

Produced for



by



CHAINSECURITY

Contents

1	Executive Summary	3
2	Assessment Overview	5
3	Limitations and use of report	12
4	Terminology	13
5	Findings	14
6	Resolved Findings	16
7	Informational	24
8	Notes	26

1 Executive Summary

Dear all,

Thank you for trusting us to help Herodotus with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Herodotus on Starknet according to [Scope](#) to support you in forming an opinion on their security risks.

Herodotus provides a bridge between Ethereum's L1 and Starknet's L2, allowing for trustless proofs of state and storage values of Ethereum accounts on Starknet. Data integrity is ensured through on-chain verification mechanisms leveraging Merkle Mountain Range (MMR) and Merkle Patricia Trie (MPT) verifications.

The most critical subjects covered in our audit are security vulnerabilities and the validity and integrity of the state and storage proofs. Amongst others, the following issues have been uncovered:

- [Broken CairoLib Dependency](#)
- [MMR: Verify Against An Intermediate Node Is Possible](#)
- [Empty/inexistent storage slots can not be proven](#)

All high severity issues have been resolved.

The general subjects covered are functional correctness, robustness and usability.

In summary, we find that the codebase provides a good level of security. It's worth noting that more thorough testing could have identified most of these issues early. Moreover, there is still room for enhancement in the testing processes. Core functionality of the project is tested with minimal test cases only.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

-Severity Findings	0
-Severity Findings	4
•	4
-Severity Findings	3
•	3
-Severity Findings	4
•	2
•	1
•	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Herodotus on Starknet repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	19 September 2023	922117d3dbb97a69c8db3268033f2406ea62fa84	Initial Version
2	5 October 2023	728fdaddf91e86ec6080313e4fda2b8d52a1bad6	Bugfixes parallel to review
3	30 October 2023	48b14eeb4d276770ee0b58744755eec4b9492831	After Intermediate Report
4	1 November 2023	aac22133057764fd8052d10b2fc12c80835b7a85	Fixes
5	2 November 2023	9c8300a3aa4dc85b63037c7a02a3a916a1b1210b	Non Inclusion Proofs
6	9 November 2023	6db52a3339df4df04b80cd545f100b093188b15a	Fixes

For the solidity smart contracts, the compiler version is floating as $^0.8.13$. After the intermediate report the compiler version was fixed to $0.8.21$.

For the cairo code, the compiler version $2.2.0$ was chosen. At the time of this review (October 2023) Starknet v0.12.2 was live on mainnet. This review cannot account for future changes and possible bugs in Starknet and it's libraries.

The following file was in scope of this review:

```

11/src/L1MessagesSender.sol
11/lib/FormatWords64.sol
11/lib/Uint256Splitter.sol

src/core/commitments_inbox.cairo
src/core/evm_facts_registry.cairo
src/core/headers_store.cairo

src/remappers/interface.cairo
src/remappers/timestamp_remappers.cairo

```

2.1.1 Excluded from scope

Any file not listed above is excluded from the scope.

Cairo-lib, the library used for low level operations has been reviewed as part of another report. Several issues uncovered in that report affect Herodotus on Starknet. These issues have not been repeated in this report, please refer to the report of Cairo-lib.

2.2 System Overview

This system overview describes the initially received version () of the contracts as defined in the [Assessment Overview](#).

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Herodotus allows anyone to trustlessly prove any past headers, state, and storage values of L1 contracts to other L2 contracts via MMR (Merkle Mountain Range) and MPT (Merkle Patricia Trie) verifications.

2.2.1 L1 Contracts

The L1MessageSender contract acts as a bridge to relay data (Block Hashes and MMR Trees) from L1 / Ethereum to L2 / Starknet.

Upon the contract's deployment, several key values are initialized and `msg.sender` will be assigned the owner role.

- `starknetCore`: Core starknet contract used for L1->L2 messaging.
- `l2RecipientAddr`: Designated recipient on Starknet.
- `AggregatorFactory`: `AggregatorFactory` produces accurate MMR encapsulating historical blockhashes. Fully trusted.

