Cymetrics

LazyOtter Audit Report

Cymetrics

September 3, 2024

Contents

1 About Cymetrics **4**

2 Introduction **4**

3 Risk Classification **5**
3.1 Impact 5
3.2 Likelihood 5

4 Security Assessment Summary **6**
4.1 Project Summary 6
4.2 Scope 6

5 Findings **8**
5.0.1 Private key leak 8
5.0.2 Unchecked status code in `RhoMarketsVault._deposit_()` and `RhoMarketsVault._withdraw_()` 10
5.0.3 Unable To Withdraw All Funds After `Emergency-Withdraw` For `RhoMarketsVault` When The Market Cap Is Low 15
5.1 Medium Risk Findings 19
5.1.1 `RhoMarketsVault` Doesn't Implement `maxMint` And `maxRedeem`, Making It Incompatible With `EIP4626` 19
5.1.2 The calculation error in the simpleInterestFactor has harmed depositors' earnings and poses a potential systemic risk. 25
5.1.3 `maxMint` and `maxDeposit` Should Always Return `0` When `paused` 29
5.1.4 `Vault.execute()`: Unchecked return value from low-level `call()` 31

5.1.5 Third-Party Dependencies Could Cause Unintended Issues . . . 32
5.2 Low Risk Findings . . . 34
5.2.1 The `RhoMarketsVault::maxDeposit` Constraint Is Not Strictly Followed . . . 34
5.2.2 `vault::execute` Is Over-Designed To Retrieve `Ether` from the `vault` . . . 40
5.3 Info Risk Findings . . . 43
5.3.1 Redundant `roleAdmin` Assignment . . . 43
5.3.2 Foundry Console Import Should Be Removed . . . 46
5.3.3 Typo Error in Storage Location Calculaiton . . . 47

6 Appendix: Technical doc 49
6.1 Centralization risk . . . 49
6.2 Scroll vs. Ethereum differences . . . 49
6.3 Classic vault attacks . . . 50
6.3.1 Inflation attack . . . 50
6.3.2 Vault reset attack . . . 50
6.3.3 Rounding directions . . . 50
6.3.4 Slippage . . . 50
6.3.5 Reentrancy . . . 51
6.4 Vault functionalities analysis . . . 51
6.4.1 Emergency withdrawal . . . 51
6.4.2 Types of vaults . . . 51
6.5 External Protocol Integration . . . 51
6.6 Other comments . . . 52

7 Appendix: 4naly3er Report 52

List of Figures

1 mint 11
2 image 12
3 testPoCMaxDepositModified output 27

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 28th, 2024 to September 1st, 2024. One security engineer from Cymetrics and two white hats from DeFiHackLabs(TaiChi) participated in this audit.

The white hats are:

- @icebear
- @ret2basic
- @jesjupyter

4.1 Project Summary

4.2 Scope

Summary

Project Name	LazyOtter
Repository	https://github.com/lazyotter-finance/lazyotter-contract/tree/develop/src/vaultsUpgradable
Commit hash	ca1ca1ff8e56fdd29d7defdcd957a97bf0dab521
File in Scope	every contract under vaultsUpgradable/

Issues Found

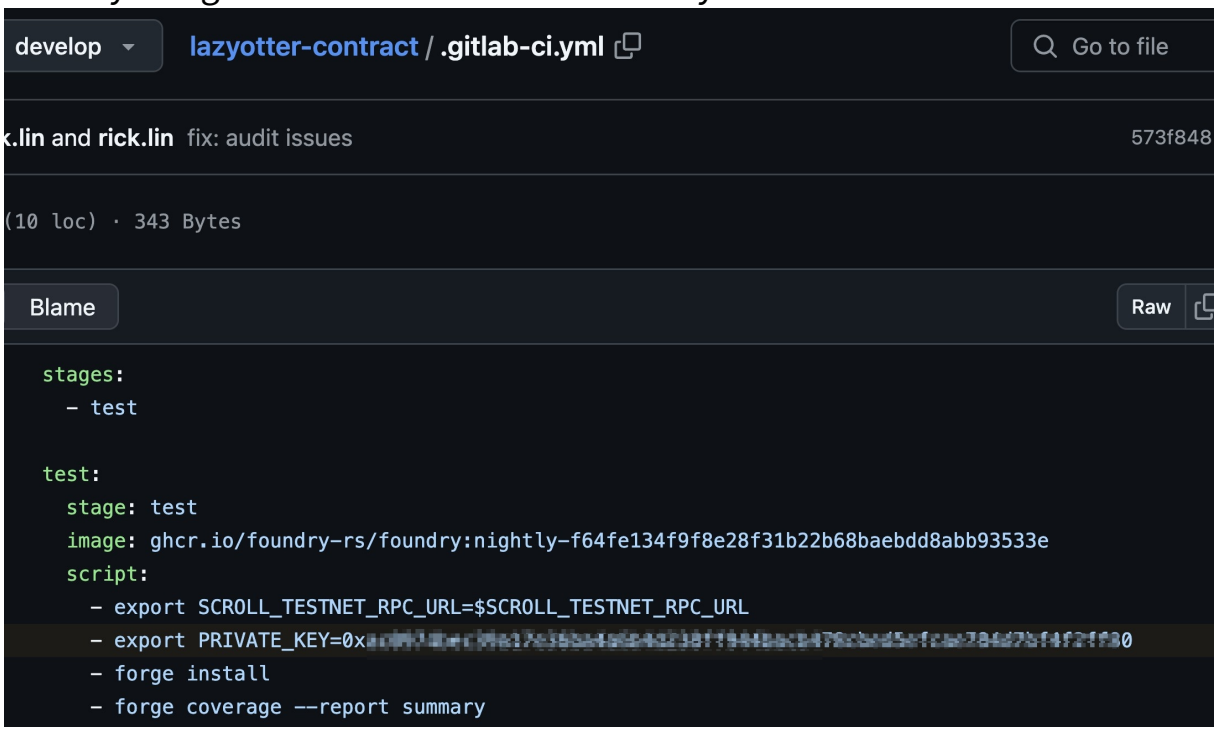
Severity	Count	Fixed	Acknowledged
High	3	3	0
Medium	5	4	1
Low	2	0	2
Informational	3	3	0
Total	13	10	3

5 Findings

5.0.1 Private key leak

Description

`.gitlab-ci.yml` exposes private key in plaintext. We can't verify if it's a real private key containing assets or it's just a key for testing (for auditor integrity concerns). Exposing private key on Github is extremely dangerous because there is no way to delete records from Github.



Also even if this key does not contain any asset, it will be used for deploying LazyOtter contracts. Attacker can claim this private key and become owner of all contracts.

