MorpheusAI - Findings Report

Table of contents

Contest Summary

Sponsor: MorpheusAI

Dates: Jan 30th, 2024 - Feb 3rd, 2024

See more contest details here

Results Summary

Number of findings:

- High: 1
- Medium: 0
- Low: 2

High Risk Findings

H-01. MOR can be minted to incorrect address on Arbitrum

Relevant GitHub Links

https://github.com/Cyfrin/2024-01-Morpheus/blob/07c900d22073911afa23b7fa69a4249ab5b713c8/contracts/L2MessageReceiver.sol#L105

Summary

Currently the protocol assumes that the MOR tokens will be minted to the correct user address. This is not true for all cases:

1. If the user is a Multisig (Contract Account), the same address on the destination chain (Arbitrum) might not be controlled/owned by the same user.
2. If the user is a contract/dapp, the same address on the destination chain is not controlled/owned by the same user.

Impact

MOR tokens would be minted to an address not owned by the user, leading to loss of funds.

Vulnerability Details

Here is the whole process:

Let us understand how MOR tokens would be lost

1. The user (i.e. using multisig/contract/dapp) calls the claim() function below by passing in the `poolId_` and `user_` (user's address).

- Consider the function works as expected from Lines 156-172 (this is independent of the issue)
- On Line 176, the function sendMintMessage() is called on the L1Sender.sol contract with user_, the pending rewards (amount of MOR that will be minted) and the refund address (user_ is the msg.sender).

```
File: Distribution.sol
155:     function claim(uint256 poolId_, address user_) external payable poolExists(poolId_) {
156:         Pool storage pool = pools[poolId_];
157:         PoolData storage poolData = poolsData[poolId_];
158:         UserData storage userData = usersData[user_][poolId_];
159:
160:         require(block.timestamp > pool.payoutStart + pool.claimLockPeriod, "DS: pool claim is locked");
161:
162:         uint256 currentPoolRate_ = _getCurrentPoolRate(poolId_);
163:         uint256 pendingRewards_ = _getCurrentUserReward(currentPoolRate_, userData);
164:         require(pendingRewards_ > 0, "DS: nothing to claim");
165:
166:         // Update pool data
167:         poolData.lastUpdate = uint128(block.timestamp);
168:         poolData.rate = currentPoolRate_;
169:
170:         // Update user data
171:         userData.rate = currentPoolRate_;
172:         userData.pendingRewards = 0;
173:
174:         // Transfer rewards
175:
```

```
176:         L1Sender(l1Sender).sendMintMessage{value: msg.value}(user_,
pendingRewards_, _msgSender());
177:
178:         emit UserClaimed(poolId_, user_, pendingRewards_);
179:     }
```

2. In function sendMintMessage(), the following occurs:

- On Line 129, the user_ and the pending rewards amount is encoded into bytes memory payload variable.
- On Line 132, the send() function is called on the LayerZero endpoint contract

```
124:    function sendMintMessage(address user_, uint256 amount_, address
refundTo_) external payable onlyDistribution {
125:        RewardTokenConfig storage config = rewardTokenConfig;
126:
127:
128:        bytes memory receiverAndSenderAddresses_ =
abi.encodePacked(config.receiver, address(this));
129:        bytes memory payload_ = abi.encode(user_, amount_);
130:
131:
132:        ILayerZeroEndpoint(config.gateway).send{value: msg.value}(
133:            config.receiverChainId, // communicator LayerZero chainId
134:            receiverAndSenderAddresses_, // send to this address to
the communicator
135:            payload_, // bytes payload
136:            payable(refundTo_), // refund address
137:            address(0x0), // future parameter
138:            bytes("") // adapterParams (see "Advanced Features")
139:        );
140:    }
```

3. After the send() function call, LayerZero relays the cross-chain transaction to call the receivePayload() function on the Endpoint contract (Arbitrum), which ultimately calls the lzReceive() function on the L2MessageReceiver.sol contract.

