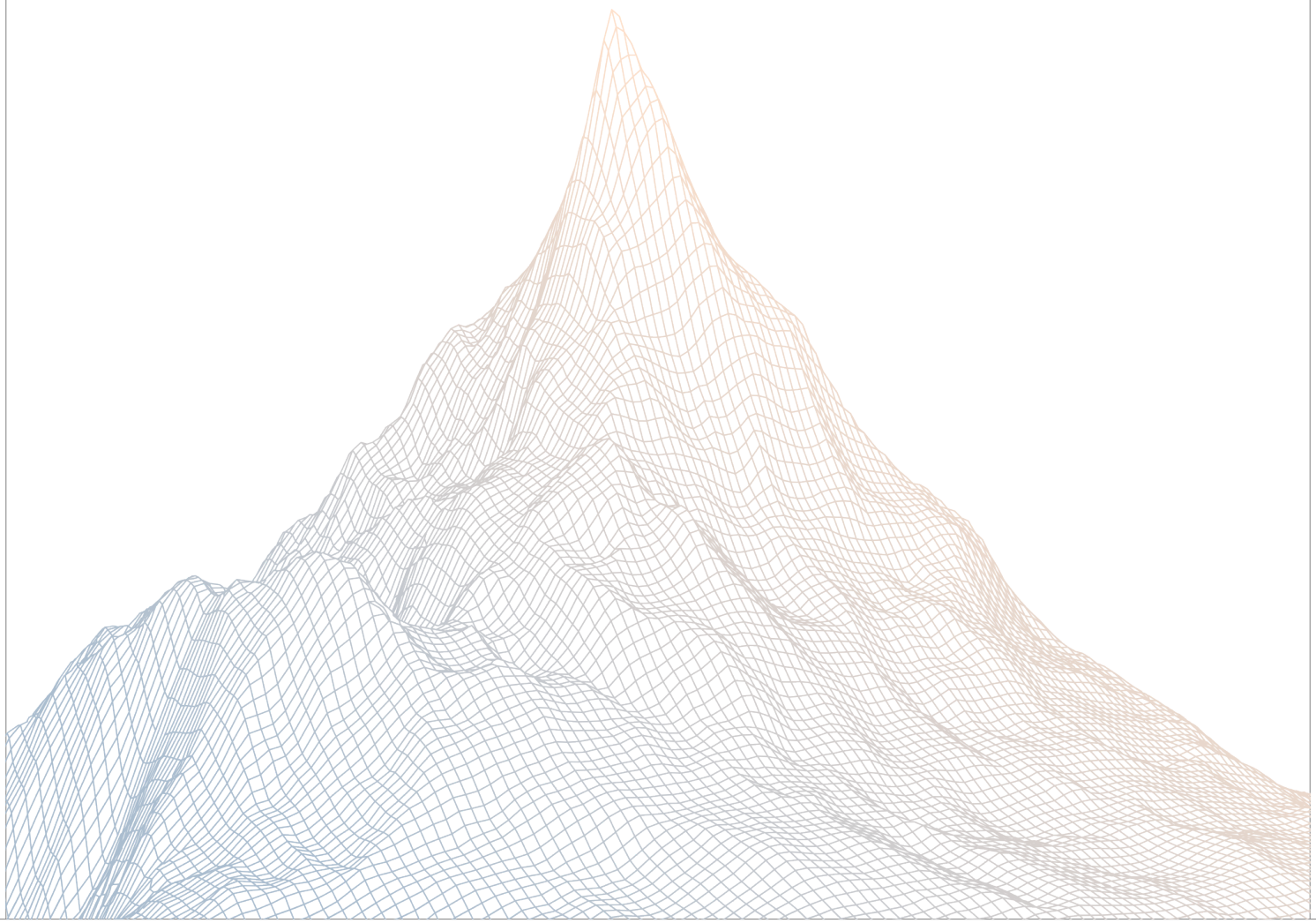
Zenith

Succinct

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: March 21st to March 22nd, 2025

AUDITED BY: hash
said

Contents

1 Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Succinct

SP1 is a performant, open-source zero-knowledge virtual machine (zkVM) that verifies the execution of arbitrary Rust (or any LLVM-compiled language) programs.

