Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Malda

Veridise Inc.

April 14, 2025

► **Prepared For:**

Malda
https://docs.malda.xyz

► **Prepared By:**

Mark Anthony
Petr Susil
Benjamin Sepanski

► **Contact Us:**

contact@veridise.com

► **Version History:**

Apr. 14, 2025	V6 - Incorporate issue fixes, address typos
Apr. 11, 2025	V5 - Incorporate issue fixes
Apr. 02, 2025	V4 - Incorporate L1 inclusion review
Mar. 17, 2025	V3 - Metadata updates and incorporating developer response
Feb. 28, 2025	V2 - Incorporated issue fixes
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 Rust programs intended to be run in the RISC Zero zkVM, as well as the Malda smart contracts, which together implement a cross-chain over-collateralized lending protocol. Following this review, Malda engaged Veridise from Mar. 24 to Mar. 27 to conduct a security assessment of the Malda L1 Inclusion feature. This report only focuses on the zk-coprocessor programs. A companion report discusses the findings from the coincident smart contract review. Veridise conducted the first assessment over 12 person-weeks, with 3 security analysts reviewing the project over 4 weeks on commit `5a570514`. The second assessment occurred over 8 person-days, with 2 security analysts reviewing the project over 4 days on commit `2095dda1`. 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 first security assessment covered the Rust programs used to create proofs, as well as the Rust programs executed inside of the RISC Zero zkVM. Malda provided two programs to the Veridise developers. Both are intended to execute a view call within an Ethereum execution environment using the RISC Zero Steel library*. Subsequently, the programs attempt to validate the execution environment by proving there is a sufficiently long chain of blocks ending in a "validated" beacon block.

The two programs differ in how the final block is validated. The first program, intended to be used in production, validates that the network Sequencer (either Linea, Optimism, or Base) has signed the final block. For the Ethereum network, the `L1Block` contract† on the Optimism network is used to obtain the block hash. The second program, not yet planned for production, runs the Ethereum light client sync protocol‡ as implemented by Helios§.

The second review covered the Malda L1 Inclusion workflow. To protect against potential block reorgs, users may prove the claimed L2 block is included in the L1 blockchain. This is done by accessing the L1 state through a block signed by the Optimism sequencer. As the L1 block posted to Optimism is delayed by two L1 block confirmations, this allows users to prove their L2 block is cross-safe. For OP-stack chains, this path no longer makes assumptions about the Optimism sequencer implementation preventing reorgs, but instead only relies on the Optimism sequencer to not attest to spoofed or malicious L1 blocks. For Linea chains, only the block number is checked against the L1. 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).

* https://github.com/risc0/risc0-ethereum/tree/release-1.2/steel
† https://github.com/ethereum-optimism/optimism/blob/0e4b867e08ed4dfcb5f1a76693f17392b189a7f6/packages/contracts-bedrock/src/L2/L1Block.sol
‡ https://github.com/ethereum/consensus-specs/blob/7480d0334fd81cd029bc85b38c5c7bdc2967acbf/specs/altair/light-client/sync-protocol.md
§ https://github.com/a16z/helios/tree/0.6.0

The scope of this security assessment explicitly excludes the security implications of relying upon a sequencer signature or the latest block signed by the sync committee. For a discussion of these risks, review the companion report assessing the Malda smart contracts, which discusses the integration and usage of these programs within the context of the over-collateralized lending protocol.

Code Assessment. The Malda developers provided the source code of the Malda contracts for the code review. The source code appears to be mostly original code written by the Malda developers. It contains some documentation in the form of READMEs and 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.

For the second review, the Malda engaged in several detailed discussions with the Veridise team. They then provided a writeup of their planned implementations along with the code and access to an internal machine used for running tests.

The source code contained a test suite covering proof generation workflow and panic conditions for functions used in proof generation. The second review's test suite was expanded with some tests which check for successful trace generation when using contracts deployed on the Linea testnet.

Summary of Issues Detected. Issues from the initial security assessment are listed in Section 4. The initial security assessment uncovered 14 issues, 2 of which are assessed to be of high or critical severity by the Veridise analysts. Specifically, V-MLDZK-VUL-001 describes how to prove arbitrary outputs by changing the contract address, and V-MLDZK-VUL-002 flags incorrect padding which prevents proper execution. The Veridise analysts also identified 2 low-severity issues, including V-MLDZK-VUL-004, which identifies that any sequencers may falsely attest to another sequencer's chain, and V-MLDZK-VUL-005, which describes Malda's intentional use of the wrong view call at the beginning of the audit.

Finally, the Veridise analysts identified 6 warnings, and 4 informational findings. These include unused functions V-MLDZK-VUL-006, missing checks in the light client V-MLDZK-VUL-007, and out-dated dependencies V-MLDZK-VUL-008.

Issues from the second security assessment are listed in Section 5. The second assessment uncovered 9 issues, 2 of which are assessed to be of high or critical severity by the Veridise analysts. Specifically, V-MLDZK2-VUL-001 describes how the Optimism dispute game resolution status is not checked when proving L1 state, and V-MLDZK2-VUL-002 shows that the Ethereum testnet is always used to prove L1 inclusion. The Veridise analysts also identified 2 medium-severity issues, including V-MLDZK2-VUL-003 in which an incorrect inequality allows attackers to bypass the L1 inclusion check and V-MLDZK2-VUL-004 in which checking the wrong state root may deny service to honest users.

Finally, the Veridise analysts identified 3 warnings and 2 informational findings. Most importantly V-MLDZK2-VUL-005 explains that the L1 inclusion check is used in place of a sequencer signature check for OP-chains, rather than in addition to a sequencer signature check. This was

necessary for the escalation of both critical issues from a weakened security guarantee to theft of funds.

Of the 14 acknowledged issues from the initial review, Malda has fixed 12 issues. This includes all high- and critical-severity issues. Malda does not plan to fix the other 2 acknowledged issues at this time. Namely, V-MLDZK-VUL-004 describes that the optimistic header is used for proofs, instead of the finalized one, and V-MLDZK-VUL-013 describes a future upgrade which will be required to handle Optimism blocks after the Isthmus upgrade.

Among the 9 issues raised in the follow up review, Malda has fixed 8 issues and provided partial fixes to 1 more. This includes all 2 critical-severity issues and 1 medium-severity issue. Malda does not plan to fix the 1 remaining acknowledged issue at this time. Namely V-MLDZK2-VUL-005, which describes how the Optimism sequencer check is skipped in the L1 inclusion path. Malda has mentioned in a detailed developer response why addressing the issue would not be practical for them. The analysts recommend addressing the issue for a stronger security guarantee.

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

Additional Testing. Developers should perform additional and rigorous testing on the new L1 inclusion component. The current tests only check that one chain interaction executes successfully. All supported chains should be tested. The tests should check the output of the journals match the expected behavior. The Veridise analysts recommend executing the tests with a forked blockchain and mocked L2 sequencers so that the system may be rigorously tested without being deployed. For example, this could have prevented V-MLDZK2-VUL-002.

Monitoring. Developers should keep in close contact with the L2 teams, and monitor the blocks signed by the Sequencers (see, e.g., V-MLDZK-VUL-013). Any changes in sequencer operation, such as key rotations, sequencer decentralization, or protocol upgrades may prevent the circuits from operating.

Overflow checks. Integer operations may overflow in Rust. Activating overflow checks in the configuration as described in the rust book will fully remove any potential attack surface related to overflows. Alternatively, rust provides separate functions like `checked_mul()`, `checked_add()` which may be used for operations where overflow is critical to be checked as part of the proof.

