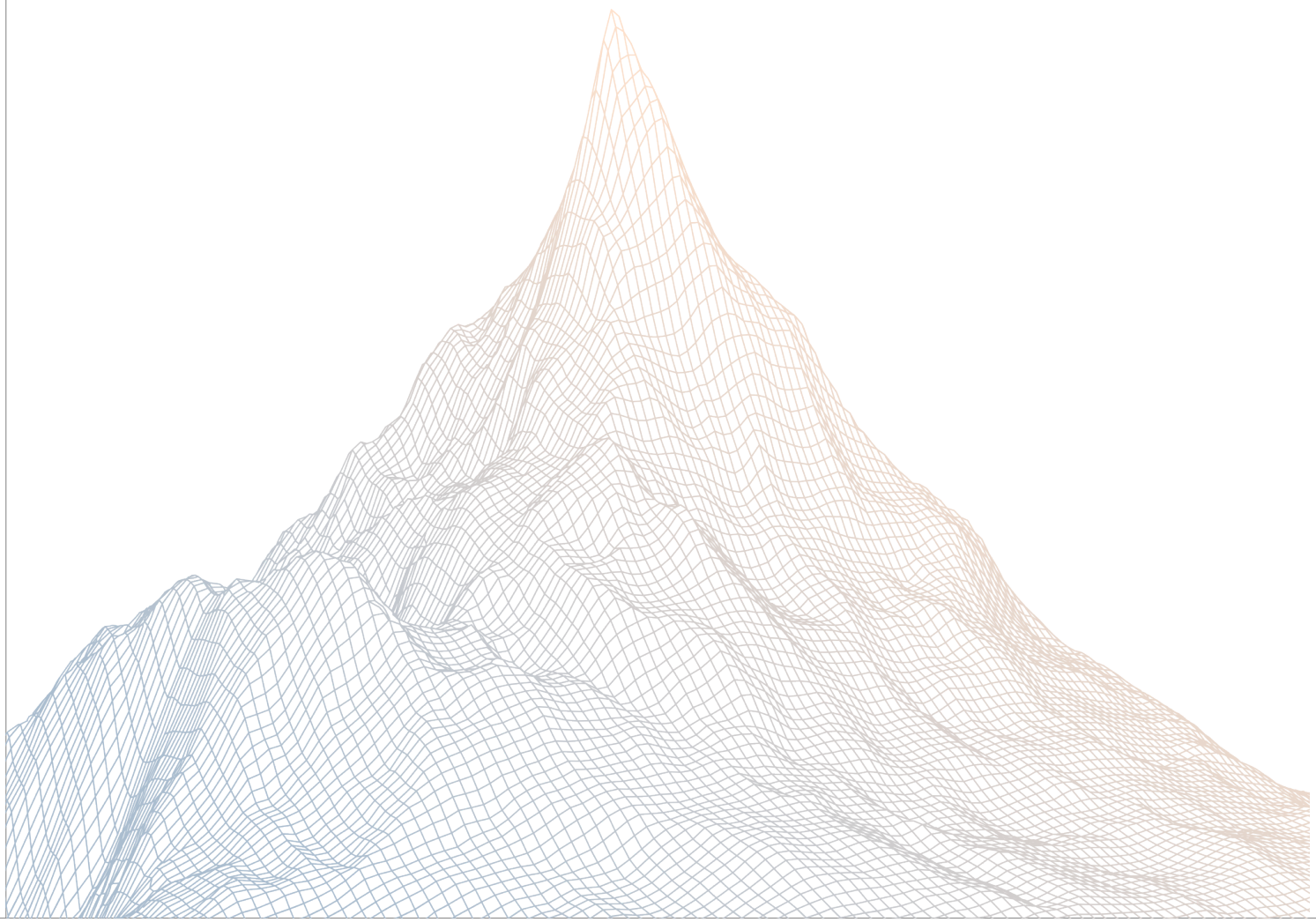
Zenith

Parallel Protocol

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: February 3rd to February 12th, 2025

AUDITED BY: 0xLadboy
ether_sky

Contents

1 Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

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

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Parallel Protocol

Parallel is a decentralized protocol that issues stablecoins, the € stablecoin (PAR) and the $ stablecoin (paUSD), on the Ethereum, Polygon and Fantom blockchains. The PAR & paUSD stablecoin are decentralized, non-custodials, and collateral-backed FIAT stablecoins.

2.2 Scope

The engagement involved a review of the following targets:

Target	tokenomics
Repository	https://github.com/parallel-protocol/tokenomics
Commit Hash	`cd3992aec0847063c2b979c51f6cac7dd2ff03bd`

Files	`sPRL/*`
	`fees/*`
	`rewardMerkleDistributor/*`

Target	PRL-token
Repository	https://github.com/parallel-protocol/PRL-token
Commit Hash	`0f35481fa57cd217066816fb4fa9318baf67d319`

Files	`principal/*`
	`peripheral/*`
	`layerZero/*`

2.3 Audit Timeline

DATE	EVENT
February 03, 2025	Audit start
February 12, 2025	Audit end
February 19, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	4
Medium Risk	2
Low Risk	6
Informational	4
Total Issues	**16**

3

Findings Summary

ID	DESCRIPTION	STATUS
H-1	sPRL2.sol cannot claim extra reward token because of interface mismatch	Resolved
H-2	CVX rewards cannot be claimed in sPRL2	Resolved
H-3	The calculation of withdrawable WETH and PRL amounts in the withdrawPRLAndWeth function of sPRL2 is incorrect	Resolved
H-4	MainFeeDistributor.sol#‘swapLzToken‘ function always revert because of interface mismatch	Resolved
M-1	The SideChainFeeCollector does not work correctly when the fee token has decimals other than 18	Acknowledged
M-2	Migrating PRL tokens to another chain may fail due to insufficient gas	Resolved
L-1	The epoch should only be updated once	Resolved
L-2	Paying fees with LzToken is not allowed in the migrateToPRLAnd-Bridge function of the PrincipalMigrationContract	Resolved
L-3	Consider use block.timestamp for ERC20Vote token clock mode	Resolved
L-4	‘updateMerkleDrop‘’s safety check might end up not checking accurately	Acknowledged
L-5	A re-org may affect ‘SideChainFeeCollector.sol::release()‘	Resolved
L-6	‘unlockingAmount‘ is not updated after the BPT token withdrawal is completed in sPRL2.sol	Resolved
I-1	User might receive less amount then they are willing to accept for ‘withdraw‘ if it clashes with a penalty fee increase	Resolved
I-2	‘updateFeeReceivers‘ should check for duplicate fee receiver entries which will cause reward distribution to be wrong for all recipients	Resolved
I-3	Some rewards may be sent to wrong fee receiver in sPRL2.sol	Resolved
I-4	Underlying rewards that are already sitting in contract may be distributed with the wrong fee distribution ratio	Resolved

4

Findings

4.1 High Risk

A total of 4 high risk findings were identified.

