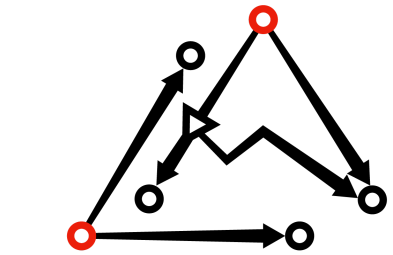

Least Authority

PRIVACY MATTERS

Mobile Application

Security Audit Report

Rabby Wallet

Final Audit Report: 2 September 2025

Table of Contents

Overview

Background

Rabby Wallet has requested that Least Authority perform a security audit of their Mobile Application to assess developments since the previous review, for which a final audit report was delivered on October 18, 2024.

Project Dates

- **August 11, 2025 - 21 August, 2025:** Initial Code Review *(Completed)*
- **August 22, 2025:** Delivery of Initial Audit Report *(Completed)*
- **September 1, 2025:** Verification Review *(Completed)*
- **September 2, 2025:** Delivery of Final Audit Report *(Completed)*

Review Team

- Paul Lorenc, Security Researcher and Engineer
- Michael Rogers, Security Researcher and Engineer
- Burak Atasoy, Project Manager
- Jessy Bissal, Technical Editor

Coverage

Target Code and Revision

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

The following code repository is considered in scope for the review:

- https://github.com/RabbyHub/rabby-mobile

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

- `932ec85044bbd806b0ac32660d0e4274e2944401`

For the verification, we examined the Git revision:

- `2069644732985a2bc60d0df521d125db5365686b`

We focused on the following changes, as requested by the Rabby Wallet team:

- https://github.com/RabbyHub/rabby-mobile/pull/560
- https://github.com/RabbyHub/rabby-mobile/pull/907

Additionally, we reviewed all modifications to the codebase since the previous Least Authority audit to identify any changes likely to affect the security of the application:

- https://github.com/LeastAuthority/rabby-wallet/pull/2

Specific changes that were considered likely to be security-relevant are identified in this report.

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

- https://github.com/LeastAuthority/rabby-wallet/tree/audit2

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:

- Website:
  https://rabby.io
- Previous audit reports:
  - SlowMist Audit Report - Rabby mobile wallet iOS.pdf *(shared with Least Authority via email on 5 August 2024)*
  - SlowMist Audit Report - Rabby mobile wallet Android.pdf *(shared with Least Authority via email on 5 August 2024)*
  - Least Authority Audit Report - Rabby Wallet:
    https://leastauthority.com/wp-content/uploads/2024/10/Least-Authority-Debank-Rabby-Wallet-Final-Audit-Report.pdf
  - Least Authority Audit Report - Rabby Wallet Extension:
    https://leastauthority.com/wp-content/uploads/2024/12/Least-Authority-DeBank-Rabby-Wallet-Extension-Final-Audit-Report.pdf

In addition, this audit report references the following documents:

- EIP-1193: Ethereum Provider JavaScript API:
  https://eips.ethereum.org/EIPS/eip-1193
- OWASP Password Storage Cheat Sheet:
  https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html
- GitHub Advisory for CVE-2025-27789:
  https://github.com/advisories/GHSA-968p-4wvh-cqc8

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Adversarial actions and other attacks on the wallet;
- Attacks that impact funds, such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Malicious attacks and security exploits that would impact the wallet;
- Vulnerabilities in the wallet code and whether the interaction between the related network components is secure;
- Exposure of any critical or sensitive information during user interactions with the wallet and use of external libraries and dependencies;
- Proper management of encryption and storage of private keys;
- 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

Our team performed a security audit of the Rabby Wallet Mobile Application, which supports multiple hardware and software wallets for the Ethereum blockchain. The mobile application can manage and host dApps from numerous providers and enables users to interact with the hosted dApps through a unified interface. We previously audited Rabby Wallet and delivered a final audit report on October 18, 2024. Since then, several features have been introduced, including the ability to synchronize accounts between the Rabby browser extension and mobile application and the ability to back up seed phrases to the cloud.

