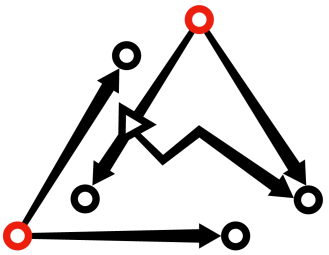

Least Authority

PRIVACY MATTERS

Auro Wallet Extension

Security Audit Report

Mina Foundation

Updated Final Audit Report: 6 August 2021

Table of Contents

Overview

Background

Mina Foundation requested that Least Authority perform a security audit of the Auro Wallet Extension, a browser extension wallet for the Mina Protocol. The Auro Wallet Extension aims to provide a more convenient way for users to participate in the Mina Network through secure local account storage, management of Mina assets, convenient and simplified staking, user-owned private keys, and a user-friendly interface.

Project Dates

- **May 19 - June 15**: Code review *(Completed)*
- **June 18**: Delivery of Initial Audit Report *(Completed)*
- **July 28 - 29:** Verification *(Completed)*
- **July 30:** Delivery of Final Audit Report *(Completed)*
- **August 6:** Delivery of Updated Final Audit Report *(Completed)*

Review Team

- Ann-Christine Kycler, Cryptography Researcher and Engineer
- Jan Winkelmann, Cryptography Researcher and Engineer
- Gabrielle Hibbert, Security Researcher and Engineer
- May-Lee Sia, Security Researcher and Engineer
- Mirco Richter, Cryptography Researcher and Engineer

Coverage

Target Code and Revision

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

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

- Auro Wallet Extension: https://github.com/bitcat365/auro-wallet-browser-extension

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

`C46ad68638085ffaee4ea4d27230f09c28bb9b6f`

For the verification, we examined the Git revision:

`990f595cd96820c6b39d8548332017b7d3ad46df`

For the updated verification, we examined the Git revision:

`1fbc0013e7ecc9f3db25bb8ff494e0020d57718b`

For the review, this repository was cloned for use during the audit and for reference in this report:

https://github.com/LeastAuthority/auro-wallet-browser-extension

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:

- J. Bonneau, I. Meckler, V. Rao, and E. Shapiro, 2020, "Mina: Decentralized Cryptocurrency at Scale" New York University, O(1) Labs [BMR+20]
- Mina Extension Document: mina_wallet_for_audit_0407-en.pdf (*provided by Bit Cat via Discord on 7 April 2021)*
- Auro Extension Wallet.png: https://github.com/LeastAuthority/auro-wallet-browser-extension/blob/main/docs/auro-extension-wallet.png

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation and adherence to best practices;
- Exposure of any critical information during user interactions with the blockchain and external libraries, including authentication mechanisms;
- Adversarial actions and other attacks that impact funds, such as the draining or the manipulation of funds;
- Mismanagement of funds via transactions;
- Vulnerabilities in the code, as well as secure interaction between the related and network components;
- Proper management of encryption and storage of private keys, including the key derivation process;
- 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

The Auro Wallet Extension is a Chrome and Firefox browser extension that enables users to generate a key pair using a BIP-39 library, store the private key locally, and perform common wallet functions. These functions include sending, receiving, and staking Mina assets, as well as account management. The repository in scope consists of a user interface (Auro UI) component and a backend component (Auro background), which includes security critical sub-components, such as local storage of private keys and Mina Network communication functionality.

The Auro Wallet team has been proactive in response to our questions and feedback throughout this security review, demonstrating that security is a strong consideration and priority in their development approach.

System Design

Our team performed a manual review of the codebase and examined all security critical components for security vulnerabilities and implementation errors.

To reinforce the overall security of the system, we encourage adherence to web application security best practices. We suggest utilizing secure web application resources, such as OWASP GraphQL Security and the OWASP Authentication Cheat Sheet, in order to promote implementation best practices according to generally accepted security guidelines.

