Smart Contract
Security Audit Report

Table Of Contents

1 Executive Summary

On 2023.11.07, the SlowMist security team received the Lazyotter team's security audit application for Lazyotter, developed the audit plan according to the agreement of both parties and the characteristics of the project, and finally issued the security audit report.

The SlowMist security team adopts the strategy of "white box lead, black, grey box assists" to conduct a complete security test on the project in the way closest to the real attack.

The test method information:

Test method	Description
Black box testing	Conduct security tests from an attacker's perspective externally.
Grey box testing	Conduct security testing on code modules through the scripting tool, observing the internal running status, mining weaknesses.
White box testing	Based on the open source code, non-open source code, to detect whether there are vulnerabilities in programs such as nodes, SDK, etc.

The vulnerability severity level information:

Level	Description
Critical	Critical severity vulnerabilities will have a significant impact on the security of the DeFi project, and it is strongly recommended to fix the critical vulnerabilities.
High	High severity vulnerabilities will affect the normal operation of the DeFi project. It is strongly recommended to fix high-risk vulnerabilities.
Medium	Medium severity vulnerability will affect the operation of the DeFi project. It is recommended to fix medium-risk vulnerabilities.
Low	Low severity vulnerabilities may affect the operation of the DeFi project in certain scenarios. It is suggested that the project team should evaluate and consider whether these vulnerabilities need to be fixed.
Weakness	There are safety risks theoretically, but it is extremely difficult to reproduce in engineering.
Suggestion	There are better practices for coding or architecture.

2 Audit Methodology

The security audit process of SlowMist security team for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using automated analysis tools.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

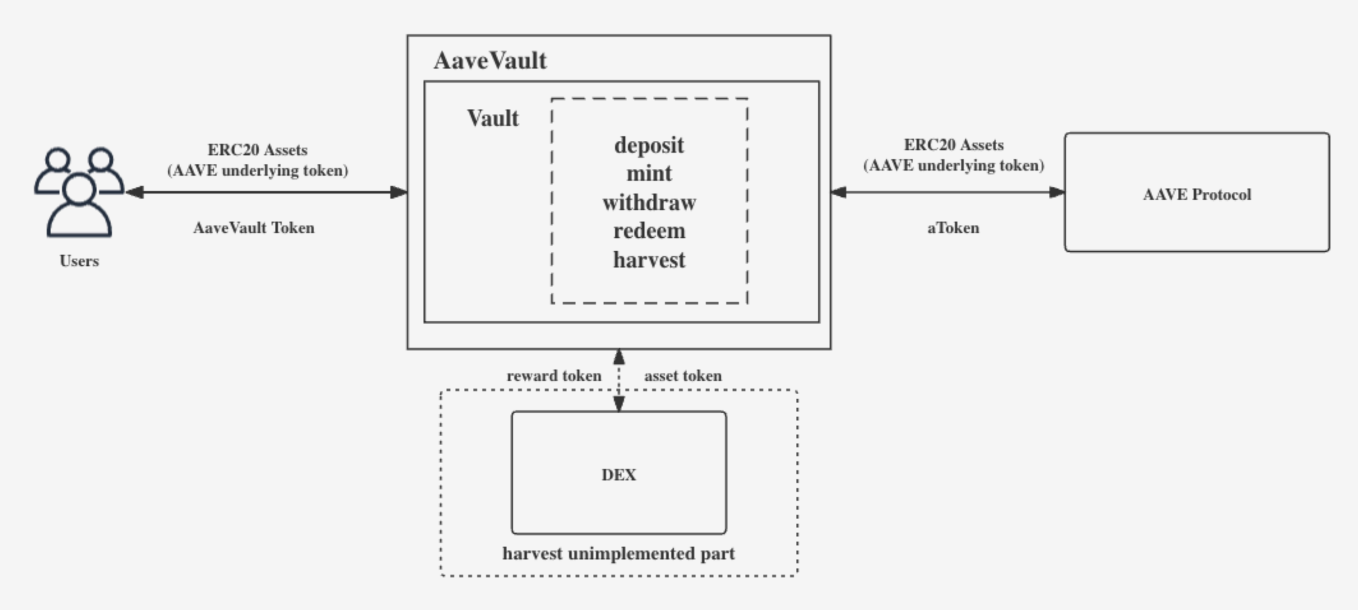
Serial Number	Audit Class	Audit Subclass
1	Overflow Audit	-
2	Reentrancy Attack Audit	-
3	Replay Attack Audit	-
4	Flashloan Attack Audit	-
5	Race Conditions Audit	Reordering Attack Audit
6	Permission Vulnerability Audit	Access Control Audit
		Excessive Authority Audit
7	Security Design Audit	External Module Safe Use Audit
		Compiler Version Security Audit
		Hard-coded Address Security Audit
		Fallback Function Safe Use Audit
		Show Coding Security Audit
		Function Return Value Security Audit
		External Call Function Security Audit

