in every number in the uint256 range.

After doing some fuzzy testing we found it fails when doing the square root of the max uint256 number (`2**256 -1`). That overflow happens because inside the `sqrt` function, they sum 1 to the entered argument value and since we entered max uint256, you can not add 1 to that value without causing an overflow.

It is clear this is an edge case but a proper `sqrt` function defined to accept a `uint256` input should support every possible value inside the supported input type. In this case, `sqrt` should support any value from `0` to `2**256-1` , which is not the case.

Impact

Suboptimal input type support can cause unexpected reverts when trying to do the `sqrt` of some inputs.

Proof of Concept

- Case 1: https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/contracts/TokenisableRange.sol#L66
- Case 2: https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/contracts/lib/Sqrt.sol#L8
- Cause of the overflow with an input value of `2**256-1` : https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/contracts/lib/Sqrt.sol#L9

Tools Used

Fuzzy testing with Foundry.

Recommended Mitigation Steps

Our recommendation is to use another `sqrt` function. There are two battle-tested options that are the market standards:

- One would be the one used by Uniswap V2, which supports `2**256-1` without reverting: https://ethereum.stackexchange.com/a/87713

- The other one would be using the popular `PBRMath` library, which has a `sqrt` method that also supports every input value: https://github.com/PaulRBerg/prb-math

[N-02] Missing check to make sure added tokens corresponds to the oracle/pool

Summary

In the `LPOracle.sol` and `RoeRouter.sol` files, there are functions that don't verify if the tokens sent in the params are part of the oracle/pool address also sent in the params. That can allow human mistakes that lead to wrong price calculations.

Vulnerability Details

There is no check to make sure that the tokens constituting the LP/oracle are the same as the chainlink price tokens. Due to human error, a mistake can be made and you end up with a price for token 0 with decimals of token 1, which would completely destroy any price calculations.

Impact

Potential wrong price calculations if the wrong tokens or the right tokens but in a wrong order are submitted when deploying the oracle.

Proof of Concept

- https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/contracts/helper/LPOracle.sol#L28-L29
- https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/contracts/RoeRouter.sol#L54-L79

Recommended Mitigation Steps

Implement a check to verify the tokens added in the params are the same (and in the same order) as the one used in the LP/oracle added also in the params.

[N-03] Math not following natspec

Summary

Unexpected return range based on the specified natspec docs.

Vulnerability Details

In this case, natspec function comment says its returned value follows a linear model: from `baseFeeX4 / 2` to `baseFeeX4 * 2`.

But in the code, if `adjustedBaseFeeX4` is greater than `baseFeeX4 * 3/2` (which is smaller than `baseFeeX4 * 2`), the code will set the `adjustedBaseFeeX4` to `baseFeeX4 * 3/2`; making impossible to reach a value of `baseFeeX4 * 2` ever although being stated like that in the natspec.

Above that function there is a comment saying `// Adjust from -50% to +50%` but again, that doesn't follow the defined natspec of the function, which are the ruling comments in a function.

Impact

Since NatSpec documentation is considered to be part of the contract's public API, NatSpec-related issues are assigned a higher severity than other code-comment-related issues and the code **must** follow the specified natspec, which is not happening.

Proof of Concept

- Return range is defined in the following Natspec: https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/contracts/GeVault.sol#L443
- But wrong max range is set here: https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/contracts/GeVault.sol#L457

Recommended Mitigation Steps

There are two options:

1. Change the Natspec like this:

2. /// @dev Simple linear model: from baseFeeX4 / 2 to baseFeeX4 * 2
3. /// @dev Simple linear model: from baseFeeX4 / 2 to baseFeeX4 * 3 /2

4. Change the max range math to this:

5. if (adjustedBaseFeeX4 > baseFeeX4 * 3 / 2) adjustedBaseFeeX4 = baseFeeX4 * 3 / 2;
6. if (adjustedBaseFeeX4 > baseFeeX4 * 2) adjustedBaseFeeX4 = baseFeeX4 * 2;

[N-04] Very old OpenZeppelin version being used

Summary

A very old version of OpenZeppelin is being used (version `4.4.1`).

Vulnerability Details

As said in the summary, the project is using a really old OpenZeppelin version (`4.4.1`), the latest stable one the `4.9.3`.

Using old versions can lead to less gas performance and security issues. We checked for known issues and although there are some known issues in the version being used, none of the audited contracts are using those features. But in any case, it is not recommended to use an old version so our recommendation is to upgrade to the latest stable version.

Impact

Using an old version can lead to worse gas performance, known contract issues, and potential 0-day issues.

Proof of Concept

- Known issues of the used old version: https://security.snyk.io/package/npm/@openzeppelin%2Fcontracts/4.4.1

Tools Used

Snyk.io.

Recommended Mitigation Steps

Upgrade to the latest stable OpenZeppelin version. At the time of the audit, the latest stable version is the `4.9.3`.

[N-05] Dead code in some contracts

Summary

There are some occurrences of unused (`dead`) code in some contracts.

Vulnerability Details

Avoiding to have unused code in the contracts is a good practice since it helps to keep the code clean and readable.

In `RangeManager.sol`, the `POS_MGR` constant is declared but not being used, so it could be removed: https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/contracts/RangeManager.sol#L36C47-L36C55

In `GeVault.sol`, the `rangeManager` is being declared but it is not being used so it could also be removed. Also, if this variable is removed, the import `import "./RangeManager.sol";` could also be deleted.

Impact

As said before, removing unused code can keep the code clean and more readable.

Proof of Concept

Affected lines:

- https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/contracts/RangeManager.sol#L36C47-L36C55
- https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/contracts/GeVault.sol#L41

Recommended Mitigation Steps

Remove the unused code.

[N-06] Inconsistent contract and file names

Summary

Inconsistent names between the contract file name and the contract name.

Vulnerability Details

As per the official Solidity documentation, it is recommended to have the same name for the file and the contract contained in it: https://docs.soliditylang.org/en/v0.8.10/style-guide.html#contract-and-library-names

```
Contract and library names should also match their filenames.
```

Impact

The current name structure can be misleading.

Proof of Concept

Affected contract:

- https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/interfaces/IAaveLendingPoolV2.sol#L8

Recommended Mitigation Steps

Rename the interface or change the file name.

[N-07] Double import

Summary

In the `RangeManager.sol` file, `SafeERC20` is imported twice.

Vulnerability Details

As said in the summary, the `SafeERC20` is imported twice.

Impact

It is useless and misleading to import the same import in the same contract twice.

Proof of Concept

- Import 1: https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/contracts/RangeManager.sol#L5
- https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/contracts/RangeManager.sol#L7

Recommended Mitigation Steps

Remove one of the duplicated imports:

```
import "./openzeppelin-solidity/contracts/token/ERC20/ERC20.sol"
import "./openzeppelin-solidity/contracts/token/ERC20/utils/Safe
import "./openzeppelin-solidity/contracts/utils/cryptography/ECD
```

```
- import "./openzeppelin-solidity/contracts/token/ERC20/utils/Sa
import "./openzeppelin-solidity/contracts/security/ReentrancyGua
import "../interfaces/AggregatorV3Interface.sol";
import "../interfaces/ILendingPoolAddressesProvider.sol";
import "../interfaces/IAaveLendingPoolV2.sol";
import "../interfaces/IAaveOracle.sol";
import "../interfaces/IUniswapV2Pair.sol";
import "../interfaces/IUniswapV2Factory.sol";
import "../interfaces/IUniswapV2Router01.sol";
import "../interfaces/ISwapRouter.sol";
import "../interfaces/INonfungiblePositionManager.sol";
import "./TokenisableRange.sol";
import "./openzeppelin-solidity/contracts/proxy/beacon/BeaconPro
import "./openzeppelin-solidity/contracts/access/Ownable.sol";
import {IPriceOracle} from "../interfaces/IPriceOracle.sol";
```

[N-08] Interfaces only used in tests should be separated from core interfaces

Summary

In the project, there are several interfaces only used in the tests. Those interfaces should not be in the same place as the interfaces used in the protocol contracts since they could be misleading.

Vulnerability Details

In the `contracts/interfaces` folder there are interfaces only used in tests and also interfaces used in the protocol contracts. Having them mixed together can be misleading and confusing so the ideal thing should be to separate them in different folders.

Impact

This can cause confusion and even use the wrong interfaces in tests or core protocol contracts.

Proof of Concept

Interfaces only used in tests that should be moved to another folder:

- https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/interfaces/IAToken.sol#L9
- https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/interfaces/ICreditDelegationToken.sol#L4
- https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/interfaces/IERC20.sol#L9
- https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/interfaces/IInitializableAToken.sol#L12
- https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/interfaces/ILendingPoolConfigurator.sol#L5
- https://github.com/code-423n4/2023-08-goodentry/blob/4b785d455fff04629d8675f21ef1d1632749b252/interfaces/IScaledBalanceToken.sol#L4C11-L4C30

Recommended Mitigation Steps

Separate the test interfaces and the protocol interfaces in different folders.

gzeon (Judge) commented:

> These downgraded items were also considered in scoring:
>
> Low:
>
> `ticks[]` in GeVault.sol not ordered in ascending price order
>
> Incorrect Oracle decimal assumption breaks price calculation
>
> Incorrect decimal checks excludes all valid pricefeed pairs
>
> Non-critical:
>
> Lack of validation between Tokenised Ranges and GeVault Uniswap Pool

Keref (Good Entry) acknowledged and commented:

> Removed dead code and correct natspec.

Gas Optimizations

For this audit, 19 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **JCK** received the top score from the judge.

