Kraken Mobile Wallet

Security Assessment

January 10, 2024

Prepared for:
Andrew Koller, Eric Kuhn, and Thomas Roth
Payward, Inc.

Prepared by: **Jim Miller, Emilio López, Maciej Domański, and Paweł Płatek**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineering directors were associated with this project:

Anders Helsing, Engineering Director, Application Security
anders.helsing@trailofbits.com

Jim Miller, Engineering Director, Cryptography
james.miller@trailofbits.com

The following consultants were associated with this project:

Jim Miller, Engineering Director, Cryptography
james.miller@trailofbits.com

Emilio López, Consultant
emilio.lopez@trailofbits.com

Maciej Domański, Consultant
maciej.domanski@trailofbits.com

Paweł Płatek, Consultant
pawel.platek@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
September 28, 2023	Pre-project kickoff call
October 10, 2023	Status update meeting #1
October 16, 2023	Status update meeting #2
October 20, 2023	Delivery of report draft
October 20, 2023	Report readout meeting
January 10, 2024	Delivery of comprehensive report with fix review

Executive Summary

Engagement Overview

Payward engaged Trail of Bits to review the security of the Kraken mobile wallet, a non-custodial wallet for the iOS and Android platforms.

A team of four consultants conducted the review from October 2 to October 19, 2023, for a total of seven engineer-weeks of effort. Our testing efforts focused on the Kraken mobile application. With full access to the React Native source code, we performed static and dynamic testing of the Kraken mobile wallet, using automated and manual processes. We manually reviewed the system's uses of cryptography for any vulnerabilities to known cryptographic attacks, focusing on algorithm selection, dependencies, and common best practices for cryptography.

Observations and Impact

We found that the Kraken mobile wallet is well structured and generally written defensively. We did not identify any high-severity issues that could enable a direct, remote attack that would result in a significant compromise, such as theft of user funds.

Given the financial nature of the Kraken mobile application, our audit also concentrated on assessing the application's resistance against two major threats: the presence of malware and physical access to users' smartphones by malicious actors. Regarding the former, we found a feasible scenario in which a malicious application could capture users' screens displaying mnemonic phrases, thereby leading to a loss of funds (TOB-KMW-6). Regarding the latter, we discovered vulnerabilities that could enable attackers to bypass local biometric authentication (TOB-KMW-8), missing reauthentication on critical functionalities (TOB-KMW-9), and the lack of a lockdown mechanism following incorrect password entries (TOB-KMW-15).

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Kraken take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.

- **Add the WalletConnect Verify API to the product.** The Verify API enables applications to securely validate whether the end user is on the correct domain. This solution makes phishing attacks harder to conduct.

- **Implement static analysis tools in the CI/CD pipeline.** Implementing additional tools such as those presented in appendix D will help to automatically find issues in the code that could lead to security vulnerabilities before they are merged into the codebase.

- **Support multiple data providers on the wallet's back end (Kraken Harmony).** Details of the Kraken back-end service were not reviewed during the audit. Kraken should ensure that its system is properly decentralized. To do so, we recommend that Kraken host blockchain nodes of its own instead of using centralized services like Infura. Alternatively, connect to multiple data providers and cross-verify the results from them.

- **Consider allowing wallet users to configure their own list of trusted nodes.** This will reduce the number of third parties that users have to trust implicitly, which will improve user trust in the mobile application. Currently, wallet users have to send requests via Kraken Harmony. Moreover, this feature would enable users to eliminate the risk of transaction front-running.

- **Protect the application against malicious files that could pass through the NFT feature.** Implement device-side countermeasures like sanitization with appropriate unit testing for these scenarios, such as rendered SVG files that call external resources.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	8
Low	8
Informational	8
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	2
Authentication	3
Configuration	4
Cryptography	2
Data Exposure	4
Data Validation	3

Denial of Service	1
Error Reporting	1
Patching	3
Timing	1

Project Goals

The engagement was scoped to provide a security assessment of the Kraken mobile wallet. Specifically, we sought to answer the following non-exhaustive list of questions:

- Does the Kraken mobile wallet safely manage secret data?
- Is it possible to extract users' wallet seeds from the application?
- Are cryptographic algorithms implemented and used securely and according to their specifications in the Kraken wallet?
- Is it possible to bypass any of the Kraken wallet's confirmation screens (e.g., to automatically confirm transactions without user consent)?
- Is there any threat associated with using a QR code scanner in the Kraken mobile wallet?
- How securely are secrets stored?
- Can an attacker exploit any exported component of the Android application?
- Are there any architectural design flaws in the Android or iOS applications?
- Does the Android application use the `WebView` class?
- Does the minimum API level required for the Kraken wallet pose any risk?
- Can users' privacy or identity be compromised?
- Can users be tricked into signing misleading transactions?
- Is the NFT feature securely implemented?

Project Targets

The engagement involved a review and testing of the following target:

mobile

Repository	N/A
Version	Zip file provided on October 2, 2023
Type	React Native
Platforms	Android, iOS

Payward also provided access to the following codebases to aid the review:

harmony

Repository	N/A
Version	Zip file provided on October 2, 2023
Type	TypeScript
Platform	Linux

groundcontrol

Repository	N/A
Version	Zip file provided on October 2, 2023
Type	TypeScript
Platform	Linux

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- A review of the provided documentation and threat model
- Static analysis of the codebase and triaging of the results
- Local building of the codebase for the Android platform for manual testing
- A review of the cryptographic primitives used throughout the mobile wallet
- A manual review of the derivation and storage of cryptographic secrets
- A manual review of the WalletConnect implementation
- A review of the wallet for common mobile wallet issues
- An analysis of the QR code scanner's security
- An analysis of whether the application is vulnerable to intent injection attacks
- An analysis of the Android APK with the Android manifest file configuration for potential misconfiguration of permissions
- A review of the security of the transaction singing mechanism against phishing attacks
- A review of how the wallet consumes data received from remote services
- A review the decentralization of the wallet in relation to data providers
- A review the NFT transactions along with the security context of the rendered NFTs

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- We did not test the robustness of the WalletConnect Verify API, which we recommend implementing to make phishing attacks harder to conduct, or the source code of WalletConnect itself.

- We were granted access to both the `harmony` and `groundcontrol` modules; however, these modules were provided solely to assist in our review process and were not subjected to detailed testing or evaluation as part of this audit.
- We noted the presence of several outdated dependencies, and we referenced third-party code while reviewing specific components. However, we did not perform a detailed review of third-party dependencies.
- Our main focus of the audit was the Android platform. Although the Android and iOS versions of the application share the React Native codebase, we did not perform a dynamic analysis of the iOS application.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Semgrep	A static analysis tool designed to identify bugs and specific code patterns across multiple languages	Appendix D
CodeQL	A code analysis engine developed by GitHub to automate security checks	Appendix D

Areas of Focus

Our automated testing and verification work focused on the following system properties:

- The system does not produce undefined behavior.
- The code does not contain security or quality issues.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	We found no significant issues concerning the proper use of mathematical operations.	**Satisfactory**
Auditing	We did not find any secrets that are exposed in local logs. We did not have access to the remote `harmony`, `groundcontrol`, or ElectrumX logs.	**Further Investigation Required**
Authentication / Access Controls	The wallet follows general security principles, such as allowing users to enable biometric authentication and to set up passwords. However, the application could have a better password security policy (TOB-KMW-10). We found that local biometric authentication could be bypassed (TOB-KMW-8). Reauthentication is not enforced on every sensitive operation (TOB-KMW-9). Also, there is no lockout mechanism following several consecutive failed password attempts (TOB-KMW-15).	**Moderate**
Complexity Management	The Kraken mobile codebase is generally well organized, divided by functionality across various directories.	**Satisfactory**
Configuration	We found some minor configuration issues in the codebase, but they are not indicative of a recurring pattern that affects security. The SafetyNet Verify and Play Integrity APIs would enhance the security of Kraken users' wallets, but they are not implemented (TOB-KMW-19); we recommend following the Android documentation for the best security practices.	**Satisfactory**
Cryptography	The Kraken mobile wallet uses modern cryptographic	**Satisfactory**

and Key Management	algorithms for important operations, such as deriving keys, encrypting, and signing. In addition, for each of these operations, Kraken relies on well-known cryptographic dependencies that are maintained (with the exception of the dependency indicated in TOB-KMW-4). Sensitive wallet keys can be protected with both biometric authentication and encryption with a cryptographic key derived from a password. Again, these keys are protected by modern cryptographic algorithms. However, we do note that these authentication protections are prone to bypass (TOB-KMW-8). Also, we found two cryptographic issues in the Harmony component: an issue in the Harmony proof-of-work scheme could allow an attacker to obtain API keys that will not expire (TOB-KMW-16), and the application reuses the same cryptographic key in two different contexts (TOB-KMW-17).	
Data Handling	Generally, Kraken takes the necessary precautions when validating most types of incoming data; our analysis did not reveal any issues that could enable typical injection attacks, such as cross-site scripting. We found minor issues related to the UI; addressing them could make the product more secure (TOB-KMW-12, TOB-KMW-13). Also, the UI needs a careful review and redesign to make its behavior less surprising to users (TOB-KMW-18). The distinction between data received from online services and data hard-coded in the application should be clarified. Currently, much of the information received or hard-coded is redundant—either it is not used at all or one data source would be enough for all functionalities. This systemic issue makes it harder to audit the application (TOB-KMW-24) and may confuse users (finding 7 in appendix C).	**Moderate**
Documentation	We had access to the threat model, but we did not have access to the internal documentation. The solution should have high-level documentation showing the way specific components interact with each other, the data they exchange, the way specific functionalities are implemented (like local authentication and fee	**Weak**

	estimation), and the security assumptions for security-related components (e.g., password storage). The code should contain more comprehensive docstrings for methods. Currently, only a few parts of the code have any documentation.	
Maintenance	We found that the application has outdated dependencies (TOB-KMW-21), but none of them contain vulnerabilities that could significantly impact the application. Ensure that there is a proper process for keeping dependencies up to date (e.g. using Dependabot).	**Moderate**
Memory Safety and Error Handling	We did not find any memory safety issues. The solution is based on React Native, so exposure to these vulnerabilities is limited. However, we did encounter prevalent cases in which the Kraken wallet could crash (TOB-KMW-7, TOB-KMW-23).	**Weak**
Testing and Verification	The solution lacks fuzz tests. Many issues that could cause mobile wallet application crashes could be caught at the functional and unit test level.	**Weak**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	The QR code scanner is configured to detect all code types	Configuration	**Informational**
2	Hard-coded Infura API key	Data Exposure	**Informational**
3	Use of unpinned third-party scripts and images in CI	Configuration	**Low**
4	react-native-argon2 is unmaintained	Patching	**Informational**
5	Missing certificate validation in electrum-client	Cryptography	**Medium**
6	Third-party applications can take and read screenshots of the Android client screen	Data Exposure	**Medium**
7	Users may accidentally break wallet initialization	Denial of Service	**Informational**
8	Local biometric and password authentication can be bypassed	Access Controls	**Medium**
9	Reauthentication not required for all sensitive actions	Access Controls	**Medium**
10	Password policy issues on extra password protection	Authentication	**Low**
11	Sensitive content exposed via Clipboard	Data Exposure	**Low**
12	Truncated message content when signing via WalletConnect	Data Validation	**Medium**

