Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Malda Lending

Veridise Inc.
April 17, 2025

► **Prepared For:**

Malda
https://docs.malda.xyz

► **Prepared By:**

Mark Anthony
Petr Susil
Benjamin Sepanski

► **Contact Us:**

contact@veridise.com

► **Version History:**

Apr. 17, 2025	V5 - Incorporate issue fixes
Apr. 02, 2025	V4 - Incorporate review of L1 inclusion routes
Mar. 17, 2025	V3 - Metadata updates and incorporating developer response
Feb. 28, 2025	V2 - Incorporated issue fixes, updated recommendations
Feb. 21, 2025	V1
Feb. 20, 2025	Initial Draft

Contents

1 Executive Summary

From Jan. 20, 2025 to Feb. 18, 2025, Malda engaged Veridise to conduct a security assessment of their Malda lending protocol. The security assessment covered the Malda smart contracts, as well as a Rust program intended to be run in the Risc Zero zkVM. This report focuses on only the smart contracts. A companion report discusses the findings from the coincident zk-coprocessor review. Veridise conducted the first assessment over 12 person-weeks, with 3 security analysts reviewing the project over 4 weeks on commit `fde7102d`. Following this review, Malda engaged Veridise from Mar. 24 to Mar. 27 to conduct a security assessment of the Malda L1 Inclusion feature. The second assessment occurred over 8 person-days, with 2 security analysts reviewing the project over 4 days on commit `e1bba48a`. The review strategy involved a tool-assisted analysis of the program source code performed by Veridise security analysts as well as thorough code review.

Project Summary. The Malda smart contracts form a cross-chain over-collateralized lending protocol, with many of the core components based on Compound V2. As in other over-collateralized lending protocols, users may deposit ERC-20 tokens into a `mToken` *market* contract, in return for `mERC20`s. These `mERC20`s are yield-bearing tokens that simultaneously act as collateral for the supplier. Different market contracts are connected together by a shared `Operator` contract, which tracks the debt and collateral value of all users across all connected markets. Individuals may borrow ERC-20 tokens from a market contract as long as their total borrowed value stays below the value of their collateral, with each collateral token weighted by a protocol-configured collateral weight. If this is not the case, the users' funds may be liquidated, a process by which any user may purchase a protocol-determined proportion of the insolvent user's collateral at a premium (called the *liquidation incentive*).

Malda differs from other similar over-collateralized protocols in its cross-chain nature. All debt and `mToken` balances are tracked on a single *host* chain (in this case, Linea). However, users may supply funds for minting, repaying debt, or performing liquidations on several connected *extension* chains (in this case, Optimism and Base). Similarly, users may withdraw or borrow against `mToken` balances on the host chain in order to receive funds on any of the extension chains.

The host chain acts as the synchronous ledger tracking all debt and collateral balances. Risc Zero ZK proofs are used to execute a view call within the state of an L2 block, and to verify that the appropriate Sequencer has signed the last entry in a sufficiently long chain of blocks built atop of the L2 block. The zkVM application was reviewed by Veridise as part of this assessment, but is discussed in a separate, companion report. For the purposes of this report, it will be assumed the off-chain ZK co-processing is implemented correctly.

To prevent replay attacks, the host chain tracks the monotonically increasing total amount each user has deposited on and sent to an extension chain since deployment. Similarly, the extension chains track the (monotonically increasing) total amount each user has deposited and received (on this particular extension chain) since deployment. The proven view call attests to these values. Consequently, the host chain can compare the total amount a user has deposited on an

extension chain with the total amount of funds that same user has already been credited for sending along that connection. If there is a difference between the two values, the user may mint and the record on the host will be updated, preventing repeated mints.

A deposit on an extension chain may be used to perform any action (mint, repay, or liquidate) on the host. In contrast, extension chain actions begun on the host will be specified as *either* a borrow or withdrawal, since the host chain will update a user's debt immediately upon initiating a cross-chain borrow.

Finally, in order to ensure that funds are available to fulfill user requests, the Malda protocol provided *rebalancer* contracts. The rebalancer may bridge funds from one chain to another to ensure sufficient funds are available for borrows and withdrawals.

The first security assessment covered the smart contracts implementing the markets, operator, rebalancers, and a reward distribution mechanism designed to incentivize borrowing and lending on Malda.

The second security assessment covered the addition of the Malda L1 Inclusion routine as part of the "slow lane" implementation in the zk-coprocessor. This feature allows Malda to avoid a severe trade-off between usability and security.

The permissionless slow-lane ensures users self-custody their funds. Any user posting proofs through the slow lane must prove the inclusion of the particular L2 block on L1 (see the code assessment below for a discussion on how this differs slightly for the Linea chain). This extra validation enforces the condition that the L2 block is "cross-safe," i.e. included in the chain derived from L1 data. While this takes longer, a higher requirement for provable security is maintained for this permissionless operation. Faster cross-chain operations can be approved by the Malda-operated "fast lane".

In the fast lane, the Malda-operated "Malda Sequencer" (and possibly a small number of known, whitelisted entities) is able to post proofs based on the unsafe head of a chain. Malda is expected to derive each L2 chain and decide when (or if) to post proofs to the fast lane based on a variety of factors. These factors include the value of the cross-chain operation, the number of L2 confirmations, the number of L1 confirmations on the posted L2 data, as well as several key risk signals (discussed at length with the Veridise analysts) being actively monitored for by the Malda Sequencer. Users should note that this does not replace any of the checks required by the proof. Malda only obtains the ability to censor individuals from the fast lane, but does not gain authority to transfer or lock their funds except by pausing or upgrading the contracts.

The second assessment also covered the introduction of an "outflow volume limit" to the protocol which is designed to control the outflow of funds within a given time period. This change was introduced as a response to Veridise recommendation to limit the damages on a particular chain in case of a block re-org or any other unforeseen event.

Code Assessment - Initial Review. The Malda developers provided the source code of the Malda contracts for the code review. The source code borrows from Compound V2. All of the cross-chain implementations and integration of those features into the core logic is made up of original code written by the Malda developers. It contains some documentation in the form of documentation comments on functions and storage variables. To facilitate the Veridise security analysts' understanding of the code, the Malda developers provided an independent security

review of the design, which described the intended behavior as well as several potential issues and mitigations.

The source code contained a test suite, which the Veridise security analysts noted covers the success and failure conditions of core functions and the workflow between host and extension chain. The test suite makes thorough checks for expected reverts related to access control, protocol limits, and configuration errors. However, these tests primarily check for success or failure, and are limited in the full set of scenarios covered. For example, checking a full sequence of depositing, borrowing, and attempted liquidation could have identified V-MLD-VUL-001, and additional testing of the core workflows would likely have prevented V-MLD-VUL-002.

As mentioned in the project summary, Malda allows cross-chain actions to occur when a sequencer has signed a block. In this process, the sequencer is assumed to be a trusted actor and to not post re-orged blocks. In the current design of the supported L2s, sequencers are centralized entities and do intentionally plan to not post re-orged blocks. However, in the case of key exploitation, bugs in sequencers, or vulnerabilities (such as past sequencer manipulation on Arbitrum), this trust assumption will be invalid.

For example, the L2 network Optimism classifies the unsafe block head—the latest block signed by the sequencer—as local safe, "enough to reproduce the local L2 chain content, but not to reason about cross-L2 interactions." For normal operations, Veridise expects the sequencer trust assumption (using the local head) to work well. However, adverse network conditions would require the stronger guarantees of the cross-safe head–the latest block which is submitted on the L1.

The main issue at hand is the trade-off between user experience (fast cross-chain actions), security (allowing anyone to use the protocol without threatening benign user funds), and censorship (no centralized entity can prevent cross-chain actions). This was addressed by the Malda team through the introduction of the Malda L1 Inclusion feature, discussed further in the code assessment of the second review. This allows for a strong user experience, prevents censorship (ignoring the case of contract upgrades/pauses, see V-MLD-VUL-013 for a full discussion of centralization risks), and avoids trusting the entire sequencer implementation and operation. It should be noted that it is the responsibility of the Malda Sequencer to ensure the protocol correctly excludes reorged blocks submitted to the fast lane in the case of an L2 reorg.

In the recommendations section below, Veridise include several mitigations to improve security against attackers without sacrificing user experience in the common case. These changes were incorporated by Malda and reviewed as part of the Malda L1 Inclusion feature.

Code Assessment - Second Review. For the second review, the Malda engaged in several detailed discussions with the Veridise team regarding the suggested changes. They then provided a writeup of their planned implementations, along with the code, and tests to facilitate testing out the new changes.

The new slow-lane/fast-lane division protects Malda users against block reorgs triggered by sequencers. For OP-Stack chains, cross-chain actions performed through the slow lane must wait for L1 inclusion. Since the L1 state used by the ZK circuits is attested to by the Optimism sequencer, attacking any operation from a non-Optimism OP-Stack chain would require compromising the Optimism sequencer *and* causing a reorg in the target chain. For Linea, cross-chain actions performed through the slow lane must wait until the Linea block

number recorded on the L1 by the Linea sequencer is at least the claimed Linea block's number. So long as the Linea sequencer does not act maliciously, this leaves Malda the time between L2 block generation and the L2 block being posted on the L1 to identify a reorg (at least six hours). Consequently, so long as Malda responds within six hours to any Linea reorgs, the slow lane is safe against reorgs which happen due to sequencer bugs. It is still vulnerable to sequencer key compromise if the Optimism sequencer or Linea sequencer is compromised.

Cross-chain actions performed through the fast lane will be validated independently by the Malda Sequencer, which is expected to re-derive the L2 chains itself and require a number of block confirmations based on the operation-size. These actions may be more likely to be affected by a reorg or sequencer compromise, but are subject to active vigilance by the Malda team, which should impose stricter limitations for large operations to limit any potential damage.

Summary of Issues Detected. Issues from the initial security assessment are listed in Section 4. The security assessment uncovered 39 issues, 3 of which are assessed to be of high or critical severity by the Veridise analysts. Specifically, V-MLD-VUL-001 observed that liquidation could only be performed on solvent accounts, V-MLD-VUL-002 showed that the exchange rate could be manipulated, and V-MLD-VUL-003 identified that the chain ID was used instead of the domain ID for the Connext bridge (see also V-MLD-VUL-005).

The Veridise analysts also identified 4 medium-severity issues, including use of a bridge which was about to be sunset (V-MLD-VUL-005), storage collisions in an upgradeable contract (V-MLD-VUL-004), possible privilege escalation in case of a compromised rebalancer (V-MLD-VUL-006), and a lack of slippage protection (V-MLD-VUL-007). Additionally, the Veridise analyst identified 7 low-severity issues, 19 warnings, and 6 informational findings. These include limited support for tokens with hooks (V-MLD-VUL-008, V-MLD-VUL-027), operations which may not be pausable (V-MLD-VUL-009, V-MLD-VUL-030), front-running risks which can cause a temporary denial-of-service (V-MLD-VUL-022), missing oracle checks (V-MLD-VUL-024), and unchecked constants (V-MLD-VUL-017, V-MLD-VUL-018, V-MLD-VUL-021). Finally, the Veridise team documented further trust assumptions and centralization risks in V-MLD-VUL-013.

Of the 39 acknowledged issues in the initial review, Malda has fixed 29 issues, including all high- and critical-severity issues. They further provided partial fixes to two warning issues related to best practices and duplicate code.

Malda does not plan to fix the other 8 acknowledged issues at this time. In response to V-MLD-VUL-005, Malda indicated that the Connext bridge would not be deployed. All remaining issues are of low severity or lower, including acknowledgement of deployment and centralization risks (V-MLD-VUL-013 and V-MLD-VUL-009), as well as 4 warning issues and 1 info issue.

Issues from the second security assessment are listed in Section 5. The second assessment uncovered 10 issues, 1 of which is assessed to be of a medium severity by the Veridise analysts. Specifically V-MLD2-VUL-001 details how L1 inclusion is not verified on the extension chains, which can leave the protocol vulnerable to double spends.

The Veridise analysts also identified 3 low-severity issues, including V-MLD2-VUL-003 which shows that while calculating the USD value for the outflow checks the token and price decimals are handled incorrectly, V-MLD2-VUL-002 which highlights the contribution of deprecated markets to the total USD value and V-MLD2-VUL-004 which discusses various considerations

regarding the outflow limits, including how large actors can use up the limits to DoS pending user repays.

Finally, the Veridise analysts identified 5 warnings, and 1 informational finding. Notable issues include a single step transfer of ownership V-MLD2-VUL-006, missing initializations V-MLD2-VUL-009, and missing address zero-checks V-MLD2-VUL-005.

Malda has fixed 9 of the 10 issues from the second review. This includes all issues except a warning recommending a two-step ownership transfer (V-MLD2-VUL-006).

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the Malda lending protocol.

Protocol monitoring and interaction. Continuously monitor the protocol to ensure that liquidations are performed, that rebalancing is performed appropriately (see V-MLD-VUL-039), and the protocol is paused in case of an emergency scenario such as key compromise or re-orgs. Regularly accrue interest, since (as in Compound), transferring funds does not update interest on borrows.

Interest policy under pause. Currently, when the protocol is paused interest continues to accrue. The Malda team should clearly document how borrowers can protect themselves in case of a paused protocol (readers may see V-MLD-VUL-013 for a full description of the centralization risks in the protocol).

Setting and documenting limits. The protocol should take advantage of the outflow limits implemented in the second review. Outflow limits should be set and clearly documented to limit any potential damage from cross-chain interactions. See also also V-MLD2-VUL-009.

Key management. Implement the recommendations in the centralization risk issue V-MLD-VUL-013. Compromised roles may upgrade contracts, change the ZK image ID, or otherwise steal protocol funds. Great care should be taken to ensure keys are managed properly, and significant protocol changes are clearly communicated to end users ahead of their implementation.

Reducing sequencer trust. In the initial version of the project, the sequencers were fully trusted in order to improve user UX. Malda took several large steps to reduce this trust, described above as the slow and fast lanes. The Veridise analysts recommend taking the following mitigations for the fast lane to help ensure that benign users are able to benefit from the UX improvement without extending the same trust assumption to potential attackers.

1. Continuously monitor the L2s for uptime. Chainlink has oracles which can be used for this purpose. This will allow Malda to immediately pause the protocol in case of sequencer downtime.
2. Limit the fund transfer size by reorg depth protection as a precaution. This will limit the potential damage in case of a catastrophic event, ensuring not all funds are lost. This transfer limit should be both per-transfer and per-unit of time (e.g., transferred funds per day) to prevent someone from submitting a large number of small, fraudulent transfers.
3. A reorg depth larger than zero should be used for reorg protection to provide a buffer in case of emergencies, should they arise. For example, there could be a bug in the sequencer which leads to an accidental reorg. This buffer will allow the protocol to be paused in case a reorg is detected. An L1 reorg depth of one plus a few more blocks for detection and pausing would be ideal.

The reorg depth chosen for particular L2s should take into consideration the different block times and historical reorg data, if available. This depth should be large enough that monitors are likely to see the reorg and still have enough time to pause the protocol before any attacker abusing the reorg can take advantage. Combining this with recommendation (1) will allow trusted actors to operate with more agility while providing a stronger security guarantee for trustless actors.

Update to Recommendations. In response to the above recommendations, Malda discussed plans for reducing sequencer trust with the Veridise analysts. Based on these discussions, Malda divided cross-chain actions into a fast lane and a slow lane, described already above. This logic was implemented, and the on-chain/in-circuit changes implementing and enforcing the slow lane checks were reviewed by Veridise.