Permissionless functionalities, accessible for anyone:

1. Sending BlockHashes from L1 to L2 is done using the following functions:

- `sendExactParentHashToL2()`: Sends a specific parent block hash to L2.
- `sendLatestParentHashToL2()`: Sends the latest parent block hash (blocknumber-1) to L2.

2. Sending Poseidon MMR Trees:

- `sendPoseidonMMRTreeToL2()`: Sends the PoseidonMMRTree for the given ID retrieved from the trusted aggregator factory to L2.

Sending messages incurs a charge in Ether, the caller must send sufficient Ether along the transaction.

3. The owner can access functionalities to update the following system variables:

- `setL2RecipientAddr()`: Updates the L2 recipient address.
- `setAggregatorsFactoryAddr()`: Updates the address of the aggregators factory.
- `transferOwnership()`: Transfer the ownership to a new address.
- `renounceOwnership()`: Relinquish the ownership by setting the owner to zero address.

Message cancellation mechanism is not supported by L1MessageSender, once sent to Starknet Core, the message cannot be cancelled.

2.2.2 L2 Contracts

2.2.2.1 Commitments Inbox

The CommitmentsInbox contract serves as an intermediary receiving the messages from L1 and forwards the data to the HeadersStore contract.

Upon initialization the following addresses are set:

- `headers_store`: A reference to the contract responsible for storing information about valid headers.
- `l1_message_sender`: The Ethereum address permitted to send messages to this contract.
- `owner`: Optional. If not provided, the caller becomes the owner.

Getters and Setters:

- `get_headers_store()`: Retrieves the address of the headers store contract.
- `set_headers_store(address)`: Allows the owner to update the headers store address.
- `get_l1_message_sender()`: Fetches the Ethereum address stored as the L1 message sender.
- `set_l1_message_sender(address)`: Permits the owner to set a new Ethereum address as the L1 message sender.

Ownership related functionalities:

- `get_owner()`: Retrieves the contract's current owner address.
- `transfer_ownership(new_owner)`: Enables the owner to transfer ownership rights to another address. An event (`OwnershipTransferred`) is emitted following a successful transfer.
- `renounce_ownership()`: Allows the owner to relinquish their ownership. An event (`OwnershipRenounced`) is triggered post-renouncement.

Commitment Reception (Owner Only):

- `receive_commitment_owner(blockhash, block_number)`: Allows the owner to submit a blockhash for a given block number. This data is sent to the `headers_store` contract, followed by the emission of the `CommitmentReceived` event.

L1 Handler for Commitment Reception:

`receive_commitment(from_address, blockhash, block_number)`: Receives L1 messages from `l1_message_sender` and forwards the blockhash and block number to the `headers_store` contract. Emits the `CommitmentReceived` event.

L1 Handler for MMR Data Reception:

`receive_mmr(from_address, root, last_pos, aggregator_id)`: Receives L1 messages with MMR data from the L1 message sender. The data is transmitted to the `headers_store` contract for processing. An `MMRReceived` event is emitted subsequently.

2.2.2.2 Headers Store

Headers Store, bound to contract Commitments Inbox, maintains a mapping of received blocks, MMRs of block Poseidon hashes, and MMR historical records. It's main purpose is to serve as reference to check valid block headers.

The following functions are restricted to the Commitments Inbox:

- `receive_hash()`: This will be called every time the Commitments Inbox gets a L1 call to `receive_commitment()`. It simply stores the received block number and parent hash into a

mapping. Later, the received blocks can be consumed and added to any of the MMRs in a permissionless way.

- `create_branch_from_message()`: This will be called every time the Commitments Inbox gets a L1 call to `receive_mmr()`. It will create a new MMR into the MMR forest with the received `root` and `last_pos`.

Users can also checkout from an existing MMR to a new MMR (`create_branch_from`) or create a new MMR from one element in another MMR (`create_branch_single_element`).

`process_batch()` allows users to batch append block hashes to any MMRs. Users need to submit a batch of consecutive blocks with descending block numbers. It only needs to verify the inclusion or the receipt of the most recent block, since a child block refers to a parent block deterministically: the parent block header can be verified by `keccak256` hashing and comparing it with the `parentHash` stored in the child block.

