Audit of RiscZero Helios

Date: April 2, 2025

Introduction

On April 2, 2025, RiscZero engaged zkSecurity to perform a short review of “R0VM Helios”. The audit focused on two sides of the same application:

1. Rust logic to produce Ethereum consensus proofs using the RiscZero’s zkVM.
2. Solidity logic to run an on-chain Ethereum light client on arbitrary EVM chains.

The **scope** specifically targeted the pull request https://github.com/risc0/r0vm-helios/pull/2/files which included:

- A new `R0VMHelios.sol` smart contract to be deployed on a destination EVM chain.
- R0VM code making use of Helios to verifiably advance the Ethereum sync committee state and the header it points to, as well as exposing an arbitrary number of storage slot proofs for the finalized header. This code included both a guest program to run inside the zkVM, and a host program to produce the private inputs to send to the guest program.

Out of scope were:

- Core R0VM code, including risc0-ethereum.
- The Helios light client itself, although some time was spent to understand if the API was correctly used by the guest R0VM program.

Note that the audit primarily focused on the protocol’s soundness rather than its liveness. More concretely, we focused on the guest program correctness, on the verification of R0VM proofs on-chain, and on the correctness and access controls of the light client smart contract. On the other hand, we overlooked glue code and operating tools (especially as the operator code had `todo!()` placeholders that prevented it to be run).

We observed that the documentation for the Helios light client was limited, which may reflect the current developmental stage of the protocol. This was also echoed in findings Next Sync Committee Might Be Arbitrarily Set and Lack of Assurance on Verified Chain Identity.

Given the short time frame of our review, we **recommend** a more comprehensive audit of the Helios light client to thoroughly evaluate its security guarantees and verify that the assumptions underpinning the R0VM Helios program are valid

Overview of R0VM Helios

From the R0VM document:

> *R0VM Helios verifies the consensus of a source chain in the execution environment of a destination chain. For example, you can run an R0VM Helios light client on Polygon that verifies Ethereum Mainnet's consensus.*

As stated above, there are two sides to this application: the production of consensus proofs locally (by operators) and their verification on-chain to maintain the state of a light client. We survey both sides in the sections below.

R0VM Helios Guest Program

An R0VM program is split into two parts: the guest program which runs inside the zkVM, and the host program which runs the zkVM and produces the private inputs to send to the guest program.

The guest program heavily relies on helios (an Ethereum light client) and alloy (mostly for Merkle proofs). The light client follows the Altair specification which introduced verifiable sync committee updates for light clients in Ethereum.

The logic of the guest program performs the following steps:

1. Deserialize Inputs. It reads private inputs sent by the host and initializes the light client state with it. The private inputs also contain a series of light client updates, as well as one finality update.

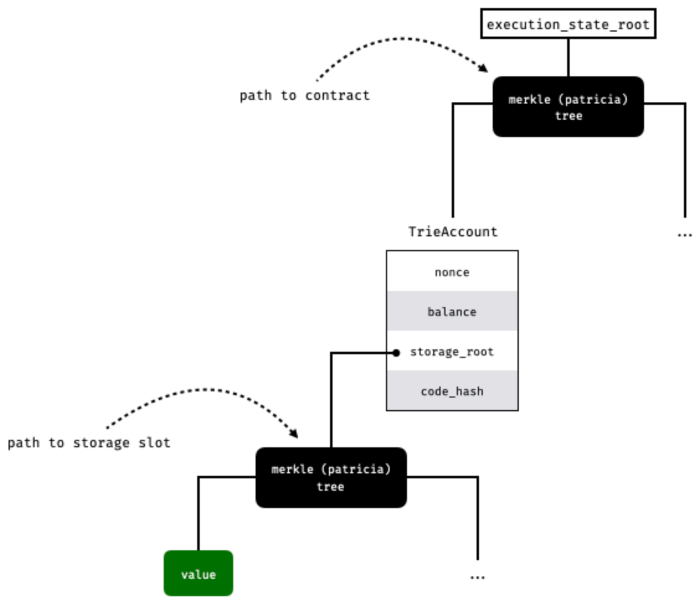
2. Process Sync Committee Updates. It iterates through a series of light client updates, verifying and applying each update sequentially to the light client state. This produces a sync committee that can verify the finality update.

