Audit Report For:

MALDA

Prepared by: yum-cutty

September 30, 2025

Contents

Protocol Summary

Malda is a Unified Liquidity Lending protocol on Ethereum and Layer 2s, delivering a seamless lending experience through global liquidity pools, all secured by zkProofs.

View Contest Page

Disclaimer

yum-cutty makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by yum-cutty is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use CodeHawks severity matrix to determine severity. See documentation for more details.

Audit Details Result

Scope

```
src/
├── blacklister/
│   └── Blacklister.sol (51)
│
├── interest/
│   └── JumpRateModelV4.sol (64)
│
├── migration/
│   ├── IMigrator.sol (4)
│   └── Migrator.sol (137)
│
├── mToken/
│   ├── BatchSubmitter.sol (147)
│   ├── extension/
│   │   └── mTokenGateway.sol (199)
│   ├── host/
│   │   ├── mErc20Host.sol (255)
│   │   ├── mErc20Immutable.sol (20)
│   │   ├── mErc20.sol (56)
│   │   └── mErc20Upgradable.sol (14)
│   ├── mTokenConfiguration.sol (55)
│   ├── mToken.sol (272)
│   └── mTokenStorage.sol (123)
│
├── Operator/
│   ├── Operator.sol (459)
│   └── OperatorStorage.sol (89)
│
├── oracles/
│   ├── gas/
│   │   └── DefaultGasHelper.sol (11)
│   ├── MixedPriceOracleV3.sol (80)
│   └── MixedPriceOracleV4.sol (124)
│
├── pauser/
│   └── Pauser.sol (95)
│
├── rebalancer/
│   ├── bridges/
│   │   ├── AcrossBridge.sol (96)
│   │   ├── BaseBridge.sol (21)
│   │   └── EverclearBridge.sol (88)
│   ├── Rebalancer.sol (103)
│   └── Roles.sol (30)
│
├── utils/
│   └── WrapAndSupply.sol (39)
│
```

```
└── verifier/
    └── ZkVerifier.sol (40)
```

Issues found

Severity	Number of Issues Found
High	1
Medium	15
Low	0
Informational	0
Gas	0
Total	**16**

Findings

High

[H-1] lack of receiver validation for rebalancing operation

Description:

`Rebalancer::sendMsg` initiates the cross-chain rebalancing operation.

```
    function sendMsg(
        address _bridge,
        address _market,
        uint256 _amount,
@>      Msg calldata _msg
    ) external payable {
        // ...
@>      require(whitelistedDestinations[_msg.dstChainId], ...());

        // check underlying
        address _underlying = ImTokenMinimal(_market).underlying();
@>      require(_underlying == _msg.token, ...());

        // min transfer size check
@>      require(_amount > minTransferSizes[_msg.dstChainId][_msg.token], ...());

        // max transfer size checks
        // struct TransferInfo { uint256 size; uint256 timestamp; }
        TransferInfo memory transferInfo = currentTransferSize[_msg.dstChainId][_msg.token];

        // ...
        uint256 _maxTransferSize = maxTransferSizes[_msg.dstChainId][_msg.token];
        if (_maxTransferSize > 0) {
@>          require(transferInfo.size + _amount < _maxTransferSize, ...());
        }
        // ...
        // approve and trigger send
        SafeApprove.safeApprove(_msg.token, _bridge, _amount);
        IBridge(_bridge).sendMsg{value: msg.value}(
           _amount, _market, _msg.dstChainId, _msg.token, _msg.message, _msg.bridgeData
        );
        // ...
    }
```

then internally calls `EverclearBridge::sendMsg`

```
      function sendMsg(
          uint256 _extractedAmount,
          address _market,
          uint32 _dstChainId,
          address _token,
@>        bytes memory _message,
          bytes memory // unused
      ) external payable onlyRebalancer {
          IntentParams memory params = _decodeIntent(_message);

@>        require(params.inputAsset == _token, Everclear_TokenMismatch());
@>        require(_extractedAmount >= params.amount, BaseBridge_AmountMismatch());
          // ...

          SafeApprove.safeApprove(...);
          (bytes32 id,) = everclearFeeAdapter.newIntent(
              params.destinations,
@>            params.receiver,
              params.inputAsset,
              params.outputAsset,
              params.amount,
              params.maxFee,
              params.ttl,
              params.data,
              params.feeParams
          );
          emit MsgSent(_dstChainId, _market, params.amount, id);
      }
```

as a side note, below is the `Msg` struct

```
    struct Msg {
        uint32 dstChainId;
        address token;
        bytes message;
        bytes bridgeData;
    }
```

`Rebalancer::sendMsg` function validates `Msg` arguments, follow by a furthur validation on the `Msg.message` in `EverclearBridge::sendMsg`.

however, we notice that there's no validation on the `params.receiver` address.

> Rebalancer is semi-trusted - Rebalancer only has DDOS abilities for the protocol via constant rebalancing.

as stated in the README, Rebalancer user role are semi-trusted. without a validation on `params.receiver` address, it puts an extra layer of risk that the protocol funds will ended up at nowhere other than the expected receiver.

Impact:

a malicious rebalancer role can spoof an encoded `params.receiver` to drain/steal the protocol funds.

Exploitation:

since there's no validation on `params.receiver` in `EverclearBridge::sendMsg` function, a malicious actor with rebalancer role can pass in with an encoded custom `params.receiver` address, such as their own address, which will ended up stealing all the funds that are meant for cross-chain rebalancing operation.

Recommended Mitigation:

add an extra validation checks on the `params.receiver` to ensures it is the market address, where it is expected to be transfered to, rather than allowing custom address that left unchecked before initiate the operation.

```
+   require(params.receiver == _market, ...());

    SafeApprove.safeApprove(...);
    (bytes32 id,) = everclearFeeAdapter.newIntent(
        params.destinations,
        params.receiver,
        params.inputAsset,
        params.outputAsset,
        params.amount,
        params.maxFee,
        params.ttl,
        params.data,
        params.feeParams
    );
```

Medium

[M-1] drainage in redemption due to incorrect rounding

Description:

`mToken::__redeem()` function involve calculating the amount of underlying to redeem, based on the specified `mToken` amount to burn, or vice versa.

