Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

RISC ZERO

Steel

Veridise Inc.
May 7, 2025

► **Prepared For:**

Risc Zero
https://risczero.com/

► **Prepared By:**

Jon Stephens
Mark Anthony
Victor Faltings

► **Contact Us:**

contact@veridise.com

► **Version History:**

May 7, 2025	V2
April 30, 2025	V1
April 28, 2025	Initial Draft

Contents

1 Executive Summary

From April 14, 2025 to April 24, 2025, Risc Zero engaged Veridise to conduct a security assessment of their Steel library. The security assessment covered the updates made to the Steel zkVM application library. Compared to the previous version, which Veridise has audited previously*, the new version adds several new features, including the ability to create historical proofs for older execution blocks, the ability to prove that an event was emitted in a block and the ability to verify a steel commitment with respect to another environment. Veridise conducted the assessment over 27 person-days, with 3 security analysts reviewing the project over 9 days on commit `2c99f46`. 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 security assessment covered the Risc Zero Steel library. The goal of this library is to allow developers to prove the execution of queries made on a given Ethereum state within a zkVM guest program. This library focuses on supporting the execution of *view functions*, a subset of functions which do not modify the state of a contract. The updates to the library, which were the focus of this audit, mainly comprised the addition of 4 new features:

- **History commits:** this feature allows the creation of proofs for historical proofs, greatly extending the range of blocks that Steel can make proofs about. Users supply a *execution block*, representing the state on which the queries will be executed, and a *commitment block*, representing the state which can be verified by an on-chain verifier. Steel then constructs a chain of beacon blocks between both of these blocks, using the *Beacon roots contract* from EIP-4788.
- **Event queries:** this feature allows guest applications to prove the result of event queries matching a given filter. The process of proving such queries remains similar to proving the execution of view functions, with a pre-flight step on the host application before the guest application can execute the same query.
- **Steel verifier:** this feature allows users to verify that a Steel commitment is an ancestor of another. To do so, an application verifies the correctness of one commitment with respect to the environment of another commitment.
- **Account information:** with this new feature, users can query the state for information about a particular account such as their balance or nonce.

Code Assessment. The Steel developers provided the source code of the Steel contracts for the code review. The source code appears to be mostly original code written by the Steel 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 Steel developers shared a documentation page containing an overview of the project† as well as a number of examples‡ showcasing its usage. The Steel developers also met with the

* The previous audit report can be found on Veridise's website at `https://veridise.com/audits/`
† Available at https://docs.beboundless.xyz/developers/steel/what-is-steel
‡ Available at https://github.com/risc0/risc0-ethereum/blob/main/examples

Veridise security analysts and provided a high level walkthrough of the new features and their functionality.

The source code contained a test suite, which the Veridise security analysts noted covered most of the important workflows of the project.

Summary of Issues Detected. The security assessment uncovered 6 issues, 1 of which are assessed to be of a high severity by the Veridise analysts. Specifically, V-STL-VUL-003 highlights that when extending a host environment with another compatible environment, the `log_filters` field of the `ProofDb` is not extended, which may result in the guest environment not possessing relevant information when an event query is made within the guest. And V-STL-VUL-001 shows how the `SteelVerifier` may validate commitments with environment configurations that are different from the execution environment.

The Veridise analysts also identified 4 low-severity issues, including V-STL-VUL-002 which explains why the current method of traversal over the ring buffer within the beacon roots contract does not guarantee capturing the required historical blocks for a history commitment, V-STL-VUL-004 which highlights that the `ProofDb` may overwrite storage tries with conflicting roots when converting the host environment to a guest input, and V-STL-VUL-005 which details that the environment chain specifications need to be explicitly defined otherwise it may silently default to an environment which is inconsistent with the provided commitment, as well as 1 informational finding.

The Steel developers have acknowledged and provided fixes for the reported issues.

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
Steel	`ee1c455-2c99f46`	Rust	Risc Zero zkVM

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
April 14–April 24, 2025	Manual & Tools	3	27 person-days

Table 2.3: Vulnerability Summary.

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

Table 2.4: Category Breakdown.

Name	Number
Logic Error	3
Data Validation	2
Maintainability	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of Steel's source code. During the assessment, the security analysts aimed to answer questions such as:

- Can the guest application be made to prove something incorrect about the state of the blockchain?
- Does the guest perform the necessary validations during transaction execution?
- Can the guest application be blocked from being able to execute queries?
- Can the historical commitment feature be used to generate proofs about state in the future?
- Does the host collect and provide all the information required to perform the queries within the guest?
- Does the commitment contain the required information regarding the guest execution environment, and is it verified correctly when validating steel commitments?
- Can future upgrades to Ethereum have any effect on the Steel library?
- Are the commitments to the `BlockInput`, `BeaconInput` and `HistoryInput` appropriately validated within the guest?

