Uniswap Mobile Wallet

Security Assessment

December 12, 2022

Prepared for:
Padmini Pyapali
Uniswap

Prepared by: **Alexander Remie, Bo Henderson, Emilio López, and Tjaden Hess**

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 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

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

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

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

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

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

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

Notices and Remarks

Copyright and Distribution

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

About Trail of Bits 1

Notices and Remarks 2

Table of Contents 3

Executive Summary 5

Summary of Recommendations 7

Project Summary 8

Project Goals 9

Project Targets 10

Project Coverage 11

Automated Testing 13

Codebase Maturity Evaluation 14

Summary of Findings 16

Detailed Findings 18

1. iOS client is susceptible to URI scheme hijacking 18
2. The iOS client does not disable custom keyboards 19
3. Encrypted iCloud backups use low-entropy keys 20
4. Users are allowed to create unencrypted iCloud backups 22
5. Remote timing side channel in WalletConnect library 23
6. Use of libraries with known vulnerabilities 26
7. Sending funds to user-owned addresses can cause spurious warnings 28
8. WalletConnect v1 reuses cryptographic keys for multiple primitives 30
9. Credentials checked into source control 31

10. NFTs with SVG images are rendered as HTML 33
11. Application does not exclude keychain items from iCloud and iTunes backups 35
12. ShakeBugs may leak mnemonic 37
13. Use of improperly pinned GitHub Actions in Testflight build 38
A. Vulnerability Categories 40
B. Code Maturity Categories 42
C. Code Quality Recommendations 44
D. Automated Analysis Tool Configuration 46
D.1. Semgrep 46
D.2. CodeQL 46
D.3. Data Theorem 46
D.4. TruffleHog 47
E. KDF Recommendations 48

Executive Summary

Engagement Overview

Uniswap engaged Trail of Bits to review the security of its mobile wallet. From August 8 to August 19, 2022, a team of four consultants conducted a security review of the client-provided source code, with four person-weeks of effort. Details of the project's scope, timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the target system, including access to the source code. We performed static and dynamic testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The audit uncovered significant flaws that could impact system confidentiality, integrity, or availability. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	7
Medium	1
Low	1
Informational	3
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Configuration	1
Cryptography	4
Data Exposure	4
Data Validation	1
Error Reporting	1
Patching	1

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-UNIMOB-001**
  The `uniswap://` URI scheme can be hijacked because the application does not use Universal Links. This allows sensitive data to be extracted by another app installed on a user's iOS device that registers the same URI scheme.

- **TOB-UNIMOB-003**
  The encryption of a user's wallet file uses a low-entropy key that can be easily brute-forced. This allows an attacker who has gained access to a user's iCloud to decrypt the wallet file and gain access to the mnemonic.

- **TOB-UNIMOB-004**
  Users are given the option not to encrypt their wallet file. This presents a footgun to users and allows an attacker with iCloud access to steal the mnemonic.

- **TOB-UNIMOB-009**
  Several credentials are stored in the Git repo, including an Ethereum account that has real world value assets on several chains.

- **TOB-UNIMOB-011**
  Sensitive wallet information that is stored in the Keychain, such as the mnemonic, is not prohibited from being included in backups of iCloud or iTunes. This allows an attacker with access to iCloud or iTunes to access the sensitive wallet data.

- **TOB-UNIMOB-012**
  The views that display the mnemonic are not marked as "private." As such, screenshots taken for the ShakeBugs service can include a user's mnemonic.

Summary of Recommendations

The Uniswap mobile wallet is a work in progress with multiple planned iterations. Trail of Bits recommends that Uniswap address the findings detailed in this report and take the following additional steps prior to deployment:

- Design a secure production build process using GitHub Actions that mitigates supply-chain attacks issues currently present in the Testflight build (see TOB-UNIMOB-013).
- Write user-facing documentation of the mobile application.
- Monitor the upstream WalletConnect project for an update that mitigates TOB-UNIMOB-005 and update to the newest version when available.
- Improve the manual mnemonic backup flow by adding a validation step to confirm that the user recorded the words correctly and in the right order.
- Rework the `SeedPhraseInputScreen` component to avoid working with a mnemonic string in React Native code.
- Upgrade project dependencies, focusing on those with known issues, and introduce a system to keep them up to date, such as dependabot or a CI workflow.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineers were associated with this project:

Alexander Remie, Consultant
alexander.remie@trailofbits.com

Bo Henderson, Consultant
bo.henderson@trailofbits.com

Emilio López, Consultant
emilio.lopez@trailofbits.com

Tjaden Hess, Consultant
tjaden.hess@trailofbits.com

Project Timeline

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

Date	Event
August 8, 2022	Pre-project kickoff call
August 15, 2022	Status update meeting #1
August 22, 2022	Delivery of report draft; report readout meeting
September 8, 2022	Delivery of final report

Project Goals

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

- Is sensitive data ever sent to the back end?
- Does the application safely handle sensitive data (mnemonics and private keys)?
- Does the cryptography make use of the right cipher suites and parameters?
- Is the right cryptography used for the right use case?
- Does the project use dependencies with known vulnerabilities?
- Does the repository contain sensitive data?
- Are the iCloud and manual backup mechanisms implemented securely?
- Does the application take advantage of iOS security features?

