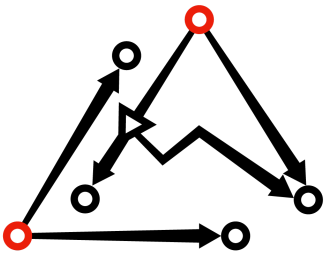

Least Authority

PRIVACY MATTERS

Rabby Wallet

Security Audit Report

DeBank

Final Audit Report: 18 October 2024

Table of Contents

- Overview
  - Background
  - Project Dates
  - Review Team
- Coverage
  - Target Code and Revision
  - Supporting Documentation
  - Areas of Concern
- Findings
  - General Comments
    - System Design
    - Code Quality
    - Documentation and Code Comments
    - Scope
    - Dependencies
  - Specific Issues & Suggestions
    - Issue A: Missing Password Strength Check
    - Issue B: Insecure Key Derivation Function
    - Issue C: Weak Encryption Method Used
    - Issue D: Weak PBKDF2 Parameters Used
    - Issue E: Supported Old Android Versions May Compromise Security of the Wallet
  - Suggestions
    - Suggestion 1: Improve Code Comments
    - Suggestion 2: Remove Duplicate Code in map.ts
    - Suggestion 3: Improve Documentation
    - Suggestion 4: Make password Property Private
    - Suggestion 5: Improve Test Coverage
    - Suggestion 6: Update Vulnerable Dependencies
- About Least Authority
- Our Methodology

Overview

Background

DeBank has requested that Least Authority perform a security audit of Rabby Wallet, a web3 wallet offering users a multi-chain DeFi experience.

Project Dates

- **September 2, 2024 - September 25, 2024:** Initial Code Review *(Completed)*
- **October 4, 2024:** Delivery of Initial Audit Report *(Completed)*
- **October 18, 2024:** *(Completed)*
- **October 18, 2024:** Delivery of Final Audit Report *(Completed)*

Review Team

- Poulami Das, Security / Cryptography Researcher and Engineer
- Nikos Iliakis, Security Researcher and Engineer
- Michael Rogers, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Rabby Wallet 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:

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

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

- `a8dea5d8c530cb1acf9104a7854089256c36d85a`

For the verification, we examined the Git revision:

- `2eeef56d65acc1a91415a55b80131d9a9ba35a5c`

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

- Rabby Mobile:
  https://github.com/LeastAuthority/rabby-wallet

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:

- Rabby Wallet Website:
  https://rabby.io
- Rabby Mobile Architecture:
  https://debankglobal.larksuite.com/docx/VjENdgDCaolxjZx6En7uBzrns0l
- 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)*

In addition, this audit report references the following documents:

- Password Storage - OWASP Cheat Sheet Series:
  https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#password-hashing-algorithms
- MMKV source code:
  https://github.com/Tencent/MMKV/blob/cc8565b997a65ae66eb1d8e5f8feaeaf11cba449/Core/aes/AESCrypt.cpp#L80
- EIP-1193: Ethereum Provider JavaScript API:
  https://eips.ethereum.org/EIPS/eip-1193
- EIP-2255: Wallet Permissions System:
  https://eips.ethereum.org/EIPS/eip-2255
- Universal XSS in Android WebView (CVE-2020-6506):
  https://alesandroortiz.com/articles/uxss-android-webview-cve-2020-6506

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, an extension wallet that aims to support multiple chains and hardware wallets (Ledger, Keystone, Onekey, etc.), providing better convenience to dApp users who require the support of various wallets and need to transact across several chains. The

Rabby Wallet application can manage and host dApps from numerous providers and enables users to interact with the hosted dApps through a unified interface.

During our review, we identified several issues and suggestions that would improve the quality of the code and contribute to the overall security of the implementation, as detailed below.

System Design

Our team found that security has been taken into consideration in the design of the Rabby Wallet as demonstrated by the storage of sensitive data in encrypted form, the disabling of the logging of private information, and the general practice of following a modular security architecture. However, our team found that the choice of cryptographic methods/parameters and the security consideration regarding the usage of passwords could be improved to mitigate the risks of losing funds.

Key Derivation and Encryption

When the user is prompted to update the default password, there is currently no mechanism to check the strength of the password entered by the user (Issue A). As a result, the user could select a very weak password or even an empty string. Since the password is later used in key derivation, it is crucial to choose a strong password to prevent dictionary attacks, which could then lead to the loss of wallet funds. We recommend adding constraints to check the strength of the password to prevent the usage of insecure passwords.

