Morpheus Smart Contracts Audit Report

Version 2.0

Audited by:

HollaDieWaldfee

alexxander

January 26, 2024

Contents

1 Introduction

1.1 About Renascence

Renascence Labs was established by a team of experts including HollaDieWaldfee, MiloTruck, alexxander and bytes032.

Our founders have a distinguished history of achieving top honors in competitive audit contests, enhancing the security of leading protocols such as Reserve Protocol, Arbitrum, MaiaDAO, Chainlink, Dodo, Lens Protocol, Wenwin, PartyDAO, Lukso, Perennial Finance, Mute and Taurus.

We strive to deliver tailored solutions by thoroughly understanding each client's unique challenges and requirements. Our approach goes beyond addressing immediate security concerns; we are dedicated to fostering the enduring success and growth of our partners.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an 'as-is' and 'as-available' basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

1.3.1 Impact

- High - Funds are **directly** at risk, or a **severe** disruption of the protocol's core functionality
- Medium - Funds are **indirectly** at risk, or **some** disruption of the protocol's functionality
- Low - Funds are **not** at risk

1.3.2 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

2 Executive Summary

2.1 About Morpheus Smart Contracts

MorpheusAI employs a set of Smart Contracts designed to facilitate the distribution of the MOR reward token among contributors involved in various phases of the project's development. The reward distribution is orchestrated through a Linear Decrease model, where claimed rewards are seamlessly bridged to the Arbitrum network. In addition to this, the generated yield from Liquidity Providers is also bridged to fund a strategic Uniswap V3 position. Core features of the protocol include a library contract that allows for adaptable reward emission configurations, message-passing contracts that integrate with Layer Zero for MOR token distribution, and contracts that integrate with the Lido Bridge Gateway for a streamlined yield transfer to Arbitrum.

2.2 Overview

Project	Morpheus Smart Contracts
Repository	SmartContracts
Commit Hash	b2115164af98...
Mitigation Hash	5af591e6ff2d...
Date	13 January 2024 - 20 January 2024

2.3 Issues Found

Severity	**Count**
High Risk	1
Medium Risk	0
Low Risk	3
Informational	4
Total Issues	**8**

3 Findings Summary

ID	Description	Status
H-1	`Distribution.getCurrentPoolRate()` function will revert after `block.timestamp` reaches the maximum end time causing unrecoverable DoS	Resolved
L-1	Redundant nonce tracking in `L2MessageReceiver.sol`	Resolved
L-2	The alias of `L1Sender.sol` on Arbitrum will hold the ETH received as a gas refund from bridging tokens	Resolved
L-3	Negative rebasing of stETH deposits in `Distribution.sol` can cause bank run and loss to new depositors	Acknowledged
I-1	`Distribution.editPool()` contains a needless check	Resolved
I-2	Rewards can be griefed when a distribution configuration emits more tokens than the reward's token cap	Resolved
I-3	`UUPSUpgradeable` contracts should implement a constructor with a call to `Initializable.disableInitializers()`	Resolved
I-4	Misleading comments in `IDIstribution.sol` and `IL2MessageReceiver.sol`	Resolved

4 Findings

4.1 High Risk

[H-1] `Distribution.getCurrentPoolRate()` function will revert after `block.timestamp` reaches the maximum end time causing unrecoverable DoS

Context: LinearDistributionIntervalDecrease.sol, Distribution.sol

Impact `Distribution._getCurrentPoolRate()` will revert when `block.timestamp` reaches the maximum end time. `Distribution.claim()`, `Distribution.withdraw()`, `Distribution.stake()`, and `Distribution.editPool()` will always revert, thus, denying users from withdrawing their staked tokens or claiming pending rewards.