Serial Number	Audit Class	Audit Subclass
7	Security Design Audit	Block data Dependence Security Audit
		tx.origin Authentication Security Audit
8	Denial of Service Audit	-
9	Gas Optimization Audit	-
10	Design Logic Audit	-
11	Variable Coverage Vulnerability Audit	-
12	"False Top-up" Vulnerability Audit	-
13	Scoping and Declarations Audit	-
14	Malicious Event Log Audit	-
15	Arithmetic Accuracy Deviation Audit	-
16	Uninitialized Storage Pointer Audit	-

3 Project Overview

3.1 Project Introduction

These are the Lazyotter Vault and AaveVault contract. The Vault contract is an ERC4626 like contract, users can deposit specific ERC20 tokens through the contract, receiving corresponding shares representing their assets in the Vault. They can withdraw their assets or redeem their shares at any time. Particularly in the AaveVault, users' assets are not only stored in the Vault but are also used to earn returns in the Aave protocol.



3.2 Vulnerability Information

The following is the status of the vulnerabilities found in this audit:

NO	Title	Category	Level	Status
N1	Risk of interest rate inflation	Design Logic Audit	High	Fixed
N2	Risk of excessive authority	Authority Control Vulnerability Audit	Medium	Acknowledged
N3	Shares not-burn but assets can be withdrawn issue	Design Logic Audit	Suggestion	Fixed
N4	Missing maxWithdraw check	Design Logic Audit	Suggestion	Fixed
N5	TODO function reminding	Others	Suggestion	Acknowledged
N6	mintETH can lock user's native token	Design Logic Audit	High	Fixed

4 Code Overview

4.1 Contracts Description

Audit Version:

https://github.com/lazyotter-finance/lazyotter-contract

commit: d804890cbb392553e8c994ae9833247d6ff2e019

Fixed Version:

https://github.com/lazyotter-finance/lazyotter-contract

commit: 610a444af56ca1a470f3b506d64509240f4a685b

Iterative Audit Fixed:

Audit socpe:

src/helper/ETHVaultHelper.sol

Audit Version:

https://github.com/lazyotter-finance/lazyotter-contract

commit: 75c0c7f9b1e292912f57df43e0c08faaeb14f360

Fixed Version:

https://github.com/lazyotter-finance/lazyotter-contract

commit: 74bf84b151f25fc7875d5fd7c7ef206a153672b1

The main network address of the contract is as follows:

The code was not deployed to the mainnet.

4.2 Visibility Description

The SlowMist Security team analyzed the visibility of major contracts during the audit, the result as follows:

Vault			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ERC20
decimals	Public	-	-
totalAssets	Public	-	-

Vault			
maxDeposit	Public	-	-
maxMint	Public	-	-
maxWithdraw	Public	-	-
maxRedeem	Public	-	-
convertToShares	Public	-	-
convertToAssets	Public	-	-
previewDeposit	Public	-	-
previewMint	Public	-	-
previewWithdraw	Public	-	-
previewRedeem	Public	-	-
deposit	External	Can Modify State	nonReentrant whenNotPaused
mint	External	Can Modify State	nonReentrant whenNotPaused
withdraw	External	Can Modify State	nonReentrant
redeem	External	Can Modify State	nonReentrant
harvest	External	Can Modify State	nonReentrant
harvest	External	Can Modify State	nonReentrant
_harvest	Internal	Can Modify State	-
_harvest	Internal	Can Modify State	-
_deposit	Internal	Can Modify State	-
_withdraw	Internal	Can Modify State	-
pause	External	Can Modify State	onlyOwnerOrKeeper
unpause	External	Can Modify State	onlyOwnerOrKeeper

Vault			
setFeeInfo	Public	Can Modify State	onlyOwner
emergencyWithdraw	External	Can Modify State	onlyOwnerOrKeeper
emergencyWithdraw	External	Can Modify State	onlyOwnerOrKeeper
_emergencyWithdraw	Internal	Can Modify State	-
execute	External	Can Modify State	onlyOwner

AaveVault			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Vault
totalAssets	Public	-	-
_harvest	Internal	Can Modify State	-
_deposit	Internal	Can Modify State	-
_withdraw	Internal	Can Modify State	-

ETHVaultHelper			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
depositETH	External	Payable	-
mintETH	External	Payable	-
withdrawETH	External	Can Modify State	-
redeemETH	External	Can Modify State	-
<Receive Ether>	External	Payable	-
<Fallback>	External	Payable	-

4.3 Vulnerability Summary

[N1] [High] Risk of interest rate inflation

Category: Design Logic Audit

Content

In the Vault contract, when the user performs a supply operation, SToken will mint a deposit certificate for the user. The minted amount is calculated by the `(assets * _totalSupply) / totalAssets()`, and the `totalAssets()` parameter is the total asset in the vault. Unfortunately, any user can increase the value of the `totalAssets()` parameter by transferring the asset tokens to the vault. This will lead to the risk of interest rate inflation. Malicious users can pre-deposit and manipulate the value of totalAssets() so that other users cannot get the expected share token(vault token) when depositing, and finally cause the user's funds to be stolen.