Auro Background

Our team closely evaluated the Auro background component, in which encrypted private keys are stored. It is apparent that the Auro Wallet team has considered security in the design of this component, as demonstrated by requiring sensitive data in the browser extension storage to be encrypted. However, under feasible preconditions, we found that the user password, which grants the user access to the wallet extension functionality and decrypts the sensitive data in the browser extension storage, can be retrieved from the `memStore` of `APIservice.js`. We recommend that the user password be cleared from memory upon each user lock out (Issue A).

Furthermore, we found that the encryption library used to encrypt sensitive data stored in the browser extension uses a CPU-bound key derivation function, which could make low entropy user passwords brute-forceable. We recommend that a more secure key derivation function be implemented to protect against brute-force based attacks (Issue B).

Auro UI

We examined all user input fields in Auro UI and potential vulnerabilities to Cross-Site Scripting (XSS) attacks. We also investigated the security features of the Auro UI and found that the input field `Memo` in the pages Send and Stake allow for a GraphQL injection. Although we did not identify any scenario that would result in loss of funds, we recommend adhering to GraphQL best practices in order to avoid GraphQL injections (Issue C).

In reviewing Auro UI, we also found that the user is prompted by pop-ups to input the password to confirm certain requests. This password prompt is superfluous, since the password is available in plaintext in the background script state, which can be accessed using Chrome DevTools. These prompts may suggest a false sense of security to users and, as a result, we recommend that an alternative request confirmation mechanism be implemented (Issue D).

Our team also noted that user passwords, when stored for validation, are limited arbitrarily in length and whitespaces are removed. This practice reduces the security of the password and, as a result, we suggest that password handling in the system adheres to NIST best practices for memorized secret authenticators (Suggestion 5).

Network Communication

Our team examined the mechanism whereby the Auro Wallet browser extension communicates with the Mina Network to query information or broadcast transactions. Auro background makes API requests to a Mina node that is maintained by the Auro Wallet team. This current architecture requires the user to trust an external Mina node to provide correct information about wallet funds and the state of the network. At present, there is no mechanism in the Auro background to validate the veracity of data returned from this node. As a result, we suggest that the validation features of the Mina Network be utilized to enhance user and network security (Suggestion 1).

Code Quality

Our team found the Auro background component codebase to be well organized.

However, we found that the Auro UI codebase does not adhere to React coding best practices. We suggest that the GUI components and functionality components be grouped to facilitate maintenance and

review of the implementation, and that pages should be broken down further into distinct React components (Suggestion 2).

Furthermore, the code for both Auro UI and Auro-background is inconsistently formatted, with several instances of duplicate code and unused variables and functions, which make the code more difficult to reason about and comprehend. We recommend that all instances of duplicate code and unused variables and functions be removed and that a linter and code formatter be used to improve the readability of the code (Suggestion 4).

Tests

The Auro Wallet browser extension codebase includes five unit tests, and no integration tests or end-to-end tests have been written. In accordance with best practices, sufficient test coverage should test for all success and failure cases, which helps to identify potential edge cases and protect against errors and bugs that may lead to vulnerabilities or exploits. We recommend implementing a test suite, which includes a minimum of unit tests and integration tests. We also suggested end-to-end testing so that it can be comprehensively determined if the implementation behaves as intended (Suggestion 6).

Documentation

The Auro Wallet team has provided some documentation of the code and a diagram that explains the interactions between the JS code modules. However, documentation outlining the system design is currently unavailable. Furthermore, the existing documentation of the interaction between the Auro UI, Auro background, and the API node to be insufficient, as message passing functionality and GraphQL requests must be documented thoroughly in order to avoid implementation errors and to provide reviewers with a sufficient understanding of the system. In addition, there is no clear documentation on the Mina node to which the Auro Wallet browser extension sends API requests. Although this is configurable at build time, the default configuration sends API requests to a Mina node run by the Auro Wallet team. Documentation of this area is vital in evaluating the trust relationships between the different parties and components and the overall security of the system. As a result, we recommend the project documentation be improved to encompass these missing components (Suggestion 7).

