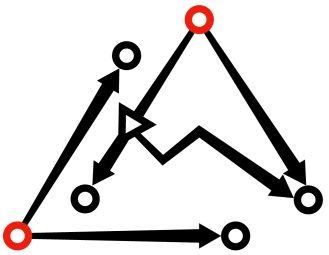

Least Authority

PRIVACY MATTERS

Lair Keystore

Security Audit Report

Holo Ltd

Final Audit Report: 23 September 2022

Table of Contents

Overview

Background

Holo Ltd has requested that Least Authority perform a security audit of the Lair Keystore. The Lair Keystore is the secure keystore that holds private keys and seed and performs any and all cryptographic functions, which need to use private keys for the various Holochain subsystems requiring decryption, encryption, or signing.

Project Dates

- **July 25 - August 26:** Initial Code Review *(Completed)*
- **August 31:** Delivery of Initial Audit Report *(Completed)*
- **September 19:** Verification Review *(Completed)*
- **September 23:** Delivery of Final Audit Report *(Completed)*

Review Team

- Steven Jung, Security Researcher and Engineer
- DK, Security Researcher and Engineer
- Nishit Majithia, Security Researcher and Engineer
- ElHassan Wanas, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Lair Keystore followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repositories are considered in-scope for the review:

- Sodoken:
  https://github.com/holochain/sodoken/tree/v0.0.1/crates/sodoken
- HC Seed Bundle:
  https://github.com/holochain/lair/tree/hc_seed_bundle-v0.1.1/crates/hc_seed_bundle
- Lair Keystore API:
  https://github.com/holochain/lair/tree/lair_keystore_api-v0.1.2/crates/lair_keystore_api
- Lair Keystore:
  https://github.com/holochain/lair/tree/lair_keystore-v0.1.2/crates/lair_keystore

Specifically, we examined the Git revisions for our initial review:

Holochain Lair: `A21c49be3a87e46b3fe968ab309f99299454e05c`

Holochain Sodoken: `47c3d3cb72d29be5e9dd95565463c99b6f66dde7`

For the review, these repositories were cloned for use during the audit and for reference in this report:

Holochain Lair:
https://github.com/LeastAuthority/holochain-lair

Holochain Sodoken:

https://github.com/LeastAuthority/holochain-sodoken

For the verification, we examined the Git revisions:

Holochain Lair: `8e565fe9aed1fb238d6f6174dc9fac4d3b1a8808`

Holochain Sodoken: `4bf8d4fdf75a77dbb195358c769584d362e307af`

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

In addition, any dependency and third-party code, unless specifically mentioned as in-scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- Sodoken: https://docs.rs/sodoken/0.0.1/sodoken/index.html
- HC Seed Bundle: https://docs.rs/hc_seed_bundle/0.1.1/hc_seed_bundle/index.html
- Lair Keystore API: https://docs.rs/lair_keystore_api/0.1.2/lair_keystore_api/index.html
- Lair Keystore: https://docs.rs/lair_keystore/0.1.2/lair_keystore_lib/index.html
- Holochain Security Review: https://hackmd.io/rfotIeMlT8SXbVwxEFOyyg?view#SCOPE-1-
- Holochain Developer Documentation https://developer.holochain.org/

In addition, this audit report references the following documents:

- Libsodium Audit Results https://www.privateinternetaccess.com/blog/libsodium-audit-results/

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation and adherence to best practices;
- Proper management of encryption and storage of private keys, including the key derivation process;
- Exposure of any critical information during user interactions with external libraries, including authentication mechanisms;
- Adversarial actions and other attacks, such as the manipulation of data;
- Vulnerabilities in the code as well as secure interaction between the related components;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Holochain is an open source network that enables users to create distributed applications. For this review, our team reviewed the Holo Lair Keystore, which is implemented in Rust and performs all cryptographic functions that need to use private keys, including providing a secure keystore for private keys. The Lair Keystore is composed of three main components: the `lair_keystore` is a library for creating and interacting with a Lair keystore, which includes the `lair_keystore_api` that enables client implementations of the keystore. The `hc_seed_bundle` implementation performs key generation and parsing functionality for the Lair Keystore. The Sodoken component creates secure memory (Buffer) that can be used for storing Lair keystore private keys securely.

Our team examined the design and implementation of the Holo Lair Keystore and found that from a security perspective, it is well designed and is implemented in adherence with best practice recommendations. Our team identified an issue and some suggestions that, if resolved, will improve the overall quality and security of the implementation.