The keychain containing the user's crypto accounts is encrypted before being stored on the mobile device. A key derivation function is used to derive an encryption key from the user's password. This key is then used in the `AES-CBC` mode to encrypt the keychain. The encrypted keychain is subsequently stored in an MMKV key-value store, which adds a second layer of encryption using the `AES-CFB` mode.

Our team found that the key derivation function of PBKDF2 is unsuitable for the derivation of password-dependent encryption keys (Issue C and Issue D). Since these derived keys are solely responsible for the secure storage of sensitive data, it is crucial to use a strong key derivation function. We recommend either strengthening the PBKDF2 parameters, as per OWASP's recommendation, or using `Argon2id`, which is a memory hard function that is more resistant against CPU-dependent brute-force attacks as well as side-channel attacks.

Regarding encryption, we found that the `AES-CBC` mode is not a sufficiently secure choice since it does not guarantee the authentication of ciphertexts, which could lead to malleability attacks on the encrypted data (Issue B). We recommend using the authenticated encryption variant of `AES-GCM` instead.

The second layer of encryption added by MMKV uses the AES-CFB mode, which shares the same issue as `AES-CBC` in that it does not guarantee the authentication of ciphertexts. Furthermore, the key used for this second layer of encryption is an easily guessed constant that can be found in the application's source code. Therefore, the additional layer of encryption added by MMVK does not provide any meaningful security.

DApp Hosting and WebView

The Rabby Wallet application acts as a host for dApps created by DeBank and a number of other providers. The application uses a WebView component to host each dApp. Inside the WebView, the dApp is allowed to execute JavaScript, load resources from the network, and display a user interface. Isolation of the dApp's execution context inside the WebView from the host application's execution context is therefore critical for the wallet's security.

We found that Rabby Wallet follows industry standards and best practices for ensuring isolation of dApps from the hosting application. Rabby Wallet injects JavaScript into the pages that are loaded in the

WebView in order to allow communication between the dApp and the host via a standard EIP-1193 API. Rabby Wallet restricts the messages that dApps are allowed to send until they have requested permissions via the mechanism described in EIP-2255. When a dApp requests permission to access the user's crypto accounts, Rabby Wallet shows a confirmation prompt that includes information about the dApp's origin, popularity, and recommendations from the community. This should help discourage users from granting permissions to untrustworthy dApps.

On Android, Rabby Wallet uses the system WebView rather than providing its own WebView. As a result, some WebView functionality varies between devices, depending on which version of the system WebView is installed. On older Android devices, we found that JavaScript running inside the WebView was able to trigger a number of interactions with other applications running on the device, such as opening the email application to compose a message. However, in all cases, the user was prompted to confirm the action before it was allowed to happen, so we do not consider this variability in WebView behavior to be a security risk. We did not find any risks of information exposure or privilege escalation via 'deep linking' to resources on the local device (e.g., URIs using the `file`, `content`, `intent`, or `android-app` schemes).

We note that Rabby Wallet patches the React Native WebView library to improve the security of dApp isolation, specifically by prompting the user to grant camera, microphone, and location permissions to dApps, independently of whether these permissions have been granted to the Rabby Wallet application itself.

Android Support

Our team also noted that the wallet can be installed on a version of Android that is no longer supported, which introduces attack surfaces. One such vulnerability that is relevant to Rabby Wallet is CVE-2020-6506, "Universal XSS in Android WebView." Although this vulnerability was published more than four years ago, Rabby Wallet supports a minimum Android version of `7.0` (API level 24). This version of Android was released in `2016`, and most devices running Android `7.0` stopped receiving security updates before `2020`. It is therefore possible, although unlikely, that some users of Rabby Wallet may still have WebViews that contain this vulnerability. The vulnerability would allow content loaded in an iframe (such as an advertisement) within a dApp to take control of the dApp, potentially leading to loss of funds if the user had allowed their accounts to be linked to the dApp. We recommend that the Rabby Wallet team make the minimum supported API level/Android version the version that receives reasonable security updates from Google (Issue E).

Screenshots

The Android and iOS versions of the application prevent screenshots of sensitive data (such as the user's seed phrase) from being taken. The content of the application window is blurred when the application is moved into the background, preventing sensitive data from being captured in the screenshots used by the system's application switching functionality.

Code Quality

We performed a manual review of the repositories in scope and found the code implementation to be generally well-organized and in adherence to best practices. However, we found incorrect code comments (Suggestion 1) and duplicate code (Suggestion 2), which we recommend be updated and removed respectively.

