Introduction

A smart contract security audit review of the Ailayer 6079 protocol was done by blockchain security researcher 0xladboy, with a focus on the security aspects of the application's implementation.

About Auditing firm

Octane Security is an AI-driven blockchain security firm dedicated to safeguarding digital assets and transactions with cutting-edge machine learning models.

https://www.octane.security/

0xladboy is a blockchain security researcher who ranks high in audit contests, captures bug bounties, and has served big-name clients such as Optimism, Notional Finance, and Graph Protocol in the past.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

Security Assessment Summary

review commit hash - **3ed689e704bc4e8f9aeeda1d9934a4a03c04cb07**

Scope

The following smart contracts were in scope of the audit:

- Distribution.sol
- L1Sender.sol
- L2TokenReceiver.sol
- L2MessageReceiver.sol
- RewardClaimer.sol
- Splitter.sol
- Token.sol

Findings

ID	Title	Severity
M-1	User can earn free reward when stETH has negative rebase	Medium
M-2	Reward time lock only applies to early stakers	Medium
L-1	Lack of function to update splitter contract and l1 sender in distribution contract	Low
L-2	Outdated Token Owner can still act as delegatee when managing token configuration.	Low
L-3	RetryMessage should have reentrancy protection	Low
L-4	State change functions are missing event emission	Low
L-5	Incomplete production deployment configuration.	Low
L-6	Owner can consider alternative to replace the RewardClaimer.sol contract	Low
L-7	Missing decrease liquidity function in L2TokenReceiver.sol	Low
L-8	Add safety check to prevent arithmic underflow in reward accounting.	Low
R-1	Security Recommendation	Recommendation

M – User can earn free reward when stETH has negative rebase

Description

user can stake stETH in exchange for token rewards,

protocol can bridge the additional yield from positive rebase of stETH to l2.

However, in case when slashing events occrus, the stETH token can undergo negative rebase.

https://docs.lido.fi/guides/lido-tokens-integration-guide#accounting-oracle

when user deposits the token, the code enforces that the miniumum stake amount must be at least pool.minimalStake,

```
require(userData.deposited + amount_ >= pool.minimalStake, "DS: amount too low");
```

in deployment configuration, pool.minimalStake is set to 0.01 ETH,

when user withdraws the token, the code enforces that the minimum stake amount left is pool.minimalStake as well:

```
uint256 newDeposited_ = deposited_ - amount_;
require(amount_ > 0, "DS: nothing to withdraw");
require(newDeposited_ >= pool.minimalStake || newDeposited_ == 0, "DS: invalid withdraw amount");
```

consider the case:

1. user A deposit and stake 1 stETH.
2. stETH has negative rebase, the 1 stETH that user stake is reduce to 0.98 stETH.

3. user A want to withdraw the full 0.98 stETH, there is no user balance left for user A to withdraw,
4. but in the user data struct after withtdraw, the userData.deposited is updated.

```
    // Update user data
    userData.rate = currentPoolRate_;
    userData.deposited = newDeposited_;
    userData.pendingRewards = pendingRewards_;
```

user.deposit will be 1 stETH - 0.98 stETH = 0.02 stETH after withdraw while in reality user.deposit should be 0 because the 0.02 stETH is not in the contract because of stETH negative rebaes,

then while user can never withdraw these 0.02 stETH, the user can earn free reward using these leftover 0.02 stETH.

Recommendation

https://docs.lido.fi/guides/lido-tokens-integration-guide/#transfer-shares-function-for-steth

consider track stETH shares instead of tracking balance to handle the negative rebasing of the stETH.

M - Reward time lock only applies to early stakers

Description

When user stakes, the reward is accured, and user can claim and bridge the reward out to l2,

```
    function claim(address receiver_) external payable {
        address user_ = _msgSender();

        UserData storage userData = usersData[user_];
```

```
        uint256 payoutStart_ = ISplitter(splitter).getPayoutStart();

        // user can stake latter does not have to wait for the payoutStart_
+ pool.claimLockPeriod
        require(block.timestamp > payoutStart_ + pool.claimLockPeriod, "DS:
pool claim is locked");

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

        // Update pool data
        poolData.distributedRewards += rewards_;
        poolData.rate = currentPoolRate_;

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

        // Transfer rewards

        IL1Sender(l1Sender).sendMintMessage{value: msg.value}(receiver_,
pendingRewards_, user_);

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

as we can see, after the timelock