System Design

Our team found that security has been taken into consideration in the design of the Holo Lair Keystore, as demonstrated by the implementation of secure memory in a Sodoken buffer, which mitigates the compromise of secret data as a result of disk swapping. The cryptographic implementation uses Blake2b and Argon2id (which are considered to be sufficiently secure hash functions), as well as the libsodium encryption library. Our team reviewed the cryptographic design and implementation and did not identify any vulnerabilities.

Code Quality

We found the code to be well organized and easy to read, generally adhering to Rust development best practices. Additionally, the codebase is structured in a way that demonstrates a clear separation of concerns. However, we identified several instances where a function was improperly defined as `public`. We recommend that functions be appropriately defined (Suggestion 3). We also found that panics are used in the drop trait, which does not adhere to Rust best practices. We recommend adherence to best practice to improve the quality and security of the implementation (Suggestion 2).

Tests

We found that sufficient test coverage has been implemented, which tests for success, failure, and edge case scenarios. This helps identify implementation errors and verify that the implementation functions as intended.

Documentation

Our team found the documentation provided by the Holo team to be generally sufficient and helpful in describing the intended functionality of the different components.

Scope

For this review, our team examined components that function within the Holochain system. We found the scope of this review to be sufficient for examining these components. Our team assumed that these components interact with the Holochain system as expected.

Dependencies

Our team identified the use of several outdated dependencies, which are known to contain security vulnerabilities. We recommend that the Holo team utilize well-maintained and audited dependencies and that dependencies be updated to the latest release to avoid bugs and issues (Suggestion 1).

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: TLS Private Key Is Not Zeroized	Resolved
Suggestion 1: Update and Maintain Dependencies	Resolved
Suggestion 2: Do Not Panic in Drop Implementation	Resolved
Suggestion 3: Make open_easy_msg_len Functions Private	Resolved

Issue A: TLS Private Key Is Not Zeroized

Severity

Medium

Location

`src/internal/tls.rs#L107`

`crates/lair_keystore_api/Cargo.toml#L23`

Synopsis

An attacker that is able to access memory (e.g., accessing core dump and exploiting vulnerabilities such as Heartbleed) may be able to retrieve non-zeroized TLS private keys. This is possible due to two reasons:

- `rcgen` is used without the zeroize feature; and
- Sodoken's buffer is created without *new_mem_locked*.

Impact

The leakage of cryptographic keys could result in the loss of security properties, such as confidentiality and privacy.

Preconditions

An attacker must be able to read memory regions that contain sensitive data.

Mitigation

We recommend enabling the `zeroize` feature for the `rcgen` crate and using `new_mem_locked` when creating Sodoken's buffers.

Status

The Holochain team has implemented suggestions to zeroize the TLS private key.

Verification

Resolved.

Suggestions

Suggestion 1: Update and Maintain Dependencies

Synopsis

`chrono`, `tempdir`, `rusqlite`, `serde_cbor`, `ansi_term`, `tokio`, `thread_local`, `regex`, `lru` and several other dependencies are outdated or have known vulnerabilities. A robust development process includes the regular maintenance and updates of dependencies in order to minimize the risk of exploiting known vulnerabilities from the codebase. To get the full list of the outdated or vulnerable dependencies, we suggest running the `cargo outdated` and `cardo audit` tools.

Mitigation

We recommend updating or replacing the reported dependencies. We also recommend updating the relevant upstream package in the event that a dependency is used by an upstream dependency. In addition, we recommend regularly running the `cargo audit` and `cargo outdated` tools.

Status

The Holochain team has updated the dependencies, as suggested.

Verification

Resolved.

Suggestion 2: Do Not Panic in Drop Implementation

Location

[lair_keystore/src/store_sqlite.rs#L56]

Synopsis

According to best practices, in a secure Rust development, the implementation of the `std::ops::Drop` trait must not panic.

Mitigation

We recommend avoiding the use of panic in the Drop trait.

Status

The Holochain team has removed the panic.

Verification

Resolved.

Suggestion 3: Make `open_easy_msg_len` Functions Private

Location

src/secretbox/xchacha20poly1305.rs#L49

src/secretbox/xsalsa20poly1305.rs#L49

src/crypto_box/curve25519xsalsa20poly1305.rs#L108

Synopsis

`Open_easy_msg_len` functions are declared `public` and can be called from outside the crate. At the same time, these functions are used in tests. Using these functions beyond specific tests can lead to the improper calculation of the length of messages and potential integer overflow.

Mitigation

We recommend making the functions listed above private.

Status

The Holochain team has removed the target functions.

Verification

Resolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, and zero-knowledge protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. We are an international team that believes we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our 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 protect 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 include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet 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. As 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 and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative and 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 before having 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 and comprehensive 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 Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, 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 a resolution that balances the impact on the users and the needs of your project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.