The following wardens also submitted reports: OxAnah, Raihan, ReyAdmirado, Sathish9098, petrichor, wahedtalash77, hunter_w3b, naman1778, Oxhex, dharma09, SY_S, Oxta, matrix_Owl, SAQ, Rageur, DavidGiladi, Rolezn, *and* K42.

Gas Optimizations

Number	Issue	Instances	Total gas saved
[G-01]	State variables which are not modified within functions should be set as constant or immutable for values set at deployment	8	80000
[G-02]	Use assembly in place of abi.decode to extract calldata values more efficiently	3	
[G-03]	Cache external calls outside of loop to avoid re-calling function on each iteration	3	
[G-04]	Use assembly to perform efficient back-to-back calls	10	
[G-05]	Use calldata instead of memory for function arguments that do not get mutated	2	720
[G-06]	Use custom errors instead of require/assert	75	3750
[G-07]	Functions guaranteed to revert when called by normal users can be marked payable	14	180
[G-08]	Use assembly to emit events	35	1292
[G-09]	Use assembly to write address storage values	2	148
[G-10]	Use hardcoded address instead address(this)	47	

Number	Issue	Instances	Total gas saved
[G-11]	Use uint256(1)/uint256(2) instead for true and false boolean states	8	136800
[G-12]	Expensive operation inside a for loop	1	
[G-13]	Use assembly to validate msg.sender	5	
[G-14]	Duplicated require()/revert() Checks Should Be Refactored To A Modifier Or Function	3	84
[G-15]	A modifier used only once and not being inherited should be inlined to save gas	1	
[G-16]	Use assembly for loops	1	
[G-17]	require()/revert() strings longer than 32 bytes cost extra gas	4	56
[G-18]	Using private rather than public for constants, saves gas	6	
[G-19]	Amounts should be checked for 0 before calling a transfer	6	
[G-20]	Use constants instead of type(uintx).max	7	
[G-21]	Should use arguments instead of state variable	1	97
[G-22]	Caching global variables is more expensive than using the actual variable (use msg.sender instead of caching it)	1	
[G-23]	Empty blocks should be removed or emit something	2	8012
[G-24]	Can make the variable outside the loop to save gas	5	
[G-25]	abi.encode() is less efficient than abi.encodepacked()	2	200
[G-26]	Avoid contract existence checks by using low level calls	6	600
[G-27]	Using delete statement can save gas	1	
[G-28]	Not using the named return variable when a function returns, wastes deployment gas	5	

Number	Issue	Instances	Total gas saved
[G-29]	Sort Solidity operations using short-circuit mode	2	4200

[G-01] State variables which are not modified within functions should be set as constant or immutable for values set at deployment

Cache such variables and perform operations on them, if operations include modifications to the state variable(s) then remember to equate the state variable to it's cached counterpart at the end

```
file:  contracts/helper/FixedOracle.sol

7    address private owner;
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/helper/FixedOracle.sol#L7

```
file:  contracts/GeVault.sol

41     RangeManager rangeManager;
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/GeVault.sol#L41

```
file: contracts/TokenisableRange.sol

54   address public TREASURY_DEPRECATED = 0x22Cc3f665ba4C8982263

55   uint public treasuryFee_deprecated = 20;
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/TokenisableRange.sol#L54

[G-02] Use assembly in place of abi.decode to extract calldata values more efficiently

Instead of using abi.decode, we can use assembly to decode our desired calldata values directly. This will allow us to avoid decoding calldata values that we will not use.

```
file: contracts/PositionManager/OptionsPositionManager.sol

45    uint8 mode = abi.decode(params, (uint8) );

48    (, uint poolId, address user, address[] memory sourceSwap)

53    (, uint poolId, address user, address collateral) = abi.de
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/PositionManager/OptionsPositionManager.sol#L45

[G-03] Cache external calls outside of loop to avoid re-calling function on each iteration

Performing STATICCALLs that do not depend on variables incremented in loops should always try to be avoided within the loop. In the following instances, we are able to cache the external calls outside of the loop to save a STATICCALL (100 gas) per loop iteration.

Cache t.getTokenAmounts(bal) outside of loop to save 1 STATICCALL per loop iteration

```
file:  main/contracts/GeVault.sol

299         for (uint k = 0; k < ticks.length; k++){
        TokenisableRange t = ticks[k];
        address aTick = lendingPool.getReserveData(address(t)).aTo
        uint bal = ERC20(aTick).balanceOf(address(this));
        (uint amt0, uint amt1) = t.getTokenAmounts(bal);
        amount0 += amt0;
        amount1 += amt1;
    }
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/GeVault.sol#L299-L306

Cache lendingPool.liquidationCall() outside of loop to save 1 STATICCALL per loop iteration

file: contracts/PositionManager/OptionsPositionManager.sol

```
93        for ( uint8 k =0; k<assets.length; k++){
        address debtAsset = assets[k];

        // simple liquidation: debt is transferred from user to li
        uint amount = amounts[k];

        // liquidate and send assets here
        checkSetAllowance(debtAsset, address(lendingPool), amount)
        lendingPool.liquidationCall(collateral, debtAsset, user, a
        // repay tokens
        uint debt = closeDebt(poolId, address(this), debtAsset, am
        uint amt0 = ERC20(token0).balanceOf(address(this));
        uint amt1 = ERC20(token1).balanceOf(address(this));
        emit LiquidatePosition(user, debtAsset, debt, amt0 - amts[
        amts[0] = amt0;
        amts[1] = amt1;

    }
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/PositionManager/OptionsPositionManager.sol#L93-L110

Cache ticks[activeTickIndex+1].getTokenAmountsExcludingFees(1e18); outside of loop to save 2 STATICCALL per loop iteration

file: main/contracts/GeVault.sol

```
431         for (activeTickIndex = 0; activeTickIndex < ticks.len
          (uint amt0, uint amt1) = ticks[activeTickIndex+1].getTok
          (uint amt0n, uint amt1n) = ticks[activeTickIndex+2].getT
          if ( (amt0 == 0 && amt0n > 0) || (amt1 == 0 && amt1n > 0
            break;
```

```
        }
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/GeVault.sol#L431-L436

[G-04] Use assembly to perform efficient back-to-back calls

If similar external calls are performed back-to-back, we can use assembly to reuse any function signatures and function parameters that stay the same. In addition, we can also reuse the same memory space for each function call (scratch space + free memory pointer), which can potentially allow us to avoid memory expansion costs. In this case, we are also able to efficiently store the function signatures together in memory as one word, saving multiple MLOADs in the process.

Note: In order to do this optimization safely we will cache the free memory pointer value and restore it once we are done with our function calls. We will also set the zero slot back to 0 if neccessary.

Use for this t.TOKEN0() back to back external call assembly

▶ Details

[G-05] Use calldata instead of memory for function arguments that do not get mutated

When you specify a data location as memory, that value will be copied into memory. When you specify the location as calldata, the value will stay static within calldata. If the value is a large, complex type, using memory may result in extra memory expansion costs.

```
file:  contracts/PositionManager/OptionsPositionManager.sol

159     function buyOptions(
    uint poolId,
    address[] memory options,
    uint[] memory amounts,
    address[] memory sourceSwap
  )
    external
  {
```

```
189    function liquidate (
       uint poolId,
       address user,
       address[] memory options,
       uint[] memory amounts,
       address collateralAsset
     )
       external
     {
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/PositionManager/OptionsPositionManager.sol#L159-L166

[G-06] Use custom errors instead of require/assert

Consider the use of a custom error, as it leads to a cheaper deploy cost and run time cost. The run time cost is only relevant when the revert condition is met.

▶ Details

[G-07] Functions guaranteed to revert when called by normal users can be marked payable

If a function modifier such as onlyOwner is used, the function will revert if a normal user tries to pay the function. Marking thefunction as payable will lower the gas cost for legitimate callers because the compiler will not include checks for whether a paymentwas provided.

▶ Details

[G-08] Use assembly to emit events

We can use assembly to emit events efficiently by utilizing scratch space and the free memory pointer. This will allow us to potentially avoid memory expansion costs. Note: In order to do this optimization safely, we will need to cache and restore the free memory pointer.

▶ Details

[G-09] Use assembly to write address storage values

```
file:  contracts/GeVault.sol

108     function setTreasury(address newTreasury) public onlyOwn
    treasury = newTreasury;
    emit SetTreasury(newTreasury);
  }
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/GeVault.sol#L108-111

```
file:  contracts/RoeRouter.sol

83     function setTreasury(address newTreasury) public onlyOwne
    require(newTreasury != address(0x0), "Invalid address");
    treasury = newTreasury;
    emit UpdateTreasury(newTreasury);
  }
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/RoeRouter.sol#L83-L87

[G-10] Use hardcoded address instead of address(this)

Instead of using address(this), it is more gas-efficient to pre-calculate and use the hardcoded address. Foundry's script.sol and solmate's LibRlp.sol contracts can help achieve this. Refrences

▶ Details

[G-11] Use uint256(1)/uint256(2) instead for true and false boolean states

Use uint256(1) and uint256(2) for true/false to avoid a Gwarmaccess (100 gas), and to avoid Gsset (20000 gas) when changing from 'false' to 'true', after having been 'true' in the past. see source: https://github.com/OpenZeppelin/openzeppelin-

contracts/blob/58f635312aa21f947cae5f8578638a85aa2519f5/contracts/security/ReentrancyGuard.sol#L23-L27

```
file: contracts/GeVault.sol

50    bool public isEnabled = true;

64    bool public baseTokenIsToken0;

75    bool _baseTokenIsToken0
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/GeVault.sol#L50

```
file: contracts/RoeRouter.sol

28    bool isDeprecated;
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/RoeRouter.sol#L28

```
file:  contracts/helper/V3Proxy.sol

65   bool acceptPayable;
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/helper/V3Proxy.sol#L65

```
file: contracts/interfaces/IAaveLendingPoolV2.sol

146   bool receiveAToken

288   bool receiveAToken

235   bool _state
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/interfaces/IAaveLendingPoolV2.sol#L146

[G-12] Expensive operation inside a for loop

```
file:  contracts/RangeManager.sol

62       for (uint i = 0; i < len; i++) {
       if (start >= stepList[i].end || end <= stepList[i].start)
         continue;
       }
       revert("Range overlap");
     }
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/RangeManager.sol#L62-L67

[G-13] Use assembly to validate msg.sender

We can use assembly to efficiently validate msg.sender for the didPay and uniswapV3SwapCallback functions with the least amount of opcodes necessary. Additionally, we can use xor() instead of iszero(eq()), saving 3 gas. We can also potentially save gas on the unhappy path by using scratch space to store the error selector, potentially avoiding memory expansion costs.

