SHERLOCK

Security Review For Malda

Public Best Efforts Audit Contest Prepared For: **Malda**
Lead Security Expert: **cergyk**
Date Audited: **July 24 - August 14, 2025**

Introduction

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.

Scope

Repository: malda-protocol/malda-lending

Audited Commit: 0f62f27fbfc8a69e256dbbd45244450c9468cd80

Final Commit: 7316e565ee97797318891fdd486d2da0ea0e63f5

Files:

- src/interfaces/IOperator.sol
- src/interfaces/IPauser.sol
- src/interfaces/IRebalancer.sol
- src/interfaces/IRewardDistributor.sol
- src/libraries/SafeApprove.sol
- src/migration/IMigrator.sol
- src/migration/Migrator.sol
- src/mToken/BatchSubmitter.sol
- src/mToken/extension/mTokenGateway.sol
- src/mToken/host/mErc20Host.sol
- src/mToken/mErc20Immutable.sol
- src/mToken/mErc20.sol
- src/mToken/mErc20Upgradable.sol
- src/mToken/mTokenConfiguration.sol
- src/mToken/mToken.sol
- src/mToken/mTokenStorage.sol
- src/Operator/Operator.sol
- src/Operator/OperatorStorage.sol
- src/oracles/MixedPriceOracleV3.sol
- src/oracles/MixedPriceOracleV4.sol
- src/pauser/Pauser.sol
- src/rebalancer/bridges/AcrossBridge.sol

- src/rebalancer/bridges/BaseBridge.sol
- src/rebalancer/bridges/ConnextBridge.sol
- src/rebalancer/bridges/EverclearBridge.sol
- src/rebalancer/bridges/LZBridge.sol
- src/rebalancer/Rebalancer.sol
- src/Roles.sol
- src/utils/ExponentialNoError.sol
- src/utils/WrapAndSupply.sol
- src/verifier/ZkVerifier.sol

Repository: malda-protocol/malda-zk-coprocessor

Audited Commit: b5b2fd3ffa7bfff8a55f1a9b65234875f8b43dcb

Final Commit: 813060dd27ad8658a2e6009260b05e69bafaab8d

Files:

- malda_rs/src/constants.rs
- malda_rs/src/elfs_ids.rs
- malda_rs/src/lib.rs
- malda_rs/src/viewcalls.rs
- malda_utils/src/constants.rs
- malda_utils/src/cryptography.rs
- malda_utils/src/lib.rs
- malda_utils/src/types.rs
- malda_utils/src/validators.rs
- methods/build.rs
- methods/guest/src/bin/get_proof_data.rs
- methods/src/lib.rs

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.

Issues Found

High	Medium
1	16

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

0x213
0xBlackie
0xDemon
0xEkko
0xKann
0xShoonya
0xSolus
0xapple
0xfocusNode
0xlucky
0xmechanic
0xsai
10ap17
5am
7
8olidity
Angry_Mustache_Man
BADROBINX
Bizarro
Bobai23
BusinessShotgun
Cybrid
DeveloperX
EQUATION
ExtraCaterpillar
HeckerTrieuTien
IvanFitro
Ivcho332
JeRRy0422
KiroBrejka
Kvar
LeFy
MacroWang
PeterSR
PratRed
RaiseUp
Richard796
Sa1ntRobi
SafetyBytes
Sevyn
Sparrow_Jac
TECHFUND-inc
TopStar
WillyCode20
Yaneca_b
ZanyBonzy
ZeroTrust
Ziusz
axelot
befree3x
blockace
bube
bulgari
cergyk
coin2own
dan__vinci
dany.armstrong90
davies0212
dimah7
dimulski
djshaneden
dreamcoder
elolpuer
elyas
farismaulana
gh0xt
har0507
harry
holtzzx

joicygiore
kelvinclassic11
magbeans9
mahdifa
maigadoh
maxim371
molaratai
mussucal
onudasatoshi
oxelmiguel
oxwhite
pashap9990
pollersan
pyk
sakibcy
sheep
softdev0323
swarun
teoslaf1
v10g1
vangrim
weblogicctf
who_is_rp
yoooo
zach223

Issue H-1: Rebalancer can steal funds from markets by sending to custom receiver through Everclear Bridge

Source: https://github.com/sherlock-audit/2025-07-malda-judging/issues/124

Found by

Angry_Mustache_Man, ExtraCaterpillar, SafetyBytes, Sevyn, Ziusz, axelot, bulgari, cergyk, elolpuer

Description

The rebalancer contract is called on `sendMsg` with a message encoded as bytes (`_msg.message`). This message is specific for the bridge contract, and in the case of `EverclearBridge` it is of the form `IntentParams`:

EverclearBridge.sol#L40-L50:

```
struct IntentParams {
    uint32[] destinations;
    //@audit receiver is left unchecked
    bytes32 receiver;
    address inputAsset;
    bytes32 outputAsset;
    uint256 amount;
    uint24 maxFee;
    uint48 ttl;
    bytes data;
    IFeeAdapter.FeeParams feeParams;
}
```

We notice that the parameter `receiver` is left entirely unchecked, and the rebalancer EOA can provide an address controlled by it, stealing the user funds which should be send for rebalancing.

EverclearBridge.sol#L111-L121:

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

Recommendation

Check that `params.receiver` is a valid market on the destination chain

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/malda-protocol/malda-lending/pull/103

CergyK

Fix looks good, now `params.receiver` is forced to be equal to `_market`

Issue M-1: Wrong direction of rounding in redeem may lead to drain if exchange rate grows large

Source: https://github.com/sherlock-audit/2025-07-malda-judging/issues/81

Found by

cergyk

Description

In the mToken contract, forked from compound v2, the same wrong direction rounding is introduced in the `__redeem` function:

mToken.sol#L614-L630:

```
    if (redeemTokensIn > 0) {
      /*
       * We calculate the exchange rate and the amount of underlying to be
        ↪  redeemed:
       *  redeemTokens = redeemTokensIn
       *  redeemAmount = redeemTokensIn x exchangeRateCurrent
       */
      redeemTokens = redeemTokensIn;
      redeemAmount = mul_ScalarTruncate(exchangeRate, redeemTokensIn);
    } else {
      /*
       * We get the current exchange rate and calculate the amount to be
        ↪  redeemed:
       *  redeemTokens = redeemAmountIn / exchangeRate
       *  redeemAmount = redeemAmountIn
       */
>>    redeemTokens = div_(redeemAmountIn, exchangeRate); //@audit should
↪  round up, as this is the amount of tokens pulled from user
      redeemAmount = redeemAmountIn;
    }
```

This wrong direction of rounding leads to pull less shares from the user while the amount sent out is the same as requested. In most cases the losses should be negligible (1 wei of shares per operation), because the division operates at 1e18 precision, and the exchange rate starts as a small value initially (0.02).

