Cymetrics

Lazy Otter Audit Report

Cymetrics

July 30, 2024

Contents

List of Figures

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 June 12th, 2024 to June 19th, 2024. One 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/tree/develop/src
Commit hash	b2c87185f8fb95c01b211459d55efab843198a13
File in Scope	every contracts under `src/`

Issues Found

Severity	Count	Fixed	Acknowledged
High	1	1	0
Medium	9	0	9
Low	6	0	6
Informational	3	0	3
Total	19	1	18

5 Findings

5.1 High Risk Findings

5.1.1 Wrong rounding directions in Vault.sol

Description

There are 3 instances where rounding direction deviates from EIP-4626:

- `maxWithdraw()` should round down (now it is rounding up)
- `previewWithdraw()` should round up (now it is rounding down)
- `previewRedeem()` should round down (now it is rounding up)

Quoted from EIP-4626:

> EIP-4626 Vault implementers should be aware of the need for specific, opposing rounding directions across the different mutable and view methods, as it is considered most secure to favor the Vault itself during calculations over its users:
>
> If (1) it's calculating how many shares to issue to a user for a certain amount of the underlying tokens they provide or (2) it's determining the amount of the underlying tokens to transfer to them for returning a certain amount of shares, it should round down.
>
> If (1) it's calculating the amount of shares a user has to supply to receive a given amount of the underlying tokens or (2) it's calculating the amount of underlying tokens a user has to provide to receive a certain amount of shares, it should round up.

For example, currently `previewRedeem()` rounds up, which means user can get slightly more assets when redeeming. Although the impact seems low for a single redeem, it can be dangerous when vault operates for a long time or arbitrager deliberately exploits it when gas fee is low. This issue is more

serious when decimal is low (such as USDC - 6 decimals), since `1/10**6` is a lot larger than `1/10**18`.

Code Snippet

- Vault.sol#L145-L147
- Vault.sol#L181-L183
- Vault.sol#L190-L192

Recommendation

Fix rounding directions according to EIP-4626.
For a reference of actual implementation:

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/extensions/ERC4626.sol

Status

Cymetrics: Fixed.Commit hash : e4d76f9e64c57e3b9018698aec4b1b5935720436

5.2 Medium Risk Findings

5.2.1 Admin can adjust fee rate to harm fee recipients and vault users

Description
Consider the following scenario:

- After a user deposits or mints, the admin can arbitrarily call setFeeInfo to reset `FeeInfo`. The admin can set `withdrawalFeeRate` to 100%.
- Admin can arbitrarily call `setFeeInfo()` to reset `FeeInfo`.

The admin can set `withdrawalFeeRate` to 0, the recipients are unable to receive the withdrawal fee.

Code Snippet

- Vault.sol#L429

Recommendation
Ensure that the protocol team and its users are aware of the risks of such an event and develop a contingency plan to manage it.

Status

Cymetrics: Acknowledged.

5.2.2 No slippage control for deposit/mint/withdraw/redeem

Description
Currently there is no slippage protection in deposit/mint/withdraw/redeem.

Quoted from EIP-4626 security considerations:

> If implementors intend to support EOA account access directly, they should consider adding an additional function call for deposit/mint/withdraw/redeem with the means to accommodate slippage loss or unexpected deposit/withdrawal limits, since they have no other means to revert the transaction if the exact output amount is not achieved.

This issue is only impactful when share price can go down. In current setting it seems the share price will never go down, but this can change in the future when more external integrations are used.

Code Snippet

- Vault.sol#L228
- Vault.sol#L246
- Vault.sol#L264
- Vault.sol#L304

Recommendation
Implement slippage control (as function input) for deposit/mint/withdraw/redeem.

Status

Cymetrics: Acknowledged.

5.2.3 Some tokens might not have `decimals()` implemented

Description

Quoted from EIP-20:

> Returns the number of decimals the token uses - e.g. 8, means to divide the token amount by 100000000 to get its user representation.
>
> OPTIONAL - This method can be used to improve usability, but interfaces and other contracts MUST NOT expect these values to be present.

