



Least Authority
PRIVACY MATTERS

Rabby Mobile Application
Security Audit Report

Rabby Wallet

Final Audit Report: 9 October 2026

Table of Contents

[Overview](#)

[Background](#)

[Project Dates](#)

[Review Team](#)

[Coverage](#)

[Target Code and Revision](#)

[Supporting Documentation](#)

[Areas of Concern](#)

[Findings](#)

[General Comments](#)

[Code Quality](#)

[Documentation and Code Comments](#)

[Scope](#)

[Specific Issues & Suggestions](#)

[Issue A: Password Derivation Provides Limited Resistance to Offline Guessing](#)

[Issue B: Rejected Screenshot Events Skip Native Capture-File Cleanup](#)

[Issue C: Batch Security Review Covers Only the Final Transaction](#)

[Issue D: Bridge Deposit Transactions Are Not Checked Against Local Intent](#)

[Issue E: Local Swap Verification Does Not Check the Output Recipient](#)

[Issue F: Mnemonic Backup Views Do Not Enable Screenshot Protection](#)

[Issue G: Production Clipboard Policy Does Not Expire Copied Wallet Secrets](#)

[Issue H: Missing Frame Authentication Can Misattribute Android dApp Requests](#)

[Issue I: Failed Password Attempts Are Weakly Limited](#)

[Issue J: Clearing Keyring Storage Before Replacement Can Remove the Recoverable Wallet State](#)

[Issue K: Contract-Deployment Action Omits the Attached Native-Token Amount](#)

[Suggestions](#)

[Suggestion 1: Preserve the Safe Operation Across Approval and Signing](#)

[Suggestion 2: Improve Test Coverage](#)

[Suggestion 3: Remove Sensitive Information from Navigation State](#)

[Suggestion 4: Exclude Keychain Debugging Code from Production Builds](#)

[Suggestion 5: Replace or Upgrade Vulnerable Dependencies](#)

[About Least Authority](#)

[Our Methodology](#)

Overview

Background

Rabby Wallet has requested that Least Authority perform a security audit of its Mobile Application to assess developments since the previous review, for which the final audit report was delivered on September 2, 2025.

Project Dates

- **August 31, 2026 - September 9, 2026:** Initial Code Review (*Completed*)
- **September 11, 2026:** Delivery of Initial Audit Report (*Completed*)
- **October 8, 2026:** Verification Review (*Completed*)
- **October 9, 2026:** Delivery of Final Audit Report (*Completed*)

Review Team

- Nikos Iliakis, 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 was considered in scope for the review:

- Rabby Mobile Application:
<https://github.com/RabbyHub/rabby-mobile/compare/932ec...37c09>

Annual Audits will focus on the changes to the Rabby Wallet Mobile Application repository since the previous audit. Specifically, the scope of this audit will be based on the differences (diffs) identified in the link provided above, with the following areas of focus:

- Checking that Rabby Wallet has not introduced functionality that could leak its private key mnemonic phrase outside the application.
- Considering potential interface-related security issues, such as those previously reported by whitehat researchers.

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

- <https://github.com/RabbyHub/rabby-mobile/compare/932ec...37c09>

For the verification, we examined the following Git revision:

- 1fdb107bd592015bbc83ba33549bfe09e1b50a09

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

- Rabby Mobile Application:
<https://github.com/LeastAuthority/rabby-wallet/tree/audit3>

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:

- N/A.

In addition, this audit report references the following documents:

- Android native-bridge guidance:
<https://developer.android.com/privacy-and-security/risks/insecure-webview-native-bridges>
- Previous Security Audit Report by Least Authority on Rabby Wallet's Mobile Application (PDF)
(delivered via email on 2 September 2025)

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;
- Leakage or exposure of the private key mnemonic phrase outside the application;
- 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 from Ethereum-compatible networks. The mobile application combines account management with an in-app dApp browser, transaction review, swaps, bridging, and Gas Account functionality, requiring consistent handling of signing authority, account state, external requests, and sensitive information across both JavaScript and native components. This audit follows our previous Rabby Wallet Mobile audit, delivered on September 2, 2025, and focuses on changes introduced since that review.

We reviewed the key-derivation parameters protecting the local wallet vault and mnemonic cloud backups and identified an issue with resistance to offline password guessing. This concern was reported in our previous audit and remains applicable at the reviewed revision ([Issue A](#)). Our review of the native screenshot-recapture feature used for in-app feedback identified an issue with cleanup of the captured file ([Issue B](#)). We identified an issue with the scope of the security-engine review applied by the MiniSign batch-signing path to multi-transaction requests ([Issue C](#)). In the Perps bridge-deposit and Gas Account bridge top-up flows, we identified an issue with validation of backend-provided transactions ([Issue D](#)). The local consistency check applied to swap quotes also presented an issue with recipient validation ([Issue E](#)).

Our team also examined capture protection on the manual mnemonic backup and reveal screens and identified an issue with the application's screenshot-prevention control ([Issue F](#)). Our assessment of the lifecycle of mnemonics and private keys copied to the clipboard identified an issue with the production cleanup policy ([Issue G](#)). In addition, we evaluated how the Android in-app browser bridge attributes incoming requests to a dApp and identified an issue with frame authentication ([Issue H](#)). We also found that the limits applied to failed wallet-unlock attempts provide insufficient throttling ([Issue I](#)).

We further identified a write-sequence issue in how the keyring service persists its encrypted state ([Issue J](#)). We also found an issue with how the contract-deployment approval screen displays the attached native-token amount ([Issue K](#)). In addition, examining the operation field of a Safe transaction request revealed an opportunity to improve how its value is carried through pre-execution, display, and signing ([Suggestion 1](#)).

We noted that sensitive information was included in the React Native navigation state, which could lead to unintentional exposure through debugging, telemetry, or future integrations ([Suggestion 3](#)).

Beyond these findings, our team reviewed key material lifetime, WalletConnect and Safe integration, automatic authentication signatures, as well as feedback and telemetry data handling.