Description: For a given period bound by `startTime_` and `endTime_` and an `interval_` `LinearDistributionIntervalDecrease.sol.getPeriodReward()` checks If that period is contained within a single interval or spread over multiple ones. In the case where the provided period is spread over several intervals, the reward calculations use the `LinearDistributionIntervalDecrease._calculatePartPeriodReward()` and `LinearDistributionIntervalDecrease._calculateFullPeriodReward()` functions.

```
# LinearDistributionIntervalDecrease.sol

// Calculate interval that less then 'interval_' range
        uint256 timePassedBefore_ = startTime_ - payoutStart_;
>       if ((timePassedBefore_ / interval_) == ((endTime_ - payoutStart_) /
interval_)) {
            uint256 intervalsPassed_ = timePassedBefore_ / interval_;
            uint256 intervalFullReward_ = initialAmount_ - intervalsPassed_ *
            decreaseAmount_;

            return (intervalFullReward_ * (endTime_ - startTime_)) / interval_;
        }

        // Calculate interval that more then 'interval_' range
        uint256 firstPeriodReward_ = _calculatePartPeriodReward(...);

        uint256 secondPeriodReward_ = _calculateFullPeriodReward(...);

        uint256 thirdPeriodReward_ = _calculatePartPeriodReward(...);

        return firstPeriodReward_ + secondPeriodReward_ + thirdPeriodReward_;
```

A problem arises when `initialAmount` is not exactly divisible by `decreaseAmount`. When `endTime_` is equal to or beyond the `maxEndTime_` (the cutoff point) it is regarded as a new interval. Therefore, if `startTime_` is contained within the last period, the reward calculation inside `LinearDistributionIntervalDecrease._calculateFullPeriodReward()` will revert since `LinearDistributionIntervalDecrease._divideCeil()` will round up and `initialAmount_ - decreaseRewardAmount_` will underflow.

```
uint256 timePassedBefore_ = startTime_ - payoutStart_;
uint256 intervalsPassedBefore_ = _divideCeil(timePassedBefore_, interval_);

uint256 decreaseRewardAmount_ = intervalsPassedBefore_ * decreaseAmount_;

// Overflow impossible because 'endTime_' can't be more then 'maxEndTime_'
uint256 initialReward_ = initialAmount_ - decreaseRewardAmount_;
```

Here is a mock example. Assume the following configuration

```
initialAmount_ = 10
decreaseAmount = 3
payoutStart = 100
interval = 10
startTime = 135
endTime = 140
```

First we have the condition `if ((timePassedBefore_ / interval_) == ((endTime_ - payoutStart_) / interval_)) ....` This evaluates to `(35 / 10) == (40 / 10)` which is false. The reward calculation will continue inside `LinearDistributionIntervalDecrease._calculateFullPeriodReward()`

The intervals passed are computed `uint256 intervalsPassedBefore_ = _divideCeil(timePassedBefore_, interval_);` This evaluates to `(35 + 10 - 1) / (10)` which is 4

The decrease amount is `uint256 decreaseRewardAmount_ = intervalsPassedBefore_ * decreaseAmount_;` This evalueates to `(4x3 = 12)`

The interval reward amount is `uint256 initialReward_ = initialAmount_ - decreaseRewardAmount_;` This evaluates to `(10 - 12)` and the function reverts.

The Impact implications are that `Distribution._getCurrentPoolRate()` will revert when `block.timestamp` becomes or surpasses the maximum end time of the distribution. This will block the stake, withdraw, claim, and edit functionality inside `Distribution.sol`.

Note that for this issue to occur `initialAmount` must not be evenly divisible by `decreaseAmount` which according to the Whitepaper it is not:

> The block reward will start at 14,400 MOR per day and then decline by 2.468994701 MOR each day, until the reward reaches 0 on day 5,833.

Recommendation:

```diff
## LinearDistributionIntervalDecrease.sol

@@ -142,7 +142,10 @@ library LinearDistributionIntervalDecrease {

         uint256 decreaseRewardAmount_ = intervalsPassedBefore_ * decreaseAmount_;

-        // Overflow impossible because 'endTime_' can't be more then 'maxEndTime_'
+        // Overflow possible if startTime is within the last interval
+        if (decreaseRewardAmount_ >= initialAmount_) {
+            reutrn 0;
+        }
         uint256 initialReward_ = initialAmount_ - decreaseRewardAmount_;
         // END
```

Morpheus: Fixed in commit 5af591e6ff2d7296f41ccb04bd5494ef0988a5ed.

Renascence: The issue has been fixed as recommended.

