Lybra Finance - Vault - Guardian

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **December 18th, 2023 - December 22nd, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	12/18/2023
0.2	Document Updates	12/21/2023
0.3	Draft Review	12/22/2023
1.0	Remediation Plan	01/02/2024
1.1	Remediation Plan Review	01/02/2024

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Lybra Finance engaged Halborn to conduct a security assessment on their smart contracts beginning on December 18th, 2023 and ending on December 22nd, 2023. The security assessment was scoped to the smart contracts provided to the Halborn team.

1.2 ASSESSMENT SUMMARY

The team at Halborn was provided one week for the engagement and assigned a full-time security engineer to verify the security of the smart contracts. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified some security risks that were accepted and acknowledged by the Lybra Finance Team.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions. (solgraph)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions. (Slither)
- Testnet deployment. (Foundry)

2. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

2.1 EXPLOITABILITY

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Attack Complexity (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

Metrics:

Exploitability Metric (m_E)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 IMPACT

Confidentiality (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Integrity (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Availability (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Deposit (D):

Measures the impact to the deposits made to the contract by either users or owners.

Yield (Y):

Measures the impact to the yield generated by the contract for either users or owners.

Metrics:

Impact Metric (m_I)	Metric Value	Numerical Value
Confidentiality (C)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

2.3 SEVERITY COEFFICIENT

Reversibility (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Scope (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Severity	Score Value Range
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

2.4 SCOPE

1. IN-SCOPE TREE & COMMIT :

The security assessment was scoped to the following smart contracts:

Commit ID :

- 637cd8eae2aedbc6e873133c5a93d2995bf44421

In-scope Contracts :

- CollateralRatioGuardian.sol
- LybraETHxVault.sol

Out-of-scope

- Third-party libraries and dependencies.
- Economic attacks.

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	0	3

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) LATESTROUNDDATA CALL MAY RETURN STALE RESULTS	Medium (6.2)	RISK ACCEPTED
(HAL-02) OWNABLE USES SINGLE-STEP OWNERSHIP TRANSFER	Informational (0.0)	ACKNOWLEDGED
(HAL-03) RE-ENTRANCY RISK IN AUTO-REPAYMENT FUNCTION	Informational (0.0)	ACKNOWLEDGED
(HAL-04) UNSAFE HANDLING OF ERC20 TRANSFER RESULTS	Informational (0.0)	ACKNOWLEDGED

FINDINGS & TECH DETAILS

4.1 (HAL-01) LATESTROUNDDATA CALL MAY RETURN STALE RESULTS - MEDIUM (6.2)

Description:

The return values of the latestRoundData() call are:

- roundId: The round ID. Oracles provide periodic data updates to the aggregators. Data feeds are updated in rounds. Rounds are identified by their roundId, which increases with each new round. This increase may not be monotonic. Knowing the roundId of a previous round allows contracts to consume historical data.
- answer: The data that this specific feed provides, in this case, the price of an asset. It can return 0.
- startedAt: Timestamp of when the round started.
- updatedAt: Timestamp of when the round was updated.
- answeredInRound: The round ID of the round in which the answer was computed.

In the current implementation, There is no check for stale prices.

Code Location:

CollateralRatioGuardian.sol#L115C1-L118C6

Listing 1

```
    function getAssetPrice(address vault) public view returns (
↳ uint256) {
        (,int price, , , ) = priceFeed.latestRoundData();
        return ILybra(vault).getAsset2EtherExchangeRate() * uint(
↳ price) / 1e8;
    }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:M/Y:N/R:N/S:U (6.2)

Recommendation:

getAssetPrice calls out to a Chainlink oracle receiving the latestRoundData(). If there is a problem with Chainlink starting a new round and finding consensus on the new value for the oracle (e.g. Chainlink nodes abandon the oracle, chain congestion, vulnerability/attacks on the Chainlink system) consumers of this contract may continue using outdated stale or incorrect data (if oracles are unable to submit no new round is started).

Remediation Plan:

RISK ACCEPTED: The Lybra Finance team accepted the risk of this issue.

4.2 (HAL-02) OWNABLE USES SINGLE-STEP OWNERSHIP TRANSFER - INFORMATIONAL (0.0)

Description:

Single-step ownership transfer means that if a wrong address was passed when transferring ownership or admin rights, it can mean that role is lost forever. The ownership pattern implementation for the protocol is in Ownable.sol where a single-step transfer is implemented.This can be a problem for all methods marked in onlyOwner throughout the protocol, some of which are core protocol functionality.

Code Location:

CollateralRatioGuardian.sol#L11

Listing 2

```
contract CollateralRatioGuardian is Ownable {}
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

It is a best practice to use a two-step ownership transfer pattern, meaning the ownership transfer gets to a “pending” state and the new owner should claim his new rights; otherwise, the old owner still has control of the contract. Consider using OpenZeppelin’s Ownable2Step contract.

Remediation Plan:

ACKNOWLEDGED: The Lybra Finance team acknowledged this finding.

4.3 (HAL-03) RE-ENTRANCY RISK IN AUTO-REPAYMENT FUNCTION - INFORMATIONAL (0.0)

Description:

The execute function in the given smart contract is designed to enable automatic repayments for users under certain conditions. However, it poses a potential re-entrancy risk, particularly if the vault token has fallback functionalities like ERC777 or ERC677.