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.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Malda	`5a570514`	RISC Zero	Linea, Optimism, Base
Malda L1 Inclusion	`2095dda1`	RISC Zero	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: Malda.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	1	1	1
High-Severity Issues	1	1	1
Medium-Severity Issues	0	0	0
Low-Severity Issues	2	2	1
Warning-Severity Issues	6	6	6
Informational-Severity Issues	4	4	3
TOTAL	14	14	12

Table 2.4: Vulnerability Summary: Malda L1 Inclusion.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	2	2	2
High-Severity Issues	0	0	0
Medium-Severity Issues	2	2	2
Low-Severity Issues	0	0	0
Warning-Severity Issues	3	3	2
Informational-Severity Issues	2	2	2
TOTAL	9	9	8

Table 2.5: Category Breakdown: Malda.

Name	Number
Data Validation	5
Maintainability	4
Logic Error	3
Information Leakage	1
Usability Issue	1

Table 2.6: Category Breakdown: Malda L1 Inclusion.

Name	Number
Logic Error	4
Maintainability	2
Data Validation	1
Denial of Service	1
Missing/Incorrect Events	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

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

- Is the RISC Zero environment initialized correctly?
- Is the data contained in the proof packed correctly?
- Is the proof bound to the requested chain?
- Do the claims in the proof correspond to the state of a valid mToken?
- Does the proof ensure a valid history for blocks which are used as a part of the proof? This includes:
  - Are there sufficient protections against reorgs?
  - Is the signature from a sequencer valid, and does it match the appropriate chain?
  - Are the checkpoints for the light client sufficiently validated?
- Can a proof be replayed to extract funds?
- Are there any opportunities for an attacker to exploit the protocol due to the zk cross-chain communication?
- Are the trust anchors and trust base for the proof minimized?
- Are there any opportunities for an attacker to perform denial-of-service?
- Are there opportunities for an attacker to create conditions, which can lead to the insolvency of a user?
- Are the project dependencies up-to date and secure?
- Can the L1 inclusion checks be bypassed?
- How does the security guarantee of the L1 inclusion path compare to the fast-lane verification path?
- Are L2 blocks correctly validated to reside on the L1?

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 the open-source tools `cargo audit` and Semgrep. These tools are designed to find known issues in dependencies and common vulnerabilities in Rust programs.

Scope. The scope of the initial security assessment at commit `5a570514` is limited to the following folders of the source code provided by the Malda developers, which contains the implementation of Malda:

- `malda_rs/src/`
- `methods/guest/guest_utils/src/`
- `methods/guest/src/`
- `methods/src/`
- `patch/ethereum_hashing/src/`

During the fix review, the directory `methods/guest/guest_utils/src/` was removed, and its contents were moved to `malda_utils/src`.

The scope of the second security assessment at commit `2095dda1` is limited to the following folders of the source code provided by the developers:

- `malda_utils/src`

This includes all changes to the guest code (i.e. the verification logic checked on-chain) of the protocol.

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 both static analyzers and automated testing.

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 `5a570514`. 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-MLDZK-VUL-001	Arbitrary data committed to journal . . .	Critical	Fixed
V-MLDZK-VUL-002	Balance queries performed on incorrect . . .	High	Fixed
V-MLDZK-VUL-003	Hard-Coded Sequencers	Low	Fixed
V-MLDZK-VUL-004	Optimistic update used in light client	Low	Acknowledged
V-MLDZK-VUL-005	balanceOf used in place of getProofData	Warning	Fixed
V-MLDZK-VUL-006	Unused/duplicate program constructs	Warning	Fixed
V-MLDZK-VUL-007	Light client circuit may accept future blocks	Warning	Fixed
V-MLDZK-VUL-008	Helios 0.6.0 missing checks	Warning	Fixed
V-MLDZK-VUL-009	Hardcoded RPC with API keys	Warning	Fixed
V-MLDZK-VUL-010	Signature may be malleable	Warning	Fixed
V-MLDZK-VUL-011	Typos and Incorrect comments	Info	Fixed
V-MLDZK-VUL-012	Best practices	Info	Fixed
V-MLDZK-VUL-013	Execution blocks assumed to be V3	Info	Acknowledged
V-MLDZK-VUL-014	The OP Stack environment validation will . . .	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-MLDZK-VUL-001: Arbitrary data committed to journal permits spoofed mints

Severity	Critical	**Commit**	5a57051
Type	Data Validation	**Status**	Fixed
File(s)	`./methods/guest/src/bin/balance_of.rs`		
Location(s)	main()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-zkcoprocessor/pull/9`, `https://github.com/malda-protocol/malda-lending/pull/51`, 15f0acc, 65845f4		

In `balance_of.rs`, the method `main()` reads the necessary state information from the provided `EthEvmInput` and proceeds to execute and validate the balance queries across multiple accounts and tokens through `validate_balance_of_call()`. See the snippet below for context.

```
fn main() {
    let mut output: Vec<Bytes> = Vec::new();
    let length: u64 = env::read();
    for _i in 0..length {
        // Read the input data for this application.
        let env_input: EthEvmInput = env::read();
        let chain_id: u64 = env::read();
        let account: Vec<Address> = env::read();
        let asset: Vec<Address> = env::read();
        let sequencer_commitment: Option<SequencerCommitment> = env::read();
        let env_op_input: Option<EthEvmInput> = env::read();
        let linking_blocks: Vec<RlpHeader<Header>> = env::read();

        validate_balance_of_call(chain_id, account, asset, env_input,
    sequencer_commitment, env_op_input, linking_blocks, &mut output);
    }
    env::commit_slice(&output.abi_encode());
}
```

Snippet 4.1: Snippet of `main()` from `balance_of.rs`

The resulting output of the view calls is then committed to the journal, which is decoded when the proof is submitted for verification in the contracts. The decoded journal data is used to verify and ascertain the change in balances pertaining to the specified user, the `mToken` market, and the concerned `chainId`. See the snippet of `_mintExternal` below to understand how the decoded journal data is used.

```
function _mintExternal(bytes memory singleJournal, uint256 mintAmount, address
    receiver) internal {
(address _sender, address _market, uint256 _accAmountIn,, uint32 _chainId, uint32
    _dstChainId) =
    mTokenProofDecoderLib.decodeJournal(singleJournal);

    receiver = _sender;
```

```
6
7       // base checks
8       {
9       _checkSender(msg.sender, _sender);
10      require(_dstChainId == uint32(block.chainid), mErc20Host_DstChainNotValid());
11      require(_market == address(this), mErc20Host_AddressNotValid());
12      require(allowedChains[_chainId], mErc20Host_ChainNotValid());
13      }
```

Snippet 4.2: Snippet of `_mintExternal()` from `mErc20Host.sol`

The issue is that the data committed to the journal is taken entirely from the output of the view call. This allows anyone to prove `mToken` balance changes by calling a spoofed contract which implements the same function selector, and returns arbitrary values for the journal data. This is possible because the validations performed in the guest do not verify that the balance queries have been performed on valid `mToken` addresses.

Impact Anyone can create false proofs of balance changes for their desired destination and account address. This allows someone to directly steal funds from the protocol.

Recommendation Environment specific information committed to the journal data should be read from the zkVmInput instead of the view call output. This includes the `asset`, `account` and the `chain_id`.