The redeem calculation in this function `__redeem()` is as follows:

```
    // note
    // get exchange rate
    Exp memory exchangeRate = Exp({ mantissa: _exchangeRateStored() });

    uint256 redeemTokens;
    if (redeemTokensIn > 0) {
        // note
        // user specify amount of mTokens to burn
        // calculate underlying to return back to user
        redeemTokens = redeemTokensIn;
        redeemAmount = mul_ScalarTruncate(exchangeRate, redeemTokensIn);
    } else {
        // note
        // user specify amount of underlying to get back
        // calculate mTokens to burn
@>      redeemTokens = div_(redeemAmountIn, exchangeRate);
        redeemAmount = redeemAmountIn;
    }
```

The helper math calculation formula is as follows:

```
div_()               --> redeemAmountIn * 1e18 / exchangeRate
mul_ScalarTruncate() --> exchangeRate * redeemTokensIn / 1e18
```

The main issue is in `div_()` as it rounds down the operations

Impact:

Due to the fact that division in solidity does not contain any decimals, the result of an operation will always be in integer, any decimals number will be ignored and truncated.

the `div()` function rounding down, may potentially burn `1 token` less than it should, while still being able to redeem the full expected underlying amount.

Losing `1 token` might not be a big deal. However, if the `exchangeRate` is large enough to be greater than `redeemAmountIn * 1e18`, user is able to claim underlying for free, without burning any tokens in return.

Exploitation:

A hacker can slowly inflates the interest rate, potentially lure more depositor with the high interest rate, then hacker starts withdraw.

Step #1:

if hacker is the first depositor, at least 20 wei should be donated to bypass the bootstrap as seen in `mToken::__mint`

```
    // note
    // get the actual underlying tokens sent by the user
    uint256 actualMintAmount = doTransfer
                    ? _doTransferIn(minter, mintAmount)
                    : mintAmount;
    totalUnderlying += actualMintAmount;

    // note
    // calculate amount of tokens to get in return
    uint256 mintTokens = div_(actualMintAmount, exchangeRate);
    require(mintTokens >= minAmountOut, mt_MinAmountNotValid());

    // note
    // special case for first depositor
    // - mint 1000 mTokens to address(0)
    // - subtract 1000 from the user's calculated minting amount
@>  if (totalSupply == 0) {
        totalSupply = 1000;
        accountTokens[address(0)] = 1000;
        mintTokens -= 1000;
    }
```

with an initial interest rate of 0.02, at least 20 wei is required because its equivalent the amount of 1000 tokens, enough to be subtracted in the bootstrap, otherwise underflow happens. the interest rate calculation can be seen as follow `mTokenStorage::_exchangeRateStored`:

```
function _exchangeRateStored() internal view virtual returns (uint256) {
    uint256 _totalSupply = totalSupply;
    if (_totalSupply == 0) {
        return initialExchangeRateMantissa;
    } else {
        uint256 totalCash = _getCashPrior();
       uint256 cashPlusBorrowsMinusReserves = totalCash + totalBorrows - totalReserves;
       uint256 exchangeRate = (cashPlusBorrowsMinusReserves * expScale) / _totalSupply;
        return exchangeRate;
    }
}
```

Step #2:

Using loop for about ~1000 times, for each loop, deposit as little as `1 wei` and a redemption of half of the deposited wei in token (~25 tokens), to slowly and gradually inflates the interest rate.

> Note that the host chain is **Linea**, despite having thousands of loop that will cause tons of gas execution (propably more than 10,000,000), the gas would still be very cheap. At the time of writing, per 1000 iterations will only costs about **$0.50** worth of ETH. Addition to that, linea gas limit per block is 2,000,000,000, which is not impossible to perform these large amount of loops

Step #3:

Hacker can wait patiently for more depositor to get involved into the deposits to increase the interest rate again. The inflated interest rate is enough to lure depositors.

Then, hacker can keep calling `mToken::__redeem()` to perform the redemption, by specifying the underlying amount. Now that the exchange rate is extremely high, trying to redeem small amount of underlying, using such operations `redeemTokens = div_(redeemAmountIn, exchangeRate)`, will return 0.

Meaning hacker is able to withdraw the underlying, without burning any tokens.

Recommended Mitigation:

Use round up to perform pulling token calculation

```
    } else {
        // note
        // user specify amount of underlying to get back
        // calculate mTokens to burn
-       redeemTokens = div_(redeemAmountIn, exchangeRate);
+       redeemTokens = divUp_(redeemAmountIn, exchangeRate);
        redeemAmount = redeemAmountIn;
    }
```

[M-2] blacklister can be bypassed in `outHere`

Description:

`mTokenGateway::outHere()` function is as below:

```
function outHere(
    bytes calldata journalData,
    bytes calldata seal,
    uint256[] calldata amounts,
    address receiver
) external
  notPaused(OperationType.AmountOutHere)
  ifNotBlacklisted(msg.sender)
  ifNotBlacklisted(receiver)
{
    // ...
    for (uint256 i; i < journals.length;) {
@>      _outHere(journals[i], amounts[i], receiver);

        unchecked {
            ++i;
        }
    }
}

function _outHere(bytes memory journalData, uint256 amount, address receiver)
    internal {
    (
        address _sender,
        address _market,,
        uint256 _accAmountOut,
        uint32 _chainId,
        uint32 _dstChainId,
    ) = mTokenProofDecoderLib.decodeJournal(journalData);

@>  receiver = _sender;

    // checks
@>  _checkSender(msg.sender, _sender);
    require(_market == address(this), ...());
    require(_chainId == LINEA_CHAIN_ID, ...()); // allow only Host
    require(_dstChainId == uint32(block.chainid), ...());
    require(amount > 0, ...());
    require(_accAmountOut - accAmountOut[_sender] >= amount, ...());
    require(IERC20(underlying).balanceOf(address(this)) >= amount, ...());

    // effects
@>  accAmountOut[_sender] += amount;

    // interactions
@>  IERC20(underlying).safeTransfer(_sender, amount);

    emit mTokenGateway_Extracted(
        msg.sender,
@>      _sender,
@>      receiver,
        accAmountIn[_sender],
        accAmountOut[_sender],
        amount,
        uint32(_chainId),
        uint32(block.chainid)
    );
```

```
}

function _checkSender(address msgSender, address srcSender) private view {
    if (msgSender != srcSender) {
        require(
            allowedCallers[srcSender][msgSender] ||
            msgSender == owner() ||
            _isAllowedFor(msgSender, _getProofForwarderRole()) ||
            _isAllowedFor(msgSender, _getBatchProofForwarderRole()),
            mTokenGateway_CallerNotAllowed()
        );
    }
}
```

This function is likely meant to finalize the cross-chain transaction in the destination chain, that was initialized from the source chain

However, despite having lots of `ifNotBlacklisted` checks, `_sender` from journal is never checked, making a blacklisted user being able to use this function `outHere`

Impact:

A blacklisted user, that's meant to be refrain from being involved in any of the DeFI activity, including from using this `outHere` function, could completely bypass without a revert

Exploitation:

- Bob is blacklisted
- Alice is not blacklisted

Step #1:

- Bob creates valid journalData where `_sender = Bob`.
- This is the value that will be extracted inside `_outHere(...)`.

```
(address Bob, ...) = decodeJournal(journalData);
```