13	Removal of URL protocol when pairing with WalletConnect	Data Validation	**Low**
14	Exposure of misconfigured GCP API key	Configuration	**Low**
15	Absence of account lockout mechanism	Authentication	**Low**
16	Harmony proof of work allows attacker to tamper with expiration	Authentication	**Medium**
17	Harmony reuses the same HMAC key for signing proof-of-work challenges and image URLs	Cryptography	**Informational**
18	WalletConnect transaction confirmation screen may be suddenly switched	Timing	**Medium**
19	SafetyNet Verify Apps and Play Integrity APIs not implemented in the Android client	Configuration	**Informational**
20	No explicit verification of the Android security provider	Patching	**Informational**
21	Project dependencies are not monitored for vulnerabilities	Patching	**Informational**
22	Device-to-device backups are not disabled	Data Exposure	**Low**
23	Application crashes when SVG image is tapped twice in the NFT module	Error Reporting	**Low**
24	Fee amounts are not displayed and can be controlled by remote services	Data Validation	**Medium**

Detailed Findings

1. The QR code scanner is configured to detect all code types	
Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-KMW-1
Target: `mobile/src/components/Camera.tsx`	

Description

The QR code scanner within the application uses the `expo-barcode-scanner` library to scan user-provided QR codes. However, the library can scan different types of barcodes, and the wallet application does not reconfigure the library to scan only QR codes. As a result, it may scan a barcode format that was not intended for the wallet. This increases the attack surface of the system and may lead to other issues.

Exploit Scenario

An attacker identifies a vulnerability in the underlying implementation of the Aztec code detector and decoder. She uses this vulnerability to exploit the mobile wallet application by crafting a malicious code and scanning it with the wallet application.

Recommendations

Short term, disallow the detection of all barcode types except QR codes in the `expo-barcode-scanner` library. Review the `BarCodeScanner` documentation for how to configure allowed types.

Long term, add a functional, integration, or device test that would ensure that the application does not scan barcode types other than QR codes, such as with the example images from the ZBar repository and malicious barcodes from this website.

2. Hard-coded Infura API key

Severity: **Informational**	Difficulty: **Low**
Type: Data Exposure	Finding ID: TOB-KMW-2
Target: `mobile/config.ts`	

Description

The application repository contains a hard-coded Infura project ID. This API key allows the Infura services to be queried and may be subject to rate limits. An attacker may extract this key from the application and use it for other purposes or cause denial of service through rate limit exhaustion.

```
export const INFURA_PROJECT_ID: string = 'REDACTED';
```

Figure 2.1: The API key is hard-coded in the codebase. (`mobile/config.ts`)

The severity of this finding is informational because this API key appears to be used only for debugging purposes.

Exploit Scenario

An attacker extracts the API key from the published application or public GitHub repository. They then proceed to use this key to request data from Infura, costing Kraken money and/or causing issues for developers using the debug features due to rate limit exhaustion.

Recommendations

Short term, remove the API key from the repository and rotate the secret. Inject the secret into the application only on debug-enabled builds via a configuration file or environment variable. Configure adequate rate limits on the API key to avoid unexpected billing.

Long term, integrate Trufflehog into the CI/CD system to detect unintended secrets flowing into the codebase.

3. Use of unpinned third-party scripts and images in CI

Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-KMW-3
Target: `mobile/.gitlab-ci.yml`, `mobile/.gitlab/ci/android-template.yml`, `mobile/.gitlab/ci/phrase.yml`, `mobile/.gitlab/ci/android-template.yml`	

Description

Several CI jobs reference third-party scripts, binaries, and container images without pinning them to known versions or otherwise checking for integrity. An attacker with access to these external servers, repositories, or registries may replace a script, binary, or container without being noticed and obtain the ability to execute code in the context of the CI jobs.

A set of examples where we found this pattern is included in figures 3.1 through 3.6.

```
before_script:
  - wget -O phrase_linux_amd64.tar.gz
https://github.com/phrase/phrase-cli/releases/download/2.5.1/phrase_linux_amd64.tar.
gz
  - tar -xzvf phrase_linux_amd64.tar.gz && cp phrase_linux_amd64 /usr/bin/phrase
```

Figure 3.1: A third-party binary is downloaded and installed without checking for integrity. (`mobile/.gitlab/ci/phrase.yml`)

```
script:
  - wget -O phrase_linux_amd64.tar.gz
https://github.com/phrase/phrase-cli/releases/download/2.5.1/phrase_linux_amd64.tar.
gz
  - tar -xzvf phrase_linux_amd64.tar.gz && cp phrase_linux_amd64 /usr/bin/phrase
```

Figure 3.2: A third-party binary is downloaded and installed without checking for integrity. (`mobile/.gitlab/ci/phrase.yml`)

```
variables:
  ANDROID_DOCKER_IMAGE: 'reg-kakarot.chorse.space/kakarot/web3-wallet/mobile:latest'
```

Figure 3.3: An unpinned container image is used in the workflow. (`mobile/.gitlab/ci/android-template.yml`)

```
  - export MAESTRO_VERSION=$MAESTRO_VERSION; curl -Ls
"https://get.maestro.mobile.dev" | bash
```

Figure 3.4: A third-party script is directly piped into the shell. (`mobile/.gitlab-ci.yml`)

```
# install nodejs and yarn packages from nodesource
RUN curl -sL https://deb.nodesource.com/setup_${NODE_VERSION} | bash - \
    && apt-get update -qq \
    && apt-get install -qq -y --no-install-recommends nodejs \
    && npm i -g yarn \
    && rm -rf /var/lib/apt/lists/*
```

Figure 3.5: A third-party script is directly piped into the shell.
(`mobile/docker/react-native-android/Dockerfile`)

```
RUN curl -sS https://dl.google.com/android/repository/${SDK_VERSION} -o /tmp/sdk.zip
\
    && mkdir -p ${ANDROID_HOME}/cmdline-tools \
    && unzip -q -d ${ANDROID_HOME}/cmdline-tools /tmp/sdk.zip \
    && mv ${ANDROID_HOME}/cmdline-tools/cmdline-tools
${ANDROID_HOME}/cmdline-tools/latest \
```

Figure 3.6: A third-party script is directly piped into the shell.
(`mobile/docker/react-native-android/Dockerfile`)

Exploit Scenario

An attacker gains access to the `phrase/phrase-cli` repository and replaces the 2.5.1 release with one containing malicious binaries. When the CI job is executed, the attacker obtains the ability to execute code in the CI context and exfiltrates any available CI secrets or code.

Recommendations

Short term, pin third-party scripts, binaries, and container images to specific versions using hash references when possible. Verify the checksum of downloaded binaries and scripts to ensure they have not been tampered with.

Long term, review the different build and deployment processes across the platform to ensure they are not vulnerable to supply chain attacks.

4. react-native-argon2 is unmaintained

Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-KMW-4
Target: `mobile/package.json`	

Description

The Kraken mobile wallet uses the Argon2 password-based key derivation function to create API keys. Specifically, users are required to solve a random challenge with Argon2 in order to be granted an API key. This serves as a proof of work that rate limits users accessing the API.

To implement the logic for this, the mobile wallet relies on `react-native-argon2`. This library appears to be unmaintained, as there have not been any commits in over two years, and there are multiple unresolved issues. This library is essentially a wrapper around two different native Argon2 implementations (one for iOS and one for Android). Fortunately, the native Android library, `argon2kt`, appears to be actively maintained. However, the iOS implementation, `CatCrypto`, is not.

Given the nature of these libraries, we do not see this as a significant risk to the system. However, we wanted to document the status of this library in the event that bugs or other flaws are discovered in either `react-native-argon2` or its native dependencies in the future. If this occurs, resolving the issues could be difficult without support from the maintainers of both `react-native-argon2` and `CatCrypto`.

Unfortunately, there do not appear to be any better Argon2 React Native alternatives. If significant issues arise with `react-native-argon2`, the best alternative appears to be directly relying on better maintained native libraries, such as `argon2kt` for Android and `Argon2Swift` for iOS.

Exploit Scenario

A significant flaw is discovered in either `react-native-argon2` or its iOS dependency, `CatCrypto`. Since neither of these libraries is actively maintained, resolving the issue consumes significant developer time.

Recommendations

Short term, monitor both `react-native-argon2` and `CatCrypto` for security issues or other implementation flaws that may be discovered in the future.

Long term, if flaws are discovered in either `react-native-argon2` or `CatCrypto`, consider switching to native dependencies like `argon2kt` and `Argon2Swift`.