- On Line 40, function _blockingLzReceive() is called internally.

```
File: L2MessageReceiver.sol
32:     function lzReceive(
33:         uint16 senderChainId_,
34:         bytes memory senderAndReceiverAddresses_,
35:         uint64 nonce_,
36:         bytes memory payload_
37:     ) external {
38:         require(_msgSender() == config.gateway, "L2MR: invalid
gateway");
39:
```

```
        _blockingLzReceive(senderChainId_, senderAndReceiverAddresses_, nonce_, payload_);
    }
```

4. Function _blockingLzReceive() calls the nonBlockingLzReceive() function here, which ultimately calls the function _nonBlockingLzReceive() here.

5. In function _nonBlockingLzReceive() here, the following occurs:

- Lines 97 to 103, the function checks the source id and sender are correct.
- On Line 105, the payload is decoded to variables user_ and amount_.
- On Line 107, the mint() function is called on the MOR token contract on Arbitrum to mint tokens to the user_. But since the user_ address on the destination chain is not owned by the actual user, the MOR tokens would be sent to someone else or no one in case the address is not being used. This would mean permanent loss of MOR tokens for the user.

```
File: L2MessageReceiver.sol
092:     function _nonblockingLzReceive(
093:         uint16 senderChainId_,
094:         bytes memory senderAndReceiverAddresses_,
095:         bytes memory payload_
096:     ) private {
097:         require(senderChainId_ == config.senderChainId, "L2MR: invalid sender chain ID");
098:
099:         address sender_;
100:         assembly {
101:             sender_ := mload(add(senderAndReceiverAddresses_, 20))
102:         }
103:         require(sender_ == config.sender, "L2MR: invalid sender address");
104:
105:         (address user_, uint256 amount_) = abi.decode(payload_, (address, uint256));
106:
107:         IMOR(rewardToken).mint(user_, amount_);
108:     }
```

Through this we can see how the issue arises.

Tools Used

Manual Review

References

Similar issues were also found in the C4 Maia contest:

1. First issue

2. Second issue

Recommendations

Encode an extra address field into the payload on the source chain. This address field would be the recipient of the MOR tokens on the destination chain.

Additionally, note that the claim() function currently allows anyone to call claim() on behalf of any user since the user_ address is taken in as a parameter (see here). Make sure to remove this user_ parameter and use msg.sender instead so that an attacker cannot provide his own recipient address and call claim() on user's behalf.

Low Risk Findings

L-01. In message blocking can occur due to incorrect non-blocking implementation

Relevant GitHub Links

https://github.com/Cyfrin/2024-01-Morpheus/blob/07c900d22073911afa23b7fa69a4249ab5b713c8/contracts/L2MessageReceiver.sol#L31

Impact

In-message blocking state is possible due to incorrect implementation of non blocking pattern. The issue arises because the lzReceive() does not use a try-catch with an ExcessivelySafeCall library to call _blockingLzReceive() in function lzReceive(). This would cause the message channel to be blocked since the gas sent can be intentionally or unintentionally set to a low value.

These are the following impacts:

1. There would be permanent DOS until the team clears the payload by calling retryPayload() on the endpoint contract on Arbitrum.
2. Any claim() calls during the DOS period would cause permanent loss of rewards for users. These claim() calls can be forced by an attacker since it allows anyone to claim the tokens on behalf of a user.

There are two root causes to this issue:

1. The gas amount supplied can be arbitrary. This is against what LayerZero recommends i.e. using estimateFees() to obtain the gas value to be used.
2. The lzReceive() function does not implement a try-catch and a library like ExcessivelySafeCall to minimize the gas used to call _blockingLzReceive().

Vulnerability Details

Here is the whole process:

Execution path from Distribution.sol to Endpoint:

claim() => sendMintMessage() => send()

Execution path from Endpoint to L2MessageReceiver.sol:

receivePayload() => lzReceive() => _blockingLzReceive()

1. First we'll take a look at the claim() function, which allows the attacker to pass in an arbitrary amount of gas. **(Note: The attacker is making a tx for himself as user_ here)**