Also, consider validating the `mToken` addresses on which the queries are performed within the guest itself.

Developer Response `getProofData()` now only returns the input/output amounts. The user, market address, and chain ID input by the host (and used by the ZK proof) are now written directly to the journal.

4.1.2 V-MLDZK-VUL-002: Balance queries performed on incorrect addresses due to padding

Severity	High	**Commit**	5a57051
Type	Logic Error	**Status**	Fixed
File(s)	`./methods/guest/guest_utils/src/validators.rs`, `./malda_rs/src/viewcalls.rs`		
Location(s)	validate_balance_of_call(), get_balance_call_input()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-zkcoprocessor/pull/9`, 15f0acc		

In `validate_balance_of_call`, the call data for performing balance queries is constructed by concatenating together the function selector and the encoded address on which the query is to be performed. See snippet below for context.

```
// Create function selector for balanceOf(address)
let selector = [0x70, 0xa0, 0x82, 0x31]; // keccak256("balanceOf(address)")[:4]
let account_bytes: [u8; 32] = account.into_word().into();

// Create calldata by concatenating selector and encoded address
let mut call_data = Vec::with_capacity(36); // 4 bytes selector + 32 bytes address
call_data.extend_from_slice(&selector);
call_data.extend_from_slice(&[0u8; 12]); // pad address to 32 bytes
call_data.extend_from_slice(&account_bytes);

calls.push(Call3 {
    target: *asset,
    allowFailure: false,
    callData: call_data.into(),
});
```

Snippet 4.3: Snippet from `validate_balance_of_call()`

When concatenating, an extra padding of `12` bytes is added to the `call_data` to pad the account address to `32` bytes-the EVM word size. This is not required because, when the account address is converted to bytes using `account.into_word()`, it is already left padded to `32` bytes.

This results in balance queries being made on incorrect account addresses because of the extra `12` bytes of padding. A proof of concept validating the issue can be found here.

Impact The balance queries are performed on incorrect account addresses because of the extra padding. Users will not be able to leverage the cross chain capabilities of the protocol, because the balance queries will return `0`.

Recommendation Remove the additional `12` bytes of padding.

Developer Response The call-data is no longer padded.

4.1.3 V-MLDZK-VUL-003: Hard-Coded Sequencers

Severity	Low	**Commit**	5a57051
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	15f0acc		

Within the malda-zkcoprocessor circuit, the sequencer addresses are hard-coded.

```
/// The address of the Optimism sequencer contract.
pub const OPTIMISM_SEQUENCER: Address = address!("
    AAAA45d9549EDA09E70937013520214382Ffc4A2");
/// The address of the Base sequencer contract.
pub const BASE_SEQUENCER: Address = address!("
    Af6E19BE0F9cE7f8afd49a1824851023A8249e8a");
/// The address of the Linea sequencer contract.
pub const LINEA_SEQUENCER: Address = address!("8
    f81e2e3f8b46467523463835f965ffe476e1c9e");
```

Snippet 4.4: Snippet from `methods/guest/guest_utils/src/constants.rs`

In various portions of the protocol, these hard-coded addresses, along with their testnet counterparts are used. For example, when validating the environment of a block claimed to reside in an OP-stack chain, this `match` statement chooses the appropriate sequencer address to validate against.

```
 match chain_id {
    OPTIMISM_CHAIN_ID => commitment
        .verify(OPTIMISM_SEQUENCER, OPTIMISM_CHAIN_ID)
        .expect("Failed to verify Optimism sequencer commitment"),
    BASE_CHAIN_ID => commitment
        .verify(BASE_SEQUENCER, BASE_CHAIN_ID)
        .expect("Failed to verify Base sequencer commitment"),
    OPTIMISM_SEPOLIA_CHAIN_ID => commitment
        .verify(OPTIMISM_SEPOLIA_SEQUENCER, OPTIMISM_SEPOLIA_CHAIN_ID)
        .expect("Failed to verify Optimism Sepolia sequencer commitment"),
    BASE_SEPOLIA_CHAIN_ID => commitment
        .verify(BASE_SEPOLIA_SEQUENCER, BASE_SEPOLIA_CHAIN_ID)
        .expect("Failed to verify Base Sepolia sequencer commitment"),
    _ => panic!("invalid chain id"),
}
```

Snippet 4.5: Snippet from `validate_opstack_env()`.

As a consequence, if any of the keys corresponding to these addresses is compromised, then the entire protocol will be compromised. For example, if a testnet key for `OPTIMISM_SEPOLIA` is leaked, then the private key for the testnet may be used by an attacker to sign blobs with the *incorrect* chain ID. This will allow an attacker to execute a view call within a private fork that has, e.g., the `OPTIMISM_CHAIN_ID` chain ID, but sign it with the `OPTIMISM_SEPOLIA_CHAIN_ID` to ensure verification passes.

Impact Key compromise of any of the supported networks *or test networks* may be used to steal

funds from the protocol. This will be the case even if the network is not one of the `allowedChains` of the on-chain protocol, since a compromised sequencer may lie about its chain ID.

Additionally, in the case of a key compromise, the circuit would need to be recompiled and deployed with a new image ID. This increases the possibility of errors or mistakes.

Recommendation Include the sequencer key and chain ID (or a commitment to them) in the public input, so that on-chain programs may whitelist trusted sequencers.

Developer Response The developers changed the method so that

1. The `chain_id` passed from the host ID determines which sequencer to use.
2. That *same* `chain_id` is written directly to the journal.

This prevents any sequencer from lying about its own chain ID.

4.1.4 V-MLDZK-VUL-004: Optimistic update used in light client

Severity	Low	**Commit**	5a57051
Type	Data Validation	**Status**	Acknowledged
File(s)	`methods/guest/guest_utils/src/validators_ethereum_light_client.rs`		
Location(s)	build_beacon_chain()		
Confirmed Fix At	N/A		

The light client implementation uses the latest optimistic header, which can be reorged.

```
let latest_beacon_root = self.store.optimistic_header.tree_hash_root();
Ok(B256::new(latest_beacon_root.0))
```

Snippet 4.6: Final statements in `L1ChainBuilder::build_beacon_chain()`

Impact L1 reorganizations may allow double-spends, draining the protocol.

Recommendation Output the number of valid updates built on top of the block. Limit transfer sizes to be within the protocol's tolerance for a potential reorg.

Developer Response We acknowledge latest optimistic header is not safe against reorg. Reorg protection depth will be set to an appropriate value per chain or volume limits depending on chain length.

4.1.5 V-MLDZK-VUL-005: balanceOf used in place of getProofData

Severity	Warning	**Commit**	5a57051
Type	Logic Error	**Status**	Fixed
File(s)	`methods/guest/guest_utils/src/validators.rs`, `methods/guest/guest_utils/src/validators_ethereum_light_client.rs`		
Location(s)	validate_balance_of_call()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-zkcoprocessor/pull/9`, `https://github.com/malda-protocol/malda-lending/pull/51/`, 15f0acc, dfb60d8		

At the beginning of the audit, the developers reported that they used `balanceOf` in place of `getProofData`, but are planning to swap out the view calls during the review.

Impact Using `balanceOf`, the contracts will not operate correctly, and will be vulnerable to non-monotonic changes in balance which are not possible with `getProofData`.

Recommendation Swap out the view calls to use the intended function. Update the `Journal` struct in `validators_ethereum_light_client`. Add end-to-end testing locally validating the project works as expected on a local devnet.