Scenario:

Bob finds out that Alice is making a deposit (e.g.Pre-condition: no one deposit before or the assets in the pool have all been previously extracted via the withdraw function.)

Now, Alice wants to deposit 1 (1 * 1e18 wei) WETH(assert token), and the tx is spied on by the attacker(Bob).

Scenario Example		
	totalSupply	totalAssets
original state	0	0
(after) Step 1	1	1
(after) Step 2	1	1e18 + 1
(after) Step 3	1	1e18 + 1

1.Bob front-runs alice and deposits 1 wei WETH(asset) and gets 1 share: since _totalSupply is 0, shares = amount = 1.

2.Bob also transfers 1 * 1e18 wei WETH, making the WETH(asset) totalAssets of the vault become 1e18 + 1 wei.

3.Alice deposits 1e18 wei WETH. However, Alice gets 0 shares: 1e18 * 1 (totalSupply) / (1e18 + 1) = 1e18 / (1e18 + 1) = 0. Since Alice gets 0 shares, totalSupply remains at 1.

4.Bob still has the 1 only share ever minted and thus the withdrawal of that 1 share takes away everything in the pool, including Alice's 1e18 wei WETH(asset).

Reference:

https://ethereum-magicians.org/t/address-eip-4626-inflation-attacks-with-virtual-shares-and-assets/12677

Code location:

vaults/Vault.sol#95-101,127-137

```
    function convertToShares(uint256 assets) public view returns (uint256) {
        uint256 _totalSupply = totalSupply();
        if (_totalSupply == 0) {
            return assets;
        }
        return (assets * _totalSupply) / totalAssets();
    }

    function previewDeposit(uint256 assets) public view returns (uint256) {
        return convertToShares(assets);
    }

    function deposit(uint256 assets, address receiver) external nonReentrant
whenNotPaused returns (uint256) {
        uint256 shares = previewDeposit(assets);
        require(shares > 0, "ZERO_SHARES");

        _mint(receiver, shares);
       ...
    }
```

Solution

It is recommended to send a certain amount of shares to the blackhole address when the protocol accepts deposits for the first time. Or use the method of enlarging the decimal for relief.

Status

Fixed

[N2] [Medium] Risk of excessive authority

Category: Authority Control Vulnerability Audit

Content

In the Vault contract, the owner can change the FeeInfo through the setFeeInfo functions, and the change of the FeeInfo can affect the amount of assets the user withdraws and the amount of fees charged.

Code location:

vaults/Vault.sol#264-276

```
    function setFeeInfo(FeeInfo memory _feeInfo) public onlyOwner {
        require(_feeInfo.recipients.length == _feeInfo.recipientWeights.length,
"length error");
        require(_feeInfo.withdrawalFeeRate <= MAX_FEE_RATE, "withdrawalFeeRate
error");
        require(_feeInfo.harvestFeeRate <= MAX_FEE_RATE, "harvestFeeRate error");
        feeInfo = FeeInfo({
            recipients: _feeInfo.recipients,
            recipientWeights: _feeInfo.recipientWeights,
            harvesterWeight: _feeInfo.harvesterWeight,
            harvestFeeRate: _feeInfo.harvestFeeRate,
            withdrawalFeeRate: _feeInfo.withdrawalFeeRate
        });
        totalRecipientsWeight = _feeInfo.recipientWeights.sum();
    }
```

Solution

In the short term, transferring owner ownership to multisig contracts is an effective solution to avoid single-point risk. But in the long run, it is a more reasonable solution to implement a privilege separation strategy and set up multiple privileged roles to manage each privileged function separately. And the authority involving user funds should be managed by the community, and the authority involving emergency contract suspension can be managed by the EOA address. This ensures both a quick response to threats and the safety of user funds.

Status

Acknowledged

[N3] [Suggestion] Shares not-burn but assets can be withdrawn issue

Category: Design Logic Audit

Content

In the Vault contract, users can withdraw assets through the withdraw function. Users need to burn shares when withdrawing assets. The calculation method is `(assets * _totalSupply) / totalAssets()`. It should be noted that due to the accumulation of interest rates, totalAssets will grow slowly when totalSupply remains unchanged, which will make the user's shares more valuable. However, since Solidity cannot perform decimal calculations, when totalAssets is greater than totalSupply, the result of the division operation will be 0. So malicious users can use this method to ensure that assets * _totalSupply is less than totalAssets to extract assets for free. However, due to the existence of the gas fee, the attacker will have no profit.