```
file:  contracts/PositionManager/OptionsPositionManager.sol

281    if (user != msg.sender ) {

327    if (user == msg.sender) swapTokens(poolId, collateralAsse
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/PositionManager/OptionsPositionManager.sol#L281

```
file:  contracts/TokenisableRange.sol
```

```
136    require(msg.sender == creator, "Unallowed call");
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/TokenisableRange.sol#L136

```
file:  contracts/helper/FixedOracle.sol

11   require(msg.sender == owner, "Only the owner can call this
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/helper/FixedOracle.sol#L11

```
file:  contracts/PositionManager/OptionsPositionManager.sol

91    require( address(lendingPool) == msg.sender, "OPM: Call Una
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/PositionManager/OptionsPositionManager.sol#L91

[G-14] Duplicated require()/revert() Checks Should Be Refactored To A Modifier Or Function

In development process, if you changed one of them you should find all of other to change and for large and complicatedprojects possible this change will be missed.

```
file:  contracts/GeVault.sol

   /// @audit this require is duplicated on line 215, 250

203   require(poolMatchesOracle(), "GEV: Oracle Error");
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/GeVault.sol#L203

```
file: contracts/helper/V3Proxy.sol
```

```
    /// @audit this require is duplicated on line 91

87   require(acceptPayable, "CannotReceiveETH");
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/helper/V3Proxy.sol#L87

```
file:

 /// @audit this require is duplicated on line 106, 113, 125, 13

99   require(path.length == 2, "Direct swap only");
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/helper/V3Proxy.sol#L99

[G-15] A modifier used only once and not being inherited should be inlined to save gas

```
file:  contracts/helper/FixedOracle.sol

10  modifier onlyOwner() {
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/helper/FixedOracle.sol#L10

[G-16] Use assembly for loops

In the following instances, assembly is used for more gas efficient loops. The only memory slots that are manually used in the loops are scratch space (0x00-0x20), the free memory pointer (0x40), and the zero slot (0x60). This allows us to avoid using the free memory pointer to allocate new memory, which may result in memory expansion costs.

Note that in order to do this optimization safely we will need to cache and restore the free memory pointer after the loop. We will also set the zero slot (0x60) back to 0.

```
file:  contracts/GeVault.sol

314         for (uint k = 0; k < ticks.length; k++){
        removeFromTick(k);
    }
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/GeVault.sol#L314-L316

[G-17] require()/revert() strings longer than 32 bytes cost extra gas

```
file: contracts/helper/FixedOracle.sol

11    require(msg.sender == owner, "Only the owner can call this
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/helper/FixedOracle.sol#L11

```
file:  contracts/PositionManager/OptionsPositionManager.sol

495    require( amountsIn[0] <= maxAmount && amountsIn[0] > 0, "O

496    require( amountsIn[0] <= ERC20(path[0]).balanceOf(address(

525    require( amountB > 0, "OPM: Target Amount Too Low");
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/PositionManager/OptionsPositionManager.sol#L495

[G-18] Using private rather than public for constants, saves gas

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves 3406-3606 gas in deployment gas due to the compiler not having to create non-payable getter functions for deployment

calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table.

saved gas: 50436

```
file: contracts/GeVault.sol

62   uint public constant nearbyRanges = 2;
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/GeVault.sol#L62

```
file: contracts/RangeManager.sol

36   INonfungiblePositionManager constant public POS_MGR = INonf
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/RangeManager.sol#L36

```
file:  contracts/TokenisableRange.sol

58   INonfungiblePositionManager constant public POS_MGR = INonf
59   IUniswapV3Factory constant public V3_FACTORY = IUniswapV3Fa
60   address constant public treasury = 0x22Cc3f665ba4C898226353
61   uint constant public treasuryFee = 20;
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/TokenisableRange.sol#L58

[G-19] Amounts should be checked for 0 before calling a transfer

Checking non-zero transfer values can avoid an expensive external call and save gas. While this is done at some places, it's not consistently done in the solution. I suggest

adding a non-zero-value

```
file:  contracts/RangeManager.sol

96     ASSET_0.safeTransferFrom(msg.sender, address(this), amoun

98     ASSET_1.safeTransferFrom(msg.sender, address(this), amoun

176    transferAssetsIntoStep(tokenisedRanges[step], step, amoun

185    transferAssetsIntoStep(tokenisedTicker[step], step, amoun
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/RangeManager.sol#L96

```
file:  contracts/helper/V3Proxy.sol

164    ogInAsset.safeTransferFrom(msg.sender, address(this), amo

165    ogInAsset.safeApprove(address(ROUTER), amountInMax);
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/helper/V3Proxy.sol#L164

[G-20] Use constants instead of type(uintx).max

type(uint120).max or type(uint112).max, etc. it uses more gas in the distribution process and also for each transaction than constant usage.

```
file:   contracts/GeVault.sol

386   if ( ERC20(token).allowance(address(this), spender) < amou
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/GeVault.sol#L386

```
file: contracts/RangeManager.sol
```

```
118    trAmt = LENDING_POOL.withdraw(address(tokenisedRanges[step
```

```
13    uint256 ttAmt = LENDING_POOL.withdraw(address(tokenisedTic
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/RangeManager.sol#L118

```
file: contracts/TokenisableRange.sol
```

```
172    amount0Max: type(uint128).max,
```

```
173    amount1Max: type(uint128).max,
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/TokenisableRange.sol#L172

```
file:  contracts/helper/OracleConvert.sol
```

```
79    return (type(uint80).max, latestAnswer(), block.timestamp,
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/helper/OracleConvert.sol#L79

```
file:  contracts/PositionManager/PositionManager.sol
```

```
81     if ( ERC20(token).allowance(address(this), spender) < amo
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/PositionManager/PositionManager.sol#L81

[G-21] Should use arguments instead of state variable

state variables should not used in emit , This will save near 97 gas

```
file: /contracts/GeVault.sol
```

```
362     emit Rebalance(tickIndex);
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/GeVault.sol#L362

[G-22] Caching global variables is more expensive than using the actual variable (use msg.sender instead of caching it)

```
file: /contracts/TokenisableRange.sol

88     creator = msg.sender;
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/TokenisableRange.sol#L88

[G-23] Empty blocks should be removed or emit something

f the block is an empty if-statement block to avoid doing subsequent checks in the else-if/else conditions, the else-if/else conditions should be nested under the negation of the if-statement, because they involve different classes of checks, which may lead to the introduction of errors when the code is later modified `(if( x ) {}else if(y) { . . . }else{ . . . } => if ( !x) {if(y) { . . . }else{ . . .}})`. Empty receive()/fallback() payable functions that are not used, can be removed to save deployment gas.

```
file: /contracts/PositionManager/PositionManager.sol

52        ) virtual external returns (bool result) {
53           }
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/PositionManager/PositionManager.sol#L52-L53

```
file: /contracts/PositionManager/OptionsPositionManager.sol
```

```
25      constructor (address roerouter) PositionManager(roeroute
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/PositionManager/OptionsPositionManager.sol#L25

[G-24] Can make the variable outside the loop to save gas

Consider making the stack variables before the loop which gonna save gas

```
file: /contracts/PositionManager/OptionsPositionManager.sol

72      address asset = assets[k];

73      uint amount = amounts[k];


94      address debtAsset = assets[k];
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/PositionManager/OptionsPositionManager.sol#L72-L73

```
file: /contracts/GeVault.sol

301      address aTick = lendingPool.getReserveData(address(t)).

302      uint bal = ERC20(aTick).balanceOf(address(this));
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/GeVault.sol#L301-L302

[G-25] abi.encode() is less efficient than abi.encodepacked()

In terms of efficiency, abi.encodePacked() is generally considered to be more gas-efficient than abi.encode(), because it skips the step of adding function signatures and other metadata to the encoded data. However, this comes at the cost of reduced safety, as abi.encodePacked() does not perform any type checking or padding of data.

Refference: https://github.com/ConnorBlockchain/Solidity-Encode-Gas-Comparison

```
file: /contracts/PositionManager/OptionsPositionManager.sol

168    bytes memory params = abi.encode(0, poolId, msg.sender, s

199    bytes memory params = abi.encode(1, poolId, user, collate
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/PositionManager/OptionsPositionManager.sol#L168

[G-26] Avoid contract existence checks by using low level calls

Prior to 0.8.10 the compiler inserted extra code, including EXTCODESIZE (100 gas), to check for contract existence for external function calls. In more recent solidity versions, the compiler will not insert these checks if the external call has a return value. Similar behavior can be achieved in earlier versions by using low-level calls, since low level calls never check for contract existence.