System Design

Our team examined the design of the mobile application and found that it has clearly been developed with security in mind. It incorporates several security features including encrypting data stored on the user's device or in the cloud, requiring a password to unlock the application, automatically locking the application when the user is not interacting with it, and clearing sensitive information from memory when the application is locked.

The application also contains an in-app browser that can host dApps interacting with it via the standard EIP-1193 API. It applies restrictions to dApp actions, with controls enforced depending on whether a dApp is currently in the foreground. User confirmation is required for potentially sensitive actions, such as connecting a wallet to a dApp or signing a transaction.

Specific Components Examined

We began our review with an initial focus on the two changes highlighted by the Rabby Wallet team as requiring particular attention: the storage of unencrypted information about hardware wallets on the user's mobile device, and the use of the `@scure/bip39` cryptography library to improve performance.

1. Storage of unencrypted information about hardware wallets. Our team identified one issue with the storage of unencrypted information about hardware wallets (Issue C).
2. Use of the `@scure/bip39` cryptography library. We did not identify any issues with the use of the `@scure/bip39` cryptography library, including patches applied by the Rabby Wallet team. However, we also report a related issue that predates the use of the new library (Issue A).

We subsequently reviewed the changes to the mobile application since the previous audit, for which we delivered a final audit report on October 18, 2024. In this process, we identified several changes with potential security implications that warrant closer examination.

3. Patch applied to the `@ledgerhq/hw-app-eth` library. We examined the Rabby Wallet team's patch for the `@ledgerhq/hw-app-eth` library, which changes the way certain error conditions are handled. The Rabby Wallet team explained that the purpose of the patch is to handle network timeouts that may occur when fetching data from the Internet. We did not identify any issues with the patch.
4. Patch applied to the `@metamask/browser-passworder` library. We examined the Rabby Wallet team's patch for the `@metamask/browser-passworder` library, which changes the way keys are derived from passwords. We did not identify any issues with the patch, although the purpose of the change remains unclear.
5. Changes to deep linking configuration. We examined changes to the way links from outside the application are handled. We did not identify any issues.
6. Changes to unlocking logic. We examined changes to the logic for unlocking the application, which impose a waiting period if several unsuccessful attempts are made to unlock the

application in a short period of time. The code implementing the waiting period is complex, but we did not find any issues.

7. Handling of mnemonics. We reviewed the code that handles mnemonics and derives private keys from them. We did not identify any issues.
8. RPC middleware. We examined the code that restricts the permitted interactions between dApps and the user's accounts. We did not identify any issues with this code, although our review led to the discovery of Issue D.
9. In-app browser. We reviewed changes to the in-app browser to support multiple tabs and identified an issue with the storage of screenshots (Issue E).
10. Cloud backup. We examined the code for backing up seed phrases to the cloud and discovered a potential privacy issue (Issue B). We further note that the cloud backup component logs data to the console on production (Suggestion 3).
11. Screenshot settings. We reviewed the code that prevents the user from taking screenshots or screen recordings of the application. We did not identify any issues.
12. Merging vaults. We reviewed the logic for merging account details when accounts stored in the browser extension and mobile application are synchronized. We did not identify any issues.
13. Ledger integration. We examined changes to the code for interacting with Ledger hardware wallets. We did not identify any issues. However, we note that there is a "`FIXME`" comment in the Ledger Keyring introduced as a temporary fix for a Ledger device issue. We suggest investigating whether Ledger has resolved this issue so that this section of code may be removed, thereby reducing the overall complexity of the Ledger Keyring component.

Dependencies

The Rabby Wallet team relies on patching upstream dependencies to provide functionality required by the application. The repository contains 21 patched dependencies. Similar to forking, patching places responsibility on the development team to track upstream changes, determine whether updates are necessary, and assess whether modifications to the patched code's semantics should be considered.