This verification is done using the actual `keccak256` hash of the block, the entry into the MMR however is the Poseidon hash (much cheaper to compute on Starknet) of the block. Hence for all MMR operations the Poseidon hash must be passed.

Once included in the MMRs, anyone can prove against the block hashes onchain by calling:

- `verify_mmr_inclusion()`: Verify an MMR inclusion proof against an MMR.
- `verify_historical_mmr_inclusion()`: Verify an MMR inclusion proof against a historical record of an MMR.

The following getters are further provided:

- `get_commitments_inbox()`: Returns the Commitments Inbox contract address.
- `get_mmr()`: Returns the MMR given a specific id.
- `get_mmr_root()`: Returns the root of a specific MMR.
- `get_mmr_size()`: Returns the `last_pos` (size) of a specific MMR.
- `get_received_block()`: Returns a received block hash given a block number.
- `get_latest_mmr_id()`: Returns the latest created MMR id.
- `get_historical_root()`: Returns the historical root of a MMR at a certain size.

2.2.2.3 EVM Facts Registry

EVM Facts Registry implements the functionalities for users to prove and register facts (the fields or the storage of an address towards a designated block) by onchain verification of MMR and MPT proofs in a permissionless way. This contract is bound to a Headers Store.

The state of an Ethereum account, the quadruplet fields (`StorageHash`, `CodeHash`, `Balance`, `Nonce`) of a 20-bytes Ethereum address are stored in the World-State-MPT with a 32-bytes key: `keccak256(address)`. Its storage of a given slot is stored in the Account-State-MPT with a 32-bytes key: `keccak256(slot)`. One can prove the quadruplet fields and the storage values by constructing proofs against the `block.stateRoot` and the `account.storageRoot` respectively.

The user can prove any fields of an account (`prove_account()`) and write a subset of the fields into the registry contract.

1. **Block Verification:** The user should submit the RLP-encoded block header together with a proof of its inclusion in one MMR of the Headers Store. After a successful verification of the block's inclusion, the world state root and the block number will be retrieved from the block header.

2. **Account Verification:** The user submitted account MPT proof can be verified with the obtained state root. Once the verification succeeds, the account fields are decoded and some selected fields (in little-endian) will be written in the registry contract.

The user can only prove the storage of an account (`prove_storage()`) once it proves and stores its `StorageHash` into the contract by `prove_account()`. One should submit a MPT proof against the previously verified `StorageHash`. The slot value (in little-endian) will also be written in the registry contract.

Note that one can only prove the fields and storage slots given included blocks in the Headers Store. Besides, `prove_account()` expects a vanilla Ethereum address, whereas `prove_storage()` expects `keccak256(slot)` instead of slot itself as input.

The following view functions are also provided:

- `get_headers_store()`: Returns the Headers Store contract address.
- `get_account_field()`: Returns the value of a field given (`account, block`). If the field has not been proved yet, it will return `Option::None`.
- `get_slot_value()`: Returns the storage value of a slot given (`account, block`). If the slot has not been proved yet, it will return `Option::None`.
- `get_account()`: Returns the value of some fields given (`account, block`) and proofs.
- `get_storage()`: Returns the storage value of a slot given (`account, block`) and proofs.

2.2.2.4 TimeStamp Remappers

Headers Store accomplishes the unordered inclusion of block hashes into the MMRs, as a consequence, users can not find the closest Ethereum block number towards a timestamp. Therefore, TimeStamp Remappers is implemented as a permissionless contract to re-index the blocks of Headers Store into MMRs with strictly ascending consecutive order. This contract is bound to a Headers Store.

Anyone can create a new mapper (`create_mapper()`) with a start block number. Each mapper is identified by its unique mapper id. The start block, last timestamp, elements number, as well as its historical MMRs are tracked in the contract.