Tests

Our team found the test coverage of the repositories in scope to be insufficient. A comprehensive test suite should include unit tests and integration tests that cover both success and failure scenarios. This helps in detecting errors and bugs, as well as guarding against potential edge cases that could result in

security-critical vulnerabilities or exploits. We recommend enhancing the unit and integration tests (Suggestion 5).

Documentation and Code Comments

Although the project documentation provided for this review offers a general overview of the architecture of the Rabby Wallet, we recommend improving the specification to accurately describe the system and its components (Suggestion 3). Additionally, while the codebase contains some comments, they are sparse and incomplete. This reduces the readability of the code and, as a result, makes reasoning about the security of the system more difficult. We therefore also recommend improving code comments (Suggestion 1).

Scope

The scope for this review was sufficient and included all security-critical components. However, we note that the currently published version of the Rabby Wallet Android application includes a security-critical feature (cloud backup) that was not included in the audited version of the application.

Cloud Backup Feature

The version of the Rabby Wallet Android application currently available from Google Play includes a cloud backup feature that allows the user to back up their crypto accounts to Google Drive. The backup appears to be encrypted with a key derived from the user's password. The cloud backup feature was not included in the version of the source code that our team was asked to audit, so we consider this feature to be outside the scope of this audit. However, we note that the feature is likely to be security-critical. If the key used for encrypting the backup is derived in a similar way to the key used for encrypting the user's crypto accounts when they are stored on the mobile device, then Issue C and Issue D would also apply to this feature.

Dependencies

We analyzed all the dependencies implemented in the codebase and found them to be generally secure, with the exception of one dependency, `web3-utils`, which is vulnerable to the Prototype Pollution vulnerability. We recommend upgrading to a patched version (`>=4.2.1`).

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: Missing Password Strength Check	Resolved
Issue B: Insecure Key Derivation Function	Unresolved
Issue C: Weak Encryption Method Used	Unresolved
Issue D: Weak PBKDF2 Parameters Used	Unresolved
Issue E: Supported Old Android Versions May Compromise Security of the Wallet	Resolved

Suggestion 1: Improve Code Comments	Unresolved
Suggestion 2: Remove Duplicate Code in map.ts	Partially Resolved
Suggestion 3: Improve Documentation	Unresolved
Suggestion 4: Make password Property Private	Resolved
Suggestion 5: Improve Test Coverage	Unresolved
Suggestion 6: Update Vulnerable Dependencies	Unresolved

Issue A: Missing Password Strength Check

Location

`service-keyring/src/keyringService.ts#L140-L150`

Synopsis

The `updatePassword` function in the `KeyringService` class takes an old password and a new password as inputs and checks for the validity of the old password. If valid, it then updates the keyring with the new password by encrypting existing keyring material with the new password. Currently, the function is missing a check on the new password. Due to this, the input password could potentially be empty or a string with very low entropy.

Impact

Since the password is later used for key derivation via PBKDF2 (see Issue B), a malicious actor could exploit this issue to brute-force the password, which might result in the encryption key being leaked. This can lead to the attacker stealing funds using the private key material retrieved by decrypting with the correct key.

Preconditions

The attacker would need to have access to the user's device.

Feasibility

In order for this exploit to occur, the attacker would need powerful hardware, such as CPUs, GPUs, and FPGAs that would allow them to perform brute-force attacks on the encryption key, in the cases where weak password were selected – in particular, for the current PBKDF2 parameters of `5000` iterations (Issue D). If the aforementioned preconditions are met, the attack is feasible.

Mitigation

We recommend that the Rabby Wallet team inform their users about password best practices and the risks associated with choosing a weak password.

Remediation

We recommend adding password strength estimation to the form validation and preventing users from choosing weak passwords.

Status

The Rabby wallet team stated that the caller to the function `keyringService.Updatepassword` ensures that the newPassword is *not* empty in `apps/mobile/src/core/apis/lock.ts`, and that the password length check is checked in the UI layer. Additionally, the team noted that the other two callers of `keyringService.Updatepassword` are currently not in use, and that a code comment has been added for further clarification. Our team agrees with the development team's response and thus considers this issue resolved.

Verification

Resolved.

Issue B: Insecure Key Derivation Function

Location

`core/services/encryptor.ts#L25-L36`

Synopsis

The AES encryption key is derived from a password using the PBKDF2 key derivation function with weak parameters (Issue D), which is not an ideal function for key derivation.

Impact

If the attacker successfully derives the correct AES encryption key, this would lead to the leakage of stored sensitive information, including private key materials, thus allowing the attacker to steal funds from the wallet.