```
file: /contracts/PositionManager/OptionsPositionManager.sol

135    (uint256 amount0, uint256 amount1) = TokenisableRange(fla

270      (uint token0Amount, uint token1Amount) = TokenisableRan

305      debt = TokenisableRange(debtAsset).deposit(token0Amount
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/PositionManager/OptionsPositionManager.sol#L135

```
file: /contracts/GeVault.sol

87    lendingPool = ILendingPool(ILendingPoolAddressesProvider(l

88    oracle = IPriceOracle(ILendingPoolAddressesProvider(lpap).
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/GeVault.sol#L87-L88

```
file: /contracts/TokenisableRange.sol

370    (uint160 sqrtPriceX96,,,,,,)  = IUniswapV3Pool(pool).slot
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/TokenisableRange.sol#L370

[G-27] Using delete statement can save gas

```
file: /contracts/PositionManager/OptionsPositionManager.sol

204      flashtype[i] = 0; // dont open debt for liquidations, n
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/PositionManager/OptionsPositionManager.sol#L204

[G-28] Not using the named return variable when a function returns, wastes deployment gas

```
file: /contracts/PositionManager/OptionsPositionManager.sol

514    internal view returns (uint amountB)
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/PositionManager/OptionsPositionManager.sol#L514

```
file: /contracts/GeVault.sol

177  function getTickLength() public view returns(uint len){
```

```
298  function getReserves() public view returns (uint amount0, u
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/GeVault.sol#L177

```
file: /contracts/RangeManager.sol

212  function getStepListLength() external view returns (uint256
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/RangeManager.sol#L212

[G-29] Sort Solidity operations using short-circuit mode

Short-circuiting is a solidity contract development model that uses OR/AND logic to sequence different cost operations. It puts low gas cost operations in the front and high gas cost operations in the back, so that if the front is low If the cost operation is feasible, you can skip (short-circuit) the subsequent high-cost Ethereum virtual machine operation.

```
//f(x) is a low gas cost operation
//g(y) is a high gas cost operation
//Sort operations with different gas costs as follows
 f(x) || g(y)
 f(x) && g(y)
```

```
file: /contracts/RangeManager.sol

63      if (start >= stepList[i].end || end <= stepList[i].start
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/RangeManager.sol#L63

