Lybra Finance

Date	August 2023

1 Executive Summary

This report presents the results of our engagement with **Lybra Finance Team** to review the **Lybra Protocol**.

The review was conducted over four and a half weeks, from **July 10, 2023** to **Aug 9, 2023**, by **George Kobakhidze** and **Sergii Kravchenko**. A total of 7 person-weeks were spent.

The Lybra Protocol is a DeFi system that focuses on collateral-based stablecoin lending and management. A particular feature of this protocol is its focus on the stablecoin holders and Liquid Staking Derivative tokens for Ethereum staking as collateral. The resulting product of the system is an interest-bearing rebasing stablecoin called `EUSD`. With the V2 version of the protocol, which is what is in the scope of this audit, the system introduces more LSD tokens along with Lido's `stETH` from the first version such as those from RocketPool and Binance. Lybra Protocol V2 also introduces `PeUSD` a non-rebasing USD stablecoin that features LayerZero's Omnichain capabilities. `PeUSD` may be minted with `EUSD` or with non-rebasing LSD tokens that are introduced in the V2. `PeUSD` may also accommodate easier integrations with other DeFi protocols as the rebasing `EUSD` may cause accounting issues.

The repository is organized and the code is written well with plentiful comments and elegant design choices. As Lybra Protocol is a complex system, users should expect to understand and digest a multitude of parameters, many of which may be changed by governance and many of which are hardcoded and may not.

At the end of the audit, the engagement was extended to verify fixes and some other code changes made during the audit.

2 Scope

Our review focused on the commit hash 48c98f288c77f57fa17e87964394f98e1e2ee636. The list of files in scope can be found in the Appendix.

Following the audit, all commits between 48c98f288c77f57fa17e87964394f98e1e2ee636 and 90285107de8a6754954c303cd69d97b5fdb4e248 were reviewed as well.

2.1 Objectives

Together with the **Lybra Finance** team, we identified the following priorities for our review:

1. Correctness of the implementation, consistent with the intended functionality and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.
3. Review the logic of all vault types and the implementation of their core functions such as depositing, borrowing, withdrawing, liquidating, redeeming, and distributing excess income through interest realization for rebase assets.
4. The Governance checks for proposals and voting are not bypassable by a malicious actor.
5. Permission checks in the `LybraConfigurator` contract work as intended.
6. The rewards staking, mining, and distribution logic work as intended in contracts such as `ProtocolRewardsPool` and `EUSDMiningIncentives`.

3 System Overview

Lybra is an interest-bearing stablecoin protocol backed by different LSD (liquid staking derivatives).

3.1 Stablecoins

- `EUSD` - an `ERC-20` rebase token representing a yield-bearing stablecoin.
- `peUSD` - a regular `ERC-20` stablecoin that can be converted from `EUSD`. The `peUSD` token doesn't earn any interest, but users that convert `EUSD` will still be getting interest from their deposited `EUSD` tokens. Also has an LayerZero Omnichain component making it a OFTV2 token as well for multi-chain capabilities.

3.2 Vaults

There are two types of vaults in the protocol. One of them takes a LSD as collateral and mints `EUSD` (`LybraEUSDVaultBase`), and the other type mints `peUSD` (`LybraPeUSDVaultBase`). The following vaults are implemented:

- `LybraStETHVault` - takes `stETH` (staked `ETH` by Lido) as collateral and mints `EUSD`.
- `LybraWstETHVault` - takes `WstETH` (wrapped `stETH`) as collateral and mints `peUSD`.
- `LybraRETHVault` - takes `RETH` (Rocket Pool `ETH`) as collateral and mints `peUSD`.
- `LybraWbETHVault` - takes `WBETH` (Wrapped Binance `ETH`) as collateral and mints `peUSD`.

The main difference between `peUSD` and `EUSD` vaults is that the collateral of the `EUSD` vault is a rebase token. So for `peUSD` vaults, every depositor's collateral value is supposed to grow over time due to ETH staking rewards. Unlike that, every individual `stETH` collateral deposit isn't directly growing due to its rebase nature. So the excess of `stETH` tokens in the vault contract is sold for `EUSD` to anyone willing to buy it. The shares of the profit are burnt in favor of every `EUSD` holder, which increases their balances due to the token's rebase nature.

