Cosmos Blockchain SDK Framework

Final Security Audit Report

Interchain Foundation

Report Version: February 22, 2019

Least Authority

PRIVACY MATTERS

Table of Contents

Overview

Interchain Foundation has requested that Least Authority perform a security audit of the Cosmos Blockchain SDK, a framework for building Proof of Stake state machines, in anticipation of the Cosmos mainnet launch in Q1 of 2019.

This audit was performed with a limited schedule, to facilitate a high level review of the project and a focused review of particular parts. The goal was to investigate high priority areas of concern as directed by the client team and to discover more obvious issues, with the understanding that the time limit prevents a more comprehensive evaluation.

The audit was performed from January 7 - 22, 2019 by Ramakrishnan Muthukrishnan and Emery Hall. The initial report was issued on January 23, 2019. An updated report has been issued following the discussion and verification phase on February 22, 2019.

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Cosmos Blockchain SDK followed by issue reporting, along with mitigation and remediation instructions outlined in this report. The following code repositories are in scope:

Specifically, we examined the cosmos-sdk Git revision and the dependencies:

`7f789d2ed342de18f4443ae434f3e43f790f1854`

For the verification, we examined the Tendermint repo Git revision:

a8dbc64319ae0db3fc2621b36d4ef96ce9d62d15 (tagged v0.29.2)

All file references in this document use Unix-style paths relative to the project's root directory.

Areas of Concern

Our investigation focused on the following areas:

- Review of BaseApp;
- Tooling for a chain initialization process;
- Review of the state and transactions documentation;
- Review of the auth and bank module specification;
- Review of the F1 Fee Distribution Module;
- Review of Tombstone.

Game theory aspects of the consensus algorithms were not specified as an area of concern and review for this audit.

Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protects users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.

Findings

Code Quality

Overall, we found the code base to be very well organized and did not appear to contain unnecessary, excess code. The clean and succinct coding style allows efficient and comprehensive code review, thus facilitating the contributions of others in finding potential vulnerabilities. The code follows ethereum best practices and avoids known bugs such as re-entrancy.

We also found that tests exist for all major modules (some of them with 100% test coverage) that increase the level of confidence in the correctness of the code. We strongly suggest continuously striving for a 100% test coverage.

The staking module code is very readable and easy to follow, which is further enhanced by the specification that is explicitly well-defined. The code that implemented this specification is structured such that it is accessible and easy to comprehend. This is particularly important for future contributions and audits side by side to the code.

We recommend that these practices are continued as the codebase is expanded in the future.

Suggestions

No issues were identified during our review. If present, we list the issues we find in the code in the order we reported them. In addition, we include detailed suggestions based on team observations during the review that may not pose an immediate risk or threat but are considered best practice.

SUGGESTION	STATUS
Suggestion 1: OS Random Number Generation for Popular Operating Systems are Good Enough	Resolved

Suggestion 1: OS Random Number Generation for Popular Operating Systems are Good Enough

Synopsis

Tendermint implements a custom random number generator in *tendermint/crypto/random.go* .

The idea is to use a stream cipher (deterministic RNG) and xor its output with the OS rng and use that output as the random number (i.e. a one time pad where plaintext input is the OS rng output and the key is the output of chacha20 stream cipher). In theory, this approach is reasonable, however, the scheme is good (forward secrecy) only if it can be guaranteed that stream cipher output is truly random (i.e. the input Key, K to the stream cipher is totally random and never repeats).

The implementation in random.go appears to be incomplete (it has code for stream cipher encryption and re-seeding but not the xor parts). The new seed is taken from the stream cipher output, hashed (sha256), and used as the new key for the next use of chacha20. The randomness of the scheme rests in the initial seed being truly random. However, to achieve true randomness, that would mean reimplementing algorithms found in the OS kernel random number generator in order to create an entropy pool. Even with this approach, keeping the initial seed a secret would be difficult since it is being implemented in the user-space.

Mitigation

In general, we would recommend using the OS prng rather than implementing a custom rng from scratch.

Status

The MixEntropy functions have been removed from the file *tendermint/crypto/random.go* as of release v0.29.2.