3. Apply Finality Update. The finality update is verified and applied to the light client state. The finalized header, the current sync committee, and the next sync committee are extracted from the finalized state.

4. Verify Storage Slot Proofs. The finalized header's state root is used to prove (using Merkle proofs) a number of arbitrary storage accesses on its post state.

5. Commit New State Outputs. The starting and ending states that comprised the proven state transition, information on the next sync committee, as well as the values read from the finalized Ethereum post state are committed to the journal (which is R0VM's term for exposing variables in the public input).

The storage accesses are proven using Merkle proofs on the authenticated state root, as illustrated in the diagram below.



We can categorize the public input data into three main groups:

Previous Light Client State.

- `prevHeader`: the header used to kickstart the state transition.
- `prevHead`: the head used to kickstart the state transition.
- `startSyncCommitteeHash`: a digest of the sync committee used to kickstart the state transition.

New Light Client State.

- `executionStateRoot`: the post-state root of the finalized header.
- newHead: the block number of the finalized block
- `newHeader`: the root of the state merkle tree of the finalized header.
- `syncCommitteeHash`: a digest of the sync committee in the sync period of the finalized header.
- `nextSyncCommitteeHash`: same but for the next sync period.

Post-State Storage Accesses.

- `slots`: an arbitrary number of storage slots accessed on the update post-state.

The on-chain light-client

The on-chain light client is responsible for maintaining and updating the consensus state of a source chain by verifying proofs produced off-chain. Its design closely follows the Altair light client specification and leverages several key components:

Access Control. The smart contract relies on OpenZeppelin's AccessControlEnumerable to manage access control to the contract's functionalities.

Proof Verification. The smart contract relies on RiscZero's own IRiscZeroVerifier contract to verify zkVM proofs submitted to the contract.

The contract is initialized with a set of "updaters" that are the only entities capable of updating the state of the on-chain light client.

The result of updates are provided and stored in the contract under different mappings, and the updates themselves are verified by verifying the R0VM proofs as can be seen below:

```
function update(bytes calldata seal, bytes calldata journalData, uint256 fromHead)
    external
    onlyRole(UPDATER_ROLE)
{
    // TRUNCATED...

    IRiscZeroVerifier(verifier).verify(seal, heliosImageID, sha256(journalData));
```

In addition, the prover can choose a number of storage slots accesses in the post-state, and these get stored in a mapping as well (recording selected storage slots and their values at specific block numbers).

Findings

Below are listed the findings found during the engagement. High severity findings can be seen as so-called "priority 0" issues that need fixing (potentially urgently). Medium severity findings are most often serious findings that have less impact (or are harder to exploit) than high-severity findings. Low severity findings are most often exploitable in contrived scenarios, if at all, but still warrant reflection. Findings marked as informational are general comments that did not fit any of the other criteria.

ID	COMPONENT	NAME	RISK
#00	r0vm	Lack of Assurance on Verified Chain Identity	**Medium**
#01	r0vm	Next Sync Committee Might Be Arbitrarily Set	**Medium**
#02	contract	Contract Lacks Recovery Mechanism	**Low**
#03	contract	Updaters Can't Be Rotated	**Low**
#04	contract	Ambiguous Storage Key Encoding	**Informational**
#05	contract	Contract Design And Implications	**Informational**
#06	contract	Improve Light Client Root-of-Trust Transparency	**Informational**

#00 - Lack of Assurance on Verified Chain Identity

Severity: Medium **Location:** r0vm

Description. In the R0VM guest program, a series of updates is applied to a light client state in order to transition it to a newer state.