3.3 Mining

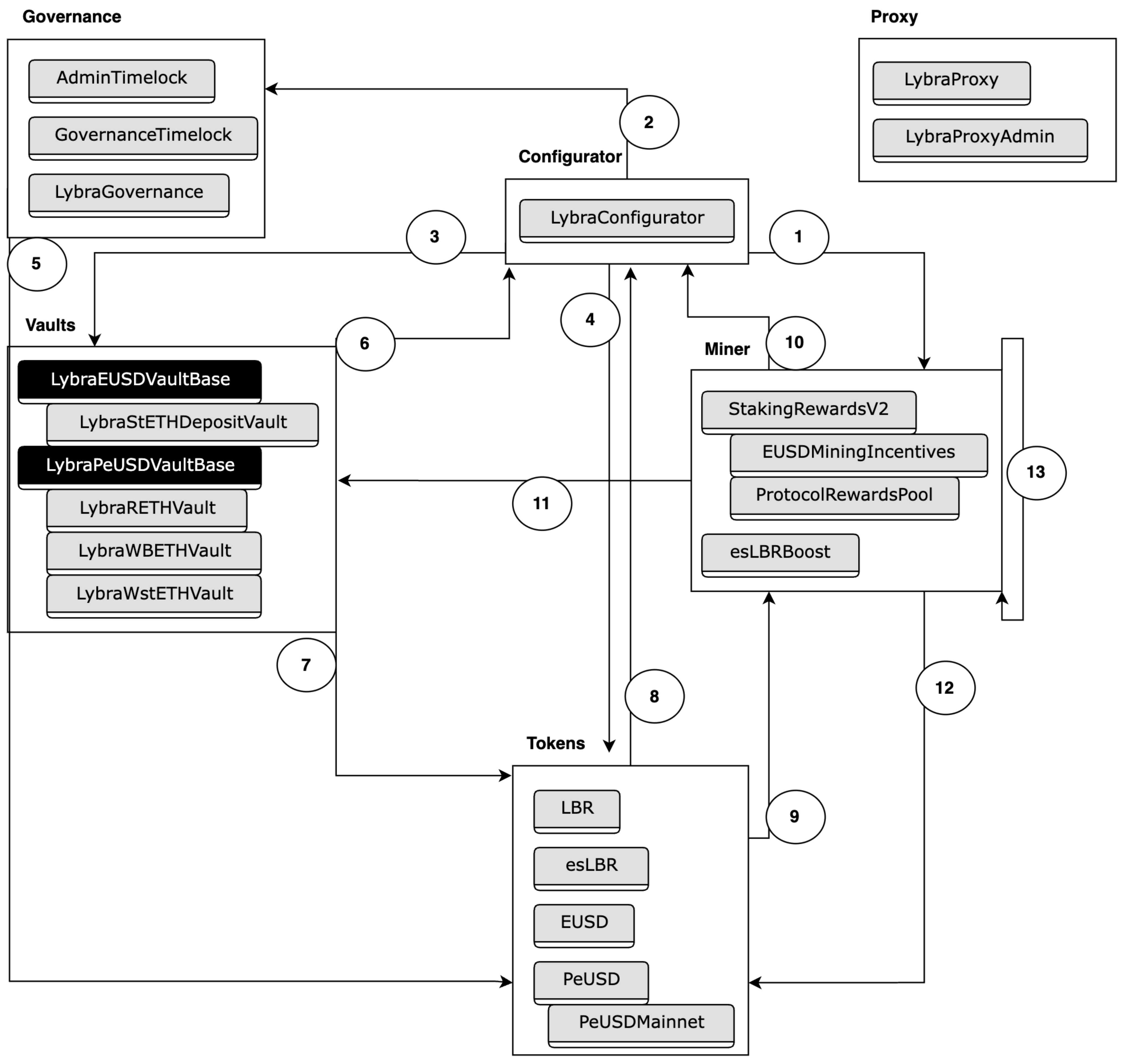
The system has two reward pools that are forked from Synthetix StakingRewards with some changes made:

- `EUSDMiningIncentives` - a staking contract that rewards borrowers of `EUSD` and `peUSD`. The users are getting rewards in the form of `esLBR` (escrowed `LBR` tokens).
- `esLBRBoost` - a contract allowing users to lock their `LBR` in exchange for boosted rewards in `EUSDMiningIncentives`.
- `ProtocolRewardsPool` - a staking contract that rewards holders of `esLBR` with `peUSD` or other external stablecoins.

3.4 System Diagram

Please find below a non-exhaustive diagram of the Lybra Protocol system that focuses on calls and data flow within the system.

1. Configurator - Miner.
    - Reward logic such as `notifyRewardAmount` and `refreshReward`.
2. Configurator - Governance.
    - Access control checks like `checkRole` and `checkOnlyRole`.
3. Configurator - Vaults.
    - Vault information such as `getVaultType`.
4. Configurator - Tokens.
    - Token transfer logic that happens during reward distribution such as `approve` and `safeTransfer`, as well as `convertToPeUSD`.
5. Governance - Tokens.
    - Voting logic, specifically `getPastVotes` from `esLBR` tokens.
6. Vaults - Configurator.
    - Parameter retrievals such as `getEUSDAddress`, `getPeUSDAddress`, `isRedemptionProvider`, `mintVaultMaxSupply`, and others for successful vault management.
    - Reward distribution logic through calls like `refreshMintReward` and `distributeReward`.
7. Vaults - Tokens.
    - Token administration logic such as `transfer` and `allowance` calls but also highly privileged methods such as `burn` and `mint`.
8. Tokens - Configurator.
    - Vault parameter retrieval such as `vaultMintPaused`, `mintVault[]`, `getProtocolRewardsPool`, `tokenMiner`.
9. Tokens - Miner.
    - Reward logic through calls such as `refreshReward`.
10. Miner - Configurator.
    - Information retrieval such as `getEUSDAddress`, `isRedemptionProvider`, `mintVault[]`.
11. Miner - Vaults.
    - Token and vault information retrieval such as `getPoolTotalCirculation` and `getBorrowedOf`.
12. Miner - Tokens.
    - Token information retrieval such as `balanceOf`, `totalSupply`, and `getUserBoost`.
    - Token administration such as `transfer` and highly privileged calls such as `mint` and `burn`.
13. Miner - Miner.
    - External calls between miner contracts such as `totalSupply`, `totalStaked`, `stakedOf`, and `userLockStatus`.