In `Vault.decimals()`:

```
1     function decimals() public view override returns (
         uint8) {
2        return ERC20(address(asset)).decimals() +
            _decimalsOffset();
3     }
```

`ERC20(address(asset)).decimals()` might not return expected decimals for some ERC20 without `decimals()` method implemented.

Code Snippet

- Vault.sol#L107

Recommendation

Implement a try-catch function to query ERC20 token decimals, for example, like what OpenZeppelin implemented:

- ERC4626.sol#L86-L97

```
function _tryGetAssetDecimals(IERC20 asset_) private view returns (bool, uint8) {
    (bool success, bytes memory encodedDecimals) = address(asset_).staticcall(
        abi.encodeCall(IERC20Metadata.decimals, ())
    );
    if (success && encodedDecimals.length >= 32) {
        uint256 returnedDecimals = abi.decode(encodedDecimals, (uint256));
        if (returnedDecimals <= type(uint8).max) {
            return (true, uint8(returnedDecimals));
        }
    }
    return (false, 0);
}
```

Status

Cymetrics: Acknowledged.

5.2.4 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#L482

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: Acknowledged.

5.2.5 Vault.sol is not compatible with EIP-4626

There are multiple locations in the Vault.sol that do not conform to ERC-4626 specifications:
1. maxMint(), maxDeposit should return the value 0 when mint(), deposit() is paused.

Description
The whenTokenNotPaused modifier is used for deposit() and mint() functions to ensure that these functionalities cannot be used when the vault is paused. According to EIP-4626 specifications:

maxDeposit

```
MUST factor in both global and user-specific limits, like
    if deposits are entirely disabled (even temporarily) it
     MUST return 0.
```

maxMint

```
MUST factor in both global and user-specific limits, like
    if mints are entirely disabled (even temporarily) it
    MUST return 0.
```

maxDeposit(), maxMint() should return the 0 during deposit(), mint() is paused.

Code Snippet

- Vault.sol#L123

- Vault.sol#L132

2. previewWithdraw(), previewRedeem() does not include withdrawal fees.

Description

According to EIP-4626 specifications:

previewWithdraw

```
MUST be inclusive of withdrawal fees. Integrators should
    be aware of the existence of withdrawal fees.
```

previewRedeem

```
MUST be inclusive of withdrawal fees. Integrators should
    be aware of the existence of withdrawal fees.
```

Code Snippet

- Vault.sol#L181
- Vault.sol#L190

Recommendation

Fix functions mentioned above according to EIP-4626. For a reference, here is a similar issue reported by OpenZeppelin:

https://blog.openzeppelin.com/pods-finance-ethereum-volatility-vault-audit-2#non-standard-erc-4626-vault-functionality

Status

Cymetrics: Acknowledged.

5.2.6 Vault.execute() gives unnecessary power to the admin

Description

There are two "emergency withdrawal" type of admin functions in the vault:

- `emergencyWithdraw()`: withdraw all or a portion of assets from Aave or Layerbank
- `execute()`: arbitrary low-level call by admin

The intention behind `execute()` is to let admin withdraw user funds when upgrading / vault is under attack. However, arbitrary low-level call is a lot more "powerful" (whilst dangerous) and it does not follow least privilege principle.

Code Snippet

- Vault.sol#L477-L484

Recommendation

Implement `sweep()` instead of `execute()`. A typical sweep() admin function will allow admin to withdraw a certain type of asset only.

Status

Cymetrics: Acknowledged.

5.2.7 Users can still withdraw / redeem when vault is paused

Description

Among `deposit()`, `mint()`, `withdraw()` and `redeem()`, only `deposit()` and `mint()` are guarded by `whenNotPaused` modifier. In other words, Attacker can withdraw even if vault is paused.

When admins spot something wrong in the vault and try to pause, attacker could already own a portion of the shares in the vault. If `withdraw()` and `redeem()` are not guarded by `whenNotPaused`, attacker can still get away with whatever profit he already made. Therefore it is better to pause all functionalities of the vault and unpause when issues are resolved.

Code Snippet

- Vault.sol#L264
- Vault.sol#L304

Recommendation

Add `whenNotPaused` modifier to both `withdraw()` and `redeem()`.

Status

Cymetrics: Acknowledged.