- On Line 174, we can see that the msg.value is neither checked nor estimateFees() is used anywhere in the function.

```
File: Distribution.sol
154:     function claim(uint256 poolId_, address user_) external payable
poolExists(poolId_) {
155:         Pool storage pool = pools[poolId_];
156:         PoolData storage poolData = poolsData[poolId_];
157:         UserData storage userData = usersData[user_][poolId_];
158:
159:         require(block.timestamp > pool.payoutStart +
pool.claimLockPeriod, "DS: pool claim is locked");
160:
161:         uint256 currentPoolRate_ = _getCurrentPoolRate(poolId_);
162:         uint256 pendingRewards_ =
_getCurrentUserReward(currentPoolRate_, userData);
163:         require(pendingRewards_ > 0, "DS: nothing to claim");
164:
165:         // Update pool data
166:         poolData.lastUpdate = uint128(block.timestamp);
167:         poolData.rate = currentPoolRate_;
168:
169:         // Update user data
170:         userData.rate = currentPoolRate_;
171:         userData.pendingRewards = 0;
172:
173:         // Transfer rewards
174:         L1Sender(l1Sender).sendMintMessage{value: msg.value}(user_,
pendingRewards_, _msgSender());
175:
176:         emit UserClaimed(poolId_, user_, pendingRewards_);
177:     }
```

2. The function sendMintMessage() calls the send() function on the Endpoint contract on Ethereum with the gas provided for the destination call.

```
File: L1Sender.sol
125:     function sendMintMessage(address user_, uint256 amount_, address
refundTo_) external payable onlyDistribution {
126:         RewardTokenConfig storage config = rewardTokenConfig;
127:
```

```
        bytes memory receiverAndSenderAddresses_ = abi.encodePacked(config.receiver, address(this));
        bytes memory payload_ = abi.encode(user_, amount_);


        ILayerZeroEndpoint(config.gateway).send{value: msg.value}(
            config.receiverChainId, // communicator LayerZero chainId
            receiverAndSenderAddresses_, // send to this address to the communicator
            payload_, // bytes payload
            payable(refundTo_), // refund address
            address(0x0), // future parameter
            bytes("") // adapterParams (see "Advanced Features")
        );
    }
```

3. LayerZero relays the call to Arbitrum and calls the receivePayload() function on the [Endpoint contract on Arbitrum](). The receivePayload() calls the lzReceive() function on the L2MessageReceiver.sol contract. The following occurs in lzReceive():

- On Line 38, the function checks if the msg.sender is the endpoint contract. We assume this is true.

```
File: L2MessageReceiver.sol
32:     function lzReceive(
33:         uint16 senderChainId_,
34:         bytes memory senderAndReceiverAddresses_,
35:         uint64 nonce_,
36:         bytes memory payload_
37:     ) external {
38:         require(_msgSender() == config.gateway, "L2MR: invalid gateway");
39:
40:         _blockingLzReceive(senderChainId_, senderAndReceiverAddresses_, nonce_, payload_);
41:     }
```

4. On Line 40 above, an internal call is made to the function _blockingLzReceive(). But since the attacker did not send enough gas, the call would revert due to OOG exception, which would then caught by the catch block in the receivePayload() function and be stored in the storedPayload mapping as seen below.

```
      try ILayerZeroReceiver(_dstAddress).lzReceive{gas: _gasLimit}(_srcChainId, _srcAddress, _nonce, _payload) {
            // success, do nothing, end of the message delivery
        } catch (bytes memory reason) {
            // revert nonce if any uncaught errors/exceptions if the ua chooses the blocking mode
            storedPayload[_srcChainId][_srcAddress] = StoredPayload(uint64(_payload.length), _dstAddress, keccak256(_payload));
```

```
            emit PayloadStored(_srcChainId, _srcAddress, _dstAddress,
_nonce, _payload, reason);
        }
```

5. Due to this now, the message channel is blocked since the check below in the receivePayload() function would cause a revert (due to a stored payload existing) whenever a claim() call arrives from Ethereum to Arbitrum.

```
        // block if any message blocking
        StoredPayload storage sp = storedPayload[_srcChainId]
[_srcAddress];
        require(sp.payloadHash == bytes32(0), "LayerZero: in message
blocking");
```