Dependencies

yarn npm audit reports several deprecated or unmaintained dependencies, including the deprecated react-native-randombytes library used for random number generation. We recommend replacing this dependency ([Suggestion 5](#)). In addition, one high-severity and two moderate-severity issues are reported in lodash and bn.js that may be relevant at runtime. We recommend upgrading the dependencies that pull in these vulnerable versions ([Suggestion 5](#)). A further moderate-severity issue in fflate affects the debug log viewer, but in this context, the issue would be very difficult to exploit and would have low impact.

The application uses three different versions of the react-native-keychain dependency. Two are forks created by the Rabby Wallet team, while the third is created by patching the upstream dependency at build time. All three versions add code to assist with debugging. One version of the dependency is selected as the default for use in production builds, while development and self-hosted builds are permitted to activate the other versions at runtime.

This approach to dependency management allows the Rabby Wallet team to debug any issues that may arise while migrating to a newer version of the dependency. Given the importance of the information stored in the keychain, a cautious approach to migration is warranted. Nevertheless, forking or patching this key dependency also risks introducing new bugs. While the added debugging code is generally only reachable via dedicated call paths, we found one instance of the debugging code exposing sensitive keychain contents on an error-handling path that is reachable in production builds ([Suggestion 4](#)).

Code Quality

Our team reviewed the repositories in scope and found that the codebase is organized into clearly defined modules, although code quality is inconsistent. Comments and developer-facing text use a mix of English and Chinese, while commented-out code, TODO/FIXME markers, and eslint-disable suppressions are widespread. However, the prevalence of these issues has decreased since the previous audit.

Tests

Our review found that test coverage has improved since the previous audit but remains insufficient given the size of the codebase ([Suggestion 2](#)).

Documentation and Code Comments

With the exception of vendored dependencies and AI agent skill files, the repository contains little documentation. Better documentation would provide insight into the design or technical choices made by the developers and would help with reasoning about the security of the components reviewed.

In addition, code comments are used sparsely and inconsistently. However, naming of types and functions is generally clear.

Scope

Given the extent of the changes since the previous audit (with approximately 375,000 lines of code added, removed, or modified), the audit focused on specific areas of concern highlighted by the Rabby Wallet team, namely the handling of private keys and mnemonic phrases, together with interface-related security concerns. The scope was sufficient, and our team did not identify any areas outside the scope requiring a follow-up audit.

Specific Issues & Suggestions

We list the issues and suggestions identified during the review in descending order of severity. 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	SEVERITY	STATUS
Issue A: Password Derivation Provides Limited Resistance to Offline Guessing	Medium	Partially Resolved
Issue B: Rejected Screenshot Events Skip Native Capture-File Cleanup	Medium	Resolved
Issue C: Batch Security Review Covers Only the Final Transaction	Medium	Unresolved
Issue D: Bridge Deposit Transactions Are Not Checked Against Local Intent	Medium	Unresolved
Issue E: Local Swap Verification Does Not Check the Output Recipient	Medium	Resolved
Issue F: Mnemonic Backup Views Do Not Enable Screenshot	Medium	Unresolved

Protection		
Issue G: Production Clipboard Policy Does Not Expire Copied Wallet Secrets	Medium	Resolved
Issue H: Missing Frame Authentication Can Misattribute Android App Requests	Medium	Resolved
Issue I: Failed Password Attempts Are Weakly Limited	Medium	Unresolved
Issue J: Clearing Keyring Storage Before Replacement Can Remove the Recoverable Wallet State	Medium	Resolved
Issue K: Contract-Deployment Action Omits the Attached Native-Token Amount	Low	Resolved
Suggestion 1: Preserve the Safe Operation Across Approval and Signing	Informational	Resolved
Suggestion 2: Improve Test Coverage	Informational	Resolved
Suggestion 3: Remove Sensitive Information from Navigation State	Informational	Resolved
Suggestion 4: Exclude Keychain Debugging Code from Production Builds	Informational	Resolved
Suggestion 5: Replace or Upgrade Vulnerable Dependencies	Informational	Partially Resolved

Issue A: Password Derivation Provides Limited Resistance to Offline Guessing

Location

[apps/mobile/src/core/services/encryptor.ts#L12-L39](#)

Synopsis

The local software-wallet vault and mnemonic cloud backups derive their encryption keys using PBKDF2-HMAC-SHA-256 with 5,000 iterations. This fixed work factor provides limited additional protection when a user chooses a guessable password and an attacker obtains the encrypted data.

Preconditions

For this issue to occur, an attacker must obtain a local vault or encrypted mnemonic backup through a separate compromise, and the particular wallet or backup must be protected by a sufficiently guessable user password.

Likelihood

Low.

An attacker must separately acquire encrypted private application data or a cloud backup, and the protecting password must fall within a feasible guessing budget. Strong, independently generated passwords materially limit this scenario.

Impact

High.

Successful password recovery could expose the software-wallet secrets contained in the local vault or the mnemonic contained in a cloud backup.

Severity

Medium.

Technical Details

The encryption key that protects seed phrases and other account secrets is derived from the user's application password using PBKDF2 with 5,000 iterations of HMAC-SHA-256. An attacker who obtains the encrypted data might be able to recover the key by brute force, as only a small amount of computation is needed to check each guess. The same derivation protects both copies of the secrets: the keyring service uses it to encrypt the local vault, and the cloud backup uses it to encrypt the mnemonic before upload.

There are no restrictions on the application password beyond a minimum length of eight characters, allowing users 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 recovering the encryption key increase significantly.

A fresh random salt and initialization vector are generated for each encryption. These prevent an attacker from reusing precomputed work across wallets and from recognizing repeated plaintexts, but they do not make individual guesses more expensive to test. The application's unlock throttling and cloud login prompts also do not mitigate this issue, as they govern access to the running application rather than offline evaluation of a copy of the ciphertext.

Users who complete onboarding using biometrics alone are not exposed in the same way, since the application generates a random 32-character password on their behalf.