Recommendation

If this is a real private key containing assets, delete the entire Github repo immediately. If it is a key for testing, remove the file and abondon the key, regenerate another one. Do not use the leaked private key for any purpose.

Status

Cymetrics: Fixed.Commit hash:c9e7faa5f8a93c58821a38fb408d5f5010d329a2

5.0.2 Unchecked status code in `RhoMarketsVault._deposit_()` and `RhoMarketsVault._withdraw_()`

Description

When user interacts with RhoMarketsVault, he calls `deposit()` with sufficiently approval to the vault. The vault pulls asset (USDC in test cases) from he and mints him corresponding amount of vault shares (done in `_deposit()`). Then the control flow goes into `_deposit_()`, where the vault mints RErc20 (Rho Markets LP token, doc is here):

```
1          uint256 currentAssets = asset.balanceOf(address(
              this));
2          if (currentAssets > 0) {
3              asset.safeIncreaseAllowance(address(RErc20),
                  currentAssets);
4              RErc20.mint(currentAssets);
5          }
```

The issue is that `RErc20.mint()` has return value but it is unchecked in current implementation. Take `ScrollMainnet.RHO_MARKETS_USDC` for example, it is deployed at 0xAE1846110F72f2DaaBC75B7cEEe96558289EDfc5 on Scroll mainnet. The arguments and return values of `mint()` function can be checked at writeProxyContract:

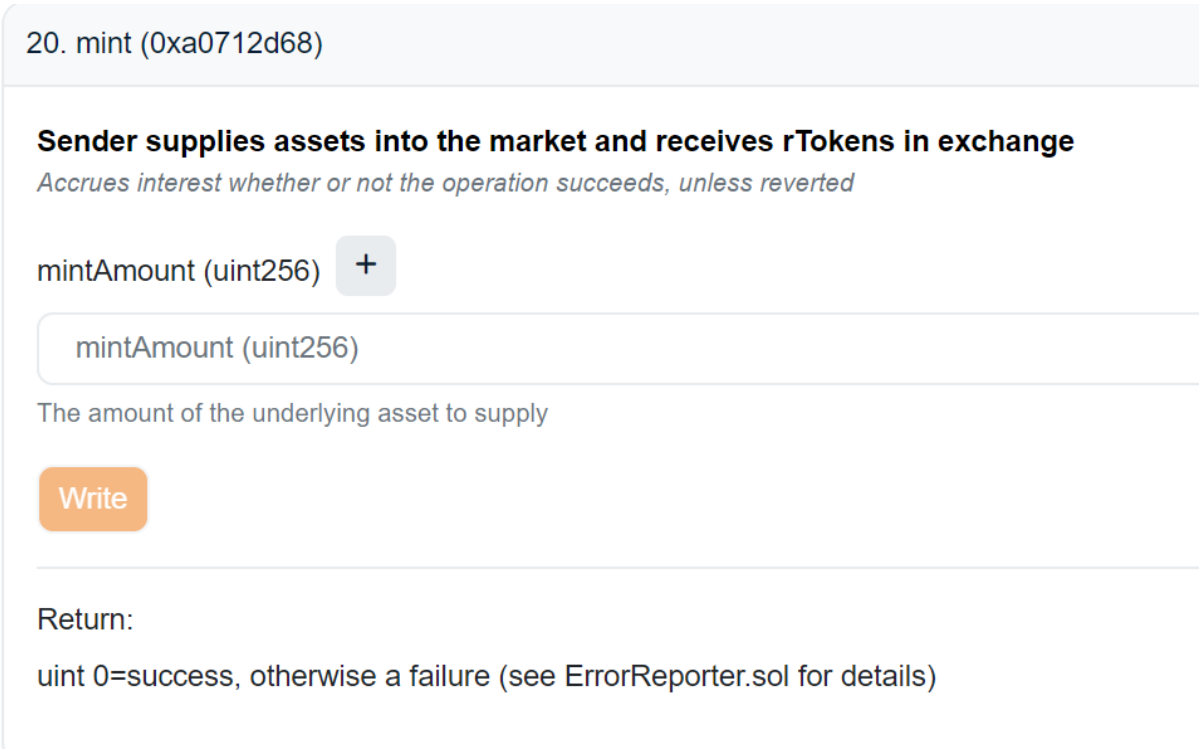


Figure 1: mint

Or we can go into the source code: We see `mint()` calls `mintFresh()`, which indeed has return value containing status code:

```
function mintFresh(address minter, uint256 mintAmount)
    internal returns (uint256, uint256) {
     ...
     return (uint256(Error.NO_ERROR), vars.actualMintAmount
        );
}
```

To make sure mint went through successfully in Rho Markets, the code must check if `mint()` returns 0, which represents success:

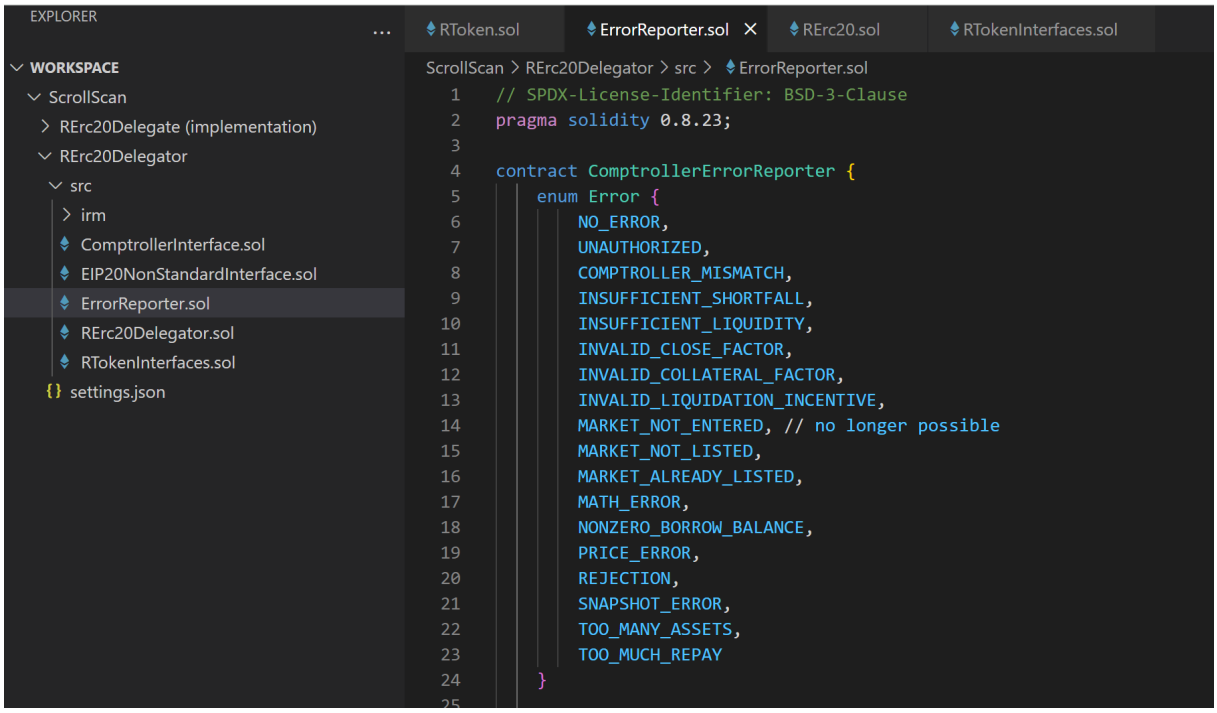


Figure 2: image

`RhoMarketsVault._withdraw_()` has the same bug for exactly the same reason: `RErc20.redeemUnderlying` returns a status code but it is unchecked in current implementation.