Step #2:

- Bob asks Alice to call `outHere(...)` on his behalf
- Alice is not blacklisted, so she passes this check:

```
ifNotBlacklisted(msg.sender) // msg.sender == Alice
ifNotBlacklisted(receiver) // receiver = any whitelisted address, or Alice herself
```

- inside `outHere()`, it looks over each journal, and for each journal is passed to `_outHere()`

- inside `_outHere()`, the line of code `receiver = _sender;` exist, overriding the Bob with the actual `receiver`

Step #3:

`_sender` is pulled into `_checkSender()` and verify some stuff:

```
// note that msg.sender == Alice
if (Alice != Bob) {
    require(
        allowedCallers[Bob][Alice] ||
        Alice == owner ||
        Alice has forwarder roles ||
        Alice has batch forwarder roles,
        "CallerNotAllowed"
    );
}
```

as long as `allowedCallers[Bob][Alice] == true`, doesn't matter the rest of the condition, the blacklisted Bob can be completely bypass, since there's no furthur checks about blacklisted user after this function execution.

Addition to that, `allowedCallers[Bob][Alice] == true` can be set freely by whoever and whenever they want, from the function `mTokenGateway::updateAllowedCallerStatus()`:

```
function updateAllowedCallerStatus(address caller, bool status) external override {
    allowedCallers[msg.sender][caller] = status;
    emit AllowedCallerUpdated(msg.sender, caller, status);
}
```

Recommended Mitigation:

```
    // checks
    _checkSender(msg.sender, _sender);
+   require (!blacklistOperator.isBlacklisted(_sender), ....);
    require(_market == address(this), ....);
    require(_chainId == LINEA_CHAIN_ID, ....); // allow only Host
    require(_dstChainId == uint32(block.chainid), ....);
```

[M-3] `EverclearBridge` does not pull tokens

Description:

`EverclearBridge::sendMsg` is a cross-chain rebalancing function that's done in the source chain

> Side Note:
>
> vault (chain a) –> rebalancer –> vault (chain b).
>
> The Rebalancer main job is to manage and balance the protocol's liquidity across different chains.

```
    bool found;
    for (uint256 i; i < destinationsLength; ++i) {
        if (params.destinations[i] == _dstChainId) {
            found = true;
            break;
        }
    }
    require(found, Everclear_DestinationNotValid());

    // note
    // if the amount transfered to rebalancer is more than whats needed that
    // defined in `_message`(params), transfer back to the vault
    if (_extractedAmount > params.amount) {
        uint256 toReturn = _extractedAmount - params.amount;
@>      IERC20(_token).safeTransfer(_market, toReturn);
        emit RebalancingReturnedToMarket(_market, toReturn, _extractedAmount);
    }

@>  SafeApprove.safeApprove(....);
@>  (bytes32 id,) = everclearFeeAdapter.newIntent(
        params.destinations,
        params.receiver,
        params.inputAsset,
        params.outputAsset,
        params.amount,
        params.maxFee,
        params.ttl,
        params.data,
        params.feeParams
    );
```

1. `IERC20(_token).safeTransfer(_market, toReturn);` transfers back the excessive tokens to market
2. `SafeApprove.safeApprove(...)` approve the fee adapter to collect fees from this contract `EverclearBridge`
3. `everclearFeeAdapter.newIntent(...)` likely to initiate cross-chain intent

However, no `safeTransferFrom` (or any other similar) function is called to pull tokens from rebalancer, into this contract `EverclearBridge`, before proceeding these 3 operation stated above, which in turn making this function trying to transfer tokens from its contract `EverclearBridge` that simply doesnt hold any tokens.

Impact:

Due to the missing `safeTransferFrom` function that pull tokens from rebalancer, all rebalancing operation that uses this bridge will fails and result in Denial of Service (DOS) for the cross-chain liquidity movement.

Recommended Mitigtion:

Add `safeTransferFrom` to pull tokens from rebalancer, ensuring that the tokens actually flowing into the right path.

```
+    IERC20(_token).safeTransferFrom(
+        msg.sender,
+        address(this),
+        _extractedAmount
+    );

     // note
     // if the amount transfered to rebalancer is more than whats needed that
     // defined in `_message`(params), transfer back to the vault
     if (_extractedAmount > params.amount) {
         uint256 toReturn = _extractedAmount - params.amount;
         IERC20(_token).safeTransfer(_market, toReturn);
         emit RebalancingReturnedToMarket(_market, toReturn, _extractedAmount);
     }
```

[M-4] `EverclearBridge` is completely broken with incorrect approvals

Description:

again in `EverclearBridge::sendMsg` function, it approves the `everclearFeeAdapter` to pull the tokens using the function `SafeApprove.safeApprove()`, follow by calling `newIntent` function to actually pull the tokens.

```
@>   SafeApprove.safeApprove(params.inputAsset, address(everclearFeeAdapter), params.amount);
@>   (bytes32 id,) = everclearFeeAdapter.newIntent(
         params.destinations,
         params.receiver,
         params.inputAsset,
         params.outputAsset,
         params.amount,
         params.maxFee,
         params.ttl,
         params.data,
         params.feeParams
     );
```

below is the `newIntent` function:

```
    function newIntent(
        uint32[] memory _destinations,
        address _receiver,
        address _inputAsset,
        address _outputAsset,
        uint256 _amount,
        uint24 _maxFee,
        uint48 _ttl,
        bytes calldata _data,
        IFeeAdapter.FeeParams calldata _feeParams
    ) external payable returns ( ... ) {
@>      _pullTokens(msg.sender, _inputAsset, _amount + _feeParams.fee);

        // ...
    }

    function _pullTokens(address _sender, address _asset, uint256 _amount) internal {
        IERC20(_asset).safeTransferFrom(_sender, address(this), _amount);
    }
```

`EverclearBridge::sendMsg` approving `everclearFeeAdapter` to spend `params.amount` amount of token as fees. However, `newIntent` is pulling a total of `params.amount` + `params.feeParams.fee` from the `EverclearBridge` contract

Impact:

`EverclearBridge::sendMsg` only approves `params.amount` amount, but a total of `params.amount` + `params.feeParams.fee` is being pulled

As a result, this will always get reverted due to inefficient allowance.

