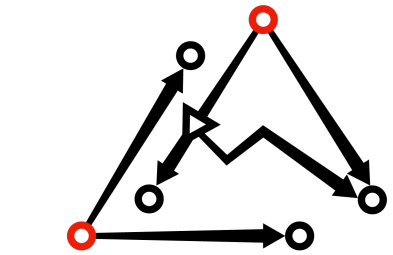

Least Authority

PRIVACY MATTERS

Rabby Wallet Extension

Security Audit Report

DeBank

Final Audit Report: 12 December 2024

Table of Contents

Overview

Background

DeBank has requested that Least Authority perform a security audit of the Rabby Wallet Extension. The Rabby Wallet is a multi-chain EVM wallet for Google Chrome, optimized for the DeFi ecosystem.

Project Dates

- **October 29, 2024 - November 19, 2024:** Initial Code Review *(Completed)*
- **November 21, 2024:** Delivery of Initial Audit Report *(Completed)*
- **12 December, 2024:** Verification Review *(Completed)*
- **12 December, 2024:** Delivery of Final Audit Report *(Completed)*

Review Team

- Will Sklenars, Security Researcher and Engineer
- Dominic Tarr, Security Researcher and Engineer

Coverage

Target Code and Revision

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

- https://github.com/RabbyHub/Rabby/commit/eb5da18727b38a3fd693af8b74f6f151f2fd361c
  - https://github.com/RabbyHub/Rabby/tree/develop/src/background/service/keyring
    - The keyring module contains encrypted storage of user privacy information and manages user private keys and mnemonics
  - https://github.com/RabbyHub/Rabby/blob/develop/src/background/webapi/storage.ts
    - The storage module is responsible for persisting information to the Chrome extension's storage
  - https://github.com/rabbyhub/eth-hd-keyring
    - The mnemonic keyring
  - https://github.com/RabbyHub/eth-simple-keyring
    - The private key keyring
- REST API is out of scope
- Dependencies are out of scope

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

- Rabby: `eb5da18727b38a3fd693af8b74f6f151f2fd361c`
- eth-hd-keyring: `26e721c4dadf2b5d83007ee346ccd0ce53d89618`
- eth-simple-keyring: `1c3428ceab7e5d1533133c9d0e92e1bdcd236fa8`

For the verification, we examined the following Git revisions:

- Rabby: `4650e438006c1e9d90416f0b7ea55eff38d1c2a5`

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

- Rabby: https://github.com/LeastAuthority/RabbyHub-Rabby-Wallet-Extension

- eth-hd-keyring: https://github.com/LeastAuthority/Rabbyhub-eth-hd-keyring
- eth-simple-keyring: https://github.com/LeastAuthority/RabbyHub-eth-simple-keyring

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 Security Audit Report by Least Authority on the Rabby Mobile Wallet (pdf) *(delivered via email on 18 October 2024)*

In addition, this audit report references the following documents:

- `semgrep`:
  https://semgrep.dev
- `synk`:
  https://snyk.io
- Compiling private to #property:
  https://stackoverflow.com/questions/75480223/compiling-private-to-property

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.

Additionally, the Rabby Wallet team requested that the following be addressed in the Rabby Wallet Extension Audit:

- Ensuring that the storage management of user private keys and mnemonics are always encrypted and stored in the Rabby Wallet, and will not be uploaded by any network request; and
- Ensuring that no private data will upload via REST API.

Findings

General Comments

DeBank has identified that users of DeFi are often interacting with multiple, different blockchains. DeBank built the Rabby Wallet from a fork of MetaMask, and extended it to improve user experience when interacting with the multi-chain DeFi ecosystem. Our team performed a security audit of the Rabby Wallet Extension, which maintains mappings from DeFi websites to chains so that when a user visits a particular website, the appropriate blockchain is selected within the wallet. A user can change this setting, selecting a different blockchain for a website, and the setting will only affect the behavior of the Rabby Wallet for the specific website, without affecting how the wallet operates on other websites.

Part of Rabby Wallet's security offering is that it seeks to summarize and clearly communicate to the user the downstream effects of a particular transaction, before a user signs the transaction. This is designed to help clarify the effects of transactions generated by DeFi websites that may contain complex contract calls.

The Rabby Wallet also screens each transaction for potential security threats and displays this information to the user before signing takes place. This is achieved through scanning each transaction with Rabby Wallet's security engine. Transactions generated by a DeFi website can be scanned, reporting potential mistakes or threats to the user. For example, the user will be warned if a transaction will send tokens to an inactive account, or interact with a contract with known vulnerabilities.