Code Snippet

- RhoMarketsVault.sol#L137
- RhoMarketsVault.sol#L152

Recommendation

Change the code to:

```
    /// @notice Handles the deposit operation
    /// _ The address of the depositor (unused in this implementation)
    /// _ The amount of assets to deposit (unused in this implementation)
    function _deposit_(address, uint256) internal override {
        RhoMarketsVaultStorage storage $ = _getRhoMarketsVaultStorage();
        IRErc20Delegator RErc20 = $.RErc20;
        IERC20 asset = IERC20(asset());

        uint256 currentAssets = asset.balanceOf(address(this));
        if (currentAssets > 0) {
            asset.safeIncreaseAllowance(address(RErc20), currentAssets);
-           RErc20.mint(currentAssets);
+           uint256 err = RErc20.mint(currentAssets);
+           require(err == 0, "RERc20.mint failed");
        }
    }

    /// @notice Handles the withdrawal operation
    /// _ The address of the withdrawer (unused in this implementation)
    /// @param assets The amount of assets to withdraw
    function _withdraw_(address, uint256 assets) internal override {
        RhoMarketsVaultStorage storage $ = _getRhoMarketsVaultStorage();
        IRErc20Delegator RErc20 = $.RErc20;
        IERC20 asset = IERC20(asset());

        uint256 currentAssets = asset.balanceOf(address(this));
        if (assets > currentAssets) {
            uint256 shortAssets = assets - currentAssets;
-           RErc20.redeemUnderlying(shortAssets);
+           uint256 err = RErc20.redeemUnderlying(shortAssets);
+           require(err == 0, "RErc20.redeemUnderlying failed");
        }
    }
```

Status

Cymetrics:Fixed.commit hash:b44bc859aad4eed569f426339f62ca468264c841

5.0.3 Unable To Withdraw All Funds After `EmergencyWithdraw` For `RhoMarketsVault` When The Market Cap Is Low

Description

In the current implementation of `RhoMarketsVault::maxWithdraw`, the function returns the minimum value between the owner's assets converted back from shares and the `cash` available in the Rho Market:

```
1     function maxWithdraw(address owner) public view
         override returns (uint256) {
2         RhoMarketsVaultStorage storage $ =
             _getRhoMarketsVaultStorage();
3         return Math.min(convertToAssets(balanceOf(owner)),
              $.RErc20.getCash());
4     }
```

- `convertToAssets(balanceOf(owner))`: Represents the amount of assets the owner should have in the vault.
- `$.RErc20.getCash()`: Represents the cash amount available in the `Rho Market`.

This approach generally works, but there is a critical edge case: 1. The `RErc20` market has a very low cap or is almost empty. 2. The vault performs an `emergencyWithdraw`, and no further deposits are made.

In this scenario, `convertToAssets(balanceOf(owner))` could be greater than `$.RErc20.getCash()`, causing `maxWithdraw` to return `$.RErc20.getCash()`, which is lower than the owner's actual assets. Consequently, if the owner attempts to withdraw more than the available cash, the transaction will be reverted due to the `maxWithdraw` restriction:

```
1         uint256 maxAssets = maxDeposit(receiver);
2         if (assets > maxAssets) {
3 @=>         revert ERC4626ExceededMaxDeposit(receiver,
     assets, maxAssets);
4         }
```

Even though the user can call `withdraw` multiple times, if the `$.RErc20.getCash()` is significantly smaller than `convertToAssets(balanceOf(owner))`, the excess funds could be locked indefinitely until other users deposit enough into the market to cover the shortfall.

Code Snippet

- RhoMarketsVault.sol#L104-L107

PoC

Here's a Proof of Concept (PoC) demonstrating that the user cannot fully redeem their assets immediately after an `emergencyWithdraw`:

```
    // Below is a PoC that the user can not fully redeem
       their asset temporarily after emergencyWithdraw
    function testMaxRedeemFailure() public {
        uint256 amount = vault.maxDeposit(address(this));
        deal(address(USDC), address(this), amount);
        USDC.approve(address(vault), amount);
        vault.deposit(amount, address(this));
        vault.emergencyWithdraw();

        console.log("We try to mimic a situation where
           after emergencyWithdraw, there is little cash
           in the markets");
        console.log("convertToAssets(vault.balanceOf(
           address(this))", vault.convertToAssets(vault.
           balanceOf(address(this))));
        console.log("totalCash", RUSDC.getCash());

        console.log("convertToAssets(vault.balanceOf(
           address(this)) > totalCash", vault.
           convertToAssets(vault.balanceOf(address(this)))
            > RUSDC.getCash());

        uint256 maxWithdraw = vault.maxWithdraw(address(
           this));
        console.log("maxWithdraw = totalCash", maxWithdraw
           );

        console.log("withdraw maxWithdraw + 1 Will Fail",
           maxWithdraw+1);
        vm.expectRevert();
        vault.withdraw(maxWithdraw+1, address(this),
           address(this));
    }
```

The output:

```
Ran 1 test for test/vaultsUpgradable/v1/RhoMarketsVault.t.sol:RhoMarketsVaultTest
[PASS] testMaxRedeemFailure() (gas: 1725662)
Logs:
  We try to mimic a situation where after emergencyWithdraw, there is little cash in the markets
  convertToAssets(vault.balanceOf(address(this)) 1768000275976
  totalCash 224906214405
  convertToAssets(vault.balanceOf(address(this)) > totalCash true
  maxWithdraw = totalCash 224906214405
  withdraw maxWithdraw + 1 Will Fail 224906214406
```

Recommendation

To mitigate this issue, it is recommended to include `asset.balanceOf(address(this))` in the `maxWithdraw` calculation. This will ensure that the vault accounts for the assets held in the vault itself, particularly after an `emergencyWithdraw`.