Recommended Mitigation:

update the approval amount

```
    // ...
    require(params.inputAsset == _token, ...);
-   require(_extractedAmount >= params.amount, ...);
+   require(_extractedAmount >= params.amount + params.feeParams.fee, ...);
    // ...

    // note
    // if the amount transfered to rebalancer is more than whats needed that
    // defined in `_message`(params), transfer back to the vault
```

```diff
-   if (_extractedAmount > params.amount) {
+   if (_extractedAmount > params.amount + params.feeParams.fee) {
-       uint256 toReturn = _extractedAmount - params.amount;
+       uint256 toReturn = _extractedAmount - (params.amount + params.feeParams.fee);
        IERC20(_token).safeTransfer(_market, toReturn);
        emit RebalancingReturnedToMarket(_market, toReturn, _extractedAmount);
    }

-   SafeApprove.safeApprove(params.inputAsset, address(everclearFeeAdapter),
 ↪  params.amount);
+   SafeApprove.safeApprove(params.inputAsset, address(everclearFeeAdapter),
 ↪  params.amount + params.amount + params.feeParams.fee);
```

[M-5] incorrect transfer size validation

Description:

`Rebalancer::sendMsg` is likely to call other `sendMsg` function from other contract to initiate the rebalancing operation in the source chain to rebalance the market liquidity

```
    // min transfer size check
    require(_amount > minTransferSizes[_msg.dstChainId][_msg.token], ... );

    // max transfer size checks
    // struct TransferInfo { uint256 size; uint256 timestamp; }
@>  TransferInfo memory transferInfo =
 ↪  currentTransferSize[_msg.dstChainId][_msg.token];
    uint256 transferSizeDeadline = transferInfo.timestamp + transferTimeWindow;
    if (transferSizeDeadline < block.timestamp) {
        // Time window expired -> reset size and timestamp
        currentTransferSize[_msg.dstChainId][_msg.token] = TransferInfo(_amount,
         ↪  block.timestamp);
    } else {
        // Still within same window -> reset size only
        currentTransferSize[_msg.dstChainId][_msg.token].size += _amount;
    }

    uint256 _maxTransferSize = maxTransferSizes[_msg.dstChainId][_msg.token];
    if (_maxTransferSize > 0) {
@>      require(transferInfo.size + _amount < _maxTransferSize, ....());
    }
```

the code showing above is part of `Rebalancer::sendMsg`, validating transfer amount to ensures that a total ideal amount is transfered within a time window

1. validate the transfer amount to be above minimum

2. update the transfer size (increment if not expired, otherwise renew it)
3. validate the size to be below maximum

while on maximum amount checks, `transferInfo`, that was declared as local memory variable, is used for the checks. However, since this was declared as memory, it was basically a snapshot of the data at the time of declaration. The changes/updates is done directly to the global storage variable thereafter the minimum validation, which has no effects to the `transferInfo` memory variable itself, therefore an outdated data is used for the maximum checks.

Impact:

When the `transferTimeWindow` has expired and `currentTransferSize` is reset directly in the global storage, the furthur execution of the maximum checks is using an outdated memory copy `transferInfo`. As such, the `transferInfo` may approaching the max limit amount, follow by adding a transfer amount, will ended up exceeding the maximum limit, and therefore always gets reverted here.

Exploitation:

- Expired on: 6pm
- Time now: 4pm
- Current existing transfered size: 100 wETH
- Max transfer size: 800 wETH

Step #1:

rebalancer EOA calls `sendMsg`, with a transfer size of 600 wETH to rebalance the liquidity across different chain.

Step #2:

since current time is not expired yet, the 600 wETH will be incremented into the current existing transfered size (100 wETH)

the updated/incremented amount will gets validated with the maximum transfer size, since the total transfer size within this time window (100 + 600 wETH) is less than 800 wETH, this checks will pass

then continue with furthur execution

Step #3:

another rebalance operation comes after 6pm, trying to transfers 300 wETH.

since it has expired, the transfer size gets renewed to 300 wETH, instead of 700 + 300 wETH

however, the maximum size checks took `transferInfo`, which was a snapshot of the transfer size (700 wETH) before it is renewed as a result, the maximum size uses 700 + 300 wETH to validate the maximum amount, which ended up exceeding the max amount and reverting a legitimate transaction.

Recommended Mitigation:

Compare the amount directly from the global variable

```diff
    uint256 _maxTransferSize = maxTransferSizes[_msg.dstChainId][_msg.token];
    if (_maxTransferSize > 0) {
        require(
-           transferInfo.size + _amount < _maxTransferSize,
+           currentTransferSize[_msg.dstChainId][_msg.token].size < _maxTransferSize,
            Rebalancer_TransferSizeExcedeed()
        );
    }
```

Or, make the `transferInfo` as storage instead of memory, then have it directly compared to the maximum amount

```diff
    // max transfer size checks
    // struct TransferInfo { uint256 size; uint256 timestamp; }
-   TransferInfo memory transferInfo =
↪   currentTransferSize[_msg.dstChainId][_msg.token];
+   TransferInfo storage transferInfo =
↪   currentTransferSize[_msg.dstChainId][_msg.token];
    uint256 transferSizeDeadline = transferInfo.timestamp + transferTimeWindow;

    .
    .
    .

    uint256 _maxTransferSize = maxTransferSizes[_msg.dstChainId][_msg.token];
    if (_maxTransferSize > 0) {
        require(
-           transferInfo.size + _amount < _maxTransferSize,
+           transferInfo.size < _maxTransferSize,
            Rebalancer_TransferSizeExcedeed()
        );
    }
```

[M-6] `wrapAndSupplyOnExtensionMarket` fails to send `msg.value`

Description:

`WrapAndSupply::wrapAndSupplyOnExtensionMarket` wraps `msg.value` into an underlying, and calls `mTokenGateway::supplyOnHost` function to pull the underlying.

```
    function wrapAndSupplyOnExtensionMarket(...)
        external
        payable
    {
        address underlying = ImTokenGateway(mTokenGateway).underlying();
        require(underlying == address(wrappedNative), ... );

@>      uint256 amount = _wrap();

        IERC20(underlying).approve(mTokenGateway, 0);
        IERC20(underlying).approve(mTokenGateway, amount);
@>      ImTokenGateway(mTokenGateway).supplyOnHost(amount, receiver, selector);
    }

    function _wrap() private returns (uint256) {
        uint256 amount = msg.value;
        require(amount > 0, WrapAndSupply_AmountNotValid());

@>      wrappedNative.deposit{value: amount}();
        return amount;
    }
```

code below is a part from `mTokenGateway::supplyOnHost`:

```
    require(amount > 0, mTokenGateway_AmountNotValid());
@>  require(msg.value >= gasFee, mTokenGateway_NotEnoughGasFee());
```

`WrapAndSupply::wrapAndSupplyOnExtensionMarket` wraps 100% of the `msg.value` sent into an underlying, and have `mTokenGateway::supplyOnHost` pulls the underlying. However, `mTokenGateway::supplyOnHost` restrict the amount of `msg.value` to be at least `gasFee`.

