Uniswap Wallet, Android

Security Assessment

November 2, 2023

Prepared for:
Tarik Bellamine
Uniswap

Prepared by: **Paweł Płatek and Maciej Domański**

About Trail of Bits

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

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

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

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

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

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

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

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

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

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

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

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

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

The following engineering director was associated with this project:

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

The following consultants were associated with this project:

Maciej Domański, Consultant
maciej.domanski@trailofbits.com

Paweł Płatek, Consultant
pawel.platek@trailofbits.com

Project Timeline

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

Date	Event
September 21, 2023	Pre-project kickoff call
October 2, 2023	Status update meeting #1
October 10, 2023	Delivery of report draft
October 10, 2023	Report readout meeting
October 18, 2023	Delivery of comprehensive report
November 2, 2023	Delivery of comprehensive report with fix review appendix

Executive Summary

Engagement Overview

Uniswap engaged Trail of Bits to review the security of its non-custodial wallet mobile application developed on the iOS and Android platforms. We conducted separate reviews for each version of the application. This report concerns the Android version.

A team of two consultants conducted the review from September 25 to October 6, 2023, for a total of four engineer-weeks of effort. Our testing efforts focused on the Android version of the application, WalletConnect integration, and deep linking. With full access to the source code and documentation, we performed static and dynamic testing of the mobile wallet, using automated and manual processes.

Observations and Impact

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

A key issue was identified in relation to the misconfiguration of the OneSignal SDK, potentially enabling an attacker to exploit debuggable WebViews (TOB-UNIMOB2-19). Additionally, it was discovered that a static nonce is used for AES-GCM encryption of cloud backups (TOB-UNIMOB2-4), but this issue is not directly exploitable, because the encryption keys are renewed each time.

Given the financial nature of the Uniswap mobile application, our audit also concentrated on assessing resistance against two major threats: the presence of malware and physical access to a user's smartphone by malicious actors. Regarding the former, we found a feasible scenario where a malicious application could capture screens displaying mnemonic phrases, thereby leading to a loss of funds (TOB-UNIMOB2-12). Regarding the latter, we discovered vulnerabilities enabling bypass of local biometric authentication (TOB-UNIMOB2-13), potential persistence of mnemonic phrases in memory (TOB-UNIMOB2-14), and evasion of lockdown following incorrect password entries (TOB-UNIMOB2-18).

Despite the convenience of multiplatform application development offered by React Native, it is critical for developers to scrutinize the final, platform-specific APK generated for Android. Several security concerns surfaced during this review, including overly broad permissions in the production release Android manifest (TOB-UNIMOB2-8); security-impaired exported activities (TOB-UNIMOB2-10); and methods that users and Google Play security measures may consider suspicious, such as extracting the list of running applications on a device (TOB-UNIMOB2-23) and accessing a device's phone number or SIM card details without explicit permission (TOB-UNIMOB2-21).

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.
- **Support multiple data providers on the wallet's back end (Uniswap service API).** Details of the Uniswap back-end service were not known during the audit, but some errors returned by the back end indicate that Uniswap uses Infura internally to connect to blockchains. To make the system more decentralized, consider hosting Uniswap-controlled RPC nodes instead. Alternatively, connect to multiple data providers and cross-verify the results from them.
- **Consider allowing wallet users to configure their own list of trusted nodes.** This will improve user trust in the mobile wallet application by reducing the number of third parties that users did not select but must trust implicitly. Currently, wallet users send some RPC requests via the Infura infrastructure and some via Uniswap's service.
- **Consider showing more data for manual validation in the "review" screen before transaction signing.** For example, display the full address in addition to the ENS name, the exact amount to transfer instead of a truncated amount (like < 0.00001), and the transaction nonce.
- **Add the WalletConnect Verify API to the product.** The Verify API enables applications to securely validate if the end user is on the correct domain. This solution makes phishing attacks harder.
- **Implement static analysis tools.** Implementing additional tools presented in appendix E will help to automatically find issues in the code that could lead to security vulnerabilities before they are merged into the codebase.
- **Implement fuzz testing for FFI bindings.** This part of the code performs manual data management using Rust's `unsafe` statements in multiple methods and may be prone to subtle memory errors.
- **Ensure that data received from external, online services cannot manipulate semantics of signed transactions.** In particular, ensure that Uniswap's endpoint reporting incorrect metadata for tokens (e.g., decimal places) does not cause the wallet to show incorrect prices to users.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	4
Low	9
Informational	13
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	3
Authentication	1
Configuration	8
Cryptography	5
Data Exposure	6
Data Validation	2
Error Reporting	1

Project Goals

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

- Does the Uniswap mobile wallet safely manage secret data?
- Are cryptographic algorithms implemented securely and according to their specifications in the Uniswap wallet?
- Is it possible to bypass any of the Uniswap wallet's confirmation screens (e.g., to automatically confirm transactions without user consent)?
- Is there any threat associated with using a QR code scanner in the Uniswap mobile wallet?
- Are deep links handled securely in the Uniswap mobile application?
- How securely are secrets stored?
- Can an attacker exploit any exported component of the Android application?
- Are there any architectural design flaws in the Android application manifest?
- Does the Android application use WebViews safely?
- Is ProGuard recommended for code obfuscation?
- Should the application remove any remnants upon manual account deletion or subsequent installs?
- Does the minimum API level required for the Uniswap wallet pose any risk?
- Are there any recommendations for Android devices that should generally be unsupported for security reasons?

Project Targets

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

[Redacted]

Repository	[Redacted]
Version	`392a770fce5656119c0b20816b70321972796a07`
Type	React Native
Platform	Android, iOS

ethers-rs-mobile

Repository	https://github.com/Uniswap/ethers-rs-mobile
Version	`0e3e3c6113aa296bdb93475d0335cb4fb0dcff6f`
Type	Rust
Platform	Multiplatform

Project Coverage

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

- A manual review of the [redacted] repository
- A review of the provided notes, including security concerns
- An automated review using CodeQL, TruffleHog, Semgrep, and Data Theorem
- A review of the WalletConnect integration
- Dynamic analysis of the Android application on an emulator and jailbroken device
- An integration and potential misconfiguration of WebViews
- Analysis of the underlying network traffic using Burp Suite Professional
- A manual review of business-logic of FFI bindings for the `EthersRs` object

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:

- Our main focus of this part of the audit was the Android version of the application. Although both the Android and iOS versions share the React Native codebase, this report focuses on issues identified mostly in the Android version.
- We did not perform a deep dive analysis of the `ethers-rs-mobile` repository. In particular, we did not audit the memory safety of this code.
- We did not perform comprehensive, dynamic scans of the Uniswap infrastructure.
- We could not 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. In particular, we did not audit features like transaction replacement and on-chain confirmation monitoring.
- 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 narrowed down our triage of Data Theorem results to the Android version of the Uniswap mobile wallet.

Automated Testing

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

Test Harness Configuration

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

Tool	Description	Policy
Semgrep	A static analysis tool designed to identify bugs and specific code patterns across multiple languages	Appendix D
TruffleHog	An open-source tool that scans Git repositories for secrets such as private keys and API tokens	Appendix D
CodeQL	A code analysis engine developed by GitHub to automate security checks	Appendix D

Areas of Focus

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

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

Codebase Maturity Evaluation

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

Category	Summary	Result
Arithmetic	We found no significant issues concerning the proper use of mathematical operations.	**Satisfactory**
Auditing	We did not have access to audit logs in remote resources. However, we did not find any issues, such as sensitive data leakage through the local logs on the device.	**Satisfactory**
Authentication / Access Controls	The authentication and access control mechanisms lack a minimum security policy, which means that users could use the application without any security mechanism (TOB-UNIMOB2-17). Also, the password policy on backups does not have a secure standard (TOB-UNIMOB2-2). We found a case where the application settings are inconsistent with the system biometric settings, which can lead to a situation where the application is not protected by biometrics (TOB-UNIMOB2-16). We also identified that local biometric authentication may be bypassed (TOB-UNIMOB2-13).	**Moderate**
Complexity Management	The Uniswap mobile codebase is generally well organized, divided by functionality across various directories.	**Satisfactory**
Configuration	We found instances where the Android Manifest configuration could make the application more robust (TOB-UNIMOB2-8, TOB-UNIMOB2-10, TOB-UNIMOB2-21). Also, we found that misconfigured logging in the OneSignal SDK exposes debuggable WebViews (TOB-UNIMOB2-19). The wallet does not conform to the BIP44 specification (TOB-UNIMOB2-24) either. The	**Moderate**

	SafetyNet Verify API (TOB-UNIMOB2-25) would enhance the user's security; thus, we recommend following the Android documentation for the best security practices.	
Cryptography and Key Management	The main cryptographic principles are implemented securely. However, we found that the static nonce is used for cloud backup encryption (TOB-UNIMOB2-4). Argon2i is used instead of Argon2id, which would be more secure for the mobile wallet (TOB-UNIMOB2-5). The application also should have countermeasures for sensitive data in RAM (TOB-UNIMOB2-14). Also, we found unimplemented certificate pinning (TOB-UNIMOB2-11).	**Moderate**
Data Handling	Generally, Uniswap takes the necessary precautions when validating most types of incoming data; our analysis did not reveal any issues that could enable typical injection attacks, such as cross-site scripting. However, developers should proactively address third-party code that exposes exported components in the application, as this code effectively increases the wallet's attack surface. Also, we found one minor bug because of lazy validation in initial views (TOB-UNIMOB2-9).	**Satisfactory**
Documentation	We did not have access to the internal documentation. Our audit was mostly informed by the concise Trail of Bits Audit 2023 notes.	**Further Investigation Required**
Maintenance	We found that some SDKs imported by the application are outdated. However, we assume a more detailed investigation is required to identify outdated and vulnerable third-party components.	**Further Investigation Required**
Memory Safety and Error Handling	Generally, error handling is implemented correctly. We found a minor issue with error handling where error messages should be unified (TOB-UNIMOB2-6). One functionality issue led to an infinite error loop (TOB-UNIMOB2-3). Errors are not specialized; the most generic `Error` is thrown and caught in multiple places instead of more specific or custom exceptions that would allow for more precise control of error flow. The unsafe Rust code needs further investigation for potential memory issues.	**Satisfactory**