```
// user can stake latter does not have to wait for the payoutStart_ +
pool.claimLockPeriod
require(block.timestamp > payoutStart_ + pool.claimLockPeriod, "DS: pool
claim is locked");
```

the code validates that after the claim lock period, the user can claim the reward,

but this reward timelock only applies to early stakers, when payoutStart_ + pool.claimLockPerio > block.timestamp,

the staker can immediately deposit and stake token and accures reward and claim the rewards after the reward timelock passes,

consider the case:

1. at day 1, the distribution contract is deployed, user can stake at day 1, but user can claim at day 2.
2. user A stake after day 1 and needs to wait for 1 day to claim the reward.
3. user B, however, stake at day 2.5, and day 2.5 already surpass day 2, user B's reward is immediately claimable while user A has to wait for the timelock.

Recommendation

the lack of timelock for later staker can be partially mitigated by setting steep reward decline rate,

When withdraw, the code tracks the last deposit time stamp and enforce a timelock and the timelock refreshes in after deposit transaction:

```
require(
    block.timestamp < payoutStart ||
        (block.timestamp > payoutStart + pool.withdrawLockPeriod &&
            block.timestamp > userData.lastStake +
pool.withdrawLockPeriodAfterStake),
    "DS: pool withdraw is locked"
);
```

note the check:

```
 block.timestamp > userData.lastStake + pool.withdrawLockPeriodAfterStake),
```

this enforces that every time when user stakes, user cannot withdraw immediately for every user,

it is recommended to apply similar timelock when user claim the reward:

```
require(block.timestamp > payoutStart_ + pool.claimLockPeriod &&
userData.lastStake + pool.userclaimLockPeriod , "DS: pool claim is
locked");
```

to make sure after payoutStart + pool.withdrawLockPeriod, user can stake and earn reward immediately.

L1 – Lack of function to update splitter contract and l1 sender in distribution contract

Description

In RewardClaimer, there are functions that update splitter and l1 sender contract,

Which indicates that if the splitter and l1 sender contract changed,

the smart contract that calling splitter and l1 sender contract method should update the contract address.

However, the update method that presents in the RewardClaimer is missing in Distribution contract.

In case when the splitter and l1 sender contract changes,

the splitter and l1 sender contract in RewardClaimer can be out of sync with splitter and l1 sender contract in Distribution.sol

Recommendation

add these two method to Distribution.sol contract as well:

```
    function setSplitter(address splitter_) external onlyOwner {
          splitter = splitter_;
      }

      function setL1Sender(address l1Sender_) external onlyOwner {
          l1Sender = l1Sender_;
      }
```

L – Outdated Token Owner can still act as delegatee when managing token configuration.

Description

In Token.sol,

the contract is a OFT leveraging layerzero v2:

```
contract Token is IToken, OFT {
```

```
        mapping(address => bool) public isMinter;

        constructor(
            string memory name_,
            string memory symbol_,
            uint256 amountToMint_,
            address receiver_,
            address layerZeroEndpoint_,
            address delegate_,
            address minter_
        ) OFT(name_, symbol_, layerZeroEndpoint_, delegate_) {
            require(minter_ != address(0), "Token: invalid minter");

            isMinter[minter_] = true;

            _mint(receiver_, amountToMint_);

            transferOwnership(delegate_);
        }
```

As we can see, the ownership is transferred to delegate_ address:

```
    transferOwnership(delegate_);
```

Both owner and delegate_ privileged role:

Owner can update minter address and minter can mint arbitrary amount of token.

Delegate help manage the OFT dapp configuration via layerzero v2 endpoint.

The inheritance relationship is:

Token -> OFT -> OFTCore -> OApp -> OAppCore

eventually we are calling endpoint.setDelegate(_delegate).

```
abstract contract OAppCore is IOAppCore, Ownable {
    // The LayerZero endpoint associated with the given OApp
    ILayerZeroEndpointV2 public immutable endpoint;

    // Mapping to store peers associated with corresponding endpoints
    mapping(uint32 eid => bytes32 peer) public peers;

    /**
     * @dev Constructor to initialize the OAppCore with the provided endpoint and delegate.
     * @param _endpoint The address of the LOCAL Layer Zero endpoint.
     * @param _delegate The delegate capable of making OApp configurations inside of the endpoint.
     *
     * @dev The delegate typically should be set as the owner of the contract.
     */
    constructor(address _endpoint, address _delegate) {
        endpoint = ILayerZeroEndpointV2(_endpoint);

        if (_delegate == address(0)) revert InvalidDelegate();
        endpoint.setDelegate(_delegate);
    }
```

the code for setDelegate is here