Impact:

`WrapAndSupply::wrapAndSupplyOnExtensionMarket` wraps all of the `msg.value` into underlying, then calls `mTokenGateway::supplyOnHost` without sending any `msg.value`.

```
    require(msg.value >= gasFee, mTokenGateway_NotEnoughGasFee());
```

As long as `gasFee` is set, `WrapAndSupply::wrapAndSupplyOnExtensionMarket` will always get reverted due to `mTokenGateway_NotEnoughGasFee`

Recommended Mitigation:

1. create a getter function in `mTokenGateway` contract that retrieve `gasFee` values.

```
    function getGasFee() external view returns(uint256) {
        return gasFee;
    }
```

2. only wraps a total of `msg.value` that has the `gasFee` amount excluded, thereby being able to transfer the rest of the `msg.value` (`gasFee`) while calling `supplyOnHost`

```
    function wrapAndSupplyOnHostMarket(...) external payable {
        // ...
-       uint256 amount = _wrap();
+       uint256 amount = _wrap(msg.value);
        // ...
    }

    function wrapAndSupplyOnExtensionMarket(...)
        external
        payable
    {
        // ...
+       uint256 fee = ImTokenGateway(mTokenGateway).getGasFee();
+       require(msg.value > fee, ...);  // optional: ensures that value > fee
-       uint256 amount = _wrap();
+       uint256 amount = _wrap(msg.value - fee);

        IERC20(underlying).approve(mTokenGateway, 0);
        IERC20(underlying).approve(mTokenGateway, amount);
-       ImTokenGateway(mTokenGateway).supplyOnHost(amount, receiver, selector);
+       ImTokenGateway(mTokenGateway).supplyOnHost{value: fee}(amount, receiver,
↪   selector);
    }

-   function _wrap() private returns (uint256) {
+   function _wrap(uint256 amount) private returns (uint256) {
-       uint256 amount = msg.value;
        require(amount > 0, WrapAndSupply_AmountNotValid());

        wrappedNative.deposit{value: amount}();
        return amount;
    }
```

[M-7] lack of batch process features

Description:

`BatchSubmitter.batchProcess` function allows user to get involved into multiple DeFi/lending activity in one single function. Below is part of this function:

```
       // ...
       _verifyProof(data.journalData, data.seal);
       // ...

       for (uint256 i = 0; i < length;) {
           // ...

@>         if (selector == MINT_SELECTOR) {
               try ImErc20Host(data.mTokens[i]).mintExternal( ... )

@>         } else if (selector == REPAY_SELECTOR) {
               try ImErc20Host(data.mTokens[i]).repayExternal( ... )

@>         } else if (selector == OUT_HERE_SELECTOR) {
               try ImTokenGateway(data.mTokens[i]).outHere( ... )

@>         } else {
               revert BatchSubmitter_InvalidSelector();
           }

       }
```

as of current implementation, this function support the following processes:

- `mErc20Host.mintExternal()`: mint mToken that subject to underlying deposited from other chain
- `mErc20Host.repayExternal()`: repay loans that was from other chain
- `mTokenGateway.outHere()`: burn mToken that subject to underlying withwithdrew from other chain

however, `mErc20Host.liquidateExternal()`, that's meant for liquidator to liquidate liquidated user from other chain, is not implemented in this batch process.

the code below shows that 4 of these functions are supported to be implemented in batch process:

```
       // mErc20Host.mintExternal()
       function mintExternal(...) external override {
@>         if (!_isAllowedFor(msg.sender, _getBatchProofForwarderRole())) {
               _verifyProof(journalData, seal);
           }
           // ...
       }

       // mErc20Host.repayExternal()
       function repayExternal(...) external override {
@>         if (!_isAllowedFor(msg.sender, _getBatchProofForwarderRole())) {
               _verifyProof(journalData, seal);
           }
           // ...
```

```
    }

    // mErc20Host.liquidateExternal()
    function liquidateExternal( ... ) external override {
@>      if (!_isAllowedFor(msg.sender, _getBatchProofForwarderRole())) {
            _verifyProof(journalData, seal);
        }
        // ...
    }

    // mTokenGateway.outHere()
    function outHere(...) external ... {
@>      if (!rolesOperator.isAllowedFor(msg.sender,
↪   rolesOperator.PROOF_BATCH_FORWARDER())) {
            _verifyProof(journalData, seal);
        }
        // ...
    }
```

4 of these functions skip `_verifyProof` if the caller is not from `BatchSubmitter` contract (`PROOF_BATCH_FORWARDER` role) because the `BatchSubmitter.batchProcess` function itself already have `_verifyProof` being executed right before calling any of these 4 functions.

Impact:

since the current implementation did not includes `mErc20Host.liquidateExternal()` function, an attempt to execute this function will always fails and gets reverted with error message `BatchSubmitter_InvalidSelector()`

Recommended Mitigation:

at `mErc20Host.liquidateExternal()` function call features into `BatchSubmitter.batchProcess` function.

```
    bytes4 internal constant MINT_SELECTOR = ImErc20Host.mintExternal.selector;
    bytes4 internal constant REPAY_SELECTOR = ImErc20Host.repayExternal.selector;
+   bytes4 internal constant LIQUIDATE_SELECTOR =
↪   ImErc20Host.liquidateExternal.selector;
    bytes4 internal constant OUT_HERE_SELECTOR = ImTokenGateway.outHere.selector;

    .
    .
    .

    if (selector == MINT_SELECTOR) {
        // ...
    } else if (selector == REPAY_SELECTOR) {
        // ...
    } else if (selector == OUT_HERE_SELECTOR) {
        // ...
+    } else if (selector == LIQUIDATE_SELECTOR) {
+        // call mErc20Host.liquidateExternal() here
    } else {
        revert BatchSubmitter_InvalidSelector();
    }
```

[M-8] possible market funds drainage due to excessive bridge fee set by rebalancer

Description:

below is part of the code from `EverclearBridge::sendMsg`

```
     SafeApprove.safeApprove(...);
     (bytes32 id,) = everclearFeeAdapter.newIntent(
         params.destinations,
         params.receiver,
         params.inputAsset,
         params.outputAsset,
         params.amount,
@>       params.maxFee,
         params.ttl,
         params.data,
         params.feeParams
     );
```

A rebalancer role calls `Rebalancer::sendMsg`, follow by calling `EverclearBridge::sendMsg` in within the same function. During these calls, there's no single line of code that validate the amount of `maxFee`, potentially affecting the bridge amount and liquidity with an excessive `maxFee`.

> Rebalancer is semi-trusted - Rebalancer only has DDOS abilities for the protocol via constant rebalancing. It cannot transfer user funds

Impact:

A semi-trusted rebalancer can set the `maxFee` as high as approaching the `params.amount`, causing very little amount to be successfully transfered to destination chain.