Project Targets

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

Uniswap Mobile Wallet

Repository	https://github.com/Uniswap/mobile
Version	`ccad95b2284f64d93f73fd776a64b8382cfea680`
Type	React Native, Typescript, Swift
Platform	Mobile (iOS)

Project Coverage

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

- An automated review using CodeQL, Semgrep, and Data Theorem to find low-complexity issues, which resulted in several findings, including TOB-UNIMOB-001, TOB-UNIMOB-002, TOB-UNIMOB-010, TOB-UNIMOB-011, and the code quality recommendations in appendix C.
- A manual and automated review using TruffleHog to find credentials stored in the repository, which resulted in TOB-UNIMOB-009.
- A review of the iCloud backup encryption mechanism, which resulted in two findings: TOB-UNIMOB-003 and TOB-UNIMOB-004.
- A review of the use of cryptographic primitives and systems used to encrypt sensitive data, which resulted in two findings: TOB-UNIMOB-005 and TOB-UNIMOB-008.
- An automated review of the dependencies to find dependencies with known vulnerabilities, which resulted in one finding: TOB-UNIMOB-006.
- A review of the GitHub Actions pipeline, which resulted in one finding: TOB-UNIMOB-013.
- A review of the use of external services that could leak sensitive data, which resulted in one finding: TOB-UNIMOB-012.
- A review of the use of iOS security features.
- A review of the screens involved in handling the mnemonic.
- A review of the transaction signing mechanism.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- Platform support for non-iOS systems
- Project dependencies and libraries

- Back-end services with which the application communicates
- The cryptographic implementations in `ethers-rs`

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.

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.1
CodeQL	A code analysis engine developed by GitHub to automate security checks	Appendix D.2
Data Theorem: Mobile Secure	A tool that catches low-complexity issues in iOS and Android applications	Appendix D.3
TruffleHog	A tool for finding credentials in Git repositories and more	Appendix D.4

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
Auditing	Uniswap uses Sentry to monitor the application; however, we did not investigate whether Sentry's event logging and error reporting are sufficient.	**Further Investigation Required**
Authentication / Access Controls	We found several flaws related to authentication and access controls. Users are allowed to skip encryption of their wallet (TOB-UNIMOB-001), and wallets are encrypted using a low-entropy six-digit PIN (TOB-UNIMOB-003).	**Moderate**
Complexity Management	The codebase is generally well structured; code responsible for different parts of the application is clearly divided into discrete packages and files. There is little duplication of complex code across multiple areas.	**Satisfactory**
Configuration	Use of custom iOS keyboards is not disabled (TOB-UNIMOB-002). The GitHub Actions workflow used to build and publish the application do not use hashes to identify external GitHub actions to execute (TOB-UNIMOB-013). The app does not make use of Universal Links, thereby allowing the uniswap:// URL to be hijacked (TOB-UNIMOB-001).	**Moderate**
Cryptography and Key Management	We found weaknesses in the encryption of iCloud backups and configuration of Keychain storage for private keys (TOB-UNIMOB-003, TOB-UNIMOB-004, TOB-UNIMOB-011). We did not discover any issues with key generation, derivation or transaction signing.	**Moderate**

	Cryptographic issues in the upstream WalletConnect are not part of the Uniswap Wallet codebase under review and are not reflected in this maturity section.	
Data Handling	We identified an issue whereby HTML is not sanitized when displaying an NFT's SVG image (TOB-UNIMOB-010). We did not identify any other issues regarding the handling of data.	**Satisfactory**
Documentation	Most functions are missing comments that describe their purpose and behavior. There is no public documentation that describes the expected behavior of the application.	**Weak**
Maintenance	We identified many outdated packages with known security issues, which indicates that the package management policy or the checks performed in the CI/CD pipeline are ineffective (TOB-UNIMOB-006). We also found multiple credentials stored in the Git repository (TOB-UNIMOB-009).	**Moderate**
Memory Safety and Error Handling	We found one issue where multiple warnings are shown, which may confuse users (TOB-UNIMOB-007).	**Satisfactory**
Testing and Verification	The testing suite consists of e2e tests and per-feature unit tests. Most of the crucial features have unit tests. Uniswap makes use of Testflight to manually test builds of the app.	**Satisfactory**

Summary of Findings

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

ID	Title	Type	Severity
1	iOS client is susceptible to URI scheme hijacking	Configuration	High
2	The iOS client does not disable custom keyboards	Data Exposure	Low
3	Encrypted iCloud backups use low-entropy keys	Cryptography	High
4	Users are allowed to create unencrypted iCloud backups	Cryptography	High
5	Remote Timing Side Channel in WalletConnect Library	Cryptography	Medium
6	Use of libraries with known vulnerabilities	Patching	Undetermined
7	Sending funds to user-owned addresses can cause spurious warnings	Error Reporting	Informational
8	WalletConnect v1 reuses cryptographic keys for multiple primitives	Cryptography	Informational
9	Credentials checked into source control	Data Exposure	High
10	NFTs with SVG images are rendered as HTML	Data Validation	Informational
11	Application does not exclude keychain items from iCloud and iTunes backups	Data Exposure	High
12	ShakeBugs may leak mnemonic	Data Exposure	High

13	Use of improperly pinned GitHub Actions in Testflight build	Access Controls	**High**

Detailed Findings

1. iOS client is susceptible to URI scheme hijacking	
Severity: **High**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-UNIMOB-001
Target: `ios/Uniswap/Info.plist`	

Description

The Uniswap mobile app defines the `uniswap://` URI scheme for receiving messages from other apps on the device. URI schemes can be hijacked by another app if the malicious app registers the same scheme and is also installed on the device. Consequently, a rogue app could receive messages sent via URI schemes intended for the Uniswap mobile app.

```
<key>CFBundleURLName</key>
<string>uniswap</string>
<key>CFBundleURLSchemes</key>
<array>
	<string>uniswap</string>
</array>
```

Figure 1.1: `ios/Uniswap/Info.plist`

Exploit Scenario

The Uniswap mobile app makes use of its URI scheme to accept OAuth tokens or credentials sent via email or SMS. Alice creates a malicious app using the same `uniswap://` URI scheme and Bob installs it. When Bob receives his credential, Alice's app receives it instead of the Uniswap mobile app.