To ensure that the sync committee updates originate from the correct blockchain (and fork) the genesis block and the expected fork IDs are passed to the Helios verification functions:

```rust
pub fn main() {
    let encoded_inputs = env::read_frame();

    let ProofInputs {
        // TRUNCATED...
        genesis_root,
        forks,
        // TRUNCATED...
    } = serde_cbor::from_slice(&encoded_inputs).unwrap();

    // TRUNCATED...

    // 1. Apply sync committee updates, if any
    for (index, update) in sync_committee_updates.iter().enumerate() {
        // TRUNCATED...
        let update_is_valid =
            verify_update(update, expected_current_slot, &store, genesis_root,
&forks).is_ok();
        // TRUNCATED...
    }

    // 2. Apply finality update
    let finality_update_is_valid = verify_finality_update(
        &finality_update,
        expected_current_slot,
        &store,
        genesis_root,
        &forks,
    )
    .is_ok();

    // TRUNCATED...

    let proof_outputs = ProofOutputs {
        executionStateRoot: execution_state_root,
        newHeader: header,
        nextSyncCommitteeHash: next_sync_committee_hash,
        newHead: U256::from(head),
        prevHeader: prev_header,
        prevHead: U256::from(prev_head),
        syncCommitteeHash: sync_committee_hash,
        startSyncCommitteeHash: start_sync_committee_hash,
        slots: verified_slots,
    };
    env::commit_slice(&proof_outputs.abi_encode());
}
```

In turn, Helios will use the genesis root and the fork ID to verify that the sync committee signature is valid:

```
let fork_version = calculate_fork_version::<S>(forks,
update.signature_slot.saturating_sub(1));
    let fork_data_root = compute_fork_data_root(fork_version, genesis_root);
    let is_valid_sig = verify_sync_committee_signature(
        &pks,
        update.attested_header.beacon(),
        &update.sync_aggregate.sync_committee_signature,
        fork_data_root,
    );

    if !is_valid_sig {
        return Err(ConsensusError::InvalidSignature.into());
    }
```

It is crucial for the light client to correctly identify the chain it is syncing to. Therefore, these two values must be exposed in the journal.

Recommendation. Expose these values, and enforce that they are equal to the expected values in the on-chain light client solidity implementation.

Right now the on-chain smart contract sets `GENESIS_VALIDATORS_ROOT` and `SOURCE_CHAIN_ID` at deployment and never use them later on.

#01 - Next Sync Committee Might Be Arbitrarily Set

Severity: Medium **Location:** r0vm

Description. The initial store is completely decided by the prover, as it is passed to the guest program via private inputs. It is not completely unconstrained though, as a number of fields are exposed as public outputs and verified to be consistent with the on-chain light client.

Still, some fields that are not exposed as public outputs might pose some problems depending on how they are handled by the Helios implementation. For example, this is the case with the initial store value of the `next_sync_committee` field that's not exposed as a public output. This could lead to a situation where the prover sets the `next_sync_committee` to an invalid value during a sync committee update.

Note that sync updates might not necessarily carry a next sync committee with them, as might be implied by its type. This is because they are first converted to `GenericUpdate`, and during the conversion a default value will be interpreted as field that is not set. You can see this in `/ethereum/consensus-core/src/types/mod.rs` in Helios:

```
impl<S: ConsensusSpec> From<&Update<S>> for GenericUpdate<S> {
    fn from(update: &Update<S>) -> Self {
        Self {
            attested_header: update.attested_header().clone(),
            sync_aggregate: update.sync_aggregate().clone(),
            signature_slot: *update.signature_slot(),
            next_sync_committee:
default_to_none(update.next_sync_committee().clone()),
            next_sync_committee_branch:
default_branch_to_none(update.next_sync_committee_branch()),
            finalized_header:
default_header_to_none(update.finalized_header().clone()),
            finality_branch: default_branch_to_none(update.finality_branch()),
        }
    }
}
```