Remediation

Our team highlighted this issue in the previous audit report. The Rabby Wallet team decided not to address the issue due to performance concerns associated with using a stronger key derivation function or a higher number of iterations. We note, however, that the application already performs a far more expensive derivation on the same devices. When synchronizing accounts from the browser extension, it decrypts the extension vault using `@metamask/browser-passworder`, whose current derivation parameters specify 900,000 iterations of PBKDF2-HMAC-SHA-256. This suggests substantial capacity to increase the iteration count used for data stored on the mobile device and in cloud backups without introducing significant performance overhead. Vault decryption occurs more frequently than extension synchronization, so we recommend calibrating the parameters against a target unlock time on the lowest supported device rather than adopting the synchronization value directly.

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, the parameters should be strengthened to at least 600,000 iterations of HMAC-SHA-256. We also recommend adopting a versioned derivation format that records and validates the parameters in use, migrating local vaults after successful authentication, and providing a deliberate migration path for existing cloud backups. Because

re-encrypting local state alone does not update cloud copies, compatibility with existing backups should be preserved until their replacements have been verified.

Status

The Rabby Wallet team has introduced a [draft branch](#) implementing upgraded PBKDF2 parameters and a migration mechanism for the local vault. However, at the time of verification, these changes were not yet active in production.

Verification

Partially Resolved.

Issue B: Rejected Screenshot Events Skip Native Capture-File Cleanup

Location

[apps/mobile/android/app/src/oldarch/RNScreenshotPreventModule.java#L151-L176](#)

[apps/mobile/ios/RabbyMobile/RNScreenshotPrevent.m#L92-L115](#)

[apps/mobile/src/components/Screenshot/hooks.ts#L459-L524](#)

[apps/mobile/src/core/storage/fs.ts#L266-L303](#)

Synopsis

The screenshot feedback feature writes a native recapture to a temporary file before JavaScript determines whether feedback is permitted. When the listener rejects the event, including on a revealed mnemonic view, it returns without deleting that file. Sensitive content can consequently remain in an additional application-owned image even when no feedback prompt appears.

Preconditions

For this issue to occur, a screenshot event must reach the native listener while the application is displaying sensitive content. For disclosure beyond Rabby Wallet, another party must be able to read the retained application-private file before the operating system evicts it.

Likelihood

Low.

A screenshot notification must occur while capture succeeds and JavaScript rejects the event. Unauthorized disclosure further requires access to Rabby Wallet's private cache or temporary directory. Standard application sandboxing, device encryption, and cache eviction constrain access and retention. This finding supplies no means to bypass them.

Impact

High.

If the recapture includes a readable mnemonic, a party able to recover the application-private file could recover accounts derived from the phrase, subject to any additional mnemonic passphrase. Less sensitive views can expose account or transaction information. Image contents and external file recovery remain conditional.

Severity

Medium.

Technical Details

On Android, the callback draws the root view into a PNG under `getCacheDir()/rabby-screen-capture`. On iOS, the notification handler renders the visible windows into a PNG under `NSTemporaryDirectory()/rabby-screen-capture`. Each is a fresh render of the view hierarchy rather than a copy of the operating system's screenshot. Both write the file before emitting the event, so neither consults the sensitive-screen flag before the capture exists on disk.

The JavaScript listener then checks the feature preference, capture success, and `shouldToastFeedbackByScreenshot`, which rejects a non-capturable or screenshot-report-free scene. The mnemonic component sets this flag while its words are revealed. These checks keep rejected images out of the feedback workflow, but deletion is coupled to the accepted branch. Only `saveScreenshotFrom(fullPath, { cleanupSource: true })` removes the native source in a finally block after copying it to the managed directory. Every earlier return leaves the file in place, and the bootstrap sweep covers `.screenshots`, not `rabby-screen-capture`.

Remediation

We recommend checking sensitivity before native recapture and assigning every generated file an explicit cleanup owner. Rejecting an event, disabling feedback, dismissing a prompt, or encountering an error should not leave a source file unmanaged.

Status

The Rabby Wallet team has implemented cleanup for rejected capture events and startup cleanup for both capture directories.

Verification

Resolved.

Issue C: Batch Security Review Covers Only the Final Transaction

Location

[apps/mobile/src/components2024/MiniSignV2/services/SignatureSteps.ts#L708-L747](#)

[apps/mobile/src/components2024/MiniSignV2/services/SignatureSteps.ts#L965-L1023](#)

[apps/mobile/src/components/Approval/components/MinisignTx/MiniSignTxV2.tsx#L631-L676](#)

[apps/mobile/src/screens/Home/components/DappActions/index.tsx#L179-L208](#)

Synopsis

MiniSign supports batches that contain several separately signed transactions. When security-engine review is enabled, the preparation service evaluates only the final transaction, and the detailed security header presents that transaction. Submission can nevertheless include every transaction in the batch. An earlier transaction can therefore be submitted without its own security-engine assessment being available for review.

Preconditions

This issue arises when an integrated feature submits two or more transactions to MiniSign with security-engine review enabled and a non-final entry contains behavior that warrants separate risk

assessment. The batch must then pass the applicable preparation and submission checks and be authorized by the user. Adversarial use additionally requires control over the relevant transaction source.

Likelihood

Low.

This requires a multi-transaction request containing a consequential earlier operation and user approval. Adversarial use also requires influence over an integrated transaction source.

Impact

High.

An earlier security-relevant transaction can affect assets or permissions without receiving the transaction-specific assessment provided for the final entry. The potential consequence depends on that transaction's effects and the user's authorization.

Severity

Medium.

Technical Details

`prepareContext` constructs and pre-executes every transaction. Each simulation includes earlier batch transactions in its pending-transaction list. After preparation, however, `getSecurityEngineResults` receives only the final calculated transaction. When security results need to be populated, `openUICore` reuses the preparation process and makes the same final-entry selection.

This engine can analyze multiple actions within that final transaction. Such action-level analysis does not cover earlier, separately signed batch entries. The UI's `MiniSecurityHeader` also receives only the final transaction and its simulation. The optional balance-change view and `onPreExecChange` callback use that final simulation response. Whether a backend response includes cumulative effects is outside the scope of this source review.