4 Security Specification

This section describes, **from a security perspective**, the expected behavior of the system under audit. It is not a substitute for documentation. The purpose of this section is to identify specific security properties that were validated by the audit team.

4.1 Actors

The relevant actors are listed below with their respective abilities:

- **Lybra Finance team - contract owners and deployers.** The contract team currently manages the deployment and initial configuration of the contracts.
- **Lybra Governance.** `LBR` (and `esLBR`) token holders who can to vote, pass, and execute proposals.
- **Vault owners.** The users that supply the collateral and leverage it for minting the Lybra Protocol's stablecoins.
- **Token holders.** The end-user holders of the `EUSD` and `PeUSD` tokens.
- **Keepers.** Users and/or bots that call functions such as excess income distribution and vault liquidations of those vaults that are below the necessary collateral ratios.
- **Price Oracles.** Third-party contracts that provide price feeds for various tokens for the Lybra Protocol.

4.2 Trust Model

In any system, it's important to identify what trust is expected/required between various actors. For this audit, we established the following trust model:

- **Lybra Finance team (contract owners and deployers).** In the current scope of this protocol, there is in fact a large amount of trust put into the deployer of the contracts to initialize them properly. As seen in the System Overview section, the `LybraConfigurator` contract, for example, contains numerous mission-critical variables that are used by other contracts in the system. Setting those up correctly in the beginning is the responsibility of the deployer. However, as this is all on a public blockchain, it would be possible to identify if the Lybra deployer has made a mistake in the configurations, so this is not a significant issue vector. That being said, it is important to point out that upon deployment of the `GovernanceTimelock` contract, the `msg.sender` explicitly gets a `DAO` and a `GOV` role, the former of which also allows it to bypass any checks that are protected by the `checkRole` function as it specifically allows a `DAO` role to go through. As a result, there is significant trust involved with the Lybra deployer for him not to get compromised as it could be disastrous for the system.
- **Lybra Governance.** As with most DAOs, the holders of governance tokens (`LBR` and `esLBR`) are trusted with voting, passing, and executing proposals as they see fit. As a result, the Governance is trusted not to pass malicious proposals.
- **Vault owners.** No particular trust assumptions are made about the vault owners. They are expected to maintain a healthy collateral-to-borrow ratio. However, if they fail to do so, there are incentive mechanisms to enable other users, such as keepers, to liquidate such vaults. Vault owners can also become redemption providers to earn additional fees and receive a boost in their rewards, so other users can use their vaults to redeem tokens directly. However, if the vault owner is marked as a redemption provider but in fact doesn't have enough liquidity to support a redemption, it will simply revert and not stop the overall system from working.
- **Token holders.** Similarly, no specific trust assumptions are made for token holders. The interactions they can make with the protocol are all trustless.

- **Keepers.** While they are not explicitly trusted to maintain the Lybra Protocol system, keepers are crucial for the overall protocol health as they run liquidations and assist in staking reward distribution through purchases of excess amounts of Liquid Staking Tokens for `EUSD`. Industry-standard mechanisms are employed to get the keepers to interact with the system, such as Dutch Auction discounts, additional keeper fees, and so on. As a result, the overall user base of the chain where the Lybra Protocol is deployed is assumed to have enough rational actors to execute the incentives and move the mechanisms forward.

4.3 Security Properties

The following is a non-exhaustive list of security properties that were identified in this audit:

- **Hardcoded properties.** The Lybra Protocol is complex and maintains many of properties necessary for its operations, such as collateral ratios, fee ratios, boosts, Dutch Auction settings, and so on. Some of these are set by the Governance through the `LybraConfigurator` contract but some of them are hardcoded in the system. It is important for users to identify if they are comfortable with both potentially changing parameters by the governance and hardcoded parameters that will not change throughout the lifecycle of the system.
- **Proxy.** From the documentation, the comments, and the files in scope, it appears that a proxy deployment specifically for the `LybraConfigurator` contract might be possible. As mentioned throughout the report, this is a mission-critical contract and users should be aware of risks associated with its upgradeability as a faulty upgrade might critically damage the system.
- **Lybra Deployer.** As mentioned above, the Lybra Deployer is explicitly given the `DAO` and `GOV` roles upon deploying the `GovernanceTimelock` contract. While the `GOV` role is the admin of other roles, the `DAO` role allows the owner of this private key to do almost anything they want with the system. For example, they can set a new contract to be a `LBR` miner, mint as many `LBR` tokens as they want, and then take full control of the DAO's governance. However unlikely, a malicious actor may compromise these keys, so this is a risk users need to take into account. It is understandable to approach the deployment of this system with some safeguards like this in the beginning that would allow the Lybra Finance team to surgically address issues such as those in configurations. However, it would be critical to then revoke the `DAO` role from that private address as the system matures. Otherwise, this poses a significant risk.

