Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

RISC ZERO

Steel

Veridise Inc.

Oct 4, 2024

► **Prepared For:**

RISC Zero
https://risczero.com/

► **Prepared By:**

Jon Stephens
Mark Anthony
Nicholas Brown

► **Contact Us:**

contact@veridise.com

► **Version History:**

Oct 2, 2024	V1
Oct 4, 2024	V2

Contents

1 Executive Summary

From Sep. 16, 2024 to Sep. 26, 2024, RISC Zero engaged Veridise to conduct a security assessment of their Steel library. The security assessment covered the Steel library which allows developers to create proofs about the state of Ethereum off-chain by using the result of queries made with the library. Veridise conducted the assessment over 24 person-days, with 3 security analysts reviewing the project over 8 days on commit `649289a2`. The review strategy involved a thorough code review of the program source code performed by Veridise security analysts.

Project Summary. The security assessment covered the RISC Zero Steel library which allows developers to prove the execution of queries made on Ethereum state, at a particular block. The Steel library can be divided into two main components, the host and the guest. The host is the untrusted portion of the Steel library which constructs the EVM environment and provides it to the guest as input. It also performs certain preflight operations to prepare the input for execution in the guest.

The guest performs the necessary validations on the input provided by the host and executes the view call within the context of the prepared EVM environment. It also creates a commitment to the view call environment in which the execution takes place, which must be validated alongside the validation of the Steel proof. Developers can choose to either commit to a block hash or to a beacon block root for the block validation.

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 a README and documentation comments on functions and storage variables. To facilitate the Veridise security analysts understanding of the code, the Steel developers shared several examples which demonstrated the use of the Steel library. The source code contained a test suite, which the Veridise security analysts noted tested most workflows and included negative tests as well.

Summary of Issues Detected. The security assessment uncovered 5 issues, 1 of which is assessed to be of a medium severity by the Veridise analysts. Specifically, V-STL-VUL-001 details how omitting details of the execution environment from the commitment can make it difficult for users to determine if the guest execution is realistic. The Veridise analysts also identified 1 low-severity issue , V-STL-VUL-002 which notes that the use of a hardcoded index to access the blockhash can cause issues if in the future more fields are added to the BeaconBlockBody, as well as 2 warnings, including V-STL-VUL-004 which outlines why the REVM version should be included in the commitment to accurately reflect the correct output of transaction execution, and 1 informational finding .

Among the 5 issues, 4 issues have been fixed by RISC Zero and 1 issue has been determined to be intended behavior after discussions with RISC Zero.

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

The security analysts recommend adding information about the execution environment and the environment configuration to the commitment to make guest execution easily verifiable for a user of the library. The analysts also recommend adding tests which test successful guest execution with commitment to a beacon block root.

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	`649289a2`	Rust	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Sep. 16–Sep. 26, 2024	Manual	3	24 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	0	0	0
Medium-Severity Issues	1	1	1
Low-Severity Issues	1	1	1
Warning-Severity Issues	2	1	1
Informational-Severity Issues	1	1	1
TOTAL	5	4	4

Table 2.4: Category Breakdown.

Name	Number
Data Validation	3
Maintainability	2

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

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

- Does the host construct the evm environment correctly?
- Does the guest perform the necessary validations during transaction execution?
- Does the commitment contain the required information regarding the guest execution environment?
- During guest execution, is the construction and management of the relevant state and storage tries performed correctly?
- Can future upgrades to Ethereum have any effect on the Steel library?
- Is it possible to prove the execution of a transaction that is canonically false?
- Are appropriate validations done in the guest pertaining to the beacon block root commitment and the execution block hash commitment?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the audit involved a thorough manual review of the protocol by human experts.

Scope. The scope of this security assessment is limited to the (`risc0-ethereum/steel/src`) folder of the source code provided by the Steel developers, which contains the implementation of the Steel library.

Methodology. Veridise security analysts inspected the provided tests, went through the provided examples and read the Steel documentation. They then began a manual review of the code.

During the security assessment, the Veridise security analysts regularly communicated with the Steel developers to ask questions about the code. The Veridise security analysts also used the ethereum specifications as a reference during the security assessment.

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.