The Home DApp Actions component is a concrete caller that passes a transaction array into `openUI` with `enableSecurityEngine: true`. When the user confirms, the manager submits the prepared array. `sendBatch` processes each unsent entry sequentially.

This is a gap in transaction-specific security review, not a bypass of all batch checks. Preparation simulates each entry, gas and nonce errors are aggregated across the batch, and total gas cost is summed. The direct-signing path additionally rejects a batch if any pre-execution failed. The submission helper parses each transaction for history but does not run the security engine or introduce a separate review of earlier entries.

Remediation

We recommend evaluating each separately signed transaction and presenting an ordered batch review that associates its security results with that transaction. Aggregate warnings should retain the most significant result and clearly identify the relevant operation.

Status

The Rabby Wallet team has determined that exploitation would require control over an integrated transaction source, a consequential non-final transaction, and user approval. Because these prerequisites increase attack difficulty and the batch flow supports required product functionality, remediation has been deferred.

Verification

Unresolved.

Issue D: Bridge Deposit Transactions Are Not Checked Against Local Intent

Location

[apps/mobile/src/screens/Perps/components/PerpsDepositPopup/index.tsx#L624-L718](#)

[apps/mobile/src/screens/Perps/hooks/usePerpsDeposit.ts#L151-L173](#)

[apps/mobile/src/core/apis/gasAccount.ts#L36-L112](#)

[apps/mobile/src/hooks/useSigner.tsx#L261-L279](#)

Synopsis

The Perps bridge-deposit and Gas Account bridge top-up flows accept a backend-provided approval spender and transaction without independently checking that they implement the deposit the user requested. Supported software accounts can submit these transactions through MiniSign's direct path. A corrupted quote can therefore supply signed operations inconsistent with the locally displayed deposit intent.

Preconditions

This issue arises when a user initiates a Perps bridge deposit or a Gas Account bridge top-up from a software account that routes to MiniSign's direct path, where the security engine is disabled, and an attacker controls the quote response for that deposit through compromise of the backend or the transport carrying it. The substituted transactions must remain executable for that account on the selected chain, passing pre-execution, gas, and resource checks and, on the Gas Account branch, the quote-amount match. The user must also authorize submission.

Likelihood

Low.

Adversarial use requires control of the specific backend response used for a deposit and a user who proceeds with that deposit.

Impact

High.

An inconsistent transaction may affect asset destinations or spending permissions for the authorized account.

Severity

Medium.

Technical Details

Both features request a quote using the account, source token and chain, and a locally calculated raw amount. They check existing allowance against the response's `approve_contract_id` and build an approval for the calculated amount when needed. They then append the response transaction's sender, destination, value, calldata, and chain without checking its decoded meaning against a locally defined deposit policy.

The allowance check determines whether an approval is needed but does not establish that the returned spender belongs to the expected bridge. After authorization, `usePerpsDeposit` resets its `MiniSign` context and passes the batch to direct signing for mnemonic and private-key accounts.

The Gas Account bridge branch additionally requires the quote amount to equal the current form amount before building transactions. This prevents reuse across differing displayed amounts. It does not establish that the returned transaction encodes that amount or the intended top-up destination. Its signing helper also resets the previous context and selects direct signing for software accounts.

The shared `openDirect` wrapper disables the optional security-engine invocation. `MiniSign` nevertheless pre-executes every transaction, rejects failed pre-execution in the direct path, and checks applicable gas and available resources. The provider requires the sender to match the selected account. These controls do not compare the signed destination or calldata with the locally requested deposit.

Locally constructed direct deposits are excluded. In particular, Gas Account's direct top-up branch selects its recipient from `L2_DEPOSIT_ADDRESS_MAP` and builds the transfer locally.

Remediation

We recommend enforcing a local deposit policy before directly submitting backend-built transactions. This includes validating the chain, sender, bridge destination, approval spender, native value, and decoded asset, amount, and recipient parameters against the request.

When a supported route cannot be validated locally, a full transaction review with clearly associated simulation and security results should be required. Existing amount-consistency, account, freshness, and pre-execution checks should be retained as additional controls.

Status

The Rabby Wallet team has determined that exploitation would require control over a quote response through compromise of the backend or authenticated transport, followed by user approval. Because local calldata decoding and intent validation across multiple bridge protocols require substantial implementation work, remediation has been deferred for this round.

Verification

Unresolved.

Issue E: Local Swap Verification Does Not Check the Output Recipient

Location

[apps/mobile/src/screens/Swap/hooks/quote.tsx#L965-L1044](#)

[apps/mobile/src/screens/Swap/hooks/quote.tsx#L473-L489](#)

[apps/mobile/src/screens/ConvertDust/hooks/useBatchSwapTask.ts#L193-L229](#)

Synopsis

Rabby Wallet checks selected fields in decoded swap calldata against the requested swap. The check omits the recipient of the output tokens, even when the decoder supplies this address. As a result, a quote can pass the local consistency check without establishing that its output is payable to the account initiating the swap.

Preconditions

This issue arises when a supported quote service returns calldata whose output recipient differs from the account initiating the swap through a router that permits a separate recipient. The transaction must still

satisfy the remaining amount, token, router, spender, and execution checks, and the user must proceed with the swap or authorize the conversion batch that uses the quote.

Likelihood

Low.

The problematic calldata must originate from a supported quote service, match the applicable router and spender restrictions, and survive the service's validation and wallet execution checks.

Impact

High.

For a supported router that permits a separate output recipient, an inconsistent quote could direct the purchased assets elsewhere.

Severity

Medium.

Technical Details

The quote request includes the user's address. For the inspected 1inch adapter, the installed swap library maps the quote service's calldata into the transaction and decodes the destination receiver from that calldata. The resulting `DecodeCalldataResult` exposes it as `toTokenReceiver`.

In `verifyCalldata`, Rabby Wallet compares the decoded input token, input amount, output token, and minimum output amount with the quote. Neither this function nor `verifySdk` takes an expected recipient for comparison.