[H-1] sPRL2.sol cannot claim extra reward token because of interface mismatch

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Context:

- sPRL2.sol

Description:

We can go over a live transaction.

This transaction trigger `getReward` on Aura: Base Reward cvxCRV contract.

0xd04e102ba298afa6e9dad526af38633941e7c30d6c8a55a41067c8896c1d812b

We can see the user gets three rewards.

First

From Aura: Base Reward cvxCRV To 0xdE587661...eBdD8f8B0 For 642.179389191386458019 ($1,245.83)

Balancer (BAL)

The BAL reward token is transferred from Aura: Base Reward cvxCRV to the user.

The Aura: Base Reward cvxCRV contract reward token is BAL, the transfer happens in this line of code on chain.

Second

From Null: 0x000...000 To 0xdE587661...eBdD8f8B0 For 1,605.448472978466145047 ($559.67)

Aura (AURA)

The AURA token is minted.

If we follow the on-chain execution path,

- 0x00a7ba8ae7bca0b10a32ea1f8e2a1da980c6cad2#code#F1#L301

```
IDeposit(operator).rewardClaimed(pid, _account, reward);
```

What is operator?

This is the operator contract

- 0xA57b8d98dAE62B26Ec3bcC4a365338157060B234#code#F1#L720

```
function rewardClaimed(uint256 _pid, address _address, uint256 _amount)
    external returns(bool){
    address rewardContract = poolInfo[_pid].crvRewards;
    require(msg.sender == rewardContract || msg.sender == lockRewards,
    "!auth");

    uint256 mintAmount =
     _amount.mul(getRewardMultipliers[msg.sender])
     .div(REWARD_MULTIPLIER_DENOMINATOR);

    if(mintAmount > 0) {
        //mint reward tokens
        ITokenMinter(minter).mint(_address, mintAmount);
    }

    return true;
}
```

The minter mint reward for user.

What is minter?

The minter is the AURA token.

So user receives the AURA token reward.

Third

From:

- 0x27921a5C...830f71555

To:

- 0xdE587661...eBdD8f8B0

For 2,330.627473 ($2,330.41)

- USDC (USDC)

This is the extra reward claiming, on-chain code is here

```
//also get rewards from linked rewards
if(_claimExtras){
    for(uint i=0; i < extraRewards.length; i++){
        IRewards(extraRewards[i]).getReward(_account);
    }
}
```

this address 0x27921a5CC29B11176817bbF5D6bAD83830f71555 is the extraRewards[2], calling `getReward` on the extraRewards[2] transfer user USDC extra reward.

But if we look at the `extra reward` claiming code.

- sPRL2.sol#L248

```
address[] memory extraRewards = AURA_VAULT.extraRewards();
uint256 extraRewardsLength = extraRewards.length;
if (extraRewardsLength > 0) {
    uint256 i;
    for (; i < extraRewardsLength; ++i) {
        IAuraStashToken auraStashToken =

    IAuraStashToken(IVirtualBalanceRewardPool(extraRewards[i]).rewardToken());
        IERC20 extraRewardToken = IERC20(auraStashToken.baseToken());
        uint256 rewardBalance = extraRewardToken.balanceOf(address(this));
        if (rewardBalance > 0) {
            extraRewardToken.safeTransfer(feeReceiver, rewardBalance);
        }
    }
}
```

The code get three extraRewards.

```
IAuraStashToken auraStashToken =
IAuraStashToken(IVirtualBalanceRewardPool(extraRewards[i]).rewardToken());
```

If we query the `extraRewards` at index 0, extra reward contract `extraRewards[i]` is 0x62D7d772b2d909A0779d15299F4FC87e34513c6d#readContract

If the query the rewardToken of the extraRewards[i], the output is 0xA13a9247ea42D743238089903570127DdA72fE44#code

then the auraStashToken is 0xA13a9247ea42D743238089903570127DdA72fE44#code,

```
IERC20 extraRewardToken = IERC20(auraStashToken.baseToken());
```

However, this auraStashToken has no function called `baseToken`, then querying `auraStashToken.baseToken` will revert and block reward claiming.

Recommendations:

If we query the `extraRewards` at index 2, extra reward contract `extraRewards[i]` is https://etherscan.io/address/0x27921a5CC29B111176817bbF5D6bAD83830f71555

the reward token of the extraRewards[i] is USDC

So there is no need to query the `IERC20(auraStashToken.baseToken())`.

The code should be changed to

```
    IERC20 extraRewardToken
    = VirtualBalanceRewardPool(extraRewards[i]).rewardToken();
    uint256 rewardBalance = extraRewardToken.balanceOf(address(this));
    if (rewardBalance > 0) {
        extraRewardToken.safeTransfer(feeReceiver, rewardBalance);
}
```

It is also recommended to do thorough integration fork testing to test that the reward-claiming function works as intended before live deployment.

Parallel: Fixed with @52f78c96f72fc...

Zenith: Verified. Admin can set rewards token array to forward the claimed reward now.

[H-2] CVX rewards cannot be claimed in sPRL2

`SEVERITY:` High	`IMPACT:` High
`STATUS:` Resolved	`LIKELIHOOD:` High

Context:

- sPRL2.sol

Description:

In the `claimRewards` function of `sPRL2`, the `getReward` function of `AURA_VAULT` is called.

- sPRL2.sol#L241

```
function claimRewards() external {
    AURA_VAULT.getReward();
    IERC20 mainRewardToken = IERC20(AURA_VAULT.rewardToken());
}
```

Then it calls `rewardClaimed` function of `BoosterLite`.