System Design

Our team performed a close review of the Rabby Wallet Extension system and identified areas of improvement that would increase the overall security of the system. We analyzed the codebase for any vulnerabilities that could lead to denial of service and an interrupted user experience. We identified one scenario that could cause infinite recursion, which would cause the application to crash (Issue A).

Key Storage and Encryption

Rabby Wallet can store private keys locally and also interfaces with a range of hardware wallets. Since the Rabby Wallet Extension is a cryptocurrency wallet, users have private keys and will lose funds if they lose access to those keys or if an attacker gains access to those keys. Therefore, in performing the audit, our team paid particular attention to the storage of private keys. The private keys are stored alongside other application state, which is stored as a JSON object in the browser extension's `localStorage`. This is mediated via the `createPesistStore` API, but there are also some parts of the code that access `localStorage` directly. The data stored in `localStorage` is specific to the browser extension and cannot be accessed by the web page that is currently open, or by other extensions. By using a JavaScript Proxy object, changes to the store are detected, and persistence is triggered. The private keys are stored in this object but encrypted under the `vault` key. The private keys are encrypted with `AES-GCM`, using a key derived via PBKDF2 and the user's password.

The Least Authority team previously audited the mobile version of the Rabby Wallet, and delivered a Final Audit Report on October 18, 2024. The Rabby Mobile Wallet and Rabby Wallet Browser Extension share a lot of code in common, including the parts of the system that deal with encryption algorithms. However, the common code is maintained separately, duplicated across the two wallets' respective repositories. In the audit for the Rabby Mobile Wallet, we analyzed the cryptography to check that the choice of algorithms was appropriate, and that the encryption algorithms were being used correctly. In that audit, we recommended replacing the PBKDF2 key derivation function with a stronger function, and enforcing the password strength. We also recommended changing the encryption algorithm from `AES-CBC` to `AES-GCM`. In response, the Rabby Wallet team stated that due to the user requirements particular to a

wallet, it was not feasible to follow the recommendations for key derivation function and password strength. The motivation for choosing a relatively strong key derivation function is that it makes it more expensive for an attacker who has access to the ciphertext to perform a dictionary attack. However, users generally run the Rabby Wallet Extension on relatively weak devices, such as a web browser or a mobile device, while an attacker is able to leverage relatively powerful hardware, such as a cryptocurrency mining rig. Therefore, there is an imbalance in available computing power between the user and the attacker, and hardening the key derivation function would severely impact the user due to their relatively weak computing power, while only moderately impacting an attacker with their relatively strong hardware. Due to this imbalance in available computing power, our team considers the Rabby Wallet team's choice of key derivation function to be appropriate for their use case. A strong unique password that is not found in a dictionary is therefore a more effective approach to defending against dictionary attacks, compared to a hardened key derivation function. However, the strength of a password is determined by whichever password the user chooses. The Rabby Wallet Extension does enforce the minimal password complexity requirement, which states that the password should be at least eight characters long. However, this allows users to choose very weak passwords, such as "password." The Rabby Wallet team acknowledged this issue but stated that they consider the threat of a user potentially forgetting their own password to be greater than the threat of a dictionary attack. Due to this, the Rabby Wallet team has chosen not to enforce stronger password requirements. While our team strongly encourages making applications as secure as possible, we acknowledge that due to the affordances of a cryptocurrency wallet, this compromise is necessary and that other approaches must be taken to maximize security.

Supply Chain Attacks

Since the encrypted vault lives on the user's device and is not uploaded, and because the strength of the protection cannot be improved upon without considerable negative impacts on the user, the security goal must focus on preventing an attacker from gaining access to the ciphertext. It is also important to protect the derived vault encryption key from an attacker. The vault key is stored in session storage because the web extension service worker could be closed at any time by the browser, such as when the user changes tabs. When the application is re-initialized, the vault key in session storage is used so that the user is not required to enter the password again. While reviewing the code responsible for persisting this state, our team identified a scenario where the browser storage cache could cause an exception (Issue B) and also detected a situation where the browser storage state can become stale (Issue C).