5 Post Audit Verifications

During the verification week after the audit, the following points have been identified as potential improvements to the additional changes provided:

- **Improve the logic of the rigidRedemption** commit. The `checkWithdrawal` check is introduced to ensure collateral doesn't leave the system within the 3 day period of having been deposited. However, the check is made on the user requested the rigid redemption whereas the collateral comes from the provider. Therefore, it seems more applicable to apply the `checkWithdrawal` check on the `provider`.
- **Improve the logic of the rigidRedemption** commit. The emitted event `RigidRedemption` has an incorrect amount emitted. If the logic of the `withdraw` function in the same contract is to be followed, the event should emit the amount that is corrected by the `checkWithdrawal` function instead of what the user first requested to withdraw (or rigid redeem).
- **fix getVaultWeight logic** commit. The vault is now checked to be active if there is no special weight assigned to it. However, if weight is assigned but it is disabled (so `mintVault` gives false), this will still return a value, which may be in conflict with the new change that appears to only want to give a `vaultWeight` back if it is enabled.

6 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

6.1 Re-Entrancy Risks Associated With External Calls With Other Liquid Staking Systems. Major ✓ Fixed

Resolution

Fixed in commit f43b7cd5135872143cc35f40cae95870446d0413 by introducing reentrancy guards.

Description

As part of the strategy to integrate with Liquid Staking tokens for Ethereum staking, the Lybra Protocol vaults are required to make external calls to Liquid Staking systems.

For example, the `depositEtherToMint` function in the vaults makes external calls to deposit Ether and receive the LSD tokens back. While external calls to untrusted third-party contracts may be dangerous, in this case, the Lybra Protocol already extends trust assumptions to these third parties simply through the act of accepting their tokens as collateral. Indeed, in some cases the contract addresses are even hardcoded into the contract and called directly instead of relying on some registry:

contracts/lybra/pools/LybraWstETHVault.sol:L21-L40

```
contract LybraWstETHVault is LybraPeUSDVaultBase {
    Ilido immutable lido;
    //WstETH = 0x7f39C581F595B53c5cb19bD0b3f8dA6c935E2Ca0;
    //Lido = 0xae7ab96520DE3A18E5e111B55EaAb095312D7fE84;
    constructor(address _lido, address _asset, address _oracle, address _config) LybraPeUSDVaultBase(_asset, _oracle, _config) {
        lido = Ilido(_lido);
    }

    function depositEtherToMint(uint256 mintAmount) external payable override {
        require(msg.value >= 1 ether, "DNL");
        uint256 sharesAmount = lido.submit{value: msg.value}(address(configurator));
        require(sharesAmount != 0, "ZERO_DEPOSIT");
        lido.approve(address(collateralAsset), msg.value);
        uint256 wstETHAmount = IWstETH(address(collateralAsset)).wrap(msg.value);
        depositedAsset[msg.sender] += wstETHAmount;
        if (mintAmount > 0) {
            _mintPeUSD(msg.sender, msg.sender, mintAmount, getAssetPrice());
        }
        emit DepositEther(msg.sender, address(collateralAsset), msg.value,wstETHAmount, block.timestamp);
    }
```

In that case, depending on the contract, it may be known what contract is being called, and the risk may be assessed as far as what logic may be executed.

However, in the cases of `BETH` and `rETH`, the calls are being made into a proxy and a contract registry of a DAO (RocketPool's DAO) respectively.

contracts/lybra/pools/LybraWbETHVault.sol:L15-L32

```
contract LybraWBETHVault is LybraPeUSDVaultBase {
    //WBETH = 0xa2e3356610840701bdf5611a53974510ae27e2e1
    constructor(address _asset, address _oracle, address _config)
        LybraPeUSDVaultBase(_asset, _oracle, _config) {}

    function depositEtherToMint(uint256 mintAmount) external payable override {
        require(msg.value >= 1 ether, "DNL");
        uint256 preBalance = collateralAsset.balanceOf(address(this));
        IWBETH(address(collateralAsset)).deposit{value: msg.value}(address(configurator));
        uint256 balance = collateralAsset.balanceOf(address(this));
        depositedAsset[msg.sender] += balance - preBalance;

        if (mintAmount > 0) {
            _mintPeUSD(msg.sender, msg.sender, mintAmount, getAssetPrice());
        }

        emit DepositEther(msg.sender, address(collateralAsset), msg.value,balance - preBalance, block.timestamp);
    }
```

contracts/lybra/pools/LybraRETHVault.sol:L25-L42

```
constructor(address _rocketStorageAddress, address _rETH, address _oracle, address _config)
    LybraPeUSDVaultBase(_rETH, _oracle, _config) {
    rocketStorage = IRocketStorageInterface(_rocketStorageAddress);
}

function depositEtherToMint(uint256 mintAmount) external payable override {
    require(msg.value >= 1 ether, "DNL");
    uint256 preBalance = collateralAsset.balanceOf(address(this));
    IRocketDepositPool(rocketStorage.getAddress(keccak256(abi.encodePacked("contract.address", "rocketDepositPool")))).deposit{v
    uint256 balance = collateralAsset.balanceOf(address(this));
    depositedAsset[msg.sender] += balance - preBalance;

    if (mintAmount > 0) {
        _mintPeUSD(msg.sender, msg.sender, mintAmount, getAssetPrice());
    }

    emit DepositEther(msg.sender, address(collateralAsset), msg.value,balance - preBalance, block.timestamp);
}
```

As a result, it is impossible to make any guarantees for what logic will be executed during the external calls. Namely, reentrancy risks can't be ruled out, and the damage could be critical to the system. While the trust in these parties isn't in question, it would be best practice to avoid any additional reentrancy risks by placing reentrancy guards. Indeed, in the `LybraRETHVault` and `LybraWbETHVault` contracts, one can see the possible damage as the calls are surrounded in a `preBalance <-> balance` pattern.