Status

Cymetrics:Fixed.Commit hash:b44bc859aad4eed569f426339f62ca468264c841

5.1 Medium Risk Findings

5.1.1 RhoMarketsVault Doesn't Implement maxMint And maxRedeem, Making It Incompatible With EIP4626

Description

The RhoMarketsVault contract, which inherits from Vault, further inherits from ERC4626Upgradeable.

```
@=> contract RhoMarketsVault is Vault { ... }

contract Vault is
    Initializable,
@=> ERC4626Upgradeable,
    AccessControlUpgradeable,
    PausableUpgradeable,
    ReentrancyGuardUpgradeable
{
    ...
}
```

While maxDeposit and maxWithdraw have been overridden in RhoMarketsVault to impose restrictions on the amount of asset that can be deposited and withdrawn, maxMint and maxRedeem have not been similarly overridden.

```
function maxDeposit(address) public view override returns
   (uint256) { ... }
function maxWithdraw(address owner) public view override
   returns (uint256) { ... }
```

As a result, these functions retain their default implementations from ERC4626Upgradeable, which may not align with the specific constraints of the RhoMarketsVault contract.

The maxMint function currently returns type(uint256).max, implying that there is no limit to the number of shares that can be minted. Similarly,

`maxRedeem` returns the balance of the owner, which does not account for the specific business logic of the `RhoMarketsVault` contract.

```
1      /** @dev See {IERC4626-maxMint}. */
2      function maxMint(address) public view virtual returns
          (uint256) {
3          return type(uint256).max;
4      }
5      /** @dev See {IERC4626-maxRedeem}. */
6      function maxRedeem(address owner) public view virtual
          returns (uint256) {
7          return balanceOf(owner);
8      }
```

According to EIP-4626, `maxMint` MUST return the maximum amount of shares that can be minted without causing a revert, and this amount MUST NOT exceed the actual maximum that would be accepted (underestimating if necessary). The same rule applies to `maxRedeem`.

For instance, in `maxDeposit`, it is known that deposits cannot be made if `comptroller.supplyCaps(address(RErc20))<= totalSupplies`, but this logic is not reflected in `maxMint`. As a result, users may mistakenly assume that they can mint an unlimited number of shares, leading to potential reverts or errors.

```
    function maxDeposit(address) public view override
       returns (uint256) {
        RhoMarketsVaultStorage storage $ =
           _getRhoMarketsVaultStorage();
        IComptroller comptroller = $.comptroller;
        IRErc20Delegator RErc20 = $.RErc20;
        IInterestRateModel interestRateModel = $.
           interestRateModel;

        // Supply cap of 0 corresponds to unlimited
           supplying
        uint256 supplyCap = comptroller.supplyCaps(address
           (RErc20));
        if (supplyCap == 0) {
            return type(uint256).max;
        }

        uint256 totalCash = RErc20.getCash();
        uint256 totalBorrows = RErc20.totalBorrows();
        uint256 totalReserves = RErc20.totalReserves();

        uint256 borrowRate = interestRateModel.
           getBorrowRate(totalCash, totalBorrows,
           totalReserves);

        uint256 simpleInterestFactor = borrowRate * (block
           .timestamp - RErc20.accrualBlockNumber());
        uint256 interestAccumulated = (
           simpleInterestFactor * totalBorrows) / 1e18;

        totalBorrows = interestAccumulated + totalBorrows;
        totalReserves = (interestAccumulated * RErc20.
           reserveFactorMantissa()) / 1e18 + totalReserves
           ;

        uint256 totalSupplies = totalCash + totalBorrows -
            totalReserves;

@=>     if (supplyCap > totalSupplies) {
            return supplyCap - totalSupplies - 1;
        }

@=>     return 0;
    }

    /** @dev See {IERC4626-maxMint}. */
    function maxMint(address) public view virtual returns
       (uint256) {
        return type(uint256).max;
    }
```

Failure to implement `maxMint` and `maxRedeem` in alignment with the business logic of `RhoMarketsVault` leads to incompatibility with the `EIP-4626` standard. This can introduce potential integration issues, where external systems interacting with `RhoMarketsVault` under the assumption that it fully conforms to EIP-4626 may encounter unexpected behaviors, including reverts or incorrect operations.

Code Snippet

```
    function maxDeposit(address) public view override
       returns (uint256) {
        RhoMarketsVaultStorage storage $ =
           _getRhoMarketsVaultStorage();
        IComptroller comptroller = $.comptroller;
        IRErc20Delegator RErc20 = $.RErc20;
        IInterestRateModel interestRateModel = $.
           interestRateModel;

        // Supply cap of 0 corresponds to unlimited
           supplying
        uint256 supplyCap = comptroller.supplyCaps(address
           (RErc20));
        if (supplyCap == 0) {
            return type(uint256).max;
        }

        uint256 totalCash = RErc20.getCash();
        uint256 totalBorrows = RErc20.totalBorrows();
        uint256 totalReserves = RErc20.totalReserves();

        uint256 borrowRate = interestRateModel.
           getBorrowRate(totalCash, totalBorrows,
           totalReserves);

        uint256 simpleInterestFactor = borrowRate * (block
           .timestamp - RErc20.accrualBlockNumber());
        uint256 interestAccumulated = (
           simpleInterestFactor * totalBorrows) / 1e18;

        totalBorrows = interestAccumulated + totalBorrows;
        totalReserves = (interestAccumulated * RErc20.
           reserveFactorMantissa()) / 1e18 + totalReserves
           ;

        uint256 totalSupplies = totalCash + totalBorrows -
            totalReserves;

@=>     if (supplyCap > totalSupplies) {
            return supplyCap - totalSupplies - 1;
        }

@=>     return 0;
    }

    /** @dev See {IERC4626-maxMint}. */
    function maxMint(address) public view virtual returns
       (uint256) {
        return type(uint256).max;
    }
```

Recommendation

To address this issue, `maxMint` and `maxRedeem` should be overridden to reflect the actual constraints and business logic of the `RhoMarketsVault` contract, ensuring full compatibility with EIP-4626. This can be achieved by implementing logic similar to that used in `maxDeposit` and `maxWithdraw` to accurately represent the maximum mintable and redeemable amounts.