Testing and Verification	We did not find any fuzz tests; these tests would be especially beneficial for the `ethers-rs-mobile` codebase. However, we did not measure the testing coverage and complexity during our audit.	**Further Investigation Required**

Summary of Findings

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

ID	Title	Type	Severity
1	Use of improperly pinned GitHub Actions	Access Controls	Low
2	Password policy issues on wallet backup with Google Drive	Authentication	Low
3	Infinite errors loop when the QR code is invalid	Error Reporting	Informational
4	Static AES-GCM nonce used for cloud backup encryption	Cryptography	Medium
5	Argon2i algorithm is used instead of Argon2id	Cryptography	Low
6	Errors from cryptographic operations contain too much information	Cryptography	Informational
7	Device-to-device backups are not disabled	Data Exposure	Low
8	Overly broad permission requests	Configuration	Low
9	Transaction amounts are obscured and lazily validated in initial views	Data Validation	Low
10	Potentially insecure exported NotificationOpenedReceiver activity	Configuration	Informational
11	Lack of certificate pinning for connections to the Uniswap server	Cryptography	Low
12	Third-party applications can take and read screenshots of the Android client screen	Data Exposure	Medium

13	Local biometric authentication is prone to bypasses	Access Controls	**Medium**
14	Wallet private keys and mnemonics may be kept in RAM	Data Exposure	**Informational**
15	Wallet sends requests with private data before the application is unlocked	Data Exposure	**Informational**
16	Biometric is not enabled for app access after enrollment	Configuration	**Informational**
17	Wallet does not require a minimum device-access security policy	Access Controls	**Informational**
18	Bypassable password lockout due to reliance on the phone's clock for time comparisons	Data Validation	**Low**
19	Debuggable WebViews	Configuration	**Medium**
20	Misconfigured GCP API key exposed	Configuration	**Low**
21	Lack of permissions for device phone number access or SIM card details	Configuration	**Informational**
22	An insecure HostnameVerifier that disables SSL hostname validation	Cryptography	**Informational**
23	Sentry SDK uses getRunningAppProcesses to get a list of running apps	Data Exposure	**Informational**
24	BIP44 spec is not followed	Configuration	**Informational**
25	SafetyNet Verify Apps API not implemented in the Android client	Configuration	**Informational**
26	Leakage of data to third-party	Data Exposure	**Informational**

Detailed Findings

1. Use of improperly pinned GitHub Actions	
Severity: **Low**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-UNIMOB2-1
Target: [redacted] GitHub Actions	

Description

The GitHub Actions workflows use several third-party actions 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.

```
17       - name: Build dev folder
18         uses: borales/actions-yarn@v4
// (...)
61             --clientSecret ${{ secrets.CI_GOOGLE_CLIENT_SECRET }}
```

Figure 1.1: The `borales/actions-yarn` action pinned only to a tag ([redacted])

```
27    - name: Create Pull Request
28     uses: peter-evans/create-pull-request@v3
29     with:
30       token: ${{ secrets.SERVICE_ACCOUNT_PAT }}
```

Figure 1.2: The `peter-evans/create-pull-request` action pinned only to a tag ([redacted])

```
44    uses: peter-evans/create-pull-request@v3
45    with:
46     token: ${{ secrets.SERVICE_ACCOUNT_PAT }}
```

Figure 1.3: The `peter-evans/create-pull-request` action pinned only to a tag ([redacted])

Exploit Scenario

An attacker gains unauthorized access to the account of a GitHub Actions owner. The attacker manipulates the action's code to secretly insert a backdoor. As a result, the hidden code is subsequently injected into the final version of the product, which remains undetected by the end users.

Recommendations

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

2. Password policy issues on wallet backup with Google Drive	
Severity: **Low**	Difficulty: **High**
Type: Authentication	Finding ID: TOB-UNIMOB2-2
Target: Uniswap Android application	

Description

When a user is creating a wallet with a backup using Google Drive, the Uniswap application requires that the password be at least eight characters long. This is a secure requirement; however, the application does not verify (or at least warn users about using) passwords that are low entropy or are composed of a small set of characters (e.g., aaaaaaaa). Also, rampant password reuse would trivialize the effort required to decrypt wallets in the event of a back-end breach. Recent research indicates that 62% of users reuse passwords across multiple sites. Many of those reused passwords are likely to have been leaked by unrelated hacks, allowing credential stuffers to purchase those credentials and theoretically decrypt wallets at little expense.

Exploit Scenario

A victim sets up a wallet using the backup Google Drive and password 12345678. The attacker gets unauthorized access to the victim's Google Drive and then can easily decrypt the Uniswap wallet using a common password wordlist, which leads to stolen funds.

Recommendations

Short term, consider using specific properties for password fields that will allow an OS to auto-generate strong passwords. From a UI perspective, implement a password strength meter to help users set a stronger password.

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

3. Infinite errors loop when the QR code is invalid

Severity: **Informational**	Difficulty: **High**
Type: Error Reporting	Finding ID: TOB-UNIMOB2-3
Target: Uniswap Android application	

Description

Upon scanning a QR code encoded with data that is inconsistent with the set logic for the Uniswap mobile application—such as having an invalid URI (figure 3.1)—the application responds with an error message (figure 3.2). The "try again" feature does not work as intended, so it is not possible to turn off this error message. As a result, the user is left with no option but to restart the application.

Figure 3.1: Example QR code with the encoded && 1=1/ string*

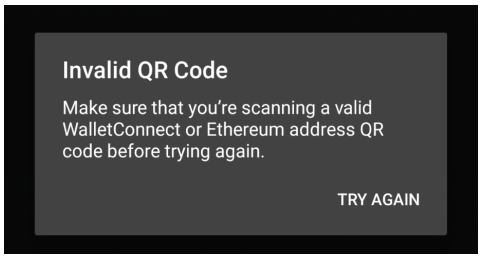

Figure 3.2: Error message upon scanning an invalid QR code

Recommendations

Short term, fix the application so that it handles the error correctly. The "try again" button should allow a user to scan a QR code again.

Long term, extend the testing suite with the collection of potentially invalid or malicious QR codes.

References

- MalQR: A collection of malicious QR codes and barcodes

4. Static AES-GCM nonce used for cloud backup encryption	
Severity: **Medium**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-UNIMOB2-4
Target: Uniswap Android application, cloud backups	

Description

The cloud backup feature of the Uniswap mobile wallet allows users to store encrypted mnemonics in the Google Drive application data folder. AES-GCM cipher mode is used for the encryption. The mode requires a unique, random nonce for every encryption operation. However, the Uniswap mobile wallet uses a constant nonce of 16 zeros. The vulnerable nonce generation is highlighted in figure 4.1.

```
fun encrypt(secret: String, password: String, salt: String): String {
    val key = keyFromPassword(password, salt)
    val cipher = Cipher.getInstance("AES/GCM/NoPadding")
    cipher.init(Cipher.ENCRYPT_MODE, SecretKeySpec(key, "AES"),
IvParameterSpec(ByteArray(16)))
    val encrypted = cipher.doFinal(secret.toByteArray(Charsets.UTF_8))
    return Base64.encodeToString(encrypted, Base64.DEFAULT)
}
```

Figure 4.1: AES-GCM encryption called by the `backupMnemonicToCloudStorage` function ([redacted])

While using constant nonces with AES-GCM is usually a critical vulnerability, the impact of the bug is reduced in the context of the Uniswap mobile wallet because the encryption key changes before every encryption. The key is derived from a password and a random salt, as shown in figures 4.1 and 4.2.

```
val encryptionSalt = generateSalt(16)
val encryptedMnemonic =
  withContext(Dispatchers.IO) { encrypt(mnemonic, password, encryptionSalt) }
```

Figure 4.2: Part of the `backupMnemonicToCloudStorage` function ([redacted])

If the encryption key had been reused, the following attacks would be possible:

- Authentication key recovery: The sub-key of the encryption key used for ciphertext authentication could be recovered from a few ciphertexts. This would allow an attacker to modify ciphertext in a meaningful way and recompute a valid authentication tag for the new version.
- Reuse of keystream: The XOR of two ciphertexts would result in the XOR of two plaintexts. Given such data, an attacker could perform statistical analysis to recover both plaintexts.

Exploit Scenario 1

In future releases of the Uniswap mobile wallet, the random salt is reused in a few subsequent encryptions. Users upload their mnemonics encrypted with a key that is the same for a few ciphertexts. The encrypted mnemonics are stolen from Google Drive and XORed pairwise. The criminals perform statistical analysis and obtain mnemonics in plaintext. They steal all users' tokens.

Exploit Scenario 2

Users reverse-engineer the Uniswap mobile wallet and learn that a constant nonce is used for AES-GCM encryption of backups. They publicly discuss this information on X (Twitter). Uniswap's credibility is negatively impacted.

Recommendations

Short term, replace the constant 16-byte nonce with a randomly generated, unique nonce in the `encrypt` function. Consider using nonces that are 12 bytes long instead of 16 bytes, as this length is more standard. Ensure that a strong, cryptographically secure pseudorandom number generator is used.