```
    /// @notice delegate is authorized by the oapp to configure anything in
layerzero
    function setDelegate(address _delegate) external {
        delegates[msg.sender] = _delegate;
        emit DelegateSet(msg.sender, _delegate);
    }
```

the delegate address can clear message payload

```
    function clear(address _oapp, Origin calldata _origin, bytes32 _guid,
bytes calldata _message) external {
        _assertAuthorized(_oapp);

        bytes memory payload = abi.encodePacked(_guid, _message);
        _clearPayload(_oapp, _origin.srcEid, _origin.sender, _origin.nonce,
payload);
        emit PacketDelivered(_origin, _oapp);
    }
```

and delegate address can burn / skip message or nullify message

The owner and delgatee roles are both important,

but consider the case:

1. token is deployed, address A is both owner and delegatee.
2. address A is compromised, protocol detect it and transfer the owner to address B.
3. address B is the owner, but address A is still the delegatee, which is clearly of erase cross-chain payment and make user lose fund (erase pending message for wstETH bridging request or reward minting request).

Recommendation

When triggering transfer ownership, it is recommended the protocol calling setDelegate as well

L – RetryMessage should have reentrancy protection

Description

In L2MessageReceiver, the method retryMessage code is below:

```
    function retryMessage(
          uint16 senderChainId_,
          bytes memory senderAndReceiverAddresses_,
          uint64 nonce_,
          bytes memory payload_
      ) external {
          bytes32 payloadHash_ = failedMessages[senderChainId_]
[senderAndReceiverAddresses_][nonce_];
          require(payloadHash_ != bytes32(0), "L2MR: no stored message");
          require(keccak256(payload_) == payloadHash_, "L2MR: invalid
payload");

          _nonblockingLzReceive(senderChainId_, senderAndReceiverAddresses_,
payload_);

          delete failedMessages[senderChainId_][senderAndReceiverAddresses_]
[nonce_];

          emit RetryMessageSuccess(senderChainId_,
senderAndReceiverAddresses_, nonce_, payload_);
      }
```

the state is cleared after external call:

```
    _nonblockingLzReceive(senderChainId_, senderAndReceiverAddresses_,
payload_);

    delete failedMessages[senderChainId_]
```

while assume the reward token is always the Token contract in scope,

```
    function _nonblockingLzReceive(
        uint16 senderChainId_,
        bytes memory senderAndReceiverAddresses_,
        bytes memory payload_
    ) private {
        require(senderChainId_ == config.senderChainId, "L2MR: invalid
sender chain ID");
        address sender_;
        assembly {
            sender_ := mload(add(senderAndReceiverAddresses_, 20))
        }
        require(sender_ == config.sender, "L2MR: invalid sender address");

        (address user_, uint256 amount_) = abi.decode(payload_, (address,
uint256));

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

the only external call is:

```
    IToken(rewardToken).mint(user_, amount_);
```

and the user cannot trigger a callback to re-execute retryMessage multiple times,

it is still good to add nonReentrant modifier to prevent reentrancty.

Recommendation

Adding nonReentrant modifier to function retryMessage

L - State change functions are missing event emission

Description

A lot of state change function maanged by owner address does not emit events

In RewardClaimer contract:

```
    function setSplitter(address splitter_) external onlyOwner {
        splitter = splitter_;
    }

    function setL1Sender(address l1Sender_) external onlyOwner {
        l1Sender = l1Sender_;
    }
```

In Splitter contract:

```
function updatePool(Pool memory pool_) external onlyOwner {

function updateGroupShares(address[] calldata groups_, uint256[] calldata
shares_) external onlyOwner {
```

It is recommended to add event emission for offline service to track state and configuration changes.

Recommendation

It is recommended to add event emission for offline service to track state and configuration changes for all function managed managed by owner.

```
function updatePool(Pool memory pool_) external onlyOwner {

function updateGroupShares(address[] calldata groups_, uint256[] calldata
shares_) external onlyOwner {
```

In Splitter contract, these two method can impact reward accounting so it is highly recommended to add event emission to two method above to track how the share weight is updated.

L – Incomplete production deployment configuration.