Additionally, our team found that the Rabby Wallet Extension codebase is not inherently resistant to supply chain attacks. As the Rabby Wallet Extension is JavaScript based, the Rabby Wallet team follows some practices to take security into consideration and mitigate against these attacks, such as refraining from updating dependencies (because a supply chain attack is typically performed when new code is introduced), pinning all the dependencies, and installing from the resolved dependencies detailed in the `yarn.lock` file, which is checked into the repository. When deploying, the Rabby Wallet team builds the project on multiple machines and ensures that every build resolves to the same hash digest before deploying the new build. This guards against an attack that may seek to compromise a build machine to inject malicious code. The team also uses the auditing services `semgrep` and `synk`. Moreover, the Rabby Wallet Extension currently has compile time mitigations that make it more difficult for a supply chain attack to be performed; however, there are no runtime mitigations to account for a case where an attacker does manage to circumvent that line of defense (Suggestion 4).

Code Quality

We performed a manual review of the repositories in scope and found the codebases to be generally well-organized. The code is written in TypeScript and follows object-oriented patterns. However, TypeScript strict mode has not been enabled, and there are several implicit `any` types. Increasing the use of type definitions would make the code easier to maintain and help prevent runtime errors. While our team noted that enabling strict mode would help enforce the use of types during development, we

acknowledge that this would require significant changes and development efforts.

Additionally, during our review, we identified a debounce module that is used incorrectly (Suggestion 1), and a function name that does not accurately describe the purpose of the function (Suggestion 3).

Tests

The repositories in scope include some tests; however, our team found that the tests have not been maintained, and some of them do not currently pass. We recommend making all tests pass (Suggestion 5).

Documentation and Code Comments

There was no documentation provided for the Rabby Wallet Extension system. However, the codebase contained descriptive code comments, and the lack of documentation was not an issue.

Scope

The scope of this review was sufficient and included all security-critical components.

Dependencies

Our team did not identify any issues with the dependencies currently used. However, due to the risks associated with supply chain attacks on cryptocurrency wallets, we recommend implementing further mitigations against these types of attacks to improve the security of the dependencies (Suggestion 4).

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: Infinite Recursion Can Result in Denial of Service	Resolved
Issue B: Setting the Cache Before It Has Been Initialized Will Cause an Exception	Unresolved
Issue C: persistStore Module Can Become Out of Sync With Browser Local Storage	Unresolved
Suggestion 1: Remove or Correct Use of debounce Module	Resolved
Suggestion 2: Ensure Ledger JavaScript Bindings and TransportWebHID Are Initialized Once Only	Resolved
Suggestion 3: Rename convertToBigint Function	Resolved
Suggestion 4: Add Runtime Mitigations for Supply Chain Attack	Unresolved
Suggestion 5: Make All Tests Succeed	Unresolved

Issue A: Infinite Recursion Can Result in Denial of Service

Location

keyring/eth-lattice-keyring/eth-lattice-keyring.ts#L146

Synopsis

The `catch` block in the `getAddress` function contains a recursive call to itself. If a persistent exception occurs, `getAddress` will be called repeatedly until the call stack limit is reached.

Impact

This scenario would result in a "Maximum call stack size exceeded" error (stack overflow), causing the extension to become unresponsive, or crash.

Preconditions

Some of the scenarios that can result in an exception are listed in the code comments of the `catch` block:

```
// This will get hit for a few reasons. Here are two possibilities:
// 1. The user has a SafeCard inserted, but not unlocked
// 2. The user fetched a page for a different wallet, then switched interface
on the device
// In either event we should try to resync the wallet and if that fails throw
an error
```

The `SafeCard` scenario seems likely to result in the call stack limit being reached, as a user action (unlocking the safecard) would be required to break the recursion.

Feasibility

The existence of the `retry` logic implies that errors are expected to occur from time to time, and the code comments also support this. Thus, it is expected that some users would experience denial of service caused by the stack overflow.

Technical Details

The comments state the following:

```
we should try to resync the wallet and if that fails throw an error
```

However, throwing an error after the second failed `retry` would require keeping some state between retries, such as a `counter`. This has not been implemented.

Remediation

We recommend limiting retries using a `counter` so that subsequent retries do not overwhelm the call stack. If successful execution requires waiting for user action, we recommend waiting a period of time between retries so that the counter limit is not reached too quickly. We therefore recommend only throwing an error once a reasonable amount of retries has occurred without success.

Status

The Rabby Wallet team has added a `retry` counter, which, after `20` unsuccessful retries, will cause an exception. As a result, the issue of infinite recursion has been remediated.