Long term, create an inventory of ciphers and cryptographic parameters used in the Uniswap mobile wallet. An inventory could easily catch vulnerabilities like weak parameters or incorrect generation of certain cryptographic data. The inventory must be kept up-to-date to be useful, so an internal process should be created for that task. For example, any pull request changing cryptographic code should include an update to the inventory. Moreover, the inventory should be periodically compared to the current cryptographic standards.

References

- Antoine Joux, "Authentication Failures in NIST version of GCM"

5. Argon2i algorithm is used instead of Argon2id

Severity: **Low**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-UNIMOB2-5
Target: Uniswap Android application, cloud backups	

Description

The Uniswap mobile wallet encrypts cloud backups with a key derived from a user's password and a random salt. The Argon2i algorithm is used for this task. This algorithm has a few known attacks that reduce its memory requirements. Therefore, it is more prone to brute-force attacks than the recommended Argon2id algorithm.

```
fun keyFromPassword(password: String, salt: String): ByteArray {
    val hash: Argon2KtResult = Argon2Kt().hash(
      mode = Argon2Mode.ARGON2_I,
      password = password.toByteArray(Charsets.UTF_8),
      salt = salt.toByteArray(Charsets.UTF_8),
      tCostInIterations = 3,
      mCostInKibibyte = 65536,
      parallelism = 4
    )
    return hash.rawHashAsByteArray()
}
```

Figure 5.1: Key derivation function using Argon2i algorithm ([redacted])

Differences between variants of the Argon2 algorithms are explained in RFC 9106 (see figure 5.2). Argon2id provides both protection from side-channel analysis and brute-force attacks, while Argon2i focuses on the former.

Argon2 is a memory-hard function [HARD]. It is a streamlined design. It aims at the highest memory-filling rate and effective use of multiple computing units, while still providing defense against trade-off attacks. Argon2 is optimized for the x86 architecture and exploits the cache and memory organization of the recent Intel and AMD processors. Argon2 has one primary variant, Argon2id, and two supplementary variants, Argon2d and Argon2i. Argon2d uses data-dependent memory access, which makes it suitable for cryptocurrencies and proof-of-work applications with no threats from side-channel timing attacks. Argon2i uses data-independent memory access, which is preferred for password hashing and password-based key derivation. Argon2id works as Argon2i for the first half of the first pass over the memory and as Argon2d for the rest, thus providing both side-channel attack protection and brute-force cost savings due to time-memory trade-offs. Argon2i makes more passes over the memory to protect from trade-off attacks [AB15].

Figure 5.2: Section 1 of RFC 9106 (https://www.rfc-editor.org/rfc/rfc9106.html#name-introduction)

Moreover, there is a bug in the salt generation function. The salt is Base64-encoded (see figure 5.3) and is not decoded to raw bytes before being passed to the Argon2i function. While this bug does not have security consequences, it may raise suspicion for users reading the code. Moreover, the library implementing the Argon2 function may misuse the encoded salt and, for example, truncate it to a predefined length, thereby reducing entropy.

```
fun generateSalt(length: Int): String {
    val bytes = ByteArray(length)
    val secureRandom = SecureRandom()
    secureRandom.nextBytes(bytes)
    return Base64.encodeToString(bytes, Base64.DEFAULT)
}
```

Figure 5.3: Base64-encoding of salt ([redacted])

Exploit Scenario

Adversaries steal encrypted backups from Google Drive. They perform brute-force attacks on the stolen data. The time and cost to conduct the attack are much lower than one would expect due to the usage of a weaker-than-possible algorithm.

Recommendations

Short term, replace the Argon2i function with Argon2id. This will protect the key derivation from both side-channel and brute-force attacks. Please note that Argon2d is not recommended, as the threat model of a mobile application includes side-channel attacks (e.g., performed by a malicious application running in the background).

Long term, create an inventory of cryptographic algorithms and parameters, as recommended to mitigate finding TOB-UNIMOB2-4. Provide reasoning for every algorithm and parameter choice that is not obvious.

References

- Dan Boneh, Henry Corrigan-Gibbs, and Stuart Schechter: "Balloon Hashing: A Memory-Hard Function Providing Provable Protection against Sequential Attacks"
- Joël Alwen and Jeremiah Block: "Towards Practical Attacks on Argon2i and Balloon Hashing"

6. Errors from cryptographic operations contain too much information

Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-UNIMOB2-6
Target: Uniswap Android application	

Description

The Uniswap mobile wallet should limit the amount of information about cryptographic failures. Providing too much detail on failures may allow attackers to extract valuable information about plaintexts and to, for example, mount a padding oracle attack.

While we have not observed any exploitable vulnerability in the wallet, the `restoreMnemonicFromCloudStorage` function, which decrypts AES-GCM encrypted cloud backups, is too verbose. The function returns three different errors, depending on the type of exception returned from the decrypting routine.

```
try {
  decryptedMnemonics = withContext(Dispatchers.IO) {
    decrypt(encryptedMnemonic, password, encryptionSalt)
  }
} catch (e: BadPaddingException) {
  Log.e("EXCEPTION", "${e.message}")
  promise.reject(
    CloudBackupError.BACKUP_INCORRECT_PASSWORD_ERROR.value,
    "Incorrect decryption password"
  )
} catch (e: IllegalBlockSizeException) {
  Log.e("EXCEPTION", "${e.message}")
  promise.reject(
    CloudBackupError.BACKUP_DECRYPTION_ERROR.value,
    "Incorrect decryption password"
  )
} catch (e: Exception) {
  Log.e("EXCEPTION", "${e.message}")
  promise.reject(
    CloudBackupError.BACKUP_DECRYPTION_ERROR.value,
    "Failed to decrypt mnemonics"
  )
}
```

Figure 6.1: Various errors returned by the `restoreMnemonicFromCloudStorage` function ([redacted])

Recommendations

Short term, have the `restoreMnemonicFromCloudStorage` function handle all cryptographic errors uniformly.

Long term, do a manual review to ensure that the application does not leak information via verbose cryptographic errors. Create a peer review policy that will ask reviewers to catch too verbose cryptographic errors.

References

- CVE-2019-1559: An example vulnerability whose root cause was that the vulnerable application responded differently to various decryption errors

7. Device-to-device backups are not disabled	
Severity: **Low**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-UNIMOB2-7
Target: Uniswap Android application	

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

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

Exploit Scenario 1
An adversary gains temporary physical access to a phone. He initiates a device-to-device transfer and copies the Uniswap mobile wallet's encrypted shared preferences to his device. Then he puts the phone back in place so the victim does not notice the incident. The adversary performs an offline brute-force attack and obtains the user's private keys.

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

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

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

8. Overly broad permission requests	
Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-UNIMOB2-8
Target: Uniswap Android application	

Description

The application requests the `android.permission.SYSTEM_ALERT_WINDOW` permission (figure 8.1), which appears to be broader than warranted by the respective functionalities of the application. The `SYSTEM_ALERT_WINDOW` permission is of concern because it has often been exploited; it enables an application to display over any other application without notifying the user, such as fraudulent ads or persistent screens.

The Android documentation states, "*Very few apps should use this permission; these windows are intended for system-level interaction with the user.*" Furthermore, if the application targets API level 23 or higher, the user must explicitly grant this permission to the application through a permission management screen.

```
<uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW"/>
```

Figure 8.1: The `AndroidManifest.xml` file in the Uniswap production release APK

Exploit Scenario

An attacker finds a way to use the `SYSTEM_ALERT_WINDOW` permission and prepares a tapjacking exploit. The attacker crafts a deceptive overlay on the user's device that tricks the user into thinking they are interacting with a legitimate function of the application. The user unknowingly triggers the action to send funds under the attacker's control. This results in the theft of the user's funds.

Recommendations

Short term, remove the `SYSTEM_ALERT_WINDOW` permission.

Long term, review the permissions required by the wallet and remove any permissions that the application does not need. Make an inventory of the required permissions with explanations of why they are needed.

9. Transaction amounts are obscured and lazily validated in initial views	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-UNIMOB2-9
Target: Uniswap Android application	

Description

The initial view (screen) of a new transaction and swap enables users to manually provide an amount to transfer. The amount is validated and cached if validation succeeds. The cached amount is used by the application even if the amount is changed to an arbitrary string (even an invalid one). This issue may allow adversaries to trick users into sending other amounts of tokens than the users wanted to send.

On the left of figure 9.1 is the initial view of a transaction where the number “2” was typed and later replaced with “1 2”. The user sees only the number “1”, as whitespaces are not visible. The user sees the converted USD value for the correct number “2”. Also, the user is shown the number “2” in the “review transaction” screen in the following figure.

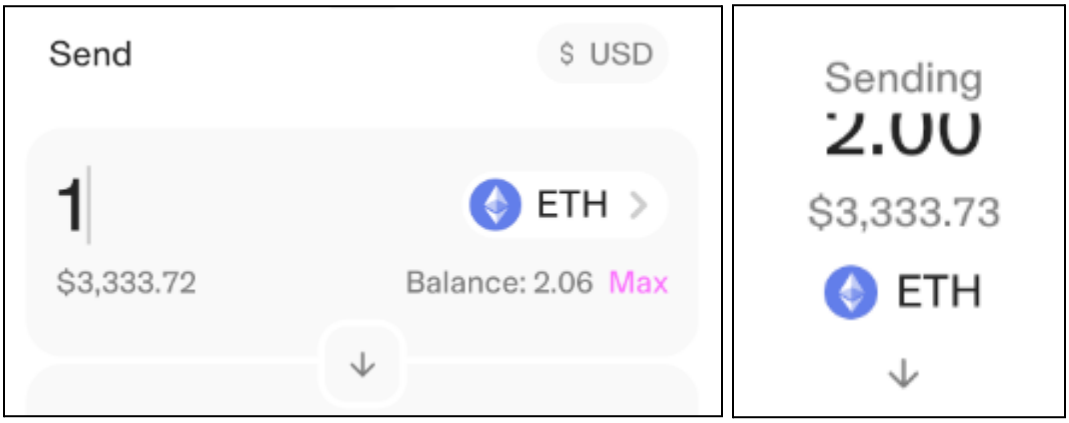

Figure 9.1: Two screenshots, the initial view and the “review transaction” screen

Exploit Scenario

Alice tricks Bob into copy-pasting an amount string with whitespaces to a transaction screen. Bob sees a small amount that he is willing to transfer to Alice and fails to validate the converted USD amount, which does not match the small amount. Bob also fails to validate the amount on the next screen and signs the transaction with a large amount of tokens.