Description

Currently the production deployement configuration set up is not completed.

```
{
  "chainsConfig": {
    "senderChainId": 101,
    "receiverChainId": 110
  },
  "rewardToken": {
    "name": "NAME",
    "symbol": "SYMBOL",
    "amountToMint": "1620000000000000000000000000",
    "receiver": "0x",
    "delegate": "0x"
  },
  "distributionPool": {
```

```
    "payoutStart": 1745330623,
    "withdrawLockPeriod": 604800,
    "claimLockPeriod": 7776000,
    "withdrawLockPeriodAfterStake": 604800,
    "minimalStake": "10000000000000000"
  },
  "splitterPool": {
    "payoutStart": 1745330623,
    "decreaseInterval": 86400,
    "initialReward": "3456000000000000000000",
    "rewardDecrease": "592558728240000000"
  },
  "feeData": {
    "fee": 1,
    "treasury": "0x",
    "feeOwner": "0x"
  },
  "L1": {
    "stEth": "0xae7ab96520DE3A18E5e111B5EaAb095312D7fE84",
    "wStEth": "0x7f39C581F595B53c5cb19bD0b3f8dA6c935E2Ca0"
  },
  "L2": {
    "swapRouter": "0xE592427A0AEce92De3Edee1F18E0157C05861564",
    "nonfungiblePositionManager":
"0xC36442b4a4522E871399CD717aBDD847Ab11FE88",
    "wStEth": "0x5979D7b546E38E414F7E9822514be443A4800529"
  },
  "swapParams": {
    "fee_1": 3000,
    "sqrtPriceLimitX96_1": 0,
    "fee_2": 3000,
    "sqrtPriceLimitX96_2": 0,
    "intermediateToken": "0x82aF49447D8a07e3bd95BD0d56f35241523fBab1"
  },
```

```
    "arbitrumConfig": {
      "arbitrumBridgeGatewayRouter":
  "0x72Ce9c846789fdB6fC1f34aC4AD25Dd9ef7031ef"
    },
    "lzConfig": {
      "lzEndpointL1": "0x66A71Dcef29A0fFBDBE3c6a460a3B5BC225Cd675",
      "lzEndpointL2": "0x3c2269811836af69497E5F486A85D7316753cf62",
      "lzEndpointL2_OFT": "0x1a44076050125825900e736c501f859c50fE728c"
    }
  }
```

while certain configuration (such as "chainsConfig", "receiverChainId", and address of stETH and wstETH) are correct,

the configuration for rewardToken is missing, the name, symbol, reciever and delegatee address (owner address that can and must set the minter) are not set;

```
    "rewardToken": {
      "name": "NAME",
      "symbol": "SYMBOL",
      "amountToMint": "162000000000000000000000000000",
      "receiver": "0x",
      "delegate": "0x"
    },
```

deploying the token using delegate address address(0) can block and revert token deployment because in openzepplin contract, the transferOwnership code enforce that owner cannot be address:

```
    function transferOwnership(address newOwner) public virtual onlyOwner {
            require(newOwner != address(0), "Ownable: new owner is the zero
    address");
            _transferOwnership(newOwner);
        }
```

the fee data is not set as well:

```
    "feeData": {
       "fee": 1,
       "treasury": "0x",
       "feeOwner": "0x"
     },
```

Recommendation

It is recommended the protocol update the deployment configuration of reward token and fee data and double check if the payout time and reward are
set up correctly

```
  "distributionPool": {
    "payoutStart": 1745330623,
    "withdrawLockPeriod": 604800,
    "claimLockPeriod": 7776000,
    "withdrawLockPeriodAfterStake": 604800,
    "minimalStake": "10000000000000000"
  },
  "splitterPool": {
    "payoutStart": 1745330623,
    "decreaseInterval": 86400,
    "initialReward": "3456000000000000000000",
    "rewardDecrease": "592558728240000000"
  },
```

the payout start time is 1745330623, if we convert this utc timestamp to date time, it is April 22, 2025, at 14:03:43.

L – Owner can consider alternative to replace the RewardClaimer.sol contract

Description

All contract function in RewardClaimer.sol is managed by owner.

```
  function claim(address receiver_) external payable onlyOwner {
        uint256 reward_ = getCurrentReward();
        require(reward_ > 0, "RC: nothing to claim");

        distributedRewards += reward_;

        IL1Sender(l1Sender).sendMintMessage{value: msg.value}(receiver_,
reward_, receiver_);
```

```
        emit RewardClaimed(receiver_, reward_);
    }

    function setSplitter(address splitter_) external onlyOwner {
        splitter = splitter_;
    }

    function setL1Sender(address l1Sender_) external onlyOwner {
        l1Sender = l1Sender_;
    }

    function getCurrentReward() public view returns (uint256) {
        return
ISplitter(splitter).getCurrentGroupTotalRewards(address(this)) -
distributedRewards;
    }
```

the main function is for owner to claim reward and mint the reward on l2 based on the split pool shares.