- BaseRewardPool.sol#L316

```
function getReward(address _account, bool _claimExtras)
    public updateReward(_account) returns(bool){
    uint256 reward = earned(_account);
    if (reward > 0) {
        rewards[_account] = 0;
        rewardToken.safeTransfer(_account, reward);
        IDeposit(operator).rewardClaimed(pid, _account, reward);
        emit RewardPaid(_account, reward);
    }
    return true;
}
```

In this function, CVX (or other `reward tokens`) are minted to the caller (`sPRL2` in this case).

- BoosterLite.sol#L575

```
function rewardClaimed(
    uint256 _pid,
    address _address,
    uint256 _amount
) external returns (bool) {
    address rewardContract = poolInfo[_pid].crvRewards;
    require(msg.sender == rewardContract, "!auth");

    if (_amount > 0) {
        //mint reward tokens
        ITokenMinter(minter).mint(_address, _amount);
    }

    return true;
}
```

The following is the deployed version.

```
function rewardClaimed(
    uint256 _pid,
    address _address,
    uint256 _amount
) external returns (bool) {
    address rewardContract = poolInfo[_pid].crvRewards;
    require(msg.sender == rewardContract, "!auth");

    if (_amount > 0) {
        // minter = 0xeC1c780A275438916E7CEb174D80878f29580606
        ITokenMinter(minter).mint(_address, _amount);
    }

    return true;
}
```

- contract: 0xeC1c780A275438916E7CEb174D80878f29580606#code

```
function mint(address _to, uint256 _amount) external {
    require(msg.sender == booster, "!booster");
    uint256 _mintRate = mintRate();
    require(_mintRate > 0, "!mintRate");

    uint256 amount = (_amount * _mintRate) / 1e18;

    accBalRewards = accBalRewards.sub(_amount);
    accAuraRewards = accAuraRewards.sub(amount);

    // to = sPRL2
    IERC20(auraOFT).safeTransfer(_to, amount);
}
```

However, `sPRL2` lacks a mechanism to `claim` these reward tokens, making it impossible for users to access them.

Recommendations:

`sPRL2` is designed not to hold any tokens. All tokens are immediately `staked` in `Aura`, and `fees` are transferred to the `fee receiver`. The only exception is unclaimed `CVX reward tokens`. Adding the function below will resolve this issue.

```
function emergencyRescue(address _token, address _to, uint256 _amount)
    external restricted {
    emit EmergencyRescued(_token, _to, _amount);
    IERC20(_token).safeTransfer(_to, _amount);
}
```

Parallel: Fixed in commit: 52f78c96f72fcaf43b0aa4d68f3ad659d20becbc - Refactor claimRewards() that should fix this issue and issue #8

Zenith: Verified.

[H-3] The calculation of withdrawable WETH and PRL amounts in the withdrawPRLAndWeth function of sPRL2 is incorrect

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Context:

- sPRL2.sol

Description:

In the `withdrawPRLAndWeth` function, the `withdrawable WETH` and `PRL` amounts are calculated within the `_exitPool` function.

- sPRL2.sol#L232

```
function withdrawPRLAndWeth(
    uint256[] calldata _requestIds,
    uint256 _minPrlAmount,
    uint256 _minWethAmount
)
    external
    nonReentrant
    returns (uint256 prlAmount, uint256 wethAmount)
{
    (uint256 totalBptAmount, uint256 totalBptAmountSlashed)
    = _withdrawMultiple(_requestIds);
```

```
@->     (prlAmount, wethAmount) = _exitPool(totalBptAmount,
    totalBptAmountSlashed, _minPrlAmount, _minWethAmount);

    emit WithdrawlPRLAndWeth(_requestIds, msg.sender, prlAmount, wethAmount,
    totalBptAmountSlashed);
    PRL.safeTransfer(msg.sender, prlAmount);
    WETH.transfer(msg.sender, wethAmount);
}
```

The `_exitPool` function returns the `WETH` amount as the first value.

- sPRL2.sol#L362

```
function _exitPool(
    uint256 _bptAmount,
    uint256 _bptAmountSlashed,
    uint256 _minPrlAmount,
    uint256 _minWethAmount
)
    internal
@->     returns (uint256 _wethAmount, uint256 _prlAmount)
{
    ...
    _prlAmount = PRL.balanceOf(address(this));
    _wethAmount = WETH.balanceOf(address(this));
}
```

However, in the `withdrawPRLAndWeth` function, this `WETH` amount is incorrectly treated as the `PRL` amount. As a result, the `WETH` and `PRL` amounts are reversed, causing the `withdrawal` to fail.

Recommendations:

```
function _exitPool(
    uint256 _bptAmount,
    uint256 _bptAmountSlashed,
    uint256 _minPrlAmount,
    uint256 _minWethAmount
)
    internal
    returns (uint256 _wethAmount, uint256 _prlAmount)
    returns (uint256 _prlAmount, uint256 _wethAmount)
{
```

```
    ...
    _prlAmount = PRL.balanceOf(address(this));
    _wethAmount = WETH.balanceOf(address(this));
}
```

Parallel: Resolved with This issue has been acknowledged by Parallel Protocol, and a fix was implemented in commit 8b39a4614b91138c75ac116d8528076935911b84.

Zenith: Verified.

[H-4] MainFeeDistributor.sol#swapLzToken function always revert because of interface mismatch

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Context:

- MainFeeDistributor.sol

Description:

Anyone can use the swapLzToken function to swap lz token

```
/// @notice swap Lz-Token to Token if limit not reached.
/// @dev lzToken doesn't need approval to be swapped.
function swapLzToken() external nonReentrant whenNotPaused {
    uint256 balance = bridgeableToken.balanceOf(address(this));
    if (balance == 0) revert NothingToSwap();

    uint256 maxSwapAmount
    = IBridgeableToken(address(bridgeableToken)).getMaxCreditableAmount();
    if (maxSwapAmount == 0) revert MaxSwappableAmountIsZero();

    uint256 swapAmount = balance > maxSwapAmount ? maxSwapAmount : balance;

    emit LzTokenSwapped(swapAmount);
    IBridgeableToken(address(bridgeableToken))
        .swapLzTokenToPrincipalToken(swapAmount);
}
```

However, the bridgeable token contract's swapLzTokenToPrincipalToken function takes two parameter instead of one parameter.

```
function swapLzTokenToPrincipalToken(address _to, uint256 _amount)
    external nonReentrant whenNotPaused {
```

This interface mismatch will revert the function call.

```
IBridgeableToken(address(bridgeableToken))
    .swapLzTokenToPrincipalToken(swapAmount);
```