5. Missing certificate validation in electrum-client	
Severity: **Medium**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-KMW-5
Target: `electrum-client` dependency	

Description

The Kraken wallet uses the `electrum-client` package from `BlueWallet/rn-electrum-client` to connect to the ElectrumX server on the host `electrum.wallet.kraken.com` using TLS. However, the `electrum-client` package disables the client-side verification of the server certificate, allowing for server impersonation and person-in-the-middle attacks.

```
this._socket = this._tls.connect({ port: port, host: host, rejectUnauthorized: false
}, () => {
  console.log('TLS Connected to ', host, port);
  return callback();
});
```

Figure 5.1: The certificate validation is disabled.
(`rn-electrum-client/lib/TlsSocketWrapper.js#71–74`)

Exploit Scenario

An attacker runs an open WiFi hotspot in a coffee shop. The DNS server used in the hotspot serves a malicious “A” record, pointing `electrum.wallet.kraken.com` to an attacker-controlled ElectrumX server with a self-signed certificate. Alice, a Kraken mobile wallet user, connects to the public WiFi hotspot and opens her Kraken mobile wallet. The mobile wallet application connects to the attacker server. The attacker gains information about Alice’s cryptocurrency holdings and transactions.

Recommendations

Short term, work with the `rn-electrum-client` developers to re-enable TLS certificate verification on the client side. Review the TLS client configuration to ensure it uses modern TLS protocol versions and ciphers.

Long term, perform regular dynamic testing with tools like Burp Suite and NoPE Proxy to ensure communications are adequately protected.

6. Third-party applications can take and read screenshots of the Android client screen	
Severity: **Medium**	Difficulty: **Medium**
Type: Data Exposure	Finding ID: TOB-KMW-6
Target: Kraken mobile wallet Android application	

Description

The `android.media.projection` API, introduced in Android 5.0, allows any third-party application on an Android device to take a screenshot of other running applications, including the Kraken mobile wallet. A third-party application can capture everything on the device's screen, including sensitive information such as mnemonics, and may continue recording the screen even after the user terminates the application (but not after the user reboots the device).

Enabling the `FLAG_SECURE` flag in the Kraken client will prevent third-party applications from taking screenshots of the Kraken mobile wallet.

Exploit Scenario

Alice prepares a malicious application, which Bob installs. Alice's application secretly records Bob's Kraken mobile application while he is looking at his wallet mnemonic. The malicious application exfiltrates the mnemonic, and Alice steals Bob's wallet.

Recommendations

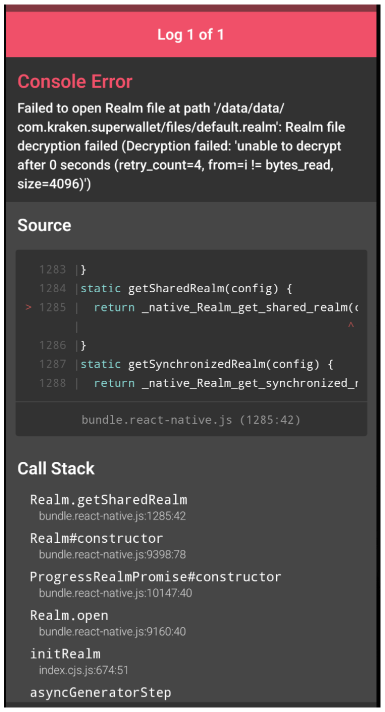
Short term, protect all sensitive windows within the Kraken Android application by enabling the `FLAG_SECURE` flag. This will prevent malicious third-party applications from recording the application and from taking screenshots of sensitive information. Also, the `FLAG_SECURE` flag will hide the Kraken application in the overview screen.

Long term, ensure that developer documentation is updated to include screen capture and recording as potential threats for data exposure.

7. Users may accidentally break wallet initialization	
Severity: **Informational**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-KMW-7
Target: Kraken mobile wallet Android application	

Description

If a user clicks the back button on their device during wallet creation, the application will close abruptly, resulting in incomplete wallet creation. However, some internal state is created during the initialization attempt. When the application is later restarted, it will throw a persistent error due to an inconsistent internal state. The only solution to overcome this error and continue with the wallet initialization is to reinstall the Kraken mobile wallet application, causing inconvenience to users.



```
Console Error
Failed to open Realm file at path '/data/data/
com.kraken.superwallet/files/default.realm': Realm file
decryption failed (Decryption failed: 'unable to decrypt
after 0 seconds (retry_count=4, from=i != bytes_read,
size=4096)')

Source
  1283 |}
  1284 |static getSharedRealm(config) {
> 1285 |  return _native_Realm_get_shared_realm(
       |                                       ^
  1286 |}
  1287 |static getSynchronizedRealm(config) {
  1288 |  return _native_Realm_get_synchronized_

      bundle.react-native.js (1285:42)

Call Stack
  Realm.getSharedRealm
    bundle.react-native.js:1285:42
  Realm#constructor
    bundle.react-native.js:9398:78
  ProgressRealmPromise#constructor
    bundle.react-native.js:10147:40
  Realm.open
    bundle.react-native.js:9160:40
  initRealm
    index.cjs.js:674:51
  asyncGeneratorStep
```

Figure 7.1: The error shown to users who click the back button during wallet creation

Recommendations

Short term, handle the scenario of application closure during wallet creation so that wallet initialization can be either resumed or completely rolled back without leaving any partial internal state.

Long term, expand the testing suite to systematically identify and mitigate issues related to a diverse range of potential user interactions, including unexpected application closures, to ensure the robustness and reliability of the Kraken mobile wallet.

8. Local biometric and password authentication can be bypassed	
Severity: **Medium**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-KMW-8
Target: Kraken mobile wallet application	

Description

The Kraken mobile wallet application provides two access control mechanisms: biometric and password authentication. They can be enabled and disabled independently of each other. Both mechanisms have weaknesses and provide less protection over data than possible.

The **biometric authentication** guards access to sensitive actions like signing and mnemonic display. There are two issues with it:

- The authentication is "event-based"— implemented as a Boolean check—instead of being "result-based"—binding data protection to a secure hardware (keychain or keystore). Therefore, the biometric authentication can be bypassed with dynamic instrumentation on rooted devices or through exploitation of operating system vulnerabilities.
- The wallet's lock screen can be easily bypassed when moving the application from the background to the foreground. A user just has to click the back button.

```
async function checkBiometrics(): Promise<boolean> {
  try {
    const { success } = await LocalAuthentication.authenticateAsync();
    return success;
  } catch (e) {
    console.log('Biometrics authentication failed: ', e);
    return false;
  }
}
```

Figure 8.1: The "event-based" local authentication implemented in the `checkBiometrics` function (`mobile/helpers/biometric-unlock.ts:13–21`)

The Kraken mobile wallet performs biometric authentication with the `checkBiometrics` function, which uses the `authenticateAsync` function from the Expo LocalAuthentication library. This library does not provide a mechanism to implement result-based authentication.

In the context of self-custody mobile wallets, result-based authentication should bind biometric authentication with users' confidential data via a secure hardware (keychain or keystore). That is, the wallet should encrypt users' data (private keys, mnemonics, etc.) using a secure hardware API. Then, the hardware should be used to decrypt the data on-demand (e.g., for transaction signing or viewing mnemonics), and the hardware should authorize decryption operations with biometric authentication.

On Android, result-based authentication can leverage the `CryptoObject` class to bind biometric authentication with cryptographic primitives. On iOS, a Keychain with a proper access control flag can be used.

The **password authentication** is used to double-encrypt data stored in the filesystem. The schema works as follows: most of the application's data is stored in an encrypted Realm (a MongoDB database). The Realm encryption key is randomly generated and stored in a secure hardware. Only the wallet application is authorized to use the key. The user-provided password is used to derive a new key, which is used to encrypt the Realm encryption key.

There are a number issues with this double-encryption schema:

- Data is encrypted only in the filesystem and not in the process's memory. After a wallet is opened by a user, all data (including the mnemonic) is kept in plaintext in RAM. The decryption key itself is also kept in memory (with calls to the `setRealmEncryptionKey` method).
- The plaintext data is kept in memory even after the application is moved to the background.
- The password does not protect access to sensitive actions. Once a wallet is unlocked with the password, all actions are accessible to a user (unless biometric authentication is also enabled)
- Not all data is encrypted; for example, the React Native's `AsyncStorage` is not encrypted. While no confidential information is kept there, important flags like `is_biometrics_enabled` are stored there in plaintext. Adversaries who are able to modify the filesystem can take actions such as disabling biometric authentication.

```
export async function getRealmEncryptionKey(encryptionPassword?: string):
Promise<Int8Array> {
  [skipped]

    credentials = await Keychain.getGenericPassword({ service: keychainServiceName
});

  [skipped]
  if (credentials) {
    password = credentials.password;
  } else {
    console.log('creating brand new realm encryption key...');
    const buf = crypto.randomBytes(64);
    password = buf.toString('hex');
    await Keychain.setGenericPassword(keychainServiceName, password, { service:
keychainServiceName, accessible: ACCESSIBLE.WHEN_UNLOCKED_THIS_DEVICE_ONLY });
    AsyncStorage.setItem(isRealmInAppEverInitialisedKey, 'true');
  }

  const buf = Buffer.from(password, 'hex');
  const ret = Int8Array.from(buf);

  if (encryptionPassword) {
    const deviceID: string = await DeviceInfo.getUniqueId();

    const decrypted = await decrypt(ret, encryptionPassword, deviceID);
    if (decrypted instanceof Int8Array) {
      return decrypted;
    } else {
      throw new Error('Failed to decrypt');
    }
  }

  return ret;
}
```

Figure 8.2: The double-encryption with the Realm encryption key from the keychain and the key derived from the user-provided password (`mobile/modules/encryptionKeyUtils.ts:19–65`)

The Kraken mobile wallet performs password authentication with the `getRealmEncryptionKey` method, which retrieves the Realm encryption key from the keychain and decrypts it with the `decrypt` method.