ZKPs enable a diversity of use-cases in blockchain and beyond, including:

- Rollups: Use SP1 to generate a ZKP for the state transition function of your rollup and connect to Ethereum, Bitcoin or other chains with full validity proofs or ZK fraud proofs.
- Interoperability: Use SP1 for fast-finality, cross rollup interoperability
- Bridges: Use SP1 to generate a ZKP for verifying consensus of L1s, including Tendermint, Ethereum's Light Client protocol and more, for bridging between chains.
- Oracles: Use SP1 for large scale computations with onchain state, including consensus data and storage data.
- Aggregation: Use SP1 to aggregate and verify other ZKPs for reduced onchain verification costs.
- Privacy: Use SP1 for onchain privacy, including private transactions and private state.

2.2 Scope

The engagement involved a review of the following targets:

Target	sp1-tee
Repository	https://github.com/succinctlabs/sp1-tee
Commit Hash	`fe54785b2a47ef2444b902018429f914bab5ccc8`

Files	`SP1TeeVerifier.sol`
	`SignersMap.sol`
	`SimpleOwnable.sol`

2.3 Audit Timeline

March 21, 2025	Audit start
March 22, 2025	Audit end
March 28, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	1
Low Risk	1
Informational	5
Total Issues	**7**

3

Findings Summary

ID	Description	Status
M-1	Flawed encoding due to dynamic types allow an attacker to pass signature verification using malicious values	Resolved
L-1	The event emits an incorrect previousOwner address when transferOwnership is called	Resolved
I-1	Index updation is incorrect when removing signers	Resolved
I-2	Missing events for signers management operations	Resolved
I-3	Providing version is not enforced in verifyProof	Acknowledged
I-4	Insufficient proof length validation	Resolved
I-5	Missing address validation and event emission in the SimpleOwnable constructor	Resolved

4

Findings

4.1 Medium Risk

A total of 1 medium risk findings were identified.

[M-1] Flawed encoding due to dynamic types allow an attacker to pass signature verification using malicious values

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- SP1TeeVerifier.sol

Description:

The message hash is currently constructed as `keccak256(abi.encodePacked(version, programVKey, publicValues))`

```
function verifyProof(bytes32 programVKey, bytes calldata publicValues,
    bytes calldata proofBytes) external view {
   ........

    // Extract the version from the proof bytes.
    uint8 version_len = uint8(proofBytes[69]); // 1 byte: version_len
    bytes memory version = proofBytes[70:70 + version_len]; // version_len
    bytes: version

    // Compute the expected hash of the message
    bytes32 message_hash = keccak256(abi.encodePacked(version, programVKey,
    publicValues));
```

Since `version` and `publicValues` are of dynamic type, the same final encoded bytes can be attributed to multiple combinations

Eg: actual version = 0x0101 actual programVKey = bytes32(uint(1)) == 0x0000000000000000000000000000000000000000000000000000000000000001 actual publicValues = 0x01

alternate values that produce the same final encoding, alternate version = 0x01 alternate

programVKey = 0x0100000000000000000000000000000000000000000000000000000000000000 alternate publicValues = 0x0101

both produce the same encoded bytes :
0x0101000000000000000000000000000000000000000000000000000000000101

Hence this allows an attacker to pass the verification using values different from what was signed

Recommendations:

Use hashes of version and publicValues instead

```
bytes32 message_hash = keccak256(abi.encodePacked(keccak256(version),
    programVKey, keccak256(publicValues)));
```

Succinct: Resolved with PR-3

Zenith: Verified.

4.2 Low Risk

A total of 1 low risk findings were identified.

[L-1] The event emits an incorrect `previousOwner` address when `transferOwnership` is called

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- SimpleOwnable.sol#L40

Description:

The event emits the wrong `previousOwner` value because it uses `owner` after it's been updated.

```
    function transferOwnership(address newOwner) external onlyOwner {
        if (newOwner == address(0)) {
            revert("New owner cannot be the zero address");
        }

        owner = newOwner;

>>>     emit OwnershipTransferred(owner, newOwner);
    }
```

Recommendations:

store the old owner before updating the state.

```
    function transferOwnership(address newOwner) external onlyOwner {
        if (newOwner == address(0)) {
            revert("New owner cannot be the zero address");
        }
```

```
        address oldOwner = _owner;
        owner = newOwner;

        emit OwnershipTransferred(owner, newOwner);
        emit OwnershipTransferred(oldOwner, newOwner);
    }
```

Succinct: Resolved with PR-6

Zenith: Verified.