4.2 Low Risk

[L-1] Redundant nonce tracking in `L2MessageReceiver.sol`

Context: L2MessageReceiver.sol, Endpoint.sol

Description: `L2MessageReceiver.sol` implements its own nonce tracking via a mapping of `chainID => nonce`. When a message from LayerZero's Endpoint reaches `L2MessageReceiver._nonblockingLzReceive()` the nonce is checked if it has already been used and upon successful execution, it's marked as used "`true`" for the given `chainID`.

```
# L2MessageReceiver.sol

mapping(uint16 => mapping(uint64 => bool)) public isNonceUsed;
```

```
# L2MessageReceiver.sol

function _nonblockingLzReceive(
    uint16 senderChainId_,
    bytes memory senderAndReceiverAddresses_,
    uint64 nonce_,
    bytes memory payload_
) private {
    require(!isNonceUsed[senderChainId_][nonce_], "L2MR: invalid nonce");
    require(senderChainId_ == config.senderChainId, "L2MR: invalid sender chain ID");

    // execution logic ...

    isNonceUsed[senderChainId_][nonce_] = true;
}
```

The nonce tracking, however, is redundant since LayerZero implements its own inbound nonce tracking which is a mapping of source chain ID + sender's source address. In the context of Morpheus implementation of `L2MessageReceiver.sol`, the additional performed checks in `L2MessageReceiver._nonblockingLzReceive()` do not contribute towards increased security but also introduce a risk of the L2MessageReceiver contract denying minting of rewards. In the rare case that `config.sender` is set to the address of a new `L1Sender.sol` the nonce values from the new sender address will start at 0 and will already have been used. This can lead to LP's losing rewards since their to-be-minted rewards will not execute in `L2MessageReceiver._nonblockingLzReceive()`.

```
# Endpoint.sol

function receivePayload(uint16 _srcChainId, bytes calldata _srcAddress, address
_dstAddress, uint64 _nonce, uint _gasLimit, bytes calldata _payload) external
override receiveNonReentrant {
    // assert and increment the nonce. no message shuffling
>   require(_nonce == ++inboundNonce[_srcChainId][_srcAddress], "LayerZero: wrong
nonce");
    // code ...
}
```

Recommendation:

```
## L2MessageReceiver.sol

@@ -14,7 +14,6 @@ contract L2MessageReceiver is ILayerZeroReceiver,
IL2MessageReceiver, OwnableUpg

     Config public config;

-    mapping(uint16 => mapping(uint64 => bool)) public isNonceUsed;
     mapping(uint16 => mapping(bytes => mapping(uint64 => bytes32))) public
     failedMessages;

     function L2MessageReceiver__init() external initializer {
@@ -94,7 +93,6 @@ contract L2MessageReceiver is ILayerZeroReceiver,
IL2MessageReceiver, OwnableUpg
         uint64 nonce_,
         bytes memory payload_
     ) private {
-        require(!isNonceUsed[senderChainId_][nonce_], "L2MR: invalid nonce");
         require(senderChainId_ == config.senderChainId, "L2MR: invalid sender chain
         ID");

         address sender_;
@@ -107,7 +105,6 @@ contract L2MessageReceiver is ILayerZeroReceiver,
IL2MessageReceiver, OwnableUpg

         _mintRewardTokens(user_, amount_);

-        isNonceUsed[senderChainId_][nonce_] = true;
     }
```

Morpheus: Fixed in commit 5af591e6ff2d7296f41ccb04bd5494ef0988a5ed.

Renascence: The issue has been fixed as recommended and the nonce tracking has been removed. The nonce is now only used to retry a LayerZero message. And nonce tracking for the failed messages is unique to each `senderAndReceiverAddresses_` so the described issue cannot occur there.

[L-2] The alias of `L1Sender.sol` on Arbitrum will hold the ETH received as a gas refund from bridging tokens

Context: L1Sender.sol, L2TokenReceiver.sol

Description: Sending tokens to Arbitrum via a gateway requires ETH to cover gas fees where the excess ETH sent is given as a refund on Arbitrum. Lido's `L1ERC20TokenGateway.outboundTransfer(.., address _to, ..)` makes a call to `L1OutboundDataParser.decode()` to extract the sender's, i.e., "from" address. The fetched "from" address is then passed down as the `address sender_` in a call to `L1CrossDomainEnabled.sendCrossDomainMessage(address sender_, ...)`. In this function, there is a call to `IInbox.createRetryableTicket()` to create the cross-chain transaction. The issue here is that the `sender_` address is the address of the `L1Sender` contract, therefore, the L2 Alias of this address will receive the refund on Arbitrum. Currently, the only way around to rescue the excess ETH is for the owner to deploy a contract on Arbitrum that can rescue the ETH. However, this assumes the owner can use the same account nonce on Arbitrum that he has used to deploy on the Ethereum mainnet.