5.2.8 Harvest functionalities are unuseable

Description

At this moment there is no implementation for `_harvest()`. In Vault.sol, `_harvest()` always returns 0:

```
1     function _harvest() internal virtual returns (uint256)
          {
2         return 0;
3     }
```

In AaveVault.sol, `_harvest()` is commented out:

```
// function _harvest() internal override returns (uint256) {
//     address self = address(this);
//     uint256 beforeAssets = asset.balanceOf(self);

//     address[] memory aTokens = new address[](1);
//     aTokens[0] = address(aToken);

//     (address[] memory rewardsList,) = rewardsController.claimAllRewardsToSelf(aTokens);

//     uint256 rewardsListLength = rewardsList.length;

//     if (rewardsListLength == 0) {
//         return 0;
//     }

//     for (uint256 i = 0; i < rewardsListLength; i++) {
//         // This function will swap the reward token for the asset token.
//         // However, we haven't yet decided which DEX to use.
//         // _processReward(rewardsList[i]);
//     }

//     uint256 afterAssets = asset.balanceOf(self);
//     uint256 harvestAssets = afterAssets - beforeAssets;
//     return harvestAssets;
// }
```

The result is that `harvest()` does nothing since `harvestAssets` is always 0, rendering the harvest functionalities useless.

Code Snippet

- Vault.sol#L389-L391
- Vault.sol#L90-L114

Recommendation

Either implement `_harvest()` in AaveVault.sol or remove this functionality

all together.

Status

Cymetrics: Acknowledged.

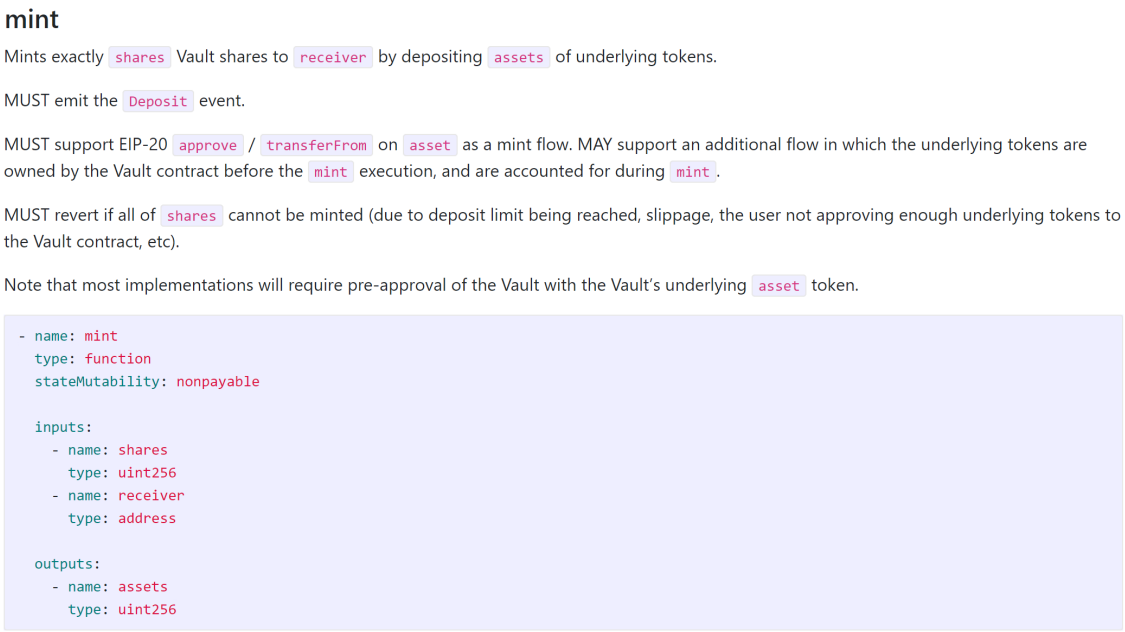
5.2.9 Vault.mint() returns wrong value

Description

Current implementation of `Vault.mint()` returns numbers of shares as return value:

```
1     function mint(uint256 shares, address receiver)
         external nonReentrant whenNotPaused returns (
         uint256) {
2        uint256 assets = previewMint(shares);
3
4        _mint(receiver, shares);
5        asset.safeTransferFrom(msg.sender, address(this),
            assets);
6        _deposit(receiver, assets);
7
8        emit Deposit(msg.sender, receiver, assets, shares)
            ;
9        return shares; // @audit-issue should return
            assets
10    }
```

Per EIP-4626, `mint()` function should return numbers of assets:



mint

Mints exactly `shares` Vault shares to `receiver` by depositing `assets` of underlying tokens.