Developer Response The selectors have been switched, and the journal is properly written to now based on the results of `getProofData()`. See also comments on V-MLDZK-VUL-001.

4.1.6 V-MLDZK-VUL-006: Unused/duplicate program constructs

Severity	Warning	**Commit**	5a57051
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	44225b3		

Description The following program constructs are duplicated or unnecessary:

1. `methods/guest/guest_utils/src/cryptography.rs`:
    a) `public_key_to_address()`: This method is copied from Alloy (`https://docs.rs/alloy-primitives/latest/alloy_primitives/struct.Address.html#method.from_public_key`).

Impact These constructs may not benefit from updates to Alloy, leading to errors if used in the future.

Developer Response The developers now use Alloy's function directly.

4.1.7 V-MLDZK-VUL-007: Light client circuit may accept future blocks

Severity	Warning	**Commit**	5a57051
Type	Data Validation	**Status**	Fixed
File(s)	`methods/guest/guest_utils/src/validators_ethereum_light_client.rs`		
Location(s)	L1ChainBuilder::verify_update()		
Confirmed Fix At	8a512cc		

Helios' `verify_update` function (see here) corresponds to the ethereum light client sync protocol's `validate_light_client_update` (see here). The `current_slot` argument (set to `update.signature_slot` in the below snippet) is intended to invalidate blocks which are from the "future".

```
pub fn verify_update(&self, update: &Update) -> Result<()> {
    verify_update(
        update,
        update.signature_slot,
        &self.store,
        OldB256::from(self.genesis_root.0),
        &self.forks,
    )
}
```

Snippet 4.7: Definition of `L1ChainBuilder::verify_update()`

Impact Invalid blocks may be accepted, causing the Malda protocol's light client to diverge from other Ethereum light clients.

Recommendation Output the slot on the latest update, and check on-chain that it is not from a future slot.

Developer Response The developers now output the slot of the last finalized header into the journal.

4.1.8 V-MLDZK-VUL-008: Helios 0.6.0 missing checks

Severity	Warning	**Commit**	5a57051
Type	Data Validation	**Status**	Fixed
File(s)	`methods/guest/guest_utils/src/validators_ethereum_light_client.rs`		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/malda-protocol/malda-zkcoprocessor/pull/9`, 5acf6de		

The light client protocol in `validators_ethereum_light_client.rs` relies on Helios, currently set to version 0.6.0, in order to actually implement the light protocol logic.

Before Helios 0.7.0, Helios does not include a call to `is_valid_header()` inside of `verify_bootstrap()`(see this line), which corresponds to the below line in validate_light_client_update from the consensus specs. Other calls to `is_valid_header` were also missing in 0.6.0, and were added in 0.7.0.

Additionally, Helios is missing the below check when the `update_finalized_slot` is the `GENESIS_SLOT`.

```
1 assert update.finalized_header == LightClientHeader()
```

Impact As in V-MLDZK-VUL-007, this requires the sync committee to sign an invalid block. Nonetheless, in emergency scenarios it has the possibility of causing a divergence between Malda's view of the blockchain and the canonical one.

Recommendation Upgrade to Helios 0.7.0.

Since Malda is checkpointing after genesis, the final missing check should never be necessary.

Developer Response The recommendation was implemented.

4.1.9 V-MLDZK-VUL-009: Hardcoded RPC with API keys

Severity	Warning	**Commit**	5a57051
Type	Information Leakage	**Status**	Fixed
File(s)	`malda_rs/src/constants.rs`		
Location(s)	See description		
Confirmed Fix At	`https://github.com/malda-protocol/malda-zkcoprocessor/pull/9`, 439d8ce		

URLs for RPC calls are leaking API keys which may be abused or spammed. See snippet below for an example.

```
pub const RPC_URL_LINEA: &str =
    "https://linea-mainnet.g.alchemy.com/v2/...";
```

Impact The leakage of API keys may lead to rate limiting, increased costs, or even denial-of-service if these need to be blocked. Changing RPCs or API keys would lead to a new ELF executable and require an update.

Recommendation RPC URLs and API keys should be configurable and should be read from environment variables.

Developer Response The developers now read the RPC URLs from environment variables.

Updated Veridise Response The Veridise analysts confirmed with the developers that the git history would not be published.

4.1.10 V-MLDZK-VUL-010: Signature may be malleable

Severity	Warning	**Commit**	5a57051
Type	Data Validation	**Status**	Fixed
File(s)	`methods/guest_utils/src/cryptography.rs`		
Location(s)	signature_from_bytes()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-zkcoprocessor/pull/9`, 32ed548		

The function comment for `signature_from_bytes()` indicates that the function should panic if the byte slice is not exactly 65 bytes. However, it will only panic if the length of the byte slice is *less than* 65 bytes.

Additionally, the function will parse any `v` byte as valid, rather than only accepting the bytes 0 and 1 (see equation 314 in Appendix F of the Ethereum yellowpaper).

```
/// This function will panic if the input byte slice is not exactly 65 bytes long or
    if the ‘r‘ or ‘s‘ components
/// cannot be parsed into ‘U256‘ values.
pub fn signature_from_bytes(signature: &Bytes) -> Signature
    let r_array: [u8; 32] = signature
        .slice(0..32)
        .to_vec()
        .try_into()
        .expect("Failed to convert first 32 bytes into r component array");
    let r = U256::from_be_bytes(r_array);

    let s_array: [u8; 32] = signature
        .slice(32..64)
        .to_vec()
        .try_into()
        .expect("Failed to convert second 32 bytes into s component array");
    let s = U256::from_be_bytes(s_array);

    let v_array: [u8; 1] = signature
        .slice(64..65)
        .to_vec()
        .try_into()
        .expect("Failed to convert last byte into v component array");
    let v = v_array[0] == 1;

    Signature::new(U256::from(r), U256::from(s), v)
}
```

Snippet 4.8: Definition of `signature_from_bytes()`

Impact Signatures with random extra bytes or invalid `v` values may be accepted. This does not affect the current protocol. However, any third party who relies on signature non-malleability to prevent replays may be made unexpectedly vulnerable.

Recommendation Check that the signature length is 65.

Developer Response The developers implemented the recommendation.

4.1.11 V-MLDZK-VUL-011: Typos and Incorrect comments

Severity	Info	**Commit**	5a57051
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	0532d68		

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

1. `methods/guest/guest_utils/src/types.rs`:
    a) `struct Withdrawal`: The struct doc comment does not reference Alloy, which it appears to be copied from (`https://docs.rs/alloy/latest/alloy/eips/eip4895/struct.Withdrawal.html`).

Impact These minor errors may lead to future developer confusion.

Developer Response The developers added a comment which makes a reference to Alloy in the concerned location.