Recommendation. There are a few ways to address this issue besides exposing the `next_sync_committee` in the public output and verifying its consistency with what is on chain. Instead, one could ensure that all initial values in a light client store are set to mimic the bootstrapping process:

```rust
pub fn apply_bootstrap<S: ConsensusSpec>(
    store: &mut LightClientStore<S>,
    bootstrap: &Bootstrap<S>,
) {
    *store = LightClientStore {
        finalized_header: bootstrap.header().clone(),
        current_sync_committee: bootstrap.current_sync_committee().clone(),
        next_sync_committee: None,
        optimistic_header: bootstrap.header().clone(),
        previous_max_active_participants: 0,
        current_max_active_participants: 0,
        best_valid_update: None,
    };
}
```

But this could lead to issues if no update can be applied to set the `next_sync_committee`. Another solution could be to expose simply expose the entire store of the light client, while this seems like a much stronger solution this would force on-chain updaters to replicate the exact state of the light client, which might not necessarily by straightforward.

#02 - Contract Lacks Recovery Mechanism

Severity: Low **Location:** contract

Description. The light client smart contract lacks a mechanism to recover if it becomes stuck due to a lack of updates for a specific (1 week) period of time. This is due to the following hardcoded check in the contract:

```
/// @notice Maximum number of time behind current timestamp for a block to be used
for proving
/// @dev This is set to 1 week to prevent timing attacks where malicious
validators
/// could retroactively create forks that diverge from the canonical chain. To
minimize this
/// risk, we limit the maximum age of a block to 1 week.
uint256 public constant MAX_SLOT_AGE = 1 weeks;

// TRUNCATED...

function update(bytes calldata seal, bytes calldata journalData, uint256 fromHead)
    external
    onlyRole(UPDATER_ROLE)
{
    // TRUNCATED...

    // Check if the head being proved against is older than allowed.
    if (block.timestamp - slotTimestamp(fromHead) > MAX_SLOT_AGE) {
        revert PreviousHeadTooOld(fromHead);
    }
```

This could happen either if block production on the source chain fell below a certain threshold (e.g., no block exceeds the 2/3 majority within a day or the system lags for over a week) or if no updaters provided updates for over a week.

Recommendation. Consider the risk and the likelihood of such an event happening. If needed, implement a fallback mechanism that allows an operator/updater to restart the contract using a new trusted starting point within a weak subjectivity period.

#03 - Updaters Can't Be Rotated

Severity: Low **Location:** contract

Description. In the `R0VMHelios.sol` smart contract, that implements the on-chain Ethereum light client, only a privileged set of updaters can update the state of the light client.

```
function update(bytes calldata seal, bytes calldata journalData, uint256 fromHead)
    external
    onlyRole(UPDATER_ROLE)
{
    // TRUNCATED...
}
```

This set of updaters is decided at deployment time and can’t be changed afterwards. This is done using the OpenZeppelin’s `AccessControlEnumerable` contract and by setting an inexistant role (`bytes32(0)`) as administrator of the updater role:

```
// TRUNCATED...
struct InitParams {
    // TRUNCATED...
    address[] updaters;
}

constructor(InitParams memory params) {
    // TRUNCATED...
    if (params.updaters.length == 0) {
        revert NoUpdatersProvided();
    }

    // Make UPDATER_ROLE not have any admin roles that can manage it
    // This freezes the updater set - no role can add or remove updaters
    _setRoleAdmin(UPDATER_ROLE, bytes32(0));

    // Add all updaters
    for (uint256 i = 0; i < params.updaters.length; i++) {
        address updater = params.updaters[i];
        if (updater != address(0)) {
            _grantRole(UPDATER_ROLE, updater);
            emit UpdaterAdded(updater);
        }
    }
}
```

Due to this, loss of keys or compromise of updaters will negatively impact the smart contract.