Recommendations:

```
function swapLzToken() external nonReentrant whenNotPaused restricted {
function swapLzToken(address to) public nonReentrant whenNotPaused
    restricted  {
    uint256 balance = bridgeableToken.balanceOf(address(this));
    if (balance == 0) revert NothingToSwap();
    uint256 maxSwapAmount
    = IBridgeableToken(address(bridgeableToken)).getMaxCreditableAmount();
    if (maxSwapAmount == 0) revert MaxSwappableAmountIsZero();
    uint256 swapAmount = balance > maxSwapAmount ? maxSwapAmount : balance;
    emit LzTokenSwapped(swapAmount);
    - IBridgeableToken(address(bridgeableToken))
        .swapLzTokenToPrincipalToken(swapAmount);
    + IBridgeableToken(address(bridgeableToken))
        .swapLzTokenToPrincipalToken(to, swapAmount);
}
```

Add access control and the address parameter to ensure `swapLzToken` function does not revert.

Parallel: Resolved with [@6b09f03008e4...](https://github.com/parallel-protocol/tokenomics/commit/6b09f03008e4fcf68fbbaa3f49734d05483a7f5d)

Zenith: Verified.

4.2 Medium Risk

A total of 2 medium risk findings were identified.

[M-1] The SideChainFeeCollector does not work correctly when the fee token has decimals other than 18

SEVERITY: Medium	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: Medium

Context:

- SideChainFeeCollector.sol

Description:

The gathered `fee tokens` in `SideChainFeeCollector` can be transferred to the `Main Chain` using `bridgeableToken`. The shared decimals in `OFT` are `6`, and `BRIDGEABLE_CONVERSION_DECIMALS` is set to `1e12`.

- SideChainFeeCollector.sol#L21

```
uint256 private constant BRIDGEABLE_CONVERSION_DECIMALS = 1e12;
```

This conversion only works correctly when the `fee token` has `18 decimals`.

- If the `fee token`'s `decimals` are greater than `18`, some tokens will be lost during conversion in the `bridgeableToken`.
- If the `decimals` are less than `18`, the transferrable amount calculation is incorrect.
  - SideChainFeeCollector.sol#L129

```
function _calcBridgeableAmount() private view returns (uint256) {
    uint256 feeTokenBalance = feeToken.balanceOf(address(this));
    return (feeTokenBalance / BRIDGEABLE_CONVERSION_DECIMALS)
    * BRIDGEABLE_CONVERSION_DECIMALS;
}
```

Recommendations:

Ensure that all `fee tokens` have `18 decimals`, or adjust `BRIDGEABLE_CONVERSION_DECIMALS` dynamically based on the token's `decimals`.

Parallel: Acknowledged. We are aware of this and accept it. The tokens that will be bridge is the PAR which is a 18 decimals and part of our protocol.

- Documentation

Zenith: Acknowledged

[M-2] Migrating PRL tokens to another chain may fail due to insufficient gas

`SEVERITY`: Medium	`IMPACT`: Medium
`STATUS`: Resolved	`LIKELIHOOD`: Medium

Context:

- PrincipalMigrationContract.sol

Description:

Users can migrate `MIMO` tokens from `Chain A` to `PRL` tokens on `Chain B` through the `Main Chain`. They initiate the `migration` by calling the `migrateToPRL` function with the appropriate parameters in `PeripheralMigrationContract`.

- PeripheralMigrationContract.sol#L87

```
/// @notice Migrates MIMO tokens to PRL tokens on another chain
/// @param _receiver The address that will receive the PRL tokens
/// @param _amount The amount of MIMO tokens to migrate
/// @param _dstEid The destination endpoint ID
/// @param _extraSendOptions Gas settings for A → Main Chain
/// @param _extraReturnOptions Gas settings for Main Chain → Final chain
/// @return msgReceipt The receipt of the LayerZero message
function migrateToPRL(
    address _receiver,
    uint256 _amount,
    uint32 _dstEid,
    address _refundAddress,
```

```
    bytes calldata _extraSendOptions,
    bytes calldata _extraReturnOptions
)
    external
    payable
    whenNotPaused
    returns (MessagingReceipt memory msgReceipt)
{
    if (_refundAddress == address(0)) revert ErrorsLib.AddressZero();
    uint256 fee = msg.value;
    bytes memory options =
        combineOptions(_dstEid, _extraSendOptions.length > 0 ?
    SEND_AND_MIGRATE : SEND, _extraSendOptions);

    MIMO.safeTransferFrom(msg.sender, address(this), _amount);
    msgReceipt = _lzSend(
        mainEid,
        _encodeMessage(_receiver, _amount, _dstEid, _extraReturnOptions),
        options,
        // Fee in native gas and ZRO token.
        MessagingFee(msg.value, 0),
        // Refund address in case of failed source message.
        _refundAddress
    );
    emit MigrationMessageSent(msgReceipt.guid, _dstEid, msg.sender,
    _receiver, fee, _amount);
}
```

As noted in the comments, `_extraReturnOptions` is used to set `gas` options for the `migration` from the `Main Chain` to `Chain B`. This `_extraReturnOptions` is included in the transferred message.

Ultimately, the `_send` function of `EndPointV2` is called, which triggers the `send` function of the `send` library.

- EndpointV2.sol#L133

```
function _send(
    address _sender,
    MessagingParams calldata _params
) internal returns (MessagingReceipt memory, address) {

    address _sendLibrary = getSendLibrary(_sender, _params.dstEid);

    // messageLib always returns encodedPacket with guid
    (MessagingFee memory fee, bytes memory encodedPacket)
```

```
    = ISendLib(_sendLibrary).send(
        packet,
        _params.options,
        _params.payInLzToken
    );
}
```

While the `message length` is considered for `fee calculation`, there is no mechanism to account for the gas required for the final `migration` from the `Main Chain` to `Chain B`.

