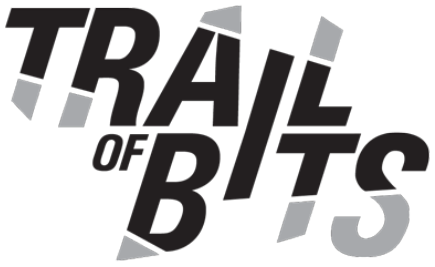

Tezos Crypto Wallets

Security Assessment

June 8, 2020

Prepared For:
Tezos Foundation
security@tezos.com

Prepared By:
Artur Cygan | *Trail of Bits*
artur.cygan@trailofbits.com

Changelog:
July 30, 2020: Added new iOS and Android findings
September 10, 2020: Initial fix review
September 21, 2020: Additional fix review
September 29, 2020: Additional fix review
September 29, 2020: Final report for publication

Executive Summary

From June 1 through June 5, 2020, Tezos Foundation engaged Trail of Bits to review the security of the Magma Wallet. Trail of Bits conducted this assessment over the course of one person-week with one engineer working from:

- 4b5e84d3156393717582323388887fd0debd7b7b0 from the `kotlin-tezos` repository
- 539c8459964b2e78dfd451aa777498510d03efc3 from the `magmawallet-android` repository
- 48c9ca512e95ac0a51a8a2cf9443b96b5629f505 from the `magmawallet-ios` repository

We performed manual review of the provided code around the most critical spots, and verified that both wallets could be built and run on simulators. We also took a closer look at the user interface.

In the course of our review, we found one high-, eleven medium-, thirteen low-, and four informational-severity findings. The high-severity issue is the copy mnemonic to clipboard feature which can cause it to be stolen by a malicious application or to be leaked accidentally by one of popular applications which automatically read the system clipboard content. Most of the medium-severity findings also concern security of the mnemonic indicating places where the wallet could be improved to reduce the possibility for an exposure. The remaining minor issues revolve around testing methodology, imprecise error handling, cryptography, and data validation.

Both the Android and iOS wallets are lacking when it comes to their current level of security, considering the importance of the secrets they store. The codebases are under active development which includes occasional work-in-progress or commented-out code. The Android wallet's code is slightly better in terms of quality than that of the iOS wallet. Imprecise error handling and data validation are systemic issues that don't pose an immediate threat, but might manifest as a number of security issues in the future.

Overall, we recommend rethinking the user interface to favor security over convenience. Minimizing secrets' lifetimes will require some attention and debugging. The remaining minor issues can be fixed with simple code changes, and adopting our long-term recommendations will help prevent them in the future.

Update: On September 10, 2020, Trail of Bits reviewed fixes implemented for the issues presented in this report—see details in Appendix G.

Update: On September 21, 2020, Trail of Bits discussed the previously reviewed solution to finding #14 with Tezos and, upon further research, the finding was determined to be resolved.

Update: On September 29, 2020, Trail of Bits updated finding #2 after discussing code updates contained in commits that were part of the previously reviewed Pull Requests.

Project Dashboard

Application Summary

Name	Magma Wallet
Version	`kotlin-tezos`: 4b5e84d3156393717582323388887fd0debd7b7b0
	`magmawallet-android`: 539c8459964b2e78dfd451aa777498510d03efc3
	`magmawallet-ios`: 48c9ca512e95ac0a51a8a2cf9443b96b5629f505
Type	Kotlin, Java, Swift
Platforms	iOS, Android

Engagement Summary

Dates	June 1 through June 5, 2020
Method	Whitebox
Consultants Engaged	1
Level of Effort	1 person-week

Vulnerability Summary

Total High-Severity Issues	1	■
Total Medium-Severity Issues	11	■■■■■■■■■■■
Total Low-Severity Issues	11	■■■■■■■■■■■
Total Informational-Severity Issues	4	■■■■
Total	27	

Category Breakdown

Access Controls	2	■■
Authentication	2	■■
Configuration	2	■■
Cryptography	7	■■■■■■■
Data Exposure	8	■■■■■■■■

Data Validation	2	■■
Error Reporting	3	■■■
Testing	1	■
Total	27	

Engagement Goals

The engagement was scoped to provide a security assessment of the Android wallet, Android SDK, and the iOS wallet.

Specifically, we sought to answer the following questions:

- Is the wallet securely stored?
- Is it possible for an adversary to execute unauthorized actions?
- Is calling external APIs safe?
- Is error handling implemented correctly?

Coverage

Key management and storage. Manually reviewed with the help of automated static analysis using Data Theorem and debugging tools.

User interface. Manually reviewed with the help of automated static analysis using Data Theorem.

Cryptography. Manually reviewed with the help of automated static analysis using Data Theorem.