Code Comments

The Auro Wallet code base has very few code comments that explain the intended functionality of the code. Code comments are the most basic form of documentation and should be comprehensive in documenting every function and entry point. We suggest the Auro Wallet team improve code comments to explain the intended functionality of all components to facilitate understanding and reasoning about security vulnerabilities (Suggestion 3).

Scope & Dependencies

The scope of this security review was sufficient and covered all security critical components of the system. We examined all dependencies implemented in the codebase, and generally found that well audited and maintained dependencies have been utilized correctly. However, we did identify one unmaintained dependency, the QR Code Generator, which was last updated in September 2019. The Auro Wallet team confirmed that the dependency has been pinned to the current latest version. As a result the system will not automatically update to a newer, potentially malicious version, and thereby follows best practices to prevent such attacks.

Unmaintained libraries pose a higher risk of being used as a vector for supply-chain attacks. Inactive maintainers are often willing to grant push access to anyone who volunteers to maintain the library, and it is difficult to verify the credibility of the new maintainer. As a result, careful maintenance and management of dependencies is critical to the security of any system.

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: Sensitive Data Not Cleared Upon Locking	Resolved
Issue B: Encryption Library Provides Insufficient Security for Low-Entropy Passwords	Resolved
Issue C: Memo Allows GraphQL Injections	Resolved
Issue D: Password Prompts on an Unlocked Wallet Can be Circumvented	Unresolved
Suggestion 1: Verify State Received From API Using SNARKs	Unresolved
Suggestion 2: Improve Auro UI Architecture	Unresolved
Suggestion 3: Improve Code Comments	Unresolved
Suggestion 4: Improve Code Readability	Unresolved
Suggestion 5: Improve Password Handling	Unresolved
Suggestion 6: Increase Test Coverage	Unresolved
Suggestion 7: Improve Project Documentation	Unresolved

Issue A: Sensitive Data Not Cleared Upon Locking

Location

`src/background/APIService.js`

Synopsis

The local password can be extracted after locking due to insufficient state clearing.

Impact

With access to the locked extension, an attacker can interact with the browser extension and extract the password from the internal state of the background script. This password can be used to decrypt the keystore, which stores the private key.

Preconditions

The attack requires an opportunity to interact with the extension, which requires physical access to the machine. Alternatively, a means of control of the inputs to the browser extension would be needed, which would require successful exploitation of the machine beforehand.

The extension must be opened and unlocked, and then locked again, storing the password in memory.

Feasibility

If the preconditions hold, the attack is feasible using the Chrome DevTools, which are part of every Chrome installation.

Technical Details

The password is stored in the `memStore` of the `APIService`. The internal state of this object can be inspected using the Chrome DevTools.

Remediation

We recommend clearing the password upon locking, which entails deleting the password from the `memStore` object.

Status

The Auro Wallet team has updated all code paths that lock the wallet, such that all sensitive data is deleted from the `memStore`.

Verification

Resolved.

Issue B: Encryption Library Provides Insufficient Security for Low-Entropy Passwords

Location

`src/background/APIService.js#L12`

Synopsis

The `browser-passworder` encryption library uses the PBKDF2 key derivation function using a SHA hash, which is purely CPU-bound. This means brute-force attacks can be sped up relatively easily using ASICs and FPGAs.

Impact

A successful brute force attack on the password based encryption compromises the private key.

Preconditions

The attacker has access to a locked, encrypted wallet.

Feasibility

Moderately feasible. The attack requires access to a hardware implementation (ASIC or FPGA) of `PBKDF2-HMAC-SHA256`. The initial investment for an FPGA implementation is low to moderate.

Technical Details

The `browser-passworder` encryption library uses PBKDF2 to derive a key from a password. PBKDF2 is a standard for iterated invocation of a keyed hash function. The function used in `browser-passworder` is HMAC-SHA256. This function is purely CPU-bound, and can therefore be sped up using specialized hardware that has little memory. Such hardware is not very expensive and very energy efficient. In order to protect against brute-force attacks carried out with the help of such devices, a memory-hard function should be used. This makes brute-force attacks based on specialized hardware infeasible, because they require access to a considerable amount of fast memory. This increases both the hardware cost and the amount of energy required for the derivation, which minimizes the gap to the efficiency of CPUs.