Inflating exchange rate

We will show that the first depositor is able to increase the exchange rate by an arbitrary factor, by using the rounding against him during `mint` and `redeem`(shares), when supply is initially zero. As can be seen in the mint function:

mToken.sol#L699-L707:

```
uint256 mintTokens = div_(actualMintAmount, exchangeRate);
require(mintTokens >= minAmountOut, mt_MinAmountNotValid());

// avoid exchangeRate manipulation
if (totalSupply == 0) {
    //@audit 1000 of shares are minted to the zero address
    totalSupply = 1000;
    accountTokens[address(0)] = 1000;
    //@audit underflow if initial mint is below 1000
    mintTokens -= 1000;
}
```

When the totalSupply is zero, at least 1000 shares have to be minted initially, which means that 20 wei of token has to be supplied (due to the 0.02 initial exchange rate).

Reach exchange rate of exp Then the first depositor repeats the steps:

1. Mint by providing 1 wei of underlying (mints 50 shares to the depositor)
2. Redeem 25 shares *2 -> sends 0 underlying to the depositor

each time these steps are repeated, the `totalUnderlying` of the market is increased by 1 while `totalSupply` is unchanged. This means the `exchangeRate` is increasing. We need to repeat this 980 times to reach `echangeRate == exp`

> As a side note, the host chain here is `Linea`, and although repeating these steps many times costs a lot of gas (around 50M), the gas would be very cheap; Such a step of 1000 iterations only costs around 0.20$ at current ethereum price. Also please note that the gas limit by block is very large (2B).

Increase exchange rate exponentially When `exchangeRate > exp`, the process is simpler we only need to provide 1 wei and 0 shares are minted. From now on, we double the exchange rate at each step of 1000 iterations:

Indeed when exchange rate is X*exp, we can call mint with X wei, which will mint 0 shares.

> With this in mind and taking the example of USDC, reaching an exchange rate value of 1e6*exp, would take approx 20 iterations and cost ~5$ of gas, plus the cost of donating 1000$ of token to the market.

Target exchange rate of 1e6*exp reached The attacker just has to wait for new depositors to supply into the market, and then he can call redeemUnderlying 1e6 by 1e6, which will burn zero shares.

Recommendation

Consider fixing the rounding in redeem, make the division by exchange rate round up: mTokenn.sol#L614-L630:

```
        } else {
            /*
             * We get the current exchange rate and calculate the amount to be
              ↪  redeemed:
             *  redeemTokens = redeemAmountIn / exchangeRate
             *  redeemAmount = redeemAmountIn
             */
-           redeemTokens = div_(redeemAmountIn, exchangeRate);
+           redeemTokens = divUp_(redeemAmountIn, exchangeRate);
            redeemAmount = redeemAmountIn;
        }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/malda-protocol/malda-lending/pull/117

CergyK

Fix looks good, redeem underlying now rounds up, in favor of the protocol

Issue M-2: First depositor can brick market by forcing very large borrow rate

Source: https://github.com/sherlock-audit/2025-07-malda-judging/issues/86

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

Bobai23, Kvar, LeFy, cergyk, dan__vinci

Description

The borrow rate is computed in `JumpRateModelV4` as a function of the utilisation:

JumpRateModelV4.sol#L140-L150:

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


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

Utilization rate being defined as:

JumpRateModelV4.sol#L127-L135:

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

We notice that utilizationRate can be greater than 1e18, when `cash + borrows - reserves` < `borrows`. In the following section, we will show a scenario where the ratio can become very big.

First depositor manipulation Assuming a 18 decimals market:

1. Alice the first depositor can mint for 1e18 tokens, and borrow it all (by having additional collateral in another market):
    - cash = 0
    - totalBorrows = 1e18
    - reserves = 0
2. Alice waits for a few blocks for some interest to accrue:
    - cash = 0
    - totalBorrows = 1e18 + 1e12
    - reserves = 5e11 (assuming 50% reserve factor)
3. Alice repays totalBorrows minus reserves minus 1:
    - cash = 1e18 + 5e11 - 1
    - totalBorrows = 5e11 + 1
    - reserves = 5e11
4. Alice redeems the total supply
    - cash = 0
    - totalBorrows = 5e11 + 1
    - reserves = 5e11

utilization rate is very big: `5e11`. As a result the `getBorrowRate` is bigger than `0.0005e16` (initial max borrow value):

mToken.sol#L36-L39:

```
constructor() {
    borrowRateMaxMantissa = 0.0005e16;
}
```

As a result the market is completely bricked, because `_accrueInterest` always reverts: mTokenStorage.sol#L351-L356:

```
//@audit in _accrueInterest():
/* Calculate the current borrow interest rate */
uint256 borrowRateMantissa =
    IInterestRateModel(interestRateModel).getBorrowRate(cashPrior, borrowsPrior,
     ↪  reservesPrior);
if (borrowRateMaxMantissa > 0) {
    require(borrowRateMantissa <= borrowRateMaxMantissa, mt_BorrowRateTooHigh());
}
```

Indeed `_accrueInterest` is called for all operations, even for updating the `interestRateModel`.

Recommendation

Consider bounding the utilization to be at most 1e18 (100%), which would avoid having degenerate borrow rate values.

Issue M-3: Migrator severily underestimates slippage by using underlying instead of shares

Source: https://github.com/sherlock-audit/2025-07-malda-judging/issues/91

Found by

10ap17, Angry_Mustache_Man, LeFy, WillyCode20, cergyk, elolpuer, pashap9990, vangrim

Description

`Migrator.sol` implements a slippage protection:

Migrator.sol#L90-L100:

```
// 2. Mint mTokens in all v2 markets
for (uint256 i; i < posLength; ++i) {
    Position memory position = positions[i];
    if (position.collateralUnderlyingAmount > 0) {
        //@audit compute slippage value
        uint256 minCollateral =
            position.collateralUnderlyingAmount -
             ↪   (position.collateralUnderlyingAmount * 1e4 / 1e5);
        ImErc20Host(position.maldaMarket).mintOrBorrowMigration(
            true, position.collateralUnderlyingAmount, msg.sender, address(0),
             ↪   minCollateral
        );
    }
}
```

Unfortunately, the slippage value computed uses underlying tokens, whereas the shares amount should be used. Since exchange rate should be close to 0.02, the slippage parameter is wrong by at least an order of magnitude.

Recommendation

Underlying amount should be divided by exchange rate to have a correct slippage value: Migrator.sol#L90-L100:

```
    // 2. Mint mTokens in all v2 markets
    for (uint256 i; i < posLength; ++i) {
        Position memory position = positions[i];
        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
            );
        }
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/malda-protocol/malda-lending/pull/120

CergyK

Fix looks good, underlying was converted to shares using exchangeRate

Issue M-4: Blacklist can be completely bypassed on outHere endpoint in mTokenGateway