An additional note is that `L1Sender.sendDepositToken()` is callable by anyone. Although `L1Sender.sol` is not supposed to hold assets outside of being a temporary holder during a `Distribution.bridgeOverplus()` call, in rare cases that a user that is different from the owner calls `L1Sender.sendDepositToken()` the ETH gas refund will be again credited to the L2 alias of `L1Sender`.

```
# L1OutboundDataParser.sol

function decode(address router_, bytes memory data_)
    internal
    view
    returns (address, uint256)
{
    if (msg.sender != router_) {
        return (msg.sender, _parseSubmissionCostData(data_));
    }
    (address from, bytes memory extraData) = abi.decode(
        data_,
        (address, bytes)
    );
    return (from, _parseSubmissionCostData(extraData));
}
```

```
# L1CrossDomainEnabled.sol

function sendCrossDomainMessage(
        address sender_,
        address recipient_,
        bytes memory data_,
        CrossDomainMessageOptions memory msgOptions_
    ) internal returns (uint256 seqNum) {
        // code ...
        seqNum = inbox.createRetryableTicket{value: msg.value}(
            recipient_,
            msgOptions_.callValue,
            msgOptions_.maxSubmissionCost,
            sender_,
            sender_,
            msgOptions_.maxGas,
            msgOptions_.gasPriceBid,
            data_
        );

        emit TxToL2(sender_, recipient_, seqNum, data_);
    }
```

```
# IInbox.sol

interface IInbox {
    /// @notice Put an message in the L2 inbox that can be reexecuted for some fixed
    amount of time
    ///     if it reverts all msg.value will deposited to callValueRefundAddress on L2
    /// @param destAddr_ Destination L2 contract address
    /// @param arbTxCallValue_ Call value for retryable L2 message
    /// @param maxSubmissionCost_ Max gas deducted from user's L2 balance to cover
    base submission fee
    /// @param submissionRefundAddress_ maxGas x gasprice - execution cost gets
    credited here on L2 balance
    /// @param valueRefundAddress_ l2Callvalue gets credited here on L2 if retryable
    txn times out or gets cancelled
    /// @param maxGas_ Max gas deducted from user's L2 balance to cover L2 execution
    /// @param gasPriceBid_ Price bid for L2 execution
    /// @param data_ ABI encoded data of L2 message
    /// @return unique id for retryable transaction (keccak256(requestID, uint(0) )
    function createRetryableTicket(
        address destAddr_,
        uint256 arbTxCallValue_,
        uint256 maxSubmissionCost_,
>       address submissionRefundAddress_,
>       address valueRefundAddress_,
        uint256 maxGas_,
        uint256 gasPriceBid_,
        bytes calldata data_
    ) external payable returns (uint256);

    /// @notice Returns address of the Arbitumr's bridge
    function bridge() external view returns (address);
}
```

Recommendation: Deploy a contract that can rescue the ETH on the L2 alias address of `L1Sender`. Or otherwise, accept that any excess ETH sent is lost and protect `L1Sender.sendDepositToken()` with a check that `_msgSender() == distribution`.

```
## L1Sender.sol

@@ -92,6 +92,7 @@ contract L1Sender is IL1Sender, ERC165, OwnableUpgradeable,
UUPSUpgradeable {
         uint256 maxFeePerGas_,
         uint256 maxSubmissionCost_
     ) external payable returns (bytes memory) {
+        require(_msgSender() == distribution, "L1S: invalid sender");
         DepositTokenConfig storage config = depositTokenConfig;

         // Get current stETH balance
```

Morpheus: Fixed in commit 5af591e6ff2d7296f41ccb04bd5494ef0988a5ed.

Renascence: The recommendation has been implemented with regards to the access control. Only the `owner` of the `Distribution` contract can trigger the cross-chain transfer and he is responsible for sending along the appropriate amount of ETH.