```
file: /contracts/PositionManager/OptionsPositionManager.sol

495    require( amountsIn[0] <= maxAmount && amountsIn[0] > 0, "
```

https://github.com/code-423n4/2023-08-goodentry/blob/main/contracts/PositionManager/OptionsPositionManager.sol#L495

Keref (Good Entry) confirmed and commented:

> Partially implemented.

Audit Analysis

For this audit, 5 analysis reports were submitted by wardens. An analysis report examines the codebase as a whole, providing observations and advice on such topics as architecture, mechanism, or approach. The report highlighted below by **catellatech** received the top score from the judge.

The following wardens also submitted reports: Sathish9098, K42, digitizeworx, *and* 0xSmartContract.

Description

Good Entry is an advanced decentralized platform that allows users to deposit assets and earn yields through Tokenizable Ticks, representing positions in a specific price range of tokenized assets. It leverages cutting-edge technology from Uniswap V3 and Aave's lending market to ensure a secure and efficient experience for its users. To maintain fairness and prevent price manipulation, Chainlink Oracles are utilized.

The key contracts of the protocol for this Audit are:

- **GeVault Contract:** Allows users to deposit assets and receive Tokenizable Ticks in return. These Ticks generate yields and enable users to participate in the DeFi ecosystem.
- **RangeManager Contract:** Manages and tracks the tokenizable ranges in the protocol. Users can enter and exit these ranges using Aave's lending and liquidity services.
- **TokenisableRange Contract:** Enables users to participate in Uniswap V3 by creating tokenized ranges. Users receive a portion of fees generated by these positions.

- **PositionManager Contract**: Contains basic and reusable functions to interact with ROE lending pools. Enables leverage and deleverage on tokenized assets.
- **RoeRouter Contract**: Acts as a router for ROE lending pools and contains critical parameters related to them.
- **OptionsPositionManager Contract**: A platform for managing leveraged and deleveraged positions for tokenized assets in TR form.

Existing Patterns: The protocol uses standard Solidity and Ethereum patterns. It uses the `ERC20` standards and `Ownable` pattern for ownership management.

Approach

During the analysis, we focused on thoroughly understanding the codebase and providing recommendations to improve its functionality.

We divided the audit into 6 parts. The examined contracts were:

1. `RoeRouter.sol`
2. `PositionManager.sol`
3. `RangeManager.sol`
4. `TokenisableRange.sol`
5. `GeVault.sol`
6. `OptionsPositionManager.sol`

We started with the RoeRouter contract and then proceeded with the rest of the contracts. The reason for this sequence was that we began with the contract that had the fewest lines of code and then progressed to the others to gain a better understanding when we reached the longest contract, OptionsPositionManager.

We dedicated over 80 hours during the 6 day audit to analyse the codebase. Our goal is to provide the project team with a detailed report that gives them a broader perspective and leverages the value of our research to strengthen the security, usability, and efficiency of the protocol.

Architecture Description and Diagram

Architecture of the key contracts that are part of the Good Entry protocol:

GeVault: - The GeVault contract is a critical part of the Good Entry protocol. It allows users to deposit assets and earn yields through Tokenizable Ticks. Tokenizable Ticks are units of value created to represent positions in a specific price range of a tokenized asset. Users can deposit their assets into GeVault and receive Tokenizable Ticks in return Ticks generate yields that accumulate in the GeVault, enabling users to participate in the DeFi ecosystem and reap benefits.

RangeManager: - The RangeManager contract is responsible for managing and tracking tokenizable ranges in the Good Entry protocol. Users can enter and exit these ranges using `Aave's` lending and liquidity services, providing an efficient and secure trading experience. This contract is responsible for managing and maintaining a record of tokenized positions to ensure smooth and seamless operations.

TokenisableRange: - The main functionality of the TokenisableRange contract is to enable users to participate in `Uniswap V3` by creating tokenized ranges. By creating these ranges, users can receive a portion of the fees generated by those positions. The contract manages the position in `Uniswap V3` and ensures that fees are accumulated and reinvested safely.

PositionManager: - The “PositionManager” contract is a reusable, foundational contract used in the Good Entry protocol to interact with ROE (ROE lending pools). It provides a set of functions that facilitate the management of loan positions and flash loans in the protocol. This contract is essential to enable users to leverage and deleverage tokenized assets in the form of TR (Tokenizable Range).

RoeRouter: - The “RoeRouter” contract is a router for a list of ROE lending pools in the Good Entry protocol. It contains important parameters related to these pools, such as the treasury address and its updates. It allows for the addition and deprecation of pools to maintain the protocol’s flexibility and updates.

OptionsPositionManager: - The “OptionsPositionManager” contract is a smart-contract-based position management platform that is part of the Good Entry protocol. It offers leverage and deleverage functionalities for tokenized assets in the form of TR (Tokenizable Range). Additionally, it provides risk management and liquidation tools to ensure the safety and efficiency of operations within the protocol.

Codebase Quality

Overall, we consider the quality of the Good Entry codebase to be excellent. The code appears to be very mature and well-developed. We have noticed the implementation of various standards, such as ERC20, which TokenizableRange.sol and GeVault.sol adhere to appropriately. Additionally, it is evident that ideas and inspiration have been drawn from other successful protocols, like GMX, showcasing a good practice of learning from the best solutions in the DeFi ecosystem. Details are explained below:

Codebase Quality Categories	Comments
Unit Testing	Codebase is well-tested it was great to see the protocol using Python and the Brownie framework.
Code Comments	Comments in general were solid. However is always room for improvement. Some areas could benefit from greater clarity in comments or explanations for example in the Interfaces contracts, so that developers can address the code more effectively. Providing more detailed comments and documentation for complex or critical sections of the code can greatly enhance the codebase's overall readability and maintainability. This would not only help the current development team but also make it easier for future contributors to understand and build upon the existing code.
Documentation	It would be helpful if the docs explained how the ecosystem works from a basic contract level so that it is easier to digest for developers, users and auditors looking to integrate into the Good Entry Protocol
Organization	Codebase is very mature and well organized with clear distinctions between the 6 contracts, and their respective interfaces.

Systemic & Centralization Risks

The analysis provided highlights several significant systemic and centralization risks present in the Good Entry protocol. These risks encompass concentration risk in GeVault, liquidity and yield risk, third-party dependency risk, and centralization risks arising from the existence of an "owner" role in specific contracts. However, the documentation lacks clarity on whether this address represents an externally owned account (EOA) or a contract, warranting the need for clarification. Additionally, the absence of fuzzing and invariant tests could also pose risks to the protocol's security.

1. **Concentration risk in GeVault:**

- If the GeVault contract accumulates a significant amount of users' assets, it could create a systemic risk, especially if the assets are mismanaged or suffer significant losses. Diversifying assets across different platforms and contracts can help mitigate this risk.

2. **Liquidity and yield risk:**

   - If the contracts managing liquidity and yield do not function efficiently or securely, users may encounter issues entering or exiting positions, negatively affecting user experience and confidence in the protocol.

3. **Third-party dependency risk:**

   - If the Good Entry protocol heavily relies on other third-party protocols or services (e.g., Aave or Uniswap V3), any issues with those services could adversely affect the functioning of Good Entry.

4. **Centralization risks:**

   - It's important to note that the presence of an "owner" role raises concerns about centralization. In a decentralized protocol, it is preferable to have governance mechanisms that involve community participation in decision-making rather than relying solely on a single entity or role with elevated privileges. the following contracts have an "owner" role or variable:

     - GeVault
     - RangeManager
     - TokenisableRange
     - PositionManager
     - RoeRouter

   Good Entry's behavior can be influenced by these contracts and owner addresses, which introduce a certain level of centralization risk. If these addresses are controlled by a **malicious entity**, they could disrupt Good Entry's operations or perform actions that are not in the best interests of other Traders / holders.

5. **Weird ERC-20 Tokens:**

- If a user deposits malicious or “weird” ERC20 tokens into the Good Entry protocol, there could be several negative consequences, such as:

    - **Loss of funds**: If the contract is not designed to handle these malicious tokens properly, there could be vulnerabilities that allow an attacker to steal the deposited funds or negatively impact the integrity of the protocol.
    - **Price manipulation**: Malicious tokens could have adverse effects on liquidity and the overall functioning of the protocol, which could affect prices and the stability of the assets involved.

6. No having, fuzzing and invariant tests could open the door to future vulnerabilities.

Properly managing these risks and implementing best practices in security and decentralization will contribute to the sustainability and long-term success of the Good Entry protocol.

Recommendations

1. **Diversification of Oracles:**

    - Consider the possibility of using multiple price oracle sources like TWAP from Uniswap, instead of relying solely on Chainlink oracles. This would help mitigate the risk of failures in a single oracle and provide greater resilience to the system.

2. **Detailed Documentation:**

- Improve the documentation of the contracts and the protocol in general. Ensure that comments and explanations in the code are clear and comprehensive so that other developers can easily understand it and address any issues or improvements effectively.

3. **Governance and Upgrades:**

   - Establish a robust governance system that allows the community to participate in decision-making and implement upgrades efficiently. This will ensure that the protocol can adapt to market changes and remain secure and efficient over time.

4. **Liquidity Management:**

   - Implement mechanisms to manage liquidity effectively, especially in scenarios of concentrated liquidity, to minimize impermanent loss and optimize user returns.

5. Throughout the codebase, there is repeated use of functions with the same or even similar names. It is recommended to avoid this practice as it makes the code very confusing and may have unintended consequences by calling the wrong function.
6. Too much inheritance and nested functions can make certain functions difficult to read and audit. Consider reducing this if possible as this can make the code more readable and reveal potential bugs.
7. It might be beneficial to implement a circuit breaker or pause mechanism. This could help in situations where a bug or vulnerability is discovered, allowing contract operations to be halted while the issue is resolved.

Gas Optimization

Good Entry is generally efficient in terms of gas optimizations, many generally accepted gas optimizations have been implemented, gas optimizations with minor effects are already mentioned in automatic finding, but gas optimizations will not be a priority considering code readability and code base size.

Conclusion

In general, the Good Entry project exhibits an interesting and well-developed architecture we believe the team has done a good job regarding the code, but the

identified risks need to be addressed, and measures should be implemented to protect the protocol from potential malicious use cases. Additionally, it is recommended to improve the documentation and comments in the code to enhance understanding and collaboration among developers. It is also highly recommended that the team continues to invest in security measures such as mitigation reviews, audits, and bug bounty programs to maintain the security and reliability of the project.

Time Spent

A total of 5 days were spent to cover this audit, broken down into the following:

- 1st Day: Trying to understand the protocol flows and implementation
- 2nd Day: Focus on the 5 main contracts, such as GeVault, RangeManager, TokenisableRange, PositionManager and RoeRouter .
- 3rd Day: Focus on the rest of the smart contracts.
- 4th Day: Finish up QA and Analysis.
- 5th Day: Write the report for high and medium findings.

Time spent:
90 hours

Keref (Good Entry) acknowledged

Mitigation Review

Introduction

Following the C4 audit, 3 wardens (3docSec, xuwinnie and kutugu) reviewed the mitigations for all identified issues. Additional details can be found within the Good Entry Mitigation Review repository.

Overview of Changes

Summary from the Sponsor:

Simple errors like the sqrt version were corrected. The main change is to the fee system in `TokenizsableRange` . Because repaying a TR debt exactly is tricky and

introduces several problems (`addDust` , fee clawing on deposit...), the system has been changed. `TokenisableRange` fees aren't compounded anymore directly in TR, but are sent to the corresponding `GeVault` . The `GeVault` address is queried from a list (new addition to `RoeRouter`) (or to treasury is no such vault exists). We added a `depositExactly` function to TR, which takes an additional `expectedAmount` parameter. When depositing in TR, if because of rounding the difference between the expected liquidity and the actually minted liquidity is dust (as defined by: value is 0, or lower than 1 unit of the underlying token), then mint the expected liquidity.

Another set of changes is for the `GeVaults` : the `activeIndex` system has been changed so that the index point to the first tick above current price. If 2 ticks below or above exist, it tries to deposit assets in them (and gracefully ignores errors so as to prevent revert, eg when price is inside a tick). Instead of depositing half of the assets into each of the 2 ticks above and below, this has been parameterized, allowing to change asset distribution in case of high volatility.

Mitigation Review Scope

Branch

All PRs can be seen **here**, and have been linked in each issue's comments.

Individual PRs

URL	Mitigation of	Purpose
https://github.com/GoodEntry-io/ge/pull/4	H-01, H-04	Remove complex fee clawing strategy
https://github.com/GoodEntry-io/ge/commit/a8ba6492b19154c72596086f5531f6821b4a46a2	H-02	Take unused funds into account for TVL
https://github.com/GoodEntry-io/ge/pull/3	H-03	Scale down sqrtPriceX96 to prevent overflow
https://github.com/GoodEntry-io/ge/pull/2	H-05	Send back unused funds to user
https://github.com/GoodEntry-io/ge/commit/8b0feaec0005937c8e6c7ef9bf039a0c2498529a	H-06	Use correct Uniswap for sol ^0.8 libs
https://github.com/GoodEntry-io/ge/pull/10	M-01	Added explicit require msg.sender == to

URL	Mitigation of	Purpose
https://github.com/GoodEntry-io/ge/commit/bbbac57c110223f45851494971a34f57c55922c7	M-02	Prevent collect from reverting by adding a check that it doesnt try to collect 0
https://github.com/GoodEntry-io/ge/pull/11	M-03	Reworked activeTickIndex as per desc above
https://github.com/GoodEntry-io/ge/pull/8	M-05, M-07	Removed addDust mechanism, replaced by depositExactly in TR
https://github.com/GoodEntry-io/ge/pull/3	M-06	Added return value check

Out of Scope

Good Entry sponsors decided to not mitigate M-04.

Per the sponsor: This is not a problem, as the team deploys the contract and can deposit a very small initial amount. The attack would then steal a negligible amount (likely less than the gas cost).

Mitigation Review Summary

Original Issue	Status	Full Details
H-01	Mitigation confirmed	Reports from kutugu, xuwinnie and 3docSec
H-02	Mitigation confirmed	Reports from kutugu, xuwinnie and 3docSec
H-03	Mitigation confirmed	Reports from kutugu, xuwinnie and 3docSec
H-04	Mitigation confirmed	Reports from kutugu, xuwinnie and 3docSec
H-05	Mitigation confirmed	Reports from kutugu, xuwinnie and 3docSec
H-06	Mitigation confirmed	Reports from kutugu, xuwinnie and 3docSec
M-01	Mitigation confirmed	Reports from kutugu, xuwinnie and 3docSec
M-02	Mitigation confirmed	Reports from kutugu, xuwinnie and 3docSec

Original Issue	Status	Full Details
M-03	Mitigation error	Reports from kutugu, xuwinnie and 3docSec - details also shared below
M-05	Mitigation confirmed	Reports from kutugu and 3docSec
M-06	Mitigation confirmed	Reports from kutugu, xuwinnie and 3docSec
M-07	Mitigation confirmed	Reports from kutugu, xuwinnie and 3docSec
M-08	Not fully mitigated	Reports from kutugu and 3docSec - details also shared below

The wardens confirmed the mitigations for all in-scope findings except for M-03 (Medium severity mitigation error) and M-08 (unmitigated). They also surfaced several new issues: 2 High severity, 2 Medium severity, and several Low severity/Non-critical.

M-03 Mitigation error: `getActiveTickIndex` implementation error

Submitted by **kutugu**, *also found by* **xuwinnie** *and* **3docSec**

Severity: Medium

Lines of code

https://github.com/GoodEntry-io/ge/blob/c7c7de57902e11e66c8186d93c5bb511b53a45b8/contracts/GeVault.sol#L470

Impact

The implementation of `getActiveTickIndex` is wrong and the searched ticks do not meet expectations; which is causing funds to be incorrectly allocated to edge ticks, and there is basically no staking income.

Proof of Concept

```
    // if base token is token0, ticks above only contain base to
    if (newTickIndex > 1)
      depositAndStash(
        ticks[newTickIndex-2],
        baseTokenIsToken0 ? 0 : availToken0 / liquidityPerTick,
        baseTokenIsToken0 ? availToken1 / liquidityPerTick : 0
      );


  /// @notice Return first valid tick
  function getActiveTickIndex() public view returns (uint active
    // loop on all ticks, if underlying is only base token then
    for (uint tickIndex = 0; tickIndex < ticks.length; tickIndex
      (uint amt0, uint amt1) = ticks[tickIndex].getTokenAmountsE
      // found a tick that's above price (ie its only underlying
      if( (baseTokenIsToken0 && amt0 == 0) || (!baseTokenIsToken
    }
    // all ticks are below price
    return ticks.length;
  }
```

According to the code comments:

- If `baseTokenIsToken0` is true, ticks above current price only contain base token, that is `token0`, so `amt1` is `0`.
- And if `baseTokenIsToken0` is false, ticks below current price only contain quote token, that is `token1`, so `amt0` is `0`.

Function `getActiveTickIndex` checks `amt0` twice in the code is wrong, which causes `baseTokenIsToken0 && amt0 == 0` to be true when the tick is below the current price. That is, the searched tick is the first tick lower than the current price, not the first tick greater than the current price; which is the first tick in the list. This results in funds being staked to marginal ticks and unable to obtain staking income.

Recommended Mitigation Steps