Source: https://github.com/sherlock-audit/2025-07-malda-judging/issues/96

Found by

0xEkko, 0xKann, 5am, 8olidity, JeRRy0422, Kvar, PeterSR, TopStar, ZanyBonzy, Ziusz, blockace, bulgari, cergyk, dimulski, gh0xt, joicygiore, kelvinclassic11

Description

The mTokenGateway exposes the `outHere` endpoint to allow withdrawal on an extension chain. This endpoint is protected by two `ifNotBlacklisted` modifiers:

mTokenGateway.sol#L254-L262:

```
/**
 * @inheritdoc ImTokenGateway
 */
function outHere(bytes calldata journalData, bytes calldata seal, uint256[]
↪   calldata amounts, address receiver)
    external
    notPaused(OperationType.AmountOutHere)
    ifNotBlacklisted(msg.sender)
    ifNotBlacklisted(receiver)
{
```

Unfortunately both can be bypassed, because:

1. `msg.sender` can be any address in `allowedCallers` for the `_sender` in the journal data
2. receiver value passed to the endpoint is overwritten by `_sender` in `_outHere`:

   mTokenGateway.sol#L281-L286:

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


       // temporary overwrite; will be removed in future implementations
       receiver = _sender;
   ```

So a blacklisted user can still use `outHere` to withdraw their funds.

Recommendation

Add an additional check for _sender to not be in blacklist:

mTokenGateway.sol#L281-L286:

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


        // temporary overwrite; will be removed in future implementations
        receiver = _sender;
+       require (!blacklistOperator.isBlacklisted(_sender),
 ↪  mTokenGateway_UserBlacklisted());
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/malda-protocol/malda-lending/pull/112

CergyK

Fix looks good

Issue M-5: Rebalancer can send to unallowed destination chains through EverclearBridge

Source: https://github.com/sherlock-audit/2025-07-malda-judging/issues/123

Found by

cergyk

Description

The rebalancer contract is called on `sendMsg` with a message encoded as bytes (`_msg.message`). This message is specific for the bridge contract, and in the case of `EverclearBridge` it is of the form `IntentParams`:

EverclearBridge.sol#L40-L50:

```
struct IntentParams {
    uint32[] destinations;
    bytes32 receiver;
    address inputAsset;
    bytes32 outputAsset;
    uint256 amount;
    uint24 maxFee;
    uint48 ttl;
    bytes data;
    IFeeAdapter.FeeParams feeParams;
}
```

We notice that multiple destination chains can be provided (`destinations`); Out of these chains, it is only checked that one is the `_dstChainId` provided as argument of `sendMsg`:

EverclearBridge.sol#L95-L102:

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

However, the intent is created on Everclear using all the provided destinations, and can be fulfilled on any of the chains of the list. If it is fulfilled on a chain that Malda does not handle, the funds are lost.

Recommendation

`destinations` should be checked to be of length 1, and that `destinations[0] == _dstChainId`

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/malda-protocol/malda-lending/pull/119/files

CergyK

Fix looks good, only 1 destination chain allowed now

Issue M-6: `EverclearBridge` does not pull tokens from the Rebalancer, causing all rebalancing operations to fail

Source: https://github.com/sherlock-audit/2025-07-malda-judging/issues/177

Found by

0xDemon, 0xmechanic, 0xsai, 10ap17, Angry_Mustache_Man, BusinessShotgun, Cybrid, ExtraCaterpillar, IvanFitro, Sa1ntRobi, SafetyBytes, ZanyBonzy, ZeroTrust, Ziusz, bube, bulgari, dan__vinci, davies0212, maigadoh, oxelmiguel, vangrim

Summary

The `EverclearBridge` contract does not transfer tokens from the Rebalancer contract before attempting to perform bridging operations. It assumes custody of tokens without actually calling `safeTransferFrom` to pull tokens from the Rebalancer. As a result, the bridge contract does not have the required tokens, causing all rebalancing attempts to fail.

Root Cause

In `EverclearBridge.sol`, the sendMsg function does not call `IERC20(_token).safeTransferFrom(msg.sender, address(this), params.amount)` to transfer tokens from the Rebalancer to itself. Without this transfer, the bridge cannot perform any bridging logic that requires token custody.

Internal Pre-conditions

1. The Rebalancer approves the bridge contract to spend tokens.
2. The Rebalancer calls `sendMsg` on the bridge contract without transferring tokens.

External Pre-conditions

None.

Attack Path

1. The Rebalancer approves the bridge contract.

2. The bridge contract attempts to perform bridging logic but fails due to lack of token custody.
3. No tokens are bridged and rebalancing fails.

Impact

All rebalancing operations using the bridge contract will fail, resulting in a complete denial of service for cross-chain liquidity movement. This can halt protocol operations that depend on rebalancing and may lead to liquidity fragmentation or loss of protocol functionality.

PoC

No response

Mitigation

Update the `EverclearBridge` contract to call `safeTransferFrom` and pull the required tokens from the Rebalancer before proceeding with bridging logic.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/malda-protocol/malda-lending/pull/105

CergyK

Fix looks good. Tokens are now pulled from Rebalancer into EverclearBridge

Issue M-7: Incorrect transfer size validation after time window reset can lead to rebalancing DoS

Source: https://github.com/sherlock-audit/2025-07-malda-judging/issues/234

Found by

0x213, 0xBlackie, 0xEkko, 0xShoonya, 0xSolus, 0xapple, 0xmechanic, 10ap17, 5am, 7, 8olidity, BADROBINX, Bizarro, Cybrid, DeveloperX, EQUATION, HeckerTrieuTien, Ivcho332, KiroBrejka, LeFy, MacroWang, PratRed, RaiseUp, Sparrow_Jac, WillyCode20, Yaneca_b, ZanyBonzy, Ziusz, axelot, coin2own, dan__vinci, davies0212, djshaneden, dreamcoder, elolpuer, elyas, farismaulana, gh0xt, har0507, harry, magbeans9, mahdifa, maxim371, onudasatoshi, oxelmiguel, oxwhite, pollersan, sheep, softdev0323, swarun, teoslaf1, v10g1, vangrim, who_is_rp, yoooo, zach223

Summary

Using stale `TransferInfo` in the `Rebalancer::sendMsg` function will cause a false `Rebalancer_TransferSizeExcedeed` revert for EOA rebalancers as in normal use flow, the contract checks the old transfer size even after resetting it for a new time window, leading to unexpected reverts and potential disruption of rebalancing operations.

