

LazyOtter Audit Report

Cymetrics

August 30, 2024



Contents

1	About Cymetrics	3
2	Introduction	3
3	Risk Classification	4
3.1	Impact	4
3.2	Likelihood	4
4	Security Assessment Summary	5
4.1	Project Summary	5
4.2	Scope	5
5	Findings	7
5.1	Medium Risk Findings	7
5.1.1	The calculation error in the simpleInterestFactor has harmed depositors' earnings and poses a potential systemic risk	7

List of Figures

1 testPoCMaxDepositModified output 9

List of Tables

1 About Cymetrics

Cymetrics is the cybersecurity arm of OneDegree Global, incorporated in Singapore with a strong presence in the APAC & Middle East regions. From on-demand cybersecurity assessments to Red Team services, Cymetrics helps secure your enterprise cyber defense with proprietary SaaS-based technology and market-leading intelligence.

2 Introduction

LazyOtter is a risk intelligence platform designed to enhance security in DeFi. DeFi holds the potential to surpass traditional finance in safety due to blockchain technology. However, the complexity and lack of transparency in DeFi leave room for malicious actors. LazyOtter aims to provide a safer investment alternative by identifying and mitigating real risks, enabling users to invest confidently.

Disclaimer: This review does not guarantee against a hack. It is a snapshot in a time of commit (commit hash) according to the specific commit. Any modifications to the code will require a new security review.

3 Risk Classification

Severity Level	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	High	Medium	Low
Likelihood: Medium	Medium	Low	Informational
Likelihood: Low	Low	Informational	Informational

3.1 Impact

- High: leads to a loss of assets in the protocol, or significant harm to a majority of users.
- Medium: function or availability of the protocol could be impacted or losses to only a subset of users.
- Low: State handling, function incorrect as to spec, issues with clarity, losses will be annoying but bearable.

3.2 Likelihood

- High: almost certain to happen, easy to perform, or not easy but highly incentivized.
- Medium: only conditionally possible or incentivized, but still relatively likely.
- Low: requires stars to align, or little-to-no incentive.

4 Security Assessment Summary

The audit work was conducted in the time frame August 14th, 2024 to August 15th, 2024. One security engineer from [Cymetrics](#) and two white hats from [DeFiHackLabs](#) participated in this audit.

The white hats are:

- [@icebear](#)
- [@ret2basic](#)

4.1 Project Summary

4.2 Scope

Summary

Project Name	LazyOtter
Repository	https://github.com/lazyotter-finance/lazyotter-contract/blob/develop/src/vaults/RhoMarketsVault.sol
Commit hash	73a84e99c923486f8f0d90af7b79538a8b27b1be
File in Scope	RhoMarketsVault.sol

Issues Found

Severity	Count	Fixed	Acknowledged
High	0	0	0
Medium	1	1	0
Low	0	0	0
Informational	0	0	0
Total	1	1	0

5 Findings

5.1 Medium Risk Findings

5.1.1 The calculation error in the `simpleInterestFactor` has harmed depositors' earnings and poses a potential systemic risk

Description

In `maxDeposit()`, the `simpleInterestFactor` used to calculate interest accumulation employs inconsistent units to calculate the difference between blocks.

The `block.timestamp` and `RErc20.accrualBlockNumber()` have different units, and directly subtracting them leads to calculation errors.

As a result, the `simpleInterestFactor` is significantly underestimated, harming depositors' earnings. Parameters that rely on `simpleInterestFactor`, such as `interestAccumulated`, `totalBorrows`, and `totalReserves`, are also subject to systemic risk due to the incorrect calculations.

Code Snippet

- [RhoMarketsVault.sol#L67](#)

PoC

In contract `RhoMarketsVault.sol`, add a new function named `maxDepositModified`, whose content is copied from `maxDeposit` and modify the `block.timestamp` in [line 67](#) to `block.number`:


```
1      function maxDepositModified(address) public view
2          returns (uint256) {
3              // Supply cap of 0 corresponds to unlimited
4              // supplying
5              uint256 supplyCap = comptroller.supplyCaps(address
6                  (RErc20));
7              if (supplyCap == 0) {
8                  return type(uint256).max;
9              }
10
11              uint256 totalCash = RErc20.getCash();
12              uint256 totalBorrows = RErc20.totalBorrows();
13              uint256 totalReserves = RErc20.totalReserves();
14
15              uint256 borrowRate = interestRateModel.
16                  getBorrowRate(totalCash, totalBorrows,
17                      totalReserves);
18
19              uint256 simpleInterestFactor = borrowRate * (block
20                  .number - RErc20.accrualBlockNumber());
21              uint256 interestAccumulated = (
22                  simpleInterestFactor * totalBorrows) / 1e18;
23
24              totalBorrows = interestAccumulated + totalBorrows;
25              totalReserves = (interestAccumulated * RErc20.
26                  reserveFactorMantissa()) / 1e18 + totalReserves
27                  ;
28
29              uint256 totalSupplies = totalCash + totalBorrows -
30                  totalReserves;
31
32              if (supplyCap > totalSupplies) {
33                  return supplyCap - totalSupplies - 1;
34              }
35
36              return 0;
37          }
```

In the test file `RhoMarketsVault.t.sol`, add a new test case `testPoCMaxDepositModified`:

```
1 function testPoCMaxDepositModified() public {
2     uint256 totalAmount = 100 * 1e6;
3
4     deal(address(USDC), address(this), totalAmount);
5     USDC.approve(address(vault), totalAmount);
6     vault.deposit(totalAmount, address(this));
7
8     uint256 maxDeposit = vault.maxDeposit(address(this));
9     console.log("maxDeposit: ", maxDeposit);
10
11     uint256 maxDepositModified = vault.
12         maxDepositModified(address(this));
13     console.log("maxDepositModified: ",
14         maxDepositModified);
15
16     console.log("Difference = ", maxDepositModified -
17         maxDeposit);
18 }
```

Run the test case:

```
1 forge test --match-contract RhoMarketsVaultTest --match-
   test testPoCMaxDepositModified -vv
```

Output:

```
Ran 1 test for test/RhoMarketsVault.t.sol:RhoMarketsVaultTest
[PASS] testPoCMaxDepositModified() (gas: 807809)
Logs:
  maxDeposit: 1818171499547
  maxDepositModified: 89431808427932
  Difference = 87613636928385

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 458.36ms (3.67ms C
PU time)

Ran 1 test suite in 463.99ms (458.36ms CPU time): 1 tests passed, 0 failed, 0 s
kipped (1 total tests)
```

Figure 1: testPoCMaxDepositModified output

Here we can clearly see that `block.timestamp` and `block.number`

produce different outputs, in particular, the current implementation (using `block.timestamp`) returns a smaller value.

Numerically, the results in this specific scenario looks pretty bad: `maxDepositModified` is actually 48 times larger than `maxDeposit`!

```
1 >>> 89431808427932 // 1818171499547
2 49
```

In other words, the “potential” of `maxDeposit` is heavily limited in current implementation, which can cause problem in integration phase. Other devs might build their own contracts and use `RhoMarketsVault.sol` as a moving part, but this surprisingly small `maxDeposit` output could lead contracts to unknown states.

Recommendation

Change the `block.timestamp` in [this line](#) to `block.number`:

```
1 uint256 simpleInterestFactor = borrowRate * (block.number
  - RErc20.accrualBlockNumber());
```

Status

Cymetrics: Fixed. Commit hash: cc658bdd859014d0162b907ec77aab1a8bd4a711