The verdict is used by the quote-selection interface and by `ConvertDust`'s provider selection. The transaction builder preserves the quote's calldata.

Remediation

We recommend passing the intended receiving account into the swap verifier and checking every supported decoder's recipient against it. Router-specific recipient conventions should be handled explicitly. When a decoder cannot establish the recipient, this limitation should be represented separately from successful verification, with appropriate review required before automated execution.

Status

The Rabby Wallet team has added the expected recipient to swap verification and applied the result to `ConvertDust`.

Verification

Resolved.

Issue F: Mnemonic Backup Views Do Not Enable Screenshot Protection

Location

[apps/mobile/src/screens/Address/CreateChooseBackup.tsx#L66-L79](#)

[apps/mobile/src/screens/Address/Backup.tsx#L215-L249](#)

[apps/mobile/src/components2024/SeedPhraseManualBackup/SeedPhrase.tsx#L695-L705](#)

[apps/mobile/src/components2024/GlobalBottomSheetModal/GlobalBottomSheetModal.tsx#L134-L139](#)

[apps/mobile/src/hooks/navigation.tsx#L711-L723](#)

Synopsis

Rabby Wallet provides native screenshot protection for sensitive screens. The manual mnemonic backup views used during wallet creation and existing-wallet backup explicitly disable that protection, and their routes provide no fallback. The revealed words and the existing-wallet QR representation can therefore be displayed without activating the application's capture-prevention control.

Preconditions

This issue arises when a user reaches wallet creation or the existing-wallet backup flow and reveals the mnemonic or its QR representation, with no other active sensitive view independently enabling capture prevention. Disclosure additionally requires a capture mechanism to record the revealed view and another party to obtain the resulting image.

Likelihood

Low.

The user must reach a backup flow and deliberately reveal the words or QR code. These requirements, together with platform-specific capture restrictions, limit the likelihood of unauthorized disclosure.

Impact

High.

A readable capture can disclose the mnemonic. A party who obtains it can recover accounts derived from that phrase, subject to any additional mnemonic passphrase.

Severity

Medium.

Technical Details

Both entry points open `SEED_PHRASE_MANUAL_BACKUP` with `preventScreenshotOnModalOpen: false`. `CreateChooseBackup` does so during wallet creation. The global modal manager marks a modal sensitive only when that flag is truthy, and neither route appears in `PROTECTED_SCREEN`.

`SeedPhrase` initially blurs the word list. Once revealed, its read-mode QR action passes `words.join('')` to a further modal whose flag is also false, and `SeedPhraseQRCode` uses that string directly as the QR value. The component sets a separate screenshot-report-free flag while revealed, but that only suppresses the feedback prompt. The warning shown after detection cannot retract a capture already taken. The independently enabled iOS app switcher blur indicates that this finding does not establish an app switcher snapshot failure.

Remediation

We recommend restoring capture protection for every modal that displays mnemonic words or their QR representation. Components containing secrets should acquire protection before rendering and retain it until the content is removed. Route-level coverage should also be added where appropriate.

Status

The Rabby Wallet team has accepted the risk because screenshot availability is considered a product requirement.

Verification

Unresolved.

Issue G: Production Clipboard Policy Does Not Expire Copied Wallet Secrets

Location

[apps/mobile/src/hooks/appSettings.ts#L177-L184](#)

[apps/mobile/src/utils/clipboard.ts#L17-L100](#)

[apps/mobile/src/components2024/SeedPhraseManualBackup/SeedPhrase.tsx#L830-L835](#)

[apps/mobile/src/screens/Address/BackupPrivateKeyScreen.tsx#L135-L145](#)

Synopsis

Rabby Wallet permits copying mnemonic phrases and private keys to the system clipboard. Production builds select a cleanup policy that disables the application's time-based clearing mechanism. As a result, copied secrets have no Rabby-enforced expiry and can remain available until another clipboard operation or platform control removes them.

Preconditions

This issue arises whenever a user reveals a mnemonic or private key in a backup view and uses the Copy action, since every production build selects the 'pasted' policy. Disclosure additionally requires the copied value to remain on the clipboard and become accessible to an unintended recipient.

Likelihood

Low.

The user must intentionally reveal and copy a secret, and copying a mnemonic also requires confirmation of a warning. Unauthorized disclosure requires a separate permitted clipboard consumer, an unintended paste, or local access.

Impact

High.

A clipboard recipient who obtains a private key can control the associated account. A mnemonic can expose accounts recoverable from the phrase, subject to any additional mnemonic passphrase.

Severity

Medium.

Technical Details

Both copy actions reach one helper. The mnemonic confirmation writes `words.join(' ')` with `Clipboard.setString` and calls `onCopiedSensitiveData`, while the private-key view passes the revealed key to `CopyAddressIcon`, which calls the same helper from its success callback. The policy getter, however, returns 'pasted' whenever `__DEV__` is false, and both `onCopiedSensitiveData` and the startup `startCheckClearAction` require 'copy' before scheduling any action.

The 'pasted' policy has a working cleanup path. `onPastedSensitiveData` clears the clipboard, but is triggered only when an instrumented Rabby Wallet import input receives the paste, so it provides no expiry after copying and never runs when the user pastes into another application. The installed clipboard

dependency writes through standard `ClipData.newPlainText` on Android and the general clipboard's string property on iOS, neither of which offers an application-specific expiry or local-only option.

Remediation

We recommend avoiding clipboard export for mnemonic phrases and private keys where practical. If retained, a short production cleanup lifetime should be applied, clearing only the value Rabby Wallet originally placed on the clipboard.

Status

The Rabby Wallet team has implemented sensitive clipboard handling in a [separate PR](#) outside the target verification revision.

Verification

Resolved.

Issue H: Missing Frame Authentication Can Misattribute Android dApp Requests

Location

[apps/mobile/src/core/bridges/useBackgroundBridge.ts#L111-L164](#)

[apps/mobile/src/core/bridges/middlewares/index.ts#L29-L45](#)

[apps/mobile/src/components/WebView/DappWebViewCore.tsx#L524-L540](#)

[packages/rn-webview-bridge/src/browserScripts.ts#L209-L255](#)

Synopsis

The Android dApp bridge accepts an origin reported inside a WebView message without authenticating the sending frame. The installed native message interface is available to subframes, while the JavaScript origin wrapper is applied to the main frame. Consequently, a request from third-party frame content can be attributed to the connected top-level dApp.