Recommendations

Short term, validate amount strings after any changes and do not allow users to proceed with a transaction if the amount string is not valid. That is, instead of caching the last valid amount, always use the string that the user is shown on the screen.

Long term, ensure that on all screens with external data, the data shown to the user is exactly the same as that used by the application. In other words, what the user sees is what the application uses. Take special care with whitespaces, special characters, and UTF-8 encoded strings.

10. Potentially insecure exported NotificationOpenedReceiver activity

Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-UNIMOB2-10
Target: Uniswap Android application	

Description

The `com.onesignal.NotificationOpenedReceiver` activity in Uniswap's Android mobile wallet is exported (figure 10.1) and could allow access to internal components of the Uniswap mobile wallet. For example, the exported `NotificationOpenedReceiver` activity can behave as a proxy for the unexported content providers.

```
<activity android:theme="@android:style/Theme.Translucent.NoTitleBar"
android:name="com.onesignal.NotificationOpenedReceiver" android:exported="true"
android:taskAffinity="" android:excludeFromRecents="true" android:noHistory="true"/>
```

Figure 10.1: The exported activity in the `AndroidManifest.xml` file

The following proof-of-concept application (figure 8.2) calls the exported `com.onesignal.NotificationOpenedReceiver` activity. It then tries to access the unexported `com.uniswap.FileSystemFileProvider` provider because the provider has granted URI permissions (figure 8.3), so it is possible to use the `content://` URI scheme (figure 8.2, line 26). Even though we were unable to steal any Uniswap mobile wallet internal files during the audit, it proves that they are reachable from the perspective of a malicious application on the same device.

```
1   package com.example.myexploit
2
3   import android.content.Intent
4   import android.os.Bundle
5   import androidx.activity.ComponentActivity
6
7   const val BUNDLE_KEY_ACTION_ID = "actionId"
8   const val BUNDLE_KEY_ANDROID_NOTIFICATION_ID = "androidNotificationId"
9   const val BUNDLE_KEY_ONESIGNAL_DATA = "onesignalData"
10
11  class MainActivity : ComponentActivity() {
12     override fun onCreate(savedInstanceState: Bundle?) {
13         super.onCreate(savedInstanceState)
14         val intent = Intent("android.intent.action.VIEW")
15         intent.setClassName(
16             "com.uniswap",
17             "com.onesignal.NotificationOpenedReceiver"
18         )
19         intent.putExtra(BUNDLE_KEY_ACTION_ID, 123)
```

```
20            intent.putExtra("summary", "abc")
21            intent.putExtra(BUNDLE_KEY_ANDROID_NOTIFICATION_ID, 1337111)
22            intent.putExtra("action_button", false)
23            intent.putExtra("dismissed", false)
24            intent.putExtra("grp", "whatever")
25
26            val myString = "{ \"alert\": \"Test Msg\", \"custom\": { \"i\": \"UUID\", \"u\":
\"content://com.uniswap.FileSystemFileProvider/expo_files/\" } }"
27            intent.putExtra(BUNDLE_KEY_ONESIGNAL_DATA, myString)
28            startActivity(intent)
29        }
30    }
```

Figure 10.2: A proof of concept that uses the exported activity and unexported file system provider

```
<provider android:name="expo.modules.filesystem.FileSystemFileProvider"
android:exported="false" android:authorities="com.uniswap.FileSystemFileProvider"
android:grantUriPermissions="true">
```

Figure 10.3: The unexported provider with the URI permissions granted

The issue was discussed in a OneSignal GitHub issue when the `NotificationOpenedReceiver` was a broadcast receiver. The OneSignal developer responded that the `NotificationOpenedReceiver` becomes an activity and is unexported, but it became "silently" exported in commit `560203a`.

The issue is of informational severity because we were not able to exploit this vulnerability during the audit, and our attempts indicate that there is no current threat to the Uniswap wallet from this issue.

Exploit Scenario

A victim installs a malicious application on his device. The malicious application uses the exported activity to steal sensitive files from the victim's Uniswap mobile wallet, which allows the attacker to steal the victim's funds.

Recommendations

Short term, contact upstream library maintainers to resolve the issue or revise the necessity of using this exported activity. Keep in mind that removing the `exported="true"` flag from the `com.onesignal.NotificationOpenedReceiver` activity in the `AndroidManifest.xml` file could break functionality.

11. Lack of certificate pinning for connections to the Uniswap server	
Severity: **Low**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-UNIMOB2-11
Target: Uniswap Android application, Uniswap iOS application	

Description
The Uniswap mobile wallet does not use certificate pinning to require HTTPS connections to the Uniswap server to use a specific and trusted certificate or to be signed by a specific certificate authority (CA).

Certificate pinning is a method of allowing a specific server certificate or public key within an application to reduce the impact of person-in-the-middle (PITM) attacks. When making a connection to the back-end server, if the certificate presented by the server does not match the signature of the pinned certificate, the application should reject the connection.

The more general approach for certificate pinning is to limit the set of trusted CAs to only those that are actually used by a system.

The issue is of high difficulty because a successful attack requires installing a new CA on a target device or compromising one of the CAs trusted by the device. A CA compromise would be a security incident impacting the whole internet, and the chance that adversaries would target the Uniswap wallet is small. Moreover, modern mobile operating systems have mitigations for compromised CA incidents.

The impact of a successful PITM attack is similar to what would have happened if somebody compromised the Uniswap server, so it is of low severity (from the perspective of the wallet users).

Exploit Scenario 1
As part of a high-profile attack, an attacker compromises a CA and issues a malicious but valid SSL certificate for the server. Several trusted CAs have been compromised in the past, as described in this Google Security blog post.

Exploit Scenario 2
An attacker tricks a user into installing a CA certificate within their device's trust store. The attacker issues a malicious but valid SSL certificate and performs a PITM attack.

Recommendations

Short term, implement certificate pinning by embedding the specific certificates. If the server rotates certificates on a regular, short basis, then instead of pinning server certificates, limit the set of trusted CAs to the ones that will be used by the server.

Long term, implement unit tests that verify that the application accepts only the pinned certificate.

References

- OWASP: Certificate and Public Key Pinning Control
- TrustKit: Easy SSL pinning validation and reporting for iOS, macOS, tvOS, and watchOS

12. Third-party applications can take and read screenshots of the Android client screen

Severity: **Medium**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-UNIMOB2-12
Target: Uniswap Android application	

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

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

The finding is of high difficulty because the user would have to install malicious software on their device, and then the software would have to correctly guess the time point at which to make the screenshot. The severity of the finding is medium because the worst-case result of a successful exploit is that the user's private keys would be stolen.

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

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

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

13. Local biometric authentication is prone to bypasses	
Severity: **Medium**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-UNIMOB2-13
Target: Uniswap application, biometric authentication	

Description

The Uniswap mobile wallet application uses local biometric authentication to authorize sensitive actions like transaction signing or showing the mnemonic screen. The authentication is based on a simple `if` statement ("event-based") and not on any cryptographic primitive or remote endpoint ("result-based"). Result-based authentication is the recommended way to implement local authentication, because it is harder to bypass. With the result-based authentication, the wallet's data cannot be obtained even by attackers with physical access to the device and with root privileges. The event-based authentication can be bypassed by, for example, using dynamic instrumentation on rooted devices or by exploiting operating system vulnerabilities.

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

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

```
const result = await authenticateAsync(authenticateOptions)
if (result.success === false) {
  return BiometricAuthenticationStatus.Rejected
}
```

Figure 13.1: The "event-based" local authentication implemented in the `tryLocalAuthenticate` function ([redacted])

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

The issue is of high difficulty because exploitation requires root access to the device.

Exploit Scenario

Bob steals Alice's mobile device. He uses a public exploit to gain temporary root access to the device. He modifies the `/data/data/com.uniswap.dev/files/mmkv/mmkv.default` file to change the wallet's state, disabling biometric requirements. He then opens the wallet, displays a plaintext mnemonic, and uses it to steal all of Alice's funds.

Recommendations

Short term, reimplement local authentication to be result-based. Preferably, users' private keys and mnemonics should be stored encrypted with keychain's or keystore's key, and decrypted only on-demand and with biometric or screen lock PIN or password authorization. This solution may require replacing the currently used react-native Expo library.

The event-based local authentication may be kept for "app access" authentication, as this authentication does not protect any confidential information. However, we recommend implementing result-based authentication even for that part of authentication in order to store data that is not confidential but still sensitive, such as wallets' addresses and mnemonic IDs in encrypted form. Alternatively, ensure that the non-confidential data is stored encrypted with operating system mechanisms like the Data Protection entitlement.