MUST emit the `Deposit` event.

MUST support EIP-20 `approve` / `transferFrom` on `asset` as a mint flow. MAY support an additional flow in which the underlying tokens are owned by the Vault contract before the `mint` execution, and are accounted for during `mint`.

MUST revert if all of `shares` cannot be minted (due to deposit limit being reached, slippage, the user not approving enough underlying tokens to the Vault contract, etc).

Note that most implementations will require pre-approval of the Vault with the Vault's underlying `asset` token.

```
- name: mint
  type: function
  stateMutability: nonpayable

  inputs:
    - name: shares
      type: uint256
    - name: receiver
      type: address

  outputs:
    - name: assets
      type: uint256
```

Figure 1: EIP-4626 mint

This is problematic if further computations done by users / frontend utilize the return value of `mint()`.

Code Snippet

- Vault.sol#L254

Recommendation

Change the code to:

```
1     function mint(uint256 shares, address receiver)
         external nonReentrant whenNotPaused returns (
         uint256) {
2        uint256 assets = previewMint(shares);
3
4        _mint(receiver, shares);
5        asset.safeTransferFrom(msg.sender, address(this),
            assets);
6        _deposit(receiver, assets);
7
8        emit Deposit(msg.sender, receiver, assets, shares)
            ;
9        return assets;
10    }
```

Status

Cymetrics: Acknowledged.

5.3 Low Risk Findings

5.3.1 USDC/USDT blacklisted fee recipients can brick `Vault.withdraw()` and `Vault.redeem()`

Description

When the token implements a blacklist, which is common for tokens (e.g. USDC/USDT implementing blacklist/blocklist; See: https://github.com/d-xo/weird-erc20).

Currently `Vault.withdraw()` and `Vault.redeem()` implement a for loop that transfers withdrawal fee to a list of recipients. If any of the recipients is blacklisted by USDC/USDT, the entire `Vault.withdraw()` / `Vault.redeem()` function will be bricked. In other words, even users can't withdraw their funds, the vault will be DoSed permanently.

We understand that fee recipients can be updated by admin calling `setFeeInfo()`, therefore the impact is small.

The following steps describe this issue, consider this scenario:

1. Fee recipients was set via `setFeeInfo()` when it was not on the token blacklist.
2. A fee recipient was blacklisted before user calls `withdraw()` or `redeem()`.
3. User calls `withdraw()` or `redeem()`, but because the fee recipient was blacklisted, the `safeTransfer()` to recipient would revert, and user is unable to retrieve assets from the vault.

Code Snippet

- Vault.sol#L287
- Vault.sol#L323

Recommendation

In `withdraw()` and `redeem()`, consider checking if the fee recipient is

blacklisted. If so, send that fee to a temporary place and let admin extract the fee later.

Or even better, refactor the code using the "pull over push" pattern: set up accounting and let fee recipients claim the fee at a later time, instead of sending tokens to them.

Status

Cymetrics: Acknowledged.

5.3.2 Uniswap V3 fee tiers

Description

Uniswap V3 has 0.01% fee tier. What if the only existent pool is of 0.01% fee tier, in other words, what if 0.05%/0.3%/1% pools don't exist? In that case, `getBestFee()` will return 0, rendering this function unless.

Reference: https://support.uniswap.org/hc/en-us/articles/20904283758349-What-are-fee-tiers

Code Snippet

- Vault.sol#L21-L24

Recommendation

Add another fee tier 0.01% in `getBestFee()`.

Status

Cymetrics: Acknowledged.