Additionally, our analysis identified one vulnerable dependency in the project, and we recommend updating it while adopting a secure dependency management process to reduce the risk of supply-chain attacks (Suggestion 2).

Code Quality

The overall structure of the codebase is well-organized, with code divided into modules with well-defined purposes. However, the quality of the application code is inconsistent. We found many instances of misspelled names for variables, functions, and classes. Sections of code are duplicated, unused, or commented out. The code contains minor errors highlighted by static analysis tools, such as the omission or unnecessary use of the `await` keyword.

Some of the code uses a mixture of English and Chinese for comments and developer-facing outputs. Such practices risk impairing the maintainability of the codebase if not all developers are fluent in both languages.

Tests

Our team found the test coverage of the repository to be insufficient. The codebase contains only `5,000` lines of test code out of a total of `289,000` lines of code, and some features appear to be intended for manual testing but are disabled in production builds. A robust test suite should include unit and integration tests covering both success and failure cases to help identify errors and protect against potential edge cases, which may lead to security-critical vulnerabilities or exploits. We recommend improving test coverage (Suggestion 1).

Documentation and Code Comments

The project's documentation and code comments remain limited, with only about 800 lines of documentation excluding change logs. This provides little insight into the design or technical choices and reduces readability, making it more difficult to reason about the security of the components reviewed. We recommend that documentation and comments be extended to adequately cover the newly introduced features.

Scope

The scope of this audit was limited to the changes to the mobile application since the previous audit by our team, for which a final report was delivered on October 18, 2024.

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: Weak Key Derivation Function and Parameters	Unresolved
Issue B: Wallet Addresses Are Not Encrypted in Cloud Backups	Unresolved
Issue C: Hardware Wallet Addresses Are Not Encrypted in Mobile Device Storage	Unresolved
Issue D: In-App Browser Remains Open Over Application Lock Screen	Resolved
Issue E: Screenshots Are Not Encrypted in Mobile Device Storage	Unresolved
Suggestion 1: Improve Test Coverage	Planned
Suggestion 2: Update Vulnerable Dependencies	Unresolved
Suggestion 3: Remove Logging for Production	Resolved

Issue A: Weak Key Derivation Function and Parameters

Location

`apps/mobile/src/core/services/encryptor.ts`

Synopsis

The encryption used to store seed phrases and other account details on the user's mobile device and in cloud backups relies on a weak key derivation function with weak parameters.

Impact

High.

If an attacker were to obtain the encrypted seed phrases and successfully decrypt them, they would have full control over the user's wallet, including the ability to transfer the user's funds to their own account.

Feasibility

Low.

To obtain the application data stored on the user's mobile device, the attacker would either need to exploit the Rabby Mobile Wallet Application to execute code in the context of the application, or run code with root privileges on the mobile device. In the case of a rooted or jailbroken mobile device, this might be done by first persuading the user to install an application controlled by the attacker, and then persuading the user to grant root access for some apparently legitimate reason.

In the case of an Android device with disk encryption disabled (which is an uncommon configuration in devices released in recent years), an attacker might be able to obtain the application data from a lost, discarded, or stolen device without compromising any applications.

To obtain the data stored in a Google cloud backup, the attacker would need to gain access to the user's cloud account and then use the Google Drive API to download the application-specific data for the Rabby Mobile Wallet Application. This, in turn, would require extracting some API parameters from the Android application package.

Severity

Medium.

Preconditions

The attacker would need to obtain the encrypted seed phrases stored on the user's mobile device or in a cloud backup.

Technical Details

The encryption key used for encrypting seed phrases and other account details is derived from the user's application password using the key derivation function PBKDF2 with `5,000` iterations of `SHA-256`. This key derivation function is not adequately secure with such a low number of iterations. An attacker who managed to obtain the encrypted seed phrases might be able to guess the encryption key by brute force, as only a small amount of computation is needed to check each guess.

There are no restrictions on the application password other than a minimum length of eight characters, so many users are likely to choose weak passwords, such as dictionary words or sequences of repeated or consecutive digits. By trying these weak passwords first, an attacker's chances of guessing the encryption key increase significantly.