Status

Cymetrics:Fixed.Commit hash:b44bc859aad4eed569f426339f62ca468264c841

5.1.2 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#L86

PoC

In contract `RhoMarketsVault.sol`, add a new function named `maxDepositModified`, whose content is copied from `maxDeposit` and modify the `block.timestamp` in line 86 to `block.number`:

```
function maxDepositModified(address) public view
   returns (uint256) {
    // Supply cap of 0 corresponds to unlimited
       supplying
    uint256 supplyCap = comptroller.supplyCaps(address
       (RErc20));
    if (supplyCap == 0) {
        return type(uint256).max;
    }

    uint256 totalCash = RErc20.getCash();
    uint256 totalBorrows = RErc20.totalBorrows();
    uint256 totalReserves = RErc20.totalReserves();

    uint256 borrowRate = interestRateModel.
       getBorrowRate(totalCash, totalBorrows,
       totalReserves);

    uint256 simpleInterestFactor = borrowRate * (block
       .number - RErc20.accrualBlockNumber());
    uint256 interestAccumulated = (
       simpleInterestFactor * totalBorrows) / 1e18;

    totalBorrows = interestAccumulated + totalBorrows;
    totalReserves = (interestAccumulated * RErc20.
       reserveFactorMantissa()) / 1e18 + totalReserves
       ;

    uint256 totalSupplies = totalCash + totalBorrows -
        totalReserves;

    if (supplyCap > totalSupplies) {
        return supplyCap - totalSupplies - 1;
    }

    return 0;
}
```

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

```
    function testPoCMaxDepositModified() public {
        uint256 totalAmount = 100 * 1e6;

        deal(address(USDC), address(this), totalAmount);
        USDC.approve(address(vault), totalAmount);
        vault.deposit(totalAmount, address(this));

        uint256 maxDeposit = vault.maxDeposit(address(this
            ));
        console.log("maxDeposit: ", maxDeposit);

        uint256 maxDepositModified = vault.
            maxDepositModified(address(this));
        console.log("maxDepositModified: ",
            maxDepositModified);

        console.log("Difference = ", maxDepositModified -
            maxDeposit);
    }
```

Run the test case:

```
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 3: 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 resutls in this specific scenario looks pretty bad: `maxDepositModified` is acutally 48 times larger than `maxDeposit`!

```
>>> 89431808427932 // 1818171499547
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`:

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

Status

Cymetrics:Fixe.Commit hash:b44bc859aad4eed569f426339f62ca468264c841

5.1.3 `maxMint` and `maxDeposit` Should Always Return `0` When `paused`

Description

According to EIP4626, the `maxDeposit` function MUST return the maximum amount of assets that can be deposited without causing a revert. The same rule applies to the `maxMint` function. These functions are crucial for ensuring that users can determine the limits of their actions before executing them.

In the current implementation, the vault forbids `deposit` and `mint` operations when the contract is paused, as enforced by the `whenNotPaused` modifier:

```
1     function deposit(uint256 assets, address receiver)
         public override nonReentrant whenNotPaused returns
         (uint256) {
2         uint256 shares = super.deposit(assets, receiver);
3         return shares;
4     }
5
6     function mint(uint256 shares, address receiver) public
          override nonReentrant whenNotPaused returns (
         uint256) {
7         uint256 assets = super.mint(shares, receiver);
8         return assets;
9     }
```

However, the `paused` state is not considered in the `maxMint` and `maxDeposit` functions. This could lead to situations where these functions return non-zero values even when deposits and mints would revert due to the paused state, making the protocol incompatible with `EIP-4626`.

Code Snippet

Deposit And Mint

```
    function deposit(uint256 assets, address receiver)
       public override nonReentrant whenNotPaused returns
       (uint256) {
        uint256 shares = super.deposit(assets, receiver);
        return shares;
    }

    function mint(uint256 shares, address receiver) public
        override nonReentrant whenNotPaused returns (
       uint256) {
        uint256 assets = super.mint(shares, receiver);
        return assets;
    }
```

Recommendation

To ensure compliance with `EIP-4626`, the `maxMint` and `maxDeposit` functions should be overridden to consider the `paused` state. Specifically, when the contract is paused, both `maxMint` and `maxDeposit` should return 0. This change will prevent any confusion or errors when users query these functions during a paused state.

Status

Cymetrics:Fixed.Commit hash:b44bc859aad4eed569f426339f62ca468264c841

5.1.4 `Vault.execute()`: Unchecked return value from low-level `call()`

Description

Unchecked return value of low-level `call()`/`delegatecall()` The call/delegatecall function returns a boolean value indicating whether the call was successful. However, it is important to note that this return value is not being checked in the current implementation.

As a result, there is a possibility that the call wasn't successful, while the transaction continues without reverting.

Code Snippet

- Vault.sol#L148

Recommendation

Update the code to:

```
    function execute(
        address _to,
        uint256 _value,
        bytes calldata _data
    ) external onlyOwner returns (bool, bytes memory) {
        (bool success, bytes memory result) = _to.call{
           value: _value}(_data);
        require(success, "execute() failed")
        return (success, result);
    }
```

Status

Cymetrics:Fixed.Commit hash:b44bc859aad4eed569f426339f62ca468264c841

5.1.5 Third-Party Dependencies Could Cause Unintended Issues

Description

The vault currently relies on third-party protocols such as AAVE and Rho Market. However, relying on third-party dependencies can lead to unintended consequences if changes are made upstream.

For example, in the RhoMarketsVault contract, the IRErc20Delegator from Rho Market is used. By examining the interface of IRErc20Delegator, it is evident that functions such as _setComptroller and _setInterestRateMod can be called to modify critical parameters.

Currently, in the RhoMarketsVault implementation, these values are fixed once they are set during the initialization:

```
1      function initialize(
2          IERC20 asset_,
3          string memory name_,
4          string memory symbol_,
5          address keeper_,
6          IRErc20Delegator RErc20_
7      ) public initializer {
8          super.initialize(asset_, name_, symbol_, keeper_);
9
10         RhoMarketsVaultStorage storage $ =
               _getRhoMarketsVaultStorage();
11         $.RErc20 = RErc20_;
12 @=>    $.comptroller = IComptroller(RErc20_.comptroller()
       );
13 @=>    $.interestRateModel = IInterestRateModel(RErc20_.
       interestRateModel());
14     }
```

If the comptroller or interestRateModel is changed by the upstream protocol (although rare, it is still possible), the vault contract may still refer to the outdated versions, leading to incorrect calculations, such as an incorrect maxDeposit amount, thereby affecting the normal deposit operations.

Code Snippet

- RhoMarketsVault.sol#L50-L63

Recommendation

To mitigate potential issues caused by third-party dependencies, it is recommended to **Fully Understand Dependencies** and **Do Regularly Monitor and Updates**.

Status

Cymetrics:Acknowledged.

5.2 Low Risk Findings

5.2.1 The `RhoMarketsVault::maxDeposit` Constraint Is Not Strictly Followed

Description

The `RhoMarketsVault::maxDeposit` function calculates the maximum amount that can be deposited into the vault by strictly checking against the `supplyCap` from the `comptroller`. However, there's a potential issue where the actual amount being deposited into the protocol might not match the intended amount passed in due to the way the deposit process is handled.

