Audit Report

ssz-rs

v1.0

September 28, 2023

Table of Contents

License

Disclaimer

THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND.

THE AUTHOR AND HIS EMPLOYER DISCLAIM ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT.

THIS AUDIT REPORT IS ADDRESSED EXCLUSIVELY TO THE CLIENT. THE AUTHOR AND HIS EMPLOYER UNDERTAKE NO LIABILITY OR RESPONSIBILITY TOWARDS THE CLIENT OR THIRD PARTIES.

This audit has been performed by

Oak Security

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security has been engaged by Snowfork to perform a security audit of ssz-rs.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/ralexstokes/ssz-rs
Commit	`b8729699f07f0d348053251dd6ddf838656849d1`
Scope	All code was in scope.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

ssz-rs is a Rust implementation of Simple Serialize (SSZ), the serialization method used on the Ethereum Beacon Chain.
SSZ is designed to be deterministic and also to Merkleize efficiently. SSZ can be thought of as having two components: a serialization scheme and a Merkleization scheme that is designed to work efficiently with the serialized data structure.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending,** **Acknowledged**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	Medium	-
Code readability and clarity	Medium-High	-
Level of documentation	High	-
Test coverage	Low-Medium	`cargo tarpaulin` reports a test coverage of 47.30%.

Summary of Findings

No	Description	Severity	Status
1	Excess bytes in the encoding of `None` do not trigger errors	Minor	Resolved
2	`is_zero` method from the SSZ specification is not implemented	Minor	Acknowledged
3	`debug_assert_eq!` macro checks are not performed in production code	Minor	Resolved
4	Overflow checks not enabled for release profile	Minor	Acknowledged
5	Vectors and arrays of different lengths have the same root hash	Informational	Acknowledged
6	Redundant mutable reference in `Merkleized` interface	Informational	Acknowledged
7	Duplicate length validation of a fixed serialized composite	Informational	Resolved
8	Misleading error name	Informational	Resolved
9	Duplicated code decreases maintainability	Informational	Resolved
10	Duplicated `ValidationState` validation	Informational	Resolved
11	Use of magic numbers decreases maintainability	Informational	Resolved

Detailed Findings

1. Excess bytes in the encoding of `None` do not trigger errors

Severity: Minor

The `deserialize` function, defined in `ssz-rs/src/union.rs:45`, invokes the deserialization of the inner value after determining the input as `Some`.

This process can potentially lead to the occurrence of a `DeserializeError::AdditionalInput` error if the encoding contains more bytes than required for deserializing the inner value.

However, the code does not inspect any bytes of the encoding beyond the initial byte when handling the `None` case. As a consequence, if an encoding of `None` includes redundant bytes, the code will not throw an error.

Recommendation

We recommend ensuring that only the canonical single-byte encoding of `None` is accepted and an error is thrown otherwise as performed in the `deserialize` function in `ssz-rs/src/boolean.rs`.

Status: Resolved

2. `is_zero` method from the SSZ specification is not implemented

Severity: Minor

The SSZ specification defines an `is_zero` method that returns `true` if the argument is a default value.

The relevant specification can be found at https://github.com/ethereum/consensus-specs/blob/fa09d896484bbe240334fa21ffaa454bafe5842e/ssz/simple-serialize.md#is_zero.

Recommendation

We recommend implementing the `is_zero` method to be fully compliant with the `SSZ` specification.

Status: Acknowledged

3. `debug_assert_eq!` macro checks are not performed in production code

Severity: Minor

In the `serialize_composite_from_components` function, defined in `ssz-rs/src/ser.rs`, parameters are validated using the `debug_assert_eq!` macro.

This macro will be ignored by the compiler in release mode, i.e. after switching to production.

This is because the compiler optimizes the code in release mode, which results for example in the omission of most of the debugging and error-logging functions.

Recommendation

We recommend implementing these assertions together with the error-handling logic, or using the `assert_eq!` macro, which is not ignored by the compiler in release mode.

Status: Resolved

4. Overflow checks not enabled for release profile

Severity: Minor

The `ssz_rs` and `ssz_rs_derive` crates do not enable `overflow-checks` in the `Cargo.toml` release profile.

Consequently, the entire codebase does not have implemented protection against overflow or underflow.

Recommendation

We recommend enabling overflow checks in all packages, including those that do not currently perform calculations, to prevent unintended consequences if changes are added in future releases or during refactoring. Note that enabling overflow checks in packages other than the workspace manifest will lead to compiler warnings.

Status: Acknowledged

5. Vectors and arrays of different lengths have the same root hash

Severity: Informational

The `hash_tree_root` method of the `Merkleized` trait, defined for vectors in `vector.rs:264-264` and for arrays in `array.rs:77-91`, returns the same Merkle root for values of different sizes.

It is important to note that, according to SSZ specification, those types are statically sized, meaning that vectors and arrays of different lengths represent different types. Therefore, it is possible for different types to have the same Merkle roots and encodings, as SSZ encodings are not self-contained and require a specific scheme for proper deserialization. However, it is crucial to emphasize this in the user documentation, as incorrect assumptions about Merkle roots could potentially be exploited by library consumers.

A further observation regarding Merkle roots is that vectors clusterize by their Merkle roots. We can denote the expression `Vector::<u8, n>::try_from(vec![0; n]).hash_tree_root()` as *M(n)*. In this context, the following equalities hold true:

```
M(1) = M(2) = … = M(32),
M(33) = M(34) = … = M(64),
…
```

The same observation applies to arrays where:

```
M(1) = M(2) = … = M(32)
```

A test case is provided in Appendix.