Remediation

Our team highlighted this issue in the previous audit report, at which time it was only limited to data stored on the user's mobile device, but did not affect cloud backups. The Rabby Wallet team decided not to address the issue due to performance concerns with using a stronger key derivation function or a higher number of iterations. However, we note that the application also uses PBKDF2 when synchronizing accounts between the browser extension and the mobile application, and in that context `900,000` iterations of `SHA-256` are used with no observed performance degradation. This suggests that there is substantial capacity for increasing the iteration count used for data stored on the mobile device and in cloud backups without introducing significant performance overhead.

In line with OWASP guidelines, we recommend replacing PBKDF2 with a memory-hard function such as Argon2id. If PBKDF2 must be retained for performance reasons, then we recommend strengthening the parameters to at least `600,000` iterations of `HMAC-SHA-256`.

Status

The Rabby Wallet team stated that they must conduct additional benchmarking for more than `600,000` iterations, particularly on Android devices, before making decisions. As this process may be lengthy, they noted that the fix is unlikely to be included at this time.

Verification

Unresolved.

Issue B: Wallet Addresses Are Not Encrypted in Cloud Backups

Location

`apps/mobile/src/core/utils/cloudBackup.ts#L68`

`apps/mobile/src/core/utils/cloudBackup.ts#L20`

Synopsis

When backing up a seed phrase to the cloud, the wallet address is used as the filename, exposing the user's wallet addresses to anyone with the ability to access the user's cloud account.

Impact

Low.

This issue affects the user's privacy by allowing an attacker to see the user's wallet addresses and link them to the identity associated with the user's cloud account.

Feasibility

Low.

As noted in Issue A, to obtain the data stored in a Google cloud backup, the attacker would need to gain access to the user's cloud account and then use the Google Drive API to download the application-specific data for the Rabby Mobile Wallet Application. This would, in turn, require extracting some API parameters from the Android application package.

Severity

Low.

Preconditions

An attacker would need to gain access to the user's cloud account in order to view the filenames containing the user's wallet addresses.

Technical Details

Each seed phrase that is backed up to the user's cloud account is encrypted and stored in a separate file. The filename is the wallet address associated with the seed phrase.

Remediation

A simple remediation would be to use a random filename for each backup file. However, if a user repeatedly backed up the same seed phrase, a new file would be created each time, which might

eventually impact backup restoration performance. Duplicate seed phrases would be detected and discarded after decryption, so the existence of multiple files would not result in duplicate keyring entries.

To avoid this drawback, we recommend storing a salt value in the backup folder and deriving each filename from the corresponding wallet address by using a suitable key derivation function, such as Argon2id, with the wallet address and salt as its inputs.

Status

The Rabby Wallet team stated that the proposed remediation could enable associations between backup files and addresses but considered this only a minimal privacy risk. For this reason, they noted that they will not address the issue.

Verification

Unresolved.

Issue C: Hardware Wallet Addresses Are Not Encrypted in Mobile Device Storage

Location

`packages/service-keyring/src/keyringService.ts#L759`

`packages/service-keyring/test/mergeValut.data1.ts#L96`

Synopsis

When seed phrases and other account data are stored on the user's mobile device, some accounts are excluded from encryption to allow them to be used while the application is locked. The information excluded from encryption may consist of wallet addresses.

Impact

Low.

This issue affects the user's privacy by allowing an attacker to view the user's wallet addresses for certain hardware wallets.

Feasibility

Low.

As noted in Issue A, to obtain the application data stored on the user's mobile device, the attacker would either need to exploit the Rabby Mobile Wallet Application itself in order to execute code in the context of the application, or need to run code with root privileges on the mobile device. In the case of a rooted or jailbroken mobile device, this might be achieved by first convincing the user to install an application controlled by the attacker and then persuading the user to grant root access to the application for some apparently legitimate reason.