Finally, Malda expressed their confidence in sequencer operations, as the sequencers have undergone many audits and have been running without major exploits for several years. Reorg blocks only recently became documented, but a 10+ L2 block confirmation is much larger than historical reorgs*. Despite this confidence, they have implemented the above plan to minimize all potential attack surfaces.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

* `https://optimistic.etherscan.io/blocks_forked`

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Malda	`fde7102d`	Solidity	Linea, Optimism, Base
Malda L1 Inclusion	`e1bba48a`	Solidity	Linea, Optimism, Base

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Jan. 20–Feb. 18, 2025	Manual & Tools	3	12 person-weeks
Mar. 24–Mar. 27, 2025	Manual & Tools	2	8 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	1	1	1
High-Severity Issues	2	2	2
Medium-Severity Issues	4	4	3
Low-Severity Issues	7	7	5
Warning-Severity Issues	19	19	13
Informational-Severity Issues	6	6	5
TOTAL	39	39	29

Table 2.4: Vulnerability Summary: Malda L1 Inclusion.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	0	0	0
Medium-Severity Issues	1	1	1
Low-Severity Issues	3	3	3
Warning-Severity Issues	5	5	4
Informational-Severity Issues	1	1	1
TOTAL	10	10	9

Table 2.5: Category Breakdown: Malda.

Name	Number
Data Validation	10
Maintainability	8
Logic Error	5
Denial of Service	4
Usability Issue	3
Frontrunning	3
Access Control	3
Reentrancy	2
Missing/Incorrect Events	1

Table 2.6: Category Breakdown: Malda L1 Inclusion.

Name	Number
Data Validation	4
Logic Error	2
Maintainability	2
Denial of Service	1
Authorization	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of Malda's smart contracts. During the assessment, the security analysts aimed to answer questions such as:

- Are there common smart contract vulnerabilities like access control issues, reentrancy, Denial-of-Service, front-running risks, or first depositor attacks?
- Are common zkVM vulnerabilities like missing validation or replay attacks possible?
- Are slippage protections in place to avoid market manipulation during asynchronous interactions?
- Can the cross-chain interactions lead to any unexpected intermediate states allowing unfair liquidation or extraction of funds?
- Are all cross-chain factors of a message's sender such as chain ID, timestamp/block number, smart contract address on the other chain, and user address on the other chain validated? Are those same factors validated for the intended recipient?
- Is all data packing implemented correctly?
- Does the addition of the slow lane cause any issues with the existing functionality? And is the L1 inclusion verified correctly and wherever necessary?
- Is the outflow limit implemented correctly? Does it introduce avenues for an attacker to cause denial of service?
- What roles or centralized entities can affect core protocol functionality?
- How are delays or price changes handled during cross-chain interactions?
- Are other chain state accesses securely validated?
- Can cross-chain transactions fail after being initiated?
- Are unusual ERC20 features supported?
- Is the protocol properly initialized?
- Can the L1 inclusion check be bypassed?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the security assessment involved a combination of human experts and automated program analysis & testing tools. In particular, the security assessment was conducted with the aid of the following technique:

- *Static analysis.* To identify potential common vulnerabilities, security analysts leveraged Veridise's custom smart contract analysis tool Vanguard, as well as the open-source tool semgrep. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.

Scope. The scope of the initial security assessment at commit `fde7102d` is limited to the `src/` folder of the source code provided by the Malda developers, which contains the smart contract

implementation of the Malda lending protocol. Within `src/`, the following contracts were excluded:

- `src/Counter.sol`
- `src/libraries/BytesLib.sol`
- `src/libraries/CREATE3.sol`
- `src/oracles/ChainlinkOracle.sol`

Additionally, the developers indicated that `src/rebalancer/messages/LZMessageOnlyBridge.sol` would not be used by the protocol.

During the security assessment, the Veridise security analysts referred to the excluded files but assumed that they have been implemented correctly.

The scope of the second security assessment at commit `e1bba48a` is limited to changes in the following files of the source code provided by the developers:

- `src/Operator/Operator.sol`
- `src/Operator/OperatorStorage.sol`
- `src/interfaces/IOperator.sol`
- `src/interfaces/ImTokenGateway.sol`
- `src/interfaces/ImErc20Host.sol`
- `src/mToken/host/mErc20Host.sol`
- `src/mToken/BatchSubmitter.sol`
- `src/mToken/extension/mTokenGateway.sol`
- `src/mToken/mTokenStorage.sol`
- `src/verifier/ZkVerifier.sol`
- `src/libraries/mTokenProofDecoderLib.sol`

The changes in the fix review include the implementation of an outflow volume limit on protocol actions which limits the outflow of funds from the host, addition of the L1 inclusion check as part of the slow lane implementation and some general refactoring.

Methodology. Veridise security analysts reviewed the reports of previous audits for Malda, inspected the provided tests, and read the Malda documentation. They then began a review of the code assisted by static analyzers.

During the security assessment, the Veridise security analysts regularly met with the Malda developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

The likelihood of a vulnerability is evaluated according to the Table 3.2.

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

The impact of a vulnerability is evaluated according to the Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

This section presents the vulnerabilities found during the initial security assessment on commit `fde7102d`. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-MLD-VUL-001	Incorrect condition for liquidation	Critical	Fixed
V-MLD-VUL-002	Exchange rate manipulable by external mints	High	Fixed
V-MLD-VUL-003	Chain ID used for Domain ID	High	Fixed
V-MLD-VUL-004	Unit/Operator slot collision	Medium	Fixed
V-MLD-VUL-005	The Connext bridge should not be used	Medium	Acknowledged
V-MLD-VUL-006	Rebalancer EOA can steal funds	Medium	Fixed
V-MLD-VUL-007	No slippage protection on mint	Medium	Fixed
V-MLD-VUL-008	Support for reward tokens with hooks	Low	Fixed
V-MLD-VUL-009	Liquidation allowed after pausing all...	Low	Fixed
V-MLD-VUL-010	Non-atomic deployment/configuration	Low	Acknowledged
V-MLD-VUL-011	Can withdraw to unsupported chains	Low	Fixed
V-MLD-VUL-012	Transfer limits are uninitialized and unitless	Low	Fixed
V-MLD-VUL-013	Centralization Risk	Low	Acknowledged
V-MLD-VUL-014	getUnderlying uses wrong symbol	Low	Fixed
V-MLD-VUL-015	Initializable best practices	Warning	Fixed
V-MLD-VUL-016	Solidity best practices	Warning	Partially Fixed
V-MLD-VUL-017	highLimit assigned incorrect constant value	Warning	Fixed
V-MLD-VUL-018	Multiplier can be higher than the...	Warning	Fixed
V-MLD-VUL-019	Unchecked max/min close factor	Warning	Fixed
V-MLD-VUL-020	ERC20 Front-running approval risk	Warning	Acknowledged
V-MLD-VUL-021	Missing address zero-checks	Warning	Fixed
V-MLD-VUL-022	Front-running may lead to DoS	Warning	Acknowledged
V-MLD-VUL-023	Unsafe cast for blockchain IDs	Warning	Acknowledged
V-MLD-VUL-024	Missing Chainlink Oracle Checks	Warning	Fixed
V-MLD-VUL-025	Sweeping check may be insufficient	Warning	Fixed
V-MLD-VUL-026	Unused/duplicate program constructs	Warning	Partially Fixed
V-MLD-VUL-027	Tokens with sender hooks are not supported	Warning	Acknowledged
V-MLD-VUL-028	Malicious caller can cause bridge funds to...	Warning	Fixed
V-MLD-VUL-029	Locked dust in rebalancer	Warning	Fixed
V-MLD-VUL-030	Unpausable operation	Warning	Fixed
V-MLD-VUL-031	Responsibilities of a bridge guardian role	Warning	Fixed
V-MLD-VUL-032	Missing events	Warning	Fixed
V-MLD-VUL-033	Unchecked return value on...	Warning	Fixed
V-MLD-VUL-034	Missing check of supported market	Info	Fixed
V-MLD-VUL-035	Interest is per-second, not per-block	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-MLD-VUL-001: Incorrect condition for liquidation

Severity	Critical	**Commit**	fde7102
Type	Logic Error	**Status**	Fixed
File(s)	`Operator.sol`		
Location(s)	beforeMTokenLiquidate		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/39`, e7ca8e7		

The function `beforeMTokenLiquidate()` is called by `__liquidate` to ensure liquidation conditions are met. The account state is retrieved using function `(collateral, shortfall) = _getHypotheticalAccountLiquidity`. The `require` statement reverts with error if the condition is not met.

```
1 (, uint256 shortfall) = _getHypotheticalAccountLiquidity(borrower, address(0), 0, 0);
2 require(shortfall == 0, Operator_InsufficientLiquidity());
```

Snippet 4.1: Snippet from `beforeMTokenLiquidate()`

Impact Insolvent users cannot be liquidated, and solvent users may be liquidated.

Recommendation Ensure that shortfall is non-zero before liquidation.

Developer Response The developers now check that the shortfall is positive, rather than zero.

4.1.2 V-MLD-VUL-002: Exchange rate manipulable by external mints

Severity	High	**Commit**	fde7102
Type	Logic Error	**Status**	Fixed
File(s)	`src/mToken/mErc20.sol`, `src/mToken/mToken.sol`, `src/mToken/host/mErc20Host.sol`		
Location(s)	mErc20._getCashPrior(), mToken.__mint(), mErc20Host._mintExternal()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/33`, c8beef5		

In the Malda lending protocol, users may deposit underlying collateral on any chain, then mint `mTokens` on the "host" chain (Linea, in the planned deployment) based on their deposit. Chains other than Linea (e.g. Arbitrum, Optimism, or Base) are called "extension chains". When the deposit occurs on an extension chain, users create a ZK-proof attesting to their deposit, and call `mErc20Host.mintExternal()` on the host chain to mint `mTokens`. Currently, due to how the exchange rate is computed, an attacker can exploit the external minting process.

The exchange rate is computed based on the protocol borrows, reserves, `mToken` supply, and cash stored *on the host chain*. This means that, when minting from an extension chain, the `mToken` total supply changes, but the cash stored on the host chain does not. When this happens, the exchange rate decreases, i.e. the `mTokens` become less valuable.

```
/*
 *  exchangeRate = (totalCash + totalBorrows - totalReserves) / totalSupply
 */
uint256 totalCash = _getCashPrior();
uint256 cashPlusBorrowsMinusReserves = totalCash + totalBorrows - totalReserves;
uint256 exchangeRate = (cashPlusBorrowsMinusReserves * expScale) / _totalSupply;
```

Snippet 4.2: Snippet from `_exchangeRateStored()`, computing the number of underlying tokens per collateral `mToken`.

```
  function _getCashPrior() internal view virtual override returns (uint256) {
      return IERC20(underlying).balanceOf(address(this));
  }
```

Snippet 4.3: Definition of `mErc20._getCashPrior()`, which returns the amount of underlying held by the protocol on the host chain.

As a concrete example, suppose that the protocol begins with `100 WETH` on Linea backing `100 mWETH`, and no other markets/deposits. At this initial stage, the exchange rate is one-to-one. For the purposes of this example, assume Alice has provided `10 WETH` in return for `10 mWETH`.

We will ignore reserves, and assume the collateral and close factors are 90%.

1. Alice takes out a loan of `9 WETH` against her `10 mWETH`.
2. Bob (the attacker) deposits `25 WETH` on Optimism, and then calls `mintExternal()` on Linea. Since the current exchange rate is 1-1, this mints `25 mWETH` to Bob.
3. Bob now attempts to liquidate Alice. The protocol has `91 WETH` on Linea, `9 WETH` of borrows, and a total supply of `125 mWETH`. So, the exchange rate is `(91 + 9 WETH) / (125 mWETH) = 0.8 WETH / mWETH`. This means Alice's collateral of

`10 mWETH` is worth `10 mWETH * 0.8 WETH/mWETH = 8 WETH`. With a collateral factor of 90%, Alice now has a shortfall of

```
9 WETH - 90% * 8 WETH = 1.8 WETH
```

Since Alice has a shortfall, Bob may liquidate up to 90% of her borrow balance, buying her `9 mWETH` for at most `9 mWETH * 0.8 ETH/mWETh  = 7.2 WETH`.

At this point, Bob has deposited `25 ETH`, gained `25 mWETH`, gained `9 mWETH`, and spent at most an additional `7.2 WETH`. So, Bob has gained at least `34 mWETH`, and spent at most `32.2 WETH`, even though the initial exchange rate was only 1-1.

For Bob to profit, he must now wait for someone else to withdraw from Linea to Optimism, or for the protocol to rebalance funds, restoring the exchange rate to be closer to 1-1.

Impact `mintExternal()` and `withdrawExternal()` change the exchange rate, allowing users to forcibly liquidate others, or exploit third-party protocols trading `mTokens`.

This can lead to direct theft from users (as shown above), or theft from third-party protocols relying on a stable exchange rate.

Recommendation Change `_getCashPrior()` to include assets deposited on extension chains, and to exclude liabilities on extension chain (produced by a call of `withdrawExternal()`).

Developer Response The developers now explicitly track the amount of underlying tokens on-hand using a storage variable which is updated on each mint, redeem, borrow, repay, and reserve change.

4.1.3 V-MLD-VUL-003: Chain ID used for Domain ID

Severity	High	**Commit**	fde7102
Type	Logic Error	**Status**	Fixed
File(s)	`src/rebalancer/bridges/ConnextBridge.sol`, `src/rebalancer/bridges/EverclearBridge.sol`		
Location(s)	sendMsg()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/55/`, 6ea4071		

In Connext and Everclear, "domain IDs" are used for chains, rather than chain IDs. These are typically distinct values. See this Connext link and this Everclear link for a list of chain IDs and domain IDs. For example, in Connext, Optimism has a domain ID of 1869640809 but a chain ID of 10. Currently, Everclear domain IDs match chain IDs (and the local domain is set to `block.chainid` in the hub and spoke initializers), but this is not guaranteed to be the case by governance or documented.

The `Rebalancer` passes a destination chain ID to each bridge, which is then used for the destination parameter.

```
1  destinations[0] = _dstChainId;
2  SafeApprove.safeApprove(_token, address(everclearSpoke), amount);
3  (bytes32 _intentId,) = everclearSpoke.newIntent(destinations, market, _token,
     outputAsset, amount, 0, 0, data);
```

Snippet 4.4: Snippet from `EverclearBridge.sendMsg()`

Impact Funds may be locked, sent to the wrong chain. See also V-MLD-VUL-005.

Recommendation Use a bridge guardian-configured chain ID-to-domain map.

Developer Response The developers added a (`GUARDIAN_BRIDGE`-configured) mapping from destination chain ID to domain ID.