One can update a designated remapper (`reindex_batch()`) by inserting the next blocks. For each of the blocks:

1. **Expected Block Number:** The user submitted block header will be decoded to retrieve its block number and timestamp. The block number should match the exact next one in the remapper.
2. **Block Header Verification:** The block hash will be computed from the header and its inclusion will be verified towards a historical MMR in Headers Store.
3. Once everything succeeds, the block timestamp will be appended to the remapper MMR.

One can query the last timestamp of a remapper by `get_last_mapper_timestamp()`.

Function `get_closest_l1_block_number()` is implemented to find the closest block number that is no later than the input timestamp. As the remapper MMR does not store the leaves (timestamps) itself, users should submit the MMR inclusion proofs of the binary search's mid-point elements. It will do a binary search on a remapper MMR and verify the inclusion of the mid-point in each iteration. An inclusion proof of the result leaf should also be provided by the user.

This provides the utility to map arbitrary timestamps to L1 block numbers.

2.2.3 Roles and Trust Model

The state and storage proofs work in a permissionless way with the contracts ensuring the integrity of the data. Contracts have permissionless entrypoints for any callers to initiate updates, the integrity of the data is ensured in the contracts by validating the submitted proofs. A proof can only be valid if it reflects the actual state of the L1 data.

There are some caveats to this trustless system:

- The CommitmentInbox has a privileged owner, which can update commitments of `blockhash` and `blocknumber`. For the trustless operation, it's imperative that the owner has renounced ownership without having submitted incorrect data before.
- Not only blockhashes & numbers directly queried with the EVM Opcode can be sent to L2 but also pre-aggregated MMRs thereof provided by the AggregatorFactory. The AggregatorFactory (out of scope of this review) hence must be fully trusted to provide valid MMRs only. Otherwise, forged block hashes can be inserted, on top of which all the proven states could be manipulated.
- L1 contract L1MessageSender has an owner which can update the L2 recipient address and the aggregator factory. Due to the ability of being able to update the aggregator, this role must be fully trusted. Otherwise, a malicious aggregator could be added, which can propagate forged block hashes to L2.

The smart contracts are immutable and not to be deployed behind proxies.

Users interacting with the system are untrusted.

The cross-chain message of Herodotus on Starknet is dependent on the Starknet Core Contracts on L1 and the Starknet Sequencer. The Starknet Core Contracts are upgradable and could be changed in the future. The sequencer has not yet been decentralized at the time of this review (October 2023) and could impose censorship on messages. They are untrusted.

2.2.4 Changes in Version 3

The following functional changes were introduced in `evm_facts_registry`:

- `get_storage()`: Actual value stored is now returned instead of the RLP encoded value.

`headers_store`:

- Events `BranchCreatedFromElement`, `BranchCreateFromL1` and `BranchCreateClone` replace the generic `BranchCreated` event.

2.2.5 Changes in Version 4

The following functional changes were introduced in `evm_facts_registry`:

- Values of account fields and storage slots are now handled in big endian.

2.2.6 Changes in Version 5

The following functional changes were introduced in `evm_facts_registry`:

- Support to prove non-inclusion of accounts in the World State Trie has been added.

2.2.7 *Changes in Version 6*

The following functional changes were introduced in :
evm_facts_registry:

- Support to prove unset slots / slots with a value of zero in the Account Storage Trie has been added.

Header Store:

- The event (`ProcessedBatch`) emitted from `process_batch()` has been updated to be inclusive on both the start block and the end block appended.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High			
Medium			
Low			

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the [Resolved Findings](#) section. The findings are split into these different categories:

- : Related to vulnerabilities that could be exploited by malicious actors
- : Architectural shortcomings and design inefficiencies
- : Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

-Severity Findings	0
-Severity Findings	0
-Severity Findings	0
-Severity Findings	2

- [Loosely Restricted Account Type](#)
- [Floating Pragma and Dependency](#)

5.1 Loosely Restricted Account Type

CS-HRDS-001

In contract evm facts registry, function `get_account_field` and `get_account` accepts an input account (Ethereum address) as a `felt252`. As an Ethereum address is only 160-bit long, it is loosely restricted. Though the internal `get_account()` will eventually revert upon casting `u256` into `u64` if the input's upper 96 bits are not zeros.