Re-entrancy is a vulnerability that occurs when external contract calls are made (in this case, token.transferFrom and lybraPool.burn) allowing the called contract to regain control and re-enter the calling contract before the initial function execution completes. If the vault token has ERC777 or ERC677 features, it can trigger hooks that might call back into the execute function, potentially leading to issues like double spending.

Code Location:

CollateralRatioGuardian.sol#L79

Listing 3

```

    /**
    * @dev Allows any third-party keeper to trigger automatic
    ↳ repayment for a user.
    * Requirements:
    * `user` must have enabled the automatic repayment feature.
    * Current collateral ratio of the user must be less than or
    ↳ equal to userSetting.triggerCollateralRatio.
    * `user` must have authorized this contract to spend eUSD in
    ↳ an amount greater than the repayment amount + fee.
    */
    function execute(address user, address vault) external {
        require(configurator.mintVault(vault), "NV");
        RepaymentSetting memory userSetting =
userRepaymentSettings[user][vault];
        require(userSetting.active == true, "The user has not enabled the automatic repayment");
        uint256 userCollateralRatio = getCollateralRatio(user, vault);
        require(userCollateralRatio <= userSetting.triggerCollateralRatio, "The user's collateralRate is not below the trigger collateralRate");

        ILybra lybraPool = ILybra(vault);
        uint256 targetDebt = (lybraPool.depositedAsset(user) * getAssetPrice(vault)) * 100 / userSetting.expectedCollateralRatio;
        uint256 repayAmount = lybraPool.getBorrowedOf(user) - targetDebt ;
        IERC20 token = lybraPool.getVaultType() == 0 ? IERC20(configurator.getEUSDAddress()) : IERC20(configurator.peUSD());
        token.transferFrom(user, address(this), repayAmount + fee);
        lybraPool.burn(user, repayAmount);
        uint256 balance = token.balanceOf(address(this)) < fee ? token.balanceOf(address(this)) : fee;
        token.transfer(msg.sender, balance);
        emit ExecuteAutoRepayment(user, vault, msg.sender, repayAmount, balance, block.timestamp);
    }
```

Proof Of Concept:

- A user enables automatic repayment and sets a triggerCollateralRatio .
- The execute function is called for this user.
- In the process, token.transferFrom is used to transfer tokens from the user to the contract. If the token is ERC777/ERC677 compliant, it can trigger a tokensToSend or similar hook.
- This hook can make a recursive call back to execute, leading to re-entrance before the initial execution of execute is completed.
- As a result, the same tokens might be used multiple times for repayment, depleting the user's funds unfairly or affecting the contract's state in unintended ways.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

Implement a reentrancy guard, such as OpenZeppelin’s ReentrancyGuard contract, to prevent nested (reentrant) calls.

Remediation Plan:

ACKNOWLEDGED: The Lybra Finance team acknowledged this finding.

4.4 (HAL-04) UNSAFE HANDLING OF ERC20 TRANSFER RESULTS - INFORMATIONAL (0.0)

Description:

It was identified that the execute function in the CollateralRatioGuardian contract does not verify the return value of the transferFrom function call, which facilitates the token transfer from the caller to the contract. Some tokens (e.g., ZRX) return false instead of reverting in the event of failure or insufficient balance.

Code Location:

CollateralRatioGuardian.sol#L79

Listing 4

```

    /**
    * @dev Allows any third-party keeper to trigger automatic
↳ repayment for a user.
    * Requirements:
    * `user` must have enabled the automatic repayment feature.
    * Current collateral ratio of the user must be less than or
↳ equal to userSetting.triggerCollateralRatio.
    * `user` must have authorized this contract to spend eUSD in
↳ an amount greater than the repayment amount + fee.
    */
    function execute(address user, address vault) external {
        require(configurator.mintVault(vault), "NV");
        RepaymentSetting memory userSetting =
↳ userRepaymentSettings[user][vault];
        require(userSetting.active == true, "The user has not
↳ enabled the automatic repayment");
        uint256 userCollateralRatio = getCollateralRatio(user,
↳ vault);
        require(userCollateralRatio <= userSetting.
↳ triggerCollateralRatio, "The user's collateralRate is not below
↳ the trigger collateralRate");

        ILybra lybraPool = ILybra(vault);
        uint256 targetDebt = (lybraPool.depositedAsset(user) *
↳ getAssetPrice(vault)) * 100 / userSetting.expectedCollateralRatio;
        uint256 repayAmount = lybraPool.getBorrowedOf(user) -
↳ targetDebt ;
        IERC20 token = lybraPool.getVaultType() == 0 ? IERC20(
↳ configurator.getEUSDAddress()) : IERC20(configurator.peUSD());
        token.transferFrom(user, address(this), repayAmount + fee)
↳ ;
        lybraPool.burn(user, repayAmount);
        uint256 balance = token.balanceOf(address(this)) < fee ?
↳ token.balanceOf(address(this)) : fee;
        token.transfer(msg.sender, balance);
        emit ExecuteAutoRepayment(user, vault, msg.sender,
↳ repayAmount, balance, block.timestamp);
    }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

It is recommended to use OpenZeppelin's SafeERC20 wrapper and the safeTransferFrom function to transfer the tokens.

Remediation Plan:

ACKNOWLEDGED: The Lybra Finance team acknowledged this finding.

AUTOMATED TESTING

5.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their ABIS and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

Slither results:

- No major issues found by Slither.

THANK YOU FOR CHOOSING

HALBORN