4.1.4 V-MLD-VUL-004: Unit/Operator slot collision

Severity	Medium	**Commit**	fde7102
Type	Denial of Service	**Status**	Fixed
File(s)	`src/Operator/Unit.sol`		
Location(s)	Unit		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/25`, 54d68a6		

The `Unit` contract acts similarly to a transparent proxy for the `Operator` contract. As shown in the below snippet, most function calls to a `Unit` are delegated to the code stored in the `operatorImplementation` address.

```
fallback() external payable {
    // delegate all other functions to current implementation
    (bool success,) = operatorImplementation.delegatecall(msg.data);
        // ...
```

Snippet 4.5: Snippet from `Unit.fallback()`

The `operatorImplementation` is intended to be an instance of the `Operator` contract, as can be seen in the below snippet from `DeployOperator`. The deployed setup will have a `Unit` contract (the proxy) acting as the operator (the implementation) by delegating all calls from contracts or users to the implementation contract, but executing within the context (i.e. with the storage and account balance of) the proxy.

```
// Set up the implementation in the proxy
Unit(payable(proxy)).setPendingImplementation(implementation);
Operator(implementation).become(proxy);

Operator(proxy).setPriceOracle(oracle);

// [VERIDSE] elided...

return proxy;
```

Snippet 4.6: Snippet from `DeployOperator.run()`

The `Unit` has several storage variables, stored in slots zero through three.

```
> forge inspect Unit storage-layout

|--------------------------------+---------+------+--------+-------+-----------------------------|
| Name                           | Type    | Slot | Offset | Bytes | Contract                    |
+=============================================================================================+
| admin                          | address | 0    | 0      | 20    | src/Operator/Unit.sol:Unit  |
|--------------------------------+---------+------+--------+-------+-----------------------------|
| pendingAdmin                   | address | 1    | 0      | 20    | src/Operator/Unit.sol:Unit  |
```

```
|-------------------------------+---------+------+--------+-------+-----------------------|
| operatorImplementation        | address | 2    | 0      | 20    | src/Operator/Unit
    .sol:Unit |
|-------------------------------+---------+------+--------+-------+-----------------------|
| pendingOperatorImplementation | address | 3    | 0      | 20    | src/Operator/Unit
    .sol:Unit |
|-------------------------------+---------+------+--------+-------+-----------------------|
```

Snippet 4.7: Storage layout of `Unit`.

Unfortunately, this storage layout conflicts with the `Operator`.

```
forge inspect Operator storage-layout

|-------------------------------+----------------+------+--------+-------+-----------------------
| Name                          | Type           | Slot | Offset | Bytes | Contract
                              |
+=============================================================================================================
| admin                         | address        | 0    | 0      | 20    | src/
    Operator/Operator.sol:Operator |
|-------------------------------+----------------+------+--------+-------+-----------------------
| pendingAdmin                  | address        | 1    | 0      | 20    | src/
    Operator/Operator.sol:Operator |
|-------------------------------+----------------+------+--------+-------+-----------------------
| rolesOperator                 | contract IRoles | 2    | 0      | 20    | src/
    Operator/Operator.sol:Operator |
|-------------------------------+----------------+------+--------+-------+-----------------------
| oracleOperator                | address        | 3    | 0      | 20    | src/
    Operator/Operator.sol:Operator |
|-------------------------------+----------------+------+--------+-------+-----------------------
| closeFactorMantissa           | uint256        | 4    | 0      | 32    | src/
    Operator/Operator.sol:Operator |
| ...                           | ...            | ...  | ...    | ...   | ...
```

Snippet 4.8: Storage layout of `Operator`.

This means that the `Unit` and `Operator` contracts assign different logical meanings to the same storage slot (namely, slots two and three, and possibly slots zero and one if the contracts are intended to have different admins).

This can be seen with the below proof of concept (added to `Base_Unit_Test`, and run with `forge test --match-test StorageCollision --match-contract mErc20_mint`), which fails on the final assertion.

```
function test_StorageCollision() public {
    address originalRoles = address(operator.rolesOperator());
```

```
3
4       Unit unit = new Unit(address(this));
5       // overwrites oracleOperator
6       unit.setPendingImplementation(address(operator));
7       // overwrites rolesOperator
8       operator.become(address(unit));
9
10      Operator proxy = Operator(address(unit));
11
12      assertEq(originalRoles, address(operator.rolesOperator()));
13      assertEq(originalRoles, address(proxy.rolesOperator()));
14  }
```

Impact In practice, this would most likely lead to an immediate denial of service once deployed, as calling `Operator.setRolesOperator()` or `Operator.setPriceOracle` would overwrite the `pendingOperatorImplemenation` or `operatorImplementation` on the `Unit` contract.

To deal with this, developers would need to upgrade the contracts and re-deploy. However, as the `Unit` contract is not upgradeable, this would require a rewrite of the `Operator` contract, which is much more involved in terms of storage usage.

Recommendation Avoid reimplementing upgradeability. Instead, rely on well-tested libraries such as OpenZeppelin's `OwnableUpgradeable`. Additionally, consider adding `_disableInitializers()` to the `constructor` of the `Operator` after `Unit` is removed.

See also related issue V-MLD-VUL-015.

Developer Response The developers implemented the recommendation, using `OwnableUpgradeable` in place of the `Unit` contract.

Updated Veridise Response This resolves the issue. It is best practice to also remove the (now unused) `Unit` contract.

4.1.5 V-MLD-VUL-005: The Connext bridge should not be used

Severity	Medium	**Commit**	fde7102
Type	Usability Issue	**Status**	Acknowledged
File(s)	`src/rebalancer/bridges/ConnextBridge.sol`		
Location(s)	See description		
Confirmed Fix At	N/A		

The Connext bridge is being sunset as mentioned here and should not be used to bridge funds anymore.

> We encourage all users to remove funds from pools and bridge nextAssets back to their canonical domain.

Impact As the Connext bridge is being sunset, it may no longer be supported. If used, it can lead to a loss of funds for the protocol, or other irregularities.

Recommendation Do not use the Connext bridge to bridge funds.

Developer Response The developers have acknowledged the issue.

4.1.6 V-MLD-VUL-006: Rebalancer EOA can steal funds

Severity	Medium	Commit	fde7102
Type	Frontrunning	Status	Fixed
File(s)	src/rebalancer/bridge/*.sol		
Location(s)	sendMsg()		
Confirmed Fix At	https://github.com/malda-protocol/malda-lending/pull/57, 354864e		

As seen in the below snippet, the `Rebalancer` contract is allowed to remove funds directly from the gateway or host mToken contracts in order to rebalance underlying tokens across the protocol.

```
function extractForRebalancing(uint256 amount) external {
    if (!rolesOperator.isAllowedFor(msg.sender, rolesOperator.REBALANCER())) revert
    mErc20Host_NotRebalancer();
    IERC20(underlying).safeTransfer(msg.sender, amount);
}
```

Snippet 4.9: Snippet from `mErc20Host`

Using a contract to perform this should limit the actions that can be taken. Indeed, the `Rebalancer` contract is managed by two roles:

1. A `GUARDIAN_BRIDGE` which is trusted to whitelist bridges, limit transfer sizes, and configure bridge-specific parameters.
2. A `REBALANCER_EOA` which is trusted to send funds back and forth between markets on different chains using the underlying bridges.

It is expected that the `REBALANCER_EOA` can threaten funds by making inefficient transfers. However, they can additionally make simple mistakes which directly hurt the protocol users, or even just transfer funds to themselves. This can be seen in the following ways:

1. The `REBALANCER_EOA` specifies the following through an (unchecked) `_msg` argument passed to `Rebalancer.sendMsg()`:
    a) (All bridges): The destination chain ID is not validated, allowing a malicious rebalancer EOA to deploy a protocol they own on a different chain, then transfer funds to that protocol.
    b) (All bridges): The amount to be transferred is not checked to match the rebalance amount.
    c) (`LZBridge`): The destination address is not validated to be a trusted address.
    d) (`AcrossBridge`) The `outputAmount` is set arbitrarily, with no limit on slippage protection for the users.
    e) (`ConnextBridge`, `AcrossBridge`) The `delegate` is set to an arbitrary address, which may revert the transfer leading to locked funds.

Impact Since the `REBALANCER_EOA` is likely to be a relatively hot key, compromise of this key may be more likely than other roles.

If compromised, an attacker may escalate their privileges from rebalancing to direct theft of protocol funds, or burning a large number of funds through setting poor slippage parameters.

Recommendation Instead of taking an arbitrary `_msg`, invoke `*Bridge.sendMsg` with a validated destination chain ID, a `GUARDIAN_BRIDGE`-configured market and output token for that destination chain ID, and a slippage maximum bps to ensure that the rebalancer is not setting slippage to be too low.

See related issue V-MLD-VUL-013.

Developer Response The `GUARDIAN_BRIDGE` now whitelists destination chains, relayers for the `AcrossBridge`, and delegates for the `Connext` and `Everclear` bridges. The amount and market passed to the `Rebalancer` are passed to the bridges and validated to match the actual bridged destination/amounts.

A maximum slippage of 10% is set in the `Across` bridge, and the `LZMessageOnlyBridge` is deleted.

4.1.7 V-MLD-VUL-007: No slippage protection on mint

Severity	Medium	**Commit**	fde7102
Type	Frontrunning	**Status**	Fixed
File(s)	`src/mToken/mErc20.sol`		
Location(s)	mint()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/56/`, 5e0dd2b		

Great care should be taken when deploying the project, as other compound forks have suffered from deployment-time attacks (see: the hundred finance hack).

As mentioned above, first-depositor attacks have happened in the wild (see: the hundred finance hack). Malda mitigates first-depositor attacks by inserting 1000 dead shares. Hard-coding in 1000 requires an attacker to have `1000` times the capital ready to risk, so this provides some protection. However, a slightly higher number (e.g. around `10000`) and slippage protection in `mint()` could reduce this risk.

Impact Frontrun initial deposits may be stolen.

Recommendation Add slippage protection to `mint()`. See also V-MLD-VUL-010 for other deployment-related concerns.

Developer Response The developers added a `minAmountOut` parameter for mints which reverts if the received tokens is not as large as expected.

4.1.8 V-MLD-VUL-008: Support for reward tokens with hooks

Severity	Low	**Commit**	fde7102
Type	Reentrancy	**Status**	Fixed
File(s)	malda-lending/src/rewards/RewardDistributor.sol		
Location(s)	_grantReward		
Confirmed Fix At	https://github.com/malda-protocol/malda-lending/pull/29, 89c95da		

As seen in the below snippet, the transfer of funds in the function `_grantReward()` is executed before the state `rewardAccrued` is updated. If the reward token parameter to function `_claim()` is a token with a transfer hook, such as an instance of ERC777, the user may call `claim()` again and receive an additional reward before their `rewardAccrued` is reset to zero. By repeating this process, they may drain the reward contract's balance in the reentrant token.

```
function _claim(address rewardToken, address[] memory holders) internal {
    for (uint256 j = 0; j < holders.length;) {
        IRewardDistributorData.RewardAccountState storage accountState =
    rewardAccountState[rewardToken][holders[j]];
        accountState.rewardAccrued = _grantReward(rewardToken, holders[j],
    accountState.rewardAccrued);
        unchecked {
            ++j;
        }
    }
}
```

Snippet 4.10: Definition of `RewardDistributor._claim()`

```
function claim(address[] memory holders) public override {
    for (uint256 i = 0; i < rewardTokens.length;) {
        _claim(rewardTokens[i], holders);
        unchecked {
            ++i;
        }
    }
}
```

Snippet 4.11: Definition of `RewardDistributor.claim()`

Impact If a reward token with a transfer hook is whitelisted by the contract owner, all the rewards for that token may be withdrawn by a single user. For instance, tokens implementing ERC777 should not be supported as reward tokens.

Recommendation Use a reentrancy guard, and update the rewardAccrued before transferring funds.

Developer Response The developers added a `nonReentrant` guard to `Operator.claimMalda()`.

Updated Veridise Response Users may call `RewardDistributor.claim()` directly. The `nonReentrant` guard must instead be added to this function.

Updated Developer Response The developers added a `nonReentrant` guard to `RewardDistributor.claim()`.

4.1.9 V-MLD-VUL-009: Liquidation allowed after pausing all market operations

Severity	Low	**Commit**	fde7102
Type	Logic Error	**Status**	Fixed
File(s)	`src/pauser/Pauser.sol`		
Location(s)	_pauseAllMarketOperations()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/41`, 454392c		

The malda protocol allows administrators to pause individual operations. In cases of emergency when all operations should be paused, the `emergencyPauseMarket()` method invokes `_pauseAllMarketOperations`. This function is supposed to pause all market operations, but leaves `OperationType.Liquidate` unchanged.

```
function _pauseAllMarketOperations(address _market) private {
      _pauseMarketOperation(_market, OperationType.AmountIn);
      _pauseMarketOperation(_market, OperationType.AmountOut);
      _pauseMarketOperation(_market, OperationType.AmountInHere);
      _pauseMarketOperation(_market, OperationType.AmountOutHere);
      _pauseMarketOperation(_market, OperationType.Mint);
      _pauseMarketOperation(_market, OperationType.Borrow);
      _pauseMarketOperation(_market, OperationType.Transfer);
      _pauseMarketOperation(_market, OperationType.Seize);
      _pauseMarketOperation(_market, OperationType.Repay);
      _pauseMarketOperation(_market, OperationType.Redeem);
      emit MarketPaused(_market);
}
```

Snippet 4.12: Definition of `_pauseAllMarketOperations()`

Impact If an exploit occurs leading to invalid protocol states, attackers may continue to liquidate positions. Since all other operations are paused, users will be left defenseless.

Note, however, that since `Seizes` are paused, attackers will only be able to liquidate positions if the collateral token is from a different, unpaused market.

Recommendation Add `OperationType.Liquidate` to the set of operations paused by `_pauseAllMarketOperations()`.

Developer Response Operation `_pauseAllMarketOperations` now includes `OperationType.Liquidate` as well.

4.1.10 V-MLD-VUL-010: Non-atomic deployment/configuration

Severity	Low	**Commit**	fde7102
Type	Data Validation	**Status**	Acknowledged
File(s)			
Location(s)			
Confirmed Fix At	N/A		

The Malda protocol is extensive and involves a large amount of configuration and setup as part of its deployment. The following considerations should be made when deploying and configuring the project:

1. Deployment:
   a) `mTokenHost` and the `mTokenGateway(s)` for a particular market must have the same address. Any error on this front will cause the protocol to break. Similarly, the `AcrossBridge` contract address needs to be the same across chains. This contract is used to interact with the Across Bridge and while sending funds, the recipient of the bridged funds is set to itself.
   b) `mErc20Immutable.constructor()` leaves `rolesOperator` and `reserveMantissa` initialized to zero. These must both be initialized after deployment.
   c) `Operator` initialization leaves `liquidationIncentiveMantissa` initialized to zero, which must be initialized before `mTokens` are deployed.
2. Configuration:
   a) `ZkVerifier` should point to the risc zero verifier *router*, see `https://dev.risczero.com/api/blockchain-integration/contracts/verifier#verifier-implementations`.
   b) The "host" chain must never support itself.
   c) `setRolesOperator` can cause `roles` on `mTokenConfiguration` to diverge from `operator.rolesOperator()`, and must be called each time `setOperator()` is called.
   d) Mapping by symbol in `MixedPriceOracleV3` requires careful consideration to avoid collisions.

Impact Deployment across multiple chains, if misconfigured, can break the protocol. Some contracts may be deployed in unusable situations, leading to errors on launch. Configuration errors while the protocol is operating may lead to inadvertently DoSing the protocol, potentially causing liquidations or insolvency.

Recommendation Atomically deploy and configure contracts where possible. Elsewhere, deploy the protocol as paused. Rigorously test the deployment procedure and all configuration changes.

Maintain a clear registry of contracts used for oracles, bridges, and Malda contracts deployed on various chains. See also V-MLD-VUL-013.