```
    // found a tick that's above price (ie its only underlying
    if( (baseTokenIsToken0 && amt1 == 0) || (!baseTokenIsToken
```

Assessed type

Context

gzeon (judge) decreased severity to Medium

M-08 Not fully mitigated: The success of low-level calls is not checked in V3Proxy

Submitted by kutugu, *also found by* 3docSec

Lines of code

https://github.com/code-423n4/2023-08-goodentry/blob/71c0c0eca8af957202ccdbf5ce2f2a514ffe2e24/contracts/helper/V3Proxy.sol#L156
https://github.com/code-423n4/2023-08-goodentry/blob/71c0c0eca8af957202ccdbf5ce2f2a514ffe2e24/contracts/helper/V3Proxy.sol#L174
https://github.com/code-423n4/2023-08-goodentry/blob/71c0c0eca8af957202ccdbf5ce2f2a514ffe2e24/contracts/helper/V3Proxy.sol#L192

Comments

The success of low-level calls is not checked in V3Proxy. If `msg.sender` is a contract and the fallback function has additional logic, the protocol will succeed transfer by default, which will result in the loss of user funds.

Mitigation

There is no corresponding repair link posted in the readme, and there are no related repairs in https://github.com/GoodEntry-io/ge. The fix link in the issue is marked as https://github.com/GoodEntry-io/ge/pull/3/files. I think this is a misunderstanding. The sponsor mistook the issue for transfer and modified it to a call.

Suggestion

What needs to be fixed is the low-level calls in the V3Proxy contract. Their success return values should be checked for success. Code positioning can be done based on

the three links I mentioned above.

Conclusion

Needs repair.

Assessed type

Call/delegatecall

gzeon (judge) commented:

> @Keref - please note that the fix in PR14 is faulty due to a duplicated transfer here.

Keref (Good Entry) commented:

> Thanks, been fixed in commit .

Note: for full discussion, see here.

Attacker can extract value from pool by sandwiching themselves at `swapAll` during close

Submitted by xuwinnie

Severity: High

Lines of code

https://github.com/GoodEntry-io/ge/blob/c7c7de57902e11e66c8186d93c5bb511b53a45b8/contracts/PositionManager/OptionsPositionManager.sol#L454

Vulnerability details

An attacker can drain the lending pool by leveraging two facts:

1. `swapAll` allows 1% slippage.
2. There is no Health Factor check after `close` .

Alice and Bob are good friends, the steps are (in one single tx):

1. Alice deposits 10000 USDT and borrows `$` 7000 worth of TR.
2. Bob buys ETH at AMM to push up the price to oracle + 1%.
3. Alice `close` but only repays 1 wei debt. The real intention is to swap from USDT collateral to ETH collateral.
4. Bob sells ETH at AMM to pull down the price to oracle - 1%.
5. Alice `close` but only repays 1 wei debt to swap to USDT collateral.
6. Repeat.
7. Alice has 0 collateral and Bob gains 10000 USDT by sandwiching.

By continuously sandwiching Alice, Bob can extract value from the pool. A simple mitigation is to add a `HF` check after each swap.

Assessed type

Context

gzeon (judge) commented:

> Besides sandwiching is usually out-of-scope, warden's POC of Alice and Bob acting together failed to show it is profitable as an attack.

xuwinnie (warden) commented:

> Hey @gzeon, this is not just a sandwich, the main point is lack of health factor check. Sponsor has confirmed and fixed this in PR17.

> The attack is profitable because originally, Alice should not be able to remove all the collateral since they have outstanding debt but now they will be able to do so.

`removeFromAllTicks` should be done before `getTVL`

Submitted by xuwinnie, also found by kutugu (1,2) and 3docSec

Severity: High

Lines of code

https://github.com/GoodEntry-io/ge/blob/c7c7de57902e11e66c8186d93c5bb511b53a45b8/contracts/GeVault.sol#L265-L293

Vulnerability details

After the mitigation, the TR fee is directly sent to GE vault. Suppose 0.1 eth trading fee has accumulated in TR:

```
uint vaultValueX8 = getTVL();
uint adjBaseFee = getAdjustedBaseFee(token == address(token0));
// Wrap if necessary and deposit here
if (msg.value > 0){
  require(token == address(WETH), "GEV: Invalid Weth");
  // wraps ETH by sending to the wrapper that sends back WETH
  WETH.deposit{value: msg.value}();
  amount = msg.value;
}
else {
  ERC20(token).safeTransferFrom(msg.sender, address(this), amoun
}

// Send deposit fee to treasury
uint fee = amount * adjBaseFee / 1e4;
ERC20(token).safeTransfer(treasury, fee);
uint valueX8 = oracle.getAssetPrice(token) * (amount - fee) / 10


require(tvlCap > valueX8 + vaultValueX8, "GEV: Max Cap Reached")


uint tSupply = totalSupply();
// initial liquidity at 1e18 token ~ $1
if (tSupply == 0 || vaultValueX8 == 0)
  liquidity = valueX8 * 1e10;
else {
  liquidity = tSupply * valueX8 / vaultValueX8;
}

rebalance();
```

As above, when depositing, the 0.1 eth fee is not reflected in `getTVL`. Only after `removeFromAllTicks` (in `rebalance`) will the fee be collected and sent to GE vault. Therefore, an attacker can take a flashloan, deposit and then withdraw to steal almost all of the 0.1 eth trading fee. The process is similar to what H-04 has been described.

When withdrawing, similarly, user will incur a loss, since latest trading fee is not accounted for.

Assessed type

Context

Transaction origin check in ROE Markets make `Options` positions opened by contract users impossible to reduce or close

Submitted by 3docSec, *also found by* xuwinnie

Severity: Medium

Lines of code

https://github.com/GoodEntry-io/GoodEntryMarkets/blob/2e3d23016fadb45e188716d772cec7c2096fae01/contracts/protocol/lendingpool/LendingPool.sol.0x20#L492
https://github.com/GoodEntry-io/ge/blob/c7c7de57902e11e66c8186d93c5bb511b53a45b8/contracts/PositionManager/OptionsPositionManager.sol#L386
https://github.com/GoodEntry-io/ge/blob/c7c7de57902e11e66c8186d93c5bb511b53a45b8/contracts/PositionManager/OptionsPositionManager.sol#L387
https://github.com/GoodEntry-io/ge/blob/c7c7de57902e11e66c8186d93c5bb511b53a45b8/contracts/PositionManager/OptionsPositionManager.sol#L412

Vulnerability details

The Roe Markets `LendingPool.sol` that `OptionsPositionManager` uses is a modified version of Aave V2 with an added `PMTransfer` functionality, which is used by `OptionsPositionManager` when closing or reducing positions.

This `PMTransfer` only works when the user whose position is being operated on is in soft liquidation, or when the user initiated the transaction themselves:

```
if (tx.origin != user) {
  (,,,, uint256 healthFactor) = GenericLogic.calculateUserAc
    user,
    _reserves,
    _usersConfig[user],
    _reservesList,
    _reservesCount,
    _addressesProvider.getPriceOracle()
    );
  require(healthFactor <= softLiquidationThreshold, "Not ini
```

However, when positions are opened, `OptionsPositionManager` attributes debt to `user = msg.sender`.

```
function buyOptions(
  // ...
)
  external
{
  // ...
  LP.flashLoan( address(this), options, amounts, flashtype, ms
}
```

This means that any user (EOA or contract) can open option positions, only for themselves, but only EOAs are materially able to close these positions.

Impact

The user interacting with `OptionsPositionManager` via a contract will be forced to stay into their positions until defaulting; only then, they can pass the check in `PMTransfer` and liquidate the position.

Proof of Concept

Setting up is fairly simple in terms of steps, but requires interaction with a real Roe Markets lending pool (i.e. would work with a mainnet fork):

- Have a contract provide a generous amount of liquidity to the lending pool.
- Have the contract open a relatively little leveraged position through `OptionsPositionManager`'s `buyOptions` function. Little enough to not be in liquidation territory.
- Have the contract close the position through `OptionsPositionManager`'s `close`.
- The `close` call will revert, having the user stuck in their position, accumulating debt until liquidation.