Configure the wallet to require reauthorization before any action (instead of using time-based unlocking). This can be done with the `setUserAuthenticationRequired` and `setUserAuthenticationParameters` methods on Android, and `SecAccessControlCreateFlags` flags on iOS. On Android, the `RnEthersRs` class that uses Encrypted Shared Preferences can be leveraged, as it is already responsible for decrypting users' mnemonics and private keys.

Invalidate keys when a new fingerprint is added with the `InvalidatedByBiometricEnrollment` flag on Android and the `biometryCurrentSet` flag on iOS. Please note that this will require the user to recover the mnemonic whenever they change their device's PIN or add new fingerprints.

Ensure the keychain item is constrained by the device state, preferably with the `kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly` accessibility class flag on iOS and using the `canAuthenticate` method on Android. For iOS, ensure that the item belongs to the wallet's Access Group.

Ensure that decrypted (plaintext) users' private keys and mnemonics are not kept in wallet process' memory when not necessary. This security measure will limit the time window for forensics attacks.

When using the react-native wrapper library, ensure that the library correctly configures both Android and iOS authentication.

Assuming that biometric requirement settings are left configurable, and depending on the actual implementation of the recommendations above, React's storage used to persist these settings may require encryption. Otherwise, an adversary may be able to modify a plaintext file to disable the result-based authentication. For example, the currently used react-native-mmkv module's `encryptionKey` setting may be used for encryption. Please note that it requires further security investigation to determine if using the `encryptionKey` setting is enough to protect the wallet.

Long term, implement a second-factor authentication mechanism in addition to biometric authentication. Example second factors include user-provided passwords, passkeys, single sign-on with third-party identity providers, hardware devices like Yubico and Ledger, or login to a remote Uniswap service. User-provided passwords are a common mechanism for additional data encryption. Such solutions usually work by asking users to input their passwords, deriving the encryption key from the provided password, and decrypting users' data with it. The password should be used only in addition to the more convenient biometric or screen lock PIN authentication. This mechanism would protect users' data against offline attacks on the mobile device's secure hardware.

References

- OWASP: "Local Authentication on Android" and "Local Authentication on iOS" guidances
- Leonard Eschenbaum: "Bypassing Android Biometric Authentication", June 12, 2023
- Panagiotis Papaioannou: "A closer look at the security of React Native biometric libraries", April 6, 2021

14. Wallet private keys and mnemonics may be kept in RAM

Severity: **Informational**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-UNIMOB2-14
Target: [Redacted]	

Description

The Uniswap Android mobile wallet decrypts data stored in encrypted shared preferences and caches the data in the wallet process' memory. The cache is implemented in the `RnEthersRs` class (figure 14.1). The data contains plaintext private keys. Therefore, plaintext private keys may be kept unencrypted in RAM until the application is closed, even when the phone is locked.

```
private fun retrieveOrCreateWalletForAddress(address: String): Long {
  val wallet = walletCache[address]
  if (wallet != null) {
    return wallet
  }
  val privateKey = retrievePrivateKey(address)
  val newWallet = walletFromPrivateKey(privateKey)
  walletCache[address] = newWallet
  return newWallet
}
```

Figure 14.1: The method of the `RnEthersRs` class that caches private keys in memory ([redacted])

The issue is informational because the wallet React application creates new instances of the `RnEthersRs` class on-demand (e.g., as shown in figure 14.2); therefore, the cache implemented by the class is short-living. However, this behavior invalidates the benefits of having a cache, so it may be assumed that the intended use of the `RnEthersRs` class is to be a singleton. Moreover, the class is registered as a native module (figure 14.3), which again indicates that the class was intended to be a singleton.

```
val ethersRs = RnEthersRs(reactContext)
```

Figure 14.2: Example use of the `RnEthersRs` class ([redacted])

```
override fun createNativeModules(
    reactContext: ReactApplicationContext
): List<NativeModule> = listOf(
```

```
  RNEthersRSModule(reactContext),
  ThemeModule(reactContext),
)
```

Figure 14.3: The `RnEthersRs` class is registered as a native module. ([redacted])

Exploit Scenario

An attacker steals a user's phone but cannot unlock it. He exfiltrates the RAM content, which contains the user's private keys. The attacker uses the private key to take over the user's wallet and drain her funds.

Recommendations

Short term, remove the cache mechanism from the `RnEthersRs` class so that data stored in encrypted shared preferences is decrypted only on demand.

Long term, ensure that the application decrypts sensitive data only when it is needed (e.g., to sign a transaction or to display the wallet mnemonic) and removes it from RAM when it is no longer needed.

15. Wallet sends requests with private data before the application is unlocked	
Severity: **Informational**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-UNIMOB2-15
Target: Uniswap Android application	

Description

When the application launches while still awaiting to be unlocked by biometric or PIN authentication, the application starts fetching profile information over the network. This leads to the disclosure of information about the wallet registered in the application without needing to unlock it first. The wallet information is public, but associating a device or application instance with an account without needing to unlock the application is still a privacy issue.

Requests sent before the wallet is unlocked go to the `api.uniswap.org` endpoint, which contains operations like `TransactionList` and `PortfolioBalances` as well as the addresses of the currently registered mnemonic.

Recommendations

Short term, do not send HTTP requests before the application is unlocked.

16. Biometric is not enabled for application access after enrollment	
Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-UNIMOB2-16
Target: Uniswap application, biometric authentication	

Description

There are two settings for biometric authentication in the Uniswap mobile wallet: application access and transaction signing. When enabling biometric authentication during initial application enrollment, only the latter is enabled, and users are not informed that they should explicitly enable the other biometric setting.

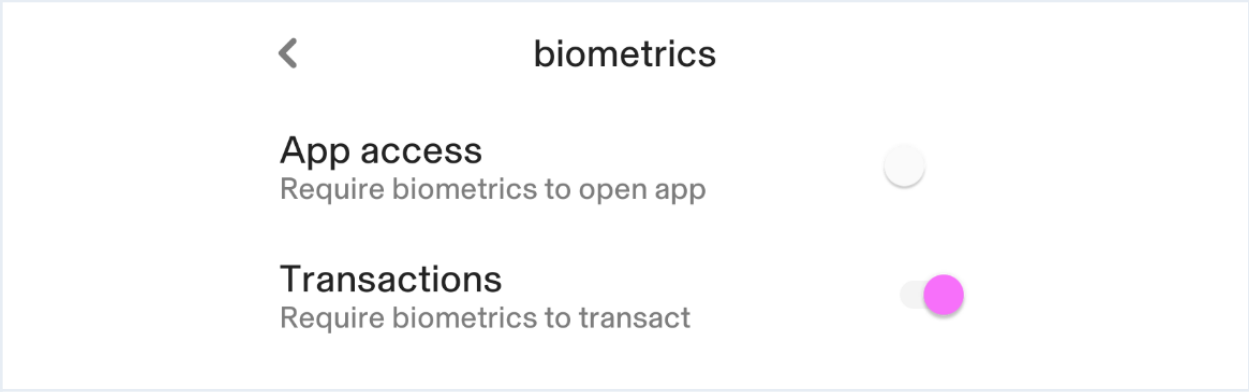

Figure 16.1: Default settings for biometric authentication after initial enrollment

This issue is only informational because it is the wallet user's responsibility to configure the wallet securely. Nevertheless, implementing the recommendations provided below would make the application more secure by default, or at least would increase users' awareness of the security-relevant configurations.

Exploit Scenario

Alice installs the Uniswap mobile wallet application and enables biometric authentication during initial enrollment. She is unaware that biometric authentication was enabled only for transaction signing and that there is a separate setting for application access. She views her mnemonic, authorizing access with a fingerprint. She then moves the wallet application to the background and uses other applications. Suddenly, Bob grabs Alice's phone, runs, and later moves the wallet application to the foreground. Since the application access setting is not enabled, he can see the plaintext mnemonic and steals Alice's funds.

Recommendations

Short term, enable both biometric authentication requirements (application access and transaction signing) when the user has enabled biometric authentication during the initial enrollment process. Alternatively, inform the user that they should manually enable the application access setting.

17. Wallet does not require a minimum device-access security policy

Severity: **Informational**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-UNIMOB2-17
Target: Uniswap application, biometric authentication	

Description

The Uniswap wallet application does not enforce a minimum device-access security policy, such as requiring the user to set a device passcode or PIN or to enroll a fingerprint for biometric authentication.

If a user removes the operating-system-level PIN, the wallet ceases to require biometric authentication. Moreover, the wallet settings still show "enabled" for biometric requirements. This behavior may be surprising to users and is a security "footgun."

The vulnerability is presented in figure 17.1, which shows that the wallet treats both successful authentication and disabled authentication on the operating system level as a success.

```
if (
  biometricAuthenticationSuccessful(authStatus) ||
  biometricAuthenticationDisabledByOS(authStatus)
) {
  successCallback?.(params)
} else {
  failureCallback?.()
}
```

Figure 17.1: Part of the `useBiometricPrompt` method ([redacted])

The issue is informational because it cannot be properly fixed without fixing TOB-UNIMOB2-13. And if that finding is resolved as recommended, then this finding is also fixed.

Exploit Scenario

A user of the Uniswap wallet enables biometric requirements for application access and transactions in the wallet. Then, he turns off biometric authentication on the OS level. He is convinced that the wallet is still protected with biometric authentication. The attacker steals the user's phone and gets access to his private keys. The user blames Uniswap for failing to protect the wallet.

Recommendations

Short term, implement result-based authentication as recommended in finding TOB-UNIMOB2-13. This will mitigate the vulnerability described in this finding, as disabling the device-access security policy would make the wallet unusable on the cryptographic level. If the user has enabled biometric authentication requirements in the wallet but has disabled OS-level authentication, then switch off the requirements so that users will not be misguided. Consider whether disabling or changing OS-level authentication should make the wallet delete all data, as is recommended in TOB-UNIMOB2-13.

