Zellic

Prepared for
Nathan
Uma
Succinct Labs

Prepared by
Junghoon Cho
Jaeeu Kim
Zellic

August 7, 2025

SP1 Helios
Smart Contract Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Succinct Labs from July 18th to July 29th, 2025. During this engagement, Zellic reviewed SP1 Helios's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is the execution-state root constrained correctly with the current usage of Helios?
- Can a malicious actor brick the contract to prevent further on-chain verification from proceeding?
- Is the verification of a specific storage slot thoroughly enforced?
- Is the SP1Helios contract initialized correctly?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped SP1 Helios contracts, we discovered five findings. One critical issue was found. One was of medium impact and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Succinct Labs in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	**Count**
🟥 Critical	1
🟧 High	0
🟨 Medium	1
🟩 Low	0
⬜ Informational	3

2. Introduction

2.1. About SP1 Helios

Succinct Labs contributed the following description of SP1 Helios:

> SP1 Helios proves the verification of the light client sync protocol, as well as storage proofs of committed state roots.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding’s impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and

Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

SP1 Helios Contracts

Types	Solidity, Rust
Platform	EVM-compatible

Target	sp1-helios
Repository	https://github.com/succinctlabs/sp1-helios ↗
Version	`51b1e4aaee2e3e614dd589b1fa83594aa7b528b6`

Programs	`contracts/src/SP1Helios.sol`
	`primitives/src/lib.rs`
	`primitives/src/types.rs`
	`program/src/light_client.rs`
	`program/src/storage.rs`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.5 person-weeks. The assessment was conducted by two consultants over the course of one calendar week.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Junghoon Cho
Engineer
junghoon@zellic.io

Jaeeu Kim
Engineer
jaeeu@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

July 18, 2025	Start of primary review period
July 22, 2025	Kick-off call
July 29, 2025	End of primary review period

3. Detailed Findings

3.1. No test suite is in place

Target	contracts/src/SP1Helios.sol, program/src/light_client.rs, program/src/storage.rs		
Category	Code Maturity	**Severity**	Critical
Likelihood	N/A	**Impact**	Critical

Description

The project lacks a test suite for both the verifier contract SP1Helios.sol and the SP1 Helios programs under the `/program` directory, which are used to generate ZK proofs. From integration tests for Helios with a node setup that supplies Beacon chain data to basic unit tests for core functions, these tests are critical to eliminate potential security risks in the project.

Impact

When building a complex contract ecosystem with multiple moving parts and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios. Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

Good test coverage has multiple effects.

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization (e.g., gas cost).
- It displays code maturity.
- It bolsters customer trust in your product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

The last point seems contradictory, given the time investment to create and maintain tests. To expand upon that, tests help developers trust their own changes. It is difficult to know if a code refactor — or even just a small one-line fix — breaks something if there are no tests. This is especially true for new developers or those returning to the code after a prolonged absence. Tests have your back here. They are an indicator that the existing functionality most likely was not broken by your change to the code.

Due to the absence of a test suite, it can be difficult to verify whether the implemented functionalities in the project work exactly as intended by the developer, which may potentially lead to malfunctions.

Additionally, as the project evolves and its complexity increases, it will become more difficult to detect potential bugs that may arise during future expansion and maintenance.

Recommendations

If possible, a test suite that covers all integrations and functionalities within the project should be implemented. We recommend implementing at minimum mock-based unit tests for fundamental coverage.

Remediation

This issue has been acknowledged by Succinct Labs.

3.2. Incorrect state-variable initialization in the `constructor`

Target	contracts/src/SP1Helios.sol		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The constructor of the SP1Helios contract initializes the contract using the fields of the passed `InitParams` struct.

```
constructor(InitParams memory params) {
    GENESIS_VALIDATORS_ROOT = params.genesisValidatorsRoot;
    GENESIS_TIME = params.genesisTime;
    SECONDS_PER_SLOT = params.secondsPerSlot;
    SLOTS_PER_PERIOD = params.slotsPerPeriod;
    SLOTS_PER_EPOCH = params.slotsPerEpoch;
    SOURCE_CHAIN_ID = params.sourceChainId;
    syncCommittees[getSyncCommitteePeriod(params.head)]
    = params.syncCommitteeHash;
    lightClientVkey = params.lightClientVkey;
    storageSlotVkey = params.storageSlotVkey;
    headers[params.head] = params.header;
    executionStateRoots[params.head] = params.executionStateRoot;
    head = params.head;
    verifier = params.verifier;
    guardian = params.guardian;
}
```

While the contract initialization is generally performed correctly, it is insufficient in setting the following two state variables.