4.3 Informational

A total of 5 informational findings were identified.

[I-1] Index updation is incorrect when removing signers

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: High

Target

- SignersMap.sol

Description:

The index of the lastSigner is always updated as `indexToRemove`. This is flawed as if the signer being removed is the last signer, then the index will be re-written as `length - 1` instead of setting it to 0

```
function removeSigner(SignersMap storage self, address signer) internal {
        .......

        uint256 indexToRemove = self.signerIndex[signer];
        delete self.signerIndex[signer];

        .......

        self.signerIndex[lastSigner] = indexToRemove;

    }
```

Recommendations:

If the indexToRemove is `length - 1`, then pop without performing swaps. Also the SignersMap could be refactored to contain just a single map rather than maintaining 2 different ones by using the index map to determine existence as well (start real indexes from 1 and use 0 for non-existance)

Succinct: Resolved with PR-4

Zenith: Verified.

[I-2] Missing events for signers management operations

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- SP1TeeVerifier.sol#L41-L53

Description:

The `addSigner` and `removeSigner` functions in `SP1TeeVerifier` perform signer changes but don't emit events to track these changes. Makes it harder to monitor for potentially malicious signer additions/removals.

Recommendations:

Add events for signer management operations.

Succinct: Resolved with PR-7

Zenith: Verified.

[I-3] Providing `version` is not enforced in `verifyProof`

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- ERC20.sol

Description:

The contract allows a `version_len` of 0, causing `version` to become an empty `bytes` array. This could remove explicit version specification if version enforcement is intended or lead to confusion if the protocol requires version information.

```
    function verifyProof(bytes32 programVKey, bytes calldata publicValues,
    bytes calldata proofBytes) external view {
        // Assure the proof type is correct for this verifier.
        bytes4 receivedSelector = bytes4(proofBytes[:4]);
        bytes4 expectedSelector = bytes4(VERIFIER_HASH());
        if (receivedSelector ≠ expectedSelector) {
            revert WrongVerifierSelector(receivedSelector,
    expectedSelector);
        }

        // Extract the recovery id and the signature.
        uint8 v = uint8(proofBytes[4]); // 1 byte: v
        bytes32 r = bytes32(proofBytes[5:37]); // 32 bytes: r
        bytes32 s = bytes32(proofBytes[37:69]); // 32 bytes: s

        // Extract the version from the proof bytes.
>>>     uint8 version_len = uint8(proofBytes[69]); // 1 byte: version_len
        bytes memory version = proofBytes[70:70 + version_len]; //
    version_len bytes: version

        // Compute the expected hash of the message
        bytes32 message_hash = keccak256(abi.encodePacked(version,
    programVKey, publicValues));
         // ...
}
```

Recommendations:

Consider adding a minimum `version_len` check to ensure `version` is always provided.

Succinct: Acknowledged

[I-4] Insufficient proof length validation

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- SP1TeeVerifier.sol#L85-L125

Description:

The `verifyProof` function in `SP1TeeVerifier` performs multiple array accesses on the `proofBytes` parameter without first validating its length. The function expects at least length of 70 bytes for the proof structure (4 bytes selector + 1 byte v + 32 bytes r + 32 bytes s + 1 byte `version_len`), but there's no explicit check for this.

If a `proofBytes` shorter than 70 bytes is provided, the function will revert with an out-of-bounds error, which is less clear than a custom error message.

Recommendations:

Add an explicit length check at the start of the `verifyProof`.

Succinct: Resolved with PR-5

Zenith: Verified.

[I-5] Missing address validation and event emission in the `SimpleOwnable` constructor

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- SimpleOwnable.sol#L17-L19

Description:

In `SimpleOwnable`, the constructor lacks a zero-address check, which could permanently lock the contract if deployed with a zero address. Additionally, it does not emit an event for ownership initialization.

Recommendations:

Consider to add the check and emit the `OwnershipTransferred` event.

```
constructor(address _owner) {
    if (_owner == address(0)) {
        revert OwnableInvalidOwner(address(0));
    }
     owner = _owner;
    emit OwnershipTransferred(address(0), _owner);
 }
```

Succinct: Resolved with PR-6

Zenith: Verified.