Recommmendation. Best practice is to give options for updaters to be rotated.

#04 - Ambiguous Storage Key Encoding

Severity: Informational **Location:** contract

Description. As we have seen throughout this report, the on-chain light client uses RiscZero to process verified updates to its state. However these updates also come with arbitrary verified storage slot accesses. Each of these accesses is stored in a mapping (`storageValues`), and can be accessed via a deterministically generated access key.

The access key encodes three elements that uniquely identify the storage slot: the contract address at which the storage slot was accessed, the storage slot number, and the block number at which the access was made.

The encoding used comes from the Solidity ABI specification and packs the arguments as follows:

```
function computeStorageKey(uint256 blockNumber, address contractAddress, bytes32
slot)
    public
    pure
    returns (bytes32)
{
    return keccak256(abi.encodePacked(blockNumber, contractAddress, slot));
}
```

According to the specification says, the encoding is **ambiguous**:

In general, the encoding is ambiguous as soon as there are two dynamically-sized elements, because of the missing length field.

As the arguments encoded in the implementation are all fixed-length (with the address being interpreted as a `uint160` type), this is not an issue. That being said it is worth noting that this can lead to issues if the types of the arguments change in the future, or if clients do not properly handle truncated arguments (as seen in the past with short-address attacks).

#05 - Contract Design And Implications

Severity: Informational **Location:** contract

Description. We noticed two unusual behaviors in the way the on-chain light client was implemented. After discussion with RiscZero, it seems like these oddities are due to the original permissionless design of the contract: anyone could attempt to update the on-chain light client state. This design was later changed, and the contract we looked at was permissioned as explained in the finding Updaters Can’t Be Rotated.

Non-Sequential Updates. The contract does not enforce that updates must begin with the most recent head/header (which is stored in the head field of the smart contract state). In other words, an update may use an older state as its starting point. This approach can be inefficient and may not reflect the optimal state progression.

Redundant Event Emissions. Obsolete updates that do not actually advance the light client state can be submitted repeatedly. While these redundant updates do not compromise security, they cause the contract to emit the same events (e.g., HeadUpdate) multiple times. This behavior could confuse users, particularly if their client software does not handle out-of-order or duplicate events gracefully.

Although these behaviors do not pose security risks, they can lead to inefficiencies and potential bugs in client applications that assume events are unique and sequential.

#06 - Improve Light Client Root-of-Trust Transparency

Severity: Informational **Location:** contract

Description. The on-chain light client is initialized using a trusted starting point (or “weak subjectivity checkpoint”) in its constructor, as shown below:

```
contract R0VMHelios is AccessControlEnumerable {
    constructor(InitParams memory params) {
        GENESIS_VALIDATORS_ROOT = params.genesisValidatorsRoot;
        GENESIS_TIME = params.genesisTime;
        // TRUNCATED...
        SOURCE_CHAIN_ID = params.sourceChainId;
        syncCommittees[getSyncCommitteePeriod(params.head)] =
params.syncCommitteeHash;
        // TRUNCATED...
        headers[params.head] = params.header;
        executionStateRoots[params.head] = params.executionStateRoot;
        head = params.head;
        // TRUNCATED...
    }
```

This method is common practice to prevent long-range attacks by anchoring the light client to a known checkpoint. However, the design has two debatable issues:

Potential Disconnection Between Initialization Values. The various initial values such as the sync committee hash, header, and execution state root are not explicitly verified to be interrelated. This lack of cross-verification may result in a situation where these values do not consistently represent a single, cohesive checkpoint.

Opaque Root-of-Trust Identification. There is no clear, on-chain mechanism to indicate which checkpoint serves as the root of trust for the light client. Users must currently rely on off-chain observation, such as retrieving the first `HeadUpdate` event, to determine the initial root of trust. This indirect method can hinder transparency and independent verification.