Exploitation:

Step 1:

Malicious rebalancer provides a valid `_amount` to bridge, for example 10 ETH. At the same time, malicious rebalancer sets `maxFee` as high as 9.9 ETH

Step 2:

Due to this, 9.9 ETH is used for fees, 0.1 ETH (or less) is successfully transfered to the destination chain. Despite malicious rebalancer won't get anything in return, they could repeats these steps until the funds in source chain is drained due to the excessive `maxFee`

Recommended Mitigation:

two recommended mitigation: 1. have a trusted role, e.g. `GUARDIAN_BRIDGE` to manage the rebalancing operation, particularly in creating the amount of `maxFee` 2. have a percentage being `maxFee` of the `_amount` so that the fee is not set solely by a semi-trusted `Rebalancer`

[M-9] first depositor can break the market by manipulating the borrow rate

Description:

both utility rate and borrow rate calculation is located inside the `JumpRateModelV4` contract

> Utility Rate - % has been borrowed out already
>
> Borrow Rate - % interest rate per block ("cost to borrow")

```
        function utilizationRate(...) public pure override returns (uint256) {
            if (borrows == 0) {
                return 0;
            }
@>          return borrows * 1e18 / (cash + borrows - reserves);
        }

        function getBorrowRate(...) public view override returns (uint256) {
            uint256 util = utilizationRate(cash, borrows, reserves);

            // multiplierPerBlock - slope curve when util below kink
            // jumpMultiplierPerBlock - slop curve when util above kink

            if (util <= kink) {
                return util * multiplierPerBlock / 1e18 + baseRatePerBlock;
@>          } else {
                uint256 normalRate = kink * multiplierPerBlock / 1e18 + baseRatePerBlock;
                uint256 excessUtil = util - kink;
                return excessUtil * jumpMultiplierPerBlock / 1e18 + normalRate;
            }
        }
```

below is a part of `mTokenStorage::_accrueInterest` function

```
    function _accrueInterest() internal {
        // ...
        uint256 borrowRateMantissa =
            IInterestRateModel(interestRateModel).getBorrowRate(...);
        if (borrowRateMaxMantissa > 0) {
@>          require(borrowRateMantissa <= borrowRateMaxMantissa, ...());
        }
        // ...
```

as a side note:

below is how utilization rate is calculated:

$$\text{UtilizationRate} = \frac{\text{TotalBorrows} \times 10^{18}}{\text{TotalCash} + \text{TotalBorrows} - \text{TotalReserves}}$$

to calculate borrow rate, we first check if the utilization rate (UR) is above kink level. the calculation below represent a borrow rate when utlization rate is below kink level.

$$\text{BorrowRate} = \frac{\text{UR} \times \text{MultiplierPerBlock}}{10^{18}} + \text{BaseRatePerBlock}$$

calculation below represent a borrow rate when utilization rate is above kink level

$$\text{BorrowRate} = \left(\frac{\text{Kink} \times \text{MultiplierPerBlock}}{10^{18}} + \text{BaseRatePerBlock}\right) + \frac{(\text{UR} - \text{Kink}) \times \text{JumpMultiplierPerBlock}}{10^{18}}$$

we notice that the denominator in the utilization rate formula `(cash + borrows - reserves)` is possible to be adjusted and resulting in a very small amount during the first deposit

when the denominator resulting in small amount, the utilization rate is going to be extremely high, follow by the borrow rate calculation, which will also makes the borrow rate very high as well. when the large borrow rate returned in the `_accrueInterest` function, it is very likely that the large borrow rate is greater than `borrowRateMaxMantissa` and therefore break the protocol functionality

Impact:

the first depositor can manipulate the utilization rate and causes the borrow rate to be extremely high, until it has surpass the max borrow rate, and therefore break the protocol functionality.

Exploitation:

- reserveFactor = 50%

- interest accrual = 1e12 (per block)

Step #1:

initial state of the protocol before first depositor hop in:

```
cash = 0
totalBorrows = 0
totalReserves = 0
```

Step #2:

attacker being first borrower supply some underlying to enable borrow power, and also supply some underlying as `cash` (e.g. 10 ETH), then borrow all of the supplied `cash` (10 ETH)

```
cash = 0
totalBorrows = 10 ETH
totalReserves = 0
```

since the attacker has borrowed all of the available `cash`, utility rate is going to be 100%. after that, wait for some time to pass, to generate interest rate.

```
utillity rate = totalBorrowerd * 1e18 / (cash + totalBorrowerd - reserves)
              = 10 ETH * 1e18 / (0 + 10 ETH - 0)
              = 1e18
```

Step #3:

now `_accrueInterest` function runs:

```
cash = 0
totalBorrows = 10 ETH + 1e12
totalReserves = 5e11
```

then attacker repays almost everything, leaving tiny amount of debt.

```
repay_amount = totalBorrows - totalReserves - 1
```

```
cash = repay_amount
totalBorrows = (10 ETH + 1e12) - repay_amount = 5e11 + 1
totalReserves = 5e11
```

Step #3:

attacker withdraw/redeem all of the `cash`

```
cash = 0
totalBorrows = 5e11 + 1
totalReserves = 5e11
```

now this has causes the utilizity rate to be extremely high due to small denominator

```
utillity rate = borrows * 1e18 / (cash + borrows - reserves)
              = (5e11 + 1) * 1e18 / (0 + (5e11 + 1) - 5e11)
              = (5e11 + 1) * 1e18 / 1
              = (5e11 + 1) * 1e18
```

as compared to the max borrow rate, `uint256 public borrowRateMaxMantissa = 0.0005e16;`, all of the future interaction is broken and cant be performed, due to the error message `mt_BorrowRateTooHigh()`

Recommended Mitigation:

consider adding a safety check on utilization rate, ensuring it is a valid percentage, by checking the denominator.

[M-10] miscalculation of minimum shares return with minimum underlying deposit

Description:

`Migrator::migrateAllPositions` migrates user's full position, such as collaterals and shares, from old protocol (mendi) to new protocol (malda)

```
    function migrateAllPositions() external {
        // ...
        for (uint256 i; i < posLength; ++i) {
            Position memory position = positions[i];
            if (position.collateralUnderlyingAmount > 0) {
@>              uint256 minCollateral = position.collateralUnderlyingAmount -
                            (position.collateralUnderlyingAmount * 1e4 / 1e5);
@>              ImErc20Host(position.maldaMarket).mintOrBorrowMigration(
                    true, position.collateralUnderlyingAmount, msg.sender, address(0),
                    ↪ minCollateral
                );
            }
        }
        // ...
    }
```

below is the `mErc20Host::mintOrBorrowMigration` that mint/borrow shares

```
    function mintOrBorrowMigration(
        bool mint,
        uint256 amount,
        address receiver,
        address borrower,
@>      uint256 minAmount
    ) external onlyMigrator {
        require(amount > 0, mErc20Host_AmountNotValid());

        if (mint) {
            _mint(receiver, receiver, amount, minAmount, false);
            emit mErc20Host_MintMigration(receiver, amount);
        } else {
            _borrowWithReceiver(borrower, receiver, amount);
            emit mErc20Host_BorrowMigration(borrower, amount);
        }
    }
```

the `minAmount` param in the `mintOrBorrowMigration` function refers to the minimum shares to be transfered to the `receiver`, that act as a slippage protection.

however, we notice that the argument for `minAmount` parameters calculated in `migrateAllPositions` function is referring to the minimum underlying, rather than minimum shares, which resulting in way less shares to be set as minimum for slippage protection.