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 techniques:

- *Static analysis.* To identify potential common vulnerabilities, security analysts made use of the open-source tool `cargo audit`, which is designed to find known issues in dependencies in Rust programs.
- *Fuzzing/Property-based Testing.* Security analysts used fuzz testing to determine if the behavior within the guest deviates from expected behavior. To determine if an inconsistency can be found, they made use of a custom fuzzing tool to compare the execution output of view calls within the guest environment and the host environment.

Scope. The scope of this security assessment is limited to the additions/modifications made to the `risc0-ethereum/crates/steel/src` directory from commit `ee1c455` to commit `2c99f46`. This directory provided by the Steel developers contains the source code for the Steel library.

Methodology. Veridise security analysts reviewed the reports of previous audits for Steel, inspected the provided tests, and read the Steel documentation. Before the security assessment began, the Veridise security analysts met with the Steel developers to ask questions about the code and to get an understanding of the new features implemented. They then began a manual review of the code assisted by automated testing.

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 Trust Model

4.1 Operational Assumptions.

In addition to assuming that any out-of-scope components behave correctly, Veridise analysts assumed the following properties held when modeling security for Steel.

- Any information in a Steel commitment will be written to the journal of the guest application and be verified *on-chain*.
- All external crates used by Steel (such as `revm` and `alloy`) do not have any vulnerabilities within them.

5 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. 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-STL-VUL-001	SteelVerifier can verify commitments with . . .	High	Fixed
V-STL-VUL-002	Traversal of the ring buffer does not . . .	Low	Fixed
V-STL-VUL-003	ProofDb does not extend log_filters when . . .	Low	Fixed
V-STL-VUL-004	ProofDb overwrites storage tries with . . .	Low	Fixed
V-STL-VUL-005	Implicit dependency on with_chain_spec . . .	Low	Fixed
V-STL-VUL-006	Code quality improvements	Info	Fixed

5.1 Detailed Description of Issues

5.1.1 V-STL-VUL-001: SteelVerifier can verify commitments with different environments

Severity	High	**Commit**	2c99f46
Type	Data Validation	**Status**	Fixed
File(s)	`src/verifier.rs`		
Location(s)	verify		
Confirmed Fix At	https://github.com/risc0/risc0-ethereum/pull/561, 03472b2		

The steel library allows an execution environment to be validated against another commitment via the `verify` function shown below in the `SteelVerifier`. Doing so will check that the input commitment is within the history of the environment by either validating that the beacon root at the indicated timestamp matches the root given by the commitment or by validating that the block hash at the indicated block number matches the commitment. While this does validate that the block described by the commitment has a state consistent with the current environment's state, it does not ensure that they were executed with the same execution configuration. As an example, this environment could describe a configuration on Ethereum Mainnet and validate a commitment with a state consistent with the current block's history but with a manipulated REVM configuration.

```
pub fn verify(&self, commitment: &Commitment) {
    let (id, version) = commitment.decode_id();
    match version {
        0 => {
            let block_number =
                validate_block_number(self.env.header().inner(), id).expect("Invalid
    id");
            let block_hash = self.env.db().block_hash(block_number);
            assert_eq!(block_hash, commitment.digest, "Invalid digest");
        }
        1 => {
            let db = WrapStateDb::new(self.env.db(), self.env.header());
            let beacon_root = BeaconRootsContract::get_from_db(db, id)
                .expect("calling BeaconRootsContract failed");
            assert_eq!(beacon_root, commitment.digest, "Invalid digest");
        }
        v => unimplemented!("Invalid commitment version {}", v),
    }
}
```

Snippet 5.1: Definition of the `verify` function

Impact A user could make it appear as though they were executing a block under a different set of conditions. For some applications, this could allow someone to prove information that is inconsistent with the execution environment described in the current environment. As an example, consider a case where an application checks that the current ChainID is *not* Ethereum mainnet. Since the ChainID is not included in a block but rather is a property of the environment and ConfigID, the above function will accept commitments from different chains. Therefore, someone could bypass this check using a configuration with some other ChainID but make it appear as though the call were performed on mainnet by verifying the commitment against an Ethereum mainnet commitment.