Variable: `executionStateRoots`

The `executionStateRoots` state variable is a mapping from the execution layer's block number to its state root. The `update` function stores the block number and the new state root from a verified ZK proof into this mapping. Subsequently, the `updateStorageSlot` function relies on the information stored in `executionStateRoots` to verify a storage-slot value at a specific block number.

```
executionStateRoots[params.head] = params.executionStateRoot;
```

Therefore, in the constructor, `executionStateRoots` should store `params.executionStateRoot` using `params.executionBlockNumber` as the key. However, the current implementation incorrectly uses `params.head` as the key.

Variable: `executionBlockNumber`

The `executionBlockNumber` state variable is used by the `latestExecutionStateRoot` and `latestExecutionBlockNumber` functions to retrieve the most recently updated state root of the execution layer and its corresponding block number.

```
function latestExecutionStateRoot() public view returns (bytes32) {
    return executionStateRoots[executionBlockNumber];
}

function latestExecutionBlockNumber() public view returns (uint256) {
    return executionBlockNumber;
}
```

As of the time of writing, `executionBlockNumber` is not being initialized in the constructor.

Impact

Since `executionStateRoots` is initialized with an incorrect key, the `verifyStorageSlotsProof` function cannot fetch the state root for the user-provided block number. This remains the case until the `executionStateRoots` mapping is first updated by a call to the `update` function. Therefore, any call to `updateStorageSlot` before this initial update will always fail.

Furthermore, it is also impossible to retrieve the most recently updated state root of the execution layer and its corresponding block number via the `latestExecutionStateRoot` and `latestExecutionBlockNumber` functions.

Recommendations

Modify the `constructor` function as follows.

```
executionStateRoots[params.head] = params.executionStateRoot;
executionStateRoots[params.executionBlockNumber] = params.executionStateRoot;
executionBlockNumber = params.executionBlockNumber;
```

Remediation

This issue has been acknowledged by Succinct Labs, and a fix was implemented in commit `477ed4af` ↗.

3.3. Rust bindings of SP1Helios contract are out of sync

Target	primitives/src/types.rs		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In types.rs, Rust bindings for the SP1Helios contract are defined using the `sol!` macro, allowing the contract's functions to be used at the Rust code level. These bindings conveniently include view functions such as `getSyncCommitteePeriod`, `getCurrentSlot`, and `getCurrentEpoch` in addition to the `update` function. However, the current SP1Helios contract does not include a `getCurrentSlot()` function; instead, it defines a `getStorageSlot(uint256 blockNumber, address contractAddress, bytes32 key)` function.

```
#[allow(missing_docs)]
#[sol(rpc)]
contract SP1Helios {
    [...]
    function getSyncCommitteePeriod(uint256 slot)
    internal view returns (uint256);
    function getCurrentSlot() internal view returns (uint256);
    function getCurrentEpoch() internal view returns (uint256);
}
```

Impact

In the project's Rust-level code, attempting to call the `getCurrentSlot` function, which does not exist in the SP1Helios contract, may result in an error. Additionally, the `getStorageSlot` function cannot be used because its interface is not defined.

However, this view function is currently not used anywhere within the project.

Recommendations

Update the Rust bindings for SP1Helios defined in types.rs to match the latest version of the contract.

Remediation

This issue has been acknowledged by Succinct Labs.