Impact:

since the minimum shares for slippage protection is set by the minimum underlying, the minimum shares amount is going to be way more lesser than it should, making it overestimate the slippage protection.

Exploitation (Math Scenario):

- 100 underlying
- 0.02 exchange rate

as a side note, below is how shares is calculated based on the amount of underlying deposited

```
shares = underlying / exchange rate
       = 100 / 0.02
       = 5,000 shares
```

calculation for slippage protection with the current buggy code (mistook minimum shares as minimum underlying):

```
minAmount = underlying - (underlying * 1e4 / 1e5)
          = 100 - (100 * 1e4 / 1e5)
          = 90
```

correct calculation for slippage protection

```
minAmountInUnderlying = underlying - (underlying * 1e4 / 1e5)
                      = 100 - (100 * 1e4 / 1e5)
                      = 90
minAmountInShares = underlying / exchange rate
                  = 90 / 0.02
                  = 4500
```

the comparison of the slippage protection amount between the current implementation (buggy) and the correct behavior is huge. the buggy calculation allows a smaller unit than it should to be as minimum shares to be transfered to user

Recommended Mitigtion:

continue with furthur division to turn the minimum underlying into minimum shares

```
    if (position.collateralUnderlyingAmount > 0) {
-       uint256 minCollateral =
-           position.collateralUnderlyingAmount - (position.collateralUnderlyingAmount
  ↪ * 1e4 / 1e5);
+       uint256 minCollateral = div_(
+           position.collateralUnderlyingAmount - (position.collateralUnderlyingAmount
  ↪ * 1e4 / 1e5),
+           _exchangeRate
+       );
        ImErc20Host(position.maldaMarket).mintOrBorrowMigration(
            true, position.collateralUnderlyingAmount, msg.sender, address(0),
             ↪  minCollateral
        );
    }
```

[M-11] incorrect assumption of same decimals returned by different oracle

Description:

`MixedPriceOracleV4::_getLatestPrice` function returns the latest price and decimal from an oracle

```
    function _getLatestPrice(...) internal view returns (uint256, uint256) {
        // ...
        // get price and last update price timestamp
        // API3 Oracle and eOracle
        (, int256 apiV3Price,, uint256 apiV3UpdatedAt,) = IDefaultAdapter(config.api3Feed).latestRoundData();
        (, int256 eOraclePrice,, uint256 eOracleUpdatedAt,) = IDefaultAdapter(config.eOracleFeed).latestRoundData();

        // check price staleness
        uint256 _staleness = _getStaleness(symbol);
        bool apiV3Fresh = block.timestamp - apiV3UpdatedAt <= _staleness;

        // get differences/delta of the 2 prices
@>      uint256 delta = _absDiff(apiV3Price, eOraclePrice);
@>      uint256 deltaBps = (delta * PRICE_DELTA_EXP) / uint256(eOraclePrice < 0 ? -eOraclePrice : eOraclePrice);

        uint256 deltaSymbol = deltaPerSymbol[symbol];
        if (deltaSymbol == 0) {
            deltaSymbol = maxPriceDelta;
        }

        uint256 decimals;
        uint256 uPrice;
        // decide which price to use
        if (!apiV3Fresh || deltaBps > deltaSymbol) {
            require(block.timestamp - eOracleUpdatedAt < _staleness, ...());
            decimals = IDefaultAdapter(config.eOracleFeed).decimals();
            uPrice = uint256(eOraclePrice);
        } else {
            require(block.timestamp - apiV3UpdatedAt < _staleness, ...());
            decimals = IDefaultAdapter(config.api3Feed).decimals();
            uPrice = uint256(apiV3Price);
        }

        return (uPrice, decimals);
    }

    function _absDiff(int256 a, int256 b) internal pure returns (uint256) {
@>      return uint256(a >= b ? a - b : b - a);
    }
```

`_getLatestPrice` function fetches price and decimal from both API3 oracle and eOracle, follow by determine the best price to return, by checking the staleness and price dfference.

when calculating the price different of an asset from both oracle, the calculation is done by having an absolute subtraction. however this is not a safe option to calculate the price different because the price fetched from both oracle might have different decimals, which may always result in large differences.

Impact:

based on the documentation, API3 Oracle data feeds usually returns 18 decimals, while eOracle data feeds usually returns 8 decimals.

```
    if (!apiV3Fresh || deltaBps > deltaSymbol) {
        // eOracle price
    } else {
        // API3 oracle price
    }
```

as a result, eOracle will always get chosen to fetch the price

Recommended Mitigation:

normalize both price decimals (e.g. make both of them to be in 18 decimals) before calculating the delta

---

[M-12] rebalancer can send unsupported chain via `EverclearBridge`

Description:

`EverclearBridge::sendMsg` involve a sanity checks on the encoded `_message` whether or not the desired destination chain is included by having it to be compared with `_dstChainId`.

```
    function sendMsg(
        uint256 _extractedAmount,
        address _market,
        uint32 _dstChainId,
        address _token,
        bytes memory _message,
        bytes memory // unused
    ) external payable onlyRebalancer {
        IntentParams memory params = _decodeIntent(_message);
        // ...

        uint256 destinationsLength = params.destinations.length;
        require(destinationsLength > 0, Everclear_DestinationsLengthMismatch());

        bool found;
        for (uint256 i; i < destinationsLength; ++i) {
@>          if (params.destinations[i] == _dstChainId) {
@>              found = true;
                break;
            }
        }
        require(found, Everclear_DestinationNotValid());
        // ...
```

```
            (bytes32 id,) = everclearFeeAdapter.newIntent(
@>              params.destinations,
                params.receiver,
                params.inputAsset,
                params.outputAsset,
                params.amount,
                params.maxFee,
                params.ttl,
                params.data,
                params.feeParams
            );
            emit MsgSent(_dstChainId, _market, params.amount, id);
        }
```

however, we notice that the current implementation involve only checking whether at least one desired destination chain exist in the `params.destinations` loop (from `_message`).