6. This stored payload can only be cleared by either calling retryPayload() manually or forceResumeReceive() from the L2MessageReceiver contract. Since the L2MessageReceiver does not have an implementation to call the forceResumeReceive() function, the only option left is the former one.

```
    function retryPayload(uint16 _srcChainId, bytes calldata _srcAddress,
bytes calldata _payload) external override receiveNonReentrant {
        StoredPayload storage sp = storedPayload[_srcChainId]
[_srcAddress];
        require(sp.payloadHash != bytes32(0), "LayerZero: no stored
payload");
        require(_payload.length == sp.payloadLength && keccak256(_payload)
== sp.payloadHash, "LayerZero: invalid payload");

        address dstAddress = sp.dstAddress;
        // empty the storedPayload
        sp.payloadLength = 0;
        sp.dstAddress = address(0);
        sp.payloadHash = bytes32(0);

        uint64 nonce = inboundNonce[_srcChainId][_srcAddress];

        ILayerZeroReceiver(dstAddress).lzReceive(_srcChainId, _srcAddress,
nonce, _payload);
        emit PayloadCleared(_srcChainId, _srcAddress, nonce, dstAddress);
    }
```

7. The attacker is in a win-win situation now since if retryPayload() is used, he will get his MOR tokens. But if retryPayload() has not been used yet, all claim() function calls arriving from Ethereum to Arbitrum would revert due to the check mentioned in step 5. Since the calls would revert on destination, the state changes on the source chain remain the same since the LayerZero relayer is the one transmitting the calls and has no ability as such to "revert" transactions that were successfully completed on the source chain.

8. Due to this, the pending rewards in the claim() function on the source chain is set to 0 and the rewards are lost.

- An additional point to note over here is that, the claim() function call needs to arrive while the message channel is blocked. Since the claim() function allows anyone to pass in _user and claim for the user, the attacker could intentionally call this claim() function to target addresses that have higher rewards. This would then lead to permanent loss of rewards for those addresses due to the issue mentioned in step 7 above.
- On Line 171, we can see that pending rewards for the user is set to 0.

```
File: Distribution.sol
154:     function claim(uint256 poolId_, address user_) external payable
poolExists(poolId_) {
155:         Pool storage pool = pools[poolId_];
156:         PoolData storage poolData = poolsData[poolId_];
157:         UserData storage userData = usersData[user_][poolId_];
158:
159:         require(block.timestamp > pool.payoutStart +
pool.claimLockPeriod, "DS: pool claim is locked");
160:
161:         uint256 currentPoolRate_ = _getCurrentPoolRate(poolId_);
162:         uint256 pendingRewards_ =
_getCurrentUserReward(currentPoolRate_, userData);
163:         require(pendingRewards_ > 0, "DS: nothing to claim");
164:
165:         // Update pool data
166:         poolData.lastUpdate = uint128(block.timestamp);
167:         poolData.rate = currentPoolRate_;
168:
169:         // Update user data
170:         userData.rate = currentPoolRate_;
171:         userData.pendingRewards = 0;
172:
173:         // Transfer rewards
174:         L1Sender(l1Sender).sendMintMessage{value: msg.value}(user_,
pendingRewards_, _msgSender());
175:
176:         emit UserClaimed(poolId_, user_, pendingRewards_);
177:     }
```

Through this, we can see that an attacker can not only block the message channel but also cause permanent loss of rewards for users by calling the claim() function intentionally.

Tools Used

Manual Review, Similar issue

Recommendations

1. Implement estimateFees() in the claim() function and check if the msg.value provided is enough.

2. Use a try-catch block in lzReceive() with the ExcessivelySafeCall library to minimize the gas used to call _blockingLzReceive().
3. Additionally, only limit the user_ themselves to call the claim() function instead of anyone.

Additional solutions:

1. Implement an access controlled function in L2MessageReceiver.sol to call forceResumeReceive() on the endpoint contract. This would force eject any payload without executing it and clear the blocked channel.
2. Implement a NonBlockingLZApp provided by LayerZero that allows messages to flow regardless of error (which will all be stored on the destination to be dealt with anytime - see here).