Developer Response The developers have acknowledged the issue.

4.1.11 V-MLD-VUL-011: Can withdraw to unsupported chains

Severity	Low	**Commit**	fde7102
Type	Data Validation	**Status**	Fixed
File(s)	`src/mToken/host/mErc20Host.sol`		
Location(s)	withdrawOnExtension(), borrowOnExtension()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/49`, 5378465		

The functions `withdrawOnExtension()` and `borrowOnExtension()` update state on the host chain so that someone may withdraw funds on an extension chain. However, neither function validates that the specified chain ID is allowed (i.e. `allowedChains[dstChainId]` may be false).

```
function withdrawOnExtension(uint256 amount, uint32 dstChainId) external override {
    require(amount > 0, mErc20Host_AmountNotValid());

    // actions
    accAmountOutPerChain[dstChainId][msg.sender] += amount;
    _redeemUnderlying(msg.sender, amount, false);

    emit mErc20Host_WithdrawOnExtensionChain(msg.sender, dstChainId, amount);
}
```

Snippet 4.13: Definition of `withdrawOnExtension()`.

Impact Users who use the wrong chain ID will have their funds locked.

Recommendation Check that the `dstChainId` is allowed for both the `withdrawOnExtension` and `borrowOnExtension` methods.

Developer Response The developers implemented the recommendation.

4.1.12 V-MLD-VUL-012: Transfer limits are uninitialized and unitless

Severity	Low	**Commit**	fde7102
Type	Maintainability	**Status**	Fixed
File(s)	`src/rebalancer/bridges/BaseBridge.sol`		
Location(s)	setMinTransferSize(), sendMsg()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/54`, 7e35382		

The `BaseBridge` allows setting the minimum and maximum transfer size for a particular transaction. These are checked before any transfer is sent via a bridge.

By default, the minimum transfer is `50_000 * 1e18`, and the maximum transfer amount is not initialized. This will prevent transfer of any funds on the bridge. In addition, this will prevent changing the minimum transfer size as it must be set to something less than the maximum.

Once initialized, these transfer limits are applied to *all tokens*. As seen below, a minimum transfer of `50_000 * 1e18` will likely prevent all transfers of an 8-decimal token with non-negligible value.

```
function sendMsg(uint32 _dstChainId, address _token, bytes memory _message, bytes
    memory)
    external
    payable
    onlyRebalancer
{
    // decode message & checks
    (address market, address outputAsset, uint256 amount, bytes memory data) =
        abi.decode(_message, (address, address, uint256, bytes));
    require(amount >= minTransfer && amount <= maxTransfer, BaseBridge_AmountNotValid
    ());
```

Snippet 4.14: Snippet from `setMinTransferSize()`

Finally, the maximum transfer may be bypassed by performing multiple calls to `sendMsg` in the same transaction.

Impact The transfer limits are not set based on the value of a token, but instead on its numerical representation. Additionally, these limits are uninitialized, and the maximums can be bypassed by performing repeated transfers.

Recommendation Set `maxTransferSize` in the constructor. Make the transfer limits based on token value. Limit the maximum transferred over time, rather than the maximum transferred per transaction.

Developer Response The developers restructured the `Rebalancer` to have a maximum per-chain per-token transfer size.

Updated Veridise Response We have the following concerns with this fix:

1. `setMaxTransferSize` is not access controlled.
2. The minimum transfer size was removed, which was in place to reduce potential fee costs / spurious transfers.
3. In `sendMsg()`:
   a) `transferInfo.size` is required to be less than `currentTransferSize[][].size` instead of `maxTransferSize[][].size`.
   b) `transferInfo.size` is checked to be within bounds, instead of checking `transferInfo.size + _amount`, allowing one transfer per window to violate the maximum.

Updated Developer Response The developers resolved the above items.

Updated Veridise Response The Veridise analysts have the following remaining concerns:

1. The transfer limit maximums holds per-calendar day rather than holding over all periods of length `transferTimeWindow`. This should be documented.
2. Once the minimum is met once within a transfer window, arbitrarily small transfers may be performed. This may lead to excessive fees. The minimum should be enforced per-transfer, rather than per window.

Updated Developer Response The developers made the minimum per-transaction.

4.1.13 V-MLD-VUL-013: Centralization Risk

Severity	Low	**Commit**	fde7102
Type	Access Control	**Status**	Acknowledged
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	N/A		

Similar to many projects, Malda's lending protocol declares several administrator roles that are given special permissions. We break down permissions in detail below. At a high-level, administrators can

1. Upgrade the contracts.
2. Pause any operation of the pools.
3. Rebalance funds across chains, or even directly transfer user funds out of the protocol (see related issue V-MLD-VUL-006).
4. Set price oracles, configure bridges, and support additional markets/chains.
5. Configure protocol settings related to interest or liquidation factors.

In more detail, we break down the following roles in the Malda protocol:

Protocol-level Roles intended for EOAs.

1. `Operator.admin` - This is a single address who may
   a) Change the `Roles` contract used by the `Operator` (see below), and change the `admin`.
   b) Add price oracles and markets (`mTokens`).
   c) Configure the protocol settings, including collateral factors, close factors, liquidation incentives, and market borrow/supply caps, and reward distribution.
   d) Pause *and unpause* markets.
2. `Unit.admin` - a single address which can upgrade the `Unit` to point to a different `Operator`.
3. `Roles.owner` - An address which may grant any of the roles stored on `Roles`, or grant ownership to a new address.
4. `Roles.REBALANCER_EOA` - An account which can rebalance funds across chains by using the `Rebalancer` contract and an appropriate bridge. See related issue V-MLD-VUL-006.
5. `Roles.GUARDIAN_BRIDGE` - An account which can set oracles, whitelist (or un-whitelist) bridges, configure bridge-specific parameters, and set limits on bridge transfer sizes.
6. `Roles.GUARDIAN_BORROW_CAP`, `Roles.GUARDIAN_SUPPLY_CAP` - Accounts which can set limits on cross-chain borrows/mints.
7. `Roles.GUARDIAN_RESERVE` - An account which can reduce reserves in a market, taking the funds directly for their own purpose.
8. `Roles.PROOF_FORWARDER` - A role which can submit proofs on behalf of any sender to receive the results of cross-chain actions.
9. `Roles.PAUSE_MANAGER` - An address which can pause any operation in the protocol.
10. `Roles.CHAINS_MANAGER` - An address which can update the allowed chains on `mErc20Host`.

Token-level Roles intended for EOAs.

1. `mErc20Host.admin` - An address which can update allowed chains, **set the verifier contract/image ID**, submit proofs, **upgrade the contract**, and perform any operations permitted to the `mErc20.admin`.

2. `mErc20.admin` - An address which may delegate votes earned from the Malda token, sweep accidentally transferred (non-underlying, see V-MLD-VUL-025) tokens from the protocol, and perform any operations permitted to the `mToken.admin`.
3. `mToken.admin` - An address which may reduce reserves in the market (just as `Roles.GUARDIAN_RESERVE`) and perform any operation permitted to the `mTokenConfiguration.admin`.
4. `mTokenConfiguration.admin` - An address which may **set the operator**, which allows them control over all other roles for the token. They can also set parameters like the interest rate model, maximum borrow, and reserve factor.
5. `mTokenGateway.owner`: This account is authorized to **upgrade the contract**, pause the contract, **set the verifier contract/image ID**, and act as the `Roles.PROOF_FORWARDER`.

Utility roles.

1. `JumpRateModelV4.owner` - This address can configure the jump-rate model.
2. `BatchSubmitter.owner` - This address can **set the verifier/image ID**.
3. `RewardDistributor.owner` - This address can set the `operator`, whitelist tokens, change reward configurations, and directly grant rewards.

Roles intended for contracts.

1. `Roles.REBALANCER` ****- This is the role which should be held by the actual `Rebalancer` contract.
2. `Roles.GUARDIAN_PAUSE` ****- This is the role which only the `Pauser` contract should have, allowing the contract to pause (but not unpause) any market (see `Operator.setPaused`, `mTokenGateway.setPaused`).
3. `Roles.PROOF_BATCH_FORWARDER` - This is the role for the `BatchSubmitter`. It can perform cross-chain actions without a proof.

Upgradeable contracts.

The following contracts appear to be upgradeable:

1. `Unit`, with `Operator` as its implementation
2. `mErc20Upgradeable`
    a) Transitively: `mErc20Host`
3. `mTokenGateway`
4. `RewardDistributor`
5. `ZkVerifier`
    a) Transitively: `BatchSubmitter`, `mTokenGateway`, `mErc20Host`

Additional Notes.

Pausing redeems has the side effect of pausing transfers.

Additionally, callers can choose to perform mints instead of repays, which may leave a vulnerable user closer to insolvency.

Impact If a private key were stolen, a hacker would have access to sensitive functionality that could compromise the project. For example, a malicious user could upgrade the contracts, change the verifier ID, malicious rebalance, or change the roles in order to steal funds.

Recommendation Privileged operations should be operated by a multi-sig contract or decentralized governance system. Non-emergency privileged operations should be guarded by a timelock to ensure there is enough time for incident response. The risks in this issue may be partially mitigated by validating that the protocol is deployed with the appropriate roles granted to the timelock and multi-sig contracts.

Full validation of operational security practices is beyond the scope of this review. Users of the protocol should ensure they are confident that the operators of privileged keys are following best practices such as:

1. Never storing a protocol key in plaintext, on a regularly used phone, laptop, or device, or relying on a custom solution for key management.
2. Using separate keys for each separate function.
3. Storing multi-sig keys in a diverse set of key management software/hardware services and geographic locations.
4. Enabling 2FA for key management accounts. SMS should **not** be used for 2FA, nor should any account which uses SMS for 2FA. Authentication apps or hardware are preferred.
5. Validating that no party has control over multiple multi-sig keys.
6. Performing regularly scheduled key rotations for high-frequency operations.
7. Securely storing physical, non-digital backups for critical keys.
8. Actively monitoring for unexpected invocation of critical operations and/or deployed attack contracts.
9. Regularly drilling responses to situations requiring emergency response such as pausing/unpausing.

Developer Response The developers acknowledged the issue and will lock the admin functions behind multisigs, with thresholds depending on the required security level. The address and threshold for these multisigs will be published in the documentation for mainnet release. The full publishing of this is out of scope for the audit, but the developers will publish it in their documentation.

4.1.14 V-MLD-VUL-014: getUnderlying uses wrong symbol

Severity	Low	**Commit**	fde7102
Type	Logic Error	**Status**	Fixed
File(s)	`MixedPriceOracleV3.sol`, `ChainlinkOracle.sol`		
Location(s)	getUnderlyingPrice		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/61`, c4431f4		

The following snippets show `getUnderlyingPrice()` for MixedPriceOracleV3 and ChainlinkOracle.

The price is looked up based on `ImTokenMinimal(mToken).symbol()`. Subsequently, the price of `mToken` is multiplied by a constant `10 ** (18-config.underlyingDecimals)` to compute the return value of `getUnderlyingPrice()`.

```
    function getUnderlyingPrice(
        address mToken
    ) external view override returns (uint256) {
        string memory symbol = ImTokenMinimal(mToken).symbol();
        IDefaultAdapter.PriceConfig memory config = configs[symbol];
        uint256 priceUsd = _getPriceUSD(symbol);
        return priceUsd * 10 ** (18 - config.underlyingDecimals);
    }
```

Snippet 4.15: Snippet from `MixedPriceOracleV3.getUnderlyingPrice()`

```
function getUnderlyingPrice(address mToken) external view override returns (uint256)
    {
        string memory symbol = ImTokenMinimal(mToken).symbol();
        uint256 feedDecimals = priceFeeds[symbol].decimals();

        (uint256 price,) = _getLatestPrice(symbol);
        return (price * (10 ** (36 - feedDecimals))) / baseUnits[symbol];
    }
```

Snippet 4.16: Snippet from `MixedPriceOracleV3.getUnderlyingPrice()`

Currently, `getUnderlyingPrice(mToken)` is used in the following functions and leads to inaccurate conditions in liquidations and seizing tokens.

function liquidateCalculateSeizeTokens(...) { ... uint256 priceBorrowedMantissa = IOracleOperator(oracleOperator).getUnderlyingPrice(mTokenBorrowed); ... numerator = mul_(Exp({mantissa: liquidationIncentiveMantissa}), Exp({mantissa: priceBorrowedMantissa})); ... ratio = div_(numerator, denominator); return mul_ScalarTruncate(ratio, actualRepayAmount); }

```
function _getHypotheticalAccountLiquidity(...) {
   ...
   vars.oraclePriceMantissa = IOracleOperator(oracleOperator).getUnderlyingPrice(
    _asset);
   vars.oraclePrice = Exp({mantissa: vars.oraclePriceMantissa});
   vars.tokensToDenom = mul_(mul_(vars.collateralFactor, vars.exchangeRate), vars.
    oraclePrice);
   ...
```

```
7    vars.sumBorrowPlusEffects =
8        mul_ScalarTruncateAddUInt(vars.tokensToDenom, redeemTokens, vars.
     sumBorrowPlusEffects);
```

Impact The return value for `getUnderlyingPrice(mToken)` does not reflect the real underlying value. This may influence third party integrations.

Recommendation Use `ImTokenMinimal(mToken).underlying().symbol()` for the price lookup in `getUnderlyingPrice()`.

Developer Response The developers implemented the recommendation.

4.1.15 V-MLD-VUL-015: Initializable best practices

Severity	Warning	**Commit**	fde7102
Type	Data Validation	**Status**	Fixed
File(s)	`src/mtoken/host/mErc20Host.sol`, `src/rewards/RewardDistributor.sol`		
Location(s)	mErc20Host.constructor(), RewardDistributor.constructor()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/31`, `https://github.com/malda-protocol/malda-lending/pull/29`, d92b2a6, 89c95da		

The following contracts are `Initializable`, but do not follow OpenZeppelin upgradeability best practices.

1. `mErc20Host`
2. `RewardDistributor`

Impact This may lead to potential scams or errors by allowing a malicious party to initialize an implementation contract or failing to initialize values.

Recommendation Upgradeable contracts should invoke `_disableInitializers()` in the constructor.

Developer Response The developers added `_disableInitializers()` into the constructor for `mErc20Upgradeable`.

Updated Veridise Response This partially resolves the issue. `_disableInitializers()` should also be added to the `RewardDistributor` constructor.

Updated Developer Response The developers added `_disableInitializers()` into the constructor for `RewardDistributor` as well.

4.1.16 V-MLD-VUL-016: Solidity best practices

Severity	Warning	**Commit**	fde7102
Type	Maintainability	**Status**	Partially Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/32`, 7fb9ed0		

Consider implementing the following Solidity best practices:

1. **Two-phase ownership transfers.** The following contracts are `Ownable`, but transfer ownership in a single step. Using `Ownable2Step` will ensure ownership is not transferred to an account which is unable to accept ownership:
   a) `Roles`
2. **Check result of token transfers.** The following locations perform transfers on `MTokens` but do not check the `transfer` returned `true`. While current implementations never fail, future implementations or extensions may be able to fail a transfer without reverting.
   a) `RewardDistributor.grantReward()`
3. **Return values**. Several `external` functions ignore the values returned by internal functions, instead of passing them along to the end user. This may make development of third-party applications more difficult. This occurs in the following locations:
   a) `mToken._repay()` does not return the value returned by `__repay()`.
   b) `mToken._repayBehalf()` does not return the value returned by `__repay()`.
   c) `RewardDistributor.grantReward()` does not return the value returned by `_grantReward()`.
4. **Missing Length Checks.** Some functions take as input two arrays which are expected to be the same length, but do not verify that the arrays are indeed the same length.
   a) `ChainlinkOracle.constructor()`
   b) `MixedPriceOracleV3.constructor()`
5. **Missing TODOs.** Some functions have remaining TODOs which should be implemented.
   a) `LZBridge.sendMsg()`: The event should add the result's `guid` to the event.

