Smart Contract
Security Audit Report

Table Of Contents

1 Executive Summary

On 2024.06.24, the SlowMist security team received the Morph Labs team's security audit application for Morphl2 - LRTDepositV1, 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

This is the LRT Deposit part of the Morph L2 protocol. The audit only includes the LRTDepositV1 module.

LRTDepositV1 allows users to deposit ETH or LST tokens as well as other ERC20 tokens supported by the protocol, and perform cross-chain operations on their own deposits.

3.2 Vulnerability Information

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

NO	Title	Category	Level	Status
N1	Potential risks of one-time maximum approval	Others	Suggestion	Fixed
N2	Missing event records	Others	Suggestion	Fixed
N3	Token compatibility issues	Design Logic Audit	Medium	Fixed
N4	Potential risks of open calls	Design Logic Audit	Critical	Fixed
N5	Optimizable token transfer method	Others	Suggestion	Fixed
N6	The value parameter is redundant in the dropMessage operation	Design Logic Audit	Suggestion	Fixed
N7	Optimizable gasFee check	Others	Suggestion	Fixed

4 Code Overview

4.1 Contracts Description

Audit Version:

https://github.com/morph-l2/LRT-Deposit/blob/main/contracts/LRTDepositV1.sol

commit: e2e2651c3e56dfa47b864a8c13867a45dd9b1e14

Fixed Version:

https://github.com/morph-l2/LRT-Deposit/blob/main/contracts/LRTDepositV1.sol

commit: 2d190933849e74cce98da7475c8bfd6729ed17ad

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:

LRTDeposit			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
initialize	External	Can Modify State	initializer
pause	External	Can Modify State	onlyOwner
unpause	External	Can Modify State	onlyOwner
setTokenWhitelist	External	Can Modify State	onlyOwner
setTokenCap	External	Can Modify State	onlyOwner
setSelectorWhitelist	External	Can Modify State	onlyOwner
changeDepositStatus	Public	Can Modify State	onlyOwner
depositETH	Public	Payable	nonReentrant whenNotPaused whenDepositEnabled
depositEETH	Public	Can Modify State	nonReentrant whenNotPaused whenDepositEnabled
depositWEETH	Public	Can Modify State	nonReentrant whenNotPaused whenDepositEnabled
depositSTETH	Public	Can Modify State	nonReentrant whenNotPaused whenDepositEnabled
depositUSDCAndConvertToUSDA	Public	Can Modify State	nonReentrant whenNotPaused whenDepositEnabled
depositETHAndConvertToStone	Public	Payable	nonReentrant whenNotPaused whenDepositEnabled
depositOtherERC20	Public	Can Modify State	nonReentrant whenNotPaused whenDepositEnabled
depositETHandConvertToERC20	Public	Payable	nonReentrant whenNotPaused whenDepositEnabled
depositERC20andConvertToERC20	Public	Can Modify State	nonReentrant whenNotPaused whenDepositEnabled

LRTDeposit			
_depositETHandConvertToEETH	Internal	Can Modify State	-
_isDepositAvailable	Internal	-	-
_recordDeposit	Internal	Can Modify State	-
changeWithdrawStatus	Public	Can Modify State	onlyOwner
withdraw	Public	Can Modify State	nonReentrant whenNotPaused whenWithdrawEnabled
getL1MessageQueueWithGasPriceOracle	Public	-	-
getL1CustomERC20Gateway	Public	-	-
changeBridgeStatus	Public	Can Modify State	onlyOwner
_setMainnetBridge	Internal	Can Modify State	-
_setL1MessageQueueWithGasPriceOracle	Internal	Can Modify State	-
bridge	Public	Payable	nonReentrant whenNotPaused whenBridgeEnabled
_bridgeCheck	Internal	-	-
dropMessage	External	Can Modify State	nonReentrant whenNotPaused whenBridgeEnabled
setL1CrossDomainMessenger	Public	Can Modify State	onlyOwner
replayMessageToMessenger	Public	Payable	whenNotPaused whenBridgeEnabled

4.3 Vulnerability Summary

[N1] [Suggestion] Potential risks of one-time maximum approval

Category: Others

Content

In the LRTDeposit contract, the protocol approves the corresponding tokens to weETHAddress, etherfiVampireAddress, and angleTransmuterAddress with the maximum allowance during initialization, in order to make subsequent deposits to these addresses. It is important to note that the contract does not have additional logic to continue approving these addresses. If the initialized allowances are exhausted in the future, some functions of the contract will become unusable.

Code location: contracts/LRTDepositV1.sol#L131-L142

```
function initialize() external initializer {
    ...
    bool success = eETH.approve(weETHAddress, type(uint256).max);
    require(success, "eETH.approve failed");
    bool success2 = stETH.approve(etherfiVampireAddress, type(uint256).max);
    require(success2, "stETH.approve failed");
    bool usdc_success = usdc.approve(
        angleTransmuterAddress,
        type(uint256).max
    );
    require(usdc_success, "approve angleTransmuterAddress failed");
}
```

Solution

It is recommended to add a new function to re-approve the above addresses or to approve these addresses based on the deposit amount when users make deposits.