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

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

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

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

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 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 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-STL-VUL-001	Commitment Should Include Execution . . .	Medium	Fixed
V-STL-VUL-002	Generalized Index for Beacon Block Hash . . .	Low	Fixed
V-STL-VUL-003	Execution Failure Should be Handled . . .	Warning	Fixed
V-STL-VUL-004	Commitment Should Include REVM Version	Warning	Intended Behavior
V-STL-VUL-005	Missing Documentation	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-STL-VUL-001: Commitment Should Include Execution Environment

Severity	Medium	**Commit**	649289a
Type	Data Validation	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	N/A		

The configuration of REVM in the guest program can be essential to ensuring that a transaction is executed correctly. As it stands, Steel allows developers to define the specifications of REVM configurations at different points in time as well as conditions that determine when different specifications can be used. These specifications are not included in Steel's default commitment and therefore are left within the purview of the guest executable. Over time, however, it is likely that a guest execution's chain specifications will become outdated as new versions of the blockchain are released. When this occurs, users need a way of determining whether executions are made under the correct operating conditions. Additionally, in cases where users of the steel library provide their own chain configurations, it might be difficult to determine whether the chain configurations are tampered with, particularly if the project is closed-source.

```
1    // Read the input from the guest environment.
2    let input: EthEvmInput = env::read();
3
4    // Converts the input into a `EvmEnv` for execution. The `with_chain_spec` method
      is used
5    // to specify the chain configuration. It checks that the state matches the state
      root in the
6    // header provided in the input.
7    let env = input.into_env().with_chain_spec(&ETH_SEPOLIA_CHAIN_SPEC);
```

Snippet 4.1: Snippet from example guest implementation

Impact A transaction could be executed under a configuration that does not correspond to a realistic execution environment. With the current model, it can be difficult for users to determine if this execution is realistic.

Recommendation Include information about the environment configuration and chain id in the commitment to make validation of the guest execution environment more straightforward. This could be done by hashing the important fields (like the chain id and the fork) of the environment configuration and including this hash in the commitment so that common execution environments can easily be identified. Proofs could then be easily checked to ensure that execution happened using an accepted configuration.

Developer Response The developers have added a hash of the chain id and the fork configuration to the commitment.

4.1.2 V-STL-VUL-002: Generalized Index for Beacon Block Hash Should Be More Robust

Severity	Low	**Commit**	649289a
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	N/A		

During the course of the audit, the RISC Zero developers notified the Veridise team that they had found a vulnerability with the `beacon` portion of the Steel library. This vulnerability would allow users to generate proofs about arbitrary block data. In essence, the `block_hash` of the block was validated by ensuring it was present in the Merkle Tree for the beacon block. However, the index of the leaf in the Merkle Tree was allowed to be set by the host. This meant that the value in a different field of the beacon block could be treated as the `block_hash`. Since beacon blocks contain a `graffiti` field that can contain arbitrary data set by the block proposer, this could allow the block proposer to generate a Steel proof about a block with arbitrary data.

The Steel developers resolved this issue by setting a fixed `GeneralizedIndex` for the `block_hash` which will ensure that the correct leaf index is used in the Merkle proof. However, problems could still arise in the future if the structure of the beacon block changes because the index of the `block_hash` field could change.

```
1 /// The generalized Merkle tree index of the `block_hash` field in the `BeaconBlock`
2 pub const BLOCK_HASH_LEAF_INDEX: merkle::GeneralizedIndex = merkle::GeneralizedIndex
      ::new(6444);
```

Snippet 4.2: Definition of the `GeneralizedIndex` in the fix PR for this issue

Impact The beacon block is encoded using the SSZ format which allows the data-structure to be merklized. This means that as changes are made to the beacon block format, the underlying merkle tree will also change. Most notably, if any struct in the datastructure grows too large, the depth of the tree may increase. So far, changes have continued to be made to the beacon block format by adding new fields to the BeaconBlockBody, which could result in different indexes for the block hash in the future. With Risc0's current solution, if this were to occur eventually the `BLOCK_HASH_LEAF_INDEX` would read the wrong hash.

Recommendation Include a more robust way for configuring the `GeneralizedIndex` rather than hard coding it to make the value easier to update, if the structure of the beacon block changes.

Developer Response We have made the Generalized Index more configurable by introducing a new type that contains the leaf index as a const generic. This allows us to more easily adapt to protocol (and thus index) changes.