- SendLibBaseE2.sol#L120-L122

```
function _payWorkers(
    Packet calldata _packet,
    bytes calldata _options
) internal returns (bytes memory encodedPacket, uint256 totalNativeFee) {
    // split workers options
    (bytes memory executorOptions, WorkerOptions[] memory validationOptions)
    = _splitOptions(_options);

    // handle executor
    ExecutorConfig memory config = getExecutorConfig(_packet.sender,
    _packet.dstEid);
    uint256 msgSize = _packet.message.length;
    _assertMessageSize(msgSize, config.maxMessageSize);
    totalNativeFee += _payExecutor(config.executor, _packet.dstEid,
    _packet.sender, msgSize, executorOptions);

    // handle other workers
    (uint256 verifierFee, bytes memory packetBytes) = _payVerifier(_packet,
    validationOptions); //for ULN, it will be dvns
    totalNativeFee += verifierFee;

    encodedPacket = packetBytes;
}
```

In the `_lzReceive` function of the `PrincipalMigrationContract`, if `extraOptionsLength` is not zero, the `send` function of `LockBox` is called, using `msg.value` as gas.

- PrincipalMigrationContract.sol#L209

```
function _lzReceive(
    Origin calldata _origin,
    bytes32 _guid,
    bytes calldata _message,
    address, /*_executor*/ // @dev unused in the default implementation.
```

```
    bytes calldata /*_extraData*/ // @dev unused in the default
    implementation.
)
    internal
    override
{
    if (extraOptionsLength == 0 || destEid == endpoint.eid()) {
    } else {
        SendParam memory sendParam = _buildSendParam(_message, receiver,
    amount, destEid, extraOptionsLength);
        emit MIMOToPRLMigratedAndBridged(
            _origin.sender.bytes32ToAddress(), receiver, sendParam,
    MessagingFee(msg.value, 0)
        );
        PRL.approve(address(lockBox), amount);
        lockBox.send{ value: msg.value }(sendParam, MessagingFee(msg.value,
    0), receiver);
    }
}
```

However, there is no guarantee that the provided `gas` will be sufficient.

`PRL` tokens should be minted on the `Main Chain` at least, in case `migration` to `Chain B` is not possible. This can be handled later using `LockBox`.

Recommendations:

In the `_lzReceive` function, if the provided `msg.value` is less than the `quoted fee value` of the `LockBox`, the `PRL` tokens can be minted directly on the `Main Chain`.

Parallel: Resolved with @99d668176821...

Zenith: Verified.

4.3 Low Risk

A total of 6 low risk findings were identified.

[L-1] The epoch should only be updated once

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Context:

- RewardMerkleDistributor.sol

Description:

There is no check to verify whether the `epoch` has already been updated.

- RewardMerkleDistributor.sol#L180

```
function updateMerkleDrop(uint64 _epoch, MerkleDrop memory _merkleDrop)
    external restricted {
    if (_merkleDrop.totalAmount > TOKEN.balanceOf(address(this)))
    revert NotEnoughRewards();
    if (_merkleDrop.expiryTime - _merkleDrop.startTime < EPOCH_LENGTH)
    revert EpochExpired();
    merkleDrops[_epoch] = _merkleDrop;
    emit MerkleDropUpdated(
        _epoch, _merkleDrop.root, _merkleDrop.totalAmount,
    _merkleDrop.startTime, _merkleDrop.expiryTime
    );
}
```

Since `totalClaimedPerEpoch` and `hasClaimed` are directly tied to the `epoch`, updating the `epoch` multiple times could break the system. Due to this being a protected operation, the likelihood of such an issue occurring is extremely low.

Recommendations:

```
function updateMerkleDrop(uint64 _epoch, MerkleDrop memory _merkleDrop)
    external restricted {
    if (_merkleDrop.totalAmount > TOKEN.balanceOf(address(this)))
    revert NotEnoughRewards();
    if (_merkleDrop.expiryTime - _merkleDrop.startTime < EPOCH_LENGTH)
    revert EpochExpired();

+^^Iif (merkleDrops[_epoch].expiryTime > 0) revert();
```

```
    merkleDrops[_epoch] = _merkleDrop;
    emit MerkleDropUpdated(
        _epoch, _merkleDrop.root, _merkleDrop.totalAmount,
    _merkleDrop.startTime, _merkleDrop.expiryTime
    );
}
```

Parallel: The issue has been resolved with @b1ce781f60bf.... After internal discussion we only allow the caller to update epoch that didn't started (that way no user rewards will be impacted) and block gaps between epochs (doing 1,3,4 is not possible as epoch 2 missing).

Zenith: Verified.

[L-2] Paying fees with LzToken is not allowed in the migrateToPRLAndBridge function of the PrincipalMigrationContract

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Context:

- PrincipalMigrationContract.sol

Description:

Users might attempt to pay the `fee` using `LzToken`.

- PrincipalMigrationContract.sol#L124-L128

```
function migrateToPRLAndBridge(
    SendParam calldata _sendParam,
    MessagingFee calldata _fee,
    address _refundAddress
)
    external
    payable
    whenNotPaused
    nonReentrant
```

```
{
    if (_refundAddress == address(0)) revert ErrorsLib.AddressZero();
    emit MIMOToPRLMigratedAndBridged(msg.sender,
    _sendParam.to.bytes32ToAddress(), _sendParam, _fee);
    MIMO.safeTransferFrom(msg.sender, address(this), _sendParam.amount);
    PRL.approve(address(lockBox), _sendParam.amount);
@->     lockBox.send{ value: msg.value }(_sendParam, _fee, _refundAddress);
}
```

However, because they don't transfer the `LzToken` to the `PrincipalMigrationContract`, the `_payLzToken` function will revert.

- OAppSender.sol#L83

```
function _lzSend(
    uint32 _dstEid,
    bytes memory _message,
    bytes memory _options,
    MessagingFee memory _fee,
    address _refundAddress
) internal virtual returns (MessagingReceipt memory receipt) {
    // @dev Push corresponding fees to the endpoint, any excess is sent back
    to the _refundAddress from the endpoint.
    uint256 messageValue = _payNative(_fee.nativeFee);
    if (_fee.lzTokenFee > 0) _payLzToken(_fee.lzTokenFee);
}

function _payLzToken(uint256 _lzTokenFee) internal virtual {
    // @dev Cannot cache the token because it is not immutable in the
    endpoint.
    address lzToken = endpoint.lzToken();
    if (lzToken == address(0)) revert LzTokenUnavailable();

    // Pay LZ token fee by sending tokens to the endpoint.
@->     IERC20(lzToken).safeTransferFrom(msg.sender, address(endpoint),
    _lzTokenFee);
}
```

Recommendations:

Users should not be allowed to pay fees with `LzToken`.

```
function migrateToPRLAndBridge(
    SendParam calldata _sendParam,
```

```
    MessagingFee calldata _fee,
    address _refundAddress
)
    external
    payable
    whenNotPaused
    nonReentrant
{
    if (_refundAddress == address(0)) revert ErrorsLib.AddressZero();

+^^Iif (_fee.lzTokenFee > 0) revert();

    emit MIMOToPRLMigratedAndBridged(msg.sender,
    _sendParam.to.bytes32ToAddress(), _sendParam, _fee);
    MIMO.safeTransferFrom(msg.sender, address(this), _sendParam.amount);
    PRL.approve(address(lockBox), _sendParam.amount);
    lockBox.send{ value: msg.value }(_sendParam, _fee, _refundAddress);
}
```

Parallel: Resolved with @aec7f14d7c9c..