Recommendations

Short term, confirm that the `uniswap://` URI scheme is not used for sensitive messaging, and document the code to ensure that it never shall be.

Long term, transition to “Universal Links” introduced in iOS 9. These allow apps to register web domains that are solely owned by the app.

References

- Apple Developer Documentation: Support Universal Links

2. The iOS client does not disable custom keyboards	
Severity: **Low**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-UNIMOB-002
Target: Uniswap iOS application	

Description

The Uniswap mobile app client does not disable custom keyboards. Since iOS 8, users have been able to replace the system's default keyboard with custom keyboards that can be used in any application. Custom keyboards can—and very frequently do—log and exfiltrate the data that users enter.

Custom keyboards are not enabled when users type into "secure" fields (such as password fields). However, they could log all of a user's keystrokes in regular fields, such as those in which users type their personal information.

Exploit Scenario

Alice creates a custom keyboard that Bob uses. Alice's keyboard silently exfiltrates all of Bob's keystrokes in the Uniswap mobile app. When Bob imports his wallet mnemonic in the Uniswap application, Alice's keyboard exfiltrates the seed phrase.

Recommendations

Short term, disable third-party keyboards within the Uniswap mobile app to prevent leakage of sensitive data entered by the user. Third-party keyboards can be disabled by adding the `application:shouldAllowExtensionPointIdentifier:` method to the application client's `UIApplicationDelegate`.

Long term, track iOS platform changes and analyze data exfiltration opportunities. This will help the Uniswap team take steps to prevent data exposure risks for Uniswap mobile app users.

3. Encrypted iCloud backups use low-entropy keys	
Severity: **High**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-UNIMOB-003
Target: `src/features/CloudBackup`	

Description

During onboarding and new wallet creation, the iOS wallet application presents the user with two backup options: a manual backup of the wallet mnemonic and an iCloud backup. The iCloud backup may be optionally encrypted using a six-digit numeric pin. If the option to encrypt is selected, the wallet derives an AES-GCM encryption key from the pin via 310000 rounds of PBKDF2 with a random salt.

```
export const PIN_LENGTH = 6
```

Figure 3.1: `src/features/CloudBackup/cloudBackupSlice.ts#L11`

```
func encrypt(secret: String, password: String, salt: String) throws -> String {
  let key = try keyFromPassword(password: password, salt: salt)
  let secretData = secret.data(using: .utf8)!

  // Encrypt data into SealedBox, return as string
  let sealedBox = try AES.GCM.seal(secretData, using: key)
  let encryptedData = sealedBox.combined
  let encryptedSecret = encryptedData!.base64EncodedString()

  return encryptedSecret
}
```

Figure 3.2: `EncryptionHelper.swift#L18-L28`

If an attacker were to obtain access to the user's iCloud account and retrieve the encrypted wallet file, they would be able to easily decrypt the file due to the low entropy of the six-digit pin. While password-based key derivation functions provide a linear slowdown to attackers brute-forcing passwords, there are only one million six-digit pins. At an extremely conservative estimate of one CPU-second per password attempt, an attacker could brute-force the password in one million CPU-seconds; at the time of writing, this would cost about $650 on AWS's serverless compute service.

Because PBKDF2 is not GPU-resistant, an attacker could potentially try all of the possible passwords in minutes to hours on a single consumer GPU.

Exploit Scenario

An attacker obtains access to a user's iCloud account via a spear phishing campaign, a zero-day iCloud exploit, or exploitation of account recovery procedures. The attacker downloads the encrypted backup file and uses a GPU to try all possible pins in parallel, thus decrypting the wallet mnemonic and gaining full control of all user funds.

Recommendations

Short term, require full textual passwords for iCloud wallet backups.

Long term, implement password complexity estimation to warn users when they attempt to use a weak password; one example is react-password-strength-bar, which makes use of zxcvbn. Consider replacing PBKDF2 with a modern memory-hard password hash such as Argon2 or Scrypt.

4. Users are allowed to create unencrypted iCloud backups	
Severity: **High**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-UNIMOB-004
Target: Uniswap iOS wallet iCloud backups	

Description

During user onboarding and wallet creation, users are given the option to back up a copy of their mnemonic private key to iCloud. Users are prompted to enter a six-digit pin for encryption (see TOB-UNIMOB-003) with the option to skip encryption. Skipping encryption prompts a warning dialogue that notifies the user that their keys will be unprotected if their iCloud account is compromised.

The option to skip encryption, even with a warning, presents a footgun to users. iCloud accounts have a large attack surface and can be compromised (e.g., through phishing campaigns or zero-day server exploits).

Users may use the manual seed phrase backup to store their wallet as securely or insecurely as they choose, but the default cloud backup flow should be secure in all circumstances.

Exploit Scenario

An attacker carries out a phishing campaign to gain access to the user's iCloud account. The attacker can then immediately gain control of all of the user's assets using the unencrypted mnemonic.

Recommendations

Short term, remove the option for users to skip password creation during the iCloud backup flow.

Long term, require minimum password complexity standards for iCloud backups.

5. Remote timing side channel in WalletConnect library

Severity: **Medium**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-UNIMOB-005
Target: `WalletConnectSwift/Sources/Internal/AES_256_CBC_HMAC_SHA256_Codec.swift`	

Description

WalletConnect is a protocol for relaying messages between DApps and user Wallets. Setup is accomplished by sharing a QR code specifying a symmetric private key as well as a Bridge node responsible for relaying requests and maintaining pub/sub queues. The Bridge node is designed as an untrusted intermediary that blindly passes encrypted and authenticated messages between the Wallet and DApp.