```
TransferInfo memory transferInfo = currentTransferSize[_msg.dstChainId][_msg.token];
 uint256 transferSizeDeadline = transferInfo.timestamp + transferTimeWindow;
 if (transferSizeDeadline < block.timestamp) {
     currentTransferSize[_msg.dstChainId][_msg.token] = TransferInfo(_amount,
      ↪  block.timestamp);
 } else {
     currentTransferSize[_msg.dstChainId][_msg.token].size += _amount;
 }

 uint256 _maxTransferSize = maxTransferSizes[_msg.dstChainId][_msg.token];
 if (_maxTransferSize > 0) {
     require(transferInfo.size + _amount < _maxTransferSize,
      ↪  Rebalancer_TransferSizeExcedeed());
     //@audit if `transferSizeDeadline < block.timestamp` than `transferInfo` won't
      ↪  be updated but we changed `currentTransferSize` to be
      ↪  `TransferInfo(_amount, block.timestamp)` so we will check using the old
      ↪  transfer size + amount instead in this case using only amount which will
      ↪  lead to unexpected DoS
 }
```

Root Cause

In `Rebalancer.sol`, the `sendMsg` function resets `currentTransferSize[_msg.dstChainId][_msg.token]` to `TransferInfo(_amount, block.timestamp)` when the time window expires, but still uses the old `transferInfo.size` for the max transfer size check (`require(transferInfo.size + _amount < _maxTransferSize)`), causing incorrect validation and potential DoS.

https://github.com/sherlock-audit/2025-07-malda/blob/main/malda-lending/src/rebalancer/Rebalancer.sol#L141-L152

Internal Pre-conditions

1. In normal use, an authorized rebalancer performs transfers that bring the recorded `currentTransferSize` close to the maximum allowed for a token and destination chain within the current time window.
2. Once the time window expires, the next transfer resets `currentTransferSize` to the new transfer amount with an updated timestamp.
3. However, during this next transfer, the contract still checks the sum of the old (stale) transfer size plus the new amount against the max limit, causing an unexpected revert and blocking valid transfers.

External Pre-conditions

none

Attack Path

1. An authorized rebalancer calls `sendMsg` multiple times within a single `transferTimeWindow`, pushing `currentTransferSize[_dstChainId][_token].size` close to `maxTransferSizes[_dstChainId][_token]`.
2. The `transferTimeWindow` expires without further transfers, so `currentTransferSize[_dstChainId][_token]` remains at a high value but stale.
3. On the first `sendMsg` call after the window resets, the contract resets `currentTransferSize` to the new amount but still validates using the old (stale) `transferInfo.size`.
4. Because the validation uses the stale size, the contract rejects the transfer by reverting with `Rebalancer_TransferSizeExcedeed`, even though the new window should allow it.
5. This causes a denial of service for rebalancing operations on that token and destination chain until the next window reset or state update.

Impact

The authorized rebalancers cannot execute valid rebalancing transfers for certain tokens and destination chains immediately after a transfer time window resets. This causes a denial of service in normal use flow, disrupting expected protocol operations without any malicious actor involved.

PoC

No response

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/malda-protocol/malda-lending/pull/138

CergyK

Fix looks good

Issue M-8: mErc20Host: It is not possible to permissionlessly call "external" endpoints when source chain is Eth mainnet, because l1Inclusion flag cannot be set to true

Source: https://github.com/sherlock-audit/2025-07-malda-judging/issues/264

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

LeFy, cergyk, pyk

Description

We see that in order to submit a proof to the `malda-lending` contracts permissionlessly (user is not proof forwarder or batch proof forwarder), the l1Inclusion flag has to be set:

mErc20Host.sol#L378-L399:

```
    function _verifyProof(bytes calldata journalData, bytes calldata seal) internal
     ↪  view {
        require(journalData.length > 0, mErc20Host_JournalNotValid());


        // Decode the dynamic array of journals.
        bytes[] memory journals = _decodeJournals(journalData);


        // Check the L1Inclusion flag for each journal.
        bool isSequencer = _isAllowedFor(msg.sender, _getProofForwarderRole())
            || _isAllowedFor(msg.sender, _getBatchProofForwarderRole());


        if (!isSequencer) {
            for (uint256 i = 0; i < journals.length; i++) {
>               (,,,,,, bool L1Inclusion) =
 ↪  mTokenProofDecoderLib.decodeJournal(journals[i]);
>               if (!L1Inclusion) {
>                   revert mErc20Host_L1InclusionRequired();
>               }
            }
        }


        // verify it using the IZkVerifier contract
        verifier.verifyInput(journalData, seal);
    }
```

However if attempting to request a proof for data originating from the ETH mainnet, with `l1Inclusion` flag set, it will panic in `viewcalls.rs`, during the `get_env_input_for_l1_inclusion_and_l2_block_number` call:

viewcalls.rs#L933-L970:

```
pub async fn get_env_input_for_l1_inclusion_and_l2_block_number(
    chain_id: u64,
    is_sepolia: bool,
    l1_inclusion: bool,
    ethereum_block: Option<u64>,
    fallback: bool,
) -> (Option<EvmInput<EthEvmFactory>>, Option<u64>) {
    if !l1_inclusion {
        // If L1 inclusion is not required, return None for both values
        (None, None)
    } else {
        //@audit handle the l1_inclusion == true case
        // Prepare the L1 RPC URL
        let l1_rpc_url = get_rpc_url("ETHEREUM", fallback, is_sepolia);
        // Determine the L1 block to use for inclusion
        let l1_block = if is_linea_chain(chain_id) {
            ethereum_block.unwrap()
        } else {
            if is_sepolia {
                ethereum_block.unwrap() - REORG_PROTECTION_DEPTH_ETHEREUM_SEPOLIA
            } else if !is_sepolia {
                ethereum_block.unwrap() - REORG_PROTECTION_DEPTH_ETHEREUM
            } else {
                panic!("Invalid chain ID");
            }
        };


        // Delegate to the appropriate helper based on chain type
        if is_opstack_chain(chain_id) {
            get_env_input_for_opstack_dispute_game(chain_id, l1_block,
             ↪  fallback).await
        } else if is_linea_chain(chain_id) {
            get_env_input_for_linea_l1_call(chain_id, l1_rpc_url, l1_block).await
        } else {
            //@audit if ETH mainnet && l1_inclusion, panic
>           panic!(
>               "L1 Inclusion only supported for Optimism, Base, Linea and their
 ↪  Sepolia variants"
>           );
        }
    }
}
```

Impact

It is impossible to carry some `external` actions on host in a permissionless way (caller is not sequencer), when source chain is Eth mainnet. The impact is even more acute since there is no clear path to trigger some external endpoints for execution by the sequencer (see issue about liquidateExternal).

Likelihood

No preconditions, the bug is inevitable

Recommendation

The process of validating the data (guest program, logic of which is in `validators.rs`) allows for validating a proof with `l1Inclusion` set to true for ETH_MAINNET, it only requires the `env_input_eth_for_l1_inclusion` to be `Some` (this value is not used anymore after determining `validate_l1_inclusion`).

malda-zk-coprocessor/malda_utils/src/validators.rs#L195:

```
let validate_l1_inclusion = env_input_eth_for_l1_inclusion.is_some();
```

So we should just handle the ethereum case by returning a default ENV value:

```
        // Delegate to the appropriate helper based on chain type
        if is_opstack_chain(chain_id) {
            get_env_input_for_opstack_dispute_game(chain_id, l1_block,
             ↪  fallback).await
        } else if is_linea_chain(chain_id) {
            get_env_input_for_linea_l1_call(chain_id, l1_rpc_url, l1_block).await
-       } else {
-          panic!(
-              "L1 Inclusion only supported for Optimism, Base, Linea and their
 ↪  Sepolia variants"
-          );
+       } else if is_ethereum(chain_id) {
+          Env::default();
+       } else {
+          panic!(
+              "L1 Inclusion only supported for Optimism, Base, Linea and their
 ↪  Sepolia variants"
+          );
+       }
```

> If disallowing l1Inclusion for Eth mainnet is really intended, it should then also be enforced in the guest program, which is not the case currently, and any user can provide alternative inputs to prove for Eth mainnet with l1Inclusion, as shown above (set parameter `env_input_eth_for_l1_inclusion` to a dummy non-empty value).

In that case, one can change the contracts to accept proofs which do not have l1Inclusion in case chainId is the one from Eth mainnet:

mErc20Host.sol#L378-L399:

```
function _verifyProof(bytes calldata journalData, bytes calldata seal) internal
 ↪  view {
    require(journalData.length > 0, mErc20Host_JournalNotValid());


    // Decode the dynamic array of journals.
    bytes[] memory journals = _decodeJournals(journalData);


    // Check the L1Inclusion flag for each journal.
    bool isSequencer = _isAllowedFor(msg.sender, _getProofForwarderRole())
        || _isAllowedFor(msg.sender, _getBatchProofForwarderRole());


    if (!isSequencer) {
        for (uint256 i = 0; i < journals.length; i++) {
-           (,,,,,, bool L1Inclusion) =
 ↪  mTokenProofDecoderLib.decodeJournal(journals[i]);
-           if (!L1Inclusion) {
+           (,,,,uint chainId,, bool L1Inclusion) =
 ↪  mTokenProofDecoderLib.decodeJournal(journals[i]);
+           if (!L1Inclusion && !(chainId == ETHEREUM_MAINNET)) {
                revert mErc20Host_L1InclusionRequired();
            }
        }
    }


    // verify it using the IZkVerifier contract
    verifier.verifyInput(journalData, seal);
}
```

Discussion

Barnadrot

Acknowledged:

We wont fix as the intended/crucial and required use-case for self-sequencing is permissionless exit which requires proving Linea state.

The fix is extensive and we have further proof upgrades required by external circumstances, therefore this is going to be remedied in a future version where self-sequencing latency and centralized sequencer latency is closer together.

Issue M-9: The rebalancing system is broken when everclear bridge contract is used

Source: https://github.com/sherlock-audit/2025-07-malda-judging/issues/304

Found by

0xmechanic, 10ap17, Angry_Mustache_Man, Cybrid, ExtraCaterpillar, PeterSR, ZanyBonzy, bube, bulgari, dan__vinci, elolpuer, farismaulana, oxwhite

Summary

The sendMsg in EverclearBridge contract only approves the **transfer amount** (not including the fee) for the **FeeAdapter** when calling `newIntent`. However, the **FeeAdapter** attempts to pull **both the amount and the fee** from the bridge contract. This results in a revert due to insufficient allowance causing cross-chain rebalancing to fail.

Root Cause

The root case stem from a mismatch between the **approved amount** and the **actual amount required** by the `FeeAdaptor`.

The sendMsg() in everclearBridge.sol is as follows:

```
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());

       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());

    if (_extractedAmount > params.amount) {
      uint256 toReturn = _extractedAmount - params.amount;
@>      IERC20(_token).safeTransfer(_market, toReturn);//@audit-info the
↪ excess is sent back to the market
      emit RebalancingReturnedToMarket(_market, toReturn, _extractedAmount);
    }

@>    SafeApprove.safeApprove(params.inputAsset, address(everclearFeeAdapter),
↪ params.amount);//@audit approve is done for amount
    (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);
  }
```

The newIntent method in feeAdaptor contract is defined as:

```
/// @inheritdoc IFeeAdapter
 function newIntent(
  uint32[] memory _destinations,
  bytes32 _receiver,
  address _inputAsset,
  bytes32 _outputAsset,
  uint256 _amount,
  uint24 _maxFee,
  uint48 _ttl,
  bytes calldata _data,
  IFeeAdapter.FeeParams calldata _feeParams
 ) external payable returns (bytes32 _intentId, IEverclear.Intent memory _intent) {
  // Transfer from caller
@> _pullTokens(msg.sender, _inputAsset, _amount + _feeParams.fee);//@audit
↪ amount+fee is pulled from bridge contract

  // Create intent
  (_intentId, _intent) =
   _newIntent(_destinations, _receiver, _inputAsset, _outputAsset, _amount,
    ↪ _maxFee, _ttl, _data, _feeParams);
 }
```

As shown, `_amount + _feeParams.fee` is pulled from bridge contract, which will result in revert as the approve was done only for the specified amount.

Internal Pre-conditions

There is an imbalance and `sendMsg` is invoked in the rebalancer contract.

External Pre-conditions

Market B requires rebalancing Assets are available in Market A

Attack Path

1. Market A on X chain has **12,000 USDC** extractable value.
2. Market B on Y chain needs **10,000 USDC**.
3. The `sendMsg()` function is invoked in the `Rebalancer` contract with `_amount = 12,000 USDC`.
4. The **12,000 USDC** is sent to the `EverclearBridge` contract. Since the extracted amount is greater than the amount specified in the message, the excess **2,000 USDC** is sent back to Market A.
5. The `FeeAdaptor` contract is **approved** for only **10,000 USDC**.
6. The `newIntent()` function is called on the `FeeAdaptor`. This function attempts to **pull** `10,000 USDC + fee`, but it **reverts** due to insufficient approval. As result, the rebalancing will not succeed.

Additinal Notes: According to the FeeAdaptor and Spoke contract logic in everclear platform, the fee is not deducted from the specified transfer amount but instead added on top during token transfer. Therefore, the fee cannot be zero, and any under-approval results in failure. If the fee were instead deducted from the approved amount, Market B would receive less than 10,000 USDC, potentially leading to issues like insufficient liquidity.

Impact

All cross-chain rebalancing operations using EverclearBridge will fail.

PoC

See the function flow and contract state change given above

Mitigation

Update the allowance logic in EverclearBridge to approve enough amount for the FeeAdapter.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/malda-protocol/malda-lending/pull/114/files

CergyK

Fix looks good

Issue M-10: Unenforced maxFee and ttl Parameters in sendMsg Function

Source: https://github.com/sherlock-audit/2025-07-malda-judging/issues/317

Found by

0xsai, yoooo

Summary

In the EverclearBridge which is used by rebalancer to send cross-chain intents using `Everclear` does not enforce that the `maxFee` and `ttl` parameters are set to 0, as required for the netting pathway. The Everclear documentation specifies that rebalancers must use the netting pathway, where maxFee == 0 (no solver fees) and ttl == 0 (immediate processing by the hub). However, the contract passes these parameters to everclearFeeAdapter.newIntent without validation, allowing non-zero values that could misroute intents to the unsupported solver pathway.

Root Cause

The `sendMsg` function in `EverclearBridge.sol` lacks validation checks to ensure `params.maxFee == 0 and params.ttl == 0` before calling everclearFeeAdapter.newIntent as specified in the everclear docs..