Status

Fixed; Fixed in commit 57d45d7e09b5743becf257e52627a4e9300186be.

[N2] [Suggestion] Missing event records

Category: Others

Content

In the LRTDeposit contract, the owner role can modify the isTokenAllowed, erc20Cap, isSelectorAllowed, changeWithdrawStatus, and isDepositEnabled parameters through the setTokenWhitelist, setTokenCap, setSelectorWhitelist, isWithdrawEnabled, and changeDepositStatus functions respectively, but no events are emitted to record these changes.

Code location: contracts/LRTDepositV1.sol#L160-L181

```
    function setTokenWhitelist(
        address token,
        bool isAllowed
    ) external onlyOwner {
        isTokenAllowed[token] = isAllowed;
    }

    function setTokenCap(address token, uint256 cap) external onlyOwner {
        erc20Cap[token] = cap;
    }

    function setSelectorWhitelist(bytes4 functionSelector, bool isAllowed) external
onlyOwner {
        isSelectorAllowed[functionSelector] = isAllowed;
    }

    function changeDepositStatus(bool _isDepositEnabled) public onlyOwner {
        isDepositEnabled = _isDepositEnabled;
    }

    function changeWithdrawStatus(bool _isWithdrawEnabled) public onlyOwner {
        isWithdrawEnabled = _isWithdrawEnabled;
    }
```

Solution

It is recommended to emit events when modifying sensitive contract parameters to facilitate future self-inspection or community audits.

Status

Fixed; Fixed in commit 5152be46489b99f926253d9ff8588eb53c1df686.

[N3] [Medium] Token compatibility issues

Category: Design Logic Audit

Content

In the depositOtherERC20 function of the LRTDeposit contract, users are allowed to deposit ERC20 tokens that are permitted by the protocol. The contract first records the token amount passed in by the user through the `_recordDeposit` function, and then performs the transfer operation using transferFrom. It is important to note that if the protocol supports deflationary tokens, the deposit amount received by the contract will be less than the token amount passed in by the user. This will cause the contract to incorrectly record the actual deposit amount, ultimately leading to losses for the protocol.

Similarly, the depositERC20andConvertToERC20 function may also have this risk. And, when transferring ERC20 tokens, some token implementations do not comply with the EIP20 standard (e.g., USDT), which may cause their return values to always be false. As a result, the protocol will not be able to properly integrate these tokens.

Code location: contracts/LRTDepositV1.sol#L365-L367

```
function depositOtherERC20(
    address token,
    uint256 amount
) public nonReentrant whenNotPaused whenDepositEnabled returns (uint256) {
    require(amount != 0, "amount can not be zero");
    require(
        _isDepositAvailable(token, amount),
        "input amount exceeds token CAP"
    );
    _recordDeposit(token, amount, token, amount);

    bool success = IERC20(token).transferFrom(
        msg.sender,
        address(this),
        amount
    );
    require(success, "deposit failed");

    return amount;
}
```

Solution

It is recommended that when supporting other ERC20 tokens, the transfer operation should be performed first, and the difference in the contract balance before and after the transfer should be calculated as the actual deposit amount of the user.

Status

Fixed; Fixed in commit 5152be46489b99f926253d9ff8588eb53c1df686.

[N4] [Critical] Potential risks of open calls

Category: Design Logic Audit

Content

In the LRTDeposit contract, to support other protocols in the future, the depositETHandConvertToERC20 and depositERC20andConvertToERC20 functions allow users to pass in arbitrary target addresses and execute user-specified data. The LRTDeposit contract only restricts the function that users can call. Unfortunately, this may not effectively mitigate the risk of arbitrary external calls. Although the contract limits the functions that can be called, many contracts have functions with the same name but different logic, which can pose significant risks to other users' funds or the funds in the contract. In particular, if the contract supports the transferFrom function, then the funds of all users who have approved tokens to this contract will be at risk. In fact, even if the target contracts that can be called are restricted, it may not be possible to mitigate all risks. The protocol should use this approach cautiously for functional expansion.

Code location:

contracts/LRTDepositV1.sol#L390

contracts/LRTDepositV1.sol#L422

```
    function depositETHandConvertToERC20(
        ...
    ) public payable nonReentrant whenNotPaused whenDepositEnabled returns (uint256)
{
        ...
        (bool success, ) = target.call{value: msg.value}(data);
        ...
    }

    function depositERC20andConvertToERC20(
        ...
    ) public nonReentrant whenNotPaused whenDepositEnabled returns (uint256) {
        ...
        (bool _success, ) = target.call(data);
        ...
    }
```

Solution

It is recommended to use a modular approach to support other protocols. The contract can load other modules through delegatecall to support other protocols, rather than using the current open-ended calling approach.