18. Bypassable password lockout due to reliance on the phone's clock for time comparisons

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-UNIMOB2-18
Target: Uniswap Android application	

Description

The password lockout feature relies on the `Date.now` function to calculate the time left until a user can attempt to enter a password again. The `Date.now` function uses the system's clock, so an attacker can brute force the password by repeatedly trying a new password and advancing the phone's clock to bypass the lockout feature.

```
function calculateLockoutEndTime(attemptCount: number): number | undefined {
  if (attemptCount < 6) {
    return undefined
  }
  [skipped]
  if (attemptCount % 2 === 0) {
    return Date.now() + ONE_HOUR_MS
  }
  return undefined
}
```

Figure 18.1: The method for computing lockout's end time ([redacted])

```
const remainingLockoutTime = lockoutEndTime ? Math.max(0, lockoutEndTime -
Date.now()) : 0
const isLockedOut = remainingLockoutTime > 0
```

Figure 18.2: Code checking if wallet should be locked out ([redacted])

Exploit Scenario

An attacker steals a user's phone. The attacker tries a large number of passwords from a list of common passwords, changing the phone's time back and forth in between each attempt to bypass the lockout feature. The attacker finally learns the correct password and steals the user's funds.

Recommendations

Short term, instead of `Date.now`, use a source of time that returns the monotonic timestamp since the system booted. When a user reboots their device, the timestamp will

return to zero because zero seconds have passed since the system booted. A timestamp that is more recent than the timestamp of the last failed password attempt indicates that the system has rebooted and that the stored timestamp can safely be updated to zero. This measure means that users will have to wait through the full lockout time again after a reboot, but it ensures that attackers cannot manipulate the time left on a lockout. For monotonic timestamps on Android, use the `elapsedRealtime` function, and on iOS, use the `clock_gettime` function with the `CLOCK_MONOTONIC` argument.

19. Debuggable WebViews	
Severity: **Medium**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-UNIMOB2-19
Target: Uniswap Android application	

Description

The OneSignal SDK enables debugging of web contents loaded into any WebViews of the application for the debugging log level (figure 19.1) by the `setWebContentsDebuggingEnabled` flag. The OneSignal SDK allows Chrome Remote Debugging if `OneSignal.LOG_LEVEL` is equal to level DEBUG (5) or higher. The Uniswap mobile application has a verbose (6) log level enabled (figure 19.2), so any malicious application could inspect or modify the state of any WebView in the application. It is worth noting that access to the WebView context is not limited to OneSignal only; for example, it is possible to access the "Privacy Policy" view in the application.

```
private static void enableWebViewRemoteDebugging() {
    if (Build.VERSION.SDK_INT < 19 ||
!OneSignal.atLogLevel(OneSignal.LOG_LEVEL.DEBUG)) {
        return;
    }
    WebView.setWebContentsDebuggingEnabled(true);
}
```

Figure 19.1: The `enableWebViewRemoteDebugging` method in the `com.onesignal.WebViewManager` package

```
// 0 = None, 1 = Fatal, 2 = Errors, 3 = Warnings, 4 = Info, 5 = Debug, 6 = Verbose
export const initOneSignal = (): void => {
  OneSignal.setLogLevel(6, 0)
```

Figure 19.2: OneSignal setup in the Uniswap mobile wallet ([redacted])

Exploit Scenario

An attacker discovers that OneSignal uses the Chrome DevTools protocol for debugging, which exposes the content in WebViews over Unix domain sockets. He prepares the malicious application that sniffs the content of a WebView in the Uniswap mobile application. An attacker obtains the sensitive data from the Uniswap mobile wallet, which is then used to steal funds.

Recommendations

Short term, change the `setLogLevel` (figure 19.2) to level 4 or lower.

Long term, periodically check if the application exposes debuggable content on an Android device from the development machine. Also, using the jadx-gui tool, check if the decompiled APK contains the `setWebContentsDebuggingEnabled` flag and under what circumstances it enables debugging.

References

- Chrome for Developers: Remote debug Android devices
- `react-native-onesignal`: Disabled setWebContentsDebuggingEnabled

20. Misconfigured GCP API key exposed

Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-UNIMOB2-20
Target: Uniswap Android application, Uniswap iOS application	

Description

The Google Cloud Platform API key is embedded in the Uniswap mobile wallet code (figure 20.1, 20.2), which makes it publicly accessible. A test of the API key endpoint responds with an HTTP 200 status code, denoting insufficient key restrictions (figure 20.3). This could result in unanticipated costs and changes to the application's quota.

```
<string name="google_api_key">AIzaSyClPibETzdx02cLZtOW5oH7-nrpWDk77bI</string>
<string
name="google_crash_reporting_api_key">AIzaSyClPibETzdx02cLZtOW5oH7-nrpWDk77bI</strin
g>
```

Figure 20.1: The part of the `res/values/strings.xml` file in the decompiled Uniswap mobile APK

```
AIzaSyARi91A4ka3Tgk_lmbtF5pQE8kvt-odYr4
```

Figure 20.2: The part of the `GoogleService-Info.plist` file in the iOS `Uniswap.app` application

```
$ curl
https://www.googleapis.com/discovery/v1/apis\?key\=AIzaSyClPibETzdx02cLZtOW5oH7-nrpW
Dk77bI
{
  "kind": "discovery#directoryList",
  "discoveryVersion": "v1",
  "items": [
    {
      "kind": "discovery#directoryItem",
(...)
```

Figure 20.3: A proof of the key without sufficient restrictions that gives an "HTTP 200" response and JSON data

Recommendations

Short term, specify the Android application that can use the key: set the application restriction to "Android apps" and add the application package name with the SHA-1 signing

certificate fingerprint. For the iOS application, set the application restriction to “iOS apps” and add the bundle ID of the Uniswap iOS application.

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

References

- Authenticate by using API keys

21. Lack of permissions for device phone number access or SIM card details	
Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-UNIMOB2-21
Target: Uniswap Android application	

Description

The Uniswap mobile application references both Phone Number APIs (through the `react-native-device-info` dependency) and SIM card details (through AppsFlyer) without specifying the necessary permissions, such as `Manifest.permission.READ_PHONE_STATE`. While the application may not use the Phone Number APIs or the `getSimOperatorName` method during runtime, their presence in the application binary (figure 21.1, 21.2) or any included SDKs requires permission declaration. This absence of permissions can also lead to Google Play warnings during application review.

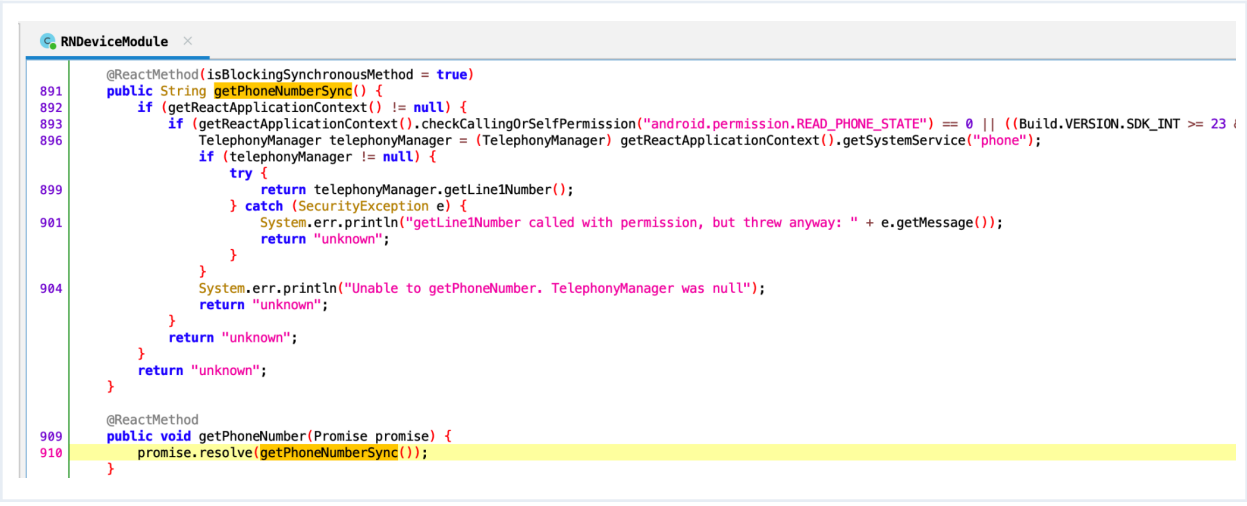

Figure 21.1: The `getPhoneNumberSync` method calling `getLine1Number`

```
telephonyManager = (TelephonyManager) context.getSystemService("phone");
str2 = telephonyManager.getSimOperatorName();
```

Figure 21.2: Usage of the `getSimOperatorName()` method from `TelephonyManager` in the `com.appsflyer.internal` package

Recommendations

Short term, if the `react-native-device-info` or AppsFlyer dependency contains references to the Phone Number APIs or TelephonyManager and is unnecessary, consider

removing it or asking the vendors for a build that does not contain code to access the data. If access to the phone number or SIM card details is required, declare the correct permissions.

22. An insecure HostnameVerifier that disables SSL hostname validation

Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-UNIMOB2-22
Target: Uniswap Android application	

Description

The `NoopHostnameVerifier` class in the `org.apache.http.conn.ssl` package defines a `HostnameVerifier()` method that does not validate the server's hostname (figure 22.1). This allows an attacker to perform a PITM attack on a user's connection to spoof the server with the user's hostname by providing a certificate from another host. Due to the lack of hostname verification, the client would accept this certificate.