The whole of third party Liquid Staking systems' operations need not be compromised, only these particular parts would be enough to cause critical damage to the Lybra Protocol.

Recommendation

After conversations with the Lybra Finance team, it has been assessed that reentrancy guards are appropriate in this scenario to avoid any potential reentrancy risk, which is exactly the recommendation this audit team would provide.

6.2 The Deployer of `GovernanceTimelock` Gets Privileged Access to the System. Major ✓ Fixed

Resolution

As per discussions with the Lybra Finance team, this has been acknowledged as a temporary measure to configure anything before the launch of V2. Following the discussions, the Lybra Finance team has revoked the deployer's permissions in transaction 0x12c95eec095f7e24abc6a127f378f9f0fb3a0021aeac82b487c11afa01b793af and updated the `GovernanceTimelock` code in commit 77e8bc3664fb1b195fd718c2ce1d49af8530f981 to instead introduce a multisig address that will have the `ADMIN` role whose only permission within the configurator contract is to pause the minting function in emergency situations.

Description

The `GovernanceTimelock` contract is responsible for Roles Based Access Control management and checks in the Lybra Protocol. It offers two functions specifically that check if an address has the required role - `checkRole` and `checkOnlyRole`:

contracts/lybra/governance/GovernanceTimelock.sol:L24-L30

```
function checkRole(bytes32 role, address _sender) public view  returns(bool){
    return hasRole(role, _sender) || hasRole(DAO, _sender);
}

function checkOnlyRole(bytes32 role, address _sender) public view  returns(bool){
    return hasRole(role, _sender);
}
```

In `checkRole`, the contract also lets an address with the role `DAO` bypass the check altogether, making it a powerful role.

For initial role management, when the `GovernanceTimelock` contract gets deployed, its constructor logic initializes a few roles, assigns relevant admin roles, and, notably, assigns the `DAO` role to the contract, and the `DAO` and the `GOV` role to the deployer.

contracts/lybra/governance/GovernanceTimelock.sol:L14-L23

```
constructor(uint256 minDelay, address[] memory proposers, address[] memory executors, address admin) TimelockController(minDelay

    _setRoleAdmin(DAO, GOV);
    _setRoleAdmin(TIMELOCK, GOV);
    _setRoleAdmin(ADMIN, GOV);
    _grantRole(DAO, address(this));
    _grantRole(DAO, msg.sender);
    _grantRole(GOV, msg.sender);
}
```

The assignment of such powerful roles to a single private key with the deployer has inherent risks. Specifically in our case, the `DAO` role alone as we saw may bypass many checks within the Lybra Protocol, and the `GOV` role even has role management privileges.

However, it does make sense to assign such roles at the beginning of the deployment to finish initialization and assign the rest of the roles. One could argue that having access to the `DAO` role in the early stages of the system's life could allow for quick disaster recovery in the event of incidents as well. Though, it is still dangerous to hold privileges for such a system in a single address as we have seen over the last years in security incidents that have to do with compromised keys.

Recommendation

While redesigning the deployment process to account for a lesser-privileged deployer would be ideal, the Lybra Finance team should at least transfer ownership as soon as the deployment is complete to minimize compromised private key risk.

6.3 The `configurator.getEUSDMaxLocked()` Condition Can Be Bypassed During a Flashloan

Medium ✓ Fixed

Resolution

Fixed in f6c3afb5e48355c180417b192bd24ba294f77797 by checking `eUSD` amount after flash loan.

Description

When converting `EUSD` tokens to `peUSD`, there is a check that limits the total amount of `EUSD` that can be converted:

contracts/lybra/token/PeUSDMainnet.sol:L74-L77

```
function convertToPeUSD(address user, uint256 eusdAmount) public {
    require(_msgSender() == user || _msgSender() == address(this), "MDM");
    require(eusdAmount != 0, "ZA");
    require(EUSD.balanceOf(address(this)) + eusdAmount <= configurator.getEUSDMaxLocked(),"ESL");
```

The issue is that there is a way to bypass this restriction. An attacker can get a flash loan (in `EUSD`) from this contract, essentially reducing the visible amount of locked tokens (`EUSD.balanceOf(address(this))`).

Recommendation

Multiple approaches can solve this issue. One would be adding reentrancy protection. Another one could be keeping track of the borrowed amount for a flashloan.

6.4 Liquidation Keepers Automatically Become `eUSD` Debt Providers for Other Liquidations. Medium

✓ Fixed

Resolution

Fixed in commit bbcf1867ef66cfdcd4b4fd26df39518048fbde1f by adding an alternative check to the allowance flag to see if `msg.sender == provider`, thus removing the danger for solo liquidators that don't use another provider.