4.1.12 V-MLDZK-VUL-012: Best practices

Severity	Info	**Commit**	5a57051
Type	Maintainability	**Status**	Fixed
File(s)	`malda_rs/src/viewcalls.rs`		
Location(s)	get_user_balance_zkvm_input()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-zkcoprocessor/pull/9/`, c23b932		

The following shows code extracted from `get_user_balance_zkvm_input()`. The `else` statement is reachable exactly when the `chain_id` is `LINEA_CHAIN_ID` or `LINEA_SEPOLIA_CHAIN_ID`. However, this is not validated to be the case.

```
let rpc_url = match chain_id {
        BASE_CHAIN_ID => RPC_URL_BASE,
        OPTIMISM_CHAIN_ID => RPC_URL_OPTIMISM,
        LINEA_CHAIN_ID => RPC_URL_LINEA,
        ETHEREUM_CHAIN_ID => RPC_URL_ETHEREUM,
        OPTIMISM_SEPOLIA_CHAIN_ID => RPC_URL_OPTIMISM_SEPOLIA,
        BASE_SEPOLIA_CHAIN_ID => RPC_URL_BASE_SEPOLIA,
        LINEA_SEPOLIA_CHAIN_ID => RPC_URL_LINEA_SEPOLIA,
        ETHEREUM_SEPOLIA_CHAIN_ID => RPC_URL_ETHEREUM_SEPOLIA,
        _ => panic!("Invalid chain ID"),
    };

    let (block, commitment) = if chain_id == OPTIMISM_CHAIN_ID
        || chain_id == BASE_CHAIN_ID
        || chain_id == ETHEREUM_CHAIN_ID
        || chain_id == OPTIMISM_SEPOLIA_CHAIN_ID
        || chain_id == BASE_SEPOLIA_CHAIN_ID
        || chain_id == ETHEREUM_SEPOLIA_CHAIN_ID
    {
       ...
    } else {
       ...
    }
```

Snippet 4.9: Snippet from `get_user_balance_zkvm_input()`

Impact This may lead to higher maintainability costs and cause potential issues when more chains are supported.

Recommendation Use a match pattern or enumerate all expected options in `else if` branches, and panic on unexpected entries, as done elsewhere in the codebase.

Developer Response The developers implemented the recommendation.

4.1.13 V-MLDZK-VUL-013: Execution blocks assumed to be V3

Severity	Info	**Commit**	5a57051
Type	Usability Issue	**Status**	Acknowledged
File(s)	`methods/guest/guest_utils/src/types.rs`		
Location(s)	<ExecutionPayload as TryFrom<&SequencerCommitment»::try_from()		
Confirmed Fix At	N/A		

The current implementation of `<ExecutionPayload as TryFrom<&SequencerCommitment>>::try_from()` assumes the block was listed on the V3 topic, in which the execution payload starts at byte 32. The V3 topic has the blocks since the ecotone upgrade, but the V4 topic will start being used when the Isthmus upgrade occurs, which (based on prior cadence and this discussion) seems to be likely in Q2 of 2025

Once blocks start arriving on the V4 topic, this will no longer function correctly, as the V4 payload includes the withdrawals root.

Additionally, if a block came on the V1/V2 topics, the first 32 bytes will be ignored.

```
1 fn try_from(value: &SequencerCommitment) -> Result<Self> {
2       let payload_bytes = &value.data[32..];
3       ExecutionPayload::from_ssz_bytes(payload_bytes).map_err(|_| eyre::eyre!("decode
      failed"))
4   }
```

Snippet 4.10: Definition of `<ExecutionPayload as TryFrom<&SequencerCommitment>>::try_from()`

Impact This circuit may not support upcoming network upgrades. Historical blocks delivered on the V1/V2 topics may have their first 32 bytes dropped. In some rare cases, this may allow someone to parse an invalid block and lie about the execution environment.

Recommendation Validate that the block number is after the ecotone upgrade for Optimism/Base blocks.

Prepare an upgrade to be released in tandem with the next Optimism upgrade, after which point a different parsing scheme should be used based on the block number. Consider pausing the protocol during the Isthmus upgrade to ensure there are no errors during the network upgrade.

Developer Response Acknowledged. We will follow the suggestions and closely monitor Opttimism upgrade to V4 topics and have a corresponding upgrade to our program ready.

4.1.14 V-MLDZK-VUL-014: The OP Stack environment validation will fail for Ethereum Sepolia

Severity	Info	**Commit**	5a57051
Type	Logic Error	**Status**	Fixed
File(s)	`validators.rs`		
Location(s)	get_ethereum_block_hash_via_opstack()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-zkcoprocessor/pull/9`, 5643803		

In `validators.rs`, the method `get_ethereum_block_hash_via_opstack()` uses Optimism's `L1Block` contract to fetch the L1 block hash. See snippet below for details.

```
pub fn get_ethereum_block_hash_via_opstack(
    commitment: SequencerCommitment,
    input_op: EthEvmInput,
) -> B256 {
  let env_op = input_op.into_env();
  validate_opstack_env(
      OPTIMISM_CHAIN_ID, &commitment, env_op.commitment().digest
  );
  let l1_block = Contract::new(L1_BLOCK_ADDRESS_OPTIMISM, &env_op);
  let call = IL1Block::hashCall {};
  l1_block.call_builder(&call).call()._0
}
```

Snippet 4.11: Snippet from `get_ethereum_block_hash_via_opstack()`

Currently, `get_ethereum_block_hash_via_opstack()` is supported for both `ETHEREUM_CHAIN_ID` and `ETHEREUM_SEPOLIA_CHAIN_ID`. But while validating the OP Stack environment through `validate_opstack_env()`, it uses only the `OPTIMISM_CHAIN_ID`.

As can be seen in the snippet below, the validation will not be performed against the correct sequencer for `ETHEREUM_SEPOLIA_CHAIN_ID`, which is `OPTIMISM_SEPOLIA_SEQUENCER`. Therefore, the OP Stack environment validation for `ETHEREUM_SEPOLIA_CHAIN_ID` will fail.

```
pub fn validate_opstack_env(
    chain_id: u64, commitment: &SequencerCommitment, env_block_hash: B256
) {
    match chain_id {
      OPTIMISM_CHAIN_ID => commitment
          .verify(OPTIMISM_SEQUENCER, OPTIMISM_CHAIN_ID)
          .expect("Failed to verify Optimism sequencer commitment"),
      BASE_CHAIN_ID => commitment
          .verify(BASE_SEQUENCER, BASE_CHAIN_ID)
          .expect("Failed to verify Base sequencer commitment"),
      OPTIMISM_SEPOLIA_CHAIN_ID => commitment
          .verify(OPTIMISM_SEPOLIA_SEQUENCER, OPTIMISM_SEPOLIA_CHAIN_ID)
          .expect("Failed to verify Optimism Sepolia sequencer commitment"),
      BASE_SEPOLIA_CHAIN_ID => commitment
          .verify(BASE_SEPOLIA_SEQUENCER, BASE_SEPOLIA_CHAIN_ID)
          .expect("Failed to verify Base Sepolia sequencer commitment"),
```

```
      _ => panic!("invalid chain id"),
    }
    let payload = ExecutionPayload::try_from(commitment)
      .expect("Failed to convert sequencer commitment to execution payload");
    assert_eq!(payload.block_hash, env_block_hash, "block hash mismatch");
}
```

Snippet 4.12: Snippet from `validate_opstack_env()`

Impact Since the validation of the environment will fail internally, execution of the balance queries on `ETHEREUM_SEPOLIA_CHAIN_ID` will always fail.

Recommendation Within `get_ethereum_block_hash_via_opstack()`, extract the `chain_id` from the environment configuration and pass that to `validate_opstack_env()`. This will ensure that the environment is validated against the intended sequencer.

Developer Response The developers extended `get_ethereum_block_hash_via_opstack()` with `chain_id`. It is then used to make a distinction between the `OPTIMISM_CHAIN_ID` and `OPTIMISM_SEPOLIA_CHAIN_ID`.