In the context of self-custody mobile wallets, the user-provided password could be used to protect confidential data (e.g., the mnemonic) both at-rest (data stored in the filesystem) and in the process's memory. The data should be decrypted only on-demand before a sensitive action.

Lastly, the local authentication renders the application nonfunctional when a user disables operating system-level authentication (the screen lock). The user has to reinstall the application to make it functional again.

Exploit Scenario

Alice installs and uses the Kraken wallet application on her mobile device. She moves the wallet to the background while using other applications. Then, she locks her device. Bob steals Alice's device. He exploits a local privilege escalation vulnerability in the device's operating system and gains root access to the device. He then replaces the `is_biometrics_enabled` flag that is stored in the plaintext `AsyncStorage` on the filesystem, moves the Kraken wallet application to the foreground, and displays the mnemonic.

Alternatively, instead of exploiting OS vulnerabilities, Bob does a forensic analysis of the device and extracts the content of the device's RAM. There, he finds the plaintext mnemonic.

Recommendations

Short term, rewrite the local authentication to protect confidential data both at-rest and when the application is open. Ensure that plaintext private keys and mnemonics are not kept in the wallet process's memory when not necessary. This security measure will limit the time window for forensic attacks. Finally, take the following actions:

1. Fix the ability to bypass biometric authentication using the back button.
2. Reimplement biometric authentication to be result-based.
3. Reimplement the password authentication to be required not only to open the wallet but also to perform sensitive actions (in addition to biometrics). Alternatively, at the very least, have data be encrypted after the wallet is moved to the background.
4. Either encrypt `AsyncStorage` or move security-relevant information from there to encrypted storage.
5. Decide how the wallet should behave if operating system–level authentication is disabled.

For item 2, have the wallet store users' private keys and mnemonics encrypted with the keychain's or keystore's key, and allow them to be decrypted only on-demand and with biometric authentication. Configure the wallet to require reauthorization before any action (instead of using time-based unlocking, for example). This can be done with the `setUserAuthenticationRequired` and `setUserAuthenticationParameters` methods on Android and with the `SecAccessControlCreateFlags` flags on iOS. Note that this proposed fix may require the currently used `react-native` Expo library to be replaced.

For item 3, separate non-confidential (e.g., addresses, wallet names) and confidential (e.g., private keys, mnemonics) data and have them encrypted with different keys. Have the non-confidential data decrypted once, when the wallet is opened, and have the confidential data decrypted only on-demand (e.g., when the user wants to sign a transaction or display a mnemonic). For example, the non-confidential data may be kept in the encrypted Realm (as it is now), but the confidential data should be protected by other means, as the Realm encryption does not offer a way to decrypt on-demand.

For item 4, review the data that is stored outside of the encrypted Realm and move all security-relevant information to the Realm. This information should include at least the `is_biometrics_enabled` flag (in the current implementation of the biometric authentication). Consider using the `NSFileProtectionCompleteUnlessOpen` protection level on iOS for additional encryption of at-rest data.

Note that both security controls—biometric and password—should work together. There are two approaches:

1. Require both biometric and password authentication for every sensitive action.
2. Require biometric authentication for sensitive actions and password authentication only for the initial unlocking of the application.

The first approach is preferable from a security perspective: even offline attacks against a device's secure hardware would be circumvented by the password protection. However, it may downgrade user experience.

For item 5, there are two scenarios involving the disabling of operating system–level authentication. Both start with the authentication enabled in both the operating system and the wallet:

- The user disables operating system–level authentication. The wallet can then take one of the following actions:
    - Automatically disable local authentication
    - Invalidate all of its keys and ask the user to initialize the application from scratch (import the wallet again or create a new one)
    - Lock its state, forbidding the user from opening the wallet and asking the user to re-enable the operating system–level authentication or reinitialize the wallet
- The user disables operating system–level authentication and then enables it, or the user changes the operating system–level PIN or adds a new fingerprint. The wallet can then take one of the following actions:
    - Invalidate all of its keys after any of these events
    - Keep functioning with the old keys

From a security point of view, we recommend having the wallet invalidate all of its keys and asking users for reinitialization in both scenarios. This can be achieved with the `InvalidatedByBiometricEnrollment` flag on Android and the `biometryCurrentSet` flag on iOS. Note that the recommended approach will require the user to recover the mnemonic whenever they change their device's PIN or add new fingerprints, which may be surprising to users and result in the loss of private keys. On the other hand, users are asked to back up their mnemonics, so they should be prepared for wallet reinitialization.

Depending on what approach the Kraken team chooses, the implementation of the recommendations in items 1–5 will differ.

Long term, do UI automation tests to verify that clicking the back button does not bypass authentication screens.

References

- OWASP: Local Authentication on Android and Local Authentication on iOS
- Leonard Eschenbaum: Bypassing Android Biometric Authentication
- Panagiotis Papaioannou: A closer look at the security of React Native biometric libraries

9. Reauthentication not required for all sensitive actions

Severity: **Medium**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-KMW-9
Target: Kraken mobile wallet application	

Description

All sensitive actions should require users to reauthenticate. Currently, the Kraken wallet application requires reauthentication for operations such as revealing the secret recovery phrase or making transactions, but the following operations do not require reauthentication:

- **Changing the wallet name:** An attacker with local access to a device could change the wallet name, confusing the user.
- **Removing all data:** An attacker with local access to a device could remove all Kraken application data, which could cause the user to lose their funds.
- **Confirming WalletConnect signing requests:** For example, the `personal_sign` method asks the wallet for a signature, and the user can generate it without biometric authorization.

Exploit Scenario

An attacker gains unauthorized access to a user's phone with the Kraken wallet open. The attacker uses the "Remove all data" feature within the application. If the user has forgotten his secret recovery phrase, he would not be able to restore the wallet, leading to a total loss of funds.

Recommendations

Short term, require users to reauthenticate before they can make sensitive changes to their accounts, such as changing wallet names and removing all data. This will prevent attackers with short-term access to a user's device from deleting the user's wallet. As users typically do not need to perform these sensitive actions regularly, reauthentication should not add much burden to the user experience. Consider removing the functionality to wipe all data after 10 unsuccessful attempts to unlock the wallet with a password. That functionality is especially dangerous if the wallet is not protected by the biometric authentication requirement.

Long term, review and document all operations and ensure that no sensitive operations are unprotected.

10. Password policy issues on extra password protection

Severity: **Low**	Difficulty: **High**
Type: Authentication	Finding ID: TOB-KMW-10
Target: Kraken mobile wallet application	

Description

A user can add extra security to his wallet by requiring password protection. However, the password policy requires that the password be at least six characters long. This is an imperfect requirement—according to NIST SP800-63B, passwords shorter than eight characters are considered to be weak. Also, the application does not verify (or at least warn users about using) passwords that are low entropy or are composed of a small set of characters (e.g., “aaaaaa”). The Kraken wallet does not defend users against rampant password reuse that would trivialize the effort required to decrypt wallets in the event of a back-end breach. Recent research indicates that 62% of users reuse passwords across multiple sites. Many of those reused passwords are likely to have been leaked by unrelated hacks, allowing credential stuffers to purchase those credentials and theoretically decrypt wallets at little expense.

Exploit Scenario

A user sets up a wallet and adds additional protection with the password 123456. An attacker gains unauthorized access to the user’s wallet and easily decrypts the Kraken wallet using a common password wordlist, which allows him to steal the user’s funds.

Recommendations

Short term, require that passwords be at least eight characters long. In the UI, implement a password strength meter to encourage users to set stronger passwords.

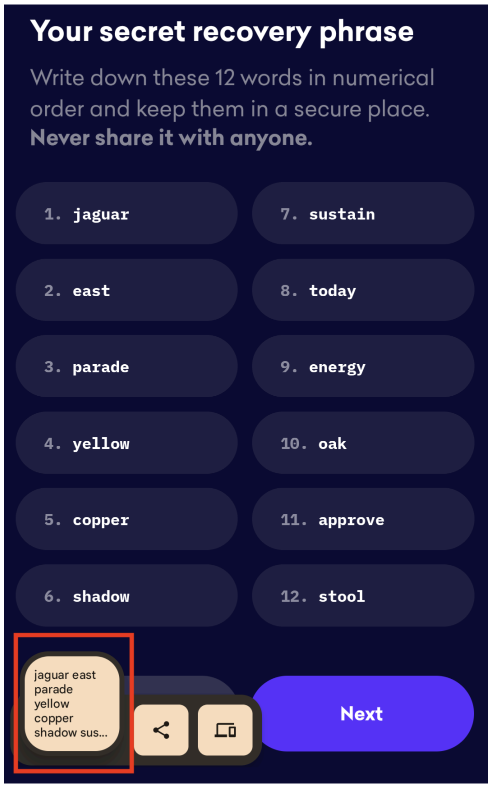
Long term, implement the Have I Been Pwned (HIBP) API in the wallet to check user passwords against publicly known passwords. If a password chosen by a user has been compromised, the wallet should inform the user and require the user to choose a new one.

11. Sensitive content exposed via Clipboard	
Severity: **Low**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-KMW-11
Target: Kraken mobile wallet Android application	

Description

Kraken's Android application exposes the secret recovery phase when it is copied to the clipboard (figure 11.1). With the introduction of Android 13, the system displays any text copied in a popover UI on the user's screen. This functionality can expose sensitive data if the text includes confidential content such as passwords or, in this case, a secret recovery phase.

To offset this vulnerability, Android 13 has introduced a new flag, `EXTRA_IS_SENSITIVE`, which, when applied to copied data, considers it sensitive and prevents its display on the user's screen.



Your secret recovery phrase

Write down these 12 words in numerical order and keep them in a secure place. **Never share it with anyone.**

1. jaguar	7. sustain
2. east	8. today
3. parade	9. energy
4. yellow	10. oak
5. copper	11. approve
6. shadow	12. stool

jaguar east parade yellow copper shadow sus...