[L-3] Negative rebasing of stETH deposits in `Distribution.sol` can cause bank run and loss to new depositors

Context: Distribution.sol

Description: The `Distribution._withdraw()` function pays out deposits on a first-come-first-serve basis. This means if there occurs a slashing of staked ETH in Lido (negative rebasing), the stETH balance in `Distribution` is insufficient to serve all users.

The `Distribution` contract accounts for this scenario by limiting the amount that can be withdrawn to the contract's balance:

```
## Distribution.sol

uint256 depositTokenContractBalance_ = IERC20(depositToken).balanceOf(address(this));
if (amount_ > depositTokenContractBalance_) {
    amount_ = depositTokenContractBalance_;
}
```

This may cause a "bank-run" where users want to withdraw their stETH as long as there is any left.

Once this "bank-run" is over, there may be a pool deposits balance greater zero and a stETH balance equal to zero. When new depositors come in, they could lose their funds to existing depositors that withdraw.

Recommendation: According to Lido docs, rebases have never been negative thus far.

Tracking `stETH` deposits directly seems to be a deliberate decision by the Morpheus team. As such the issue is simply a reminder for users interacting with the Morpheus smart contracts and the finding can be acknowledged.

Solving this issue would involve a shares tracking for every pool which requires an unreasonable amount of changes given the unlikely nature of a negative rebasing.

Morpheus: The finding has been acknowledged.

Renascence: The finding has been acknowledged as recommended. It is fair to consider a substantial negative rebase of stETH a systemic risk rather than a case that needs to be handled in the smart contracts.

4.3 Informational

[I-1] `Distribution.editPool()` contains a needless check

Context: Distribution.sol, LinearDistributionIntervalDecrease

Description: Creating or editing a pool via `Distribution.createPool()` and `Distribution.editPool()` will make a call to the private function `Distribution._validatePool()`. The condition and requirement inside the private function however are not purposeful since if `pool_.decreaseInterval = 0` a call to `LinearDistributionIntervalDecrease.getPeriodReward()` will always return a 0 reward independent of `pool_.rewardDecrease`.

The `pool_.decreaseInterval == 0` value is not sensible regardless of the `pool_.rewardDecrease`.

```
# Distribution.sol

function _validatePool(Pool calldata pool_) private pure {
        if (pool_.rewardDecrease > 0) {
            require(pool_.decreaseInterval > 0, "DS: invalid reward decrease");
        }
    }
```

```
# LinearDistributionIntervalDecrease.getPeriodReward.sol

function getPeriodReward(
        uint256 initialAmount_,
        uint256 decreaseAmount_,
        uint128 payoutStart_,
        uint128 interval_,
        uint128 startTime_,
        uint128 endTime_
    ) external pure returns (uint256) {
>       if (interval_ == 0) {
            return 0;
        }
        // code ...
    }
```

Recommendation:

```
## Distribution.sol

@@ -109,9 +109,7 @@ contract Distribution is IDistribution, OwnableUpgradeable,
UUPSUpgradeable {
     }

     function _validatePool(Pool calldata pool_) private pure {
-        if (pool_.rewardDecrease > 0) {
-            require(pool_.decreaseInterval > 0, "DS: invalid reward decrease");
-        }
+        require(pool_.decreaseInterval > 0, "DS: invalid decrease interval");
     }
```

Morpheus: Fixed in commit 5af591e6ff2d7296f41ccb04bd5494ef0988a5ed.

Renascence: The check has been fixed as recommended.

[I-2] Rewards can be griefed when a distribution configuration emits more tokens than the rewards token cap

Context: Distribution.sol, L2MessageReceiver.sol

Description: The `Distribution.claim()` function can be called by anyone to initiate the claiming of rewards for a given `address user_`. This can cause problems in case the maximum amount (cap) of the reward token is less than the emitted rewards. In such a case a malicious actor can call `Distribution.claim()` and pass as parameter an `address user_` that has pending rewards that will exceed the cap and therefore `L2MessageReceiver._mintRewardTokens()` will only mint tokens up to the cap, forfeiting the rest of the user's reward. In the context of Morpheus, this shouldn't be possible since the reward's token cap is defined as all of the tokens that will be ever emitted by `Distribution.sol`. The issue is disclosed as Informational in case Morpheus rewards distribution is altered or the aforementioned contracts are used by Morpheus or any other third party for a different purpose.

```
# L2MessageReceiver.sol

function _mintRewardTokens(address user_, uint256 amount_) private {
        uint256 maxAmount_ = IMOR(rewardToken).cap() -
        IMOR(rewardToken).totalSupply();

        if (amount_ == 0 || maxAmount_ == 0) {
            return;
        }

        if (amount_ > maxAmount_) {
            amount_ = maxAmount_;
        }

        IMOR(rewardToken).mint(user_, amount_);
    }
```