L-02. QA Report

Quality Assurance Report

[L-01] PUSH0 opcode is not supported on Arbitrum when using solc v0.8.0 or higher

This would cause deployment errors and transaction reverts on the Arbitrum L2.

```
File: MOR.sol
2: pragma solidity ^0.8.20;
```

[L-02] Use Ownable2Step in MOR contract

Currently the MOR token contract uses Ownable.sol. This would cause issues when ownership is transferred to another address since an incorrect input would lead to permanent loss of the owner role. Due to this, it is advised to use a two step mechanism to avoid this issue.

```
File: MOR.sol
6: import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";
```

[L-03] Missing event emission in setter functions

The setter functions below should include event emissions to ensure any offchain tracking mechanism is alerted of critical changes by the owner role.

```
File: L1Sender.sol
46:     function setDistribution(address distribution_) public onlyOwner {
47:         distribution = distribution_;
48:     }
49:
50:     function setRewardTokenConfig(RewardTokenConfig calldata
newConfig_) public onlyOwner {
```

```
51:         rewardTokenConfig = newConfig_;
52:     }
53:
54:     function setDepositTokenConfig(DepositTokenConfig calldata
newConfig_) public onlyOwner {
55:         require(newConfig_.receiver != address(0), "L1S: invalid
receiver");
56:
57:         DepositTokenConfig storage oldConfig = depositTokenConfig;
58:
59:         _replaceDepositToken(oldConfig.token, newConfig_.token);
60:         _replaceDepositTokenGateway(oldConfig.gateway,
newConfig_.gateway, oldConfig.token, newConfig_.token);
61:
62:         depositTokenConfig = newConfig_;
63:     }
```

[L-04] Pass adapterParams as a configurable bytes parameter instead of using bytes("")

See 7th point in the LayerZero integration list. Hardcoding adapterParams to bytes("") is not recommended since it limits the protocol from future configurability or added features.

```
File: L1Sender.sol
144:         ILayerZeroEndpoint(config.gateway).send{value: msg.value}(
145:             config.receiverChainId, // communicator LayerZero chainId
146:             receiverAndSenderAddresses_, // send to this address to
the communicator
147:             payload_, // bytes payload
148:             payable(refundTo_), // refund address
149:             address(0x0), // future parameter
150:             bytes("") // adapterParams (see "Advanced Features")
151:         );
152:     }
```

[L-05] Do not hardcode zroPaymentAddress to address(0)

On Line 149, we can see that the zroPaymentAddress field is hardcoded to address(0). This makes the protocol devoid of using the future ZRO token (confirmed) as a payment option for the cross-chain fees. See 5th point in the LayerZero integration list

```
File: L1Sender.sol
144:         ILayerZeroEndpoint(config.gateway).send{value: msg.value}(
145:             config.receiverChainId, // communicator LayerZero chainId
146:             receiverAndSenderAddresses_, // send to this address to
the communicator
147:             payload_, // bytes payload
148:             payable(refundTo_), // refund address
```

```
                address(0x0), // future parameter
                bytes("") // adapterParams (see "Advanced Features")
            );
    }
```

[L-06] Consider using mapping over array to avoid DOS

Whenever a pool is created, it is pushed to the pools array. The issue is that if this pools array grows too large, it could cause DOS or excessive gas consumption for other contract trying to retrieve all the pools. Although this does not occur in the current protocol, it is an issue to know about in case external protocols or additional contracts use the TCM model to expand on some functionality.

```
File: Distribution.sol
/**************************************************************************************************/
    function createPool(Pool calldata pool_) public onlyOwner {
        require(pool_.payoutStart > block.timestamp, "DS: invalid payout start value");

        _validatePool(pool_);
        pools.push(pool_);

        emit PoolCreated(pools.length - 1, pool_);
    }
```

[L-07] Use CEI pattern in function retryMessage()

