Serai BTC code audit

Cypher Stack[*]

August 18, 2023

This report describes the findings of a partial code audit of Serai. It reflects a limited scope provided by Serai and represents a best effort; as with any review or audit, it cannot guarantee that any protocol or implementation is suitably secure for a particular use case, nor that the contents of this report reflect all issues or vulnerabilities that may exist. The author asserts no warranty and disclaims liability for its use. The author further expresses no endorsement of Serai or its associated entities. This report has not undergone any further formal or peer review.

Contents

1 Overview **2**
- 1.1 Introduction 2
- 1.2 Summary of findings 2

2 Scope **2**

3 Findings **3**
- 3.1 Unnecessary HTTP client creation 3
- 3.2 Lack of HTTP error differentiation 3
- 3.3 Connections are checked using block information 3
- 3.4 Lack of error differentiation on decoding 4
- 3.5 Assumption of available RPC methods 4
- 3.6 Use of `unwrap` in production 4
- 3.7 Safe but unbounded loop in offset registration 4
- 3.8 Transaction scanning uses a fallible cast 5
- 3.9 Transaction machine instantiation uses a fallible cast 5
- 3.10 Offset addresses use a hardcoded network 5
- 3.11 Address generation assumes correct generation 5
- 3.12 Offsets and addresses are not bijective 5
- 3.13 Dust constant is incorrect 6

[*]`https://cypherstack.com`

1 Overview

1.1 Introduction

Serai is a distributed exchange protocol and Rust implementation. This report represents an audit of specific areas of underlying cryptographic libraries and functionality in the Serai codebase.

Serai and Cypher Stack identified the following goals for this audit:

- Assert that implementations of external specifications or protocols are done correctly
- Determine if malicious or unintended edge cases can be exploited
- Identify cases where the code may panic unexpectedly
- Note situations where documentation is insufficient to convey intent or design
- Ensure that secret data is handled as safely as is feasible with respect to memory clearing and constant-time operations
- Identify areas of the implementation where efficiency gains are possible and reasonable
- Determine the extent to which the implementation contains relevant and comprehensive tests

1.2 Summary of findings

As with a previous audit of Serai code, we find that the implementation is well designed and carefully written with secure deployment in mind. We did not identify any findings of particular immediate concern.

Consequently, the findings described in this report primarily identify recommendations for improvement. These deal with inconsistencies, documentation, tests, or areas of the codebase where fixes or changes could mitigate future issues.

However, we carefully note that we do not assign severity ratings to the issues contained herein. Such ratings are often subjective, and typically depend on the ease or difficulty of triggering specific behavior, which can be challenging to assess in a consistent way.

2 Scope

The scope of this audit was limited to the `coins/bitcoin` directory of the `bitcoin-audit` branch of the Serai repository at a specific commit[1].

[1] `https://github.com/serai-dex/serai/tree/21026136bd0c7f341ae93f08e6a0bb15fb9b250f`

Review of this directory entailed determining if it implements Serai- and Bitcoin-related cryptographic code correctly, and if it presents transaction-related issues as part of its implementation.

3 Findings

In the subsequent findings, we include actions taken and responses provided by Serai in response to an initial version of this report. Where listed, commits based on these actions refer to the `bitcoin-audit` repository branch, and function as hyperlinks if supported by your document reader software.

3.1 Unnecessary HTTP client creation

When making an RPC call, the implementation creates a new HTTP `Client` for the call. This seems unnecessary and could be inefficient, as sharing a single client across multiple requests could allow for connection pooling if supported.

Recommended fix is to refactor in order to reuse a single `Client` if feasible.

Action: Addressed in commit `c878d38`, which refactors to reuse a `Client` within an `Rpc` struct.

3.2 Lack of HTTP error differentiation

When making RPC calls and receiving responses, there is no differentiation between errors returned during the associated HTTP request and those returned on receipt of the response. It may be useful to either propagate detailed error information for debugging or handling purposes, or to simply define distinct request and response errors.

Recommended fix is to return differentiated errors, unless the single error is intentional to mitigate side channels.

Action: Determined to be unnecessary due to a desire not to rely on library-specific error handling.

3.3 Connections are checked using block information

When creating a new RPC client, the connection is checked for validity by fetching the latest block number from the server and discarding the result. However, failure of this process may not adequately differentate to the user between connectivity problems and internal node status, such as being online but not fully synchronized to the Bitcoin network.

Recommended fix is to consider a more robust RPC server and node health check, if feasible.

Action: Addressed in commit `7fa5d29`, which queries for support of a set of required methods.

3.4 Lack of error differentiation on decoding

When decoding response data, the implementation often generically maps various decoding errors to a single `InvalidResponse` error. This may not properly capture the difference between high-level encoding errors and lower-level errors within responses, which could hinder debugging.