Impact Not following best practices may lead to projects with reduced "by default" security/usability, allowing simple errors to magnify into large mistakes.

Recommendation Follow the above Solidity best practices.

Developer Response The developers have shared a fix for the `_repay*()` functions not returning the repay amounts and have acknowledged the rest of the issue.

4.1.17 V-MLD-VUL-017: highLimit assigned incorrect constant value

Severity	Warning	**Commit**	fde7102
Type	Maintainability	**Status**	Fixed
File(s)	src/mToken/Operator/Operator.sol		
Location(s)	setCollateralFactor()		
Confirmed Fix At	https://github.com/malda-protocol/malda-lending/pull/27/, d072ec6		

The function `setCollateralFactor()` allows an admin to set the collateral factor for a particular market. It takes as input the `newCollateralFactorMantissa`, which is validated to be less than the maximum collateral factor mantissa. See snippet below for context.

The value `highLimit` which is supposed to represent the maximum collateral factor mantissa is assigned the value of constant `CLOSE_FACTOR_MAX_MANTISSA`. It should instead have been assigned the value of constant `COLLATERAL_FACTOR_MAX_MANTISSA`. It does not cause an immediate issue because both constants possess the same value, but that may not always be the case.

```
1 function setCollateralFactor(address mToken, uint256 newCollateralFactorMantissa)
      external onlyAdmin {
2     // Verify market is listed
3     IOperatorData.Market storage market = markets[address(mToken)];
4     require(market.isListed, Operator_MarketNotListed());
5
6     Exp memory newCollateralFactorExp = Exp({mantissa: newCollateralFactorMantissa});
7
8     // Check collateral factor <= 0.9
9     Exp memory highLimit = Exp({mantissa: CLOSE_FACTOR_MAX_MANTISSA});
```

Snippet 4.17: Snippet from `setCollateralFactor()`

Impact The variable `highLimit` is assigned an incorrect constant. Although the value is the same, the configuration and constant values can change over time. This could lead to incorrect validations being performed on the new collateral factors.

Recommendation Assign the value `COLLATERAL_FACTOR_MAX_MANTISSA` to `highLimit`.

Developer Response The developers implemented the recommendation.

4.1.18 V-MLD-VUL-018: Multiplier can be higher than the jumpMultiplier

Severity	Warning	**Commit**	fde7102
Type	Data Validation	**Status**	Fixed
File(s)	`src/interest/JumpRateModelV4.sol`		
Location(s)	_updateJumpRateModel()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/26/`, f8e00c8		

The function `_updateJumpRateModel` is called internally to update the parameters of the jump rate model. The `multiplierPerYear` is multiplied with the utilisation rate to get the additional interest rate on top of the base rate. Once the utilisation rate crosses the `kink` value, an additional jump multiplier is applied. See the snippet below for context.

Inside `_updateJumpRateModel`, the `multiplierPerBlock` is scaled up based on the `kink`. The lower the kink, the higher the increase in the `multiplierPerYear`. This can be problematic for lower kink values, where the `multiplierPerBlock` may end up being greater than the `jumpMultiplierPerBlock`.

```
function _updateJumpRateModel(
  uint256 baseRatePerYear,
  uint256 multiplierPerYear,
  uint256 jumpMultiplierPerYear,
  uint256 kink_
) private {
  baseRatePerBlock = baseRatePerYear / blocksPerYear;
  multiplierPerBlock = multiplierPerYear * 1e18 / (blocksPerYear * kink_);
  jumpMultiplierPerBlock = jumpMultiplierPerYear / blocksPerYear;
  kink = kink_;

  emit NewInterestParams(baseRatePerBlock, multiplierPerBlock, jumpMultiplierPerBlock
    , kink);
}
```

Snippet 4.18: Snippet from `_updateJumpRateModel()`

For example, with the current settings in the deployment scripts, the `multiplierPerYear` is `2e17` and the `jumpMultiplierPerYear` is `5e17`. Therefore, for any `kink` value less than `2e17`, the effective `multiplier` will be greater than the `jumpMultiplier`.

Impact For lower values of `kink`, the `multiplierPerBlock` may end up being greater than the `jumpMultiplierPerBlock`.

Recommendation Within `_updateJumpRateModel()`, when updating the jump rate model parameters, verify that the `multiplierPerBlock` is less than the `jumpMultiplierPerBlock`.

Also consider adding some checks to ensure the kink is between a suitable range.

Developer Response `_updateJumpRateModel()` now explicitly checks that `multiplierPerBlock` is less than `jumpMultiplierPerBlock`.

4.1.19 V-MLD-VUL-019: Unchecked max/min close factor

Severity	Warning	**Commit**	fde7102
Type	Data Validation	**Status**	Fixed
File(s)	`src/Operator/Operator.sol`		
Location(s)	setCloseFactor()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/38`, b3e232f		

The close factor determines what proportion of a borrower's collateral may be liquidated. For example, a close factor of 0.9 means that up to 90% of an insolvent borrower's collateral may be liquidated.

The protocol sets two bounds on the close factor to ensure borrowers' collateral is protected, and that liquidators have a sufficient incentive.

```
// closeFactorMantissa must be strictly greater than this value
uint256 internal constant CLOSE_FACTOR_MIN_MANTISSA = 0.05e18; // 0.05

// closeFactorMantissa must not exceed this value
uint256 internal constant CLOSE_FACTOR_MAX_MANTISSA = 0.9e18; // 0.9
```

Snippet 4.19: Snippet from `OperatorStorage`

However, as shown below, the `setClose()` function allows an administrator to set the `closeFactorMantissa` to any value.

```
  function setCloseFactor(uint256 newCloseFactorMantissa) external onlyAdmin {
      emit NewCloseFactor(closeFactorMantissa, newCloseFactorMantissa);
      closeFactorMantissa = newCloseFactorMantissa;
  }
```

Snippet 4.20: Definition of `setCloseFactor()`

Impact An administrator may accidentally set the close factor to zero, preventing liquidations, or putting a borrower's entire collateral at risk.

Recommendation Check the minimum and maximum when setting the close factor.

Additionally, the close factor is initialized to zero, which may lead to deploying a system in which liquidations cannot occur.

Developer Response The developers updated `setCloseFactor()` to explicitly check that the `newCloseFactorMantissa` respects the minimum and maximum bounds.

4.1.20 V-MLD-VUL-020: ERC20 Front-running approval risk

Severity	Warning	**Commit**	fde7102
Type	Frontrunning	**Status**	Acknowledged
File(s)	`src/mToken/mToken.sol`		
Location(s)	mToken.approve()		
Confirmed Fix At	N/A		

Following the ERC20 standard, `mToken` has an `approve` method. This is vulnerable to front-running when changing an approval. For example, if Bob has approved Alice to spend 100 tokens on his behalf, and Alice sees a new approval from Bob decreasing the allowance to 50 tokens in the mempool, Alice can front-run the transaction to spend 100 tokens. At this point, Alice can then spend 50 more of Bob's tokens, spending 150 instead of the intended 50.

```
function approve(address spender, uint256 amount) external override returns (bool) {
    address src = msg.sender;
    transferAllowances[src][spender] = amount;
    emit Approval(src, spender, amount);
    return true;
}
```

Snippet 4.21: Snippet from `mToken()`

Impact Users of the `approve()` method may be front-run, causing the sum of their allowances to be spent rather than the maximum.

Recommendation Add a warning to the function about the risk. Consider adding a function to increase or decrease allowances atomically.

Developer Response Acknowledged. We approve zero and then approve the value.

Updated Veridise Response Understood. Please note that approving zero and then the value atomically does not remove the front-running risk, as the effect of the transaction is equivalent.

4.1.21 V-MLD-VUL-021: Missing address zero-checks

Severity	Warning	**Commit**	fde7102
Type	Data Validation	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	https://github.com/malda-protocol/malda-lending/pull/43, https://github.com/malda-protocol/malda-lending/pull/57/, 89ca0f7, 354864e		

Description The following functions take addresses as arguments, but do not validate that the addresses are non-zero:

1. `src/Operator/Operator.sol`
   a) `setPriceOracle()`: This function does not validate the `newOracle` is not the zero-address.
2. `src/rebalancer/bridges/messages/LZMessageOnlyBridge.sol`:
   a) `constructor()`: `roles` is not checked to be non-zero.
   b) `setLZEndpoint()`: `_endpoint` is not checked to be non-zero.

Impact If zero is passed as the address, the `mTokens` relying on the `Operator` for solvency computations will stop functioning.

Developer Response The developers now explicitly check that the `newOracle` is not the zero-address when calling `setPriceOracle()`.

The `LZMessageOnlyBridge` is deleted in the second PR.

4.1.22 V-MLD-VUL-022: Front-running may lead to DoS

Severity	Warning	**Commit**	fde7102
Type	Denial of Service	**Status**	Acknowledged
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	N/A		

When performing cross-chain actions, there is a much larger opportunity for front-running. For example, in a cross-chain mint, a user submits a transaction to a mempool for the extension chain depositing funds. Next, they create a zero-knowledge proof. Finally, the proof is submitted on the host chain to receive the minted tokens.

If a supply cap is set (limiting the amount of `mTokens` which may be minted from a certain chain), an attacker may mint up to a supply cap after someone else has performed an external mint, DoSing the deposit.

```
function afterMTokenMint(address mToken) external view override {
        uint256 supplyCap = supplyCaps[mToken];
        // Supply cap of 0 corresponds to unlimited borrowing
        if (supplyCap != 0) {
            uint256 totalSupply = ImToken(mToken).totalSupply();
            Exp memory exchangeRate = Exp({mantissa: ImToken(mToken).
    exchangeRateStored()});
            uint256 totalAmount = mul_ScalarTruncate(exchangeRate, totalSupply);
            require(totalAmount <= supplyCap, Operator_MarketSupplyReached());
        }
```

Snippet 4.22: Definition of `Operator.afterMTokenMint()`.

Additionally, cross-chain repayments may be blocked by a malicious use of `repayBehalf()`. For example, an attacker may see a repayment of 10 WETH on a loan of exactly 10 WETH which is close to liquidation. The attacker may `repayBehalf()` a small amount (e.g. 1 wei), so that calling `repayExternal()` underflows when subtracting 10 WETH from the debt.

Impact Note that mints may also be used as repays, which prevents users from using this strategy to force liquidation. However, a dedicated attacker could still use this to lock funds on a single chain temporarily in order to prevent a depositor from taking advantage of certain opportunities.

Up to 50% of a cross-chain repayment can be prevented via front-running, thus forcing someone into liquidation if they are not prepared to retry the message.

Recommendation Prepare off-chain infrastructure to retry repayments by breaking it up into smaller amounts in case of front-running.

Developer Response The developers have acknowledged the issue.

4.1.23 V-MLD-VUL-023: Unsafe cast for blockchain IDs

Severity	Warning	**Commit**	fde7102
Type	Data Validation	**Status**	Acknowledged
File(s)	`src/mToken/host/mErc20Host.sol`		
Location(s)	getProofData(), _mintExternal()		
Confirmed Fix At	N/A		

Public networks use a chain ID as a unique, global identifier for the chain. While most block chain IDs are small numbers, some block chain IDs are large. For example, Palm has a chain ID of 11297108109, which is larger than 2^{33}.

In `mErc20Host`, it is assumed that the `block.chainid` is at most $2^{32} - 1$.

```
require(_dstChainId == uint32(block.chainid), mErc20Host_DstChainNotValid());
// ...
_res[i] = mTokenProofDecoderLib.encodeJournal(
          _user,
          address(this),
          accAmountInPerChain[_dst][_user],
          accAmountOutPerChain[_dst][_user],
          uint32(block.chainid),
          _dst
      );
```

Snippet 4.23: Snippets from `_mintExternal()` and `getProofData()` in which the code assumes a `block.chainid` is at most 32 bits.

If this code is ever deployed on a chain with a chain ID larger than `type(uint32).max`, then proofs may be replayed across chains whose IDs are congruent modulo 2^{32}.

Impact If deployed on sufficiently many chains, this protocol may become vulnerable to replay attacks.

Recommendation Validate that `block.chainid` is 32 bits. This could be done during initialization to avoid repeated checks.

Developer Response Acknowledged. We do not use chains with id larger than $2^{32} - 1$.

4.1.24 V-MLD-VUL-024: Missing Chainlink Oracle Checks

Severity	Warning	**Commit**	fde7102
Type	Data Validation	**Status**	Fixed
File(s)	src/oracles/MixedPriceOracleV3.sol		
Location(s)	_getLatestPrice()		
Confirmed Fix At	https://github.com/malda-protocol/malda-lending/pull/63, e0674a5		

When consuming data from off-chain oracles, it is important to ensure the data is not stale. As shown below, `MixedPriceOracleV3._getLatestPrice()` does validate that the data read from the price feed is within a certain staleness period. However, it uses a fixed `STALENESS_PERIOD` which is not set per-oracle.

```
 // Get price and timestamp
(uint80 roundId, int256 price,, uint256 updatedAt,) = feed.latestRoundData();
require(price > 0, MixedPriceOracle_InvalidPrice());
require(roundId > 0, MixedPriceOracle_InvalidRound());

// Check for staleness
require(block.timestamp - updatedAt < STALENESS_PERIOD, MixedPriceOracle_StalePrice()
    );
```

Snippet 4.24: Snippet from `MixedPriceOracleV3._getLatestPrice()`

Impact Stale oracle data could be used, allowing attackers to steal from the protocol in emergency scenarios where an oracle is shut down, but the protocol is not.

This may occur briefly for the `MixedPriceOracleV3` if some oracles have a much slower heartbeat than others, causing the `STALENESS_PERIOD` to be larger than is appropriate.

Recommendation Chainlink has a guide (https://docs.chain.link/data-feeds/price-feeds/addresses/?network=ethereum&page=1) with oracle heartbeats. These can be used to set appropriate staleness conditions. More info on ChainLink best practices can be found here: https://medium.com/cyfrin/chainlink-oracle-defi-attacks-93b6cb6541bf.

Note that this same issue appears in the out-of-scope contract `ChainlinkOracle`.

Developer Response The developers now have a default staleness period and optionally a per-symbol staleness period that can be customized by the bridge guardian.

Updated Veridise Response As mentioned in V-MLD-VUL-031, the authorization for `setStaleness()` should be switched to the new oracle guardian role.

Developer Response The developers have updated the authorization for `setStaleness()` to `GUARDIAN_ORACLE`.

4.1.25 V-MLD-VUL-025: Sweeping check may be insufficient

Severity	Warning	**Commit**	fde7102
Type	Data Validation	**Status**	Fixed
File(s)	`src/mToken/mErc20.sol`		
Location(s)	sweepToken()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/50/`, 353348b		

The `sweepToken()` function is designed for token rescue by the admin.