The WalletConnect v1 Specification requires that clients communicate using AES-CBC for encryption along with HMAC-SHA256 to authenticate the data. When implementing authentication via HMAC, it is important that the time taken to compare the computed HMAC tag with the tag attached to the message does not depend on the content of the computed tag. If the standard string comparison function is used, the comparison will exit on the first mismatching byte and thus via a timing channel reveal how many leading bytes of the message MAC match the correct MAC for that message.

For example, the Swift implementation of the WalletConnect library compares the computed MAC with the payload MAC using the default "==" function:

```
let hmac = try authenticationCode(key: keyData, data: payload.data.data +
payload.iv.data)
guard hmac == payload.hmac.data else {
    throw CodecError.authenticationFailed(cipherText)
}
```

Figure 5.1: `Sources/Internal/AES_256_CBC_HMAC_SHA256_Codec.swift#L65`

Although this is an issue in the upstream implementation of the `WalletConnectSwift` dependency and not in the Uniswap mobile wallet implementation, the vulnerability affects the mobile wallet by potentially allowing a malicious WalletConnect bridge to forge message requests, including changing the token amounts in valid DApp requests and tampering with wallet responses to queries of balances and other blockchain states.

In order to exploit this side channel, the bridge needs to observe some timing-dependent behavior from the client. This can take many forms, including measuring the wallet response time to replayed messages.

This issue can be remediated in the WalletConnect library implementations via either of the following two methods:

1. Use a constant-time hash comparison routine from a cryptography library, such as https://developer.apple.com/documentation/cryptokit/hmac/3237468-isvalidauthenticationcode
2. Implement randomized double-HMAC blinding as described in https://paragonie.com/blog/2015/11/preventing-timing-attacks-on-string-comparison-with-double-hmac-strategy

For example, the Swift implementation in Figure 5.1 could be modified as in Figure 5.2:

```
let mask = [Int8](repeating: 0, count: 32)
let status = SecRandomCopyBytes(kSecRandomDefault, mask.count, &mask)
guard status == errSecSuccess else {
       throw CodecError.randomCopyFailed()
}
let hmac_inner = try authenticationCode(key: keyData, data: payload.data.data +
payload.iv.data)
let hmac_outer = try authenticationCode(key: keyData, data: hmac_inner)
let hmac_message = try authenticationCode(key: keyData, data: payload.hmac.data)
guard hmac_outer == hmac_message else {
    throw CodecError.authenticationFailed(cipherText)
}
```

Figure 5.2: Masked HMAC comparison example

Because this attack may be carried out against either party in the communication, the implementation must be patched in both the wallet and DApp.

Exploit Scenario

A malicious bridge node waits for a DApp to send a known transfer request. It intercepts the message and flips some bits in the message IV in order to change the transaction amount, recipient address or transaction fee. It replaces the HMAC tag with all zeros and forwards the request, which is immediately followed by a replayed valid request (e.g., an `eth_blockNumber` JSON-RPC request), and measures the time taken by the wallet before sending a response. It then increments the first byte of the HMAC tag and tries again. After trying all possible first bytes several times, the bridge deduces that the one with the longest response times is the correct first byte and then moves on to the second byte. Eventually, the bridge will be able to deduce all bytes of the correct HMAC code, and the forgery will be accepted by the wallet.

Recommendations

Because this vulnerability affects a live upstream library, we will be coordinating disclosure of this vulnerability. Please do not disclose the vulnerability before consulting with Trail of Bits. CVE-2022-38169 is reserved for this vulnerability.

Short term, update the WalletConnectSwift library version once a fix is made available.

Long term, consider using WalletConnect v2, which includes a more robust AEAD construction.

6. Use of libraries with known vulnerabilities

Severity: **Undetermined**	Difficulty: **Low**
Type: Patching	Finding ID: TOB-UNIMOB-006
Target: `package.json`	

Description

The codebase contains outdated dependencies affected by critical and high-risk vulnerabilities. We used `yarn audit` to detect a number of vulnerable packages that are referenced by the `yarn.lock` files. The most notable vulnerabilities are as follows:

- The `immer`, `minimist`, `simple-plist`, and `plist` dependencies contain critical vulnerabilities related to prototype pollution.
- The `shell-quote` package contains a critical vulnerability related to improper escapes that could allow arbitrary code execution (ACE) if the output from this package is passed to a shell.
- The `trim`, `terser`, `glob-parent`, `nth-check`, and `ansi-regex` dependencies are vulnerable to high-severity regular-expression denial-of-service (ReDoS) attacks.
- `node-fetch` and `follow-redirects` can expose sensitive data by leaking confidential HTTP headers while redirecting.
- `jpeg-js` is called in the `extractColors` utility function, potentially triggering an infinite loop leading to a denial of service (DoS).

Dependency	Vulnerability Type	Installed Version	Patched Version
`immer`	Prototype Pollution	8.0.1	9.0.6
`shell-quote`	ACE	1.6.1, 1.7.2	1.7.3
`hermes-engine`	Type Confusion	0.9.0	0.10.0
`minimist`	Prototype Pollution	1.2.5	1.2.6
`simple-plist`	Prototype Pollution	1.1.1	1.3.1
`plist`	Prototype Pollution	3.0.4	3.0.5
`trim`	ReDoS	0.0.1	0.0.3
`terser`	ReDoS	4.8.0	5.14.2

`glob-parent`	ReDoS	3.1.0	5.1.2
`node-fetch`	Data Exposure	1.7.3, 2.6.0, 2.6.1, 2.6.5	2.6.7
`nth-check`	ReDoS	1.0.2	2.0.1
`ansi-regex`	ReDoS	3.0.0, 4.1.0	3.0.1
`prismjs`	XSS	1.26.0	1.27.0
`follow-redirects`	Data Exposure	1.14.5	1.14.7
`async`	Prototype Pollution	2.6.3	2.6.4
`moment`	ReDoS, Path Traversal	2.29.1	2.29.4
`jpeg-js`	DoS	0.4.2	0.4.4

Figure 6.1: Dependencies with known critical- and high-severity vulnerabilities

Recommendations
Short term, update the build process dependencies to their latest versions wherever possible. Use tools such as `retire.js` and `yarn audit` to confirm that no vulnerable dependencies remain.