5.2 Floating Pragma and Dependency

CS-HRDS-002

For the L1 Solidity contracts, the pragma is floating and no compiler version has been fixed.

Similarly the `cairo_lib` dependency is not fixed in `Scarb.toml`, instead the most recent version of `master` is used.

Contracts should be deployed using dependencies and compiler version they have been thoroughly tested with.

Code partially corrected:

The Solidity compiler version has been fixed to version 0.8.21. However, the `cairo_lib` dependency is still floating, now tracking the branch audit.

DRAFT



6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the [Findings](#) section.

Below we provide a numerical overview of the identified findings, split up by their severity.

-Severity Findings	0
--------------------	---

-Severity Findings	4
--------------------	---

- [Wrong Values for Non-Existing Accounts](#)
- [Broken CairoLib Dependency](#)
- [Empty/inexistent Storage Slots Can Not Be Proven](#)
- [MMR: Verify Against an Intermediate Node Is Possible](#)

-Severity Findings	3
--------------------	---

- [Incorrect Expected Block Computation](#)
- [Remapper Last_Timestamp Can Be Reset to 0](#)
- [Unchecked Slot Length](#)

-Severity Findings	2
--------------------	---

- [End Block May Underflow if the Start Block Is 0](#)
- [Missing OwnershipRenounced Event by Transferring to Zero Address](#)

6.1 Wrong Values for Non-Existing Accounts

CS-HRDS-009

Version 5 added support to prove non-inclusion of accounts in the state trie, which means there is no valid path in the world state trie that matches the input key (keccak of account address). For accounts not present in the given world state trie, the following default values are returned:

```
const EMPTY_STORAGE_HASH: u256 =
    0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cad001622fb5e363b421;
const EMPTY_CODE_HASH: u256 =
    0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470;
const EMPTY_BALANCE: u256 = 0x0;
const EMPTY_NONCE: u256 = 0x0;
```

This is incorrect: For non-existing accounts at a given block, all fields are 0x0.

For existing accounts without code or storage the fields are set to EMPTY_CODE_HASH or EMPTY_STORAGE_HASH respectively.

Code corrected:



Code has been corrected to return 0x0 for fields of inexistent accounts.

6.2 Broken CairoLib Dependency

CS-HRDS-012

The smart contracts are built on Herodotus CairoLib and inherits the vulnerabilities it has. Here we provide a non-exhaustive list:

- Keccak Discards Leading Zero Bytes In Last Little Endian Words64:
 1. `HeaderStore.process_batch()` will revert in case the RLP-encoded header satisfies the condition.
 2. `EVMFactsRegistry`'s internal `get_account()` will compute account hash incorrectly in case the account satisfies the condition.
- MMR: Incorrect Root Update possible, Insufficient Peaks Validation:
 1. All the calls to `MMR.append()` in `HeaderStore` and `TimestampRemappers` are subject to this vulnerability.
- Missing Length Validation In MPT Verify:
 1. `EVMFactsRegistry`'s `get_storage()` and internal `get_account()` may revert on some valid proofs. And a forged key may succeed in verification.
- `reverse_endianness_u64()` Discards Leading Zeros:
 1. `TimestampRemappers`'s `extract_header_block_number_and_timestamp()` may return incorrect block number and timestamp if their little endian representations have leading zero bytes.

The Herodotus CairoLib has been reviewed in a separate report, please refer to it for details on the underlying issues.

Code corrected:

All issues have been resolved in the underlying CairoLib code. For details please refer to the separate report.

6.3 Empty/inexistent Storage Slots Can Not Be Proven

CS-HRDS-008

Unused / non-existing storage slots of an account default to a value of 0x0. These slots cannot be proven successfully as the `mpt.verify()` fails or starting from of the code, `mpt.verify()` returns an empty list which cannot be successfully RLP-decoded.

This e.g. prevents proving zero balance of an ERC20 token.

Code corrected:

Proving unset slots / slots with a value of zero is now possible. The change introduced in related to non inclusion proofs allows to prove non-existence of a key in a trie. `evm_facts_registry.get_storage()` has been updated to handle the case of a successful non inclusion verification.