A strong user protection would check that the `underlying.balanceOf(address(this))` did not change, as some tokens have multiple proxies: `https://github.com/d-xo/weird-erc20?tab=readme-ov-file#multiple-token-addresses`

```
function sweepToken(IERC20 token) external onlyAdmin {
    require(address(token) != underlying, mErc20_TokenNotValid());
    uint256 balance = token.balanceOf(address(this));
    token.safeTransfer(admin, balance);
}
```

Snippet 4.25: Definition of `sweepToken()`

Impact For some tokens, a compromised or malicious admin may be able to take the `underlying` token balance directly without emitting a protocol event.

Recommendation Check that the `underlying` balance is unchanged after the transfer.

Developer Response The developers implemented the recommendation.

4.1.26 V-MLD-VUL-026: Unused/duplicate program constructs

Severity	Warning	**Commit**	fde7102
Type	Maintainability	**Status**	Partially Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/37`, 6f48503		

Description The following program constructs are unused:

1. `src/Counter.sol`
    a) `contract Counter`
2. `src/interfaces/IRoles.sol`:
    a) `GUARDIAN_TRANSFER`
    b) `GUARDIAN_SEIZE`
    c) `GUARDIAN_MINT`
    d) `GUARDIAN_BORROW`
3. `src/libraries/Bytes32AddressLib.sol`:
    a) `fillLast12Bytes()`
4. `src/mToken/mTokenStorage.sol`:
    a) `error mToken_RedeemTransferOutNotPossible`
    b) `error mToken_BlockNumberNotValid`
5. `src/mToken/extension/mTokenGateway.sol`:
    a) `_outHere()`: This function overwrites the `receiver` argument with the result of the proof, making the value passed in the `receiver` argument unused.
6. `src/mToken/host/mErc20Host.sol`:
    a) `_liquidateExternal()`, `_mintExternal()`, `_repayExternal()`: All of these functions overwrite the `receiver` argument with the result of the proof, making the value passed in the `receiver` argument unused.
7. `src/Operator/OperatorStorage.sol`:
    a) `error Operator_Deactivate_SnapshotFetchingFailed`
8. `src/utils/Deployer.sol`:
    a) `contract Deployer` is only used in deployment, and should be moved out of `src/`.
9. `src/Oracles/MixedPriceOracleV3.sol`:
    a) `getUnderlyingPrice()`,`_getPriceUSD()`: should use the constant DECIMALS
10. `src/Oracles/ChainlingOracle.sol`:
    a) `getUnderlyingPrice()`,`getPriceUSD()`: should use the constant DECIMALS

The following program constructs are duplicated or unnecessary:

1. `src/mToken/mTokenConfiguration.sol`:
    a) `mTokenConfiguration._setInterestRateModel.onlyAdmin`: This internal method is only called by `initialize()` and `setInternalRateModel`, both of which are already guarded by `onlyAdmin`.

Impact These constructs may become out of sync with the rest of the project, leading to errors if used in the future.

Developer Response The developers addressed several of the above comments, taking Veridise's recommendation.

Items 1, 3, 8, 9, and 10 referencing unused code are unaddressed.

Items 5 and 6 are intentionally acknowledged via the following comment when overwriting the receiver:

```
1 // temporary overwrite; will be removed in future implementations.
```

Updated Developer Response Item 1 is removed in a separate PR, item 3 is an (imported) external library.

4.1.27 V-MLD-VUL-027: Tokens with sender hooks are not supported

Severity	Warning	**Commit**	fde7102
Type	Reentrancy	**Status**	Acknowledged
File(s)	`src/mErc20.sol`		
Location(s)	_doTransferIn()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/60`, c4c1366		

The function `_doTransferIn()` determines the actual amount transferred by calculating a difference between balance before and after the `safeTransfer()` call.

If an attacker is able to take over execution in the sender hook IERC777Sender, they may invoke `LZMessageOnlyBridge.lzReceive()` (or `AccrossBridge.handleV3AcrossMessage`, if they are the spoke pool). These functions do not have a `nonReentrant` modifier and may be used to change `balanceAfter = IERC20(underlying).balanceOf(address(this))`.

```
function _doTransferIn(address from, uint256 amount) internal virtual override
    returns (uint256) {
   uint256 balanceBefore = IERC20(underlying).balanceOf(address(this));
   IERC20(underlying).safeTransferFrom(from, address(this), amount);
   uint256 balanceAfter = IERC20(underlying).balanceOf(address(this));
   return balanceAfter - balanceBefore;
}
```

Snippet 4.26: Snippet from `_doTransferIn()`

```
function lzReceive(Origin calldata _origin, bytes32 _guid, bytes calldata _message,
    address, bytes calldata)
        external
        payable
        override
{
   require(!processedOperations[_guid], LZBridge_OperationAlreadyProcessed());
   processedOperations[_guid] = true;
   require(peers[_origin.srcEid] == _origin.sender, LZBridge_PeerNotRegistered());

   address market, uint256 amount) = abi.decode(_message, (address, uint256));
   address _underlying = ImTokenMinimal(market).underlying();
   if (amount > 0) {
      IERC20(_underlying).safeTransfer(market, amount);
   }
```

Snippet 4.27: Snippet from `LZMessageOnlyBridge.lzReceive()`

In summary, the attacker calls `repay(negligibleAmount)` when bridge rebalancing is expected. They take over execution control using IERC777Sender and perform a call `LZMessageOnlyBridge.lzReceive()` . Provided the rebalancing is being processed on the right `market`, this leads to increase of `balanceAfter = IERC20(underlying).balanceOf(address(this))`.

Impact If tokens with hooks are used in the market, the rebalancing procedure may be intercepted by an adversary.

Recommendation Ensure the `_doTransferIn()` return value is smaller than or equal to the requested amount. Consider adding reentrancy guards for all functions performing calls to `IERC20(underlying).safeTransfer`.

Developer Response Acknowledged; Adding `nonReentrant` on Across bridge.

Updated Veridise Response The reentrancy described above is cross-contract. A guard on the Across bridge will not prevent it. Rather, the Across bridge would need to check against the `nonReentrant` guard of the `mToken`.

4.1.28 V-MLD-VUL-028: Malicious caller can cause bridge funds to be stuck

Severity	Warning	**Commit**	fde7102
Type	Denial of Service	**Status**	Fixed
File(s)	`src/rebalancer/bridges/messages/LZMessageOnlyBridge.sol`		
Location(s)	lzReceive()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/57/`, 354864e		

The protocol makes use of the LayerZero bridge to efficiently rebalance funds across the cross-chain markets. The `LZMessageOnlyBridge` contract implements the `ILayerZeroReceiverV2` and interacts with the LayerZero endpoint on the destination chain, to process the received bridge payload.

To process a payload one needs to call `retryPayload()` for a particular `guid`, after the message payload has been verified by the security stack. Then, the function calls `endpoint.lzReceive()` which leads to a callback to `LZMessageOnlyBridge.lzReceive()`, after which the funds are transferred to the decoded market address. See snippet below for the implementation.

```
function lzReceive(Origin calldata _origin, bytes32 _guid, bytes calldata _message,
    address, bytes calldata)
  external
  payable
  override
{
  require(!processedOperations[_guid], LZBridge_OperationAlreadyProcessed());
  processedOperations[_guid] = true;
  require(peers[_origin.srcEid] == _origin.sender, LZBridge_PeerNotRegistered());

  (address market, uint256 amount) = abi.decode(_message, (address, uint256));
  address _underlying = ImTokenMinimal(market).underlying();
  if (amount > 0) {
      IERC20(_underlying).safeTransfer(market, amount);
  }

  emit Rebalanced(market, amount);
}
```

Snippet 4.28: Snippet from `lzReceive()`

A malicious entity can call this function with arbitrary calldata and set `processedOperations[_guid] = true` for a particular `_guid`. When the actual endpoint does a callback for that particular `_guid`, the call will revert causing the bridged funds to be stuck.

Impact A malicious user can block bridged funds on the destination chain by calling `lzReceive()` with the particular message `_guid` and arbitrary calldata. This will process the particular `_guid` beforehand, reverting the callback from the LayerZero endpoint when the actual message payload is processed, causing the bridged funds to be stuck.

This contract is not intended to be used at the moment. But if it does end up getting used as it is in the future, then it can introduce serious issues.

Recommendation Add a check to `lzReceive()` which verifies that the caller is the registered LayerZero endpoint.

Developer Response The developers have acknowledged the issue and removed the contract `LzMessageOnlyBridge.sol`.

4.1.29 V-MLD-VUL-029: Locked dust in rebalancer

Severity	Warning	**Commit**	fde7102
Type	Usability Issue	**Status**	Fixed
File(s)	`src/rebalancer/Rebalancer.sol`		
Location(s)	Rebalancer		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/53/`, dcf9b88		

`sendMsg()` is `payable`, but there is no way to send funds out of the rebalancer/bridge contracts. This may lead to dust accumulating on the contracts.

Impact If `msg.value` is set incorrectly, a large amount of funds may be lost. Refunds from overpaying for bridges will not be reclaimable.

Recommendation Add a method allowing the `REBALANCER_EOA` to pull ETH from the `Rebalancer`.

Developer Response The developers added a function to the `Rebalancer` allowing the `GUARDIAN_BRIDGE` account to transfer ETH from the `Rebalancer` to itself.

4.1.30 V-MLD-VUL-030: Unpausable operation

Severity	Warning	**Commit**	fde7102
Type	Access Control	**Status**	Fixed
File(s)	src/mToken/extension/mTokenGateway.sol		
Location(s)	extractForRebalancing()		
Confirmed Fix At	https://github.com/malda-protocol/malda-lending/pull/59, a3d7340		

The function `extractForRebalancing()` is manipulating funds, however it cannot be paused in case of an emergency. See snippet below for context.

```
function extractForRebalancing(uint256 amount) external {
  if (!rolesOperator.isAllowedFor(msg.sender, rolesOperator.REBALANCER())) revert
    mTokenGateway_NotRebalancer();
  IERC20(underlying).safeTransfer(msg.sender, amount);
}
```

Snippet 4.29: Snippet from `extractForRebalancing()`

Impact This function can be used to withdraw funds even when all other protocol operations are paused.

Recommendation Add an additional OperationType to enable pausing the rebalancing process.

Developer Response The developers added an `OperationType.Rebalancing` which can be paused, and prevents `extractForRebalancing()` from occurring on either the gateway or the host.

They added this operation to `Pauser._pauseAllMarketOperations()`.

4.1.31 V-MLD-VUL-031: Responsibilities of a bridge guardian role

Severity	Warning	**Commit**	fde7102
Type	Access Control	**Status**	Fixed
File(s)	`./src/oracles/MixedPriceOracleV3.sol`		
Location(s)	setConfig()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/24`, 4686eb0		

The privileged role `GUARDIAN_BRIDGE` is used for access control in setting the price configuration. See snippet below for context.

```
function setConfig(
        string memory symbol, IDefaultAdapter.PriceConfig memory config
    ) external {
    if (!roles.isAllowedFor(msg.sender, roles.GUARDIAN_BRIDGE())) {
      revert MixedPriceOracle_Unauthorized();
    }
```

Snippet 4.30: Snippet from `setConfig()`

Impact The name of the role does not correspond to its usage. This may lead to unintended consequences in the segregation of duties.

Recommendation Introduce a new role specifically for setting the price configuration of price oracles.

Developer Response The developers added a `GUARDIAN_ORACLE` role to manage the price oracles.

4.1.32 V-MLD-VUL-032: Missing events

Severity	Warning	**Commit**	fde7102
Type	Missing/Incorrect Eve	**Status**	Fixed
File(s)	See description		
Location(s)	See description		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/58`		

Upon a state update, it is strongly recommended that developers emit an event to indicate that a change was made. Doing so allows both external users and protocol administrators to monitor the protocol for a variety of reasons, including for potentially suspicious activity. It is therefore critical for significant changes to the protocol to be accompanied with events to enable this monitoring. The following functions make important state updates but do not emit events recording the same.

1. `RewardDistributor.setOperator()` updates the `operator` address.
2. `RewardDistributor.whitelistToken()` adds a new token to the rewards token list.
3. `RewardDistributor.notifySupplyIndex()` updates the supply index within market state for all the specified tokens and the mToken.
4. `RewardDistributor.notifyBorrowIndex()` updates the borrow index within market state for all the specified tokens and the mToken.
5. `mTokenGateway.setPaused()` updates the paused status of the mTokenGateway contract.
6. `Deployer.setNewAdmin()`
7. `Deployer.setPendingAdmin()`
8. `Deployer.acceptAdmin()`

Impact If important state updates are made without administrators noticing, it can be seriously damaging to the protocol. For instance, in the `RewardDistributor` if the operator is updated without the administrators noticing, then the `admin` address for the contract has been compromised. The attacker can now perform a host of privileged actions which can cause serious issues for the protocol.

Recommendation Add an event that indicates when the state is updated within the above mentioned functions.

Developer Response The developers implemented the recommendation as provided in the issue description.

Veridise Response The function `_updateRewardSpeed()` updates the supply index and the borrow index. Their respective event emissions should be added here as well.

Updated Developer Response The developers implemented the recommendation.

4.1.33 V-MLD-VUL-033: Unchecked return value on LZMessageOnlyBridge communication

Severity	Warning	**Commit**	fde7102
Type	Maintainability	**Status**	Fixed
File(s)	LZMessageOnlyBridge		
Location(s)	sendMsg		
Confirmed Fix At	https://github.com/malda-protocol/malda-lending/pull/57/, 354864e		

In function sendMsg for LZMessageOnlyBridge, the return value from `send{...}(...)` call is ignored, and corresponding guid is not emitted as part of event.

```
1  function sendMsg(uint32 _dstChainId, address, bytes memory _message, bytes memory
       _bridgeData)
2          external
3          payable
4          onlyRebalancer
5      {
6          require(_dstChainId > 0, LZBridge_ChainNotRegistered());
7          require(peers[_dstChainId] != bytes32(0), LZBridge_PeerNotRegistered());
8
9          MessagingFee memory fees = _getFee(_dstChainId, _message, _bridgeData);
10         if (msg.value < fees.nativeFee || fees.lzTokenFee != 0) revert
       LZBridge_NotEnoughFees();
11
12         (uint256 gasLimit, address refundAddress) = abi.decode(_bridgeData, (uint256,
        address));
13         bytes memory options = getOptionsData(gasLimit);
14
15         endpoint.send{value: msg.value}(
16             MessagingParams(_dstChainId, peers[_dstChainId], _message, options, false
       ), refundAddress
17         );
18
19         emit MsgSent(_dstChainId, _message, gasLimit, refundAddress);
20     }
```

Snippet 4.31: Snippet from `sendMsg()`

Impact The event log is incomplete which complicates a potential incident response.

Recommendation Capture the return value, and include guid in the emitted event.

Developer Response The developers have acknowledged the issue, and removed `LZMessageOnlyBridge`.