The `RhoMarketsVault::maxDeposit` has been overriden to calculate the maximum amount that can be deposited. It strictly checks the `supplyCap` restriction from the `comptroller`.

```
1      function maxDeposit(address) public view override
          returns (uint256) {
2           ...
3  @=>      return supplyCap - totalSupplies - 1;
4           ...
5      }
```

However, the deposit function does not directly use the amount passed to it. Instead, it deposits whatever amount of the asset is currently held by the vault.

```
function _deposit_(address, uint256) internal override
    { // <= The amount is never used here
    RhoMarketsVaultStorage storage $ =
       _getRhoMarketsVaultStorage();
    IRErc20Delegator RErc20 = $.RErc20;
    IERC20 asset = IERC20(asset());

    uint256 currentAssets = asset.balanceOf(address(
       this)); // <= The actual amount being deposited
        is asset.balanceOf(address(this))
    if (currentAssets > 0) {
        asset.safeIncreaseAllowance(address(RErc20),
           currentAssets);
        RErc20.mint(currentAssets);
    }
}
```

In normal situations, we assume the contract should not hold any excessive `asset`(thus `asset.balanceOf(address(this))` would only be the amount transferred in during the deposit), but this would not be the case for the following scenario

- If an `emergencyWithdraw` is called, tokens could be withdrawn from the protocol back to the vault. This could leave the vault with an excess balance of the asset.
- In this scenario, the `maxDeposit` function might return a value that assumes no excess assets are in the vault. However, when the actual deposit happens, the vault' s balance could be higher, resulting in an unexpected deposit that might exceed the intended supplyCap.
- This discrepancy could lead to a denial of service (DoS) if the excess balance causes the vault to attempt to deposit more than the `maxDeposit` amount, violating the EIP4626 standard that `maxDeposit MUST return the maximum amount of assets deposit would allow to be deposited for receiver and not cause a revert`.

Code Snippet

RhoMarketsVault::*deposit*

```
function _deposit_(address, uint256) internal override
    {
    RhoMarketsVaultStorage storage $ =
       _getRhoMarketsVaultStorage();
    IRErc20Delegator RErc20 = $.RErc20;
    IERC20 asset = IERC20(asset());

    uint256 currentAssets = asset.balanceOf(address(
       this));
    if (currentAssets > 0) {
        asset.safeIncreaseAllowance(address(RErc20),
           currentAssets);
        RErc20.mint(currentAssets);
    }
}
```

PoC

Below is a PoC that the implementation may break the `EIP4626` that `maxDeposit MUST return the maximum amount of assets deposit would allow to be deposited for receiver and not cause a revert` in extreme cases.

```
    function testRevertOnMaxDepositInExtremeCase() public {

        // deposit 10 times to quickly increase the totalSupply of the market
        for (uint i = 0; i < 10; i++) {
            console.log("Deposit times", i);
            uint256 amount = vault.maxDeposit(address(this));
            console.log("   maxDeposit returned", amount);
            console.log("   totalSupply for the market", RUSDC.totalSupply());

            deal(address(USDC), address(this), amount);
            USDC.approve(address(vault), amount);
            vault.deposit(amount, address(this));
        }

        // emergency withdraw to reset the totalSupply of the market
        console.log("Emergency withdraw 1");
        vault.emergencyWithdraw();
        vault.unpause();
        uint256 amount = vault.maxDeposit(address(this));
        console.log("   maxDeposit", amount);
        deal(address(USDC), address(this), amount);
        USDC.approve(address(vault), amount);
        vault.deposit(amount, address(this));
        console.log("   totalSupply for the market", RUSDC.totalSupply());

        // emergency withdraw to reset the totalSupply of the market, deposit again, but this time a revert would be triggered
        vault.emergencyWithdraw();
        console.log("Emergency withdraw 2");
        vault.unpause();
        amount = vault.maxDeposit(address(this));
        console.log("   maxDeposit", amount);
        deal(address(USDC), address(this), amount);
        USDC.approve(address(vault), amount);
        console.log("   Revert expected");
        vm.expectRevert();
        vault.deposit(amount, address(this));
    }
```

The output log:

```
[PASS] testRevertOnMaxDepositInExtremeCase() (gas:
   4208079)
Logs:
  Deposit times 0
     maxDeposit returned 1768000275976
     totalSupply for the market 2534419651702
  Deposit times 1
     maxDeposit returned 45845709072651
     totalSupply for the market 4279238525135
  Deposit times 2
     maxDeposit returned 38203171170969
     totalSupply for the market 49523834661714
  Deposit times 3
     maxDeposit returned 1561781157966
     totalSupply for the market 87226099186142
  Deposit times 4
     maxDeposit returned 35620637702
     totalSupply for the market 88767402814665
  Deposit times 5
     maxDeposit returned 798003309
     totalSupply for the market 88802556406997
  Deposit times 6
     maxDeposit returned 17868631
     totalSupply for the market 88803343947165
  Deposit times 7
     maxDeposit returned 401458
     totalSupply for the market 88803361581508
  Deposit times 8
     maxDeposit returned 7435
     totalSupply for the market 88803361977702
  Deposit times 9
     maxDeposit returned 0
     totalSupply for the market 88803361985039
  Emergency withdraw 1
     maxDeposit 1768000275975
     totalSupply for the market 90548180858487
  Emergency withdraw 2
     maxDeposit 1768000275976
     Revert expected
```

Recommendation

To address this issue, the `_deposit_` function should include a check against the `maxDeposit` calculation to ensure the actual deposited amount does not exceed the intended limit. This would prevent any scenario where excess assets in the vault could bypass the maxDeposit constraint.

Status

Cymetrics:Acknowledged.