Similar to how the retryPayload() function in the Endpoint contract deletes the stored payload before making a call, consider deleting the failedMessage payload stored before making a call to _nonBlockingLzReceive() on Line 64. This would adhere to the best practice to using the CEI pattern as well as prevent any reentrancy risks that could arise (though there aren't any currently).

```
File: L2MessageReceiver.sol
54:     function retryMessage(
55:         uint16 senderChainId_,
56:         bytes memory senderAndReceiverAddresses_,
57:         uint64 nonce_,
58:         bytes memory payload_
59:     ) external {
60:         bytes32 payloadHash_ = failedMessages[senderChainId_][senderAndReceiverAddresses_][nonce_];
61:         require(payloadHash_ != bytes32(0), "L2MR: no stored message");
62:         require(keccak256(payload_) == payloadHash_, "L2MR: invalid payload");
63:
64:         _nonblockingLzReceive(senderChainId_,
```

```
senderAndReceiverAddresses_, payload_);
65:
66:         delete failedMessages[senderChainId_]
[senderAndReceiverAddresses_][nonce_];
67:
68:         emit RetryMessageSuccess(senderChainId_,
senderAndReceiverAddresses_, nonce_, payload_);
69:     }
```

[L-08] Return value of approve() not checked

The unwrappedToken (StETH) returns a bool in its approve() function. This value is not checked in the current code. Although it returns true, it is best practice to check these values as true.

```
File: L1Sender.sol
78:             IERC20(unwrappedToken_).approve(newToken_,
type(uint256).max);
```

[L-09] Use outboundTransferCustomRefund() instead of outboundTransfer() as recommended by Arbitrum documentation

On Line 116, we can see that function outboundTransfer() is used to make the cross-chain call from Ethereum L1 to Arbitrum L2. The Arbitrum documentation though recommends using outboundTransferCustomRefund() when bridging from Ethereum to Arbitrum. (See first point at the bottom here).

```
File: L1Sender.sol
100:     function sendDepositToken(
101:         uint256 gasLimit_,
102:         uint256 maxFeePerGas_,
103:         uint256 maxSubmissionCost_
104:     ) external payable onlyDistribution returns (bytes memory) {
105:         DepositTokenConfig storage config = depositTokenConfig;
106:
107:         // Get current stETH balance
108:         uint256 amountUnwrappedToken_ =
IERC20(unwrappedDepositToken).balanceOf(address(this));
109:         // Wrap all stETH to wstETH
110:         uint256 amount_ =
IWStETH(config.token).wrap(amountUnwrappedToken_);
111:
112:         bytes memory data_ = abi.encode(maxSubmissionCost_, "");
113:
114:
115:         return
116:             IGatewayRouter(config.gateway).outboundTransfer{value:
msg.value}(
117:                 config.token,
```

```
                config.receiver,
                amount_,
                gasLimit_,
                maxFeePerGas_,
                data_
            );
    }
```

[L-10] Anyone can call claim() function on behalf of any other user

Currently, the claim() function in the Distribution.sol contract takes in user_ as a parameter instead of using msg.sender directly. This allows anyone to claim on behalf of the user_, which might not be what the user wanted. The user_ may have wanted to accumulate more rewards and then finally bridge to mint MOR tokens on Arbitrum. But since an attacker or anyone has already bridged the amount, this would affect the user's strategy. This does not affect the user in any way at first glance since the attacker pays the bridging fees but external applications or strategies would be affected due to this issue.

Solution: Ensure user_ is msg.sender and/or implement an approval mapping which stores approvals the user_ has given to certain external applications/contracts to call the claim() function on behalf.

```
File: Distribution.sol
    function claim(uint256 poolId_, address user_) external payable poolExists(poolId_) {
        Pool storage pool = pools[poolId_];
        PoolData storage poolData = poolsData[poolId_];
        UserData storage userData = usersData[user_][poolId_];

        require(block.timestamp > pool.payoutStart + pool.claimLockPeriod, "DS: pool claim is locked");

        uint256 currentPoolRate_ = _getCurrentPoolRate(poolId_);
        uint256 pendingRewards_ = _getCurrentUserReward(currentPoolRate_, userData);
        require(pendingRewards_ > 0, "DS: nothing to claim");

        // Update pool data
        poolData.lastUpdate = uint128(block.timestamp);
        poolData.rate = currentPoolRate_;

        // Update user data
        userData.rate = currentPoolRate_;
        userData.pendingRewards = 0;

        // Transfer rewards
        L1Sender(l1Sender).sendMintMessage{value: msg.value}(user_, pendingRewards_, _msgSender());

        emit UserClaimed(poolId_, user_, pendingRewards_);
    }
```