Recommended fix is to consider more error differentiation on response decoding.

Action: Addressed in commit `3480fc5`, which includes more fine-grained error reporting.

3.5 Assumption of available RPC methods

The implementation assumes that the `getblock`, `sendrawtransaction`, and `getrawtransaction` RPC calls are enabled and available on the server node. This is not guaranteed, and depends on the node configuration. Relying on generic failure responses may be insufficient to determine these capabilities.

Recommended fix is to consider testing for call availability when instantiating the RPC client.

Action: Addressed in commit `7fa5d29`, which queries for support of a set of required methods.

3.6 Use of `unwrap` in production

The implementation uses `unwrap` in non-test production code where logic indicates a result must be valid. The use of `unwrap` in this way is largely a matter of developer preference, and is not inherently dangerous or unsafe. However, when it is used, it may be preferable to use `expect` instead, in order to admit more controlled and actionable debugging messages. This can also help to avoid edge cases where non-public data could be leaked during an `unwrap` execution.

Recommended fix is to consider uses of non-test `unwrap`, and further consider replacing with `expect` where helpful.

Action: Addressed in commits `d75115c` and `677b9b6`, with remaining uses deemed safe.

3.7 Safe but unbounded loop in offset registration

When constructing an address using an offset, the implementation increments the candidate offset and attempts to generate a valid address with it, additionally checking that the offset is not already registered.

Assuming that the address generation callee operates correctly, the loop is guaranteed to terminate since incrementing must yield a group element corresponding to a valid address. If the callee were to malfunction (which is not an assumption here), it could be possible that the loop does not terminate.

Recommendation is to add a brief comment explaining why the loop must terminate.

Action: Addressed in commit `fa1b569`, which adds such a comment.

3.8 Transaction scanning uses a fallible cast

When scanning a transaction, the implementation iterates over its outputs and casts the output index from `usize` to `u32` using an `unwrap`. While it should not be possible to construct a valid transaction with such a structure that could trigger a panic, it is not clear whether a malicious node could trigger it in a way that is not caught by the parser.

Recommendation is to check for this case and simply fail to include any failure cases in the returned vector of received outputs.

Action: Addressed in commit `677b9b6`, which performs a check.

3.9 Transaction machine instantiation uses a fallible cast

When setting up a transaction machine for a signable transaction, the implementation iterates over the transaction's inputs and casts the input index from `usize` to `u32` using an `unwrap`. While it should not be possible to construct a valid transaction with such a structure that could trigger a panic, it is not clear whether a malicious node could trigger it in a way that is not caught by the parser.

Recommendation is to check for this case and simply fail to produce the transaction machine if it is triggered.

Action: Addressed in commit `677b9b6`, which performs a check.

3.10 Offset addresses use a hardcoded network

When registering address offsets, the implementation generates the addresses using a hardcoded network. This may limit functionality for use in other networks, especially for test cases where inadvertent use of a production network could be dangerous or risk fund loss.

Action: Addressed in commit `f66fe3c`, which generalizes networks.

3.11 Address generation assumes correct generation

When generating a Taproot address from a key, the implementations uses library functions that assume the key has undergone any necessary "tweak" to ensure safe use in practice. While the implementation uses it safely, the generator function itself is public and does not include documentation indicating what preparation (if any) should be done to the input key.

Recommendation is to improve the documentation for Taproot address generation, and consider limiting the function visibility if feasible to avoid unintentional misuse.

Action: Addressed in commit `6f9d02f`, which adds such comments.

3.12 Offsets and addresses are not bijective

When constructing an offset address, the implementation attempts to apply the requested scalar offset, incrementing if necessary to produce an unused valid address. This design implies that there is a many-to-one mapping between offsets and their corresponding addresses. While this is not inherently unsafe or incorrect, even with the existing documentation, a caller might unintentionally make this assumption as part of a future design or protocol.

Recommendation is to improve the documentation relating to this property.

Action: Addressed in commit `df67b7d`, which adds such comments.

3.13 Dust constant is incorrect

The implementation checks candidate output data to ensure that values are above a constant dust limit that would cause nodes not to relay the resulting transaction.

This constant is set to 674, with a link to external Bitcoin transaction test source code. However, this reference test appears to be a non-standard case that uses a custom fee rate. Closer examination of the transaction tests and associated constants indicates that nodes using the default relay fee settings will use a dust limit value of 546. It may be the case that the implementation's higher limit is intentional, as the reference test discusses the effects of rounding.

Recommendation is to determine if the dust constant is set as expected, and to change it if not.

Action: Addressed in commits `1eb3b36` and `5121ca7`, which address the handling of fees.