Recommendation Validate the environment's configuration against the configuration given in the input commitment.

Developer Response The developers have addressed the issue as follows:

> We have addressed this in a PR. Specifically:
>
> 1. The standard `SteelVerifier::verify()` method now enforces that the commitment's `configID` matches the verifier's environment `configID`.
> 2. We've introduced `SteelVerifier::verify_with_config_id()` for advanced scenarios where differing `configID`s are legitimate and intended (e.g., post-hard fork verification or future cross-layer L2/L1 commitment verification).
>
> This approach ensures the common verification path is secure by default, while `verify_with_config_id()` offers necessary flexibility for specific use cases, requiring explicit handling of the `configID` by the developer in those instances.

5.1.2 V-STL-VUL-002: Traversal of the ring buffer does not guarantee capturing the required historical beacon blocks

Severity	Low	**Commit**	2c99f46
Type	Logic Error	**Status**	Fixed
File(s)	src/mod.rs		
Location(s)	from_headers()		
Confirmed Fix At	https://github.com/risc0/risc0-ethereum/pull/562, e0fc9d7		

The function `from_headers()` is used to construct a `HistoryCommit`, which is a verified sequence of **beacon chain state commitments** that trace the historical relationship between two Ethereum blocks, namely the execution block and the commitment block. It iterates progressively from the environment headers number up to the commitment headers number generating `StateCommits` for each block traversed during the iteration. The implementation can be seen in the snippet below.

```
pub(crate) async fn from_headers<P>(
  evm_header: &Sealed<EthBlockHeader>,
  commitment_header: &Sealed<EthBlockHeader>,
  rpc_provider: P,
  beacon_url: Url,
) -> anyhow::Result<Self>
where
  P: Provider<Ethereum>,
{
  ensure!(
      evm_header.number() < commitment_header.number(),
      "EVM execution block not before commitment block"
  );
  let client = BeaconClient::new(beacon_url.clone()).context("invalid URL")?;

  // create a regular beacon commit to the block header used for EVM execution
  let evm_commit =
      BeaconCommit::from_header(evm_header, &rpc_provider, beacon_url).await?;
  let mut commit_ts = evm_commit.timestamp();
  // safe unwrap: BeaconCommit::from_header checks that the proof can be processed
  let mut commit_beacon_root = evm_commit.process_proof(evm_header.seal()).unwrap();

  let mut state_commits: Vec<StateCommit> = Vec::new();

  // we assume that not more than 25% of the blocks have been skipped
  // TODO(#309): implement a more sophisticated way to determine the step size
  let step = HISTORY_BUFFER_LENGTH.to::<BlockNumber>() * 75 / 100;
  let target = commitment_header.number();

  let mut state_block = evm_header.number;
  while state_block < target {
      state_block = std::cmp::min(state_block + step, target);
      /// Veridise ...elided ///
      }
  }
```


Snippet 5.2: Snippet from method `from_headers()`

These state commits are then iterated over in the guest to ensure that the state commits exist in the Beacon chains history by validating if they exist in the ring buffer within the `BeaconRoots` contract. The way this ring buffer is managed in the `BeaconRoots` contract is that each slot is used as storage for a beacon root at every second (as a timestamp sets a root mod the `HISTORY_BUFFER_LENGTH`, which is defined to be `8191`). So, there are `8191` slots in total and each slot is used as storage for a beacon root and there is space for theoretically "27 hours" of roots. An important point to note is that these roots are not necessarily in order.

During the iteration, the step size between two successive blocks is defined to be 75% of the `HISTORY_BUFFER_LENGTH`. There is an underlying assumption that fewer than 25% of the slots from the ring buffer will be skipped. But this implementation can be brittle in practice, since there is no guarantee that the required block id actually exists in the ring buffer because of varying network speeds. If the network is running slow, then the previous block root that we want to prove exists in the buffer may be overwritten since the "27 hours" worth of time has passed and the state in the ring buffer has now been overwritten.

Impact There is **no guarantee** that the beacon root from the previous commit will still be available in the ring buffer when stepping forward. If more than 25% of roots are missed because the network is running too slow, then the ring buffer may overwrite the needed state.

Recommendation Rather than stepping forward from the execution block, the logic should be restructured to build backward from the target commitment block toward the original execution block. To do so, when at the target commit always pick the slot that the execution commit will eventually occupy. When visiting that commit, if it does not correspond to the evm header, then we know that it should be occupied by an intermediate beacon root which is atleast `8191` seconds old. Keep traversing backwards in this manner until the block with the execution commit is reached.