5.2.2 `vault::execute` Is Over-Designed To Retrieve `Ether` from the `vault`

Description

The `vault::execute` function is designed to allow the contract owner to transfer `Ether` or other assets out of the vault. However, the vault contract currently lacks any mechanism to receive `Ether`, making the `value_` parameter in the execute function unnecessary and over-designed.

```
function execute(address to_, uint256 value_, bytes calldata data_)
    external
    onlyOwner
    returns (bool, bytes memory)
{
    (bool success, bytes memory result) = to_.call{value: value_}(data_);
    return (success, result);
}
```

The following PoC demonstrates that the vault cannot accept `Ether` directly sent to it:

```
// Below is a PoC to show that vault can not accept any ether directly sent to it
function testVaultCanNotAcceptEther() public {
    uint256 amount = 1 ether;
    (bool success, ) = address(vault).call{value: amount}("");
    assertEq(success, false);
}
```

With the output:

```
Ran 1 test for test/vaultsUpgradable/v1/Vault.t.sol:
   VaultUpgradableTest
[PASS] testVaultCanNotAcceptEther() (gas: 19798)
Traces:
  [19798] VaultUpgradableTest::testVaultCanNotAcceptEther
     ()
    ├─ [7989] Proxy::fallback{value:
       1000000000000000000}()
    │   ├─ [2307] Beacon::implementation() [staticcall]
    │   │   └─ ← [Return] Vault: [0
       x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f]
    │   ├─ [44] Vault::fallback{value:
       1000000000000000000}() [delegatecall]
    │   │   └─ ← [Revert] EvmError: Revert
    │   └─ ← [Revert] EvmError: Revert
    └─ ← [Stop]
```

Additionally, using `forge inspect src/vaultsUpgradable/v1/Vault.sol:Vault abi > vault.abi`. By inspecting the contract ABI, it is confirmed that no payable function exists to receive Ether. Therefore, there would be no `Ether` in the vault, rendering the `value_` parameter in execute redundant and over-designed.

Code Snippet

```
    function execute(address to_, uint256 value_, bytes
       calldata data_)
        external
        onlyOwner
        returns (bool, bytes memory)
    {
        (bool success, bytes memory result) = to_.call{
           value: value_}(data_);
        return (success, result);
    }
```

Recommendation

To mitigate this issue, it is recommended to:

1. Remove the `value_` parameter and its usage if no future modifications are planned to enable the vault to receive `Ether`. This will streamline the function and eliminate unnecessary complexity.

Status

Cymetrics:Acknowledged.

5.3 Info Risk Findings

5.3.1 Redundant `roleAdmin` Assignment

Description

In the `Vault` contract, the `initialize` function is used to set up initial states and role administration. Within this function, the `KEEPER_ROLE` is granted to a specified address and assigned `DEFAULT_ADMIN_ROLE` as its `roleAdmin`.

```
    function initialize(IERC20 asset_, string memory name_
       , string memory symbol_, address keeper_)
        public
        virtual
        initializer
    {
        __ERC4626_init(asset_);
        __ERC20_init(name_, symbol_);
        __AccessControl_init();
        __Pausable_init();
        __ReentrancyGuard_init();

        // set role
        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
@=>     _setRoleAdmin(KEEPER_ROLE, DEFAULT_ADMIN_ROLE);
        _grantRole(KEEPER_ROLE, keeper_);
    }
```

However, the `DEFAULT_ADMIN_ROLE` is the default `roleAdmin` for all roles as per the `AccessControlUpgradeable` contract:

```
 * By default, the admin role for all roles is `
    DEFAULT_ADMIN_ROLE`, which means
 * that only accounts with this role will be able to grant
    or revoke other
 * roles. More complex role relationships can be created
    by using
 * {_setRoleAdmin}.
```

This means that explicitly setting `DEFAULT_ADMIN_ROLE` as the roleAdmin

for `KEEPER_ROLE` is redundant and unnecessary, as this will be the case by default. This redundancy can lead to unnecessary confusion and bloated code.

Code Snippet

Vault::initialize

```
    function initialize(IERC20 asset_, string memory name_
       , string memory symbol_, address keeper_)
        public
        virtual
        initializer
    {
        __ERC4626_init(asset_);
        __ERC20_init(name_, symbol_);
        __AccessControl_init();
        __Pausable_init();
        __ReentrancyGuard_init();

        // set role
        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
@=>     _setRoleAdmin(KEEPER_ROLE, DEFAULT_ADMIN_ROLE);
        _grantRole(KEEPER_ROLE, keeper_);
    }
```

PoC

The following PoC demonstrates that `DEFAULT_ADMIN_ROLE` is the default `roleAdmin` for any role(under `VaultUpgradableTest`):

```
1     // Below is a PoC to show `DEFAULT_ADMIN_ROLE` is the
          `roleAdmin` for any role by default.
2     function testDefaultAdminRoleIsRoleAdminByDefault()
          public {
3         bytes32 TEST_ROLE = keccak256("TEST_ROLE");
4
5         // Verify that DEFAULT_ADMIN_ROLE is the admin for
              TEST_ROLE
6         assertEq(vault.getRoleAdmin(TEST_ROLE), vault.
              DEFAULT_ADMIN_ROLE());
7
8         // Verify that the test contract (which is the
              deployer) has the DEFAULT_ADMIN_ROLE
9         assertTrue(vault.hasRole(vault.DEFAULT_ADMIN_ROLE
              (), address(this)));
10    }
```

Recommendation

To mitigate this issue, it is recommended to remove the redundant assignment of `DEFAULT_ADMIN_ROLE` as the `roleAdmin` for `KEEPER_ROLE`. This will simplify the code and avoid any potential confusion.

Status

Cymetrics:Fixed.Commit hash:b44bc859aad4eed569f426339f62ca468264c841

5.3.2 Foundry Console Import Should Be Removed

Description

The contract files for `Vault`, `AaveVault`, `AmbientVault`, and `RhoMarketsVault` include references to the `console` contract from Foundry. The console contract is intended for development and testing purposes and should not be included in production code.

```
1 import "forge-std/console.sol";
```

Including the `console` import in production contracts negatively impacts the cleanliness and professionalism of the code.

Code Snippet

```
1 // https://github.com/lazyotter-finance/lazyotter-contract
    /blob/ca1ca1ff8e56fdd29d7defdcd957a97bf0dab521/src/
    vaultsUpgradable/v1/AaveVault.sol#L13-L14
2 // https://github.com/lazyotter-finance/lazyotter-contract
    /blob/ca1ca1ff8e56fdd29d7defdcd957a97bf0dab521/src/
    vaultsUpgradable/v1/AmbientVault.sol#L14
3 // https://github.com/lazyotter-finance/lazyotter-contract
    /blob/ca1ca1ff8e56fdd29d7defdcd957a97bf0dab521/src/
    vaultsUpgradable/v1/RhoMarketsVault.sol#L17
4 // https://github.com/lazyotter-finance/lazyotter-contract
    /blob/ca1ca1ff8e56fdd29d7defdcd957a97bf0dab521/src/
    vaultsUpgradable/v1/Vault.sol#L10
5 import "forge-std/console.sol";
```

Recommendation

To mitigate this issue, it is recommended to remove all `console` imports from the smart contract code before deployment. This ensures the contracts are clean, secure, and free from unnecessary dependencies.