4.1.3 V-STL-VUL-003: Execution Failure Should be Handled Automatically

Severity	Warning	**Commit**	649289a
Type	Data Validation	**Status**	Fixed
File(s)	`contract.rs`		
Location(s)	CallBuilder::call		
Confirmed Fix At	N/A		

Steel allows users to prove things about EVM executions by simulating a transaction or series of transactions in the guest environment. Transactions are executed using a `CallBuilder` object that allows the guest program to specify relevant parameters of the call. Once the `CallBuilder` is properly initialized, the guest program will call the `call()` function to simulate the transaction. When the guest calls this function, a `Result` is returned. In the examples, this `Result` is unwrapped by the guest, so the program will panic if the call fails. However, there is no guarantee that this result will be handled properly by the guest program.

```
    pub fn call(self) -> Result<C::Return, String> {
        // safe unwrap: env is never returned without a DB
        let state_db = self.env.db.as_ref().unwrap();
        let mut evm = new_evm::<_, H>(
            WrapStateDb::new(state_db),
            self.env.cfg_env.clone(),
            self.env.header.inner(),
        );
        self.tx.transact(&mut evm)
    }
```

Snippet 4.3: Definition of `CallBuilder::call()`

Impact Users who are unfamiliar with Rust may forget to `unwrap()` the result when they are merely trying to prove that the call succeeds. This will result in the program succeeding even if the transaction reverts. This could cause major problems with their proofs and it may not be immediately obvious what is wrong when looking at the guest code.

Recommendation The status of the execution should be handled automatically to ensure that the above situation doesn't occur. This could be done in a number of ways. Here are two different potential suggestions:

- Include the status of a call in the commitment by default when a call is made using the `CallBuilder`

or

- Update the API of the call builder in the following way:
  - `call()` should be renamed to `call_unsafe()` and documentation for this function should indicate that the program needs to handle execution failures if this function is used.
  - The new `call()` function will unwrap the result of `call_unsafe()` and return the result

Validating the execution status by default will make the library more user friendly and less prone to errors.

Developer Response The developers have updated the API of the call builder to match the second recommendation with `try_call()` being used as the name of the method that returns the Result object.

4.1.4 V-STL-VUL-004: Commitment Should Include REVM Version

Severity	Warning	**Commit**	649289a
Type	Data Validation	**Status**	Intended Behavior
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	N/A		

When a guest program is executed with RISC0, an identifier for the executable is included in the resulting proof. The steel library currently relies on the assumption that this identifier will be sufficient to identify information like the REVM version used as this is encoded directly in the guest program. Particularly in cases where the guest is not open-sourced though, this information may be difficult to determine. Additionally, it could be critical to determining the validity of the result in cases where a buggy or outdated version of REVM is used.

Impact If outdated or buggy versions of REVM are used, proofs generated using the guest program may not accurately reflect the correct output of the transaction.

Recommendation The version of REVM should be included in the commitment to allow users to verify that the version is up to date.

Developer Response The developers indicated that guest code is intended to be open source and the `Cargo.lock` file can be used to verify the versions of packages like REVM.

4.1.5 V-STL-VUL-005: Missing Documentation

Severity	Info	**Commit**	649289a
Type	Maintainability	**Status**	Fixed
File(s)	`beacon.rs`		
Location(s)	See issue description		
Confirmed Fix At	N/A		

There are a couple of locations within the code base that would benefit from additional documentation:

- `beacon.rs`
  - line 256:
    - This implementation assumes that the Deneb version of the beacon chain is the current version.
    - There should be documentation for users somewhere that indicates that this version is the expected version of the beacon chain and if the beacon chain is updated in the future, users will need to update their version of this library to support new fields if they are added.
  - line 283:
    - This loop iterates over the 32 slots after the `parent_slot` to look for the next block. Presumably 32 was chosen because it is highly unlikely that there would be 32 slots in a row without a block proposal. However, the reasoning behind choosing 32 should be documented to make the code more readable and aid the understanding of future developers.

Impact Future developers and users of this library may make mistakes if they aren't aware of this information

Recommendation Add documentation to the above locations to improve maintainability and usability of the library

Developer Response The developers have added documentation for the first point and have updated their Beacon block process, which makes the second point no longer necessary.