4.1.34 V-MLD-VUL-034: Missing check of supported market

Severity	Info	**Commit**	fde7102
Type	Data Validation	**Status**	Fixed
File(s)	src/rebalancer/Rebalancer.sol		
Location(s)	sendMsg()		
Confirmed Fix At	https://github.com/malda-protocol/malda-lending/pull/40, 07dc82c		

The function `sendMsg()` does not perform a check whether external parameter `_market` is listed among supported markets.

```
1    function sendMsg(address _bridge, address _market, uint256 _amount, Msg calldata
     _msg) external payable {
2        if (!roles.isAllowedFor(msg.sender, roles.REBALANCER_EOA())) revert
     Rebalancer_NotAuthorized();
3        require(whitelistedBridges[_bridge], Rebalancer_BridgeNotWhitelisted());
4        address _underlying = ImTokenMinimal(_market).underlying();
5        require(_underlying == _msg.token, Rebalancer_RequestNotValid());
6
7        // retrieve amounts (make sure to check min and max for that bridge)
8        IRebalanceMarket(_market).extractForRebalancing(_amount);
```

Snippet 4.32: Snippet from `sendMsg()`

Impact The call `_market.extractForRebalancing(_amount)` may try to send funds which are not listed leading to ambiguous events being emitted.

Recommendation Verify the `_market` is listed by the operator.

Developer Response They developers added a method to the `Operator` allowing other contracts to check if the market is listed. Then, they use this method in `sendMsg()` to ensure the market is listed on its own operator.

4.1.35 V-MLD-VUL-035: Interest is per-second, not per-block

Severity	Info	**Commit**	fde7102
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/30`, 66a99aa		

Throughout the codebase, references to the current block number or latest block actually use the `block.timestamp`. The Malda developers indicated this is intentional, and due to the variable amount of time per block on the Linea blockchain.

```
function _getBlockNumber() internal view virtual returns (uint256) {
    return block.timestamp;
}
```

Snippet 4.33: Snippet from `src/mToken/mTokenStorage.sol`.

Impact Users reading the protocol may misunderstand the interest rates to be much lower than they actually are, since the simple interest rates are charged per-second rather than per-block.

Recommendation Rename references to block number or per-block rates as references to block timestamp and per-second rates.

Developer Response The developers renamed the external functions and errors to reference timestamps instead of block numbers.

Updated Veridise Response To increase code clarity, we recommend replacing all references of a block number in comments and variable names to reference a block timestamp instead.

Updated Developer Response The developers have replaced all references of a block number in comments and variables to reference a block timestamp instead.

4.1.36 V-MLD-VUL-036: Typos and Incorrect comments

Severity	Info	**Commit**	fde7102
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/42`, c5a3313		

Description In the following locations, the analysts identified minor typos and potentially misleading comments:

1. `src/interfaces/ImErc20.sol`:
   a) `repay()`: The natspec comment indicates a user may pass `-1` to repay the full amount. However, the `repayAmount` variable is a `uint256`. While the binary representations of `-1` as an `int256` and `type(uint256).max` are the same, the latter value is unambiguous.
   b) `repayBehalf()`: See `repay()` above.
2. `src/interfaces/ImErc20Host.sol`:
   a) `event mErc20Host_BorrowOnExternsionChain`: `Externsion` should be rewritten as `Extension`.
3. `src/libraries/mTokenProofDecoderLib.sol:`
   a) `decodeJournal()`: the advertised address market length is inconsistent with parameter in the call (@ L34).
   b) `decodeJournal()`: the `dstChainId` is missing from the doc comment (@ L37).
4. `src/mToken/mTokenStorage.sol:`
   a) `borrowRateMaxMantissa`: The value specified for `borrowRateMaxMantissa` in the comment above is inconsistent with the actual assigned value (@ L96).
5. `src/rebalancer/bridges/AcrossBridge.sol`:
   a) `AccrossBridge` has an extra `c`.
6. `src/Operator/Operator.sol`:
   a) `setMarketSupplyCaps()`: The local variable `numBorrowCaps` holds `newSupplyCaps.length`, and should instead be called `numSupplyCaps`.

Impact These minor errors may lead to future developer confusion.

Developer Response The developers fixed all of the above except for point (5).

4.1.37 V-MLD-VUL-037: Unchecked return value for transferring rewards

Severity	Info	**Commit**	fde7102
Type	Maintainability	**Status**	Fixed
File(s)	`RewardDistributor.sol`		
Location(s)	_grantReward		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/62`, 1dd9e37		

The return value from function call `transfer(user,amount)` is unchecked. Currently, the function `mToken.transfer(user,amount)` is called, which always returns `true` after calling `mToken._transferTokens(...)`. Future code changes may rely on the declared return value from function `transfer(user,amount)`.

```
1    if (amount > 0 && amount <= remaining) {
2       ImToken(token).transfer(user, amount);
3       emit RewardGranted(token, user, amount);
4       return 0;
5    }
```

Snippet 4.34: Snippet from `_grantReward(...)`

Impact If the token is updated, token transfers may silently fail.

Recommendation Verify the return value and adjust the flow accordingly in case of future code changes.

Developer Response The developers now verify the returned value as recommended.

4.1.38 V-MLD-VUL-038: Cannot cancel cross-chain action

Severity	Info	**Commit**	fde7102
Type	Usability Issue	**Status**	Acknowledged
File(s)	`mErc20Host.sol`, mTokenGateway		
Location(s)	mintExternal, supplyOnHost		
Confirmed Fix At	N/A		

The transfer from extension chain to host cannot be canceled. The funds may be used only for mint, liquidate, or repay.

```
1 function supplyOnHost(uint256 amount, bytes4 lineaSelector) external override
      notPaused(OperationType.AmountIn) {
2     ...
3     IERC20(underlying).safeTransferFrom(msg.sender, address(this), amount);
4     // effects
5     accAmountIn[msg.sender] += amount;
```

Snippet 4.35: Snippet from `supplyOnHost()`

Impact The user may be forced to use transferred funds even if the exchange rate changed significantly.

Recommendation Consider adding a function to send the funds back to the extension from which the funds originated to allow a user to cancel a transaction from extension to host.

Developer Response The Malda team has explored potential ways to mitigate this issue, and limitations of proving a negative and attack vectors with the ability to revert currently outweighed the benefits of introducing reverts. Potential expansion of solving this problem in case of liveness issue is a future development target to improve the user experience.

4.1.39 V-MLD-VUL-039: Bridging may be arbitraged

Severity	Info	**Commit**	fde7102
Type	Denial of Service	**Status**	Fixed
File(s)	`src/mToken/host/mErc20Host.sol`, `src/mToken/extension/mTokenGateway.sol`		
Location(s)	withdrawOnExtension(), supplyOnHost()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/34`, c268d95		

Currently, users may transmit funds cross-chain for only the cost of gas. For example, `withdrawOnExtension()` can be called immediately after minting on the host, allowing someone to transmit funds without fees from the Linea chain to any of the connected extension chains.

```
function withdrawOnExtension(uint256 amount, uint32 dstChainId) external override {
    require(amount > 0, mErc20Host_AmountNotValid());

    // actions
    accAmountOutPerChain[dstChainId][msg.sender] += amount;
    _redeemUnderlying(msg.sender, amount, false);

    emit mErc20Host_WithdrawOnExtensionChain(msg.sender, dstChainId, amount);
}
```

Snippet 4.36: Definition of `mErc20Host.withdrawOnExtension()`

Similarly, `supplyOnHost()`, followed by an `externalMint()` and withdrawal allows someone to transfer funds in the reverse direction.

Entities may use the protocol to perform cross-chain swaps with close to zero fees, arbitraging Malda against other cross-chain bridges.

Impact If the arbitrage leads to an imbalance of funds on chains, benign users of the protocol will not be able to withdraw funds on their desired chain. The rebalancer will be forced to act, transferring funds manually using a bridge, and thereby charging fees to the rebalancer.

The rebalancer will be forced to pay for this rebalancing out of pocket.

Recommendation Consider adding the ability for administrators to set fees on cross-chain actions, up to a set limit.

Developer Response The developers added a gateway-owner configured gas fee. This requires the `msg.value` is at least the gas fee when supplying to host from an extension chain.

A similar thing is done on host, but with the admin-configured fee set on a per-extension chain basis.

Updated Veridise Response This does appear to resolve the issue. Note that using `transfer` instead of `call` when gathering the gas fees may prevent the usage of some common, gas-intensive wallets.

5 Vulnerability Report: L1 Inclusion

This section presents the vulnerabilities found during the second security assessment on commit `e1bba48a`. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 5.1 summarizes the issues discovered:

Table 5.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-MLD2-VUL-001	Gateway does not verify L1 inclusion	Medium	Fixed
V-MLD2-VUL-002	Deprecated markets contribute to total . . .	Low	Fixed
V-MLD2-VUL-003	Wrong decimals used for outflow checks	Low	Fixed
V-MLD2-VUL-004	Outflow limit considerations	Low	Fixed
V-MLD2-VUL-005	Missing address zero-checks	Warning	Fixed
V-MLD2-VUL-006	Single-step ownership transfer	Warning	Acknowledged
V-MLD2-VUL-007	Unused code	Warning	Fixed
V-MLD2-VUL-008	Overly permissible parsing	Warning	Fixed
V-MLD2-VUL-009	Missing initializations	Warning	Fixed
V-MLD2-VUL-010	Solidity best practices	Info	Fixed

5.1 Detailed Description of Issues

5.1.1 V-MLD2-VUL-001: Gateway does not verify L1 inclusion

Severity	Medium	**Commit**	e1bba48
Type	Data Validation	**Status**	Fixed
File(s)	`src/mToken/extension/mTokenGateway.sol`		
Location(s)	_verifyProof()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/75/`		

The function `outHere()` does not verify the proof if the `msg.sender` is the `BatchSubmitter`, which can only be used by the Malda sequencer. To ensure users control their own funds, a user can submit a proof themselves through the "slow lane." To ensure security, proofs using the slow lane must validate the claimed host block has landed on the L1. The proof is then verified through `mTokenGateway._verifyProof()`.

```
function outHere(bytes calldata journalData, bytes calldata seal, uint256[] calldata
    amounts, address receiver)
    external
    notPaused(OperationType.AmountOutHere)
{
  // verify received data
  if (!rolesOperator.isAllowedFor(msg.sender, rolesOperator.PROOF_BATCH_FORWARDER()))
     {
      _verifyProof(journalData, seal);
  }

    // [Veridise] ..elided
}
```

Snippet 5.1: Snippet from `outHere()`

During the proof verification, the L1 inclusion field from the `journalData` is not verified. This exposes the protocol to potential double spending risks in case of a sequencer bug.

```
function _verifyProof(bytes calldata journalData, bytes calldata seal) private view {
  require(journalData.length > 0, mTokenGateway_JournalNotValid());

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

Snippet 5.2: Snippet from `outHere()`

Impact Since L1 inclusion of the L2 block is not verified, an attacker can steal funds from the protocol through double spending, if they are able to exploit a sequencer bug on the host chain.

Recommendation Verify that non-Malda user request's are included on the L1 before releasing funds.

Developer Response The developers added a "sequencer" role which may withdraw gas fees.

Additionally, they ensure only a caller with the `PROOF_FORWARDER` or `PROOF_BATCH_FORWARDER` role may bypass the L1 inclusion check.

5.1.2 V-MLD2-VUL-002: Deprecated markets contribute to total USD value

Severity	Low	**Commit**	e1bba48
Type	Data Validation	**Status**	Fixed
File(s)	`./src/Operator/Operator.sol`		
Location(s)	getUSDValueForAllMarkets()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/75/`, 8a596cb		

The function `getUSDValueForAllMarkets()` in the `Operator` returns the total USD value locked within the protocol. This includes the liquidity locked in deprecated markets as well. This can lead to a misrepresentation of protocol liquidity to third party listeners and users alike.

```
function getUSDValueForAllMarkets() external view returns (uint256) {
  uint256 sum;
  for (uint256 i; i < allMarkets.length;) {
      ImToken _market = ImToken(allMarkets[i]);
      uint256 totalMarketVolume = _market.totalUnderlying();
      sum += _convertMarketAmountToUSDValue(totalMarketVolume, address(_market));
      unchecked {  ++i; }
  }
  return sum;
}
```

Snippet 5.3: Snippet from `getUSDValueForAllMarkets()`

Impact Third party listeners or aggregators like DefiLlama may incorrectly interpret the sum of the USD value locked within Malda.

Recommendation Filter out deprecated markets when calculating the total sum of the USD value for all markets.

Developer Response The developers now skip markets which are not listed.

Updated Veridise Response Every market in `allMarkets` has `isListed` set to true. The developers should instead use the `_isDeprecated()` function.

Update Developer Response The developers now make use of `_isDeprecated()` to filter deprecated markets when calculating the total USD value of all markets.

5.1.3 V-MLD2-VUL-003: Wrong decimals used for outflow checks

Severity	Low	**Commit**	e1bba48
Type	Logic Error	**Status**	Fixed
File(s)	`src/Operator/Operator.sol`		
Location(s)	_convertMarketAmountToUSDValue()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/75/`, 8ec3fe7		

Each market is connected to an `Operator` which is responsible for ensuring market users remain solvent. Additionally, the `Operator` contract can set a limit on the total amount of funds accepted from extension chains within a certain window. This amount of funds is called the "outflow," and is denominated in USD. To convert amounts of underlying into USD for comparison with outflow limits, the `Operator` contract uses the `_convertMarketAmountToUSDValue()`, shown in the below snippet.

However, for an underlying with `decimals`-many decimals (i.e. one underlying token is represented numerically as `10**decimals`), the oracle returns the USD value of one `10**-decimals` underlying token, represented with 36 decimals.

For example, if USDC is the underlying token, has 6 decimals, and is currently valued at 1 USD, then `getUnderlyingPrice()` will return `10**(36-6)`. Since `mul_(a,b) = a*b/1e18`, calling `mul_()` in the below code snippet would return `10**(18 - 6) * amount`. Since USDC has 6 decimals, the number `amount` represents `amount / 10**6` USDC. Consequently, we see that `_convertMarketAmountToUSDValue()` returns the USD value *represented in 18 decimals.*

```
1 function _convertMarketAmountToUSDValue(uint256 amount, address mToken) private view
      returns (uint256) {
2     address _asset = ImToken(mToken).underlying();
3     uint256 oraclePriceMantissa = IOracleOperator(oracleOperator).getUnderlyingPrice(
      _asset);
4     require(oraclePriceMantissa != 0, Operator_OracleUnderlyingFetchError());
5
6     Exp memory oraclePrice = Exp({mantissa: oraclePriceMantissa});
7     uint256 amountInUSD = mul_(amount, oraclePrice);
8     return amountInUSD;
9 }
```

Snippet 5.4: Definition of `Operator._convertMarketAmountToUSDValue()`

Impact Elsewhere in the codebase, the `getUnderlyingPrice()` is used in a way that expects `mul_(amount, getUnderlyingPrice())` to return the value in USD, represented in 18 decimals. For example, `liquidateCalculateSeizeTokens()` divides out the factor of `10**18`. The function `_getHypotheticalAccountLiquidity()` uses `getUnderlyingPrice()` to compute the value of collateral and borrows spread across different token types in a common denomination.

However, `_convertMarketAmountToUSDValue()` is used specifically for comparison against a limit expressed in USD. Returning the value with eighteen decimals may lead to immediate failures when an outflow limit is set.

Recommendation Clearly specify the decimals being returned by `getUnderlyingPrice`. Consider adding a helper function which performs the multiplication and returns the USD value in a fixed number of decimals for clarity.

Developer Response The developers now divide by `10**(18-assetDecimals)`.

Updated Veridise Response In testing the solution, the analysts believe there are two more issues:

1. `getUnderlyingPrice()` expects the `mToken` to be provided, not the `asset`.
2. To convert to the expected price, one would just need to divide by `1e18` (if the target decimals is 1).

Since tracking decimals can be notoriously difficult, the Veridise team recommends adding tests with different token decimals and documenting the intended decimals of the resultant value. For convenience, the test has been shared with the developers.

Updated Developer Response The developers took the above recommendation (using `1e10` rather than `1e18` since the target decimals is 8, not 1), and noted an underflow preventing usage of tokens with decimals larger than 18. This was deemed to be outside of the intended use case.

5.1.4 V-MLD2-VUL-004: Outflow limit considerations

Severity	Low	**Commit**	e1bba48
Type	Denial of Service	**Status**	Fixed
File(s)	`./src/mToken/host/mERC20Host.sol`		
Location(s)	See description		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/76`		

Malda introduces an outflow limit across all cross-chain actions, which is used to limit the flow of outward funds within a particular time period. This limit is implemented on the host chain and tracks the "outflow" across all cross-chain mints (extension to host), repays (extension to host), borrows (host to extension), and withdraws (host to extension), but not liquidations.

This limit is introduced to protect against a scenario in which one of the sequencers has been compromised or reorged. In such a scenario, the worst-case fear is that someone has a valid proof from another chain's sequencer showing that they are owed funds, but the particular proof is invalid due to reorg/sequencer compromise. See snippet below for context on how the outflow limit is implemented.