In this way, a block can be chosen to travel to farther than the 75% history block while also guaranteeing that eventually the beacon block corresponding to the execution commit will be reached.

Developer Response The developers have responded to the issue with the following changes:

> We have refactored `HistoryCommit::from_headers` to use a "backward chaining" approach. This new method starts from the commitment block and iteratively queries the historical state of the EIP-4788 BeaconRoots contract (using `eth_getStorageAt` to access specific historical slots) to reliably link back to the execution block's commitment. While this adds some host-side complexity to query historical storage, it directly addresses the concern when many slots are unobserved. We believe this new algorithm is more robust for ensuring the integrity of the `HistoryCommit`.

5.1.3 V-STL-VUL-003: ProofDb does not extend log_filters when extending another instance

Severity	Low	**Commit**	2c99f46
Type	Logic Error	**Status**	Fixed
File(s)	src/host/db/proof.rs		
Location(s)	ProofDb::extend()		
Confirmed Fix At	https://github.com/risc0/risc0-ethereum/pull/559, dcbf9a7		

The `ProofDb` database struct stores a number of fields used to track any accessed state by the host. This state is then used to track what data needs to be fetched to create an appropriate input for the guest application. More specifically, these fields are:

- `ProofDb.accounts`
- `ProofDb.contracts`
- `ProofDb.block_hash_numbers`
- `ProofDb.log_filters`
- `ProofDb.proofs`

The `HostEvmEnv` struct supports extending an environment with the contents of another compatible environment. This functionality works by extending the environments' underlying databases with each other. In the case of a `ProofDb`, the `extend()` logic is shown below:

```
pub(crate) fn extend(&mut self, other: ProofDb<D>) {
    extend_checked(&mut self.accounts, other.accounts);
    extend_checked(&mut self.contracts, other.contracts);
    extend_checked(&mut self.proofs, other.proofs);
    self.block_hash_numbers.extend(other.block_hash_numbers);
}
```

Snippet 5.3: `ProofDb::extend()`

While most of the fields are correctly extended by the `other` database, a notable omission is the `ProofDb.log_filters` field.

Impact Due to the omission of the `log_filters` field in `extend()`, the resulting environment from the extension will not contain the log filters of `other`. This can then impact the conversion of the environment into a `BlockInput`.

Particularly, when calling `ProofDb::receipt_proof` the extended database might not fetch the required receipts. As a result, the guest application might not have access to the data it needs when querying particular events.

Recommendation It is recommended to extend `self.log_filters` with `other.log_filters`.

Proof of Concept logs_testing.zip

The attached archive contains a simple Rust program that demonstrates this vulnerability.

It can be run with `cargo run` along with the following arguments:

- `--rpc-url` : an RPC URL that can also be set as an environment variable `RPC_URL`.
- `--fail` : if present, the program will trigger the issue and fail.

The program works by constructing 2 environments `env1` and `env2`, preflighting a view call on `env1` and preflighting an event query on `env2`. `env1` is then extended with `env2` (or the other way around if `--fail` is absent) and converted into a guest input. Finally, the guest attempts to query for the events that were preflighted in `env2`.

Developer Response The developers have fixed the issue with the following comments:

> We acknowledge the `ProofDb.log_filters` field not being extended in the previous `ProofDb::extend()` method. We agree this could lead to guest panics if event data was missing.
>
> This issue has been addressed. We replaced `extend()` with a new `merge()` method, which ensures all `ProofDb` fields, including `log_filters`, are correctly handled with compile-time safety for future modifications.

Additionally, the issue was originally reported at a higher severity, however following comments from the developers the severity of the issue was downgraded to match other reported issues:

> Regarding severity, while we acknowledge the issue, we note that the impact is a guest panic (affecting availability for specific event queries) rather than an incorrect proof generation. Our security model presumes a cooperative host for successful guest execution, and guest panics can be triggered by various straightforward input manipulations. We believe this context is relevant when comparing its severity to issues that might affect proof integrity.

5.1.4 V-STL-VUL-004: ProofDb overwrites storage tries with conflicting roots when converting to guest input

Severity	Low	**Commit**	2c99f46
Type	Logic Error	**Status**	Fixed
File(s)	`src/host/db/proof.rs`		
Location(s)	ProofDb::state_proof()		
Confirmed Fix At	https://github.com/risc0/risc0-ethereum/pull/557, aa8641a		

In order to prepare inputs for the guest program, a `ProofDb` must be converted into a `BlockInput`. This is done through the `BlockInput::from_proof_db()` method. Internally, this method retrieves a state trie and a list of storage tries from a `ProofDb`. This is done using the `ProofDb::state_proof()` method.