Note that the code doesn't allow to prove any storage slot for non-existing accounts. Similarly for accounts without storage the storage root is initialized to the keccak of `0x80 - 0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cad001622fb5e363b421`. No storage slots can be proven for these accounts.

6.4 MMR: Verify Against an Intermediate Node Is Possible

CS-HRDS-015

From the perspective of `mmr.verify_proof()`, it cannot distinguish the value verifying against is a leaf or an intermediate node (a Poseidon hash of its two children). Thus theoretically you can verify against any node that is inside the MMR. Herodotus on Starknet uses MMRs in both Headers Store and Timestamp Remappers. It is unlikely to abuse the verification of an intermediate node in the former, nevertheless, it is feasible in the latter.

In the following we denote a block header as h , Poseidon **hash double** operation as F and **hash many** as G , and Poseidon hash of a block header as $ph=G(h)$.

Headers Store: the MMRs leaves store the Poseidon hashes of RLP-encoded block headers ($leaf=G(h)$). In case one wants to abuse an intermediate node ($node=F(ph1, ph2)$), it needs to find a fake block header($h3$), so that the following equation holds:

$$F(ph1, ph2) = G(h3)$$

Since **hash double** and **hash many** starts with different third states in the underlying hashes permutation, it is unlikely to find a header ($h3$) that satisfies the equation above.

Timestamp Remappers: the MMRs leaves store the block timestamps directly without hashing. Upon querying `get_closest_ll_block_number()`, users need to pass the MMR inclusion proofs of mid point **leaf** nodes against one mapper's MMR during the binary search.

It seems unlikely to replace a **leaf** node with an **intermediate** node as the leaf index is fixed with respect to the mid point. However, `mmr.verify_proof()` reconstructs the peak according to the length of the proofs instead of the tree height reflected by the index. Consequently, one can replace a **leaf** node (a block timestamp) with an **intermediate** node (a Poseidon hash) to manipulate the result of binary search.

Code corrected:

This has been fixed in the underlying library. Verification against intermediate nodes is no longer possible. For details please refer to issue 5.5 MMR Verify Proof: Different Nodes Can Use the Same Index of the CairoLib report.

6.5 Incorrect Expected Block Computation

In contract timestamp remapper's `reindex_batch()`, the `expected_block` will be computed based on the start block (`start_block`) and the element amount (`elements_count`) of this remapper:

```
let mut expected_block = 0;
if (mapper.elements_count == 0) {
    expected_block = mapper.start_block;
} else {
    expected_block = mapper.start_block + mapper.elements_count + 1;
}
```

This calculation in the `else` branch is incorrect. For example, in case the start block number is 0 and the current elements count is 1, the expected next block should be block 1, nevertheless, the `else` branch will return 2.

Code corrected:

The computation of `expected_block` has been simplified and corrected.

6.6 Remapper Last_Timestamp Can Be Reset to 0

In contract timestamp remapper's `reindex_batch()`, the corresponding `mapper.last_timestamp` will be set to a local variable `last_timestamp`, which is 0 at the beginning, and gets updated according to the last element in the input block headers. However, if there is no element in the input batch, one can skip the loop and directly reset the `mapper.last_timestamp` to 0.

This can invalidate all the consecutive calls to `get_closest_ll_block_number()`, since all valid block timestamps are larger than 0, which will be larger than `mapper.last_timestamp` and the function will revert.

Code corrected:

The function now reverts if the input is an empty batch, this resolves the issue described above.

6.7 Unchecked Slot Length

In evm facts registry, function `prove_storage` accepts an input `slot`, whose name is misleading since what it actually expects is `keccak256(slot)` as per Ethereum MPT specification (see [ref.1](#) and [ref.2](#)). Moreover, users have full control over the input `slot_len`, which indicates the length of the key in the MPT. This leads to the following consequences:

- The legitimate keys of storage slots are `keccak256` digest with a 64-nibble (32-byte) fixed length. Thus the slot values should be stored in leaf nodes. However, now a user can get the value stored in a branch node with a key less than 64-nibble long, which should not be a valid use case.