5 Vulnerability Report: L1 Inclusion

This section presents the vulnerabilities found during the second security assessment of `malda_utils/` on commit `2095dda1`. 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-MLDZK2-VUL-001	OP L1 path accepts fraudulent state	Critical	Fixed
V-MLDZK2-VUL-002	Wrong chain ID used	Critical	Fixed
V-MLDZK2-VUL-003	Incorrect inequality used for L1 inclusion	Medium	Partially Fixed
V-MLDZK2-VUL-004	Linea slow path checks wrong state root	Medium	Fixed
V-MLDZK2-VUL-005	Sequencer commitment not verified in the . . .	Warning	Acknowledged
V-MLDZK2-VUL-006	Unused rust code	Warning	Fixed
V-MLDZK2-VUL-007	Fraudulent games may DoS the slow lane	Warning	Fixed
V-MLDZK2-VUL-008	Missing comments	Info	Fixed
V-MLDZK2-VUL-009	Compilation issues	Info	Fixed

5.1 Detailed Description of Issues

5.1.1 V-MLDZK2-VUL-001: OP L1 path accepts fraudulent state

Severity	Critical	Commit	2095dda
Type	Logic Error	Status	Fixed
File(s)	`malda_utils/src/validators.rs`		
Location(s)	validate_opstack_env_with_l1_inclusion()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-zk-coprocessor/pull/5`		

As shown in the below snippet, the function `validate_opstack_env_with_l1_inclusion()` attempts to validate an OpStack L2 block by checking that:

1. The `rootClaim` from the dispute game matches the expected `OutputRootProof`
2. The L2 block number has not been challenged
3. A fixed time delay (`TIME_DELAY_OP_CHALLENGE`) has passed since the dispute game's creation

This logic assumes that the absence of a challenge to the L2 block number implies validity, which is not necessarily true. Disputing the L2 block number is only possible if a dispute game is created with a claimed L2 number that does not match the pre-image of the root claim. This means that, so long as a dispute game does not lie about its claimed L2 block number, even a fraudulent root claim cannot have its L2 block number challenged.

Moreover, checking only that `TIME_DELAY_OP_CHALLENGE` seconds have passed since the game started does not guarantee that the game has resolved. The game may still be in progress or unresolved due to inactivity.

```
let l2_block_number_challenged_call = IDisputeGame::l2BlockNumberChallengedCall {};

let contract = Contract::new(game_address, &env_eth);
let returns = contract
    .call_builder(&l2_block_number_challenged_call)
    .call();

let l2_block_number_challenged = returns._0;

let output_root_proof = OutputRootProof {
    version: ROOT_VERSION_OPSTACK,
    stateRoot: op_state_root,
    messagePasserStorageRoot: msg_passer_storage_hash,
    latestBlockhash: op_block_hash,
};

let output_root_proof_hash = keccak256(output_root_proof.abi_encode());

// assert_eq!(l2_block_number, 1, "block number mismatch");
assert_eq!(root_claim, output_root_proof_hash, "root claim mismatch");
assert_eq!(
    l2_block_number_challenged, false,
```

```
    "This L2 block has been challenged"
);
assert!(
    U256::from(env_eth_timestamp) > created_at + U256::from(TIME_DELAY_OP_CHALLENGE),
    "Not enough time passed to challenge the claim"
);
```

Snippet 5.1: Snippet from `validate_opstack_env_with_l1_inclusion()`

Impact An attacker could supply a fraudulent L2 block in a dispute game. Even if this game is successfully challenged, `validate_opstack_env_with_l1_inclusion()` will accept the block's root claim as true so long as the L2 block number in its preimage matches the dispute game's claimed block number. This could allow an attacker to forge proofs and mint unbacked mTokens on the host chain.

Additionally, by relying on a delay (`TIME_DELAY_OP_CHALLENGE`, currently set to 5 minutes) rather than checking whether the game has fully resolved, this logic may validate blocks that are still under dispute.

Recommendation Update `validate_opstack_env_with_l1_inclusion()` to ensure that:

1. The game has fully resolved, ideally by checking `FaultDisputeGame.resolvedAt()` or an equivalent finality condition
2. The `AnchorStateRegistry` has determined the claim is valid

As recommended in V-MLDZK2-VUL-005, the analysts recommend still checking the Optimism sequencer signature in the slow lane. Bypassing this check via the slow lane makes this a viable attack even when the sequencer is behaving as expected.

Developer Response The developers instead perform full fault dispute game validation. More precisely, the various verifications performed are broken down as follows:

- OP Stack
  - L1 inclusion: The circuit requires that there be a chain of Ethereum blocks of at least the provided reorg depth, whose last block is attested to by OP-Stack L2's sequencer. The environment of the first block is then used to validate the OP-stack game has a valid claim.
  - No L1 inclusion: As before, the circuit requires that there be a chain of OP-Stack L2 blocks of at least the provided reorg depth, whose last block is attested to by the OP-Stack L2 sequencer.
- Linea: In either case, the circuit requires that there be a chain of L2 blocks of at least the provided reorg depth, whose last block is attested to by the Linea L2 sequencer.
  - L1 inclusion: When L1 inclusion is required, the circuit requires the prover to produce an Ethereum block attested to by the Optimism sequencer. This Ethereum block is then used to check the current Linea block number (as reported by Linea) on the L1. The circuit requires the claimed Linea block's block number be at most the current Linea block number on the L1.

- Ethereum: This circuit is unchanged, and requires a chain of L1 blocks of at least the reorg depth whose last block is attested to by Optimism's sequencer.

5.1.2 V-MLDZK2-VUL-002: Wrong chain ID used

Severity	Critical	**Commit**	2095dda
Type	Logic Error	**Status**	Fixed
File(s)	`malda_utils/src/validators.rs`		
Location(s)	get_validated_block_hash_linea(), get_validated_block_hash_opstack()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-zk-coprocessor/pull/4`		

The `get_validated_block_hash_linea()` and `get_validated_block_hash_opstack()` functions are responsible for validating a block hash is present on the Linea and Optimism chains, respectively. When `validate_l1_inclusion` is set to `true`, this is intended to be done by checking the L2 block data has landed on the L1. The L1 data is attested to by the Optimism sequencer, through the L1Block contract's hash attribute. This process of extracting an Ethereum block from Optimism expects the `chain_id` to be either Ethereum, or Sepolia.

```
pub fn get_ethereum_block_hash_via_opstack(
    commitment: SequencerCommitment,
    input_op: EthEvmInput,
    chain_id: u64,
) -> B256 {
    let env_op = input_op.into_env();
    let verify_via_chain = if chain_id == ETHEREUM_CHAIN_ID {
        OPTIMISM_CHAIN_ID
    } else {
        OPTIMISM_SEPOLIA_CHAIN_ID
    };
```

Snippet 5.2: Snippet from `get_ethereum_block_hash_via_opstack()`

However, in both `get_validated_block_hash_linea()` and `get_validated_block_hash_opstack()`, the `chain_id` used is a Linea or Optimism chain id. This means that the returned `ethereum_hash` block will always be the hash of a block from the Ethereum *testnet*. In the Linea case, this is likely to cause a denial of service. However, as shown below, this allows an attacker to bypass the sequencer signature check for OP-chains implemented in `validate_opstack_env()`.

```
pub fn get_validated_block_hash_opstack(
    chain_id: u64,
    // ...
) -> B256 {
    let last_block_hash = last_block.hash_slow();
    if validate_l1_inclusion {
        let env_state_root = env_header.state_root;
        let ethereum_hash = get_ethereum_block_hash_via_opstack(
            sequencer_commitment.unwrap(),
            env_op_input.unwrap(),
            chain_id,
        );
        validate_opstack_env_with_l1_inclusion(
            chain_id,
            env_state_root,
```

```
            env_eth_input.unwrap(),
            storage_hash.unwrap(),
            ethereum_hash,
            last_block_hash,
        );
    } else {
        validate_opstack_env(chain_id, &sequencer_commitment.unwrap(),
    last_block_hash);
    }
    last_block_hash
}
```

Snippet 5.3: The `validate_l1_inclusion` case of `get_validated_block_hash_opstack()`

Impact An attacker may submit a fraud dispute game on the Sepolia testnet, attesting to a false view of the Optimism blockchain. This may then be used to produce proofs which are redeemable for funds on the host chain.