Within the `ProofDb`, an `accounts` mapping stores a set of accessed storage keys per contract address. Combined with the `proofs` mapping which stores storage proofs on a per address basis, the `ProofDb` maintains a set of storage nodes on a *per address basis*.

When converted into a list of storage tries in `state_proof()` however, the `ProofDb` only creates a single storage trie *per storage root hash*. This logic is shown in the snippet below:

```
pub(crate) async fn state_proof(&mut self) -> Result<(MerkleTrie, Vec<MerkleTrie>)> {
    // ...
  let mut storage_tries = B256HashMap::default();
    for (address, storage_keys) in &self.accounts {
    // if no storage keys have been accessed, we don't need to prove anything
    if storage_keys.is_empty() {
      continue;
    }

    // safe unwrap: added a proof for each account in the previous loop
    let storage_proofs = &proofs.get(address).unwrap().storage_proofs;

    let storage_nodes = storage_keys
      .iter()
        .filter_map(|key| storage_proofs.get(key))
      .flat_map(|proof| proof.proof.iter());
    let storage_trie =
      MerkleTrie::from_rlp_nodes(storage_nodes).context("storageProof invalid")?;
    let storage_root_hash = storage_trie.hash_slow();

        // [VERIDISE]: A previously computed storage trie might be overwritten in the
     line below.
    storage_tries.insert(storage_root_hash, storage_trie);
    }
    let storage_tries = storage_tries.into_values().collect();

    Ok((state_trie, storage_tries))
}
```

Snippet 5.4: Snippet from `ProofDb::state_proof()`

The crucial thing to note is that storage root hashes are *not* unique to different addresses and 2 different contracts may have the same storage root hash. Because of this, a previously computed storage trie might be overwritten in the highlighted line. While both tries share the same root hash, there is no guarantee that they would have been built from the same set of storage nodes. Therefore, the overwritten trie may have contained some storage nodes required by the guest in order to prove a certain value.

Impact If a host program were to preflight calls to different contracts that shared the same storage root hash, and the calls accessed a different set of storage nodes, then the guest program will not be able to execute both of the calls.

Recommendation When collecting storage nodes into a storage trie, it is recommended to check if an existing trie was already computed and extend in that case.

Developer Response The developers have made the following changes to fix this issue:

> We acknowledge the issue where distinct sparse storage tries with the same storage root hash (accessed via different contracts/keys) could lead to one overwriting another. This has been addressed by refactoring the logic for constructing the collection of storage tries. The updated implementation now correctly de-duplicates storage tries based on their root hash and ensures that all accessed storage proof nodes for a given root are merged into a single, comprehensive Merkle Trie. This prevents the loss of necessary proof nodes and ensures the guest has the complete data required for all preflighted accesses, even with shared storage roots. We added a test for this case.

5.1.5 V-STL-VUL-005: Implicit dependency on with_chain_spec for chain specifications

Severity	Low	**Commit**	2c99f46
Type	Data Validation	**Status**	Fixed
File(s)	`src/lib.rs`		
Location(s)	new()		
Confirmed Fix At	`https://github.com/risc0/risc0-ethereum/pull/558`, e480f20		

When the `GuestEvmEnv` is initialized using `new()`, it uses a default configuration which corresponds to Ethereum mainnet with the latest `SpecId`. To configure the execution environment for a specific block, the `with_chain_spec()` method must then be called, which derives the appropriate `SpecId` from the block header's number, timestamp and the provided `ChainSpec`. See snippets below for context.

```
impl<D, H: EvmBlockHeader, C> EvmEnv<D, H, C> {
/// Creates a new environment.
///
/// It uses the default configuration for the latest specification.
pub(crate) fn new(db: D, header: Sealed<H>, commit: C) -> Self {
    let cfg_env = CfgEnvWithHandlerCfg::new_with_spec_id(Default::default(), SpecId::
    LATEST);

    Self {
        db: Some(db),
        cfg_env,
        header,
        commit,
    }
}
```

Snippet 5.5: Snippet from method `new()`

```
pub fn with_chain_spec(mut self, chain_spec: &ChainSpec) -> Self {
  self.cfg_env.chain_id = chain_spec.chain_id();
  self.cfg_env.handler_cfg.spec_id = chain_spec
      .active_fork(self.header.number(), self.header.timestamp())
      .unwrap();
  self.commit.configID = chain_spec.digest();

  self
}
```

Snippet 5.6: Snippet from method `with_chain_spec()`

It is very easy to make a mistake here, because if `with_chain_spec()` is not explicitly called after environment creation, the execution environment may be inconsistent with the provided commitment. This also can't be checked with the commitment since it does not specify the exact `SpecId` selected.