```
function mintExternal(
    bytes calldata journalData,
    bytes calldata seal,
    uint256[] calldata mintAmount,
    uint256[] calldata minAmountsOut,
    address receiver
) external override {
    if (!_isAllowedFor(msg.sender, _getBatchProofForwarderRole())) {
        _verifyProof(journalData, seal);
    }

    _checkOutflow(_computeTotalOutflowAmount(mintAmount));

        // [Veridise] .. elided
}
```

Snippet 5.5: Snippet from `mintExternal()`

The outflow check should ideally limit the maximum amount of damage that can be done within a single time window, by limiting the volume of funds that can flow out of the protocol. But, the current implementation fails to take into consideration certain scenarios which can lead to issues for the protocol. These are detailed below.

The outflow of funds within a certain time period is limited. This can be used as an avenue to block repays. Malicious actors can attack user repay operations by performing large mints and using up the outflow limit while the repay is pending. This is especially dangerous because liquidations are not beholden to the outflow limit, and therefore users may get liquidated if they are continuously DOSed when trying to repay their debt.

The outflow limit also assumes and protects only against a particular kind of host-chain reorg: one in which there is exactly one fork. If there were to be an emergency event, multiple signed forked blocks can appear. In such a case, each block would provide an opportunity to steal `outflowVolumeLimit` dollars. Since the outflow limit is designed to limit the damage to the

protocol, in such a case it should also implement a corresponding check on the outflow of funds on the extension chains.

Impact Depending on the configuration settings, actors with a large amount of liquidity can DOS pending repay operations by performing a large mint. Since liquidations are not subject to the outflow limits, this can result in users with funds on extension chains not being able to pay back their debt and consequently getting liquidated.

Secondly, user protection against Linea sequencer errors or compromise is limited. It is the responsibility of the Malda-operated, off-chain "Malda sequencer" to avoid posting proofs based on blocks which may not become part of the finalized Linea chain. This could be especially devastating if the Linea sequencer were compromised so that L2 blocks could be intentionally held back from the peer-to-peer gossip network used to obtain blocks before they are posted to the L1.

Note that this attack will not be exploitable using proofs that require L1 inclusion, since this will prove the claimed Linea block was posted to Ethereum and verified. To falsify a proof of L1 inclusion, an attacker would have to simultaneously compromise the Optimism sequencer (which is attesting to the L1 state) *and* cause a Linea sequencer bug.

Recommendation

1. To deal with repay DoS-ability, consider implementing one of the following recommendations.
    a) Allow a trusted administrator to reset the outflow limit based on the current state of the network.
    b) Consider offsetting the outflow limit with rebalanced funds.
    c) Allow users to deposit and repay from extension to the host immediately, but escrow their tokens once the outflow limit is reached. The escrow repayments can then be treated conservatively. During liquidation, the debt may be treated as (debt - escrowRepayments). For borrows, transfers and redeems may treat the debt as debt ignoring the escrow repayments. This will ensure solvency under normal conditions, while preventing any attackers from using double-spends to repay debts in the case of sequencer compromise.
2. Consider limiting the outflow of funds on the extension chains as well. This can be handled for the fast lane through an off-chain component on the Malda centralized sequencer. Since the slow-lane self-sequencing ensures that the L2 block lands on L1, this particular risk of reorgs is eliminated for the slow lane.
3. Add appropriate documentation aimed to educate the user regarding the risks above and the steps taken to mitigate it.

Developer Response Implementing escrow tokens is a valid approach, but it would also reengineer the logic of the protocol, and therefore it is out of scope before the next major update. We believe for users in high volatility the issue is the so-called latency gap between UserOp and recording the transaction on Linea. This exists even with escrow tokens, as the proving latency is 20-30 seconds while direct liquidation on Linea is 2s.

It is important for users to understand this limitation, irrespective of escrow tokens existing or not and we have to clarify this in product related communications.

The mitigation of this issue is operational and users borrowing at a point where realistic market volatility can affect them should always elect to directly repay on Linea. The DoS could also happen without malicious actions, as the amount of repay in high volatility situation is expected to increase.The cost of attack in terms of allocation is also quite high, due to the relatively high size of inflow and outflow + potential gain from liquidations.

Updated Veridise Response Given the above limitations, we recommend allowing a trusted administrator to reset the outflow limit based on independent verification of the funds on the extension chain.

Updated Developer Response The developers added a function which allows an administrator to reset the outflow limit.

5.1.5 V-MLD2-VUL-005: Missing address zero-checks

Severity	Warning	**Commit**	e1bba48
Type	Data Validation	**Status**	Fixed
File(s)	See issue description		
Location(s)	initialize(), constructor()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/75/`, f29ec96		

Description The following functions take a `zkVerifier_` address as an argument, but do not validate that the address is non-zero:

- `src/mToken/`
  - `extension/mTokenGateway.sol`:
    - `mTokenGateway.initialize()`
  - `host/mErc20Host.sol`:
    - `mErc20Host.initialize()`
  - `BatchSubmitter.sol`:
    - `BatchSubmitter.constructor()`

Impact If zero is passed as the address, the deployed protocol will be unusable.

Developer Response The developers now validate the `zkVerifier_` address is non-zero when initializing the `mErc20Host` and `mTokenGateway`.

Updated Veridise Response The `BatchSubmitter` `constructor()` function does not validate the `zkVerifier`'s address to be non-zero.

Updated Developer Response The developers now validate that the `zkVerifier` is non-zero in the constructor of the `BatchSubmitter` as well.

5.1.6 V-MLD2-VUL-006: Single-step ownership transfer

Severity	Warning	**Commit**	e1bba48
Type	Authorization	**Status**	Acknowledged
File(s)	`src/verifier/ZkVerifier.sol`		
Location(s)	ZkVerifier		
Confirmed Fix At	N/A		

The `ZkVerifier` contract inherits from `Ownable`. This provides the `ZkVerifier` with the functionality for centralized control by a single `owner` address. This `owner` address is allowed to perform specific actions, including

1. Changing the `verifier` address, i.e. the address used to verify Risc0 receipts.
2. Changing the `imageId`, i.e. the program verified against the Risc0 seal.

When ownership is transferred, `Ownable` does this in a single step. This means that any typo or error may DoS the contract by setting the `owner` to an address not controlled by the intended owning party.

Impact Ownership transfers are more dangerous, as any typo or minor error may DoS the protocol.

Recommendation Use a 2-step ownership transfer pattern, as implemented in `Ownable2Step`.

Developer Response This is not going to be an issue after launch due to a Timelock contract structure. Upgrades will be locked behind a timelock that enables self-sequencing exit from the protocol with a sufficient time delay (withdrawal self-sequencing from Linea 4 hrs, timelock 3 days). This means the main multisig will need to propose a change, and a smaller operating multisig will need to execute the change, resulting in a 2 step change needed.

5.1.7 V-MLD2-VUL-007: Unused code

Severity	Warning	**Commit**	e1bba48
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/75/`		

Description The following program constructs are unused:

1. `src/verifier/ZkVerifier.sol`:
    a) `error ZkVerifier_L1InclusionRequired`

Impact Unused code may be incorrectly relied on by third-party listeners, or introduce errors if used in later versions of the codebase.

Developer Response The developers removed the unused error.

5.1.8 V-MLD2-VUL-008: Overly permissible parsing

Severity	Warning	**Commit**	e1bba48
Type	Data Validation	**Status**	Fixed
File(s)	`src/libraries/mTokenProofDecoderLib.sol`		
Location(s)	decodeJournal()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/75`		

The `mTokenProofDecoderLib` library is used to decode the public input/output of the journals, which are verified against the Malda ZK circuit. The `decodeJournal()` function takes in the journal bytes and decodes the result into its constituent parts. The decoding logic is shown in the below snippet.

The last byte of the journal is intended to represent a bit. This is assumed in the decoding logic. In the below implementation, `0x01` decodes to `1`, and the 255 other possible bytes decode to `0`.

```
// decode action data
// | Offset | Length | Data Type                |
// |--------|--------|------------------------- |
// | 0      | 20     | address sender           |
// | 20     | 20     | address market           |
// | 40     | 32     | uint256 accAmountIn      |
// | 72     | 32     | uint256 accAmountOut     |
// | 104    | 4      | uint32 chainId           |
// | 108    | 4      | uint32 dstChainId        |
// | 112    | 1      | bool L1inclusion         |
sender = BytesLib.toAddress(BytesLib.slice(journalData, 0, 20), 0);
market = BytesLib.toAddress(BytesLib.slice(journalData, 20, 20), 0);
accAmountIn = BytesLib.toUint256(BytesLib.slice(journalData, 40, 32), 0);
accAmountOut = BytesLib.toUint256(BytesLib.slice(journalData, 72, 32), 0);
chainId = BytesLib.toUint32(BytesLib.slice(journalData, 104, 4), 0);
dstChainId = BytesLib.toUint32(BytesLib.slice(journalData, 108, 4), 0);
L1inclusion = BytesLib.toUint8(BytesLib.slice(journalData, 112, 1), 0) == 1;
```

Snippet 5.6: Definition of `decodeJournal()`

Impact If any future error allows an attacker to create a proof for a journal with an invalid `L1inclusion` byte, the journal will decode without errors. This adds an unnecessary extra dependency on the ZK circuit for correct parsing.

Recommendation Validate the `L1Inclusion` byte is either 0 or 1. Additionally, validate that the `journalData` is exactly the expected length.

Developer Response The developers now validate that the `journalData` is the expected length and the `L1Inclusion` is either 0 or 1.

5.1.9 V-MLD2-VUL-009: Missing initializations

Severity	Warning	**Commit**	e1bba48
Type	Logic Error	**Status**	Fixed
File(s)	`src/Operator/Operator.sol`		
Location(s)	Operator.initialize()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/75/`		

The `Operator` is responsible for configuring the outflow limits-limits on how many funds may be accepted from extension chains on the host. However, the `initialize()` function does not set two of these parameters:

1. `limitPerTimePeriod`: The maximum number of funds per `outflowResetTimeWindow` which can be accepted from extension chains.
2. `lastOutflowResetTimestamp`: The start of the current outflow period.

Impact Protocols may be initialized with unsafe outflow settings. Administrators may forget to set these limits, rendering them useless.

Recommendation Initialize `limitPerTimePeriod` and `lastOutflowResetTimestamp` when deploying an `Operator`.

Developer Response The developers now initialize both `lastOutflowResetTimestamp` and `limitPerTimePeriod` in the `Operator` initializer. The project is initialized with no limits.

5.1.10 V-MLD2-VUL-010: Solidity best practices

Severity	Info	**Commit**	e1bba48
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/malda-protocol/malda-lending/pull/75/`		

Following the below Solidity best practices will increase maintainability of the protocol.

1. `src/`
    a) `Operator/Operator.sol`:
        i. `setOutflowTimeLimitInUSD()`: Administrators or off-chain listeners may expect that setting the `limitPerTimePeriod` to zero will prevent all cross-chain transfers. However, it actually disables all outflow limits. This fact should be documented on the setter.
        ii. `checkOutflowVolumeLimit()`: The natspec of this function contains a typo, writing `volule` instead of `volume`.
    b) `verifier/ZkVerifier.sol`:
        i. `ZkVerifier` should implement the `IZkVerifier` interface. This will ensure that the two program constructs remain synchronized to any future changes.

Recommendation Implement the above best practices.

Developer Response The developers documented the behavior of `setOutflowTimeLimitInUSD()` and updated the `ZkVerifier` to implement the `IZkVerifier` interface.

Updated Veridise Response While point (a)(ii) remains unresolved, it is nonfunctional and has a clear intended meaning.

Glossary

Base An Ethereum L2, decentralized with the Optimism Superchain, and incubated by Coinbase. See `https://www.base.org` to learn more. 1

Chainlink An on-chain solution for off-chain data-sources, most commonly price feeds. See `https://chain.link` to learn more. 5

Compound Compound is an open source over-collateralized lending protocol. To learn more, visit `https://compound.finance`. 1

Denial-of-Service An attack in which the liveliness or ability to use a service is hindered . 9

ERC-20 The famous Ethereum fungible token standard. See `https://eips.ethereum.org/EIPS/eip-20` to learn more. 1

front-running A vulnerability in which a malicious user takes advantage of information about a transaction while it is in the mempool. 9

Linea A zkEVM layer-2 blockchain. See `https://linea.build` to learn more . 1

Optimism A layer-2 network built on top of the Ethereum blockchain as an optimistic rollup. Visit `https://www.optimism.io` to learn more. 1, 80

optimistic rollup A rollup in which the state transition of the rollup is posted "optimistically" to the base network. A system involving stake for resolving disputes during a challenge period is required for economic security guarantees surrounding finalization. 80

reentrancy A vulnerability in which a smart contract hands off control flow to an unknown party while in an intermediate state, allowing the external party to take advantage of the situation. 9

Risc Zero A zkVM intended to run programs in the Risc-V instruction set. Visit `https://risczero.com` to learn more. 1

rollup A blockchain that extends the capabilities of an underlying base network, such as higher throughput, while inheriting specific security guarantees from the base network. Rollups contain smart contracts on the base network that attest the state transitions of the rollup are valid. 80

Rust A programming language especially well-known for its borrow semantics. See `https://doc.rust-lang.org/book/` to learn more. 1

Sequencer An entity or distributed algorithm responsible for determining transaction and block ordering on a layer-2 network.. 1

smart contract A self-executing contract with the terms directly written into code. Hosted on a blockchain, it automatically enforces and executes the terms of an agreement between buyer and seller. Smart contracts are transparent, tamper-proof, and eliminate the need for intermediaries, making transactions more efficient and secure. 1, 80

zero-knowledge circuit A cryptographic construct that allows a prover to demonstrate to a verifier that a certain statement is true, without revealing any specific information about the statement itself. See `https://en.wikipedia.org/wiki/Zero-knowledge_proof` for more. 81

zkEVM A zkVM implementing the Ethereum virtual machine. 80

zkVM A general-purpose zero-knowledge circuit that implements proving the execution of a virtual machine. This enables general purpose programs to prove their execution to outside observers, without the manual constraint writing usually associated with zero-knowledge circuit development. 1, 9, 80, 81