Preconditions

This issue arises on Android when a connected dApp embeds third-party frame content that can execute JavaScript, and the top-level dApp either has a connected account or the user approves connection under that identity. The message must reach an initialized bridge with a stable origin and pass method-admission checks, and any resulting transaction or signature must still satisfy the remaining account, unlock, and approval controls.

Likelihood

Medium.

A user must visit a connected dApp that embeds attacker-controlled frame content on Android. The top-level site does not have to be compromised itself, but a suitable embedded frame must be present and under attacker control.

Impact

Medium.

The wallet can present a provider request with an incorrect dApp identity and apply the top-level dApp's connection permissions. Transactions and standard message signatures still require their downstream approval controls. Responses are sent into the top-level document, so direct disclosure of a signature to a cross-origin subframe is also not established.

Severity

Medium.

Technical Details

The installed WebView dependency registers its message object with `addJavascriptInterface`. Android documents that this exposes the object to all frames without giving the application an authenticated calling-frame origin.

Rabby Wallet's injected wrapper compares a message's declared origin with the executing document's origin. The main-frame injection setting does not remove the separate native interface available to subframes.

On receipt, `useBackgroundBridge` parses the message body and compares the host in its declared origin with the initialized bridge's host. No native frame identity is consulted. `BackgroundBridge` then forwards the message to the provider engine, whose origin middleware assigns the current top-level URL's origin.

Remediation

We recommend using a native frame-aware channel that supplies authenticated origin and main-frame information, rejecting subframe provider requests unless explicitly supported. That identity should be preserved through authorization and approval display.

Status

The Rabby Wallet team has added WebView frame capability authentication and origin validation for Android dApp-provider messages.

Verification

Resolved.

Issue I: Failed Password Attempts Are Weakly Limited

Location

[apps/mobile/src/core/utils/unlockRateLimit.ts#L4-L9](#)

[apps/mobile/src/core/utils/unlockRateLimit.ts#L73-L117](#)

[apps/mobile/src/core/apis/lock.ts#L140-L160](#)

[apps/mobile/src/core/apis/lock.ts#L514-L562](#)

[apps/mobile/src/components/AuthenticationModal/AuthenticationModal2024.tsx#L228-L337](#)

[apps/mobile/src/core/apis/mnemonic.ts#L79-L88](#)

Synopsis

Rabby Wallet limits failed password attempts when unlocking a locked wallet, but its production configuration permits 1,000 incorrect candidates before a five-minute cooldown. Password

reauthentication for sensitive operations, including mnemonic and private-key export, takes a different route when the keyring is already unlocked and applies no attempt budget at all. Neither route meaningfully constrains guessing by a party who can interact with the application on an OS-unlocked device.

Preconditions

This issue arises when the wallet uses a guessable user-selected password and another party can submit candidates through the local application, either against the lock screen or a password-authenticated sensitive action while the keyring remains unlocked. The correct password must fall within the attempted candidates, and export abuse additionally requires exportable software key material.

Likelihood

Low.

An attacker must be able to interact with the application on an OS-unlocked device, and the wallet password must fall within the candidates they can submit. Guesses go through the application interface, which limits the practical rate compared with offline guessing.

Impact

High.

A correct guess grants wallet access and can authorize export of a software wallet's mnemonic or private key, allowing control of the corresponding accounts outside the device. The configured threshold and the missing counter increase opportunities to discover a weak password. However, neither makes an incorrect password valid.

Severity

Medium.

Technical Details

On the locked route, the production `MULTIPLE_FAILED_CONF.limit` is 999, with an adjacent comment describing it as a temporary measure to disable the feature. The development limit is five, so routine development testing does not exercise the production policy. The lock API checks for an existing cooldown before submitting a password to the keyring and updates the failed-attempt record on failure, but the counter compares the previous count with the limit before incrementing it. Starting from a cleared record, failures one through 999 increment the count, and failure 1,000 starts the cooldown, by which point the thousandth candidate has already been checked. The record is persisted and reset on successful unlock.

The reauthentication route bypasses that limiter entirely. Both authentication modals use `verifyPasswordOrUnlock`, which invokes the unlock API and its limiter only when the keyring is locked. When it is already unlocked, it calls `throwErrorIfInvalidPwd`, which invokes `keyringService.verifyPassword` directly without consulting or updating the failed-unlock record. The modals display the error and allow resubmission, with no independent cooldown. The export APIs repeat this same verification, so the missing budget guards access to mnemonic and private-key export.

Lowering the production threshold does not protect the unlocked route, and adding a counter there does not correct the threshold.

Remediation

We recommend applying one shared failed-attempt policy to password verification in both routes, with a small, documented production budget and a counting rule that matches it exactly. Increasing delays should be applied where appropriate, and the failure state should be persisted. All callers should obtain a

result that distinguishes successful verification from cancellation, failure, or an attempt rejected by policy, and sensitive-operation dialogs should display the cooldown and disable submission while verification is pending.

Status

The Rabby Wallet team has reviewed the existing password attempt policy and accepted the associated risk without changing its implementation.

Verification

Unresolved.

Issue J: Clearing Keyring Storage Before Replacement Can Remove the Recoverable Wallet State

Location

[apps/mobile/src/core/services/shared.ts#L144-L155](#)

[apps/mobile/src/core/storage/mmkv.ts#L87-L105](#)

[packages/service-keyring/src/keyringService.ts #L291-L301](#)

Synopsis

Rabby Wallet stores its encrypted keyring state in a dedicated MMKV store. Each update clears that store before writing the replacement record, leaving no application-managed previous generation if the replacement does not complete. A storage or process failure at this boundary can prevent recovery of locally stored software wallets.