Remediation

We recommend using the Argon2id function for deriving keys from passwords. There are two possible approaches:

1. Provide a patch to `browser-passworder` to support Argon2id (see this GitHub Issue).
2. Derivation and encryption manually.

For deriving a key from the password, we recommend this Argon2id function library implemented with a memory parameter of 64 MB. Additionally, an iteration count (called OPSLIMIT by sodium) of 3 should be used. It is important to note that a longer processing time provides better protection against brute-force attacks.

For encryption of the browser extension storage, we recommend using the secretbox functions from libsodium.

Status

The Auro Wallet team has added code for a new encryption system based on Argon2id and AES-GCM. They also added automatic migration logic, in order to convert encrypted wallets in the old format to the new format. The security parameters of the Argon2 invocation fit the use case. For the base64 conversion, a well-maintained, web-compatible implementation of the Buffer object from Node.js is being used.

Verification

Resolved.

Issue C: Memo Allows GraphQL Injections

Location

`pages/StakingTransfer/index.js#L150-L156`

`pages/Send/index.js#L354-L360`

`background/api/gqlparams.js#L55-L156`

Synopsis

The input fields from the Send and Staking screens are inserted into the GraphQL mutation strings without validation or sanitization. If a user inserts a double-quote (“) character into a memo field, they can end the GraphQL field that constrains the results and insert additional GraphQL code.

Impact

There are several strings that, when entered as a Memo string, result in the submission of an invalid transaction, which would be rejected by the Mina Network. We did not find a string that would result in a changed destination or amount.

Preconditions

The attacker is able to choose the contents of the memo field.

Feasibility

Depending on the context, this is very feasible. For example, a merchant may require a specific memo for matching the payment to the order. This may not be a deliberate attack, however, most strings containing quotes will fail.

Technical Details

The memo field can be filled with arbitrary text. This text will then simply be inserted into the GraphQL mutation string, between two quotation marks. Since that text can also contain quotation marks, the structure of the GraphQL mutation can be escaped and thereby modified. This is referred to as GraphQL injection.

In this case, we did not find a way to perform an attack with a very strong outcome, because the signature, which is also included in the mutation, is only valid for the correct data. Therefore, we only found attacks which resulted in failed transactions.

Such injections can be prevented by properly using the GraphQL named mutations and variables features.

Remediation

We recommend adhering to the best practice guidelines provided in the GraphQL documentation. Steps to avoid GraphQL injections include:

- Use accepted naming conventions and avoid conventions where a mutation is named "`MyMutation`";
- Use variables instead of inserting strings directly;
- Mutations must be a static string; and
- Remove the `startFetchMyQuery` function, then map the proper values to these variables using the `variables` parameter of `fetchGraphQL`.

Status

The Auro Wallet team has updated the code such that the generation of query and mutation strings is minimized (e.g. through the use of GraphQL variables), which prevents GraphQL injections of any kind.

Verification

Resolved.

Issue D: Password Prompts on an Unlocked Wallet Can be Circumvented

Location

`src/popup/pages/SecurityPwdPage/index.js`

`src/background/APIService.js#L60`

Synopsis

All password prompts that are presented when the wallet is already unlocked can be circumvented, since the wallet password is stored in the background script state, which can be accessed using the Chrome DevTools.

Impact

A user may not notice that it is possible to extract the private key from any unlocked wallet and, given this false sense of security, may be less careful than is warranted. This may lead to the compromise of their secret key and/or mnemonic, or the change of the password by an attacker.

Preconditions

The wallet must be unlocked.

Feasibility

The password can be easily extracted.

Technical Details

The password is stored in the `memStore` of the `APIService`. The internal state of this object can be inspected using the Chrome DevTools. Unfortunately, this is an inherent limitation of the execution environment and cannot be prevented.

In order to decide on what action is to be taken in order to reduce the risk of leaking the secret key and seed, two attack scenarios have to be considered, as detailed below.