Impact:

since it only checks that if there's one chain consist inside the `_message` is aligned to the desired chain `_dstChainId`, the rest of the (potentially) blacklisted or unsupported chain will be successfully pass to `newIntent` to process multiple cross-chain operation, making the funds loss in nowhere in the chain given in the `_message` that are not the same as `_dstChainId`.

Recommended Mitigation:

only allow one elements in the `params.destinations`, and that will be checked against the `_desChainId` to ensures only one desired desination chain is allowed for the cross-chain operation, avoiding unsupported chain.

```
            uint256 destinationsLength = params.destinations.length;
-           require(destinationsLength > 0, ...());
+           require(destinationsLength == 1, ...());

-           bool found;
-           for (uint256 i; i < destinationsLength; ++i) {
-               if (params.destinations[i] == _dstChainId) {
-                   found = true;
-                   break;
-               }
-           }
-           require(found, ...());
+           require(params.destinations[0] == _dstChainId, ...());
```

[M-13] unhandling refunds token

Description:

`AcrossBridge::_depositV3Now` uses Across Protocol to perform rebalancing cross-chain operation.

```
    function sendMsg(...) external payable onlyRebalancer {
        // ...
        // retrieve tokens from `Rebalancer`
@>      IERC20(_token).safeTransferFrom(msg.sender, address(this), msgData.inputAmount);

        // ...
        // approve and send with Across
@>      _depositV3Now(_message, _token, _dstChainId, _market);
    }

    function _depositV3Now(...) private {
        DecodedMessage memory msgData = _decodeMessage(_message);
        // approve and send with Across
       SafeApprove.safeApprove(_token, address(acrossSpokePool), msgData.inputAmount);
@>      IAcrossSpokePoolV3(acrossSpokePool).depositV3Now(
            msg.sender, // depositor
            address(this), // recipient (this contract from other chain)
            _token,
            address(0),
            msgData.inputAmount,
            msgData.outputAmount,
            uint256(_dstChainId),
            msgData.relayer,
            msgData.deadline,
            msgData.exclusivityDeadline,
            abi.encode(_market)
        );
    }
```

based on the Across Documentation, an expired deposits to the pool for the cross-chain operation is refundable.

> When an unfilled deposit's `fillDeadline` exceeds the destination chain's `block.timestamp`, it is no longer fillable–the `SpokePool` contract on the destination chain will revert on a `fillRelay()` call.
>
> Expired deposits are technically refunded by the next root bundle proposed to the HubPool containing an expired deposit refund.

however, we notice that there's no function in the contract `AcrossBridge` that could transfer token back to rebalancer if a refund happens in the `depositV3Now` function, as stated in the documentation above.

Impact:

if a refund happens in `depositV3Now`, the token will be transfered back the `AcrossBridge` contract. however there's no function in this contract that can send the refunded token to the rebalancer, causing the refunded token to be permanently stuck in the `AcrossBridge`

Recommended Mitigation:

add an additional function that can send the refunded token back to the rebalancer (or market)

```
+    function recover(address _market) external onlyRebalancer {
+        IERC20 _underlying = ImTokenMinimal(_market).underlying();
+        _underlying.safeTransfer(
+            _market,
+            _underlying.balanceOf(address(this))
+        );
+    }
```

[M-14] unsupported token used

Description:

below is a part of the README given by the Malda

> Tokens at launch:
>
> - Stables: USDC, USDT, USDS
> - Bluechips: wBTC, wETH
> - LSTs: wstETH, weETH, ezETH, wrsETH/rsETH on mainnet
>
> Post-launch:
>
> - Stables: USDe, sUSDe
> - Others: Lombard BTC, fiammaBTC, ARB, OP, LINEA, GMX, ZKC (Boundless Token), AERO

we realized that Malda is using 2 protocols to conduct cross-chain operation: - Everclear Protocol - Across Protocol

Based on Everclear Documentation, it does not support: `USDS`, `wBTC`, `wstETH`, `weETH`, `ezETH`, `wrsETH`, `rsETH`, `USDe`, `sUSDe`, `Lombard BTC`, `fiammaBTC`, `ARB`, `OP`, `LINEA`, `GMX`, `ZKC` (Boundless Token), `AERO`

Based on Across Documentation, it does not support: `USDS`, `wstETH`, `weETH`, `ezETH`, `wrsETH`, `rsETH`, `USDe`, `sUSDe`, `Lombard BTC`, `fiammaBTC`, `ARB`, `OP`, `LINEA`, `GMX`, `ZKC` (Boundless Token), `AERO`

Impact:

An attempt to perform cross-chain in either of the protocol using the unsupported token will result in DoS

Recommended Mitigation:

two options: 1. Remove unsupported token to work with 2. Register unsupproted token on those protocol

[M-15] missing validation on `ttl` and `maxFee`

Description:

based on the Everclear Documentation, we notice that `maxFee` and `ttl` should be 0 when passing into `newIntent`.

> The `maxFee` field should **always be specified as 0** as `maxFee` is only applicable in cases where an order should be routed to the solver pathway and does not apply to the netting pathway.
>
> The `ttl` input should **always be specified as 0** to indicate the order should be routed via the netting system on the Hub. When `ttl` is non-zero an order is routed via a separate solver pathway where the intent creator requires a dedicated solver to fill the intent for a fee. This pathway will not be supported at launch; Rebalancers must ensure all netting orders always specify `ttl` as 0.

`EverclearBridge::sendMsg`:

```
    function sendMsg(
        uint256 _extractedAmount,
        address _market,
        uint32 _dstChainId,
        address _token,
        bytes memory _message,
        bytes memory // unused
    ) external payable onlyRebalancer {
        IntentParams memory params = _decodeIntent(_message);

        // ...

        SafeApprove.safeApprove(...);
        (bytes32 id,) = everclearFeeAdapter.newIntent(
            params.destinations,
            params.receiver,
            params.inputAsset,
            params.outputAsset,
            params.amount,
@>          params.maxFee,
@>          params.ttl,
            params.data,
            params.feeParams
        );
        emit MsgSent(_dstChainId, _market, params.amount, id);
    }
```

however, there's no 0 validation checks on both `params.maxFee` and `params.ttl` in the `sendMsg` function before calling `newIntent`.

Impact:

an attempt to input a non-zero value for both `params.maxFee` and `params.ttl` will likely to get reverted in `newIntent`

Recommended Mitigation:

add an additional validation to ensures that both `params.maxFee` and `params.ttl` are 0

```
    function sendMsg(...) external payable onlyRebalancer {
        IntentParams memory params = _decodeIntent(_message);

+       require(params.maxFee == 0 && params.ttl == 0, ...());
        // ...
    }
```