- In `prove_storage()`, the storage value will be retrieved according to the `slot_len` (only partial of input `slot`), nevertheless, the whole `slot` (instead of the first `slot_len` nibbles) will be stored as the key for the retrieved value.

In summary, this enables the proving of values in branch node and storing values in branch node as slot values. This applies to `get_storage()` as well.

Code corrected:

The input argument `slot` is now hashed inside the functions before being used as key. This guarantees that the keys length is always fixed to 64 nibbles. The name of the variable is no longer misleading since it now actually represents the slot.

6.8 End Block May Underflow if the Start Block Is 0

CS-HRDS-020

In contract headers store, users can append a batch of block hashes (`process_batch()`) to an MMR via two approaches:

1. First block received from the inbox: submit a reference block to query from `received_blocks`. And the `start_block` will be set to the `reference_block-1`
2. First block already included in an MMR: submit an inclusion proof of the first element towards the current MMR. In case the `reference_block` is `None`, the `start_block` will be read directly from the first header.

The function parameters with data for both approaches (`reference_block` and `mmmr_index + mmmr_proof`) are of type `Option<T>`, since for normal usage only one of both should be supplied.

In case the `reference_block` is not `None` in the second approach (passing `mmmr_index` with `mmmr_proof`), the `start_block` (initialized to 0) will not be updated, and the function may revert due to underflow when computing the `block_end`.

Code corrected:

Passing a reference block alongside an MMR proof is now prevented.

6.9 Missing OwnershipRenounced Event by Transferring to Zero Address

CS-HRDS-019

In the `commitments` inbox contract, there is no sanity checks of the `new_owner` in `transfer_ownership()`. Thus, the owner can renounce its ownership by either calling `renounce_ownership()` or transferring ownership to zero address (`transfer_ownership()`).

However, in the second option, `transfer_ownership()` only emits an event `OwnershipTransferred` instead of `OwnershipRenounced`.

Code corrected:

If the ownership is transferred to the zero address using `transfer_ownership()`, the event `OwnershipRenounced` is now emitted instead of the event `OwnershipTransferred`.

6.10 Close to Ressource Limit in Starknet

CS-HRDS-013

As of October 2023 with Starknet v0.12.2 the current step limit per transaction is 3 million, and gas limit is 3 million per block.

The complexity of executing `prove_account()` and `prove_storage()` depend on the position of the account and the storage slot in the state trie. For most accounts and slots this is feasible to execute with less than 2.5 million steps.

It is anticipated that the step limit and gas efficiency in starknet will be increased in the near future.

Code corrected:

The code was optimized significantly:

- The RLP decoder now implements `rlp_decode_list_lazy` which allows to decode only the required elements.
- The `eth_mpt()` implementation now makes use of `lazyBranch` helpers: Instead of having to load all elements within a branch node entirely, it only loads the designated element of a branch node.

6.11 Duplicated Peaks of Proof_Element

CS-HRDS-017

In `mmr_binary_search()` of Timestamp Remappers, each proof to be verified against a mid point has a `peaks` field in `proof_element`. Given the fact that the MMR will not be modified during the binary search, it is redundant to duplicate the `peaks` within every proof element.

Code corrected:

Herodotus has removed the `peaks` entry from the individual proof element and only submit it once in the input.

6.12 Option<T> but Only Actual Element Supported

CS-HRDS-016

Struct `BinarySearchTree` of the timestamp remapper defines field `left_neighbor` as `Option<ProofElement>`. The implementation doesn't support an option of this element / `None` and only works if there is a `ProofElement`.

Code corrected:

Code has been updated to only verify the `left_neighbor` in case it has not been verified within the binary search.

6.13 Unnecessary Retrieval of Header Store

CS-HRDS-010

In the Timestamp Remappers `mmr_binary_search` function, the address of the headers store is retrieved:

```
// Retrieve the header store address
let headers_store_addr = self.headers_store.read();
```

The headers store however is not used in this function. The validity of the data in the MMR is verified in `reindex_batch()` when new elements are added. When doing the binary search in the MMR it is only ensured that the elements are indeed part of the MMR.