3.4. Redundant check on `prevSyncCommitteeHash`

Target	contracts/src/SP1Helios.sol		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the `update` function, both `currentSyncCommitteeHash` and `po.prevSyncCommitteeHash` are derived from `syncCommittees[getSyncCommitteePeriod(head)]`. Therefore, the check [D] is redundant in the current implementation.

```
ProofOutputs memory po = ProofOutputs({
    prevHeader: headers[head],
    prevHead: head,
    prevSyncCommitteeHash: syncCommittees[getSyncCommitteePeriod(head)],   //
    ------------- [A]
    [...]
});

[...]

// The sync committee for the current head should always be set.
uint256 currentPeriod = getSyncCommitteePeriod(head);   // ------------- [B]
bytes32 currentSyncCommitteeHash = syncCommittees[currentPeriod];   //
    ------------- [C]

[...]

// The sync committee hash used in the proof should match the current sync
    committee.
if (currentSyncCommitteeHash != po.prevSyncCommitteeHash) {   //
    -------------  [D]
    revert SyncCommitteeStartMismatch(po.prevSyncCommitteeHash,
    currentSyncCommitteeHash);
}
```

Impact

This redundant check unnecessarily increases gas usage and adds complexity to the code, which may make it harder for users to understand.

Recommendations

Remove the following check from the `update` function.

```
if (currentSyncCommitteeHash != po.prevSyncCommitteeHash) {
    revert SyncCommitteeStartMismatch(po.prevSyncCommitteeHash,
    currentSyncCommitteeHash);
}
```

Remediation

This issue has been acknowledged by Succinct Labs, and a fix was implemented in commit `9a91a3be` ↗.

3.5. Unused errors in the SP1Helios contract

Target	contracts/src/SP1Helios.sol		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The SP1Helios contract defines error codes that represent various errors and constraint violations that may occur during operations. However, among them, `PrevHeadMismatch`, `PrevHeaderMismatch`, `HeaderRootAlreadySet`, `SyncCommitteeAlreadySet`, and `StateRootAlreadySet` are not used in the current implementation.

```
error PrevHeadMismatch(uint256 given, uint256 expected);
error PrevHeaderMismatch(bytes32 given, bytes32 expected);
error SlotBehindHead(uint256 slot);
error SyncCommitteeAlreadySet(uint256 period);
error HeaderRootAlreadySet(uint256 slot);
error StateRootAlreadySet(uint256 slot);
error SyncCommitteeStartMismatch(bytes32 given, bytes32 expected);
error SyncCommitteeNotSet(uint256 period);
error NextSyncCommitteeMismatch(bytes32 given, bytes32 expected);
error NonCheckpointSlot(uint256 slot);
error MissingStateRoot(uint256 blockNumber);
```

Impact

If unused error codes remain in the source code, they can reduce code readability and maintainability, potentially causing confusion during future development.

Recommendations

Remove the unused error codes: `PrevHeadMismatch`, `PrevHeaderMismatch`, `HeaderRootAlreadySet`, `SyncCommitteeAlreadySet`, and `StateRootAlreadySet`.

Remediation

This issue has been acknowledged by Succinct Labs, and a fix was implemented in commit `477ed4af` ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Loss of the guardian role after calling the `relinquishGuardian` function

In the SP1Helios contract, the guardian role is responsible for managing the `lightClientVkey` and `storageSlotVkey`, which are used for light-client state updates and verifying values of specific storage slots, respectively. The functions `updateLightClientVkey` and `updateStorageSlotVkey`, which allow modification of these Vkeys, can only be called by an account with the guardian role.

```
function relinquishGuardian() external onlyGuardian {
    guardian = address(0);

    emit GuardianRelinquished();
}
```

The `relinquishGuardian` function resets the state variable that stores the guardian address to the zero address. After this function is called, no one can invoke `updateLightClientVkey` or `updateStorageSlotVkey` anymore. While this allows the contract's Vkeys to be frozen, the irreversible nature of this functionality should be clearly documented.

4.2. Suggestion for gas optimization

The implementation of the SP1Helios contract contains several areas where gas usage can be optimized.

```
// Set all the storage slots.
for (uint256 i = 0; i < po.storageSlots.length; i++) {
    bytes32 key = computeStorageSlotKey(
        po.executionBlockNumber, po.storageSlots[i].contractAddress,
    po.storageSlots[i].key
    );
    storageSlots[key] = po.storageSlots[i].value;
}
```

In the `update` and `updateStorageSlot` functions, the loop variable `i` used in the for loops is incremented using the postfix increment operator. Replacing the postfix increment operator with the prefix increment operator in the for loops can help save gas.