Recommendation: Ensure that the emitted rewards by `Distribution.sol` do not exceed the reward's token cap. This does not have to be enforced on-chain but is simply a policy that the owner must follow.

Morpheus: Fixed in commit 7564d56b20b263cb943ef52358f90e8df331b2a3.

Renascence: A fix has been applied in the `L2MessageReceiver` contract. The amount to be minted is no longer constrained to the MOR supply cap. Instead when the amount exceeds the MOR supply cap, the transaction fails and can be retried later.

[I-3] `UUPSUpgradeable` contracts should implement a constructor with a call to `Initializable.disableInitializers()`

Context: Distribution.sol, L1Sender.sol, L2TokenReceiver, L2MessageReceiver.sol

Description: The best practice in contracts that inherit from `UUPSUpgradeable` is to disable the initializers since if left uninitialized they can be invoked in the implementation contract by an attacker. For example, there is a past vulnerability disclosure that demonstrates how initializers getting called in the implementation can lead to contract takeover where the attacker can appoint an owner and would self-destruct the implementation, therefore, bricking the Proxy: OZ post-mortem. Although this issue has been fixed from OZ version 4.3.2 it's still best practice to call `Initializable._disableInitializers()` in a constructor in the implementation.

```
# Initializable.sol

* [CAUTION]
 * ====
 * Avoid leaving a contract uninitialized.
 *
 * An uninitialized contract can be taken over by an attacker. This applies to both a
 proxy and its implementation
 * contract, which may impact the proxy. To prevent the implementation contract from
 being used, you should invoke
 * the {_disableInitializers} function in the constructor to automatically lock it
 when it is deployed:
 *
```

Recommendation:

```
## Distribution.sol

@@ -29,6 +29,9 @@ contract Distribution is IDistribution, OwnableUpgradeable,
UUPSUpgradeable {

     // Total deposited storage
     uint256 public totalDepositedInPublicPools;
+    constructor() {
+        _disableInitializers();
+    }

     /********************************************************************************
     ***************/
     /*** Modifiers
```

```
## L1Sender.sol

@@ -20,6 +20,10 @@ contract L1Sender is IL1Sender, ERC165, OwnableUpgradeable,
UUPSUpgradeable {
     DepositTokenConfig public depositTokenConfig;
     RewardTokenConfig public rewardTokenConfig;

+    constructor() {
+        _disableInitializers();
+    }
+
     function L1Sender__init(
         address distribution_,
         RewardTokenConfig calldata rewardTokenConfig_,
```

```
## L2TokenReceiver.sol

@@ -16,6 +16,9 @@ contract L2TokenReceiver is IL2TokenReceiver, OwnableUpgradeable,
UUPSUpgradeabl

     SwapParams public params;

+    constructor() {
+        _disableInitializers();
+    }
     function L2TokenReceiver__init(
         address router_,
         address nonfungiblePositionManager_,
```

```
## L2MessageReceiver.sol

@@ -17,6 +17,9 @@ contract L2MessageReceiver is ILayerZeroReceiver,
IL2MessageReceiver, OwnableUpg
     mapping(uint16 => mapping(uint64 => bool)) public isNonceUsed;
     mapping(uint16 => mapping(bytes => mapping(uint64 => bytes32))) public
     failedMessages;

+    constructor() {
+        _disableInitializers();
+    }
     function L2MessageReceiver__init() external initializer {
         __Ownable_init();
         __UUPSUpgradeable_init();
```

Morpheus: Fixed in commit 5af591e6ff2d7296f41ccb04bd5494ef0988a5ed

Renascence: The finding has been fixed as recommended.