First, consider a scenario where the password prompt remains in the wallet and a user believes it is secure. The user temporarily (because the user trusts the security of the password prompt) gives an attacker access to the wallet UI, and afterwards checks that no transactions were made. An attacker who is aware of this weakness and has experience with web extension development and debugging will be able to extract the secret key without the user noticing.

Second, consider a scenario where the password prompt is removed. The user is aware that they should not leave the wallet unlocked, but due to a lapse in operational security, the user leaves it open during a window of opportuning where the attacker has access to the wallet. The attacker does not need specialized knowledge to export the secret key.

The action to be taken is a trade-off between these risks. Both are real risks and an evaluation needs to happen in the context of the application and user base, which is ultimately a decision to be made by the software vendor.

Mitigation

One mitigation is to lock the wallet during these password prompts. Whenever an attacker first tries to export the keys through the UI (instead of immediately going for a DevTools-based attack), the wallet would be locked and the attack prevented. This would prevent UI-based attacks and secure the password prompt. However, it would still be vulnerable to attacks where the attacker immediately uses the DevTools to exfiltrate the secret key or seed.

The user should, if possible, use a hardware wallet, as they categorically prevent the accessing and exfiltration of secret keys. If that is not possible in the specific context of the user, the user should be very careful not to leave the wallet unlocked.

Status

The Auro Wallet team has decided not to implement the mitigation as they feel this would reduce the usability of the application. Specifically, if the mitigation is implemented and the user aborts the password prompt when attempting to view the mnemonic, they could not go back to the home screen without unlocking the wallet again. We encourage the Auro Wallet team to continue to evaluate solutions to resolve this issue. We also recommend that the Auro Wallet team recommend users to utilize a hardware wallet.

Verification

Unresolved.

Suggestions

Suggestion 1: Verify State Received From API Using SNARKs

Synopsis

In order to query information about accounts, the Auro Wallet browser extension makes API requests to a Mina node operated by the Auro Wallet team. The wallet extension trusts that the returned results are true, similar to API wallets for other blockchains. However, the central promise of Mina is that the blockchain is small and that state can be verified efficiently on any device. A wallet that fully exploits the potential of Mina would make use of the zero knowledge proofs, such that light-clients can verify the veracity of the responses of the API node.

However, since this functionality is not currently implemented in the JavaScript APIs of Mina libraries, this is not currently possible for a browser wallet extension.

Mitigation

Once the Mina Client APIs support trustless verification of the data returned by the API node, we recommend making use of this feature and verify the veracity of the data returned from the API.

Status

The Auro Wallet team responded they will address this suggestion in the next version of the wallet. As such, the suggestion remains unresolved at the time of the verification.

Verification

Unresolved.

Suggestion 2: Improve Auro UI Architecture

Location

Examples, not exhaustive:

`pages/Send/index.js#L337`

`pages/StakingTransfer/index.js#L134`

`pages/Wallet/index.js#L108`

Synopsis

The use of small, reusable components, and composing the pages out of these components is a React best practice. In addition, business logic should be isolated from the Auro UI code as much as possible.

Mitigation

We recommend adhering to GUI and React coding best practices.

Status

The Auro Wallet team responded they will address this suggestion in the next version of the wallet. As such, the suggestion remains unresolved at the time of the verification.

Verification

Unresolved.

Suggestion 3: Improve Code Comments

Location

`src/background/APIService.js`

Synopsis

There are currently very few comments throughout the codebase, and the comments that are present repeat the function and method names. The documentation contained within the code should be comprehensive and document every function and entry point, in addition to explaining the intended functionality of each of the components. This allows both maintainers and reviewers of the codebase to comprehensively understand the intended functionality of the implementation and system design, which increases the likelihood for identifying potential errors which may lead to security vulnerabilities.

Mitigation

We recommend creating code comments that explain each variable, function, and entry point.

Status

The Auro Wallet team responded they will address this suggestion in the next version of the wallet. As such, the suggestion remains unresolved at the time of the verification.

Verification

Unresolved.