Figure 11.1: Sensitive data exposed in an Android popover UI

Exploit Scenario

A user copies their secret recovery phase from the Kraken Android application to the clipboard. Because the copied data is not marked as sensitive, the secret recovery phase appears in the popover UI on the user's screen, exposing the secret recovery phase to potential adversaries.

Recommendations

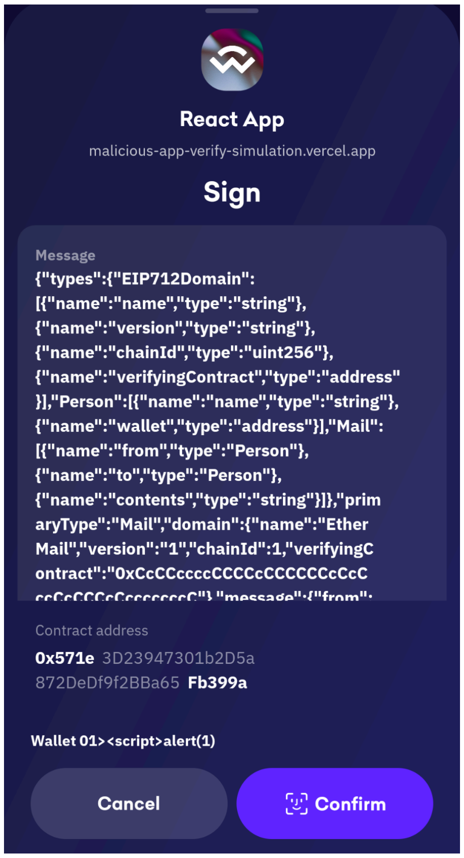
Short term, apply the `EXTRA_IS_SENSITIVE` flag to any sensitive data copied to the clipboard from the Kraken Android application. This will prevent the exposure of the data via the popover UI feature on Android 13.

Long term, conduct an extensive security review of the application to identify other potential ways through which sensitive data may be exposed, and document possible mitigations.

12. Truncated message content when signing via WalletConnect	
Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-KMW-12
Target: Kraken mobile wallet Android application	

Description

When a user tries to sign a message through WalletConnect on the Kraken Android application, the platform truncates the message, making it impossible for the user to read the entire content (figure 12.1).



React App

malicious-app-verify-simulation.vercel.app

Sign

Message

```
{"types":{"EIP712Domain":
[{"name":"name","type":"string"},
{"name":"version","type":"string"},
{"name":"chainId","type":"uint256"},
{"name":"verifyingContract","type":"address"
}],"Person":[{"name":"name","type":"string"},
{"name":"wallet","type":"address"}],"Mail":
[{"name":"from","type":"Person"},
{"name":"to","type":"Person"},
{"name":"contents","type":"string"}]},"prim
aryType":"Mail","domain":{"name":"Ether
Mail","version":"1","chainId":1,"verifyingC
ontract":"0xCcCCccccCCCCcCCCCCCcCcC
ccCcCCCcCcccccccC"},"message":{"from":
```

Contract address

0x571e 3D23947301b2D5a
872DeDf9f2BBa65 **Fb399a**

Wallet 01><script>alert(1)

Figure 12.1: The truncated view of the message when signing a message via WalletConnect

Additionally, the message may have been maliciously constructed in order to confuse the user, using extra keys and strings or superfluous whitespace in the JSON document. If users cannot review full messages, they may be misled into signing a risky payload.

Exploit Scenario
Bob, a user of Kraken's mobile wallet, attempts to sign a message for a transaction with WalletConnect. The message is truncated, preventing him from reading the full description of the transaction. As a result, he either rejects a legitimate transaction due to lack of clarity or approves a risky one because all details were not displayed.

Recommendations
Short term, ensure a full message is visible to users when attempting to sign a message through WalletConnect. Consider requiring users to scroll through the complete message before allowing them to sign it.

Long term, conduct a thorough review of the UI and the user experience elements relating to transaction signing.

References
- A UI Flaw in Top Crypto Wallets We Need to Address (a Coinspect article showing how a malicious payload can confuse users)

13. Removal of URL protocol when pairing with WalletConnect

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-KMW-13
Target: Kraken mobile wallet Android application	

Description

The Kraken Android application, when pairing with WalletConnect, uses the `removeProtocol` function (13.1) to display the paired application's URL without its protocol (figure 13.2). However, this absence of visibility of the protocol could mislead users into perceiving their connection as secure, regardless of whether it is actually a secure protocol (HTTPS) or a plain HTTP connection (figure 13.3).

```
export function removeProtocol(url: string): string {
  return url.replace(/(^[\w-]+:|^)\/\//, '');
}
```

Figure 13.1: The `removeProtocol` function responsible for the protocol truncation (`mobile/modules/text-utils.ts`)

```
<Label type="regularCaption1" color="light75" style={styles.url} numberOfLines={1}>
  {removeProtocol(removeWWWSubdomain(url))}
</Label>
```

Figure 13.2: The code responsible for displaying the URL when pairing with WalletConnect (`mobile/src/pages/ConnectApp/Header.tsx`)

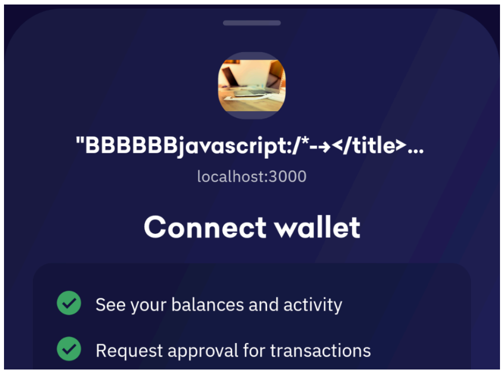

"BBBBBBjavascript:/*-→</title>...
localhost:3000
Connect wallet
See your balances and activity
Request approval for transactions

Figure 13.3: Pairing the Kraken application with WalletConnect hosted on `http://localhost:3000`

Exploit Scenario
Bob, a Kraken user, pairs his wallet with an application by using WalletConnect. He sees the URL as displayed by the Kraken application. However, the protocol from the URL has been removed, so Bob is unaware of whether he is connecting to a secure (HTTPS) or a non-secure (HTTP) site. Bob then proceeds with transactions on a potentially insecure platform, potentially exposing him to person-in-the-middle attacks.

Recommendations
Short term, modify the UI to include the application's URL protocol.

Long term, include explicit notifications highlighting the type of connection protocol being used, ensuring users are informed about their connection's security status. Consider allowing only HTTPS connections to ensure security standards are maintained.

14. Exposure of misconfigured GCP API key

Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-KMW-14
Target: Kraken mobile wallet Android application	

Description

The Google Cloud Platform API key is embedded in the Kraken mobile wallet code (figure 14.1), which makes it publicly accessible. A test of the API key endpoint responds with an HTTP 200 status code, denoting insufficient key restrictions (figure 14.2). This could result in unanticipated costs and changes to the application's quota.

```
<string name="google_api_key">AIzaSyDfANe88EOqd4HWj2orL2Zz7LfV-5WjVRo</string>
```

Figure 14.1: The part of the `res/values/strings.xml` file in the decompiled Kraken mobile APK

```
➜  ~ curl https://www.googleapis.com/discovery/v1/apis\?key\=
AIzaSyDfANe88EOqd4HWj2orL2Zz7LfV-5WjVRo
{
  "kind": "discovery#directoryList",
  "discoveryVersion": "v1",
  "items": [
    {
      "kind": "discovery#directoryItem",
      "id": "abusiveexperiencereport:v1",
      "name": "abusiveexperiencereport",
      "version": "v1",
      "title": "Abusive Experience Report API",
```

Figure 14.2: A test of the API key endpoint that gives an HTTP 200 response and JSON data, showing insufficient key restrictions

Recommendations

Short term, specify the mobile application that can use the key. For the Kraken mobile wallet on Android, set the application restriction to "Android apps" and add the application package name with the SHA-1 signing certificate fingerprint. For the Kraken mobile wallet on iOS, restrict the key to "iOS apps" and provide the appropriate bundle ID.

Long term, periodically review whether the application contains potentially sensitive API keys; if it does, ensure that these keys have configured secure restraints.

References

- Google Cloud: Authenticate by using API keys

15. Absence of account lockout mechanism	
Severity: **Low**	Difficulty: **High**
Type: Authentication	Finding ID: TOB-KMW-15
Target: Kraken mobile wallet Android application	

Description

The Kraken mobile wallet does not have an account lockout mechanism that is triggered after several consecutive failed password attempts. This means that an attacker can make an indefinite number of attempts to unlock a wallet, thereby opening a potential vector for brute-force attacks, especially on weaker passwords.

Exploit Scenario

John, a Kraken mobile wallet user, lost his mobile device without realizing it. An attacker finds the device and tries to gain access to John's Kraken wallet. Given that there is no lockout mechanism after several failed attempts, the attacker can continuously try different passwords. Depending on the complexity of John's password, the attacker eventually guesses it and gains unauthorized access to the wallet, allowing him to steal John's assets.

Recommendations

Short term, introduce an account lockout mechanism that temporarily disables an account or exponentially increases the delay between login attempts following a certain number of consecutive failed attempts. This would discourage brute-force attacks and add an extra layer of protection to user accounts.

When implementing the lockout, do not rely on the phone's clock for time comparisons. Use a source of time that returns the monotonic timestamp since the system booted. When a user reboots their device, the timestamp will return to zero because zero seconds have passed since the system booted. A timestamp that is more recent than the timestamp of the last failed password attempt indicates that the system has rebooted and that the stored timestamp can safely be updated to zero. This measure means that users will have to wait through the full lockout time again after a reboot, but it ensures that attackers cannot manipulate the time left on a lockout. For monotonic timestamps on Android, use the `elapsedRealtime` function, and on iOS, use the `clock_gettime` function with the `CLOCK_MONOTONIC` argument.