[I-4] Misleading comments in `IDIstribution.sol` and `IL2MessageReceiver.sol`

Context: IDistribution.sol, IL2MessageReceiver.sol

Description: The comment above the declaration of `IL2MessageReceiver.nonblockingLzReceive()` that states "LayerZero endpoint call this function to check a transaction capabilitiesis" is wrong since LayerZero will call `L2MessageReceiver.lzReceive()`. `L2MessageReceiver.nonblockingLzReceive()` is only called by the `L2MessageReceiver`.

```
    /**
>    * LayerZero endpoint call this function to check a transaction capabilities.
     * @param senderChainId_ The source endpoint identifier.
     * @param senderAndReceiverAddresses_ The source sending contract address from the
     source chain.
     * @param nonce_ The ordered message nonce.
     * @param payload_ The signed payload is the UA bytes has encoded to be sent.
     */
    function nonblockingLzReceive(
    // code ...
```

The comment above the declaration of `IDistribution.totalDepositedInPublicPools()` is wrong since the function will return the total deposited tokens in all public pools.

```
    /**
>    * The function to get the amount of deposit tokens that are staked in the pool.
     * @dev The value accumulates the amount amount despite the rate differences.
     * @return The amount of deposit tokens.
     */
    function totalDepositedInPublicPools() external view returns (uint256);
```

Recommendation:

```
## IL2MessageReceiver.sol

@@ -64,7 +64,6 @@ interface IL2MessageReceiver is ILayerZeroReceiver {
     function setParams(address rewardToken_, Config calldata config_) external;

     /**
-     * LayerZero endpoint call this function to check a transaction capabilities.
      * @param senderChainId_ The source endpoint identifier.
      * @param senderAndReceiverAddresses_ The source sending contract address from
      the source chain.
      * @param nonce_ The ordered message nonce.
```

```
## IDistribution.sol

@@ -209,7 +209,7 @@ interface IDistribution {
     function l1Sender() external view returns (address);

     /**
-     * The function to get the amount of deposit tokens that are staked in the pool.
+     * The function to get the amount of deposit tokens that are staked in all of the
public pools.
      * @dev The value accumulates the amount amount despite the rate differences.
      * @return The amount of deposit tokens.
      */
```

Morpheus: Fixed in commit [5af591e6ff2d7296f41ccb04bd5494ef0988a5ed](5af591e6ff2d7296f41ccb04bd5494ef0988a5ed).

Renascence: The comments have been corrected.

5 Centralization Risks

Users interacting with the MorpheusAI Smart Contracts must be aware of the privileged roles in the protocol and which actions these privileged roles can perform.

`Distribution.sol` is the contract that will hold the user's staked funds i.e. a token such as `stETH`. Currently, there are mechanisms, controlled only by the owner of the contract, that can cause indefinite custody of the user's deposited tokens. In other words, users would never be able to withdraw their staked funds again. These mechanisms are the following:

- The `pool.withdrawLockPeriod`, `pool.withdrawLockPeriodAfterStake`, and `pool.payoutStart` state variables for a given pool can be modified by the owner through the `Distribution.editPool()` function. They can be assigned to arbitrary time points in the future such that the "lock period" checks inside `Distribution._withdraw()` will revert, preventing users from withdrawing their staked funds.
- The `pool.initialReward` variable for a given pool can be set by an admin through `Distribution.editPool()` to a large number such that a call to `Distribution._getCurrentPoolRate()` will revert because of an overflow in functions such as `LinearDistributionIntervalDecrease._calculateMaxEndTime()` and `LinearDistributionIntervalDecrease._calculateFullPeriodReward()`. Since `Distribution._getCurrentPoolRate()` is invoked during `Distribution.claim()`, `Distribution.stake()`, `Distribution.withdraw()`, and `Distribution.editPool()` the user's funds can remain locked in the contract without the possibility for the pool to be edited back in a state that can recover the funds.
- `L1Sender.sol`, `L2TokenReceiver.sol`, and `L2MessageReceiver.sol` cannot disable upgrades in contrast to `Distribution.sol` which can disable upgrades via `Distribution.removeUpgradeability()`. Upgrades of these contracts could potentially include redirection of the user's earned yield to an arbitrary party or denying users of MOR token rewards.

In summary, the owner role must be fully trusted.