5.3.3 Uniswap V3 fee calculation

Description

The function `getBestFee()` only selects a pool with largest liquidity, but does not consider maximum payout for LP. Consider a toy example:

- Pool 1: 0.05% fee, 1001 liquidity
- Pool 2: 1% fee, 1000 liquidity

The current implementation will select Pool 1, but obviously Pool 2 benefits LP more.

In our understanding, this function should select a pool with highest expected profit for LP.

Code Snippet

- Vault.sol#L20

Recommendation

In `getBestFee()`, consider simulating profit computation and select a pool with highest expectetd profit.

Status

Cymetrics: Acknowledged.

5.3.4 Can steal ETH from the ETHVaultHelper.sol

Description

Consider the following two scenarios:

1. Honey pot fake vault:

- ETHVaultHelper.sol lacks verification of the vault address.
- A malicious user can create a fake honey pot vault.
- When users call mintETH() or depositETH(), they can pass in the malicious vault address.
- Because the vault is fake, the ETH sent by users cannot be retrieved using withdrawETH() or redeemETH().

2. User accidentally sends ETH:

- In mintETH(), when if `(balance > 0)`, the contract sends all ETH in the contract to the caller.
- User A accidentally sends ETH to ETHVaultHelper.sol.
- User B then calls mintETH(), and due to `uint256 balance = WETH.balanceOf(address(this))`, B withdraws the ETH accidentally sent by A.

Code Snippet

- VaultHelper.sol

Status

Cymetrics: Acknowledged.

5.3.5 Potential DoS in withdraw() and redeem()

Description

The current implementation of the `setFeeInfo()` does not have a length limit on the array. This can lead to potential DoS in `withdraw()` and `redeem()`. If a large number of fee info are set, the for loop in the `withdraw()`, `redeem()` which distributes the withdrawal fee, can consume excessive gas, potentially making the transaction fail.

Code Snippet

- Vault.sol#L429

Recommendation

Implement length checks in `setFeeInfo()`.

Status

Cymetrics: Acknowledged.

5.3.6 Small withdrawals can evade `withdrawalFee` due to precision loss

Description

The current implementation of the `withdraw()` calculates withdrawalFee based on the amount of assets being withdrawn. However, when the withdrawal assets is very small, the precision loss in the fee calculation can result in the fee being rounded down to zero. This allows users to perform multiple small withdrawals, effectively evading the `withdrawalFee`.

Code Snippet

- Vault.sol#L281
- Vault.sol#L317

Recommendation

Add 1 when calculating `withdrawalFee`:

```
uint256 withdrawalFee = (assets * feeInfo.
    withdrawalFeeRate) / MAX_FEE_RATE + 1;
```

Status

Cymetrics: Acknowledged.

5.4 Informational

5.4.1 Redundant `_setRoleAdmin()` call

Description

This line of code is redundant. Code for `_setRoleAdmin`:

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/83c7e45092dac350b07
L174

```
1     function _setRoleAdmin(bytes32 role, bytes32 adminRole
         ) internal virtual {
2         bytes32 previousAdminRole = getRoleAdmin(role);
3         _roles[role].adminRole = adminRole;
4         emit RoleAdminChanged(role, previousAdminRole,
             adminRole);
5     }
```

Here it is setting `_roles[keccak256("KEEPER_ROLE")].adminRole = 0x00;`, but `_roles[role].adminRole` defaults to 0x00 already, so calling `_setRoleAdmin(KEEPER_ROLE, DEFAULT_ADMIN_ROLE);` does nothing. In other words, `DEFAULT_ADMIN_ROLE` is the default admin for every single role, therefore it is unnecessary to set it again.

Code Snippet

- Vault.sol#L80

Recommendation

Remove call to `_setRoleAdmin()` in Vault.sol constructor:

```
    constructor(
        IERC20 _asset,
        string memory name,
        string memory symbol,
        FeeInfo memory _feeInfo,
        address _keeper
    ) ERC20(name, symbol) {
        asset = _asset;

        // role
        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
        _grantRole(KEEPER_ROLE, _keeper);

        setFeeInfo(_feeInfo);
    }
```

Status

Cymetrics: Acknowledged.