16. Harmony proof of work allows attacker to tamper with expiration

Severity: **Medium**	Difficulty: **Low**
Type: Authentication	Finding ID: TOB-KMW-16
Target: Kraken mobile wallet Android application	

Description

The Kraken mobile wallet has a proof-of-work scheme as an extra measure of defense for its API. Specifically, to obtain an API key, a user must complete a proof-of-work challenge that requires them to find the correct random solution value such that, when hashed with other values using Argon2, the result is below a predefined difficulty level. This is meant to make it more difficult for malicious users to obtain several API keys and access the API without limit.

A user who requests an API key is given a challenge for the proof-of-work scheme that is generated by the back end. This challenge consists of a random byte string, a (fixed) version number, a timestamp, and an expiration date for the API key. The user then searches by brute force for a random solution to the challenge. For each guess in the brute-force search, the user hashes the guess along with the challenge values and checks whether the output falls below the difficulty level. If it does, the user has successfully completed the proof-of-work challenge and can submit their solution to be verified.

Importantly, when generating a proof-of-work challenge, the back end signs the challenge using a cryptographic secret. When the solution is submitted, the back end checks whether there is a valid signature associated with the challenge to ensure the challenge was not generated by the user. Once the solution is verified, the user is granted the API key.

Harmony contains the logic for generating and signing challenges with an HMAC. There is also an important subtlety with how Harmony handles API key expiration. Unfortunately, as shown in figure 16.1, when Harmony signs its challenges, only the timestamp and random string are signed; the expiration date is not signed with an HMAC.

```
export function challengeToString(challenge: Challenge) {
  return `${challenge.v}.${challenge.ts}.${challenge.d}`;
}

export function createPowChallenge(secret: string): readonly [Challenge, string] {
  // Construct a challenge containing random bytes + the current timestamp.
  const salt = crypto.randomBytes(32);
  const timestamp = new Date().getTime();
  const challenge: Challenge = {
    v: 1,
    ts: timestamp,
    expiry: timestamp + 1000 * 60 * getPowKeyDuration(),
    d: salt.toString("hex"),
  };

  // Sign the challenge
  const challengeEncoded = challengeToString(challenge);

  // Calculate the hmac over the given payload
  const hmac = crypto.createHmac("sha256", secret);
  hmac.update(challengeEncoded);
  const signature = hmac.digest("hex");

  return [challenge, signature] as const;
}
```

Figure 16.1: Harmony does not include the expiration date in the HMAC.

The fact that the HMAC does not cover the expiration date means that an attacker could modify this data field without detection. As shown in figure 16.2, the back end implicitly trusts whichever expiration date is supplied with the submitted solution. Since this value is not signed with the HMAC, this means the back end could be trusting a malicious expiration date. An attacker could then change the expiration date to a time very far into the future so that they can then use their API key for an arbitrary amount of time.

```
@Route("pow/submit")
export class PowSubmitSolutionController extends Controller {
  /**
   * Submit a PoW challenge.
   */
  @Post()
  public async submitSolution(
    @Request() request: express.Request,
    @Body()
    body: {
      solution: string;
      challenge: Challenge;
      signature: string;
    }
  ): Promise<{ key: string }> {
    const secret = getPowPrivateKey();

    // Verify that the challenge is one we really issued.
    const challengeIsValid = verifyPowChallenge(body.challenge, body.signature,
secret);
    if (!challengeIsValid) {
      throw new BadRequestError({ message: "invalid pow challenge" });
    }
    // Verify that the solution satisfies the challenge
    const solutionIsValid = await verifyPowSolution(body.solution, body.challenge);
    if (!solutionIsValid) {
      throw new BadRequestError({ message: "invalid pow solution" });
    }

    // Note that we do not verify the timestamp of the challenge - the API
    // key we issue is bounded by that same timestamp itself.

    // If valid, then we bless the solution as a valid API key by singing it.
    // We also include the *original* timestamp, to make sure we can expire the
    // key. Note that we *need* to use the original timestamp from the challenge,
and
    // not the current system timestamp.
    //
    // The timestamp of the challenge was part of the data that the PoW was
calculated
    // against, so it cannot be modified without invalidating the proof.

    // If we used the current system timestamp, a client could submit the same
solution
    // multiple times to receive multiple API keys, each unique due to the included
    // timestamps being slightly different. And receiving a stable unique API key
for each
    // PoW solution that can be rate-limited against is the whole point of this
exercise.
    const key = await makeAPIKey(body.solution, body.challenge.expiry, secret);

    return {
      key,
    };
  }
}
```

Figure 16.2: Harmony trusts potentially modified expiration dates.

Exploit Scenario

An attacker notices that Harmony does not include the expiration date in their HMAC computation, so they modify the expiration date on their proof-of-work solution to a date much further in the future. Since the back end does not detect this modification, the attacker has a valid API key for a practically unlimited amount of time.

Recommendations

Short term, adjust the implementation of the Harmony proof of work so that the expiration date is included in the HMAC computation.

Long term, consider unifying the proof-of-work implementations between both Harmony and Ground Control. This could help reduce the likelihood of other future security issues by reducing the code's complexity and attack surface.

17. Harmony reuses the same HMAC key for signing proof-of-work challenges and image URLs

Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-KMW-17
Target: `harmony/bin/cf-image-worker.js`, `harmony/src/pow/config.ts`	

Description
The Kraken mobile wallet currently reuses the same cryptographic secret across two different contexts. In particular, the secret key stored in the `POW_PRIVATE_KEY` environment variable is currently used for generating HMACs for API proof-of-work challenges and for image URLs in `harmony/bin/cf-image-worker.js`.

In general, it is considered bad practice to reuse the same cryptographic key for two different purposes. The reason for this is that it increases the attack surface of the key and degrades the practical security of each context to the least secure of the two contexts. In other words, using the same key in two contexts is a weak design choice because if the key is compromised in one context, the other context is immediately compromised as well. Ideally, each context should have its own dedicated cryptographic key.

Exploit Scenario
An attacker discovers a flaw in the logic for creating proof-of-work API keys, causing the HMAC secret to be leaked and obtained by the attacker. Since this same key is also used for signing image URLs, this part of the system is immediately compromised as well.

Recommendations
Short term, adjust the Kraken mobile wallet so that HMACs for proof-of-work API key challenges and image URLs use separate cryptographic secrets.

Long term, moving forward, ensure that all cryptographic secrets are used for exclusively one purpose.

18. WalletConnect transaction confirmation screen may be suddenly switched	
Severity: **Medium**	Difficulty: **High**
Type: Timing	Finding ID: TOB-KMW-18
Target: Kraken mobile wallet application	

Description

The Kraken mobile wallet shows a popup dialog box when a dapp sends a request with the WalletConnect protocol. The dialog box shows the dapp's name and URL and the content of the request to the wallet's user, and asks the user for confirmation. When the user clicks the "Confirm" button, the request is approved, and the wallet computes a signature and sends it to the dapp. If another connected dapp sends its own request just before the user clicks the confirmation button, then the new request will be approved. As a result, the user would approve a request that they did not intend to.

Exploit Scenario

Alice connected her Kraken mobile wallet to two dapps. One of the dapps sends her an innocent request with the `personal_sign` method. Alice reviews the request and wants to approve it. When she is about to click the "Confirm" button, the other dapp sends a token transfer request with the `eth_sendTransaction` method. Alice approves the transaction instead of the message signature.

Recommendations

Short term, have the wallet push new requests to the bottom of the stack, instead of showing them on top of previous requests, and enable the confirmation button only after a few seconds so that users will not accidentally click it.

Long term, design the UI so that dialog boxes do not unexpectedly show up, elements do not move around without user interaction, and windows do not gain focus in unexpected moments. Sudden interface changes could lead users to perform other actions than intended. This issue impacts the user experience and may have security consequences.

References

- The missile warning system meme (based on the 2018 Hawaii false missile alert)

19. SafetyNet Verify Apps and Play Integrity APIs not implemented in the Android client	
Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-KMW-19
Target: Kraken mobile wallet Android application	

Description

The Kraken Android application does not use the SafetyNet Verify Apps API or the Play Integrity API.

Google Play provides the SafetyNet Verify Apps API to check whether potentially harmful applications are on a user's device. Through the Verify Apps feature, Google monitors and profiles the behavior of Android applications, informs users of potentially harmful applications, and encourages users to delete them. However, users are free to disable this feature and to ignore these warnings. The SafetyNet Verify Apps API can tell Kraken whether the Verify Apps feature is enabled and whether such applications remain on the user's device. This can provide an additional line of defense.

The Play Integrity API verifies that interactions and server requests come from the genuine application binary running on a real Android device. By detecting potentially risky and fraudulent interactions, such as from application versions that have been tampered with and untrusted environments, the application's back-end server can respond appropriately to prevent attacks and reduce abuse. The Play Integrity API is a continuation of the deprecated SafetyNet Attestation API.

Exploit Scenario

Bob has unknowingly installed a malicious application, which the Verify Apps feature detects. He ignores the warnings to uninstall the application because it includes a game that he enjoys. He also uses the Kraken application on the same device. The malicious application exploits an unpatched vulnerability in the Android system to extract the wallet keys from the phone's RAM. The malicious application also tricks Bob into transferring his assets to a third party via a tapjacking attack.

Recommendations

Short term, implement the SafetyNet Verify Apps API to require that the Verify Apps feature be enabled for all Kraken users and to ensure that known harmful applications are not installed on users' devices. Also, implement the Play Integrity API to ensure that the genuine application binary is run on a genuine Android device.

Long term, stay updated on new security features introduced in Android and continue adding relevant safety protections to the Kraken mobile applications.

References

- Android Developers: SafetyNet Verify Apps API
- Android Developers: Security guidelines

20. No explicit verification of the Android security provider

Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-KMW-20
Target: Kraken mobile wallet Android application	

Description

The Kraken Android application does not explicitly check whether it runs on a device with an up-to-date Android security provider.

A security provider is responsible for providing secure network communications, such as SSL/TLS. If the Kraken mobile wallet application is running on a device with an outdated security provider, it may be vulnerable to network attacks. For example, an attacker on the network could decrypt and compromise the wallet's TLS traffic.