Verification

Resolved.

Issue B: Setting the Cache Before It Has Been Initialized Will Cause an Exception

Location

background/webapi/storage.ts#L21

Synopsis

The storage module is responsible for persisting state to browser storage and also maintains an in-memory cache. The `set` function requires the cache to be initialized; otherwise, an exception will occur. The cache is initialized by the `get` function. Therefore, if `storage.set` is called before `storage.get`, the uninitialized cache will be referenced, and an exception will occur.

Impact

If the exception occurs, the setting of state will fail. Depending on how exceptions are handled by the modules that depend on the storage module, these set calls could fail silently, resulting in unexpected behavior.

Preconditions

`storage.set` would have to be called before `storage.get`.

Feasibility

This error can occur if the preconditions are met, as it is not a given that `get` will be called before `set`.

Remediation

We recommend adding a check in each `set` and `get` function to verify if the cache has been initialized. If the cache has not been initialized, an `initialize` function should be called to set up and populate the cache.

Status

The Rabby Wallet team has acknowledged this issue and stated that in order to avoid unexpected exceptions caused by changes, they need to evaluate and perform tests. Hence, the team noted that the recommended remediation will not be implemented at this time due to time limitations, but will be taken into consideration for future releases.

Verification

Unresolved.

Issue C: persistStore Module Can Become Out of Sync With Browser Local Storage

Location

background/utils/persistStore.ts

Synopsis

The `persistStore` module acts as a wrapper around the `storage` module. The `storage` module provides access to browser local storage and maintains an in-memory cache. The `persistStore`

module uses JavaScript Proxy to add hooks to the storage update, which are used to trigger side effects such as rendering `html`. However, there are some instances where the state reported by the `storage` module (browser storage) can become stale.

Impact

If one part of the application is referencing a particular variable through `persistantStore`, while another part of the application is referencing the variable through the `storage` module, the latter part may observe a stale value. In such a scenario, there is no single point of truth, which can lead to unexpected behavior, such as incorrect values being displayed to the user.

Feasibility

Although our team could not identify a specific situation where the stale browser state could affect the end user, we did find a scenario where the stale state can occur.

Technical Details

The following describes a scenario where the state in the `storage` module can become stale:
In `openapi.ts`, in the `baseStore` constructor, `persistentStore` is initialized with some default values, and assigned to `baseStore.store`. These initial default values are not persisted to browser storage or cache because `createPersistStore` does not call `storage.set` on initialization. It only sets up the JavaScript Proxy.
In `openapi.ts:36`, the `persistentStore` value `testnetHost` is updated to a new value. This activates the Proxy setter hook, which updates the browser storage and cache in the `storage` module.

Next, if the application is reloaded, the `baseStore` class will initialize `persistentStore` again, with the defaults. Afterwards, `baseStore.store` reports the default values; however, the browser storage has the previously updated values. If another part of the application were to access the `testnetHost` property through the `store` module, it would observe a different value than the one `baseStore` observes.

Remediation

To eliminate the possibility of a stale browser state, we recommend modifying `createPersistStore`, such that the `storage` module is always updated when `createPersistStore` is called. This will result in the browser storage and cache being updated with the new initialization values, and remove the risk of the `storage` module reporting stale values.

Status

The Rabby Wallet team has acknowledged this issue and stated that in order to avoid unexpected exceptions caused by changes, they need to evaluate and perform tests. Hence, the team noted that the recommended remediation will not be implemented at this time due to time limitations, but will be taken into consideration for future releases.

Verification

Unresolved.

Suggestions

Suggestion 1: Remove or Correct Use of debounce Module

Location

background/util/persistStore.ts:8

Synopsis

The debounce module is called incorrectly, such that it has no effect and would actually throw an error if the latest version of debounce was used.

Technical Details

The debounce module typically takes a function and returns a function that will avoid calling the first function too often by using a timer. However, in this case, the debounce module is called with the returned promise value, and the debounced function is then discarded. As a result, no debouncing is actually applied. Also note that the latest version of debounce would throw an error if used in this manner.

Mitigation

We recommend either removing the debounce module or using it correctly.

Status

This issue was reported while the audit was still in progress. The Rabby Wallet team has applied the suggestion and is now using the debounce module correctly.

Verification

Resolved.