Zenith: Verified

[L-3] Consider use block.timestamp for ERC20Vote token clock mode

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Context:

- TimeLockPenaltyERC20.sol

Description:

Both sPRL1.sol and sPRL2.sol inherit from TimeLockPenaltyERC20.sol

and TimeLockPenaltyERC20.sol inherit from ERC20Votes.sol,

By default, the clock mode is block.number instead of block.timestamp

```
/**
 * @dev Machine-readable description of the clock as specified in ERC-6372.
 */
// solhint-disable-next-line func-name-mixedcase
function CLOCK_MODE() public view virtual returns (string memory) {
    // Check that the clock was not modified
    if (clock() != Time.blockNumber()) {
        revert ERC6372InconsistentClock();
    }
    return "mode=blocknumber&from=default";
}
```

However, consider that the protocol intends to deploy in side chain / non-ethereum chain, the block.number may not be a reliable measure of proposal time.

For example,

- block-numbers-and-time#arbitrum-block-numbers

> Accessing block numbers within an Arbitrum smart contract (i.e., block.number in Solidity) will return a value close to (but not necessarily exactly) the L1 block number at which the sequencer received the transaction.

Recommendations:

Override the CLOCK_MODE function and use block.timestamp as governance proposal time measurement.

```
function CLOCK_MODE() public view virtual override returns (string memory) {
    return "mode=timestamp&from=default"; // Indicate time-based measurement
}

/// @notice Override clock() function to return block.timestamp
function clock() public view virtual override returns (uint256) {
    return block.timestamp; // Return current timestamp
}
```

Parallel: Resolved with @96770d928cea7...

Zenith: Verified.

[L-4] `updateMerkleDrop`'s safety check might end up not checking accurately

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Context:

- RewardMerkleDistributor.sol

Description:

```
function updateMerkleDrop(uint64 _epoch, MerkleDrop memory _merkleDrop)
    external restricted {
-->   if (_merkleDrop.totalAmount > TOKEN.balanceOf(address(this)))
    revert NotEnoughRewards();
      if (_merkleDrop.expiryTime - _merkleDrop.startTime < EPOCH_LENGTH)
    revert EpochExpired();
      merkleDrops[_epoch] = _merkleDrop;
      emit MerkleDropUpdated(
            _epoch, _merkleDrop.root, _merkleDrop.totalAmount,
    _merkleDrop.startTime, _merkleDrop.expiryTime
      );
}
```

The line `if (_merkleDrop.totalAmount > TOKEN.balanceOf(address(this))) revert NotEnoughRewards()` ensures that the balance of the merkle distributor is sufficient to cover the new `totalAmount` that is expected to be distributed soon.

However, the check is not as accurate as it seems, as `TOKEN.balanceOf(address(this))` does include other stuff. To be specific, these are the 2 reasons why the balanceOf is not a good indicator:

1. The amounts meant to be distributed from the **current already running epoch** might still be in the contract. Since `updateMerkleDrop` is called to distribute rewards for an **upcoming epoch**, the rewards for the **current** epoch (that is **not** expired yet) are still "distributing" and some users have yet to claim it.

2. Rewards from **expired** epochs might still be in the contract. (Meaning they **have not been forwarded by** *`forwardExpiredRewards`* **yet**, which is very likely if the previous epoch has just recently expired)

Recommendations:

Add one new storage variable for tracking:

```
uint256 accounted = 0;
 function updateMerkleDrop(uint64 _epoch, MerkleDrop memory _merkleDrop)
    external restricted {
    if (_merkleDrop.totalAmount > TOKEN.balanceOf(address(this))) revert
        NotEnoughRewards();

    if (_merkleDrop.totalAmount > TOKEN.balanceOf(address(this)) - accounted)
        revert NotEnoughRewards();
     if (_merkleDrop.expiryTime - _merkleDrop.startTime < EPOCH_LENGTH)
    revert EpochExpired();
     merkleDrops[_epoch] = _merkleDrop;
    accounted += _merkleDrop.totalAmount;
     emit MerkleDropUpdated(
          _epoch, _merkleDrop.root, _merkleDrop.totalAmount,
    _merkleDrop.startTime, _merkleDrop.expiryTime
     );
 }
```

Parallel:

Acknowledged.

[L-5] A re-org may affect `SideChainFeeCollector.sol::release()`

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Context:

- SideChainFeeCollector.sol

Description:

In SideChainFeeCollector.sol:

```
function release(bytes memory _options)
    external
    payable
    nonReentrant
    whenNotPaused
    restricted
    returns (uint256 amountSent)
{
    ....
    SendParam memory sendParam = SendParam(
        .....
        OFTMsgCodec.addressToBytes32(destinationReceiver),
        .....
    );
    .....
    bridgeableToken.send{ value: msg.value }(sendParam,
    MessagingFee(msg.value, 0), payable(msg.sender));
}

function updateDestinationReceiver(address _newDestinationReceiver)
    external restricted {
    destinationReceiver = _newDestinationReceiver;
    emit DestinationReceiverUpdated(_newDestinationReceiver);
}

function updateBridgeableToken(address _newBridgeableToken)
    external restricted {
    bridgeableToken = IOFT(_newBridgeableToken);
    emit BridgeableTokenUpdated(_newBridgeableToken);
}
```

As shown, `release` uses `destinationReceiver` and `bridgeableToken` which are both storage variables of the contract (they are both not passed into the function).