```
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());


    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());


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


    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
    );
    emit MsgSent(_dstChainId, _market, params.amount, id);
}
```

Internal Pre-conditions

1. Already does not validates `maxFee` and `ttl`

External Pre-conditions

1. both `maxFee` and `ttl` being none zero

Attack Path

1. A rebalancer calls the `sendMsg` to create a cross-chain intent with both the `maxFee` and `ttl` being none zero
2. The _decodeIntent function decodes the message into IntentParams, extracting maxFee (type uint24) and ttl (type uint48).

3. The FeeAdapter may interpret non-zero maxFee or ttl as a solver pathway intent, misrouting the intent to solvers instead of the netting system.
4. Since the solver pathway is not supported at launch, the intent may fail to settle, or, if solvers are active, it may incur unexpected fees or settle on a suboptimal chain.

Impact

1. A potential DOS
2. Non-zero maxFee or ttl could route intents to the solver pathway, which is unsupported, leading to settlement failures or delays that disrupt liquidity rebalancing across supported chains (e.g., Ethereum, Base, Linea, Optimism, Unichain, Arbitrum).
3. may lead to loss of funds.
4. when maxFee is not enforced as 0 the everclear netting pathway will not be used and protocol will not be able to cut their cost up to 10x

PoC

.

Mitigation

enforce both fields are validated..

```
  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());
    // Enforce netting pathway requirements
    require(params.maxFee == 0, Everclear_InvalidMaxFee());
    require(params.ttl == 0, Everclear_InvalidTtl());

    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());

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

    SafeApprove.safeApprove(params.inputAsset, address(everclearFeeAdapter),
    ↪   params.amount);
    (bytes32 id,) = everclearFeeAdapter.newIntent(
        params.destinations,
        params.receiver,
        params.inputAsset,
        params.outputAsset,
        params.amount,
        params.maxFee, // Now guaranteed to be 0
        params.ttl,    // Now guaranteed to be 0
        params.data,
        params.feeParams
    );
    emit MsgSent(_dstChainId, _market, params.amount, id);
}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/malda-protocol/malda-lending/pull/143

Issue M-11: There is no endpoint for triggering `liquidateExternal` from extension chain to be executed by proof forwarder

Source: https://github.com/sherlock-audit/2025-07-malda-judging/issues/370

Found by

ExtraCaterpillar, PeterSR, cergyk, mussucal

Description

mErcHost has a `liquidateExternal` endpoint for using liquidity coming from other chains to perform a liquidation, but there's no equivalent endpoint in mTokenGateway to trigger the liquidation on host chain.

> Please note that `supplyOnHost` can be used to increase `accAmountIn`, but there's no way to specify the borrower which should be liquidated.

mTokenGateway.sol#L224-L252:

```
    function supplyOnHost(uint256 amount, address receiver, bytes4 lineaSelector)
        external
        payable
        override
        notPaused(OperationType.AmountIn)
        onlyAllowedUser(msg.sender)
        ifNotBlacklisted(msg.sender)
        ifNotBlacklisted(receiver)
    {
        // checks
        require(amount > 0, mTokenGateway_AmountNotValid());
        require(msg.value >= gasFee, mTokenGateway_NotEnoughGasFee());

        IERC20(underlying).safeTransferFrom(msg.sender, address(this), amount);

        // effects
        accAmountIn[receiver] += amount;


        emit mTokenGateway_Supplied(
            msg.sender,
            receiver,
            accAmountIn[receiver],
            accAmountOut[receiver],
            amount,
            uint32(block.chainid),
            LINEA_CHAIN_ID,
            //@audit can specify selector, but not borrower to liquidate on host
            ↪  chain
>>          lineaSelector
        );
    }
```

Impact

`liquidateExternal` cannot be called from extension chains to be executed by proof forwarder. Since as we have seen in some issues such as l1Inclusion issue, it is not possible to make some actions permissionlessly (when extension chain is Eth mainnet), this makes impossible to call `liquidateExternal` from Eth mainnet.

Recommendation

Either add a new endpoint `liquidateOnHost`, or some extra data for `supplyOnHost` to be able to specify a borrower to liquidate and pass `liquidateExternal` as `lineaSelector`.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/malda-protocol/malda-lending/pull/106/files

CergyK

Fix looks good. The liquidate method has been added to mTokenGateway and the selector is now handled in BatchSubmitter

Issue M-12: Rebalancer can drain market funds via excessive bridge fees

Source: https://github.com/sherlock-audit/2025-07-malda-judging/issues/686

Found by

befree3x

Summary

The `REBALANCER_EOA` role, which is considered semi-trusted, can drain funds from any market by specifying an arbitrarily high `maxFee` when initiating a bridge transfer through `EverclearBridge.sol`. This violates the trust assumption that the Rebalancer ”cannot transfer user funds” and can only perform DDoS-style attacks, as it allows for a slow but steady extraction of value from the protocol's liquidity pools.

Root Cause

The Rebalancer::sendMsg function allows the `REBALANCER_EOA` to pass an unchecked `_message` blob to the selected bridge contract. The EverclearBridge.sendMsg function decodes this message to extract bridging parameters, including a `maxFee`. However, the bridge contract fails to validate this `maxFee` against any protocol-defined limit, instead passing it directly to the external everclearFeeAdapter.newIntent call. A malicious `REBALANCER_EOA` can therefore set this fee to an extreme value, causing the protocol to lose most of the bridged amount to fees.

Internal Pre-conditions

- A market (e.g., `mWethHost`) has liquidity.
- The `EverclearBridge` is whitelisted in the Rebalancer.

External Pre-conditions

The `REBALANCER_EOA` private key is compromised or its operator acts maliciously.

Attack Path

A malicious actor controlling the `REBALANCER_EOA` decides to drain funds from the `mWethHost` market.

- The actor calls `Rebalancer.sendMsg`, targeting the `EverclearBridge` and `mWethHost`.

- It provides a valid `_amount` to extract from the market, for example, `10 WETH`.
- It crafts the `_message` parameter. Inside this message, which is decoded by `EverclearBridge`, it sets the amount to be bridged to 10 WETH but sets the `maxFee` parameter to an extremely high value, such as `9.9 WETH`.
- The `Rebalancer` contract extracts `10 WETH` from `mWethHost` and calls `EverclearBridge.sendMsg`.
- `EverclearBridge` decodes the message but performs no validation on the `maxFee`.
- It calls `everclearFeeAdapter.newIntent`, passing along the malicious maxFee of `9.9 WETH`.
- The external `Everclear` protocol executes the bridge transfer. It sends `10 WETH` but is authorized to take up to `9.9 WETH` as a fee, which is lost from the protocol. Only 0.1 WETH (or less) arrives at the destination market.
- The attacker can repeat this process, draining a substantial portion of the market's liquidity over time through exorbitant fees.