It is important to note that, according to the policy, "Beginning March 1, 2017, Google Play will block publishing of any new apps or updates that use an unsafe implementation of `HostnameVerifier`."

```
package org.apache.http.conn.ssl;

import javax.net.ssl.HostnameVerifier;
import javax.net.ssl.SSLSession;

/* loaded from: classes5.dex */
public class NoopHostnameVerifier implements HostnameVerifier {
    public static final NoopHostnameVerifier INSTANCE = new NoopHostnameVerifier();

    public final String toString() {
        return "NO_OP";
    }

    @Override // javax.net.ssl.HostnameVerifier
    public boolean verify(String str, SSLSession sSLSession) {
        return true;
    }
}
```

Figure 22.1: The decompiled `NoopHostnameVerifier` class in the `jadx-gui` tool

The issue is of informational severity because we were unable to exploit this finding by using a certificate signed by a valid CA but for invalid hostnames. In this case, logcat shows an SSL error.

Exploit Scenario

From the user's perspective: An attacker carries out a PITM attack by using a CA-signed certificate issued for a domain the attacker owns. Because the implementation of the `HostnameVerifier` method accepts any certificate signed by a valid CA for any hostname, the attacker's certificate is accepted.

From the application owner's perspective: The application is removed from or is blocked from being published in Google Play because of the unsafe implementation of the `HostnameVerifier` method, which does not validate hostnames.

Recommendations

Short term, identify which library introduces the vulnerable code and follow potential fixes to ensure that the Uniswap mobile wallet uses the default hostname validation logic or that the custom `HostnameVerifier` interface returns `false` when the server's hostname does not meet the expected value.

References

- Google Help: How to resolve Insecure HostnameVerifier

23. Sentry SDK uses getRunningAppProcesses to get a list of running apps	
Severity: **Informational**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-UNIMOB2-23
Target: Uniswap Android application	

Description

The Sentry SDK uses the `getRunningAppProcesses` method (figure 23.1), which is intended only for debugging or building a user-facing process management the UI. Also, the Android Security 2014 Year in Review Google report states that, "*Throughout 2014, we have regularly tightened the definition of Spyware, for example in 2014 we began to classify applications that send the list of other applications on the device as Spyware*." This may cause the Uniswap mobile application to be removed from the store.

```
public static boolean isForegroundImportance(Context context) {
    List<ActivityManager.RunningAppProcessInfo> runningAppProcesses;
    try {
        Object systemService = context.getSystemService("activity");
        if (!(systemService instanceof ActivityManager) || (runningAppProcesses = ((ActivityManager) systemService).getRunningAppProcesses()) == null) {
            return false;
        }
        int myPid = Process.myPid();
        for (ActivityManager.RunningAppProcessInfo runningAppProcessInfo : runningAppProcesses) {
            if (runningAppProcessInfo.pid == myPid) {
                return runningAppProcessInfo.importance == 100;
            }
        }
        return false;
    } catch (Throwable unused) {
        return false;
    }
```

Figure 23.1: Part of the `io.sentry.android.core` package that uses the `getRunningAppProcesses` method

The issue remains open on the Sentry GitHub: Issue #2187, Consider removing function call: ActivityManager.getRunningAppProcesses().

Recommendations

Short term, refer to Sentry to get information about when the `isForegroundImportance` is finally updated, or consider removing the Sentry SDK from the Uniswap wallet.

Long term, periodically review other usages of the `getRunningAppProcesses` method using the `jadx-gui` tool on the production release APK.

24. BIP44 spec is not followed

Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-UNIMOB2-24
Target: Uniswap application	

Description

The Uniswap mobile wallet automatically imports at most the first 10 wallets associated with a mnemonic. Therefore, if a user has a mnemonic associated with more than 10 wallets, the Uniswap wallet will automatically import only the first 10. BIP44 specifies that wallets should be imported until 20 subsequent addresses have no transaction histories ("address gap limit").

```
export const NUMBER_OF_WALLETS_TO_IMPORT = 10
```

Figure 24.1: Hard-coded limit of imported wallets ([redacted])

Moreover, the Uniswap wallet filters unused wallets by balances, instead of transaction history, as the BIP44 specifies.

```
const accountsWithBalance = filteredAccounts?.filter(
  (address) => address.balance && address.balance > 0
)

if (accountsWithBalance?.length) return accountsWithBalance
```

Figure 24.2: Filtering wallets by their balances ([redacted])

Recommendations

Short term, as specified by BIP44, revise the code so that it keeps searching for wallets until it finds a gap of 20 unused wallets. Consider making a hard limit or pagination for wallets so that a bug in remote services that reports transaction histories will not make the wallet loop infinitely. Filter unused wallets by their transaction histories and not by actual balances.

Long term, ensure that the BIP44 implementation matches the BIP44 specification. Allow users to import wallets with arbitrary derivation paths.

25. SafetyNet Verify Apps API not implemented in the Android client	
Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-UNIMOB2-25
Target: Uniswap Android application	

Description

The Uniswap Android application does not use the SafetyNet Verify Apps API.

Google Play provides the SafetyNet Verify Apps API to check whether potentially harmful applications are on a user's device. Through the Verify Apps feature, Google monitors and profiles the behavior of Android applications, informs users of potentially harmful applications, and encourages users to delete them. However, users are free to disable this feature and to ignore these warnings. The SafetyNet Verify Apps API can tell Uniswap whether the Verify Apps feature is enabled and whether potentially malicious applications remain on the user's device. Uniswap can then take actions like warning users or disabling access to the wallet until the user resolves the problem or accepts the risk. This can provide an additional line of defense.

Please note that the SafetyNet Verify Apps API is distinct from the deprecated SafetyNet Attestation API, and the SafetyNet Verify Apps API should be used together with the Play Integrity API.

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

Exploit Scenario

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

Recommendations

Short term, implement the SafetyNet Verify Apps API to require that the Verify Apps feature be enabled for all Uniswap users and to ensure that known harmful applications are not installed on users' devices. If malicious applications are detected by the API, alert wallet users about that, and instruct them on recommended actions they should take (e.g., uninstalling the applications in question).

Long term, stay updated on new security features introduced in Android and continue adding relevant safety protections to the Uniswap mobile application. For added security protection, consider verifying the device's integrity using the Play Integrity API before using the Verify Apps API.

References

- Android Developers: SafetyNet Verify Apps API
- Android Developers: App Security Best Practices

26. Leakage of data to third-party

Severity: **Informational**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-UNIMOB2-26
Target: Uniswap Android application, Uniswap iOS application	

Description

We found that the Uniswap mobile wallet shares device-specific information with third-party entities, including OneSignal, Sentry, and Google Services. This practice poses privacy risks, as the shared data encompasses attributes such as time zone, device model, OS version, and Advertising Identifier.

For instance, the application sends requests with the data shown in figure 26.1 to the OneSignal API:

```
POST https://api.onesignal.com/players
SDK-Version:  onesignal/android/040805
Accept:  application/vnd.onesignal.v1+json
Content-Type:  application/json; charset=UTF-8
Content-Length:  500
User-Agent:  Dalvik/2.1.0 (Linux; U; Android 10; Pixel 3a Build/QQ1A.191205.011)

{"app_id":"5b27c29c-281e-4cc4-8659-a351d97088b0","device_os":"dt.osv.11","timezone":
-14400,"timezone_id":"America\/New_York","language":"en","sdk":"040805","sdk_type":"
react","android_package":"com.uniswap","device_model":"Pixel
4","game_version":1000001,"net_type":0,"carrier":"DT_Carrier","rooted":true,"identif
ier":"etFD-KouTlWyiNLnbq-kwR:APA91bGRu1ougP1RVPB4nJoLNr3iIrZ0s6hN44btOIGF-xt592HeKch
Z89rnn7PeGhK00a5XTu-NdRk_t0wps69JJkWfgZCTdFyy3uSEU6WG_zJUCA79uBL5TH-i206OeX35BM-Nbm7
3","device_type":1}
```

Figure 26.1: Example request from the Uniswap mobile wallet to the OneSignal API

Exploit Scenario

An attacker intercepts data transmitted by the Uniswap mobile wallet, which includes information about the user's device model, carrier, and physical location. Using this data, the adversary creates a highly targeted phishing attack that appears to be a legitimate communication from Uniswap or the identified carrier. The communication directs the user to a convincingly mimicked import wallet page. Unaware of the malicious redirect, the user enters his mnemonics. With this information, the attacker gains access to the user's account on Uniswap and outright steals the user's funds.

Recommendations

Short term, if sending device details to the parties is desired, include a comprehensive overview of this data-sharing practice in the application's privacy policy. If not, configure relevant SDKs to not share redundant user data or remove specific SDKs from the Uniswap mobile wallet codebase if they are not needed.

Long term, periodically perform network analysis via Burp Suite Professional to monitor and verify the type of data transmitted to third parties.