Suggestion 2: Ensure Ledger JavaScript Bindings and TransportWebHID Are Initialized Once Only

Location

src/background/service/keyring/eth-ledger-keyring.ts#L136-L137

Synopsis

The `makeApp` function is responsible for initializing the Ledger JavaScript bindings as well as initializing the Ledger Hardware Wallet *WebHID* implementation, TransportWebHID.

The `_reconnect` function calls the `makeApp` function every `100` milliseconds until `this.app` has been declared, as determined by the `while (!this.app)` loop in `_reconnect`.

In the `makeApp` function, there is another `this.app` check, which exits the function if `this.app` has been declared, implying that the Ledger bindings and TransportWebHID are not intended to be re-initialized once they have been initialized.

Initializing TransportWebHID is an asynchronous operation. Since the `this.app` check occurs before the asynchronous function call, if the asynchronous call takes longer than `100`ms, the Ledger JavaScript binding and TransportWebHID will be initialized twice, with the second initialization resulting in `this.app`

and `this.transport` being overwritten. This re-initialization seems counter to the intention of the code and could potentially have downstream effects.

Mitigation

We recommend storing the result of the asynchronous call in a temporary variable and performing a further `this.app` check, such that `this.app` and `this.transport` are only set once. To illustrate, consider the following:

```
const transport = await TransportWebHID.create();

if (!this.app) {

      this.transport = transport;

      this.app = new LedgerEth(this.transport);

}
```

Status

The Rabby Wallet team has removed the `_reconnect` function, which made the repeated calls to `makeApp`. As a result, the scenario where the Ledger JavaScript bindings and `TransportWebHID` could be re-initialized has been eliminated.

Verification

Resolved.

Suggestion 3: Rename convertToBigint Function

Location

keyring/eth-imkey-keyring/eth-imkey-keyring.ts#L14

Synopsis

The name of the `convertToBigInt` function suggests that the function should be responsible for converting the provided argument into the `BigInt` type. However, the function actually accepts a `BigInt` or a `number` type, and returns a `hexidecimal`.

Mitigation

We recommend renaming the function (to, for example, `convertNumberOrBigintToHex`) to explicitly and specifically describe its intended functionality.

Status

The Rabby Wallet team has renamed the `convertToBigint` function. The new name is `convertToHex`, which correctly describes the function's behavior.

Verification

Resolved.

Suggestion 4: Add Runtime Mitigations for Supply Chain Attack

Location

All repositories.

Synopsis

The Rabby Wallet team currently follows practices intended to mitigate a supply chain attack at build/compile time. However, additional practices should also be adopted to limit what a malicious module could do in the event that it does get included in the build.

Mitigation

Dependency code must be restricted from accessing stored keys, whether encrypted or not (that is, `browser.storage.local` and `browser.storage.session`). It is recommended that dependencies also be restricted from accessing network APIs through which a malicious dependency may exfiltrate secrets (such as `XmlHttpRequest`, `WebSocket`, and any element type that could trigger an `http` request). Additionally, any Rabby classes that hold private information (such as `KeyringService`) should use private fields whenever important fields are being stored, such as `password`. Note that in our previous review of the Rabby Mobile Wallet, our team had also recommended making the `password` field private. Although that suggestion has since been resolved, it was implemented in the Rabby Mobile Wallet codebase, but was not implemented in the Rabby Wallet Browser Extension codebase.

Moreover, TypeScript private fields are not by default compiled to true JavaScript private fields with the # prefix. For fields that should be private, a # prefix should be explicitly used in the `fieldname`.

We also further recommend freezing classes using `Object.seal` to prevent malicious code from manipulating other classes.The most straightforward way of accomplishing all this would be to integrate LavaMoat, which would likely require some significant changes, such as switching to `browserify` from `webpack`. LavaMoat has successfully been running in production in MetaMask for several years.

Status

The Rabby Wallet team stated that the suggested mitigation has been added to their backlog, as it is part of an ongoing and incremental process. Hence, this suggestion remains unresolved at the time of verification.

Verification

Unresolved.

Suggestion 5: Make All Tests Succeed

Location

`package.json#L20`

Synopsis

While there are some tests in the Rabby Wallet Extension repository, some of the tests are either failing or have not been maintained. Runnable tests facilitate refactoring and help identify implementation errors, which could lead to security vulnerabilities.

Mitigation

We recommend making all tests succeed.

Status

The Rabby Wallet team stated that the suggested mitigation has been added to their backlog, as it is part of an ongoing and incremental process. Hence, this suggestion remains unresolved at the time of 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 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.