Verification

Resolved

Recommendations

Per our recommendation, the *Issues and Suggestion* stated in the initial report were addressed or partially addressed and followed up with a verification by the auditing team. Additionally, we commend the Cosmos team for the code quality and recommend that the current development practices that resulted in this code are followed in the future. We recommend that continuous audits be conducted on future development releases to ensure that any potential issues and vulnerabilities are identified, addressed and verified.

Appendix 1: Activity Log

These are notes from the reviewers about their activities during the code audit. They detail the approach and investigative activities undertaken. All issues found are listed in the report. This is just for the purposes of transparency and could be helpful for another auditor to understand the evaluation activities.

Team Log:

2019-01-07

- read overviews about Practical BFT and the DLS ()Dwork, Lynch and Stockmeyer) consensus algorithms
- build the code
- failed on osx (didn't investigate)
- succeeded on Debian gnu/linux
- start reading the `cosmos-sdk/types/*.go` files
- coin.go: would be nice if denominations are a type instead of a string
- address.go: address comparisons are not constant time. Perhaps this is not a problem as it really depends where they are called from.
- read whitepaper and tendermint specs

2019-01-08

- continue with types/coin.go
- types/config.go
- types/decimal.go
- looking at `crypto` directory.

2019-01-09

- looking at the files in `crypto` directory. So, they are using _bcrypt_ for encrypting private keys using a pass phrase before writing it to the disk (keybase.go:writeLocalKey(), mintkey.go:encryptPrivKey()).
- also related files (tendermint/crypto/armor.go)
- _keybase.go_ looks pretty good.

2019-01-10

- `tendermint/crypto/hash.go`
- skim through amino spec. Looks like protobuf + support
- i see this pattern in the source:

  var _ crypto.PrivKey = PrivKeyEd25519{}

  So in this case, name is left empty and these declarations are global in a module. Perhaps this is done so that it is called when a module is imported?

  Turns out, my understanding of this usage was wrong. The above usage is meant to enforce an invariant that _crypto.PrivKey_ implement the interface PrivKeyEd25519{}

- tendermint/crypto/ed25519/ed25519.go - looks ok.
- tendermint/crypto/merkle - didn't quite go deep.
- read staking module specification

2019-01-11

- looked more closely at the tendermint/crypto/random.go's custom rng.

  The idea is to use a stream cipher (deterministic RNG) and xor its output with the OS rng and use that output as the random number. So, like a one time pad where plaintext input is the OS rng output and the key is the output of chacha20 stream cipher. In theory this looks okay though using the OS rng is just fine for all major OSes. However, the scheme is good (forward secrecy) only if we can guarantee that stream cipher output is truly random. i.e. the input Key, K to the stream cipher is totally random and never repeats.

  The implementation in random.go seem incomplete (it has code for stream cipher encryption and re-seeding but not the xor parts). The new seed is taken from the stream cipher's output and hashed (sha256) and used as the new key for the next use of chacha20, so it is in a way deterministic, given the initial seed bytes and nonce (which is zero in the current implementation).

  In general, we would recommend using the OS prng and not implement a custom rng from scratch.

- looking at tendermint/docs/spec/p2p/connection.md
- compare staking module spec to implementation, implementation is complete and correct

2019-01-14

- reviewing BaseApp

2019-01-15/16

- BaseApp and how it ties with other modules

2019-01-17/18

- read about proof of stake (ethereum faq on PoS is very good) and the need for slashing. Armed with that information, started reading the spec and code.
- nothing scary on the tombstone commit or the old slashing module.
- starting to look into f1 - getting a better sense of changes to staking / slashing module, distribution module, and changes made to the top level command line app using them.

2019-01-21/22

- review diff for staking spec upgrade
- continued working through distribution module (types, keeper, etc) and corresponding changes in staking and cli.
- ran tests and looked at test coverage. many functions have 100% test coverage which is great, there are some failed tests but since we are on the 'develop' branch, it is probably ok and someone is looking at these failures.

2019-01-23

- final review of team hack pad notes and slack
- working on audit report

2019-02-18/20

- Verification: pulled the latest cosmos-sdk code and tendermint code. All the custom rng code in tendermint/crypto/random.go has been removed.
- Drafting final report