Code location:

vaults/Vault.sol#95-101, 119-121, 150-175

```
    function convertToShares(uint256 assets) public view returns (uint256) {
        uint256 _totalSupply = totalSupply();
        if (_totalSupply == 0) {
            return assets;
        }
        return (assets * _totalSupply) / totalAssets();
    }
    function previewWithdraw(uint256 assets) public view returns (uint256) {
        return convertToShares(assets);
    }

    function withdraw(uint256 assets, address receiver, address owner) external
  nonReentrant returns (uint256) {
        ...
        uint256 shares = previewWithdraw(assets);
        if (msg.sender != owner) {
            _spendAllowance(owner, msg.sender, shares);
        }
        _burn(owner, shares);
        _withdraw(owner, assets);
        ...
    }
```

Solution

It is recommended to check that the shares are greater than 0 when the user withdraws.

Status

Fixed

[N4] [Suggestion] Missing maxWithdraw check

Category: Design Logic Audit

Content

There is a maxWithdraw function in the Vault contract, which is used to limit the amount of a user's single withdrawal. But it is not used, ignoring the maxWithdraw check should also be performed in the withdraw function whether the ms.sender or the owner has enough share to burn.

Code location:

vaults/Vault.sol# 150-175

```
    function withdraw(uint256 assets, address receiver, address owner) external
nonReentrant returns (uint256) {
        address[] memory recipients = feeInfo.recipients;
        uint256[] memory recipientWeights = feeInfo.recipientWeights;
        uint256 shares = previewWithdraw(assets);
        if (msg.sender != owner) {
            _spendAllowance(owner, msg.sender, shares);
        }

        _burn(owner, shares);
        _withdraw(owner, assets);
        ...
        return shares;
    }
```

Solution

It is recommended to perform a maxWithdraw check on the amount withdrawn by the user in the withdraw function.

Status

Fixed

[N5] [Suggestion] TODO function reminding

Category: Others

Content

In the AaveVault contract, the _harvest function has an unimplemented loop.

Code location:

vaults/AaveVault.sol#78-82

```
        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]);
        }
```

Solution

It is recommended to confirm whether the implementation of these functions meets the requirements.

Status

Acknowledged

[N6] [High] mintETH can lock user's native token

Category: Design Logic Audit

Content

In the ETHVaultHelper contract, users can call the `mintETH` function to deposit the ETH native token to the Vault and it will firstly warp the ETH to the WETH tokens. Then call the `mint` function from the Vault contract. The mint function is to specify specific shares to mint, and the `assets` that need to be provided are calculated based on the specific shares that need to be minted. When the `msg.vaule` carried by the user exceeds the calculated asstes amount, the excess ETH will be converted into WETH and stored in the current contract and cannot be withdrawn. Later users can use this excess WETH stored in the contract to redeem minting shares for themselves. Thus causing losses to previous users.

Code location:

https://github.com/lazyotter-finance/lazyotter-

contract/blob/75c0c7f9b1e292912f57df43e0c08faaeb14f360/src/helper/ETHVaultHelper.sol#L25-29

```
    function mintETH(address vault, uint256 shares, address receiver) external
payable {
        WETH.deposit{value: msg.value}();
        WETH.approve(vault, msg.value);
        IVault(vault).mint(shares, receiver);
    }
```

Solution

It is recommended to check whether the amount of `msg.vaule` is the same as the `assets` calculated after the mint function of the Vault contract specifies the minted share.

Status

Fixed

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002311090001	SlowMist Security Team	2023.11.07 - 2023.11.09	Medium Risk

Summary conclusion: The SlowMist security team uses a manual and SlowMist team's analysis tool to audit the project, during the audit work we found 2 high risks, 1 medium risk, and 3 suggestions. And 2 high risks and 2 suggestions were confirmed and being fixed; All other findings were acknowledged. The code was not deployed to the mainnet.

6 Statement

SlowMist issues this report with reference to the facts that have occurred or existed before the issuance of this report, and only assumes corresponding responsibility based on these.

For the facts that occurred or existed after the issuance, SlowMist is not able to judge the security status of this project, and is not responsible for them. The security audit analysis and other contents of this report are based on the documents and materials provided to SlowMist by the information provider till the date of the insurance report (referred to as "provided information"). SlowMist assumes: The information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed, or inconsistent with the actual situation, the SlowMist shall not be liable for any loss or adverse effect resulting therefrom. SlowMist only conducts the agreed security audit on the security situation of the project and issues this report. SlowMist is not responsible for the background and other conditions of the project.

Official Website

www.slowmist.com

E-mail

team@slowmist.com

Twitter

@SlowMist_Team

Github

https://github.com/slowmist