L1-inclusion proofs may not be possible for the Linea chain, forcing users to rely on the Malda sequencer for access to their funds on extension chains.

Note that this impact is further magnified by the problems described in V-MLDZK2-VUL-001.

Recommendation As mentioned in V-MLDZK2-VUL-005, the OP sequencer signature should still be validated. This ensures the `l1_inclusion` path is at least as strict as the fast path.

Separate chain IDs should be used for the calls to `get_ethereum_block_hash_via_opstack()` in both the Linea and Optimism cases. Consider using Rust enums to enforce this statically, preventing future errors of this type.

Developer Response The developers now use the correct Ethereum mainnet or testnet chain ID based on the OP-chain or Linea chain ID provided.

Updated Veridise Response This does resolve the issue with the chain ID. The OP sequencer signature is still not validated. See the developer responses in V-MLDZK2-VUL-001 and V-MLDZK2-VUL-005 for more details.

5.1.3 V-MLDZK2-VUL-003: Incorrect inequality used for L1 inclusion

Severity	Medium	**Commit**	2095dda
Type	Logic Error	**Status**	Partially Fixed
File(s)	`./malda_utils/src/validators.rs`		
Location(s)	validate_linea_env_with_l1_inclusion()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-zk-coprocessor/pull/3/files`		

To ensure Malda users can custody their own funds without introducing risks to the protocol, Malda's "slow lane" allows them to perform cross-chain actions once the relevant L2 block has been included in the L1. The method `validate_linea_env_with_l1_inclusion()` validates L1-inclusion for Linea blocks.

The function starts by fetching the latest L2 block posted on L1 through the `L1MessageService` contract. Then, it compares the latest L2 block number to `env_block_number`-the claimed L2 block's number. However, the inequality within the assert is implemented incorrectly. `env_block_number` is required to be greater than the latest L2 included on L1, effectively performing an L1 exclusion check.

```
pub fn validate_linea_env_with_l1_inclusion(
  chain_id: u64,
  env_block_number: u64,
  env_eth_input: EthEvmInput,
  ethereum_hash: B256,
) {
    /// [Veridise] elided...
  let current_l2_block_number_call = IL1MessageService::currentL2BlockNumberCall {};
  let contract = Contract::new(msg_service_address, &env_eth);
  let returns = contract.call_builder(&current_l2_block_number_call).call();

  let l2_block_number = returns._0;

  assert!(
      l2_block_number <= U256::from(env_block_number),
      "Block number must be lower than the last one posted to L1"
  );
}
```

Snippet 5.4: Snippet from `validate_linea_env_with_l1_inclusion()`

Impact The L1 inclusion path is intended to be cross-chain safe. Even an attacker who is able to exploit a sequencer bug which causes a re-org on the unsafe-head should not be able to generate fraudulent proofs. Unfortunately, since valid L1-inclusion proofs can be generated for L2 blocks which have not yet been included on the L1, an attacker may still take advantage of sequencer bugs. Conversely, since L2 blocks which have been included on the L1 will fail this check, the zk-coprocessor will fail to generate proofs for valid L2 blocks. This may prevent users from spending their cross-chain funds without permission from the Malda sequencer.

Finally, note that comparing the block number alone is insufficient. A re-orged block's block number will eventually be surpassed by the finalized chain's block number. Consequently, the

protection against reorgs offered by the corrected check is limited.

Recommendation Check that the claimed Linea block is included in the finalized state, e.g. using the current finalized state.

Developer Response The developers reversed the inequality.

Updated Veridise Response This does address the inequality. The Veridise team would further recommend to rely on a state root taken from the L1 rather than the block number alone. This path is intended to offer reorg protection, but a reorged block's number will eventually be surpassed by the canonical chain.

Updated Developer Response The posted Linea state root is not the standard EVM state root, and would require extensive changes to use.

5.1.4 V-MLDZK2-VUL-004: OP-Stack slow path checks wrong state root

Severity	Medium	**Commit**	2095dda
Type	Logic Error	**Status**	Fixed
File(s)	malda_utils/src/validators.rs		
Location(s)	get_validated_block_hash_opstack()		
Confirmed Fix At	https://github.com/malda-protocol/malda-zk-coprocessor/pull/5		

The `validate_get_proof_data_call()` function attempts to validate that a certain block header (the `env_header`) is the ancestor of some block (called the `last_block`) which is known to be in a supported chain. This is done in two stages:

1. First, the `last_block` is validated.
2. Second, `validate_get_proof_data_call()` invokes `validate_chain_length()` to check that the prover knows a sufficiently long chain of blocks from the `env_header` to the `last_block`.

If L1 inclusion is required for a block on an OP chain, the `get_validated_block_hash_opstack()` function is responsible for validating the last block has landed on the L1. This uses `last_block_hash` for the `op_block_hash`, but uses the `env_header`'s `state_root` as `op_state_root`. This requires that the `env_header` and `last_block` are the same length, preventing longer linking chains.

```
let env_state_root = env_header.state_root;
let ethereum_hash = get_ethereum_block_hash_via_opstack(
    sequencer_commitment.unwrap(),
    env_op_input.unwrap(),
    chain_id,
);
validate_opstack_env_with_l1_inclusion(
    chain_id,
    env_state_root,
    env_eth_input.unwrap(),
    storage_hash.unwrap(),
    ethereum_hash,
    last_block_hash,
);
```

Snippet 5.5: `validate_l1_inclusion == true` case of `get_validated_block_hash_opstack()`.

```
pub fn validate_opstack_env_with_l1_inclusion(
    chain_id: u64,
    op_state_root: B256,
    env_eth_input: EthEvmInput,
    msg_passer_storage_hash: B256,
    ethereum_hash: B256,
    op_block_hash: B256,
)
```

Snippet 5.6: Signature of `validate_opstack_env_with_l1_inclusion()`.

Impact No L1-inclusion proofs for the optimism blockchain can be made, since the minimum chain length is two. This means that users will be reliant on the Malda sequencer to access their funds.

Recommendation Validate the `last_block`'s state root is on the L1, rather than the `env_header`'s state root.

Developer Response The developers now implement a new function `sort_and_verify_relevant_params()` which sorts and correctly verifies relevant parameters for the proof data validation.

For L1 inclusion on Linea, it is validated that there is chain length of `reorg_protection_depth` on top of the block and that the L2 block number does not exceed the Linea block number recorded on L1. For OP-stack chains for the L1 inclusion flow, re-org depth check is done on L1 blocks as OPStack with L1 inclusion flow is only reading L1 state.