Description

One of the most important mechanisms in the Lybra Protocol is the liquidation of poorly collateralized vaults. For example, if a vault is found to have a collateralization ratio that is too small, a liquidator may provide debt tokens to the protocol and retrieve the vault collateral at a discount:

contracts/lybra/pools/base/LybraEUSDVaultBase.sol:L148-L170

```
function liquidation(address provider, address onBehalfOf, uint256 assetAmount) external virtual {
    uint256 assetPrice = getAssetPrice();
    uint256 onBehalfOfCollateralRatio = (depositedAsset[onBehalfOf] * assetPrice * 100) / borrowed[onBehalfOf];
    require(onBehalfOfCollateralRatio < badCollateralRatio, "Borrowers collateral ratio should below badCollateralRatio");

    require(assetAmount * 2 <= depositedAsset[onBehalfOf], "a max of 50% collateral can be liquidated");
    require(EUSD.allowance(provider, address(this)) != 0, "provider should authorize to provide liquidation EUSD");
    uint256 eusdAmount = (assetAmount * assetPrice) / 1e18;

    _repay(provider, onBehalfOf, eusdAmount);
    uint256 reducedAsset = assetAmount * 11 / 10;
    totalDepositedAsset -= reducedAsset;
    depositedAsset[onBehalfOf] -= reducedAsset;
    uint256 reward2keeper;
    if (provider == msg.sender) {
        collateralAsset.safeTransfer(msg.sender, reducedAsset);
    } else {
        reward2keeper = (reducedAsset * configurator.vaultKeeperRatio(address(this))) / 110;
        collateralAsset.safeTransfer(provider, reducedAsset - reward2keeper);
        collateralAsset.safeTransfer(msg.sender, reward2keeper);
    }
    emit LiquidationRecord(provider, msg.sender, onBehalfOf, eusdAmount, reducedAsset, reward2keeper, false, block.timestamp);
}
```

To liquidate the vault, the **liquidator** needs to transfer debt tokens from the **provider** address, which in turn needs to have had approved allowance of the token for the vault:

contracts/lybra/pools/base/LybraEUSDVaultBase.sol:L154

```
require(EUSD.allowance(provider, address(this)) != 0, "provider should authorize to provide liquidation EUSD");
```

The allowance doesn't need to be large, it only needs to be non-zero. While it is true that in the `superLiquidation` function the allowance check is for `eusdAmount`, which is the amount associated with `assetAmount` (the requested amount of collateral to be liquidated), the liquidator could simply call the maximum of the allowance the provider has given to the vault and then repeat the liquidation process. The allowance does not actually decrease throughout the liquidation process.

contracts/lybra/pools/base/LybraEUSDVaultBase.sol:L191

```
require(EUSD.allowance(provider, address(this)) >= eusdAmount, "provider should authorize to provide liquidation EUSD");
```

Notably, this address doesn't have to be the same one as the liquidator. In fact, there are no checks on whether the liquidator has an agreement or allowance from the provider to use their tokens in this particular vault's liquidation. The contract only checks to see if the provider has `EUSD` allowance for the vault, and how to split the rewards if the provider is different from the liquidator:

contracts/lybra/pools/base/LybraEUSDVaultBase.sol:L162-L168

```
if (provider == msg.sender) {
    collateralAsset.safeTransfer(msg.sender, reducedAsset);
} else {
    reward2keeper = (reducedAsset * configurator.vaultKeeperRatio(address(this))) / 110;
    collateralAsset.safeTransfer(provider, reducedAsset - reward2keeper);
    collateralAsset.safeTransfer(msg.sender, reward2keeper);
}
```

In fact, this is a design choice of the system to treat the allowance to the vault as an agreement to become a public provider of debt tokens for the liquidation process. It is important to note that there are incentives associated with being a provider as they get the collateral asset at a discount.

However, it is not obvious from documentation at the time of the audit nor the code that an address having a non-zero `EUSD` allowance for the vault automatically allows other users to use that address as a provider. Indeed, many general-purpose liquidator bots use their tokens during liquidations, using the same address for both the liquidator and the provider. As a result, this would put that address at the behest of any other user who would want to utilize these tokens in liquidations. The user might not be comfortable doing this trade in any case, even at a discount.

In fact, due to this mechanism, even during consciously initiated liquidations MEV bots could spot this opportunity and front-run the liquidator's transaction. A frontrunner could put themselves as the keeper and the original user as the provider, grabbing the `reward2keeper` fee and leaving the original address with fewer rewards and failed gas after the liquidation.

Recommendation

While the mechanism is understood to be done for convenience and access to liquidity as a design decision, this could put unaware users in unfortunate situations of having performed a trade without explicit consent. Specifically, the MEV attack vector could be executed and repeated without fail by a capable actor monitoring the mempool. Consider having a separate, explicit flag for allowing others to use a user's tokens during liquidation, thus also accommodating solo liquidators by removing the MEV attack vector. Consider explicitly mentioning these mechanisms in the documentation as well.