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 Findings

This appendix contains findings that do not have immediate or obvious security implications. However, they may facilitate exploit chains targeting other vulnerabilities, become easily exploitable in future releases, or decrease code readability. We recommend fixing the following issues:

1. **Incorrect comment in the `getNextDerivationIndex` function.** The comment states that the `nextDerivation` variable is set to `sortedMnemonicAccounts.length + 1`, but the variable is set to `sortedMnemonicAccounts.length`.

```
function getNextDerivationIndex(sortedAccounts: SignerMnemonicAccount[]): number {
  // if there is a missing index in the series (0, 1, _, 3), return this missing
index
  let nextIndex = 0
  for (const account of sortedAccounts) {
    if (account.derivationIndex !== nextIndex) {
      return Math.min(account.derivationIndex, nextIndex)
    }
    nextIndex += 1
  }
  // if all exist, nextDerivation = sortedMnemonicAccounts.length + 1
  return nextIndex
}
```

Figure C.1.1: The highlighted comment is incorrect. ([redacted])

2. **The `RnEthersRs` class' `init` method uses a deprecated version of the `EncryptedSharedPreferences.create` method.** The new version takes the `MasterKey` object as an input parameter instead of a `masterKeyAlias` string.

```
val masterKeyAlias = MasterKeys.getOrCreate(MasterKeys.AES256_GCM_SPEC)
keychain = EncryptedSharedPreferences.create(
  "preferences",
  masterKeyAlias,
  applicationContext,
  EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
  EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
)
```

Figure C.2.1: The `init` method of the `RnEthersRs` class ([redacted])

3. **Make validation of currency IDs more strict.** The IDs should contain only one dash, but this property is never validated. If a currency ID has multiple dashes, it will be accepted as valid.

```
// Currency ids are formatted as `chainId-tokenaddress`
export function currencyIdToAddress(_currencyId: string): Address {
  const currencyIdParts = _currencyId.split('-')
  if (!currencyIdParts[1]) throw new Error(`Invalid currencyId format:
${_currencyId}`)
  return currencyIdParts[1]
}
```

Figure C.3.1: One of the currency ID validation methods ([redacted])

4. **Parse and manipulate URLs with functions designated for this task.** Do not use simple string-manipulation functions.

```
export function getTwitterLink(twitterName: string): string {
  return `https://twitter.com/${twitterName}`
}
```

Figure C.4.1: An example function that modifies a URL using string substitution instead of a URL-aware method ([redacted])

```
if (!isSafeUri && !ALLOWED_EXTERNAL_URI_SCHEMES.some((scheme) =>
trimmedURI.startsWith(scheme))) {
```

Figure C.4.2: Another example of using the `startsWith` string function instead of parsing the URL with the proper library and validating the scheme ([redacted])

5. **Explain camera access in the Privacy Dashboard.** With the introduction of Android 12, users can use the Privacy Dashboard feature to monitor application permissions, especially location, microphone, and camera access. Implement access justifications in the new Privacy Dashboard or the application's permissions screen.

6. **Opt out of the metrics collection.** WebViews can upload anonymous diagnostic data to Google when the user consents.

7. **Prevent the Uniswap application and its third-party SDKs from targeting users through the Advertising ID.** Add the following entry to the Android manifest:

```
<uses-permission
android:name="com.google.android.gms.permission.AD_ID"
tools:node="remove"/>
```

This measure blocks any SDK that attempts to target ads to your users by declaring the `com.google.android.gms.permission.AD_ID` permission in their library manifest. Without this precaution, even if not directly declared in the Uniswap application's main manifest, the permission would be merged, allowing potential ad targeting.

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 multiple predefined rules simultaneously by providing multiple `--config` arguments):

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

semgrep --config auto
```

We recommend using Semgrep. To thoroughly understand the Semgrep tool, refer to the Trail of Bits Testing Handbook, where we aim to streamline the use of Semgrep and improve security testing effectiveness. Also, consider doing the following:

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

CodeQL

We installed CodeQL by following CodeQL’s installation guide.

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

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

We then ran the following command to query the database:

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

We also used private Trail of Bits query packs.

TruffleHog

We used TruffleHog to detect sensitive data such as private keys and API tokens in the repositories' Git histories.

To detect sensitive information in the `smartcontractkit` GitHub organization with TruffleHog, we used the following command:

```
trufflehog github --org=Uniswap --only-verified
```

The `--only-verified` flag specifies that only findings marked as "verified" should be included in the scan results. This helps filter out false positives and focuses on confirmed instances of sensitive information.

E. Fix Review Results

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

On October 20, 2023, 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.

In summary, of the 26 issues described in this report, Uniswap has resolved eight issues, has partially resolved one issue, and has not resolved the remaining 17 issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Use of improperly pinned GitHub Actions	Resolved
2	Password policy issues on wallet backup with Google Drive	Resolved
3	Infinite errors loop when the QR code is invalid	Resolved
4	Static AES-GCM nonce used for cloud backup encryption	Resolved
5	Argon2i algorithm is used instead of Argon2id	Resolved
6	Errors from cryptographic operations contain too much information	Unresolved
7	Device-to-device backups are not disabled	Resolved
8	Overly broad permission requests	Resolved
9	Transaction amounts are obscured and lazily validated in initial views	Partially Resolved

10	Potentially insecure exported NotificationOpenedReceiver activity	Unresolved
11	Lack of certificate pinning for connections to the Uniswap server	Unresolved
12	Third-party applications can take and read screenshots of the Android client screen	Unresolved
13	Local biometric authentication is prone to bypasses	Unresolved
14	Wallet private keys and mnemonics may be kept in RAM	Unresolved
15	Wallet sends requests with private data before the application is unlocked	Unresolved
16	Biometric is not enabled for app access after enrollment	Unresolved
17	Wallet does not require a minimum device-access security policy	Unresolved
18	Bypassable password lockout due to reliance on the phone's clock for time comparisons	Unresolved
19	Debuggable WebViews	Resolved
20	Misconfigured GCP API key exposed	Unresolved
21	Lack of permissions for device phone number access or SIM card details	Unresolved
22	An insecure HostnameVerifier that disables SSL hostname validation	Unresolved
23	Sentry SDK uses getRunningAppProcesses to get a list of running apps	Unresolved
24	BIP44 spec is not followed	Unresolved

25	SafetyNet Verify Apps API not implemented in the Android client	**Unresolved**
26	Leakage of data to third-party	**Unresolved**

Detailed Fix Review Results

TOB-UNIMOB2-1: Use of improperly pinned GitHub Actions
Resolved in commit `fd87a7fab049aece33269d8d469038136f46d50d`. Actions are pinned to specific full-length commit SHA hashes.

TOB-UNIMOB2-2: Password policy issues on wallet backup with Google Drive
Resolved in commit `ee431570dd090c9cf4d86d085cef131baea26db5`. The wallet uses `zxcvbn` to inform users about the strength of their passwords.

TOB-UNIMOB2-3: Infinite errors loop when the QR code is invalid
Resolved in commit `0e9017abd3fc033e1ce7df9e4d4f1cdbb35dd9ac`. An invalid QR code does not cause the application to go into an infinite error loop.

TOB-UNIMOB2-4: Static AES-GCM nonce used for cloud backup encryption
Resolved in commit `999da894ed437022d230efdeb5e09f96f5e39711`. The wallet uses randomly generated nonces for AES-GCM encryption implemented in the `EncryptionHelper.kt` file.

TOB-UNIMOB2-5: Argon2i algorithm is used instead of Argon2id
Resolved in commit `6705785b0bd1cae4376325983adedb1f65f9edbd`. The Argon2i function was changed to Argon2id in the `EncryptionHelper.kt` file.

TOB-UNIMOB2-6: Errors from cryptographic operations contain too much information
Unresolved.

TOB-UNIMOB2-7: Device-to-device backups are not disabled
Resolved in commit `745cd55afe8c48fc85797a7b0ab9384ec9b80b4b`. Device-to-device backups are disabled.

TOB-UNIMOB2-8: Overly broad permission requests
Resolved in commit `adec74f1307efba63ba3efb61bf7ac7b7de6a6b3`. The `SYSTEM_ALERT_WINDOW` permission is removed from production and development manifests.

TOB-UNIMOB2-9: Transaction amounts are obscured and lazily validated in initial views
Partially resolved in commit `63a8caa6446c716f4a710cb080692f42abf72880`. The text cutoff was fixed in the transaction review screen, but the main issue in the transaction creation screen remains unfixed. Valid amounts are cached and used even if an invalid amount is later typed into the input box.

TOB-UNIMOB2-10: Potentially insecure exported NotificationOpenedReceiver activity
Unresolved.

TOB-UNIMOB2-11: Lack of certificate pinning for connections to the Uniswap server
Unresolved.

TOB-UNIMOB2-12: Third-party applications can take and read screenshots of the Android client screen
Unresolved.

TOB-UNIMOB2-13: Local biometric authentication is prone to bypasses
Unresolved.

TOB-UNIMOB2-14: Wallet private keys and mnemonics may be kept in RAM
Unresolved.

TOB-UNIMOB2-15: Wallet sends requests with private data before the application is unlocked
Unresolved.

TOB-UNIMOB2-16: Biometric is not enabled for app access after enrollment
Unresolved.

TOB-UNIMOB2-17: Wallet does not require a minimum device-access security policy
Unresolved.

TOB-UNIMOB2-18: Bypassable password lockout due to reliance on the phone's clock for time comparisons
Unresolved.

TOB-UNIMOB2-19: Debuggable WebViews
Resolved in commit `74ce2b3debc1782f32e70dc507340b0e40c498be`. The OneSignal log level is changed to the default value, so the WebViews are no longer debuggable.

TOB-UNIMOB2-20: Misconfigured GCP API key exposed
Unresolved.

TOB-UNIMOB2-21: Lack of permissions for device phone number access or SIM card details
Unresolved.

TOB-UNIMOB2-22: An insecure HostnameVerifier that disables SSL hostname validation
Unresolved.

TOB-UNIMOB2-23: Sentry SDK uses getRunningAppProcesses to get a list of running apps
Unresolved.

TOB-UNIMOB2-24: BIP44 spec is not followed
Unresolved.

TOB-UNIMOB2-25: SafetyNet Verify Apps API not implemented in the Android client
Unresolved.

TOB-UNIMOB2-26: Leakage of data to third-party
Unresolved.

F. Fix Review Status Categories

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

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