But to save gas, the owner can just add a minter address and mint the reward directly in l2

Recommendation

the owner can add a minter address to token contract and mint the reward directly in l2 instead of use RewardClaiming in l1 to save gas and manage pool share configuration for rewardClaimer.

```
    function updateMinter(address minter_, bool status_) external onlyOwner
{
        isMinter[minter_] = status_;
    }
```

```
    /**
     * @notice The function to mint tokens.
     * @param account_ The address of the account to mint tokens to.
     * @param amount_ The amount of tokens to mint.
     */
    function mint(address account_, uint256 amount_) public {
        require(isMinter[_msgSender()], "Token: invalid caller");

        _mint(account_, amount_);
    }
```

L – Missing decrease liquidity function in L2TokenReceiver.sol

Description

In L2TokenReceiver.sol, the protocol implement function to increase liquidity for uniswap v3 position nft

or collect fees from the position, but there is no function to decrease liquidity.

Then the process of decrease liquidity becomes a little cumbersome:

suppose the protocol:

1. mint a uniswap v3 position nft.
2. transfer the nft int o the L2TokenReceiver contract.
3. increase the liquidity of the position.
4. collect the fee from the position.

If the protocol wants to decrease the liquidity of the position, it needs to:

5. transfer the nft out to the protocol.

6. decrease the liquidity of the position.
7. transfer the nft back to the L2TokenReceiver contract.

Recommendation

It is recommended to add a function to decrease the liquidity of the uniswap v3 position nft in the L2TokenReceiver.sol contract to avoid transfer the nft back and forth when decrease liquidity.

L – Add safety check to prevent arithmic underflow in reward accounting.

Description

In reward accounting, it is important to ensure that is a no arithmic underflow, otherwise, transaction revert with no error message and it is very difficult to debug in production.

If the arithmic underflow occurs, the user's withdraw transaction also revert, which block user from getting their fund.

Recommendation

It is recommended to add the check below in these two place to prevent arithmic underflow.

In Distribution contract

```
    function _getCurrentPoolRateAndTotalRewards() private view returns
(uint256, uint256) {

        // cache for gas saving.
        uint256 rewards_;
        uint256 currentTotalRewardRewards =
ISplitter(splitter).getCurrentGroupTotalRewards(address(this));
        uint256 distributedRewards = poolData.distributedRewards;
```

```
        if (currentTotalRewardRewards < distributedRewards) {
            rewards = 0;
        } {
            rewards_ = currentTotalRewardRewards - distributedRewards;
        }

        if (poolData.totalDeposited == 0) {
            return (poolData.rate, rewards_);
        }

        uint256 rate_ = poolData.rate + (rewards_ * PRECISION) /
poolData.totalDeposited;

        return (rate_, rewards_);
    }
```

In Splitter contract:

```
    function _getCurrentGroupTotalRewards(
        uint256 currentPoolRate_,
        GroupData memory group_
    ) private pure returns (uint256) {

        if (group_.rate > currentPoolRate_ ) {
            return group_.pendingRewards;
        }

        uint256 newRewards_ = ((currentPoolRate_ - group_.rate) *
group_.share) / PRECISION;

        return group_.pendingRewards + newRewards_;
    }
```

R - Security Recommendation

In high level,

the contract let user stake stETH token and the stETH positive rebase yield goes to yield in exchange for token reward.

there are a few high level security invariant that the protocol must maintain.

1. user should be able to withdraw their staked fund.
2. no one should not mint excessive token reward.

to maintain the first invariant, anything that block user withdraw should be treat with care:

so a few security measure should be added:

- make sure there is no arithmic underflow in reward account.
- make sure negative rebase of stETh does not block user from withdraw fund.
- add a function to let user give up reward and withdraw the original stETH directly in extreme case, the sushi master chef contract actually has a function to do that to avoid lock user fund as saftey measure.

to maintain the second invariant, a few security measure should be added:

- add minting cap and max total supply.
- make deposit and claim function pausable.
- add a expiration of the pool data and reward data to make sure after the distribution pool expires, no one can deposit and stake, while make sure withdraw function is always active.