Additionally, since the loop variable `i` is of type `uint256` and there is no risk of overflow, incrementing it within an `unchecked` block can further save gas. Lastly, caching `po.storageSlots.length` before the loop, instead of accessing it in every iteration, would also help improve efficiency.

The redundant check described in Finding 3.4. ↗ also involves unnecessary gas usage; therefore, introducing the fix for that finding can further optimize gas usage.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: SP1Helios.sol

Function: `changeGuardian(address newGuardian)`

This function is used to change the guardian of the contract. It is called only by the current guardian.

Inputs

- `newGuardian`
    - **Control**: Arbitrary.
    - **Constraints**: The new guardian cannot be the zero address.
    - **Impact**: The new guardian of the contract.

Branches and code coverage

Intended branches

- Update the guardian with the new address.
    - ☐ Test coverage

Negative behavior

- Revert if the caller is not the guardian.
    - ☐ Negative test
- Revert if the new guardian is the zero address.
    - ☐ Negative test

Function: `relinquishGuardian()`

This function is used to relinquish the guardian role of the contract. It is called only by the current guardian.

Branches and code coverage

Intended branches

- Set the guardian to the zero address, effectively relinquishing the role.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not the guardian.
  - ☐ Negative test

Function: `updateLightClientVkey(byte[32] newVkey)`

This function is used to update the verification key for the SP1 Helios light-client program. It is called only by the guardian of the contract.

Inputs

- `newVkey`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The new verification key for the SP1 Helios light-client program.

Branches and code coverage

Intended branches

- Update the light-client verification key with the new value.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not the guardian.
  - ☐ Negative test

Function: `updateStorageSlotVkey(byte[32] newVkey)`

This function is used to update the verification key for the SP1 Helios storage-slot proof program. It is called only by the guardian of the contract.

Inputs

- `newVkey`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The new verification key for the SP1 Helios storage-slot proof program.

Branches and code coverage

Intended branches

- Update the storage-slot proof verification key with the new value.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not the guardian.
  - ☐ Negative test

Function: `updateStorageSlot(bytes proof, StorageSlot[] _storageSlots, uint256 blockNumber)`

This function verifies a storage-slot proof and updates the storage slots in the contract. It uses the provided proof to verify the storage slots against the execution-state root for the given block number. If the proof is valid, it updates the storage slots in the contract.

The proof is expected to be generated by the SP1 Helios storage-slot proof program, which is a zkSNARK program that verifies the validity of the storage slots against the execution-state root. The verifier key is expected to be `STORAGE_ELF` of the SP1 Helios storage-slot proof program.

Inputs

- `proof`
  - **Control**: Arbitrary.
  - **Constraints**: Must be a valid proof for the storage slots against the execution-state root.
  - **Impact**: The proof bytes for the SP1 storage-slot proof.
- `_storageSlots`
  - **Control**: Arbitrary.
  - **Constraints**: None, but verified against the proof.

    - **Impact**: The storage slots to verify and update.
- `blockNumber`
    - **Control**: Arbitrary.
    - **Constraints**: None, but verified against the proof.
    - **Impact**: The block number of the storage slot, used to compute the storage-slot key.

Branches and code coverage

Intended branches

- Invoke `verifyStorageSlotsProof` to verify the proof with inputs.
    - ☐ Test coverage
- Update the storage slots with the provided values.
    - ☐ Test coverage

Negative behavior

- Revert if the execution-state root for the given block number is not set.
    - ☐ Negative test

Function call analysis

- `this.verifyStorageSlotsProof(proof, _storageSlots, blockNumber) -> ISP1Verifier(this.verifier).verifyProof(this.storageSlotVkey, abi.encode(sspo), proof)`
    - **What does this do?** Verifies the storage-slot proof against the execution-state root for the given block number.
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** If the execution-state root corresponding to the block number specified by the user cannot be retrieved from the contract, the call will revert and the update to the storage slot will fail.
- `ISP1Verifier(verifier).verifyProof(storageSlotVkey, abi.encode(sspo), proof)`
    - **What does this do?** The proof provided by the user is verified through the SP1 verifier contract along with the associated public values.
    - **If the return value is controllable, how is it used and how can it go wrong?** The verifier contract should be trusted, as it is a core part of the SP1 Helios light client. If the proof is invalid, it will revert.