Impact The guest environment may be executed with a chain specification which is inconsistent with the header's execution environment. Note that all invocations of `new` in the codebase does correctly set the `commitId` to the default configuration's hash.

Recommendation Consider either adding `SpecId` to the commitment *or* enforce that the the `configId` in the input commit is consistent with the default configuration.

Developer Response The developers have made the following changes to address the issue:

> We have reviewed finding V-STL-VUL-005, which highlighted the previous implicit dependency on `with_chain_spec()` and the potential for environment inconsistencies if it was not called, leading to reliance on a default chain specification.This concern has been addressed through a comprehensive refactor of how chain specifications are handled. The `with_chain_spec()` method has been removed. Instead:
>
> 1. The `ChainSpec` is now a **mandatory parameter** provided via a `.chain_spec()` method on the environment builder *before* the environment is built.
> 2. The `ChainSpec` is also **required** when converting an input back into an environment in the guest (e.g., via `EvmInput::into_env(&ChainSpec)`).
> 3. The `Default` implementation for `ChainSpec` has been removed, ensuring a specification is always explicitly chosen.
>
> These changes make the chain configuration explicit throughout the entire lifecycle of an environment, from host construction to guest execution, thereby eliminating the possibility of the previously identified implicit dependency and potential inconsistencies.

5.1.6 V-STL-VUL-006: Code quality improvements

Severity	Info	**Commit**	2c99f46
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/risc0/risc0-ethereum/pull/560`, 16a1b1a		

There are a number of minor code quality issues that were uncovered in the audit. They are listed per file below:

- `crates/steel/src/lib.rs`: In the documentation of the `Commitment.id` field, it is not mentioned how the version and identifier of the claim are combined. It would be helpful to document what the format of the ID is (i.e. the one used by `decode_id()`).
- `crates/steel/src/host/db/proof.rs`: lines 204-213 could be simplified by a call to `self.get_proof()` as is done on line 135.

Impact There is no security impact, however it is recommended to fix them in order to help future maintainability of the codebase.

Recommendation It is recommended to address the listed issues.

Developer Response The developers have added documentation for the first point and acknowledged the second issue, as the Rust borrowing rules complicate the second recommended change:

> We have addressed the first point by improving the documentation for the `Commitment.id` field. The documentation now explicitly details how the version and identifier are combined and references the `decode_id()` format, as suggested. Regarding the suggested simplification in `crates/steel/src/host/db/proof.rs` (lines 204-213), we investigated this. While a call to `self.get_proof()` would indeed seem simpler at first glance, it is not trivially possible in the current context due to Rust's borrowing rules. Specifically, `self.proofs` is already mutably borrowed at that point, preventing an immutable borrow of `self` needed for `get_proof()`. While alternative refactoring approaches exist, we have opted to leave this specific section as is for now, given the non-trivial nature of a change that wouldn't alter functionality.

6 Fuzz Testing

6.1 Methodology

One of the goals of the security assessment was to fuzz test Steel to identify any denial-of-service (DoS) related issues and over-constrained behaviors within the guest through automated testing and comparison of the guest and host execution. To that end, the Veridise security analysts developed a custom fuzzing tool to fuzz the project, which iterates over a list of contracts deployed on Ethereum, and executes the view functions of each contract on random inputs. The execution output within the guest environment is then compared against the output of execution within the host environment (during the pre-flighting) to ensure consistency.

6.2 Properties Fuzzed

The Veridise team devoted a total of 24 compute-hours to fuzzing this protocol, with an aim to validate the following properties:

- The guest is not over-constrained. When executing a view call on a guest environment, it returns the same output as the host if the view call is successful and the same error message if the transaction is reverted.
- No potential crash states. The project does not crash when making a view call within a guest environment, barring rpc related issues or out of gas errors (due to the gas configuration).

During the fuzzing process, the Veridise team performed view calls on over *2,175* contracts on Ethereum and did not detect any bugs.

Glossary

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. 20

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