Preconditions

For this issue to occur, a keyring update must clear an existing local record, and the replacement must not become recoverable before the process or storage operation fails.

Likelihood

Low.

The failure requires an interruption or unsuccessful storage operation after the clear takes effect and before a usable replacement is retained. This boundary occurs during standard keyring persistence.

Impact

High.

Loss of the only recoverable local keyring can prevent access to software wallets on that device. A user without an independent recovery phrase or private-key backup may lose access permanently.

Severity

Medium.

Technical Details

When the keyring state changes, Rabby Wallet saves it by calling `clearAll()` on the keyring store and then writing the new value with `setItem()`. Those are two separate native calls, and no mechanism ties them together. The store holds only one key, so the first call wipes the only copy of the wallet record

before the replacement is written. `clearAll()` truncates the file and flushes the change to disk before returning.

If the store returns empty, the application cannot detect it. The storage adapter returns `null` when a value is missing or cannot be parsed, `shared.ts` passes `keyringState || {}` into `loadStore`, and the keyring starts up with no vault and no boot record, exactly as it would for a wallet that had never been set up.

Remediation

We recommend keeping the previous keyring record until its replacement has been written, read back, and checked. A standard update should not destroy the only valid copy first. The new record should be written alongside the existing record and confirmed as readable and valid before the previous record is removed.

Status

The Rabby Wallet team has replaced the clear-before-write behavior with verified checkpointing and replacement-write validation.

Verification

Resolved.

Issue K: Contract-Deployment Action Omits the Attached Native-Token Amount

Location

[apps/mobile/src/components/Approval/components/Actions/DeployContract.tsx#L8-L27](#)

[apps/mobile/src/components/Approval/components/Actions/components/TransactionActionList.tsx#L196](#)

[apps/mobile/src/components/Approval/components/Actions/DeployContract.tsx#L1270-L1287](#)

[apps/mobile/src/components/Approval/components/Actions/index.tsx#L121-L155](#)

[apps/mobile/src/components/Approval/components/TxComponents/BalanceChangeWrapper.tsx#L36-L72](#)

Synopsis

A contract-creation transaction can include a native-token payment in addition to its deployment bytecode. Rabby Wallet's deployment action shows only a generic description and omits the attached amount, although that value is included in the transaction submitted for signing. A user relying on the action description may therefore approve a payment of which they were unaware.

Preconditions

This issue arises when a connected dApp requests a contract deployment that carries a nonzero native-token value, the approval screen renders the deployment action, and the user approves without noticing the payment in the raw transaction view or the balance-change display.

Likelihood

Low.

The request must be classified as a deployment and carry a nonzero value, and the user must overlook the payment in the other review elements. The raw transaction view and the balance-change display reduce this likelihood.

Impact

Medium.

The omission can lead a user to approve an unintended native-token payment. It affects only the transaction explanation. The signed value is unchanged, and user approval is still required.

Severity

Low.

Technical Details

SignTx retains the requested value in its transaction state and includes it in both the analysis requests and the final approval result. The action list, however, renders `DeployContract` without any props, and the component contains only a translated title and description, so it cannot display the value. The surrounding action component already receives the raw transaction and chain.

The raw-data view in the action header includes the transaction value, and the balance-change wrapper can show nonempty simulated changes, although it hides successful deployment simulations that record no changes. The payment is therefore not hidden from the entire screen but is missing from the description a user is most likely to rely on.

Remediation

We recommend passing the transaction value and chain into the deployment action and prominently displaying any nonzero native-token payment. The amount should be derived locally from the transaction selected for signing, independent of remote classification or simulation results.

Status

The Rabby Wallet team has updated contract-deployment actions to display any nonzero native-token value attached to the local transaction.

Verification

Resolved.

Suggestions

Suggestion 1: Preserve the Safe Operation Across Approval and Signing

Location

<apps/mobile/src/components/Approval/components/SignTx/SignTx.tsx#L839>

Synopsis

A Safe transaction's `operation` field selects between `CALL` and `DELEGATECALL`, the latter executing the target's code in the Safe's own context. Rabby's approval flow does not carry one normalized value for this field from the incoming request through pre-execution, display, and signing.

The pre-execution request derives its `delegate_call` flag from the original request using JavaScript truthiness rather than a numeric comparison:

```
const delegateCall = isGnosisAccount ? !!params?.data?.[0]?.operation :  
false;
```

Numeric 1 correctly produces `true` and numeric 0 produces `false`, but the string "0" and other nonempty strings are truthy and also produce `true` despite denoting CALL. The provider boundary validates only the sender and chain identifier, so the field arrives unvalidated and untyped. The flag is sent with the pre-execution request. Consequently, an incorrectly derived value produces a transaction explanation for an operation not specified by the request.

Mitigation

We recommend normalizing the operation once at the provider boundary to the supported Safe enum and rejecting, after normalization, values that are neither 0 nor 1 rather than coercing them.

Status

The Rabby Wallet team has implemented a numeric comparison for the Safe operation value in a separate revision outside the target verification snapshot (fad14b97be962b4bee007337773db6ce03d3ab54).

Verification

Resolved.

Suggestion 2: 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 substantially expanded the mobile test suite and introduced a dedicated integration-test CI workflow.

Verification

Resolved.

Suggestion 3: Remove Sensitive Information from Navigation State

Location

<apps/mobile/src/navigation-type.ts>

Synopsis

Sensitive information, including mnemonic seed phrases, passphrases, and private keys, is passed between application components using the React Native navigation state. Additional handling surfaces for this sensitive information increase the risk of unintended exposure through application internals, debugging, telemetry, or future integrations.

Mitigation

We recommend isolating sensitive information in a dedicated in-memory store with an explicit and auditable interface. Navigation state can include handles or identifiers that allow sensitive information to be retrieved from this store, rather than passing the sensitive information directly in the navigation state.

Status

The Rabby Wallet team has moved the sensitive information from the React Native navigation state to a dedicated in-memory store. Single-use handles are used to retrieve and remove information from the store so that it does not persist in memory for longer than necessary.

Verification

Resolved.