Code corrected:

This has been removed from the code.

6.14 Unused FormatWords64 Library

CS-HRDS-018

`FormatWords64` is implemented and imported in `L1MessagesSender`, whereas it is never used.

Code corrected:

`FormatWord64` has been removed from the codebase.

6.15 Unused `ProofElement.last_pos`

CS-HRDS-011

Struct `ProofElement` of the timestamp remapper defines a field `last_pos` which is never used.

Code corrected:

The unused element has been removed from the struct.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Front-running Process Batch, Verify Mmr, Reindex Batch

CS-HRDS-003

If front-running is feasible (consider decentralized sequencers in the future), this could be leveraged to revert certain transactions:

- `Header_store.process_batch()`: if an MMR inclusion proof is submitted, it will be verified against the current MMR. A call to `process_batch()` can be front-run by another call to `process_batch()` to update the peaks and revert the former call.
- `verify_mmr_inclusion()`: a call to this function can be front-run by another call to update the MMR and thus invalidate the previous proofs.
- `timestamp_remappers.reindex_batch()`: a call to `reindex_batch` can be front-run by a another call to it, which invalidates the previous call.

For public functions branching from existing mmr's (`create_branch_from()` and `create_branch_single_element()`) callers now additionally supply `last_pos` in addition to the `mmr_id`. This allows to verify the supplied proof against the historic mmr successfully even in case a preceding transaction updated the specific mmr.

7.2 Merge128 Discards Higher 128 Bits of Upper Input

CS-HRDS-004

`merge128()` accepts two `uint256` inputs (`lower` and `upper`) and merges them into one `uint256`. It is expected that both inputs only use the lower 128 bits of `u256`. It checks the `lower` does not exceeds `max(uint128)`, whereas the `upper` is not checked and its higher 128 bits will be discarded if they contain non zero bits.

Herodotus is aware and states this is not applicable to their codebase.

7.3 Unrestricted Aggregator ID and MMR Size

CS-HRDS-005

In `L1MessagesSender`, the aggregator id and MMR size are represented as `uint256`. They are not checked and directly used as fields in the L1 -> L2 message payload. As a result, they may exceed the maximum `usize`, which is the expected type of inputs in Commitments Inbox.

Besides, in Starknet the basic data type is `felt252`. As the aggregator id and MMR size are not restricted to fit in a `felt252`, in case they exceed `max(felt252)`, the message will be stuck and never consumed on L2.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 EIP-161

In the early days of Ethereum, empty accounts could remain in the state trie. Empty in this context means the account has a nonce of zero, zero balance and no code but might have some storage entries.

EIP-161 introduced in the Spurious Dragon hardfork at Block 2'675'000 on mainnet prevents new empty accounts and enables removal of existing empty accounts out of the state trie if a transaction touches them.

Since these empty accounts are valid accounts of the state trie, these accounts and their storage can be proven with Herodotus.

As per block 14049881 all empty accounts are supposed to have been removed from the state trie.

For more information please refer to the EIP: <https://eips.ethereum.org/EIPS/eip-161>

This stagnant EIP contains more examples and lists affected addresses: <https://eips.ethereum.org/EIPS/eip-4747>

8.2 Ethereum Block Header May Be Subject to Future Updates

The fields included in an Ethereum block header may change in future upgrades. It is expected that new fields will only be appended to the current block header, which shall not break the compatibility with Herodotus. See an example from the last Shanghai upgrade: <https://eips.ethereum.org/EIPS/eip-4895#new-field-in-the-execution-payload-header-withdrawals-root>.

Other changes of the block header (which are unlikely and not expected to happen) would break Herodotus.

8.3 None Block Header Exists in MMR

In the constructor of header store, the MMR with id 0 is initialized with the hash of an initial element "brave new world", which is not a legitimate block hash. This MMR can also be forked to another MMR with a different id via `create_branch_from()`. All external systems that integrate with `verify_mmr_inclusion()` or `verify_historical_mmr_inclusion()` should execute caution and enforce the target it verifies against is a legitimate block hash.