Exploit Scenario

An attacker exploits a new vulnerability discovered in Android to perform a person-in-the-middle attack (similar to CVE-2014-0224). Alice has not upgraded her phone to include the latest version of the security provider to mitigate this vulnerability. The attacker is able to snoop and modify Alice's TLS traffic to Kraken's Harmony API server.

Recommendations

Short term, follow Google's guidance on patching the security provider by using the Google Play services `ProviderInstaller` class. For example, implement `ProviderInstaller.installIfNeeded()` to run when the application starts.

Long term, stay updated on new security features introduced in Android and iOS and continue adding relevant safety protections to the Kraken mobile applications.

References

- Android Developers: Update your security provider to protect against SSL exploits

21. Project dependencies are not monitored for vulnerabilities

Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-KMW-21
Target: Kraken mobile wallet application	

Description
The Kraken mobile wallet application is built with React Native and uses several third-party libraries to implement different functionalities. These third-party components may contain vulnerabilities, which can be discovered and fixed in newer releases. The wallet project currently uses older versions of these libraries, and there appears to be no set process to review new releases of dependencies, assess their impact, and apply upgrades and fixes.

Exploit Scenario
An attacker notices that the Kraken mobile wallet uses an older version of React Native, which includes a known vulnerable dependency. The attacker successfully exploits the vulnerability.

Recommendations
Short term, upgrade all dependencies in the Kraken project to their latest versions. For example, use the `yarn outdated` command to obtain a list of dependencies that are out of date, and use `yarn audit` to obtain a list of known vulnerable dependencies.

Long term, configure a process to get notified of new dependency releases in a timely manner. For instance, use Dependabot on GitHub to automatically open pull requests when new releases occur. Triage new dependency releases and upgrade dependencies if security fixes are provided.

22. Device-to-device backups are not disabled	
Severity: **Low**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-KMW-22
Target: Kraken mobile wallet Android application	

Description
The Kraken mobile wallet does not disable local device-to-device transfers. Encrypted local databases may be shared with other devices.

While the wallet disables backups to Google Drive with the `allowBackup` flag, the newer Android versions (Android 12/API level 31 and higher) do not disable device-to-device transfers with this flag.

Exploit Scenario
An adversary gains temporary physical access to a phone. He initiates a device-to-device transfer and copies the Kraken mobile wallet's encrypted data to his device. He then puts the phone back in place so that the victim does not notice the incident. The adversary performs an offline brute-force attack and obtains the user's private keys.

Exploit Scenario 2
A user copies all of his data to a new device with local device-to-device transfer. The old Kraken mobile wallet's encrypted databases are transferred. The user installs the wallet on the new phone. The wallet application fails to start because it cannot decrypt the copied databases, as the encryption master key stored in the device's keystore is new. The user gets angry, and Kraken's reputation is damaged.

Recommendations
Short term, disable device-to-device transfers. To do so, add the `android:dataExtractionRules` flag to the Android manifest pointing to a file with a `<device-transfer>` section. Add the `android:fullBackupContent` flag to support older API levels.

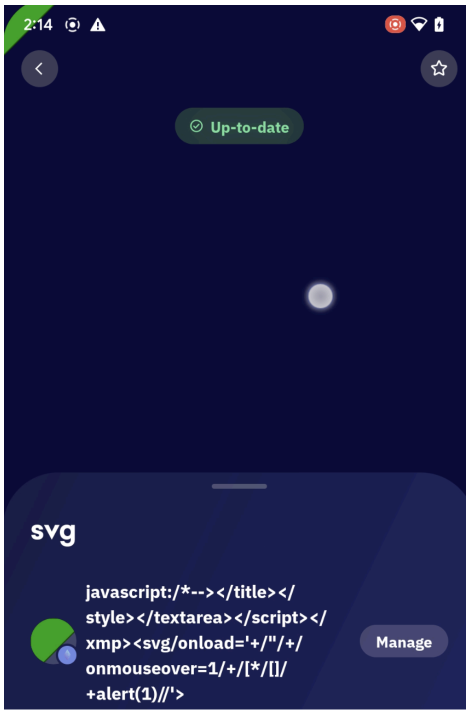
Long term, follow releases of new Android features and make sure that the Kraken mobile wallet application deals with them correctly.

References
- Android Developers: Backup and restore behavior changes for apps targeting Android 12

23. Application crashes when SVG image is tapped twice in the NFT module	
Severity: **Low**	Difficulty: **High**
Type: Error Reporting	Finding ID: TOB-KMW-23
Target: Kraken mobile wallet Android application	

Description

The Kraken mobile wallet application crashes when a user double-taps the SVG image present in the NFT module (figure 23.1). The crash is accompanied by a console error message that highlights problems with the key prop for each child in the list (figure 23.2).



javascript:/*--></title></style></textarea></script></xmp><svg/onload='+/"/+/onmouseover=1/+/[*/[]/+alert(1)//'>

Figure 23.1: The preview of the SVG image in the NFT module

```
ERROR  Warning: Each child in a list should have a unique "key" prop.

Check the render method of `NftTraits`. See https://reactjs.org/link/warning-keys
for more information.
    at View
(http://localhost:8081/index.bundle//&platform=android&dev=true&minify=false&app=com
.kraken.superwallet&modulesOnly=false&runModule=true:46837:31)
    in NftTraits (created by ViewNft)
    in RCTView (created by View)
    in View (created by ScrollView)
    in RCTScrollView (created by ScrollView)
    in ScrollView (created by ScrollView)
    in ScrollView (created by AnimatedComponent(ScrollView))
    in AnimatedComponent(ScrollView)
    in Unknown (created by NativeViewGestureHandler)
    in NativeViewGestureHandler (created by BottomSheetScrollView)
    in RCTView (created by View)
```

Figure 23.2: The error returned when the SVG image is tapped twice and the application crashes

Exploit Scenario

John is a user exploring the NFT module in the Kraken mobile wallet application. He decides to double-tap the SVG image to take a closer look at a particular NFT item. However, instead of the expected result, the application unexpectedly crashes. This crash disrupts John's user experience and makes the application appear unstable and untrustworthy.

Recommendations

Short term, fix the underlying problem to prevent the application from crashing when users interact with the SVG image.

24. Fee amounts are not displayed and can be controlled by remote services	
Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-KMW-24
Target: Kraken mobile wallet application	

Description

The Kraken mobile wallet does not display specific fee amounts. The wallet shows only one of the “fast,” “slow,” or “medium” strings. Therefore, users cannot manually validate the specific number of coins that will be spent on a transaction. Moreover, some fees are computed based on data received from remote services. If a service reports malicious data, the user may pay an unexpectedly huge fee. For example, the Bitcoin fee is estimated in the `estimateFees` function based on data received from Electrum.

Due to time constraints, we were not able to verify whether this issue impacts all chains or only some of them.

Exploit Scenario

The Electrum endpoint becomes malicious or compromised or starts malfunctioning, causing it to start reporting inflated fees. Unaware users of the Kraken wallet sign Bitcoin transactions with very large fees, without the ability to manually detect that fact before signing. They lose a large amount of money and blame Kraken for the incident.

Recommendations

Short term, allow users to manually verify the exact fee amounts that they will pay for transactions. Allow them to review other properties of transactions like gas limits and gas prices. Review how fees and other data used to construct transactions are computed (for all of the chains). Make sure that a malicious remote endpoint cannot force fees to become huge or to manipulate any part of data that is used to construct transactions. For example, hard code some sane fee amounts and have the wallet check whether the amounts received from online services are close to those hard-coded amounts. If the computed fees are unexpected, have the wallet warn users and instruct them about the actions they should take to protect their assets (e.g., do not sign the affected transactions; check the expected fees and if the warning is valid, contact the Kraken team).

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories

Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	Description
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	Description
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Configuration	The configuration of system components in accordance with best practices
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Documentation	The presence of comprehensive and readable codebase documentation
Maintenance	The timely maintenance of system components to mitigate risk
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	Description
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.

Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

This appendix contains findings that do not have immediate or obvious security implications or that were discovered but not fully investigated due to time constraints.

1. **The `encryptRealmEncryptionKey` function does not prevent multiple encryptions on the Realm encryption key.** The `encryptRealmEncryptionKey` function takes a password, derives an encryption key, and then encrypts the Realm encryption key with it. This encrypted key can then be later decrypted by passing the encryption password to the `getRealmEncryptionKey` function. However, the `encryptRealmEncryptionKey` function does not prevent multiple encryptions on the Realm encryption key, whereas the `getRealmEncryptionKey` function allows only for the decryption of keys that have been encrypted once. Furthermore, the documentation does not state that `encryptRealmEncryptionKey` should not be called multiple times on the same key, as this would prevent users from accessing the key. Currently, the mobile wallet prevents multiple encryptions by checking whether `isStorageEncryptedKey` is `true` in the `AsyncStorage` in another location, so there is currently no risk of multiple encryptions. However, we recommend either documenting this risk with surrounding comments or adding an explicit check inside of `encryptRealmEncryptionKey` to ensure that this does not become an issue as the codebase evolves over time.

2. **The Harmony `validateAPIKey` function takes a private Ed25519 key as input, but it only needs to use the public key.** The `validateAPIKey` function verifies that an API key is valid by checking the signature associated with the key against the Harmony public key. Even though only the public key is required, the function takes the secret key as input and then derives the public key from this secret key. We recommend adjusting this function to take the public key as input instead, as this could prevent the private key from being loaded into memory unnecessarily.