Suggestion 4: Exclude Keychain Debugging Code from Production Builds

Location

[.yarn/patches/react-native-keychain-npm-10.0.0-rabby.patch#L185](#)

[apps/mobile/src/core/apis/keychainCommon.ts#L751](#)

[apps/mobile/src/core/apis/keychainCommon.ts#L786](#)

[apps/mobile/src/core/apis/keychainCommon.ts#L1397](#)

Synopsis

Information intended for debugging issues with the native keychain is exposed on an error-handling path used in production builds. This information includes the username and password stored in the keychain. Most call paths strip the username and password from the debugging information, suggesting that the risk of inadvertent exposure has been recognized and these specific pieces of information are not always necessary for debugging.

Mitigation

We recommend excluding debugging code from production builds to avoid inadvertent information exposure while preserving the ability to debug issues with the native keychain. The existing mechanism that allows different builds to use different versions of the `react-native-keychain` dependency can be used to set a default version for production builds that does not include debugging code, preventing this code from being reached unexpectedly.

Additionally, we recommend minimizing the debugging information that is retrieved at the native layer, rather than relying on higher layers to sanitize the information that has been retrieved.

Status

The Rabby Wallet team has removed credentials from the debugging information in the identified error-handling path. This is a narrow solution that addresses the specific potential leak of credentials that was reported, without reducing the risk of similar issues arising in the future.

Verification

Resolved.

Suggestion 5: Replace or Upgrade Vulnerable Dependencies

Location

[apps/mobile/package.json#L214](#)

Synopsis

The application has a direct dependency on the deprecated `react-native-randombytes` library and transitive dependencies on versions of `lodash` and `bn.js` that contain vulnerabilities of high and medium severity.

Mitigation

We recommend replacing `react-native-randombytes` with `react-native-get-random-values`. In addition, resolutions should be used to prevent vulnerable versions of `lodash` and `bn.js` from being pulled in transitively.

Status

The Rabby Wallet team has removed the dependency on `react-native-randombytes`. The vulnerability in `bn.js` has been resolved by upgrading to version 4.12.5. Dependencies on a vulnerable version of `lodash` remain. The team has requested details of the specific vulnerabilities and dependency paths.

Version 4.17.21 of `lodash` is vulnerable to CVE-2026-4800, CVE-2026-2950, and CVE-2025-13465. This is a direct dependency of the `apps/mobile` workspace. These vulnerabilities can be resolved by upgrading to version 4.18.0.

Verification

Partially Resolved.

About Least Authority

Founded in 2011 and based in Berlin, Germany, since 2016, Least Authority provides security consulting for privacy-preserving technologies and decentralized systems. We are committed to transparency, openness, and advancing secure digital infrastructure.

We believe that people have a fundamental right to privacy and that secure solutions enable people to use the internet and other connected technologies more freely. Our security consulting services help teams make their solutions more resistant to unauthorized access and unintended system manipulation. We support teams from the design phase through production launch and beyond.

Our team has experience reviewing code for security vulnerabilities and specific attack vectors in multiple languages, including Rust, Go, Python, C, C++, Solidity, Cairo, Circom, Noir, Haskell, OCaml, TypeScript, Swift, Zokrates, and others. We have reviewed implementations of cryptographic protocols and distributed system architecture in blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, we use various tools to scan code and networks and build custom tools as necessary.

For more information about our security consulting, please visit:

<https://leastauthority.com/security-consulting>.

Our Methodology

We use a transparent review process that provides the project team visibility into our findings, assumptions, and remediation considerations throughout the audit. The goals of our security audits are to improve the quality of the systems we review and support sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Code Review

In manually reviewing all of the code, we identify potential issues, including those related to code logic, error handling, cryptographic implementations, and correctness with respect to specifications, standards, and best practices. We also evaluate areas where more defensive programming could reduce the risk of future mistakes and accelerate 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.

When appropriate, we also use artificial intelligence (AI) to support the efficiency of our security audits without reducing rigor or reliability. AI-assisted outputs are carefully guided, reviewed, and validated by our team so that conclusions remain accurate and defensible. This supports stronger audit decisions, not only faster review.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and white box penetration testing, supplemented by various tools, including AI, that support the code review process. Discussions with the development team help clarify their vision for the software. Installing and using the relevant software helps us understand user interactions and roles, while also informing our analysis of threat models and attack surfaces. Design documentation, relevant papers, other audit results, similar projects, source code dependencies, and open issue tickets may also be reviewed to inform the analysis. We identify potential vulnerabilities, and, when appropriate, create issue entries. For each entry, we follow the issue investigation and remediation process described below.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and supporting successful remediation. Whenever a potential issue is identified, we raise it with the client through a dedicated secure communication channel to discuss its validity and impact, to the extent these can be determined at that stage. This process allows potentially relevant findings to be reviewed with the client early, before they are documented in greater detail in the initial audit report. For each confirmed issue, we assess impact and feasibility and use those ratings to determine the overall severity, following the [OWASP Risk Severity Matrix](#).

Suggested Solutions

We identify immediate and comprehensive mitigations available to live deployments and then suggest remediation requirements for future releases. When code is already deployed in production, we include mitigation strategies in the report to provide interim support until appropriate remediation is available. The mitigation and remediation recommendations should be reviewed by the development team and deployment engineers. After delivering the Initial Audit Report, we remain available to support implementation of the proposed mitigations and remediations before the verification review.

Development teams should address findings in an order and with an urgency that prioritizes the security of their users. Findings should be addressed as soon as possible, but remediation should be carried out pragmatically and should not cause unnecessary disruption to project timelines.

Although issues must be handled on a case-by-case basis, we work toward a resolution timeline that balances the impact on users and the needs of the project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess whether the issues and suggestions have been sufficiently addressed. When this analysis is complete, we update the report and provide a Final Audit Report that can be published in full. If any critical issues remain, we recommend that the Final Audit Report not be published and that users and other stakeholders be alerted to the impact. We encourage all findings to be addressed and the Final Audit Report to be shared publicly to support transparency and advance security knowledge within the industry.