Preconditions

The attacker would need to have access to the encrypted data stored in the wallet.

Feasibility

Targeting an attack on the key derivation function PBKDF2 can be highly optimized with the use of parallel computing powers of CPUs, GPUs, and FPGAs.

Technical Details

PBKDF2 with weak parameters is not an ideal key derivation function, as explained in the feasibility section above. Alternatively, to derive password-based encryption keys, memory hard functions should be used that are designed with inherent resistance to parallelization.

Remediation

We recommend making use of key derivation functions based on memory hard functions, such as `Argon2`.

Status

The Rabby Wallet team acknowledged that strengthening the key derivation options or PBKDF2 parameters could result in higher security. However, they stated that the current settings represent a balance between security and the performance of the encryption process, and that they have aligned with the industry leader MetaMask in this regard. The team added that they may consider enhancing security in the future provided that it does not compromise performance in the process.

Verification

Unresolved.

Issue C: Weak Encryption Method Used

Location

core/services/encryptor.ts#L29-L36

apps/mobile/src/core/storage/mmkv.ts#L69

Synopsis

The AES-CBC mode of encryption that has been implemented does not provide any authentication guarantees on the ciphertext. Due to this, the ciphertext could be modified and might therefore decrypt to a different value that appears to be valid. The additional layer of encryption provided by MMKV uses the AES-CFB mode, which also fails to provide authentication of the ciphertext. Furthermore, a hard coded and easily guessed key is used for the MMKV encryption layer.

Impact

The attacker could modify the ciphertext, such that it would decrypt to the incorrect password/seed phrase/address/account. There are several scenarios that might be possible. Firstly, the user might be locked out if the password verification or seed phrase validation fails. Consequently, all of the user's funds would be locked as well. Secondly, the attacker could also trick the user to submit a transaction to an address of the attacker's choice.

Preconditions

The attacker would need to have access to the encrypted data containing sensitive information, such as addresses or accounts and have the ability to change it.

Remediation

We recommend using an authenticated encryption variant of AES, such as AES-GCM.

Status

The Rabby Wallet team acknowledged that strengthening the key derivation options or PBKDF2 parameters could result in higher security. However, they stated that the current settings represent a balance between security and the performance of the encryption process, and that they have aligned with the industry leader MetaMask in this regard. The team added that they may consider enhancing security in the future provided that it does not compromise performance in the process.

Verification

Unresolved.

Issue D: Weak PBKDF2 Parameters Used

Location

core/services/encryptor.ts#L21-L27

Synopsis

The parameter configuration of PBKDF2 does not adhere to accepted standards. Currently, PBKDF2 is being used with only 5000 iterations of SHA256.

However, even with sufficient iterations, PBKDF2 can be efficiently parallelized and therefore does not provide a strong protection against brute-force attacks (see Issue B).

Impact

Due to the weak parameter selection, the key derivation function could be vulnerable, leading to the attacker deciphering the correct encryption key, which they can therefore use to decrypt the wallet's stored sensitive information.

Feasibility

The attacker would need to have access to the encrypted data, in addition to powerful hardware.

Mitigation

We recommend increasing the iterations.The current NIST recommendation of `600,000` iterations is a more robust solution if PBKDF2 is used.

Remediation

We recommend replacing PBKDF2 with a memory hard function (see Issue B).

Status

The Rabby Wallet team acknowledged that strengthening the key derivation options or PBKDF2 parameters could result in higher security. However, they stated that the current settings represent a balance between security and the performance of the encryption process, and that they have aligned with the industry leader MetaMask in this regard. The team added that they may consider enhancing security in the future provided that it does not compromise performance in the process.

Verification

Unresolved.

Issue E: Supported Old Android Versions May Compromise Security of the Wallet

Location

`mobile/android/build.gradle`

Synopsis

The wallet can be installed on an older version of Android (Android `7.0`) that has multiple, unfixed vulnerabilities. These devices do not receive reasonable security updates from Google.

Impact

Installing an application on a vulnerable, unpatched Android version can lead to serious security risks, including remote code execution, data leakage, and privilege escalation. These vulnerabilities may allow attackers to control the device, access sensitive information, or bypass security features, posing significant privacy and operational threats.

Preconditions

This issue can occur if the user's device is not updated.

Feasibility

The attacker would need to have access to the device.

Remediation

We recommend supporting an Android version greater than `10` (API 29).

Status

The Rabby Wallet team has removed the previous configuration that hard coded the minimum version to 24, and is currently using the version suggested by Android at the time of the build.