6.5 Use the Same Solidity Version Across Contracts. Minor ✓ Fixed

Resolution

Fixed in commit 33af5c92044cd84c7f69eb8a55316d1e8535ea84 and commit b1c6ac26b262ec6011c14297583d67d9e3e94326.

Description

Most contracts use the same Solidity version with `pragma solidity ^0.8.17`. The only exception is the `StakingRewardsV2` contract which has `pragma solidity ^0.8`.

contracts/lybra/miner/stakerewardV2pool.sol:L2

```
pragma solidity ^0.8;
```

Recommendation

If all contracts will be tested and utilized together, it would be best to utilize and document the same version within all contract code to avoid any issues and inconsistencies that may arise across Solidity versions.

6.6 Duplication of Bad Collateral Ratio Minor Acknowledged

Resolution

The Lybra Finance team has acknowledged this as a choice by design and provided the following note:

> The liquidation ratio for each eUSD vault is fixed, and this has been stated in our docs. Therefore, we will keep it unchanged.

Description

It is possible to set a bad collateral ratio in the `LybraConfigurator` contract for any vault:

contracts/lybra/configuration/LybraConfigurator.sol:L137-L141

```
function setBadCollateralRatio(address pool, uint256 newRatio) external onlyRole(DAO) {
    require(newRatio >= 130 * 1e18 && newRatio <= 150 * 1e18 && newRatio <= vaultSafeCollateralRatio[pool] + 1e19, "LNA");
    vaultBadCollateralRatio[pool] = newRatio;
    emit SafeCollateralRatioChanged(pool, newRatio);
}
```

But in the `LybraEUSDVaultBase` contract, this value is fixed and cannot be changed:

contracts/lybra/pools/base/LybraEUSDVaultBase.sol:L19

```
uint256 public immutable badCollateralRatio = 150 * 1e18;
```

This duplication of values can be misleading at some point. It's better to make sure you cannot change the bad collateral ratio in the `LybraConfigurator` contract for some types of vaults.

6.7 Missing Events. Minor ✓ Fixed

Resolution

Fixed in commit 518ef434c6f89c7747373b6ae178d9665d3637f2.

Description

In a few cases in the Lybra Protocol system, there are contracts that are missing events in significant scenarios, such as important configuration changes like a price oracle change. Consider implementing more events in the below examples.

Examples

- No events in the contract:

contracts/lybra/miner/esLBRBoost.sol:L10-L30

```
contract esLBRBoost is Ownable {
    esLBRLockSetting[] public esLBRLockSettings;
    mapping(address => LockStatus) public userLockStatus;
    IMiningIncentives public miningIncentives;

    // Define a struct for the lock settings
    struct esLBRLockSetting {
        uint256 duration;
        uint256 miningBoost;
    }

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

    // Constructor to initialize the default lock settings
    constructor(address _miningIncentives) {
```

- Missing an event during a premature unlock:

contracts/lybra/miner/ProtocolRewardsPool.sol:L125-L135

```
function unlockPrematurely() external {
    require(block.timestamp + exitCycle - 3 days > time2fullRedemption[msg.sender], "ENW");
    uint256 burnAmount = getReservedLBRForVesting(msg.sender) - getPreUnlockableAmount(msg.sender);
    uint256 amount = getPreUnlockableAmount(msg.sender) + getClaimAbleLBR(msg.sender);
    if (amount > 0) {
        LBR.mint(msg.sender, amount);
    }
    unstakeRatio[msg.sender] = 0;
    time2fullRedemption[msg.sender] = 0;
    grabableAmount += burnAmount;
}
```

- Missing events for setting important configurations such as `setToken`, `setLBROracle`, and `setPools`:

contracts/lybra/miner/EUSDMiningIncentives.sol:L87-L102

```
function setToken(address _lbr, address _eslbr) external onlyOwner {
    LBR = _lbr;
    esLBR = _eslbr;
}

function setLBROracle(address _lbrOracle) external onlyOwner {
    lbrPriceFeed = AggregatorV3Interface(_lbrOracle);
}

function setPools(address[] memory _vaults) external onlyOwner {
    require(_vaults.length <= 10, "EL");
    for (uint i = 0; i < _vaults.length; i++) {
        require(configurator.mintVault(_vaults[i]), "NOT_VAULT");
    }
    vaults = _vaults;
}
```

- Missing events for setting important configurations such as `setRewardsDuration` and `setBoost`:

contracts/lybra/miner/stakerewardV2pool.sol:L121-L130

```
// Allows the owner to set the rewards duration
function setRewardsDuration(uint256 _duration) external onlyOwner {
    require(finishAt < block.timestamp, "reward duration not finished");
    duration = _duration;
}

// Allows the owner to set the boost contract address
function setBoost(address _boost) external onlyOwner {
    esLBRBoost = IesLBRBoost(_boost);
}
```

- Missing event during what is essentially staking `LBR` into `esLBR` (such as in `ProtocolRewardsPool.stake()`). Consider an appropriate event here such as `StakeLBR`:

contracts/lybra/miner/esLBRBoost.sol:L55-L58

```
if(useLBR) {
    IesLBR(miningIncentives.LBR()).burn(msg.sender, lbrAmount);
    IesLBR(miningIncentives.esLBR()).mint(msg.sender, lbrAmount);
}
```

Recommendation

Implement additional events as appropriate.