Long term, implement automated dependency auditing as part of the development workflow and CI/CD pipeline. Do not allow builds to continue with any outdated and vulnerable dependencies.

7. Sending funds to user-owned addresses can cause spurious warnings

Severity: **Informational**	Difficulty: **Low**
Type: Error Reporting	Finding ID: TOB-UNIMOB-007
Target: Uniswap iOS Wallet token send modal	

Description

When sending tokens to an account, the mobile wallet checks to see whether the user has previously interacted with the recipient address. If not, the user is warned that they are interacting with a new address and are advised to double-check that the address was input as desired. In many cases, this is a desirable warning and may prevent the user from accidentally sending funds to an incorrect address or on the wrong chain.

However, when a user creates a new account in their wallet and then attempts to transfer from an existing account to the new address, they are presented with and must dismiss two consecutive warnings, shown in figure 7.1.

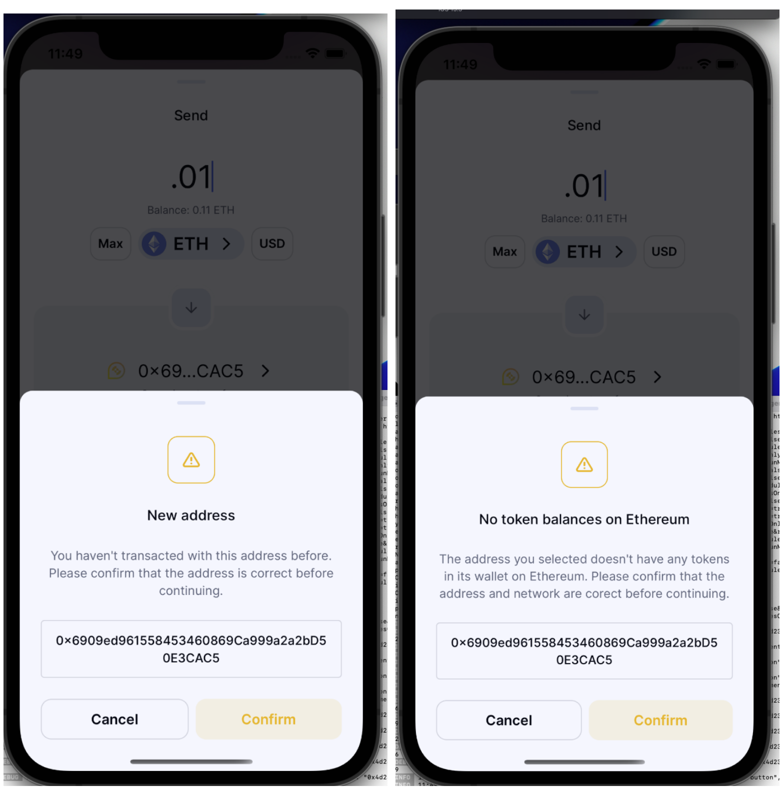

Figure 7.1: New address warning modals

Sending funds to an account for which the user holds the private key is generally safe. Presenting the user with many warnings can lead to “alert fatigue” and encourage the user to dismiss future warnings without consideration. Warnings should be presented only when there is a real risk that needs the user’s attention.

Exploit Scenario
A user becomes accustomed to dismissing the warnings during the send flow due to spurious warnings. They later mistype an address or mis-select which chain to send on, but ignore the warning and lose funds.

Recommendations
Short term, remove these warnings for addresses that are owned by the user.

Long term, analyze all warnings to the user and ensure that they are prioritized appropriately to minimize alert fatigue.

8. WalletConnect v1 reuses cryptographic keys for multiple primitives

Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-UNIMOB-008
Target: WalletConnect v1 Protocol Specification	

Description

The WalletConnect v1 Specification requires that clients communicate using AES-CBC for encryption along with HMAC-SHA256 to authenticate the data. Both the AES-CBC ciphertext and the HMAC-SHA256 tag are computed using a single shared symmetric key. Although there are no known negative interactions between AES-CBC and HMAC-SHA256, it is best practice to not use the same key for two different cryptographic primitives.

This vulnerability affects the WalletConnect v1 specification, not the Uniswap mobile client usage of the protocol.

Exploit Scenario

Future cryptanalysis allows an unexpected advantage to attackers against either AES-CBC or HMAC-SHA256 given the output of the other primitive.

Recommendations

Long term, consider moving to the WalletConnect v2 protocol, which uses a more robust cryptographic design.

9. Credentials checked into source control

Severity: **High**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-UNIMOB-009
Target: Mobile wallet Git history	

Description

The Uniswap mobile repository has several sensitive credentials checked into source control.