Verification

Resolved.

Suggestions

Suggestion 1: Improve Code Comments

Location

Examples (non-exhaustive):

`eth-keyring-ledger/src/LedgerKeyring.ts#L605`

`eth-keyring-onekey/src/eth-onekey-keyring.ts#L701`

Synopsis

Currently, all areas of the codebase significantly lack in-line code comments. This reduces the readability of the code and, as a result, makes reasoning about the security of the system more difficult. Comprehensive in-line documentation helps provide reviewers of the code with a better understanding and ability to reason about the system design.

Additionally, some code comments are incorrect and need updating. For example, in the aforementioned locations, the comment refers to "private methods." However, the class contains a mixture of private and public functions following this comment. Although some of the functions are denoted as `_functionname` – a standard process that is followed when a function is intended to be used privately – they are not explicitly declared as private. As for the public functions, it is unclear as to whether it is safe to use them from outside the scope of the class.

Mitigation

We recommend expanding and improving the code comments to facilitate reasoning about the security properties of the system. Regarding the incorrect comments, we recommend adding only the private functions after the comment "private methods," and explicitly declaring the relevant functions as private.

Status

The Rabby Wallet team acknowledged that this suggestion would be beneficial for the long-term development of their project. However, the team added that the recommended mitigation cannot be implemented in a short time frame, and that they will therefore take it into consideration for future releases.

Verification

Unresolved.

Suggestion 2: Remove Duplicate Code in map.ts

Location

`Approval/components/map.ts#L3-L12`

Synopsis

The above lines of code contain duplicate definitions for the keyring classes `mnemonic` and `privatekey`, which can lead to confusion and potential bugs.

Mitigation

We recommend removing the duplicate definitions.

Status

The Rabby Wallet team has partially removed the duplicate definitions. However, some duplicate keys remain in `WaitingSignComponent` in `map.ts` (lines 5 and 11; lines 6 and 10).

Verification

Partially Resolved.

Suggestion 3: Improve Documentation

Location

`https://rabby.io`

`https://debankglobal.larksuite.com/docx/VjENdgDCaolxjZx6En7uBzrns0l`

Synopsis

The general documentation provided by the Rabby Wallet team was insufficient. Although the Rabby mobile architecture contains some details on the general architecture, it lacks enough details on the security considerations (derivation of keys from passwords, authentication methods, etc.), the type of information that is stored in encrypted form, as well as the definitions of a keyring, account, address etc. It should also contain more details on the main functionalities (e.g., as in the case of an extension wallet).

Mitigation

We recommend that the Rabby Wallet team improve the project's general documentation by creating a high-level description of the system, each of the components, and the interactions between those components. This can include developer documentation and architectural diagrams.

Status

The Rabby Wallet team acknowledged that this suggestion would be beneficial for the long-term development of their project. However, the team added that the recommended mitigation cannot be implemented in a short time frame, and that they will therefore take it into consideration for future releases.

Verification

Unresolved.

Suggestion 4: Make password Property Private

Location

service-keyring/src/keyringService.ts#L84

Synopsis

The `password` property could be potentially accessed by other functions outside the scope of the `KeyringService` class, thus leading to the leakage of the password.

Mitigation

We recommend making the `password` property private to prevent the risk of a malicious actor gaining access to it from outside the `KeyringService` class.

Status

The Rabby Wallet team has implemented the mitigation as recommended.

Verification

Resolved.

Suggestion 5: 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 acknowledged that this suggestion would be beneficial for the long-term development of their project. However, the team added that the recommended mitigation cannot be implemented in a short time frame, and that they will therefore take it into consideration for future releases.

Verification

Unresolved.

Suggestion 6: Update Vulnerable Dependencies

Location

apps/mobile/package.json

Synopsis

Analyzing `package.json` files for vulnerable dependencies using Npm Audit shows that one transitive dependency, `web3-utils`, is vulnerable to the Prototype Pollution vulnerability.

Mitigation

This issue is indirect since it is introduced by web3 and does not pose a direct threat to the application. However, we nevertheless recommend upgrading to a patched version (`>=4.2.1`).

We additionally recommend following a process that emphasizes secure dependency usage to avoid introducing vulnerabilities to the Rabby Wallet application and to mitigate supply-chain attacks, which 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 acknowledged that this suggestion would be beneficial for the long-term development of their project. However, the team added that the recommended mitigation cannot be implemented in a short time frame, and that they will therefore take it into consideration for future releases.

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 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.