3. **The network security configuration on the Android application allows cleartext traffic to localhost and 10.0.2.2 on all builds.** Consider using `debug-overrides` instead or overriding the network security configuration with these changes only on debug builds.

```xml
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <domain-config cleartextTrafficPermitted="true">
            <domain includeSubdomains="true">10.0.2.2</domain>
            <domain includeSubdomains="true">localhost</domain>
   </domain-config>
</network-security-config>
```

Figure C.3.1: The network security configuration (`mobile/android/app/src/main/res/xml/network_security_config.xml`)

4. **The return value from the `Keychain.resetGenericPassword` method is not checked.** The `clearAppData` function may fail to reset the keychain item without notification. The issue does not have security implications because the `wipeEncryptionKey` method is called after every call to the `clearAppData` function.

```
export const clearAppData = async () => {
  await Keychain.resetGenericPassword({ service: keychainServiceName });
  await AsyncStorage.clear();
  WalletConnectSessionsManager.disconnectAllSessionsForAllAccounts();
  Realm.deleteFile({ schema: RealmSchema });
};
```

Figure C.4.1: The `clearAppData` method calls functions without proper error handling. (`mobile/src/utils/clearAppData.ts:8–13`)

5. **Simple string-manipulation functions are used to parse and manipulate URLs.** This is insecure. Functions specifically designed for parsing URLs should be used instead.

```
export function removeWWWSubdomain(url: string): string {
  return url.replace(/(?<=\/|^)www\./i, '');
}
```

Figure C.5.1: An example use of a string-manipulation function to manipulate URLs (`mobile/modules/text-utils.ts:90–92`)

```
function url2domain(url: string): string {
  return url.replace('http://', '').replace('https://', '').split('/')[0];
}
```

Figure C.5.2: Another use of insecure URL parsing (`mobile/modules/super-fetch.ts:37–39`)

6. **The `sanitizeValue` function is divergent from the `validateAmount` method.** The former function removes all non-digit characters except dots, while the latter accepts numbers in various formats, including hex strings. As a result, users are able to type in token amounts such as "0.1.2.3.4," which the `sanitizeValue` function would accept but which would be later rejected by the `validateAmount` method. On the other hand, users are not able to type in token amounts in hex format, even though the validation would accept them.

7. **There are multiple sources of token metadata.** Some metadata is hard-coded but is also received from remote services. For example, token decimals are hard-coded in the `evm.ts` file but also received from the `/api/data/v1/tokenMetadata` endpoint. The wallet uses different metadata

values for different tasks, which may lead to user confusion if one of the sources is divergent from the other.

8. **Sections in the About feature redirect to the same page.** Fix the About feature so that each section correctly redirects users to the intended information: the privacy policy, terms of service, licenses, or release notes.

9. **The language switching feature does not work.** Fix the feature so that it works as intended.

D. Automated Static Analysis

This appendix describes the setup of the automated analysis tools used during this audit.

Though static analysis tools frequently report false positives, they detect certain categories of issues, such as memory leaks, misspecified format strings, and the use of unsafe APIs, with essentially perfect precision. We recommend periodically running these static analysis tools and reviewing their findings.

Semgrep

To install Semgrep, we used `pip` by running `python3 -m pip install semgrep`.

To run Semgrep on the codebase, we ran the following commands in the root directory of the project (running multiple predefined rules simultaneously by providing multiple `--config` arguments):

```
semgrep --config "p/trailofbits" --config "p/ci" --config
“p/javascript” --config "p/security-audit" --config --metrics=off

semgrep --config auto
```

We recommend integrating Semgrep into the project’s CI/CD pipeline. To thoroughly understand the Semgrep tool, refer to the Trail of Bits Testing Handbook, where we aim to streamline the use of Semgrep and improve security testing effectiveness. Also, consider doing the following:

- Limit results to error severity only by using the `--severity` ERROR flag.
- Focus first on rules with high confidence and medium- or high-impact metadata.
- Use the SARIF format (by using the `--sarif` Semgrep argument) with the SARIF Viewer for Visual Studio Code extension. This will make it easier to review the analysis results and drill down into specific issues to understand their impact and severity.

CodeQL

We installed CodeQL by following CodeQL’s installation guide.

After installing CodeQL, we ran the following command to create the project database for the React Native repository:

```
codeql database create codeql.db --language=javascript
```

We then ran the following command to query the database:

```
codeql database analyze codeql.db -j 10 --format=csv
--output=codeql_tob.csv -- javascript-lgtm-full
javascript-security-and-quality javascript-security-experimental
```

We also used private Trail of Bits query packs.

E. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From December 18 to December 20, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Kraken team for issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 24 issues described in this report, Kraken has resolved 16 issues, has partially resolved one issue, and has not resolved the remaining seven issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	The QR code scanner is configured to detect all code types	Resolved
2	Hard-coded Infura API key	Resolved
3	Use of unpinned third-party scripts and images in CI	Resolved
4	react-native-argon2 is unmaintained	Unresolved
5	Missing certificate validation in electrum-client	Resolved
6	Third-party applications can take and read screenshots of the Android client screen	Resolved
7	Users may accidentally break wallet initialization	Resolved
8	Local biometric and password authentication can be bypassed	Resolved
9	Reauthentication not required for all sensitive actions	Partially Resolved

10	Password policy issues on extra password protection	**Resolved**
11	Sensitive content exposed via Clipboard	**Resolved**
12	Truncated message content when signing via WalletConnect	**Resolved**
13	Removal of URL protocol when pairing with WalletConnect	**Resolved**
14	Exposure of misconfigured GCP API key	**Resolved**
15	Absence of account lockout mechanism	**Resolved**
16	Harmony proof of work allows attacker to tamper with expiration	**Resolved**
17	Harmony reuses the same HMAC key for signing proof-of-work challenges and image URLs	**Resolved**
18	WalletConnect transaction confirmation screen may be suddenly switched	**Resolved**
19	SafetyNet Verify Apps and Play Integrity APIs not implemented in the Android client	**Unresolved**
20	No explicit verification of the Android security provider	**Resolved**
21	Project dependencies are not monitored for vulnerabilities	**Unresolved**
22	Device-to-device backups are not disabled	**Resolved**
23	Application crashes when SVG image is tapped twice in the NFT module	**Resolved**

24	Fee amounts are not displayed and can be controlled by remote services	Resolved

Detailed Fix Review Results

TOB-KMW-1: The QR code scanner is configured to detect all code types
Resolved. The QR code scanner no longer recognizes unpredictable code types.

TOB-KMW-2: Hard-coded Infura API key
Resolved. The `INFURA_PROJECT_ID` value is no longer exposed in the source code.

TOB-KMW-3: Use of unpinned third-party scripts and images in CI
Resolved. The Kraken team now verifies the checksum of downloaded files.

TOB-KMW-4: react-native-argon2 is unmaintained
Unresolved. The client provided the following context for this finding's fix status:

> *RISK ACCEPTED.*
> *Remediation Recommendation: Accepted*

TOB-KMW-5: Missing certificate validation in electrum-client
Resolved. The Kraken team confirmed that it uses `react-native-tcp-socket` `v5.6.2` with the `tlsCheckValidity` option.

TOB-KMW-6: Third-party applications can take and read screenshots of the Android client screen
Resolved. It is now impossible to capture the application screen from screenshots.

TOB-KMW-7: Users may accidentally break wallet initialization
Resolved. The application no longer throws a persistent error when wallet initialization is accidentally broken.

TOB-KMW-8: Local biometric and password authentication can be bypassed
Resolved. The finding was reviewed in a separate audit on the Kraken key management system.

TOB-KMW-9: Reauthentication not required for all sensitive actions
Partially resolved. The Kraken wallet application now requires reauthentication for all sensitive operations indicated in the finding except for one: changing the wallet name still does not require reauthentication.

TOB-KMW-10: Password policy issues on extra password protection
Resolved. The Kraken team implemented a password strength meter to show users the strength of the passwords they are entering; this could encourage users to input stronger passwords.

TOB-KMW-11: Sensitive content exposed via Clipboard
Resolved. The popover UI feature now shows dots instead of sensitive data.

TOB-KMW-12: Truncated message content when signing via WalletConnect
Resolved. The Kraken team implemented the ability to scroll through the message.

TOB-KMW-13: Removal of URL protocol when pairing with WalletConnect
Resolved. The Kraken team added the protocol to the domain URL.

TOB-KMW-14: Exposure of misconfigured GCP API key
Resolved. The GCP API key is now configured appropriately and returns a "permission denied" error.

TOB-KMW-15: Absence of account lockout mechanism
Resolved. The Kraken team added an account lockout mechanism that temporarily disables the ability to sign in to the application.

TOB-KMW-16: Harmony proof of work allows attacker to tamper with expiration
Resolved. Expiration is now checked correctly by Harmony, and the server returns an "invalid proof of work" error if the expiration is wrong.

TOB-KMW-17: Harmony reuses the same HMAC key for signing proof-of-work challenges and image URLs
Resolved. Harmony now uses separate cryptographic secrets for proof-of-work API key challenges and image URLs.

TOB-KMW-18: WalletConnect transaction confirmation screen may be suddenly switched
Resolved. The Kraken team extended the session request queue to include session proposals (not just session requests).

TOB-KMW-19: SafetyNet Verify Apps and Play Integrity APIs not implemented in the Android client
Unresolved. The client provided the following context for this finding's fix status:

> *RISK ACCEPTED*
> *Short term we will try to avoid adding google services as a required dependency for the app to work. So privacy sensitive users can use the app with their preferred OS.*

TOB-KMW-20: No explicit verification of the Android security provider
Resolved. The Kraken team added a check of whether the security provider is up to date on application startup.

TOB-KMW-21: Project dependencies are not monitored for vulnerabilities
Unresolved. Running the `yarn outdated` command in the application directory returns outdated dependencies, and the `yarn audit` command reports 75 vulnerabilities in dependencies.

TOB-KMW-22: Device-to-device backups are not disabled
Resolved. The Kraken team implemented appropriate settings in the Android manifest to disable backups.

TOB-KMW-23: Application crashes when SVG image is tapped twice in the NFT module
Resolved. The application now appropriately handles the SVG image in the NFT module.

TOB-KMW-24: Fee amounts are not displayed and can be controlled by remote services
Resolved. The Kraken team added the ability to see the amount of fees that will be spent on a transaction.

F. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	Description
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.