In the case of an Android device with disk encryption disabled (which is an uncommon configuration in devices released in recent years), an attacker might be able to obtain the application data from a lost, discarded, or stolen device without compromising any applications.

Severity

Low.

Preconditions

An attacker would need to gain access to the application data stored on the user's mobile device in order to access the hardware wallet addresses.

Technical Details

For most wallet types, information about the wallet is encrypted with a key derived from the application password before being stored on the user's mobile device. This prevents the information from being accessed while the application is locked. However, information about hardware wallets such as `Ledger` and `OneKey` is excluded from encryption to allow these wallets to be used while the application is locked. Private keys are not exposed since they remain on the corresponding hardware device, whereas wallet addresses can be observed.

Remediation

We recommend holding the unencrypted wallet information in the mobile device's memory but not storing it on disk. This would require the user to unlock the application once after each restart (for example, after rebooting the mobile device) so that the wallet information could be read from disk and decrypted.

Alternatively, depending on the types of interactions needed between dApps and hardware wallets while the application is locked, we recommend excluding wallet addresses from the information stored in unencrypted form.

Status

The Rabby Wallet team stated that the issue poses no security risk and only a minimal privacy risk, so they will not address it.

Verification

Unresolved.

Issue D: In-App Browser Remains Open Over Application Lock Screen

Location

`apps/mobile/src/screens/Unlock/Unlock.tsx`

`apps/mobile/src/core/bridges/middlewares/RPCMethodMiddleware.ts#L108`

`apps/mobile/src/core/controllers/rpcFlow.ts#L72`

Synopsis

If the in-app browser is open when the application automatically locks, the browser remains open over the application lock screen and can still be used.

Impact

Low.

The issue has two potential impacts: private information about the user's activity in dApps may be exposed while the application is locked, and dApps running in the in-app browser may be able to perform operations that should not be permitted while the application is locked.

Our team did not identify a method to exploit this issue from the perspective of a dApp running in the in-app browser. However, the attack surface exposed by the application to dApps running in the browser is large, and we cannot rule out the possibility that an exploit path exists.

Feasibility

High.

The issue occurs reliably if the in-app browser is open when the application automatically locks, even if this occurs while the device's screen is turned off. Users are likely to encounter this issue by chance. This behavior was observed on Android and confirmed through reproduction on an additional device.

Severity

Low.

Technical Details

The mobile application automatically locks when the user has not interacted with it for a configurable amount of time. However, if the in-app browser is open when this happens, it remains open over the application lock screen and can still be used.

The actions that can be taken by dApps running in the in-app browser are restricted by the mobile application, with tighter restrictions applying to dApps that are not currently in the foreground. However, if the application automatically locks while the in-app browser is visible, the dApp running in the active tab is still considered to be in the foreground. It is possible to switch tabs and open new tabs without unlocking the application.

Attempting to use an RPC method such as `eth_requestAccounts` from a dApp while the application is in this state results in a crash. After restarting the application, however, the in-app browser remains open. A second attempt to use an RPC method at this stage is subject to checks by the application's middleware as usual, but we note that because the browser tab is still considered active, the checks that are applied may not be the appropriate ones for the circumstances. The code performing these checks contains several to-do comments, suggesting that the developers may not be fully confident that the checks are complete and correct.

Remediation

We recommend closing the in-app browser when the application automatically locks, thereby protecting the user's activity in dApps from exposure and allowing the appropriate restrictions to be applied to RPC requests made by dApps.

Status

The Rabby Wallet team has implemented the remediation as recommended.

Verification

Resolved.

Issue E: Screenshots Are Not Encrypted in Mobile Device Storage

Location

`components/BrowserTab/index.tsx#L291`

`core/services/browserService.ts#L378`

Synopsis

Screenshots of the in-app browser are captured automatically and stored on the user's mobile device without being encrypted. These screenshots may contain information about private activity in dApps.

Impact

Low.

The issue impacts the user's privacy by allowing an attacker to view information about the user's activity in dApps.

Feasibility

Low.