6.8 Incorrect Interfaces Minor ✓ Fixed

Resolution

Fixed in commit 90285107de8a6754954c303cd69d97b5fdb4e248, commit 0ac9cd732b601d0baef2690ef9f9f02cda989331, and commit 518ef434c6f89c7747373b6ae178d9665d3637f2.

Description

In a few cases, incorrect interfaces are used on top of contracts. Though the effect is the same as the contracts are just tokens and follow the same interfaces, it is best practice to implement correct interfaces.

- `IPeUSD` is used instead of `IEUSD`

contracts/lybra/configuration/LybraConfigurator.sol:L60

```
IPeUSD public EUSD;
```

- `IPeUSD` is used instead of `IEUSD`

contracts/lybra/configuration/LybraConfigurator.sol:L109

```
if (address(EUSD) == address(0)) EUSD = IPeUSD(_eusd);
```

- `IesLBR` instead of `ILBR`

contracts/lybra/miner/ProtocolRewardsPool.sol:L29

```
IesLBR public LBR;
```

- `IesLBR` instead of `ILBR`

contracts/lybra/miner/ProtocolRewardsPool.sol:L57

```
LBR = IesLBR(_lbr);
```

Recommendation

Implement correct interfaces for consistency.

6.9 The ETH Staking Rewards Distribution Tradeoff

Description

When users deposit `stETH` to the `LybraStETHDepositVault`, they give up their rewards from `ETH` staking. In exchange, all the rewards from `stETH` are going to the `EUSD` holders proportionally. So every user is incentivized to borrow as much `EUSD` as possible to get their fair share of rewards. Additionally, if any other `LybraEUSDVaultBase` vault is added with a lower yield LSD, the depositors of that vault will receive an extra portion of `stETH` rewards. While the depositors of `LybraStETHDepositVault` will start getting less.

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
contracts/lybra/Proxy/LybraProxy.sol	`ab1ed4e8e31a501c8d1db02086fdc6e883626948`
contracts/lybra/Proxy/LybraProxyAdmin.sol	`8d745b8fb75d3a9f88db0b2253a2d5ef62e1ae0c`
contracts/lybra/configuration/LybraConfigurator.sol	`4f439afd7578a4f1f8eceae61ab41435d97d1988`
contracts/lybra/governance/AdminTimelock.sol	`535affc02992b45c3584087d361e893cad4b7ec4`
contracts/lybra/governance/GovernanceTimelock.sol	`91b1cd13d0a86ca27e94771b0e52725b98dcf730`
contracts/lybra/governance/LybraGovernance.sol	`a62a2c595ae33903375af7685f368a56100c957e`
contracts/lybra/miner/EUSDMiningIncentives.sol	`9b8b29038cc934a7d3f977495c1ffb3c4e709e4f`
contracts/lybra/miner/ProtocolRewardsPool.sol	`2c96638c9c570c8252aa3a37050b46a72385922b`
contracts/lybra/miner/esLBRBoost.sol	`9709aef5aba508a0fc2278066ff257504ebe35eb`
contracts/lybra/miner/stakerewardV2pool.sol	`305609a52c0e0d4f85da80c9d4eb2e24a1cbe97c`
contracts/lybra/pools/LybraRETHVault.sol	`8e91a0cb71f408ebedac495456a7b4c217ecb9d3`
contracts/lybra/pools/LybraStETHVault.sol	`baf5b8f6eede7e46f49b3df45744b808631a1c4a`
contracts/lybra/pools/LybraWbETHVault.sol	`07d96ce9074bfc8366572f33fd597b031f10390b`
contracts/lybra/pools/LybraWstETHVault.sol	`f1b5181faf3ecdd517ff4bdb39782be5a8b43525`
contracts/lybra/pools/base/LybraEUSDVaultBase.sol	`359dc0953fd1938834143e9573a39acfdbc18652`
contracts/lybra/pools/base/LybraPeUSDVaultBase.sol	`94631f7b46b2788e08b91fbc5a046bd8c1aeac85`
contracts/lybra/token/EUSD.sol	`dfdf7a49d4b268cdb428cf8e90bbde075696e8e1`
contracts/lybra/token/LBR.sol	`2f39fb851d45abff1dbc1ea34ab6ed0091fe1c0a`
contracts/lybra/token/PeUSD.sol	`61c63b361a82ae5eb78b73dceee3fd07c3e65a66`

File	SHA-1 hash
contracts/lybra/token/PeUSDMainnet.sol	`1e600938b49f04ce0c43a4261af87cb71d7a34d6`
contracts/lybra/token/esLBR.sol	`d722f7df6c40b82f5a55bec6c17dec173ef60e1c`

Appendix 2 - Disclosure

Consensys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via Consensys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any third party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any third party by virtue of publishing these Reports.

A.2.1 Purpose of Reports

The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

A.2.2 Links to Other Web Sites from This Web Site

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Consensys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that Consensys and CD are not responsible for the content or operation of such Web sites, and that Consensys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that Consensys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. Consensys and CD assumes no responsibility for the use of third-party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

A.2.3 Timeliness of Content

The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice unless indicated otherwise, by Consensys and CD.