Suggestion 4: Improve Code Readability

Location

`background/api/index.js#L30`

`background/api/index.js#L76`

`background/api/gqlparams.js#L24`

`pages/Send/index.js#L262-L263`

`pages/Send/index.js#L399-L400`

Synopsis

The code contains unused variables, unused functions, duplicate code, and is not consistently formatted. In addition, variables and functions often have names that are not intuitive or relay the purpose of the function and variable. These inconsistencies with coding best practices decrease the readability of the code, which makes it more difficult for reviewers to identify errors and security vulnerabilities.

Mitigation

We recommend using the linter output to improve the code (the linter ESLint is already part of the project dependencies). We recommend including the linter in the Continuous Integration (CI) pipeline to flag commits that introduce linter warnings. A code formatter like Standard JS helps with consistent code formatting. Additionally, we recommend manual auditing and refactoring to eliminate redundant or unused code and improve naming.

Status

The Auro Wallet team responded they will address this suggestion in the next version of the wallet. As such, the suggestion remains unresolved at the time of the verification.

Verification

Unresolved.

Suggestion 5: Improve Password Handling

Location

src/utils/validator.js

pages/Lock/index.js

pages/ResetPassword/index.js

pages/SecurityPwdPage/index.js

Synopsis

Editing the password by removing whitespaces decreases the security of the password. If a user starts their password with a space character, the application will also accept a password that does not contain that space character. Given that there is no resource limitation, passwords should be allowed to be longer than 32 characters.

Mitigation

We recommend following the NIST Guidelines, which are industry standard rules and best practices for handling passwords when building memorized secret authenticators.

Status

The Auro Wallet team responded they will address this suggestion in the next version of the wallet. As such, the suggestion remains unresolved at the time of the verification.

Verification

Unresolved.

Suggestion 6: Increase Test Coverage

Location

test

Synopsis

The Auro Wallet contains only five unit tests in the test suite, and no integration tests or end-to-end testing. During our review, we manually found an error that could have been avoided if there were a test written for the function, as detailed below.

In

pages/Send/index.js#L299-L303

```
let maxAmount = new BigNumber(amount).plus(fee).toString()
```

```
if (new BigNumber(amount).gt(maxAmount)) {
  Toast.info(getLanguage('balanceNotEnough'))
  return
}
```

`Checks whether amount > amount + fees which is always false.` Instead, it should check whether `amount + fees > walletBalance.`

Mitigation

We recommend that the Auro Wallet team add unit tests and use Test Driven Development (TDD) for all important checks and functions.

In addition, we recommend adding integration tests to test interaction with services outside of the Auro Wallet and adding end-to-end tests for the most critical user behaviors.

Status

The Auro Wallet team responded they will address this suggestion in the next version of the wallet. As such, the suggestion remains unresolved at the time of the verification.

Verification

Unresolved.

Suggestion 7: Improve Project Documentation

Location

mina_wallet_for_audit_0407-en.pdf

README.md

Synopsis

The existing project documentation for the Auro Wallet includes some documentation on the code, as well as one diagram that explains how different JS code modules interact with each other. However, the general project documentation for the Auro Wallet Extension is not sufficient in describing the system, and the intended functionality of each of its components.

Sufficient documentation provides a high-level description of the system, each of the components, and interactions between those components, allowing reviewers to assess the in-scope components and understand the expected behavior of the system being audited. Additionally, comprehensive user documentation helps to ensure users interact with the system correctly and as intended, which encourages secure and correct usage.

Mitigation

We recommend that the Auro Wallet team expand the general architecture and system design documentation, create documentation of the interaction between the Auro UI, Auro background script, and the Mina API node, including message passing functionality and GraphQL requests and documentation on the Mina node to which the Auro Wallet browser extension sends API requests.

In addition, we recommend creating comprehensive user documentation, which helps to ensure users interact with the system correctly and as intended.

Status

The Auro Wallet team responded they will address this suggestion in the next version of the wallet. As such, the suggestion remains unresolved at the time of the verification.

Verification

Unresolved.

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 C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. 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. Although we are a small team, we believe that 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 included manual code analysis, user interface interaction, 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 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.