Hence, there can be a senario where `updateDestinationReceiver` or `updateBridgeableToken` are called and then shortly after the admin decides to call `release()`. Hence initial order:

1. `updateDestinationReceiver` or `updateBridgeableToken`

(b) `release()`

After re-org the transaction orders could be changed:

1. `release()`

   (b) `updateDestinationReceiver` or `updateBridgeableToken`

This results in release()'s `.send` to be sent to the old destinationReceiver / old bridgeableToken which may or may not be supported anymore.

Recommendations:

Allow the caller to specify `destinationReceiver` and `bridgeableToken` in the function parameters of `release()` then after that `require(contract's destinationReceiver == caller's passed in destinationReceiver)` (and so-on for `bridgeableToken` as well)

Parallel: Resolved with @a7c7d13ba675...

Zenith: Verified.

[L-6] `unlockingAmount` is not updated after the BPT token withdrawal is completed in sPRL2.sol

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Context:

- sPRL2.sol

Description:

When user create withdraw, the code uses variable `unlockingAmount` to track locking amount in TimeLockPenaltyERC20.sol.

```
request.amount = _unlockingAmount;
request.requestTime = uint64(block.timestamp);
request.releaseTime = uint64(block.timestamp) + timeLockDuration;
request.status = WITHDRAW_STATUS.UNLOCKING;

unlockingAmount += _unlockingAmount;
```

In sPRL1.sol `withdraw` function, the `unlockingAmount` is updated when the withdraw is

completed.

```
function withdraw(uint256[] calldata _ids) external nonReentrant {
      (uint256 totalAmountWithdrawn, uint256 totalFeeAmount)
    = _withdrawMultiple(_ids);
      unlockingAmount = unlockingAmount - totalAmountWithdrawn
    - totalFeeAmount;
      if (totalFeeAmount > 0) {
          underlying.safeTransfer(feeReceiver, totalFeeAmount);
      }
      underlying.safeTransfer(msg.sender, totalAmountWithdrawn);
  }
```

However, in sPRL2.sol, the `unlockingAmount` variable is not updated after _withdrawMultiple.

The impact is that the unlock amount will always be inflated and off-chain services that query the `unlockingAmount` state will get a inflated amount that is larger than the actual value.

Recommendations:

In TimeLockPenaltyERC20.sol#_withdrawMultiple,

```
    function _withdrawMultiple(uint256[] calldata _ids)
        internal
        returns (uint256 totalAmountWithdrawn, uint256 totalSlashAmount)
    {
        uint256 i = 0;
        for (; i < _ids.length; ++i) {
            (uint256 amountWithdrawn, uint256 slashAmount)
    = _withdraw(_ids[i]);
            totalAmountWithdrawn += amountWithdrawn;
            totalSlashAmount += slashAmount;
        }

    unlockingAmount = unlockingAmount - totalAmountWithdrawn - totalSlashAm
          ount;

    }
```

In sPRL1.sol#withdraw function,

```
function withdraw(uint256[] calldata _ids) external nonReentrant {
      (uint256 totalAmountWithdrawn, uint256 totalFeeAmount)
  = _withdrawMultiple(_ids);

  unlockingAmount = unlockingAmount - totalAmountWithdrawn - totalFeeAmo
        unt;
      if (totalFeeAmount > 0) {
          underlying.safeTransfer(feeReceiver, totalFeeAmount);
      }
      underlying.safeTransfer(msg.sender, totalAmountWithdrawn);
  }
```

Parallel: Resolved with @b5e07eba9fff...

Zenith: Verified.

4.4 Informational

A total of 4 informational findings were identified.

[I-1] User might receive less amount then they are willing to accept for `withdraw` if it clashes with a penalty fee increase

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Context:

- sPRL1.sol
- TimeLockPenaltyERC20.sol

Description:

sPRL1.sol's `withdraw` calls `TimeLockPenaltyERC20.sol`'s `_withdrawMultiple`

```
function withdraw(uint256[] calldata _ids) external nonReentrant {
```

```
    (uint256 totalAmountWithdrawn, uint256 totalFeeAmount)
    = _withdrawMultiple(_ids);
    if (totalFeeAmount > 0) {
        underlying.safeTransfer(feeReceiver, totalFeeAmount);
    }
    underlying.safeTransfer(msg.sender, totalAmountWithdrawn);
}
```

`_withdrawMultiple` then calls `_withdraw` which calculates the slashed amount via `_calculateFee`

```
function _calculateFee(
    uint256 _amount,
    uint256 _requestTime,
    uint256 _releaseTime
)
    internal
    view
    returns (uint256 feeAmount)
{
    if (block.timestamp >= _releaseTime) return 0;
    uint256 timeLeft = _releaseTime - block.timestamp;
    uint256 lockDuration = _releaseTime - _requestTime;
    uint256 feePercentage = startPenaltyPercentage.mulDivUp(timeLeft,
    lockDuration);
    feeAmount = _amount.wadMulUp(feePercentage);
}
```

We can see that the code protects the user against changing lock duration by calculating the lock duration as `_releaseTime - _requestTime`, hence its the fixed duration that was calculated during the initiation of the locking.

However it uses `startPenaltyPercentage` which can be changed by `updateStartPenaltyPercentage`. So if there is a **clash between** `withdraw` and `updateStartPenaltyPercentage` (meaning, the user calls `withdraw` and around the same time, the admin calls `updateStartPenaltyPercentage` to do a normal fee increase) then the user may end up receiving way **less assets than they are willing to accept** due to the lack of slippage protection.

It is important to have a slippage protection in this case as if the penalties are increased, the user **should have a choice to wait longer to withdraw** in order to **not** incur/incur less of the penalty if the user deems the time waiting trade-off is now more worth than the increased penalty. (**which is especially true if the user is withdrawing a large amount**)

- Hence, it should **revert** if it is a value that the user cannot accept, so that the user has the choice to wait longer instead, which will incur less of the penalty since it is multiple by the `timeLeft` that the user wishes to skip

Recommendations:

You could create 2 functions with the exact same name, where one will have a default value for the slippage parameter so that users who are withdrawing large amounts can withdraw safely without having to worry about clashing with a routine penalty fee increase.

```
function withdraw(uint256[] calldata _ids) external {
    withdraw(_ids, type(uint256).max);
}
function withdraw(uint256[] calldata _ids, uint256 maxAcceptablePenalty)
    external nonReentrant {
    .....
    // normal withdraw function of sPRL1.sol (+ require(penalty ⩽
    maxAcceptablePenalty))
    ....
}
```

Parallel: Resolved with @87c3b18dedf1...