Status

Fixed; Fixed in commit 2d190933849e74cce98da7475c8bfd6729ed17ad. The project team has enhanced the checks on the target address and will conduct strict reviews of the target and 4bytes when adding to the whitelist. And the project team stated that the protocol will not support deflationary tokens.

[N5] [Suggestion] Optimizable token transfer method

Category: Others

Content

In the LRTDeposit contract, the withdraw function allows users to withdraw the deposited tokens. The contract transfers tokens to the user using the transfer function but does not check the return value of the function call. If the contract supports tokens like ZRX, this may lead to some transfer failure risks.

The same is true for the dropMessage function.

Code location:

contracts/LRTDepositV1.sol#L501

contracts/LRTDepositV1.sol#L663

```
function withdraw(
    address token,
    uint256 amount
) public nonReentrant whenNotPaused whenWithdrawEnabled returns (uint256) {
    require(amount > 0, "can not withdraw 0 amount");
    erc20Balance[token][msg.sender] -= amount; // will underflow if < 0
    currentERC20Total[token] -= amount;

    bool success = IERC20(token).transfer(msg.sender, amount);
    require(success, "withdraw failed");

    emit Withdraw(msg.sender, token, amount);
    return amount;
}

function dropMessage(
    ...
) external nonReentrant whenNotPaused whenBridgeEnabled{
    ...
    // refund token to user
    IERC20(_token).transfer(_receiver, _amount);
```

```
        emit RefundERC20(_token, _receiver, _amount);
    }
```

Solution

It is recommended to use OpenZeppelin's SafeERC20 library for token transfer operations.

Status

Fixed; Fixed in commit 5152be46489b99f926253d9ff8588eb53c1df686.

[N6] [Suggestion] The value parameter is redundant in the dropMessage operation

Category: Design Logic Audit

Content

In the LRTDeposit contract, the dropMessage function is used to retrieve funds from transactions that were skipped during cross-chain operations. It is triggered through the dropMessage function of the l1CrossDomainMessenger contract. It is important to note that all cross-chain funds in this contract are ERC20 tokens, so the cross-chain value is always 0. However, the dropMessage function still allows users to pass in the `_value` parameter, which is redundant.

Code location: contracts/LRTDepositV1.sol#L636

```
    function dropMessage(
        address _from,
        address _to,
        uint256 _value,
        uint256 _messageNonce,
        bytes calldata _message
    ) external nonReentrant whenNotPaused whenBridgeEnabled{
        ...
        _l1CrossDomainMessenger.dropMessage(
            _from,
            _to,
            _value,
            _messageNonce,
            _message
        );
        ...
    }
```

Solution

It is recommended to remove the redundant `_value` parameter and directly pass a value of 0 when making the dropMessage call.

Status

Fixed; Fixed in commit 5152be46489b99f926253d9ff8588eb53c1df686.

[N7] [Suggestion] Optimizable gasFee check

Category: Others

Content

In the LRTDeposit contract, users can perform batch cross-chain operations on the deposited tokens using the bridge function, and users need to pay the native token fees required for cross-chain transactions. However, the bridge function does not first check whether the user's `msg.value` is sufficient to pay the fees. Instead, it relies on indirect checks during the bridge and refund operations. If the fees provided by the user are low, additional gas may be required to determine that the fees paid are insufficient.

Code location: contracts/LRTDepositV1.sol#L605

```
function bridge(
    address[] calldata tokens,
    uint256 gasLimit
) public payable nonReentrant whenNotPaused whenBridgeEnabled {
    ...
    uint256 length = tokens.length;
    uint256 gasFee = gasLimit * l1MessageQueueWithGasPriceOracle.l2BaseFee();
    for (uint256 i; i < length; ++i) {
        ...
        l1CustomERC20Gateway.depositERC20{value: gasFee}(
            tokens[i],
            msg.sender,
            amounts[i],
            gasLimit
        );

        emit Bridge(
            msg.sender,
            tokens[i],
            amounts[i],
            address(l1CustomERC20Gateway)
        );
```

```
        }

        // refund extra fee to msg.sender
        uint256 _refund = msg.value - (gasFee * length);
        if (_refund > 0) {
            (bool _success, ) = msg.sender.call{value: _refund}("");
            require(_success, "LRTDeposit: Failed to refund the fee");
        }

    }
```

Solution

It is recommended to check whether the user's `msg.value` is sufficient to pay the total cross-chain gas fees before performing the cross-chain operation, in order to improve the user experience.

Status

Fixed; Fixed in commit 5152be46489b99f926253d9ff8588eb53c1df686.

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002406260002	SlowMist Security Team	2024.06.24 - 2024.06.26	Passed

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 1 critical risk, 1 medium risk, and 5 suggestions. All the findings were fixed. 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