- **What happens if it reverts, reenters or does other unusual control flow?** Reverts from this call means that the proof is invalid, which is expected behavior.

Function: `update(bytes proof, uint256 newHead, byte[32] newHeader, byte[32] executionStateRoot, uint256 _executionBlockNumber, byte[32] syncCommitteeHash, byte[32] nextSyncCommitteeHash, StorageSlot[] _storageSlots)`

This function is used to update the light client with a new header, execution-state root, and sync committee when it changes. All the inputs are verified against the proof provided, and the state of the light client is updated accordingly. In addition, the storage slots are updated with the provided values.

The proof is expected to be generated by the SP1 Helios light-client program, which is a zkSNARK program that verifies the validity of the new header, execution-state root, and sync committee against the previous state of the light client. The verifier key is expected to be `LIGHT_CLIENT_ELF` of the SP1 Helios light-client program.

Inputs

- `proof`
  - **Control**: Arbitrary.
  - **Constraints**: Verified in the `verifier` contract.
  - **Impact**: The proof bytes for the SP1 Helios light-client proof.
- `newHead`
  - **Control**: Arbitrary.
  - **Constraints**: Must be greater than the current head, divisible by 32, and a checkpoint slot, which is verified against the proof.
  - **Impact**: The slot of the new head.
- `newHeader`
  - **Control**: Arbitrary.
  - **Constraints**: None, but verified against the proof.
  - **Impact**: The new beacon block-header hash.
- `executionStateRoot`
  - **Control**: Arbitrary.
  - **Constraints**: None, but verified against the proof.
  - **Impact**: The execution-state root from the execution payload of the new beacon block.
- `_executionBlockNumber`

    - **Control**: Arbitrary.
    - **Constraints**: None, but verified against the proof.
    - **Impact**: The execution block number.
- `syncCommitteeHash`
    - **Control**: Arbitrary.
    - **Constraints**: None, but verified against the proof.
    - **Impact**: The sync-committee hash of the current period.
- `nextSyncCommitteeHash`
    - **Control**: Arbitrary.
    - **Constraints**: If the next sync committee is defined, it must match the expected value and be verified against the proof.
    - **Impact**: The sync-committee hash of the next period.
- `_storageSlots`
    - **Control**: Arbitrary.
    - **Constraints**: None, but verified against the proof.
    - **Impact**: The updated storage slots.

Branches and code coverage

Intended branches

- Call the verifier contract to verify the proof with inputs.
    - [ ] Test coverage
- Update the light-client state with the new head, header, execution-state root, and sync committee.
    - [ ] Test coverage
- Update the storage slots with the provided values.
    - [ ] Test coverage

Negative behavior

- Revert if the sync committee for the current head is not set.
    - [ ] Negative test
- Revert if the new head is behind the current head.
    - [ ] Negative test
- Revert if the new head is not a checkpoint slot (divisible by 32).
    - [ ] Negative test
- Revert if the next sync committee is defined but does not match the expected value.

- ☐ Negative test

Function call analysis

- `ISP1Verifier(this.verifier).verifyProof(this.lightClientVkey, abi.encode(po), proof)`
  - **What does this do?** The proof provided by the user is verified through the SP1 verifier contract along with the associated public values.
  - **If the return value is controllable, how is it used and how can it go wrong?** The verifier contract should be trusted, as it is a core part of the SP1 Helios light client. If the proof is invalid, it will revert.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reverts from this call means that the proof is invalid, which is expected behavior.

5.2. Programs: sp1-helios-program

The sp1-helios-program has two main components: light-client and storage.

Component: light-client

Description

The light-client component is responsible for verifying the state transitions and proofs of the SP1 Helios program. It ensures that the proofs are valid and that the state transitions are executed correctly.

The states of updates and finality updates are provided from external resources, such as the beacon chain. It should be trusted that the inputs are valid and correctly formatted.