Most notably, the `src/features/wallet/accounts/useTestAccount.ts` file contains a hard-coded mnemonic that controls real world value across several chains; leaking this mnemonic would result in the loss of funds.

```
const MNEMONIC_TEST_ONLY = '[redacted]'
```

Figure 9.1: Mnemonic checked into source of `src/features/wallet/accounts/useTestAccount.ts#8`

Additionally, Covalent, Uniswap, Infura, OpenSea, and Zerion API keys are present in the project's `.env` file. These API keys are used to secure access to more relaxed app-wide rate-limits. If these credentials were leaked, an attacker could exhaust the app's resource allocation, leading to a DoS for all users.

```
COVALENT_API_KEY=ckey_524f68db6e0e43788ba7849da43
COINGECKO_API_URL=https://api.coingecko.com/api/v3/
DEBUG=true
UNISWAP_API_URL=https://dsn76qqamreobir5plkvauf4dm.appsync-api.us-east-2.amazonaws.c
om/graphql
UNISWAP_API_KEY=da2-n65b7e42gzgxzbye4f366ukss4
INFURA_PROJECT_ID=c92fab7a5b7841ecb65f26517b129364
LOG_BUFFER_SIZE=100
ONESIGNAL_APP_ID=5b27c29c-281e-4cc4-8659-a351d97088b0
OPENSEA_API_KEY=d0a4ff8d922e41e29454b86e0426d0f6
SENTRY_DSN=https://a216a11da7354acc9504a688813ff0bf@o1037921.ingest.sentry.io/600606
1
VERSION=0.0.1
ZERION_API_KEY=Demo.ukEVQp6L5vfgxcz4sBke7XvS873GMYHy
```

Figure 9.2: Various API keys checked into source of `.env#6-17`

Lastly, there is a hard-coded Alchemy API key in the project's `hardhat.config.js` file. Although this API key is not used in the app, its exposure could disrupt the development and deployment processes.

```
const mainnetFork = {
  url: 'https://eth-mainnet.alchemyapi.io/v2/lhVWQ3rY2i5_OZtYkU4Lzg_OsDT97Eoz',
  blockNumber: 13582625,
}
```

Figure 9.3: Alchemy API key checked into source of `hardhat.config.js#1-4`

If attackers gain access to the source code of the application, they will have access to these secrets. Additionally, all employees and contractors with access to the repository have access to the above secrets. These secrets should never be kept in plaintext in source code repositories, as they can become valuable tools to attackers if the repository is compromised.

Exploit Scenario

An attacker obtains a copy of the source code from a former employee. He extracts the secrets and uses them to sweep funds from Uniswap test accounts.

Recommendations

Short term, remove hard-coded secrets from source and rotate values. To fully remove secrets from the repository's history, read GitHub's documentation on removing sensitive data from a repository.

Long term, consider storing these secrets in a secret management solution such as Vault.

10. NFTs with SVG images are rendered as HTML

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-UNIMOB-010
Target: `src/components/images/WebSvgUri.tsx`	

Description

The code in charge of displaying NFTs that contain an SVG image does not sanitize the vector image contents, and it renders the image by embedding the image contents in an HTML document. An attacker can serve arbitrary HTML on the URI associated with the token and cause the wallet to render it on a WebView.

```
const getHTML = (svgContent: string) => `
<html>
  <head>
  <meta name="viewport" content="width=device-width, initial-scale=1.0,
maximum-scale=1.0, user-scalable=0, shrink-to-fit=no">
  <style>
      <!-- snip -->
    </style>
  </head>
  <body>
    ${svgContent}
  </body>
</html>
`
```

Figure 10.1: The SVG document is embedded in an HTML document without sanitization (`src/components/images/WebSvgUri.tsx#10–40`)

The WebView instance used to render SVG images disables JavaScript, so running arbitrary code is not possible. However, it also allows arbitrary origins to load, so it is possible to perform requests to external hosts (e.g., via an `iframe` tag) or navigate to other external sites (e.g., via a `meta` redirect tag).

```
<WebView
  scalesPageToFit
  javaScriptEnabled={false}
  originWhitelist={['*']}
  scrollEnabled={false}
  showsHorizontalScrollIndicator={false}
  showsVerticalScrollIndicator={false}
  source={{ html }}
```

```
  style={[
    webviewStyle.fullWidth,
    {
      aspectRatio,
      maxHeight,
    },
  ]}
  useWebKit={false}
/>
```

Figure 10.2: JavaScript is disabled, but a wildcard origin is allow-listed (`src/components/images/WebSvgUri.tsx#83–99`)

Exploit Scenario

An attacker hosts the SVG file from figure 10.3 on a public server, and mints an NFT referencing it. She then sends the NFT to an Uniswap Mobile Wallet user. When the user opens his wallet, the Trail of Bits site loads instead of displaying the green circle image.

```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="-1 -1 2 2">
  <circle fill="green" stroke="#eee" stroke-width=".1" r=".8"/>
  <foreignobject>
    <meta http-equiv="refresh" content="0;url=https://trailofbits.com" />
  </foreignobject>
</svg>
```

Figure 10.3: Example SVG file that will redirect to `trailofbits.com` when viewed in the wallet, but will otherwise show a green circle on an SVG viewer or browser

Recommendations

Short term, embed the images using a mechanism that will not result in arbitrary HTML being evaluated in a browser context, such as using an `img` tag to display the image. Alternatively, sanitize the SVG contents before including them as part of the HTML document, using a library such as DOMPurify.

Long term, consider externally provided content as untrusted and always sanitize it as necessary.