5.1.5 V-MLDZK2-VUL-005: Sequencer commitment not verified in the slow lane

Severity	Warning	**Commit**	2095dda
Type	Data Validation	**Status**	Fixed
File(s)	`./malda_utils/src/validators.rs`		
Location(s)	get_validated_block_hash_opstack()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-zk-coprocessor/pull/5`		

The method `get_validated_block_hash_opstack()` conditionally executes `validate_opstack_env_with_l1_inclusion()` or `validate_opstack_env()` depending on the value of `validate_l1_inclusion`. Note that in the slow lane, while L1 inclusion of the L2 block is verified, it is not verified that the sequencer has signed the last block. This can be seen in the snippet below.

```
pub fn get_validated_block_hash_opstack(
    // [Veridise] ..elided
) -> B256 {
  let last_block_hash = last_block.hash_slow();
  if validate_l1_inclusion {
      let env_state_root = env_header.state_root;
      let ethereum_hash = get_ethereum_block_hash_via_opstack(
          sequencer_commitment.unwrap(),
          env_op_input.unwrap(),
          chain_id,
      );
      validate_opstack_env_with_l1_inclusion(
          chain_id,
          env_state_root,
          env_eth_input.unwrap(),
          storage_hash.unwrap(),
          ethereum_hash,
          last_block_hash,
      );
  } else {
      validate_opstack_env(chain_id, &sequencer_commitment.unwrap(), last_block_hash)
    ;
  }
  last_block_hash
}
```

Snippet 5.7: Snippet from `get_validated_block_hash_opstack()`

It is recommended to still validate within the slow lane that the sequencer signed the last block, to ensure the introduction of a slow lane is guaranteed to be at least as secure as the fast lane.

Impact Since the slow lane verifies the OP-Stack L2 block inclusion in L1 through dispute game verification, the claim cannot be considered valid until it is successfully resolved. Validating that the sequencer has signed the block can provide a stronger security guarantee, and ensure that the only potential vulnerabilities introduced by the slow lane are denial-of-service related.

See related issues V-MLDZK2-VUL-001 and V-MLDZK2-VUL-002, which take advantage of the bypassed sequencer check to escalate issues into theft.

Recommendation Validate the sequencer signed the last block irrespective of `validate_l1_inclusion` status.

Developer Response We use op-steel to get the environment for the latest finalized dispute game on L1, then get L1 state via Opstack and confirm this env corresponds to the one posted on L1 and that the game's claim is valid.

As this confirmation purely works on L1 state (by getting the L2 state posted there), we do not check any OP sequencer commitment on the state on which the viewcall is done, but only in order to receive the current L1 state. We would have to check the sequencer signature on a 7 days old block - since this is not available via standard-rpc it would defy the purpose of the slow lane for self-sequencing for censorship resistance.

5.1.6 V-MLDZK2-VUL-006: Unused rust code

Severity	Warning	**Commit**	2095dda
Type	Maintainability	**Status**	Fixed
File(s)	`./malda_utils/src/types.rs`		
Location(s)	IOptimismPortal2		
Confirmed Fix At	`https://github.com/malda-protocol/malda-zk-coprocessor/pull/5`		

The following program constructs are unused:

1. `./malda_utils/src/types.rs`:
    a) Interface `IOptimismPortal2`

Impact Unused code can become out of sync with the rest of the codebase, and can introduce errors if used in later versions of the codebase.

Recommendation Remove the unused code for better maintainability.

Developer Response The developers have removed the unused interface.

5.1.7 V-MLDZK2-VUL-007: Fraudulent games may DoS the slow lane

Severity	Warning	**Commit**	2095dda
Type	Denial of Service	**Status**	Fixed
File(s)	`./malda_utils/src/validators.rs`		
Location(s)	validate_opstack_env_with_l1_inclusion()		
Confirmed Fix At	`https://github.com/malda-protocol/malda-zk-coprocessor/pull/5`		

The method `validate_opstack_env_with_l1_inclusion()` fetches the latest game index from the Dispute Game Factory and uses it to verify that the root claim corresponding to this particular block has not been challenged. See snippet below for the implementation.

```
pub fn validate_opstack_env_with_l1_inclusion(
    chain_id: u64,
    op_state_root: B256,
    env_eth_input: EthEvmInput,
    msg_passer_storage_hash: B256,
    ethereum_hash: B256,
    op_block_hash: B256,
) {
        // [Veridise] ..elided
    let game_count_call = IDisputeGameFactory::gameCountCall {};
    let contract = Contract::new(factory_adress, &env_eth);
    let returns = contract.call_builder(&game_count_call).call();

    let latest_game_index = returns._0 - U256::from(1);
```

Snippet 5.8: Snippet from `validate_opstack_env_with_l1_inclusion()`

Using the latest game index to fetch the dispute game can be dangerous, because an attacker can spam the Dispute Game Factory with spurious games. This will update the latest game index and the slow lane will be unable to access the dispute game related to the corresponding block that is to be validated.

Impact Using the most recent game index may allow people to DoS the slow lane by regularly creating spurious dispute games. Since these will not correspond to the `output_root_proof_hash` within the slow lane, proof creation will fail.

Recommendation Instead of using the most recent game index, pass in a specified game index as an input argument and verify the block number returned from the dispute game contract with the environment block number.

Developer Response The developers now supply a `game_index` variable used to determine the game.

The game creation time is checked against the timestamp when the respected game type was last updated, which is non-zero, and the game type is checked, ensuring the game at `game_index` is respected.

5.1.8 V-MLDZK2-VUL-008: Missing comments

Severity	Info	**Commit**	2095dda
Type	Missing/Incorrect Events	**Status**	Fixed
File(s)	`./malda_utils/src/validators.rs`		
Location(s)	See description		
Confirmed Fix At	`https://github.com/malda-protocol/malda-zk-coprocessor/pull/5`		

The following newly added functions/interfaces lack comments.

1. Function `validate_linea_env_with_l1_inclusion()` in `./malda_utils/src/valdiators.rs`.
2. Interface `IL1MessageService` in `./malda_utils/src/types.rs`.

Impact Missing documentation can make it more difficult for third party entities to understand the intended behavior for these functions, leading to maintainability concerns for future developers.

Recommendation Add documentation detailing the function arguments, return values, and expected panics.

Developer Response The developers have added documentation to the recommended locations mentioned above.

5.1.9 V-MLDZK2-VUL-009: Compilation issues

Severity	Info	**Commit**	2095dda
Type	Maintainability	**Status**	Fixed
File(s)		`./Cargo.toml`	
Location(s)		See description	
Confirmed Fix At		`https://github.com/malda-protocol/malda-zk-coprocessor/pull/5`	

The project could not be compiled without removing the `sequencer_v2` package from the `members`. See snippet below for context.

```
[workspace]
resolver = "2"
members = ["malda_rs", "methods", "patch/ethereum_hashing", "sequencer", "
    sequencer_v2", "malda_utils"]
exclude = ["lib"]
```

Snippet 5.9: Snippet from `Cargo.toml`

Recommendation Remove `sequencer_v2` from the `members` to resolve the compilation issues.

Developer Response The developers removed the `sequencer` and `sequencerv2` dependencies.

Glossary

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

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, 43

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. 43

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. 43

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

Semgrep Semgrep is an open-source, static analysis tool. See `https://semgrep.dev` to learn more. 5

Sequencer An entity or distributed algorithm responsible for determining transaction and block ordering on a layer-2 network.. 1, 3

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, 43

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. 43

zkEVM A zkVM implementing the Ethereum virtual machine. 43

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, 43