Impact

`Rebalancer` can cause permanent loss of protocol funds. This attack directly drains the liquidity provided by users from the market contracts. While the `REBALANCER_EOA` does not receive the funds directly, their action causes value to be extracted from the protocol and paid to third-party bridge relayers.

PoC

No response

Mitigation

The protocol should not blindly trust the `maxFee` parameter provided by the semi-trusted `REBALANCER_EOA`. A centrally-controlled, maximum allowable fee should be enforced.

- The `GUARDIAN_BRIDGE` role (a more trusted admin) should set a maximum fee limit (e.g., as a percentage or basis points) on a per-token, per-chain basis within the `Rebalancer` or bridge contracts.
- The `EverclearBridge.sendMsg` function must validate that the `maxFee` parameter decoded from the message does not exceed the configured maximum fee limit for that specific bridging route.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/malda-protocol/malda-lending/pull/139

CergyK

Fix looks good

Issue M-13: `WrapAndSupply::wrapAndSupplyOnExtensionMarket` preventes users from supplying on host

Source: https://github.com/sherlock-audit/2025-07-malda-judging/issues/741

Found by

0xDemon, 0xKann, 0xlucky, 7, BADROBINX, Cybrid, ExtraCaterpillar, IvanFitro, Richard796, TECHFUND-inc, WillyCode20, Yaneca_b, ZanyBonzy, Ziusz, bube, dany.armstrong90, dimah7, elolpuer, farismaulana, gh0xt, harry, molaratai, mussucal, sakibcy, teoslaf1, weblogicctf

Summary

WrapAndSupply::wrapAndSupplyOnExtensionMarket first calls `deposit` to the `underlying` token contract implementation via the _wrap() function , which takes all the `msg.value` provided in the transaction, and turns it into the minted tokens. As a result, the subsequent call to `supplyOnHost` can potentialy always lead to revert if gasFee is set, since all of the `msg.value` is used AND since all of this happens in single transactoin. There is no amount left for the `gasFee`. This is a problem because it disrupts the implementation of the crucial functionality for setting gas fees. Transactions passing through the `WrapAndSupply` service for extension markets are only successful when gasFee = 0.

Root Cause

The functionality that wraps a native coint into its wrapped version uses all of the msg.value provided by the user without taking into account the possible requirement of additional value for gas fees.

Internal Pre-conditions

1. WrappedNative implementation and mToken market with `underlying` set to it have to exist
2. Onwer needs to set gasFee to the `mTokenGateway` contract.

External Pre-conditions

1. User has to call the wrap and supply utility provided by the protocol.

Attack Path

.

Impact

Breaks core contract functionality

PoC

- Used test file: ”test/unit/mTokenGateway/mTokenGateway_supplyOnHost.sol”
- added optional console logs for clarity

```
import {console} from "forge-std/console.sol";
contract mTokenGateway_supplyOnHost is mToken_Unit_Shared {
...
```

```
function test_WrapAndSupply() external {
        // set gas fee
        vm.startPrank(address(mWethExtension.owner()));
        mWethExtension.setGasFee(1 wei);
        // console.log("gasFee: ", mWethExtension.gasFee());
        vm.stopPrank();

        // wrap and supply
        vm.startPrank(address(this));
        // 10000000000000000000 10e18
        vm.deal(address(this), SMALL);
        // console.log(address(this).balance);
        WrapAndSupply wrapAndSupply = new WrapAndSupply(address(weth));
        vm.label(address(wrapAndSupply), "WrapAndSupply Helper");

        uint256 accAmountInBefore = mWethExtension.accAmountIn(address(this));
        // console.log(accAmountInBefore);

        wrapAndSupply.wrapAndSupplyOnExtensionMarket{value: SMALL}(
            address(mWethExtension), address(this),
             ↪  mTokenGateway_supplyOnHost.test_RevertWhen_AmountIs0.selector
        );
        vm.stopPrank();

    }
```

The test reverts with the ”mTokenGateway_NotEnoughGasFee()” error as expected

```
[FAIL: mTokenGateway_NotEnoughGasFee()] test_WrapAndSupply() (gas: 565393)
```

Mitigation

Possible solution is to fetch the gasFee price before calling the wrap and supply function, so he can provide enough amount to pass. And then within the `_wrap()` calculate the new amount to wrap.

- add view function to the gateway interface

```
function gasFee() external view returns (uint256);
```

- recalculate mint amount

```
uint256 _gasFee = ImTokenGateway(wrappedNative).gasFee();
        uint256 amount = msg.value - _gasFee;
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/malda-protocol/malda-lending/pull/111

CergyK

Changes requested on the fix

Barnadrot

changes have been added

CergyK

Fix looks good

Issue M-14: MixedPriceOracleV4.sol :: getUnderlyingPrice()/getPirce() will not work for some tokens because API3 and EO oracles return prices using different decimals, causing DOS scenario.

Source: https://github.com/sherlock-audit/2025-07-malda-judging/issues/945

Found by

0xEkko, 0xfocusNode, 10ap17, Cybrid, HeckerTrieuTien, IvanFitro, KiroBrejka, PeterSR, SafetyBytes, ZanyBonzy, Ziusz, axelot, dan__vinci, elolpuer, pashap9990

Summary

`getUnderlyingPrice()/getPrice()` are used to obtain the price of the assets in USD. It does this by querying two different oracles, in this case, API3 and EO oracles. The problem is that these oracles return prices with different decimals, causing the function to always end up using the EO oracle's price.

Root Cause

getUnderlyingPrice() calls `_getPriceUSD()`, which in turn calls `_getLatestPrice()`, implemented as follows:

```
function _getLatestPrice(string memory symbol, PriceConfig memory config)
        internal
        view
        returns (uint256, uint256)
    {
        if (config.api3Feed == address(0) || config.eOracleFeed == address(0))
         ↪  revert MixedPriceOracle_MissingFeed();

        //get both prices
@>      (, int256 apiV3Price,, uint256 apiV3UpdatedAt,) =
 ↪  IDefaultAdapter(config.api3Feed).latestRoundData();
@>      (, int256 eOraclePrice,, uint256 eOracleUpdatedAt,) =
 ↪  IDefaultAdapter(config.eOracleFeed).latestRoundData();
        // check if ApiV3 price is up to date
        uint256 _staleness = _getStaleness(symbol);
        bool apiV3Fresh = block.timestamp - apiV3UpdatedAt <= _staleness;