Recommendation

We recommend emphasizing the usage scenarios in which the library can be securely employed. This includes highlighting corner cases that may be confusing for users and might appear to be a hash collision.

Status: Acknowledged

6. Redundant mutable reference in `Merkleized` interface

Severity: Informational

In the `hash_tree_root` function, defined in `ssz-rs/src/merkleized/mod.rs`, the only parameter, `self`, is declared as mutable reference.

However, mutability is not utilized anywhere. Therefore, it is safe to remove the mutability from the parameter declaration.

Recommendation

We recommend declaring parameters as immutable references whenever possible. In this specific case, declaring the parameter as `&self` and refactor the code using that parameter accordingly, e.g., using the `iter()` function instead of `iter_mut()` on `self.data` in assisting functions.

Status: Acknowledged

7. Duplicate length validation of a fixed serialized composite

Severity: Informational

When deserializing Array, List, and Vector objects, depending on whether they have a fixed or variable length, one of two functions is called within `deserialize_homogeneous_composite` – either `deserialize_variable_homogeneous_composite` or `deserialize_fixed_homogeneous_composite`.

`deserialize_fixed_homogeneous_composite` is called when the type is fixed, and the corresponding `deserialize` functions have already validated the length of the object to be a multiple of the default value of that type obtained using `T::size_hint`. An example for the Array type is the validation performed in `ssz-rs/src/array.rs:51-65`.

Nevertheless, the `deserialize_fixed_homogeneous_composite` function in lines `71-78` validates whether the modulo of the length of the deserialized object and the default size for its type is different from zero. Since in the previous step, this size was multiplied by `N`, there is no possibility that the modulo will be different from zero. This validation is therefore therefore redundant, and can be removed for efficiency of the codebase.

Recommendation

We recommend removing the redundant validation performed in lines `71-78`.

Status: Resolved

8. Misleading error name

Severity: Informational

The `serialize_composite_from_components` function defined in `ssz-rs/src/ser.rs:58` verifies that the sum of the lengths of variable and constant elements does not exceed the `MAXIMUM_LENGTH` value, which is defined as the maximum range of `u32`. A `MaximumEncodedLengthExceeded` error is returned when the sum is greater than or equal to `MAXIMUM_LENGTH`. This is misleading, because, in the case of equality, the value is not exceeded.

Recommendation

We recommend renaming the error to include equality, for example, `MaximumEncodedLengthReached`.

Status: Resolved

9. Duplicated code decreases maintainability

Severity: Informational

Code duplicates make it more difficult to maintain, review, and reason about the code.

The code in `list.rs:210-214` and `vector.rs:230-234` is duplicated.

Recommendation

We recommend extracting common code into functions in order to make the codebase easier to maintain and review.

Status: Resolved

10. Duplicated `ValidationState` validation

Severity: Informational

The `SimpleSerialize` trait implements the `derive` function, which performs actions on the passed `input`. One of these actions is the validation of `input.data`, carried out in `ssz-rs-derive/src/lib.rs:548`.

This function returns `ValidationState::Validated` if the validation is successful and panics otherwise.

Consequently, after this action, there is no technical possibility that the state of `input.data` will still be `Unvalidated`. This implies that the validation performed in `ssz-rs-derive/src/lib.rs:550-553` is redundant.

Recommendation

We recommend removing duplicate validation in order to increase the efficiency and readability of the code.

Status: Resolved

11. Use of magic numbers decreases maintainability

Severity: Informational

In the codebase, hard-coded number literals without context or a description are used. Using such “magic numbers” goes against best practices as they reduce code readability and maintenance as developers are unable to easily understand their use and may make inconsistent changes across the codebase. Instances of magic numbers have been found in:

- `ssz-rs/src/uint.rs:18`
- `ssz-rs/src/uint.rs:25`

- `ssz-rs/src/uint.rs:31`
- `ssz-rs/src/list.rs:219`

Recommendation

We recommend defining magic numbers as constants with descriptive variable names and comments, where necessary.

Status: Resolved

Appendix: Test Cases

1. **Test case for "Vectors and arrays of different lengths have the same root hash"**

This test case provides evidence of two different arrays with the same hash tree root:

```
mod tests {
    use super::*;

    #[test]
    fn hash_array() {
        let mut data_1 = [1u8];

        let mut data_2 = [1u8, 0u8, 0u8, 0u8, 0u8,0u8, 0u8, 0u8, 0u8,
0u8,0u8, 0u8, 0u8, 0u8, 0u8,0u8, 0u8, 0u8, 0u8, 0u8,0u8, 0u8, 0u8, 0u8,
0u8,0u8, 0u8, 0u8, 0u8, 0u8, 0u8 , 0u8];

        let encoding_1 = data_1.hash_tree_root().unwrap();
        dbg!(encoding_1);

        let encoding_2 = data_2.hash_tree_root().unwrap();
        dbg!(encoding_2);

        assert_ne!(encoding_1, encoding_2)
    }
}
```

Similarly, this test case provides evidence of two different Vectors with the same hash tree root:

```
  #[test]
    fn hash_vector() {
        let data_1 = vec![10; 1];
        let mut vector_1 = Vector::<u8, 1>::try_from(data_1).unwrap();

        let mut data_2 = vec![0; 32];
        data_2[0] = 10;
        let mut vector_2 = Vector::<u8, 32>::try_from(data_2).unwrap();

        let encoding_1 = vector_1.hash_tree_root().unwrap();
```

```
        dbg!(encoding_1);

        let encoding_2 = vector_2.hash_tree_root().unwrap();
        dbg!(encoding_2);

        assert_ne!(encoding_1, encoding_2)
    }
```