Status

Cymetrics:Fixed.Commit hash:b44bc859aad4eed569f426339f62ca468264c841

5.3.3 Typo Error in Storage Location Calculaiton

Description

In the AmbientVault contract, the storage location AmbientVaultStorageLocation is calculated using the following expression:

```
bytes32 private constant AmbientVaultStorageLocation =
    keccak256(abi.encode(uint256(keccak256("
       ambientVaultStorage")) - 1)) & ~bytes32(uint256(0
       xff));
```

However, the comment above this calculation incorrectly references `aaveVaultStorage` instead of `ambientVaultStorage` due to a copy-paste error:

```
    // keccak256(abi.encode(uint256(keccak256("
       aaveVaultStorage")) - 1)) & ~bytes32(uint256(0xff))
    bytes32 private constant AmbientVaultStorageLocation =
        0
           x1543609c7215d70dab835e07add09594386b5e07f744a59e8ae128e3d
           ;
```

This typo causes inconsistency between the comment and the actual code, which can lead to confusion for developers reviewing or maintaining the contract.

Code Snippet

AmbientVaultStorageLocation Definition

```
    // keccak256(abi.encode(uint256(keccak256("
       aaveVaultStorage")) - 1)) & ~bytes32(uint256(0xff))
    bytes32 private constant AmbientVaultStorageLocation =
        0
           x1543609c7215d70dab835e07add09594386b5e07f744a59e8ae128e3d
           ;
```

POC

The following PoC demonstrates that `0x1543609c7215d70dab835e07add0959438` is the output of `keccak256(abi.encode(uint256(keccak256("ambientVaultStorage"))- 1))& ~bytes32(uint256(0xff))` (under `VaultUpgradableTest`)

```
    // The following PoC demonstrates that `0
       x1543609c7215d70dab835e07add09594386b5e07f744a59e8ae128e3db8a8
       ` is the output of `keccak256(abi.encode(uint256(
       keccak256("ambientVaultStorage")) - 1)) & ~bytes32(
       uint256(0xff))` (under `VaultUpgradableTest`)
    function testAmbientVaultStorageLocation() public {
        bytes32 ambientVaultStorageLocation = keccak256(
           abi.encode(uint256(keccak256("
           ambientVaultStorage")) - 1)) & ~bytes32(uint256
           (0xff));
        assertEq(ambientVaultStorageLocation, bytes32(0
           x1543609c7215d70dab835e07add09594386b5e07f744a59e8ae128e3d
           ));
    }
```

Recommendation

To mitigate this issue, it is recommended to correct the comment to match the actual code, replacing `aaveVaultStorage` with `ambientVaultStorage`. This will prevent any potential confusion and maintain consistency between the code and its documentation.

Status

Cymetrics:Fixed.Commit hash:b44bc859aad4eed569f426339f62ca468264c841

6 Appendix: Technical doc

6.1 Centralization risk

In the Lazy Otter project, there are two types of centralized roles: - **Admin**: Admins have the authority to pause the vault, unpause the vault, perform emergency withdrawals, and withdraw any remaining balance from the vault. - **Keeper**: Keepers have the authority to pause the vault, unpause the vault, and perform emergency withdrawals.

Please check the main report for related findings.

6.2 Scroll vs. Ethereum differences

LazyOtter is meant to be deployed on Scroll solely, therefore it is valuable to investigate the difference between Scroll and Ethereum to avoid subtle bugs. This is documented in Scroll doc: https://docs.scroll.io/en/developers/ethereum-and-scroll-differences/.

A few things to take notes:

- Although Scroll uses sequencer, frontrun is still possible (as we have seen in Rho Markets price oracle manipulation + MEV frontrunning)
- Total gas fee is higher on Scroll since it combines L1 gas fee + L2 gas fee
- Block time is 3-second on Scroll during normal hours, which is a lot faster than Ethereum (12-second)

6.3 Classic vault attacks

6.3.1 Inflation attack

Lazyotter utilizes "virtual decimals" `_decimalsOffset`=6 to mitigate the famous inflation attack / first depositor frontrunning attack, follow the implementation of OpenZeppelin's implementation of ERC4626Upgradeable.

This decimals offset significantly increases the cost of "donation" by the attacker, therefore mitigates the inflation attack.

6.3.2 Vault reset attack

Vault reset attack was described in Kankodu's tweet. This attack is mitigated by virtual decimal offset too.

6.3.3 Rounding directions

Rounding direction should always be in favor of the protocol. In other words, a correct implementation of ERC4626 should let users suffer a tiny bit of loss in exchange of protocol security.

Currently follow the implementation of OpenZeppelin's implementation of ERC4626Upgradeable.

6.3.4 Slippage

The idea of slippage is similar to that of AMM. You can think of `Vault.mint()` as a type of "swap()" as in AMM. In a secure implementation of ERC-4626 vault, it is neccessary to consider slippage to protect users' asset. Currently there is no slippage protection in Lazyotter. If slippage needs to be considered in the future, refer to ERC4626RouterBase.sol

6.3.5 Reentrancy

All user-level external functions are guarded by `nonReentrant` modifier, therefore simple reentrancy attacks are impossible.

6.4 Vault functionalities analysis

6.4.1 Emergency withdrawal

Emergency withdrawl gives admin the authority to pause the vault and withdraw all funds from external markets. Beyond `emergencyWithdraw()` function, there is also an `execute()` admin function that can withdraw a certain amount of ETH from the vault itself.

6.4.2 Types of vaults

There are four types of vaults in the scope:

- **Vault** -> the parent contract for all child vaults
- **AaveVault** -> interacts with Aave v3
- **RhoMarketsVault** -> interacts with Rho Markets, which is the first native lending protocol on Scroll
- **AmbientVault** -> interacts with Ambient Finance, but the logic is implemented in AmbientVaultHelper.sol, an out-of-scope contract.

In all vaults, user deposit is sent to Aave / RhoMarket/Ambient pool as LP in order to generate profit.

6.5 External Protocol Integration

The AaveVault.sol integrates with the Aave V3 lending pool. In the monitoring section, it is recommended to include synchronization of Aave V3' s status.

The RhoMarketsVault.sol integrates with Rho Markets. The RErc20Delegator may dynamically modify the comptroller and interestRateModel. Failure to synchronize these updates in RhoMarketsVault.sol could affect the accuracy of values. Please check the main report for that finding.

6.6 Other comments

1. Rho Markets suffered from a price oracle manipulation attack recently: https://olympixai.medium.com/rho-markets-on-scroll-exploit-analysis-965991270f56. The story sounds suspicious since the root cause was private key leak. Since LazyOtter interacts with Rho Markets in one of the vaults, please consider the risk of Rho Markets itself.
2. Currently AmbientVault.sol does not contain much logic. The interaction between Ambient Finance and LazyOtter ambientVault is implemented in AmbientVaultHelper.sol, but that contract is out of scope for this audit.

7 Appendix: 4naly3er Report

https://hackmd.io/@xhZ0PzqQRXWqTO8hmw7TlA/rkGw29-hC