11. Application does not exclude keychain items from iCloud and iTunes backups

Severity: **High**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-UNIMOB-011
Target: `ios/RNEthersRS.swift`	

Description

The Uniswap mobile wallet does not prohibit its keychain items from being saved to an iTunes backup or uploaded to iCloud. Both Apple, Inc. and any attacker with access to a user's iTunes or iCloud backup will have access to that user's private data. Some of the private data stored in the keychain includes mnemonic phrases and private keys for accounts. Figure 11.1 gives an example use of `keychain.set` without the `withAccess` parameter. When omitted, the accessibility class defaults to `accessibleWhenUnlocked`. A second instance can be found on line 109 of the same file.

```
func storeNewPrivateKey(address: String, privateKey: String) {
  let newKey = keychainKeyForPrivateKey(address: address);
  keychain.set(privateKey, forKey: newKey)
}
```

Figure 11.1: `ios/RNEthersRS.swift#L141-L144`

Exploit Scenario #1

Alice gains physical access to Bob's phone and knows his passcode. She initiates a backup of Bob's phone to iTunes from which she extracts all of the wallet's sensitive keychain data, such as the wallet mnemonic phrase. She uses this information to steal tokens from Bob.

Exploit Scenario #2

Mallory collects email addresses from Uniswap mobile wallet users, then uses a previously disclosed password database to guess their current iCloud passwords. She retrieves iCloud backups that contain sensitive wallet keychain data from a large number of users, and steals their funds using the retrieved mnemonic phrases.

Recommendations

Short term, explicitly set a `ThisDeviceOnly` accessibility class (such as `AccessibleWhenUnlockedThisDeviceOnly`) for all keychain items by always calling `keychain.set` with `withAccess: AccessibleWhenUnlockedThisDeviceOnly`. This should prevent keychain data from being migrated to iTunes and iCloud backups.

Long term, empirically validate that no sensitive data is stored to a backup of the Uniswap mobile wallet. Consider uniform usage of a wrapper, such as Square's Valet, to simplify storage and retrieval of data from the keychain using a certain accessibility class.

References

- Apple Developer Documentation: Keychain Services
- Square Valet

12. ShakeBugs may leak mnemonic

Severity: **High**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-UNIMOB-012

Target:
- `src/screens/Import/SeedPhraseInputScreen`,
- `src/screens/Onboarding/ManualBackupScreen`,
- `src/screens/SettingsViewSeedPhraseScreen`,
- `src/screens/SettingsManualBackup`

Description

The Uniswap mobile wallet uses ShakeBugs to help with reporting of issues in the app. When a user reports an issue, ShakeBugs can take a screenshot and send it along with the reported issue. If this happens in a screen that shows a mnemonic, the user's mnemonic would be logged to ShakeBugs.

The ShakeBugs documentation provides information on how to mark certain views as "private," which redacts the private information from the screenshot.

Exploit Scenario

Alice reports an issue while in the screen that displays the mnemonic. A screenshot that shows the mnemonic is taken and sent along with the reported issue to ShakeBugs.

Recommendations

Short term, explicitly mark the mnemonic screens as "private" as explained in the ShakeBugs documentation.

Long term, review the usage of third-party services in the application. Consider all the data they may collect and how their use may impact the users' privacy and security.

References

- ShakeBugs documentation: marking view as private

13. Use of improperly pinned GitHub Actions in Testflight build	
Severity: **High**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-UNIMOB-013
Target: Mobile wallet GitHub actions	

Description

The GitHub Actions workflows for creating an iOS application build uses several third-party actions that are pinned to a tag or branch name instead of a full commit SHA. This configuration enables repository owners to silently modify the actions. A malicious actor could use this ability to tamper with an application release or leak secrets such as application signing keys.

```
    - name: Pod Install
      uses: nick-fields/retry@v2
      with:
        timeout_minutes: 20
        retry_wait_seconds: 2
        max_attempts: 3
        command: cd ios && pod install && cd ..

    - name: Build and ship iOS App
      run: |
        export PATH="/usr/lib/ccache:/usr/local/opt/ccache/libexec:$PATH"
        export CCACHE_SLOPPINESS=clang_index_store,file_stat_matches,include_file_ctime,include_file_mtime,ivfsoverlay,pch_defines,modules,system_headers,time_macros
        export CCACHE_FILECLONE=true
        export CCACHE_DEPEND=true
        export CCACHE_INODECACHE=true
        ccache -s
        set -o pipefail
        yarn deploy:ios:alpha
        ccache -s
      shell: bash
      env:
        APP_IDENTIFIER: ${{ secrets.APP_IDENTIFIER }}
        APPLE_ID: ${{ secrets.APPLE_ID }}
        APPLE_APP_ID: ${{ secrets.APPLE_APP_ID }}
        APPLE_TEAM_ID: ${{ secrets.APPLE_TEAM_ID }}
        URL_TO_FASTLANE_CERTIFICATES_REPO: ${{ secrets.URL_TO_FASTLANE_CERTIFICATES_REPO }}
        MATCH_PASSWORD: ${{ secrets.MATCH_PASSWORD }}
        FASTLANE_APPLE_APPLICATION_SPECIFIC_PASSWORD: ${{ secrets.FASTLANE_APPLE_APPLICATION_SPECIFIC_PASSWORD }}
```

```
        CI: true
        CI_KEYCHAIN_NAME: 'CI_KEYCHAIN'
        CI_KEYCHAIN_PASSWORD: ${{ secrets.CI_KEYCHAIN_PASSWORD }}
        GIT_BRANCH_NAME: ${{ github.ref }}
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

Figure 13.1: The `nick-fields/retry` action is pinned only to a tag and can access the GitHub token among other secrets (`.github/workflows/fastlane.yml#L78-L110`)

Exploit Scenario

An attacker gains access to the `nick-fields/retry` repository and modifies the action tagged v2. The modified action introduces malicious code into the codebase before the application build is complete. A Uniswap developer then pushes changes to the repository to release a new wallet version; this code push triggers the execution of a workflow that includes the malicious action. As a result, the build artifacts produced by the workflow contain a backdoor that cannot be detected in the source code of the repository. The Uniswap team then deploys the modified application to Testflight. When users use the test version of the app and import a mnemonic, the attacker could exfiltrate the mnemonic and steal the funds in it.

Recommendations

Short term, pin each third-party action to a specific full-length commit SHA, as recommended by GitHub.

Long term, review the platform build and deployment processes to ensure that they are protected against supply chain attacks.

A. Vulnerability Categories

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

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

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

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

B. Code Maturity Categories

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

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

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **Undocumented JSDoc parameters.** Several parameters are missing a description. Adding descriptions to all parameters is considered a best practice and helps readers understand the implementation.