5.4.2 Use AccessControlDefaultAdminRules instead AccessControl to reduce centralization risk

Description

In OpenZeppelin AccessControl.sol, The role `DEFAULT_ADMIN_ROLE` has almost root authority, which increases the centralization risk of the system. There is also an extension called AccessControlDefaultAdminRules.sol, which adds some restrictions to this root role.

Quoted from OpenZeppelin doc:

> This mechanism can be used to create complex permissioning structures resembling organizational charts, but it also provides an easy way to manage simpler applications. AccessControl includes a special role, called DEFAULT_ADMIN_ROLE, which acts as the default admin role for all roles. An account with this role will be able to manage any other role, unless _setRoleAdmin is used to select a new admin role.
>
> Since it is the admin for all roles by default, and in fact it is also its own admin, this role carries significant risk. To mitigate this risk we provide AccessControlDefaultAdminRules, a recommended extension of AccessControl that adds a number of enforced security measures for this role: the admin is restricted to a single account, with a 2-step transfer procedure with a delay in between steps.

Code Snippet

- Vault.sol#L5

Recommendation

Use AccessControlDefaultAdminRules.sol instead of AccessControl.sol.

Status

Cymetrics: Acknowledged.

5.4.3 Vault.mint() lacks maxMint() check

Description

There is no maxMint check in `Vault.mint()` although maxMint() is implemented. There is no impact at this moment since maxDeposit() always returns `type(uint256).max`, but if it changes in the future then Vault.mint() needs to check maxMint().

Just for completeness, here is the code for maxDeposit() and maxMint():

```
1     function maxDeposit(address account) public view
         virtual returns (uint256) {
2         return type(uint256).max;
3     }
```

```
1     function maxMint(address receiver) public view returns
         (uint256) {
2         uint256 _maxDeposit = maxDeposit(receiver);
3         if (_maxDeposit == type(uint256).max) {
4             return type(uint256).max;
5         }
6         return _convertToShares(_maxDeposit, Math.Rounding
             .Floor);
7     }
```

Currently `return _convertToShares(_maxDeposit, Math.Rounding.Floor);` is unreachable, because maxDeposit() always returns type(uint256).max. However, since maxDeposit() is a virtual function, new vaults might override it to much smaller value, so Vault.mint() will need to check maxMint().

Code Snippet

- Vault.sol#L246-L255

Recommendation

Add maxMint() check inside Vault.mint():

```
uint256 maxShares = maxMint(receiver);
if (shares > maxShares) {
    revert("exceed maxMint limit");
}
```

Status

Cymetrics: Acknowledged.

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 reset vault fees, 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 Classic vault attacks

6.2.1 Inflation attack

Lazyotter utilizes "virtual decimals" `_decimalsOffset`=6 to mitigate the famous inflation attack / first depositor frontrunning attack, similar to OpenZeppelin's implementation of ERC4626.

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

6.2.2 Vault reset attack

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

6.2.3 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.

There are a few cases in Vault.sol where ERC4626 standard isn't strictly followed. Please check the main report for that finding.

6.2.4 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, and we already addressed this issue in the main report.

6.2.5 Reentrancy

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

6.3 Vault functionalities analysis

6.3.1 Fees

Compared to standard EIP-4626 vault, Lazyotter implements withdrawal fee (no deposit fee). The fee is computed in both `Vault.withdraw()` and `Vault.redeem()`.

6.3.2 Emergency withdrawal

Emergency withdrawl gives admin the authority to pause the vault and withdraw all funds in it. Beyond emergencyWithdraw() function, there is also an execute() admin function that can withdraw all funds without pausing the vault.

6.3.3 Harvest

User deposits will be sent to Aave / Layerbank as LP, therefore the vault will generate yields and users should be able to harvest their profit. But currently harvest functionality is not implemented.

6.3.4 Types of vaults

There are two types of vaults in the scope:

- AaveVault
- LayerBankVault

In both vaults, user deposit is sent to Aave / Layerbank pool as LP in order to generate profit. However, LayerBankVault has a caveat: the underlying asset USDC has 6 decimals but the LP token iUSDC has 18 decimals. This creates complication in further computation.

7 Appendix: 4nalyzer report

https://gist.github.com/ret2basic/fa0e98eadc552a2c80faeaabe94fb324