Screenshots are captured and stored during normal usage of the application. However, as noted in Issue A, to obtain the application data stored on the user's mobile device, the attacker would either need to exploit the Rabby Mobile Wallet Application itself in order to execute code within the application's context, or run code with root privileges on the mobile device. In the case of a rooted or jailbroken mobile device, this might be achieved by first convincing the user to install an application controlled by the attacker and then persuading the user to grant root access to the application for some apparently legitimate reason.

In the case of an Android device with disk encryption disabled (which is an uncommon configuration in devices released in recent years), an attacker could obtain the application data from a lost, discarded, or stolen device without compromising any applications.

Severity

Low.

Preconditions

An attacker would need to gain access to the application data stored on the user's mobile device in order to access the screenshots.

Technical Details

The mobile application contains an in-app browser for hosting dApps. The browser can have multiple tabs open at any time. Screenshots of tabs are shown in the browser's user interface to assist the user in switching between tabs. These screenshots are captured automatically and stored in the filesystem on the user's mobile device. The screenshots are stored without encryption and may contain private information about the user's activity in dApps.

Remediation

We recommend encrypting the screenshots before storing them. Alternatively, the screenshots could be retained in memory only, without being written to the filesystem.

Status

The Rabby Wallet team stated that the issue poses no security risk and only a minimal privacy risk, so they will not address it.

Verification

Unresolved.

Suggestions

Suggestion 1: Improve Test Coverage

Synopsis

There is insufficient test coverage implemented to test the correctness of the implementation and that the system behaves as expected. Tests help identify implementation errors, which could lead to security vulnerabilities.

Sufficient test coverage should include tests for success and failure cases (all possible branches), which helps identify potential edge cases, and protect against errors and bugs that may lead to vulnerabilities. A test suite that includes sufficient coverage of unit tests and integration tests adheres to development best practices. In addition, end-to-end testing is also recommended to assess if the implementation behaves as intended.

Mitigation

We recommend that comprehensive unit test coverage be implemented in order to identify any implementation errors and to verify that the implementation behaves as expected.

Status

The Rabby Wallet team has acknowledged this suggestion and noted that, while the recommended mitigation will not be implemented at this time, it will be taken into consideration for future releases.

Verification

Planned.

Suggestion 2: Update Vulnerable Dependencies

Location

`package.json`

`apps/mobile/package.json`

`packages/rn-webview-bridge/package.json`

Synopsis

Analyzing the project's dependencies with `yarn npm audit` reveals one vulnerable dependency: `@babel/runtime`, which is affected by CVE-2025-27789.

Mitigation

We recommend updating this dependency and following a process that emphasizes secure dependency usage to avoid introducing vulnerabilities into the Rabby Mobile Wallet Application and to mitigate supply-chain attacks. This process includes:

- Manually reviewing and assessing currently used dependencies;
- Upgrading dependencies with known vulnerabilities to patched versions with fixes;
- Replacing unmaintained dependencies with secure and battle-tested alternatives, if possible;
- Pinning dependencies to specific versions, including pinning build-level dependencies in the `package.json` file to a specific version;
- Only upgrading dependencies upon careful internal review for potential backward compatibility issues and vulnerabilities; and

- Including Automated Dependency auditing reports in the project's CI/CD workflow.

Status

The Rabby Wallet team stated that they will not update dependencies unless necessary to avoid introducing additional security risks. They also noted that CVE-2025-27789 has no impact on their implementation, as no untrusted string will be used.

Verification

Unresolved.

Suggestion 3: Remove Logging for Production

Location

`apps/mobile/src/core/utils/cloudBackup.ts`

Synopsis

The current implementation emits logs for debugging purposes in the `cloudBackup.ts` file. Although this can be beneficial during the development process, it also poses a vector for leaking sensitive data in a production release.

Mitigation

We recommend removing or conditionally disabling debug logging in production builds to avoid exposing potentially sensitive information.

Status

The Rabby Wallet team has removed debug logging from production builds.

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, zero-knowledge protocols, and consensus 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.