Zenith: Verified.

[I-2] `updateFeeReceivers` should check for duplicate fee receiver entries which will cause reward distribution to be wrong for all recipients

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Context:

- MainFeeDistributor.sol

Description:

MainFeeDistributor.sol decides each individual recipient's reward like this:

```
uint256 amount = _totalIncomeToDistribute * shares[_feeReceiver]
    / totalShares;
```

That is correct so as long as `sum{shares[each receiver]} = totalShares` then the

distribution will work perfectly fine.

Looking at `updateFeeReceivers`:

```
function updateFeeReceivers(address[] memory _feeReceivers, uint256[]
    memory _shares) public restricted {
      if (_feeReceivers.length == 0) revert NoFeeReceivers();
      if (_feeReceivers.length != _shares.length)
    revert ArrayLengthMismatch();
      delete feeReceivers;

      uint256 _totalShares = 0;
      uint256 i = 0;
      for (; i < _feeReceivers.length; ++i) {
          _totalShares += _addFeeReceiver(_feeReceivers[i], _shares[i]);
      }
      totalShares = _totalShares;
}
function _addFeeReceiver(address _feeReceiver, uint256 _shares)
    internal returns (uint256) {
      if (_feeReceiver == address(0)) revert FeeReceiverZeroAddress();
      if (_shares == 0) revert SharesIsZero();

      feeReceivers.push(_feeReceiver);
-->   shares[_feeReceiver] = _shares;
      emit FeeReceiverAdded(_feeReceiver, _shares);
      return _shares;
}
```

Focusing on the line pointed out by the arrow, we can see that it simply overwrites the current value. This means that if there are duplicate fee receiver values (caller could have accidentally inputted the same array entry twice, Low likelihood though) for example if the correct intention is for 3 recipients to receive 1/3 share each and this happens:

- A, 1 shares
- B, 1 shares
- B, 1 shares
- C, 1 shares

Then *`totalShares`* will be 4, instead of 3. And when `release` is called:

- A will get 1/4 instead of 1/3
- B will get 2/4 (as the `feeReceivor` array pushed B in twice, causing internal func `_release(B)` to be ran twice)
- C will get 1/4 instead of 1/3

Recommendations:

It is common for such functions to put in place a duplicate check to prevent such things for happening. (Or alternatively document for callers to be very careful not to include duplicate entries)

Parallel: Resolved with @394fd72f8559d1e...

Zenith: Verified.

[I-3] Some rewards may be sent to wrong fee receiver in sPRL2.sol

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Context:

- sPRL2.sol
- TimeLockPenaltyERC20.sol

Description:

`sPRL2.sol` inherits `TimeLockPenaltyERC20.sol` and uses its functions for storing and changing of `feeReceiver`.

Notably this is the function in `TimeLockPenaltyERC20.sol` that allows `sPRL2.sol` to change `feeReceiver`:

```
function updateFeeReceiver(address _newFeeReceiver) external restricted {
    emit FeeReceiverUpdated(_newFeeReceiver);
    feeReceiver = _newFeeReceiver;
}
```

Looking at where `feeReceiver` is used in `sPRL2.sol`, we can see that:

```
function claimRewards() external {
    AURA_VAULT.getReward();
    IERC20 mainRewardToken = IERC20(AURA_VAULT.rewardToken());
    uint256 mainRewardBalance = mainRewardToken.balanceOf(address(this));
    if (mainRewardBalance > 0) {
```

```
            mainRewardToken.safeTransfer(feeReceiver, mainRewardBalance);
        }

        address[] memory extraRewards = AURA_VAULT.extraRewards();
        uint256 extraRewardsLength = extraRewards.length;
        if (extraRewardsLength > 0) {
            uint256 i;
            for (; i < extraRewardsLength; ++i) {
                IAuraStashToken auraStashToken =

        IAuraStashToken(IVirtualBalanceRewardPool(extraRewards[i]).rewardToken());
                IERC20 extraRewardToken = IERC20(auraStashToken.baseToken());
                uint256 rewardBalance
        = extraRewardToken.balanceOf(address(this));
                if (rewardBalance > 0) {
                    extraRewardToken.safeTransfer(feeReceiver, rewardBalance);
                }
            }
        }
    }
```

It is used as the recipient in `claimRewards` to receive the rewards from staking in aura vault. Hence, the code should have sent **current accumulated rewards to the current** *`feeReceiver`* **before updating to the new** *`feeReceiver`*.

Since the rewards were accumulated during the "era" of the current `feeReceiver` it makes more sense to send them to the current `feeReceiver` than the new `feeReceiver`.

Recommendations:

Override `updateFeeReceiver` to make a call to `claimReward` before calling `super.updateFeeReceiver`.

```
function updateFeeReceiver(address _newFeeReceiver)
    external override restricted { // override function
    claimRewards();
    super.updateFeeReceiver(_newFeeReceiver); //call the original one after
    claiming rewards
}
```

Parallel: Resolved with @ad7381ebd1dc...

Zenith: Verified.

[I-4] Underlying rewards that are already sitting in contract may

be distributed with the wrong fee distribution ratio

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Medium

Context:

- MainFeeDistributor.sol

Description:

```
function updateFeeReceivers(address[] memory _feeReceivers, uint256[]
    memory _shares) public restricted {
    if (_feeReceivers.length == 0) revert NoFeeReceivers();
    if (_feeReceivers.length ≠ _shares.length)
    revert ArrayLengthMismatch();
    delete feeReceivers;

    uint256 _totalShares = 0;
    uint256 i = 0;
    for (; i < _feeReceivers.length; ++i) {
        _totalShares += _addFeeReceiver(_feeReceivers[i], _shares[i]);
    }
    totalShares = _totalShares;
}
```

We can see that `updateFeeReceivers` does not call `swapLzToken()` and `release()` which means that if there are underlying reward tokens in the contract, they would be distributed with the wrong ratio and possibly to the wrong new fee recipients.

This is because since these rewards **were accumulated during the "rules" of the old ratio split and recipients**, so they **should be distributed according to the current ratio** rather than the new ratio that just took effect.

Recommendations:

Adding `swapLzToken()` and `release()` at the front of `updateFeeReceivers` will fix this issue and ensure that underlying rewards are cleared with the ratio that was supposed to be the "rules" for those rewards.

Parallel: Resolved with @db2707afff7b82...

Zenith: Verified.