        // check delta
@>      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;
@>        if (!apiV3Fresh || deltaBps > deltaSymbol) {
            require(block.timestamp - eOracleUpdatedAt < _staleness,
             ↪  MixedPriceOracle_eOracleStalePrice());
            decimals = IDefaultAdapter(config.eOracleFeed).decimals();
            uPrice = uint256(eOraclePrice);
         } else {
            require(block.timestamp - apiV3UpdatedAt < _staleness,
             ↪  MixedPriceOracle_ApiV3StalePrice());
            decimals = IDefaultAdapter(config.api3Feed).decimals();
            uPrice = uint256(apiV3Price);
        }

        return (uPrice, decimals);
    }
```

As you can see, it obtains the price from both oracles and calculates the absolute difference using `_absDiff()`:

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

The problem is that the oracles do not return prices with the same decimals. According to the documentation, API3 always returns prices with 18 decimals, while the EO oracle almost always returns prices with 8 decimals.

As a result, `_absDiff()` ends up calculating the difference between an 18-decimal price and an 8-decimal price, producing an incorrect delta. This delta will be disproportionately large—essentially because `18 decimals - 8 decimals   18 decimals` —causing the first `if` condition to be triggered almost every time (because `_delta` is capped by `PRICE_DELTA_EXP = 1e5`). This leads to always selecting the EO oracle's price, effectively making the dual-oracle setup useless.

Always entering the first `if` is problematic because if `apiV3Fresh = true` (meaning the API3 price is not stale) but the EO oracle price **is** stale, the transaction will still revert with `MixedPriceOracle_eOracleStalePrice()`. This happens even when the prices themselves are correct, simply because `deltaBps > deltaSymbol` is true due to the decimal mismatch. As a result, it creates a DOS for obtaining the price, when in reality the price could be retrieved using API3 orcale if the decimals from both oracles were adjusted correctly.

Internal Pre-conditions

None.

External Pre-conditions

API decimals > EO decimals or vice versa. API price is fresh, and EO price is stale.

Attack Path

None.

Impact

ThThe price obtained using `getUnderlyingPrice()` will always come from the EO oracle. If the EO price is stale while the API3 price is fresh, the price cannot be obtained, creating a denial-of-service (DoS) scenario.

PoC

To illustrate the problem, let's use the wBTC / USD pair on the Linea chain, which is in scope for the contest. In the EO oracle, the price is returned with 8 decimals, while in API3 it's returned with 18 decimals.

1. `getUnderlyingPrice()` is called for an mToken whose underlying asset is wBTC.
2. The EO oracle price is `120000000000` (8 decimals), and the API3 oracle price is `120000000000000000000000` (18 decimals). The API3 price is fresh, while the EO oracle price is stale.
3. `_absDiff()` is calculated as `120000000000000000000000 - 120000000000   120000000000000000000000`, producing a very large delta.
4. Because `deltaBps` is much greater than `deltaSymbol`, the first `if` condition is triggered, selecting the EO oracle's price. However, this reverts because the EO price is stale.

If `deltaBps` were calculated correctly, the `else` branch would be taken, and the fresh API3 price would be used. In the current implementation, this creates a denial of service (DoS) scenario where the price cannot be obtained.

Mitigation

To solve the problem, normalize the decimals from both oracles before calculating `_absDiff()`, ensuring the `deltaBps` is computed correctly.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/malda-protocol/malda-lending/pull/134

CergyK

Fix looks good, as already mentioned in #1305

Issue M-15: If Across Bridging fails, all funds intended for bridging will become locked

Source: https://github.com/sherlock-audit/2025-07-malda-judging/issues/1309

Found by

0xapple, 10ap17, ExtraCaterpillar, SafetyBytes, ZeroTrust, cergyk

Summary

If the Across bridging process fails, funds intended for bridging become locked in the Rebalancer contract, making them inaccessible.

Root Cause

When the Across bridge calls the depositV3Now function, the depositor is set as the Rebalancer. According to the Across documentation, if a deposit expires or a fill fails, funds are refunded to the depositor on the origin chain. When the bridging process using Across fails, the refunded assets are returned to the Rebalancer, making them completely locked and inaccessible. From Across docs:

- `In cases where a slow fill can't or does not happen and a relayer does not fill the intent, the intent expires. Like a slow fill, this expiry must be optimistically verified, which takes a few hours. Once this verification is done, the user is then refunded their money on the origin chain.`

As seen bellow, Rebalancer is set as depositor:

```
function _depositV3Now(bytes memory _message, address _token, uint32 _dstChainId,
 ↪  address _market) private {
       DecodedMessage memory msgData = _decodeMessage(_message);
       // approve and send with Across
       SafeApprove.safeApprove(_token, address(acrossSpokePool),
        ↪  msgData.inputAmount);
       IAcrossSpokePoolV3(acrossSpokePool).depositV3Now( // no need for
        ↪  `msg.value`; fee is taken from amount
           msg.sender, //depositor
           address(this), //recipient
           _token,
           address(0), //outputToken is automatically resolved to the same token on
            ↪  destination
           msgData.inputAmount,
           msgData.outputAmount, //outputAmount should be set as the inputAmount -
            ↪  relay fees; use Across API
           uint256(_dstChainId),
           msgData.relayer, //exclusiveRelayer
           msgData.deadline, //fillDeadline
           msgData.exclusivityDeadline, //can use Across API/suggested-fees or 0 to
            ↪  disable
           abi.encode(_market)
       );
   }
```

https://github.com/sherlock-audit/2025-07-malda/blob/798d00b879b8412ca4049ba09dba5ae42464cfe7/malda-lending/src/rebalancer/bridges/AcrossBridge.sol#L168

Internal Pre-conditions

/

External Pre-conditions

/

Attack Path

- The Rebalancer initiates the bridging process using AcrossBridge.
- The depositV3Now function is called with the Rebalancer set as the depositor.
- The bridging process fails.
- Funds are refunded to the Rebalancer(depositor) on the origin chain.
- The funds are now locked, as there is no mechanism for the Rebalancer to use them or return them to the market.

Impact

Funds become completely locked inside the Rebalancer contract and are effectively lost.

PoC

No response

Mitigation

Consider adding helper function in Rebalancer contract, that could rescue returned tokens to the intended market.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/malda-protocol/malda-lending/pull/126

CergyK

Fix looks good

Barnadrot

Added the suggested changes, good to merge now?

CergyK

> Added the suggested changes, good to merge now?

Yes good to merge

Issue M-16: Bridges don't support all of the listed assets

Source: https://github.com/sherlock-audit/2025-07-malda-judging/issues/1477

Found by

10ap17, holtzzx

Summary

/

Root Cause

Across and Everclear, don't support all of the assets that are listed as supported. That will result in inability to rebalance markets for that specific tokens

Internal Pre-conditions

/

External Pre-conditions

/

Attack Path

No specific attack path, since the bridge don't support these assets

Impact

Rebalancing will be impossible for those markets.

PoC

No response

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/malda-protocol/malda-lending/pull/100

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users’ sole risk and is the users’ responsibility.