Inputs

- **`encoded_inputs (Vec<u8>)`**: Input data for the light-client program, which includes the state-transition proofs and the associated data. It will be passed by the `sp1_zkvm::io::read_vec` function and decoded into a `ProofInputs` struct.

```
pub struct ProofInputs {
    pub updates: Vec<Update<MainnetConsensusSpec>>,
    pub finality_update: FinalityUpdate<MainnetConsensusSpec>,
    pub expected_current_slot: u64,
    pub store: LightClientStore<MainnetConsensusSpec>,
    pub genesis_root: B256,
    pub forks: Forks,
    pub contract_storage: Vec<ContractStorage>,
}
```

Outputs

- **`proof_outputs (ProofOutputs)`**: The output of the light-client program, which contains the results of the state transition and the verified proofs. It is later passed as an argument to the `update` function in the SP1Helios contract.

```
struct ProofOutputs {
    /// The previous beacon block header hash.
    bytes32 prevHeader;
    /// The slot of the previous head.
    uint256 prevHead;
    /// The anchor sync committee hash which was used to verify the proof.
    bytes32 prevSyncCommitteeHash;
    /// The slot of the new head.
    uint256 newHead;
    /// The new beacon block header hash.
    bytes32 newHeader;
    /// The execution state root from the execution payload of the new beacon
    block.
    bytes32 executionStateRoot;
    /// The execution block number.
    uint256 executionBlockNumber;
    /// The sync committee hash of the current period.
    bytes32 syncCommitteeHash;
    /// The sync committee hash of the next period.
    bytes32 nextSyncCommitteeHash;
    /// Attested storage slots for the given block.
    StorageSlot[] storageSlots;
}
```

Expected behavior

- Decode the `encoded_inputs` into a `ProofInputs` struct.
- Get the initial sync committee, and save it to `prev_sync_committee_hash`.
- Get the header and head from `finalized_header` of the beacon state.
- Verify and apply all generic updates.
- Verify the finality update.
- Check that the new head is not greater than the previous head and that it is a checkpoint

slot.
- Commit new state root, header, and sync-committee hash.
- Verify the storage slots against the contract's storage root.
- Generate the `ProofOutputs` with the verified state root and storage slots.

Component: storage

The storage component manages the storage of accounts and requests, verifying their Merkle proofs against the root of each storage. It executes the state-transition function and updates the storage roots accordingly.

Inputs

- **`storage (Vec<ContractStorage>)`**: A vector of `ContractStorage` objects, which represent the storage data for the accounts and requests.

```
pub struct ContractStorage {
    pub address: Address,
    pub value: TrieAccount,
    /// The proof that this contracts storage root is correct
    pub mpt_proof: Vec<Bytes>,
    /// The storage slots that we want to prove
    pub storage_slots: Vec<StorageSlotWithProof>,
}

pub struct StorageSlotWithProof {
    pub key: B256,
    pub value: U256,
    /// The proof that this storage slot is correct
    pub mpt_proof: Vec<Bytes>,
}
```

- **`state_root (B256)`**: The root of the storage state, which is used to verify the integrity of the storage data.

Outputs

- **`proof_outputs (StorageProofOutputs)`**: The output of the storage program, which contains the state root and the storage slots with their proofs. It is later passed as an argument to the `updateStorageSlot` function in the SP1Helios contract.

```
struct StorageProofOutputs {
    bytes32 stateRoot;
    StorageSlot[] storageSlots;
}
```

Expected behavior

- Invoke the `verify_storage_slot_proofs` function to verify the proofs of the storage slots.
- Verify the contract's account node in the global MPT.
- Verify the storage slots against the contract's storage root.
- Generate the `StorageProofOutputs` with the verified state root and storage slots.

6. Assessment Results

During our assessment on the scoped SP1 Helios contracts, we discovered five findings. One critical issue was found. One was of medium impact and the remaining findings were informational in nature.

The overall code quality of this project is high. However, as mentioned in Finding 3.1. ↗, the verifier contract SP1Helios.sol and SP1 Helios programs lack a test suite that verifies integration of Helios and the project's core functionality. Therefore, we strongly recommend writing thorough tests targeting both the contract and the SP1 Helios programs.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.