```
/**
* Add opacity information to a hex color
* @param amount opacity value from 0 to 100
* @param hexColor
*/
```

Figure C.1: `src/utils/colors.tsx#L6-L10`

```
/**
* Returns a flat list of `Currency`s filtered by chainFilter and searchFilter
* @param currencies
* @param chainFilter chain id to keep
* @param searchFilter filter to apply to currency adddress and symbol
*/
```

Figure C.2: `src/components/CurrencySelector/filter.ts#L46-L51`

- **Code has no effect.** The `else` case in Figure C.3 has no effect, as it will return the `push` function without calling it. Judging by the implementation, the `accum` variable should not be assigned any values at index 4. Consider removing the `else` case.

```
  } else {
    accum[4].push
  }
```

Figure C.3: `src/features/transactions/hooks.ts#L185-L187`

- **Typos.** There are several typos in files. The “intial” typo in Figure C.5 is present multiple times throughout the file.

```
export interface RecieveNFTNotification extends TransferNFTNotificationBase {
  txType: TransactionType.Receive
  sender: Address
}
```

Figure C.4: `src/features/notifications/types.ts#L91-L94`

```
const intialSchemaStub = {
  ...initialSchema,
  transactions: {
```

Figure C.5: `src/app/migrations.test.ts#L182-L184`

D. Automated Analysis Tool Configuration

As part of this assessment, we performed automated testing on the Uniswap mobile wallet codebase and compiled binaries using four tools: Semgrep, CodeQL, Data Theorem, and TruffleHog.

D.1. Semgrep

We performed static analysis on the Uniswap mobile wallet source code using Semgrep to identify low-complexity weaknesses. We used several JavaScript and TypeScript rule sets, shown in figure D.1.1, which did not result in the identification of code quality issues or areas that may require further investigation.

```
semgrep --metrics=off --sarif --config="p/r2c"
semgrep --metrics=off --sarif --config="p/r2c-ci"
semgrep --metrics=off --sarif --config="p/r2c-security-audit"
semgrep --metrics=off --sarif --config="p/r2c-best-practices"
semgrep --metrics=off --sarif --config="p/eslint-plugin-security"
semgrep --metrics=off --sarif --config="p/javascript"
semgrep --metrics=off --sarif --config="p/typescript"
semgrep --metrics=off --sarif --config="p/clientside-js"
semgrep --metrics=off --sarif --config="p/react"
semgrep --metrics=off --sarif --config="p/owasp-top-ten"
semgrep --metrics=off --sarif --config="p/jwt"
semgrep --metrics=off --sarif --config="p/supply-chain"
```

Figure D.1.1: Commands used to run Semgrep

D.2. CodeQL

We used CodeQL to analyze the Uniswap mobile wallet codebase. We used our private `tob-javascript-all` query suite, which includes public JavaScript queries and some private queries. CodeQL found several code quality issues. If Uniswap intends to run CodeQL, we recommend reviewing CodeQL's licensing policies.

```
# Create the javascript database
codeql database create codeql.db --language=javascript

# Run all javascript queries
codeql database analyze codeql.db --format=sarif-latest --output=codeql_res.sarif --
tob-javascript-all.qls
```

Figure D.2.1: Commands used to run CodeQL

D.3. Data Theorem

We ran the Uniswap mobile wallet application binaries provided by the Uniswap team through the Data Theorem Mobile Secure tool. We determined the validity of the results and evaluated each result's impact on the system. The findings that could have an impact on the system, along with our recommendations for resolving these issues, are described in this report's Detailed Findings section.

D.4. TruffleHog

We ran TruffleHog on the Uniswap mobile wallet source repository. This run revealed that an Ethereum account with real world value assets and multiple API keys were committed and not correctly cleaned from the repository, as described in TOB-UNIMOB-009.

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

Figure D.4.1: Command used to run TruffleHog

E. KDF Recommendations

This section provides recommendations on the cryptographic algorithm to use for password-based key derivation in the Uniswap mobile application.

The recommended KDF is Argon2. Of the three possible variants, the Argon2id variant is preferred. This variant strikes a balance between side-channel resistance and GPU-resistance.

The Argon2Swift package offers a Swift wrapper around the reference C implementation. For determining the right parameters to use, RFC 9106 offers valuable guidance.

After reviewing the above source we recommend the following parameter values for a mobile application:

- Mode: argon2id
- Parallelism: 4
- Memory: 1GB
- Salt Length: 16 bytes
- Hash length: 32 bytes
- Iterations: This will determine the time it takes to perform the derivation. Run a benchmark where this value is increased until the resulting time is acceptable UX-wise. Aim for at least 0.5-1 seconds. If the time of the derivation already exceeds the maximum allowed time with just one iteration, lowering the memory can result in a lower derivation time.

Per RFC 9106, generic acceptable minimum parameters are as follows:

- Mode: argon2id
- Parallelism: 4
- Memory 64 MB
- Salt Length: 16 bytes
- Hash Length: 32 bytes
- Iterations: 3