```
function testNotInitiatedByUser() public {
  // we have two addresses, the contractCaller (EOA), and theC
  address contractCaller = address(uint160(uint256(keccak256("
  address theContract = address(uint160(uint256(keccak256("the

  // theContract has some collateral in the lending pool, so i
  changePrank(tokenWhale);
  WETH.transfer(theContract, 20e18);

  changePrank(operator);
  range.transfer(theContract, 1e18);

  changePrank(theContract);
  range.approve(address(RoeWethUsdcLP), 1e18);
  RoeWethUsdcLP.deposit(address(range), 1e18, theContract, 0);
  RoeWethUsdcLP.setUserUseReserveAsCollateral(address(range),

  // msg.sender different from tx.origin
  changePrank(theContract, contractCaller);

  // can open a position
  ICreditDelegationToken(0xB19Dd5DAD35af36CF2D80D1A9060f1949b1
      .approveDelegation(address(optionsPM), type(uint256).max

  address[] memory options = new address[](1);
  options[0] = address(range);

  uint256[] memory amounts = new uint256[](1);
```

```
        amounts[0] = 0.0001e18;

        address[] memory sourceSwap = new address[](1);
        sourceSwap[0] = address(USDC);

        optionsPM.buyOptions(
            0,
            options,
            amounts,
            sourceSwap
        );

        // simulate an always healthy position by changing the soft
        vm.store(address(RoeWethUsdcLP), bytes32(uint256(0x3e)), byt

        // the position can't be closed 😱 - ROE markets' PMTransfer
        WETH.approve(address(optionsPM), type(uint256).max);

        vm.expectRevert("Not initiated by user");
        optionsPM.close(0,
            theContract,
            address(range),
            0.0001e18,
            address(WETH));
    }
```

Tools Used

Code review, Foundry

Recommended Mitigation Steps

- Do not allow options positions to be opened to contracts.
- Possibly, add an argument `onBehalfOf` so that contracts can still open positions for EOAs.

Assessed type

Invalid Validation

UniswapV3 trading fees are always locked in treasury instead of going back to the protocol users through `GeVault`

Submitted by 3docSec*, also found by* kutugu

Severity: Medium

Lines of code

https://github.com/GoodEntry-io/ge/blob/c7c7de57902e11e66c8186d93c5bb511b53a45b8/contracts/TokenisableRange.sol#L179

Vulnerability details

`TokenisableRange` was redesigned to redirect collected fees to a pre-defined `GeVault`, where the protocol stakers can benefit from the added value.

However, the use of an incorrect variable makes this distribution of the fees impossible to happen, and the fees will necessarily be sent to the the protocol treasury:

- In `TokenisableRange.sol`, where fees are distributed, the vault lookup checks for a vault of (token0, token0) instead of (token0, token1):

```
try RoeRouter(roerouter).getVault(address(TOKEN0.token), address(TOKE
  vault = _vault;
}
```

- Because of the validation happening in `RoeRouter`, no `GeVault` can possibly be configured for a pair of the same address to work around the problem once the protocol is live:

```
function setVault(address token0, address token1, address vault) public
require(token0 < token1, "Invalid Order");
```

- This will lead to fee funds being temporarily locked in the Treasury instead of being distributed to the protocol users through the designed vault

Impact

Users will see the fees generated from their participation to the protocol taken away by the protocol instead of redistributed to them.

The funds are only temporarily locked because the protocol team can act to forward them to the appropriate `GeVault`, but users may perceive this issue as the protocol stealing trading fees from them.

Proof of Concept

1. Have a random account as `vault` - it can be an EOA for the sake of this PoC.
2. Call `RoeRouter.setVault(USDC, WETH, vault)`.
3. Create and fund a `TokenisableRange` for USDC/WETH.
4. Mock Uniswap V3 to arrange some fees for this `TokenisableRange`.
5. Call the range's `claimFee()` function.
6. Check the token balance of `vault` - this will be unchanged.
7. Check the token balance of the `TokenisableRange` treasury - this will have received the fees.

Tools Used

Code review, Foundry

Recommended Mitigation Steps

Consider fixing the `getVault` call as follows:

```
-       try RoeRouter(roerouter).getVault(address(TOKEN0.token), a
+       try RoeRouter(roerouter).getVault(address(TOKEN0.token), a
         vault = _vault;
       }
```

Additionally, you may want to make `RoeRouter` configurable, at least in the `TokenisableRange` constructor, because the version deployed at the currently hardcoded address `0x22Cc3f665ba4C898226353B672c5123c58751692` does not have a `getVault` method and is not upgradable.

Assessed type

Token-Transfer

`depositExactly` could be exploited

Submitted by xuwinnie

Severity: Low/Non-Critical

Lines of code

https://github.com/GoodEntry-io/ge/blob/3b80be0e86e1c01cd85906e9892e06540e12a842/contracts/TokenisableRange.sol#L244-L255

Vulnerability details

`depositExactly` benefits the user but harms the protocol. If gas goes lower and the token price goes higher (gas fee < 1e-8 token value), attacks could be profitable. I suggest we only allow whitlisted address (OPM) to call this function.

Assessed type

Context

peppelan (Good Entry) commented:

> @gzeon - I can't think of any scenario where this issue can award an attacker more than a handful of wei's of liquidity. Since `TokenisableRange` mints a baseline 1e18 liquidity at initialization, dust liquidity is necessarily meaningless.

> IMO it should be treated like a non-amplifiable rounding error, so QA seems more appropriate.

gzeon (judge) decreased severity to QA

`swapTokensForExactTokens` allows anyone to steal funds within the V3Proxy contract

Submitted by kutugu

Severity: Low/Non-Critical

Lines of code

https://github.com/GoodEntry-io/ge/blob/c7c7de57902e11e66c8186d93c5bb511b53a45b8/contracts/helper/V3Proxy.sol#L134

Impact

`swapTokensForExactTokens` will transfer the entire balance of `assetIn` to `msg.sender` after the swap, which allows anyone to steal funds stuck in the contract and conflicts with `emergencyWithdraw`.

Proof of Concept

```
    function emergencyWithdraw(ERC20 token) onlyOwner external {
        token.safeTransfer(msg.sender, token.balanceOf( address(
    }

    function swapTokensForExactTokens(uint amountOut, uint amoun
        require(path.length == 2, "Direct swap only");
        require(msg.sender == to, "Swap to self only");
        ERC20 ogInAsset = ERC20(path[0]);
        ogInAsset.safeTransferFrom(msg.sender, address(this), am
        ogInAsset.safeApprove(address(ROUTER), amountInMax);
        amounts = new uint[](2);
        amounts[0] = ROUTER.exactOutputSingle(ISwapRouter.ExactO
        amounts[1] = amountOut;
        ogInAsset.safeTransfer(msg.sender, ogInAsset.balanceOf(a
        ogInAsset.safeApprove(address(ROUTER), 0);
        emit Swap(msg.sender, path[0], path[1], amounts[0], amou
    }

    function swapTokensForExactETH(uint amountOut, uint amountIn
        require(path.length == 2, "Direct swap only");
        require(msg.sender == to, "Swap to self only");
```

```
        require(path[1] == ROUTER.WETH9(), "Invalid path");
        ERC20 ogInAsset = ERC20(path[0]);
        ogInAsset.safeTransferFrom(msg.sender, address(this), am
        ogInAsset.safeApprove(address(ROUTER), amountInMax);
        amounts = new uint[](2);
        amounts[0] = ROUTER.exactOutputSingle(ISwapRouter.ExactO
        amounts[1] = amountOut;
        ogInAsset.safeTransfer(msg.sender, amountInMax - amounts
        ogInAsset.safeApprove(address(ROUTER), 0);
        IWETH9 weth = IWETH9(ROUTER.WETH9());
        acceptPayable = true;
        weth.withdraw(amountOut);
        acceptPayable = false;
        payable(msg.sender).call{value: amountOut}("");
        emit Swap(msg.sender, path[0], path[1], amounts[0], amou
    }
```

`swapTokensForExactTokens` will transfer the entire balance of `assetIn` to `msg.sender` after swap. The correct approach should be like `swapTokensForExactETH`, only transfer `amountInMax - amount[0]`.

Recommended Mitigation Steps

Modify transfer amount

Assessed type

Context

gzeon (judge) decreased severity to QA

`GeVault`'s `depositAndStash` always reverts and ignores deposits to the active (and most important) ticker

Submitted by **3docSec**

Severity: Low/Non-Critical

Lines of code

https://github.com/GoodEntry-io/ge/blob/c7c7de57902e11e66c8186d93c5bb511b53a45b8/contracts/GeVault.sol#L359
https://github.com/GoodEntry-io/ge/blob/c7c7de57902e11e66c8186d93c5bb511b53a45b8/contracts/GeVault.sol#L360

Vulnerability details

In the new implementation, when deploying assets to the `TokenisableRange` instances, `GeVault` always sends a fixed amount for one asset and zero for the other asset:

```
if (newTickIndex > 1)
  depositAndStash(
    ticks[newTickIndex-2],
    baseTokenIsToken0 ? 0 : availToken0 / liquidityPerTick,
    baseTokenIsToken0 ? availToken1 / liquidityPerTick : 0
  );
if (newTickIndex > 0)
  depositAndStash(
    ticks[newTickIndex-1],
    baseTokenIsToken0 ? 0 : availToken0 / liquidityPerTick,
    baseTokenIsToken0 ? availToken1 / liquidityPerTick : 0
  );
if (newTickIndex < ticks.length)
  depositAndStash(
    ticks[newTickIndex],
    baseTokenIsToken0 ? availToken0 / liquidityPerTick : 0,
    baseTokenIsToken0 ? 0 : availToken1 / liquidityPerTick
  );
if (newTickIndex+1 < ticks.length)
  depositAndStash(
    ticks[newTickIndex+1],
    baseTokenIsToken0 ? availToken0 / liquidityPerTick : 0,
    baseTokenIsToken0 ? 0 : availToken1 / liquidityPerTick
  );
```

However, the active ticker (`newTickIndex`), by definition of active ticker, requires depositing non-zero amounts for both assets. If one of the two is zero, only a little

amount of liquidity will generated, and the provided asset will go mostly unused, causing the `TokenisableRange`'s deposit slippage check to fail.

Impact

`GeVault` will consistently fail to deploy assets to the active ticker. Since the active ticker is the one that's traded and generates fees, always depositing outside of that ticker makes the `GeVault` contract completely lose its intended functionality of concentrating liquidity where it produces value.

Proof of Concept

- Set up a `GeVault` with the following settings:
    - Token0: USDC (base token).
    - Token1: WETH.
    - Range 0: 1752e10, 2000e10.
- Have market sit at 1875 (0 will be active).
- Deposit to `GeVault` some USDC and WETH.
- Check the `TokenisableRange` balance of `GeVault`.

Tools Used

Code review, Foundry

Recommended Mitigation Steps

When deploying assets to the active ticker (not needed for the others), cap the sent amounts to what's needed to prevent the slippage check from failing:

```
        if (newTickIndex < ticks.length){
          (uint256 token0Amt, uint256 token1Amt) = ticks[newTickInde
            availToken0 / liquidityPerTick,
            availToken1 / liquidityPerTick
          );
          depositAndStash(
            ticks[newTickIndex],
            token0Amt,
            token1Amt
```

```
        );
    }
```

With the `exactTokenAmounts` utility function being added to `TokenisableRange`:

```
function getExactTokenAmounts(uint256 amount0, uint256 amount1
  address pool = V3_FACTORY.getPool(address(TOKEN0.token), addr
  (uint160 sqrtPriceX96,,,,,,) = IUniswapV3Pool(pool).slot0()
  uint160 lowerX96 = TickMath.getSqrtRatioAtTick(lowerTick);
  uint160 higherX96 = TickMath.getSqrtRatioAtTick(upperTick);

  uint128 inputLiquidity = LiquidityAmounts.getLiquidityForAmo

  return LiquidityAmounts.getAmountsForLiquidity(sqrtPriceX96,
}
```

The change to `GeVault` should be safe to apply because the `poolMatchesOracle()` check removes the need for a further slippage check.

Assessed type

Uniswap

Keref (Good Entry) commented:

> It's been mentioned in the review comments. Basically we don't add liquidity there for various reasons (likely already overallocated, unclear deposit strategy). Additionally, the `activeTickeIndex` doesn't mean the tick is active (maybe misnomer), but points to the first tick above. Actually, the probability that the price is in the tick is low and we just skip depositing in that case.

gzeon (judge) decreased severity to QA

peppelan (Good Entry) commented:

> Hi @gzeon, I would ask to reconsider the Med removal, considering the following:

1. The impact of this issue is no less than **#43** - in fact, it's even a little more impactful, because while with `#43` alone the `GeVault` can (by pure chance) end

up deploying to the active ticker, this issue prevents correct deployments 100% of times.

2. A similar finding was indeed reported in the original contest's **#414**, but **the team acted to mitigate the issue**, so I don't see any reason why reporting that the fix does not mitigate the impact (as per previous point) should not be in scope.

Keref (Good Entry) commented:

> This isnt a medium risk because there are no funds at risk and it follows our allocation strategy. Mitigation of the issue is to avoid reverting, not picking a different allocation strategy.

Missing withdrawal in `GeVault`'s `modifyTick` can cause `GeVault` to lock assets in discarded `TokenisableRange` instances

Submitted by **3docSec**

Severity: Low/Non-Critical

Lines of code

https://github.com/GoodEntry-io/ge/blob/c7c7de57902e11e66c8186d93c5bb511b53a45b8/contracts/GeVault.sol#L181

Vulnerability details

`GeVault` offers admins the `modifyTick` method to replace a ticker in its list. Despite being deployed behind a proxy, `TokenisableRange` instances don't have an upgradable implementation, so to upgrade the `TokenisableRange` code backing `GeVault`, `modifyTick` is the only option. Once `modifyTick` completes execution, the `TokenisableRange` liquidity tokens owned by `GeVault` can no longer be rescued because `GeVault` loses its reference to the removed `TokenisableRange`.

Impact

In the worst case scenario, a vulnerability is discovered in the `TokenisableRange` contract. The admin is forced to make the tough choice between keeping the vulnerability in the protocol, or locking the funds deployed by `GeVault` in one or more, if not all, of the referenced `TokenisableRange` contracts.

Proof of Concept

1. Start with a GeVault contract with 4 tickers and assets deployed on all of them.
2. Call `getTVL()` to record the total value locked - this is expected to be invariant.
3. Have `modifyTick()` replace one of the tickers.
4. Call `getTVL()` again to verify that part of the value locked in the contract has been lost.

Tools Used

Code review, Foundry

Recommended Mitigation Steps

Consider having `GeVault` withdraw its assets from the to-be-removed `TokenisableRange` instance before its address is deleted from storage:

```
    /// @notice Modify ticker
    /// @param tr New tick address
    /// @param index Tick to modify
    function modifyTick(address tr, uint index) public onlyOwner {
      (ERC20 t0,) = TokenisableRange(tr).TOKEN0();
      (ERC20 t1,) = TokenisableRange(tr).TOKEN1();
      require(t0 == token0 && t1 == token1, "GEV: Invalid TR");
+      removeFromTick(index, true);
      ticks[index] = TokenisableRange(tr);
      emit ModifyTick(tr, index);
    }
```

Also, it would be important to consider adding an extra parameter for stricter validation, because unlike the other currently-implemented ticker withdrawals where the tokens lended by the Roe pool are not withdrawn from the ticker, the full `aToken` balance must be removed to not be lost:

```
-   function removeFromTick(uint index) internal {
+   function removeFromTick(uint index, boolean requireFullBalanc
      TokenisableRange tr = ticks[index];
      address aTokenAddress = lendingPool.getReserveData(address(t
      uint aBal = ERC20(aTokenAddress).balanceOf(address(this));
      uint sBal = tr.balanceOf(aTokenAddress);

      // if there are less tokens available than the balance (beca
-      if (aBal > sBal) aBal = sBal;
+      if (aBal > sBal && !requireFullBalance) aBal = sBal;
      if (aBal > 0){
        lendingPool.withdraw(address(tr), aBal, address(this));
        tr.withdraw(aBal, 0, 0);
      }
    }
```

Assessed type

Uniswap

gzeon (judge) decreased severity to QA

Users withdrawing from GeVault lose their portion of fees

Submitted by 3docSec

Severity: Low/Non-Critical

Lines of code

https://github.com/GoodEntry-io/ge/blob/c7c7de57902e11e66c8186d93c5bb511b53a45b8/contracts/GeVault.sol#L230

Details

Referring to the other finding I submitted:

> New from fees rework: fees can still be stolen with a flash-loan on `GeVault`

The opposite is also true: for the same issue affecting `GeVault`'s `withdraw` function, when users withdraw their funds, `GeVault` does not award them with the fees accrued by their stake before withdrawal. This is submitted as low-risk finding because funds are not lost, and being value net-negative, it's not a valid attack path.

The suggested fix is similar to the quoted finding:

```
  function withdraw(uint liquidity, address token) public nonRee
    require(poolMatchesOracle(), "GEV: Oracle Error");
    if (liquidity == 0) liquidity = balanceOf(msg.sender);
    require(liquidity <= balanceOf(msg.sender), "GEV: Insufficie
    require(liquidity > 0, "GEV: Withdraw Zero");

+     removeFromAllTicks();
    uint vaultValueX8 = getTVL();
    uint valueX8 = vaultValueX8 * liquidity / totalSupply();
    amount = valueX8 * 10**ERC20(token).decimals() / oracle.getA
    uint fee = amount * getAdjustedBaseFee(token == address(toke

    _burn(msg.sender, liquidity);
-     removeFromAllTicks();
    ERC20(token).safeTransfer(treasury, fee);
    uint bal = amount - fee;

    if (token == address(WETH)){
      WETH.withdraw(bal);
      (bool success, ) = payable(msg.sender).call{value: bal}("
      require(success, "GEV: Error sending ETH");
    }
    else {
      ERC20(token).safeTransfer(msg.sender, bal);
    }

    // if pool enabled, deploy assets in ticks, otherwise just l
    if (isEnabled) deployAssets();
    emit Withdraw(msg.sender, token, amount, liquidity);
  }
```

Unused code

Submitted by 3docSec

Severity: Low/Non-Critical

Lines of code

https://github.com/GoodEntry-io/ge/blob/master/contracts/TokenisableRange.sol#L251
https://github.com/GoodEntry-io/ge/blob/master/contracts/TokenisableRange.sol#L252
https://github.com/GoodEntry-io/ge/blob/c7c7de57902e11e66c8186d93c5bb511b53a45b8/contracts/TokenisableRange.sol#L257
https://github.com/GoodEntry-io/ge/blob/c7c7de57902e11e66c8186d93c5bb511b53a45b8/contracts/TokenisableRange.sol#L231

Details

The rework of the fees handling and `TokenisableRange`'s `deposit` function made these blocks useless:

```
        uint val0 = u0 * ORACLE.getAssetPrice(address(TOKEN0.token
        uint val1 = u1 * ORACLE.getAssetPrice(address(TOKEN1.token



        (u0, u1) = getTokenAmountsExcludingFees(expectedAmount);



    if (fee0 > 0 || fee1 > 0){
        uint256 TOKEN0_PRICE = ORACLE.getAssetPrice(address(TOKEN0
        uint256 TOKEN1_PRICE = ORACLE.getAssetPrice(address(TOKEN1
        require (TOKEN0_PRICE > 0 && TOKEN1_PRICE > 0, "Invalid Ora
        // Calculate the equivalent liquidity amount of the non-ye
        // Assume linearity for liquidity in same tick range; calc
        uint token0decimals = TOKEN0.decimals;
        uint token1decimals = TOKEN1.decimals;
        feeLiquidity = newLiquidity * ( (fee0 * TOKEN0_PRICE / 10
                                      / ( (added0   * TOKEN0_PRICE
    }
```

They can be removed to save a considerable amount of gas.

Assessed type

call/delegatecall

Disclosures

C4 is an open organization governed by participants in the community.

C4 audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top