Communication with APIs. Manually reviewed with the help of automated static analysis using Data Theorem.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Don’t allow the mnemonic to be copied to the clipboard.** Ensure users store the phrase by requiring them to retype it in a following view during onboarding. TOB-TZWL-001

❑ **Fetch the mnemonic from the keychain only when absolutely needed, e.g., right before signing or when displaying it in settings.** Clear out sensitive information from memory when it is no longer needed. TOB-TZWL-002

❑ **Implement the highest security level for keychain items, especially for the mnemonic.** Pass the security level explicitly to the KeychainSwift library. TOB-TZWL-003

❑ **Check keychain error codes and handle them appropriately.** Unhandled errors may leave keys in the keychain unintentionally. TOB-TZWL-004

❑ **Add a timeout to the recovery phrase view that exits the view or hides the secret.** Users may leave their device unattended and expose the keyphrase unintentionally. TOB-TZWL-005

❑ **Ensure the PIN code is requested.** Once set, the wallet should never be accessible without the PIN code. TOB-TZWL-006

❑ **Validate the number of words a user provides.** Either lock the system to 12-word mnemonics only, or fully support a variable word count. TOB-TZWL-007

❑ **Mock the actual external services rather than abstract ones defined by the application.** Re-implement only what's needed right at the edge of the system. Inject faulty scenarios and test error cases in addition to happy path testing. TOB-TZWL-008

❑ **Rethink and redesign error handling in the cases where the catch-all pattern occurs.** Catching all exception types can silence important errors. TOB-TZWL-009

❑ **Change `encryptWithSalt` to signal failure through the return type or an exception, rather than falling back to a weaker hash function.** Do not silently weaken cryptographic primitives. TOB-TZWL-010

❑ **Remove the try ... catch block from the generateMnemonic function and change the return type to List<String>.** Collapsing multiple issues into a single value impedes debugging. TOB-TZWL-011

❑ **Change the Wallet structure to always include the mnemonic field.** This matches the expectation that developers will have of the SDK. TOB-TZWL-012

Long Term

❑ **Avoid copying sensitive data to the clipboard.** Clipboard data is easily lost, intentionally and unintentionally, to other applications. TOB-TZWL-001

❑ **Always handle returned errors.** TOB-TZWL-004

❑ **Do not handle unrelated exceptions unless absolutely necessary.** TOB-TZWL-009

❑ **Avoid collapsing multiple errors into one value.** TOB-TZWL-011

❑ **Choose types that describe data semantics as closely as possible.** This helps turn runtime errors into compilation errors. TOB-TZWL-012

App Findings

#	Title	Type	Severity
1	Mnemonic is copied to clipboard	Data Exposure	High
2	Mnemonic stays in memory for too long	Data Exposure	Medium
3	Keychain security level is not the strongest available	Access Controls	Medium
4	Keychain operations are not checked for errors	Error Reporting	Low
5	Recovery phrase is displayed without a timeout	Data Exposure	Low
6	Pin code is sometimes not requested	Access Controls	Low
7	Wallet can be imported from more than 12 words	Data Validation	Medium
8	Too much code mocked	Testing	Informational
9	Sometimes all exception types are handled	Error Reporting	Informational
10	Function `encryptWithSalt` uses weaker hashing on exception	Cryptography	Low
11	`generateMnemonic` function delays error handling	Error Reporting	Informational
12	Wallet structure permits inconsistent state	Data Validation	Informational
13	Server lacks OCSP stapling	Cryptography	Medium

14	Sensitive information exposed to third-party SDK (Sentry)	Data Exposure	Medium
15	Multiple servers support weak TLS protocols and ciphers	Cryptography	Low
16	Servers do not restrict access over HTTP	Cryptography	Low

1. Mnemonic is copied to clipboard

Severity: High
Type: Data Exposure
Target: iOS and Android wallets

Difficulty: Medium
Finding ID: TOB-TZWL-001

Description
The wallets allow users to copy the mnemonic (i.e., the recovery phrase) to the clipboard, potentially leaking the mnemonic either by accident or due to a malicious application. Both wallets have a copy button in two views that display the mnemonic to the user. The first view is part of onboarding, and the second is an option in settings. The mnemonic is the most sensitive piece of data the wallet has, and both iOS and Android clipboard content can be read from another app. For instance, the official Google search application automatically reads clipboard content. Furthermore, the clipboard can now be shared between Apple devices, notably macOS, which further increases the possibility of exposure.

Exploit Scenario
A wallet user copies the recovery phrase. Another application reads from clipboard and sends the phrase to the Internet.

Recommendation
Short term, don't allow users to copy the mnemonic to the clipboard. Ensure users store the phrase by requiring them to retype it in a following view during onboarding.

Long term, avoid copying sensitive data to the clipboard.

References
- Can phone apps read my clipboard?
- These popular iPhone and iPad apps are snooping on data copied to the clipboard

2. Mnemonic stays in memory for too long

Severity: Medium
Difficulty: High
Type: Data Exposure
Finding ID: TOB-TZWL-002
Target: iOS and Android wallets

Description
Both iOS and Android wallets need to load the mnemonic to memory in plaintext in order to show it to the user and derive private keys for signing transactions. To minimize risk of exposure, the mnemonic should not reside in memory except when needed for wallet operation. Both wallets are keeping the mnemonic in memory longer than necessary.

On iOS, the mnemonic is saved in the keychain and fetched at the beginning of the program. The `AppDelegate` creates `DependencyManager.shared`, which in turn creates `AccountManager`, which holds a reference to the user's wallet containing the mnemonic. This keeps the mnemonic in memory all the time.

On Android, once the mnemonic is loaded into memory (e.g., “Show mnemonic in settings”), it stays on the heap until the app is sent to the background or killed. We verified this by taking a heap snapshot after making the program load the mnemonic into memory (see Appendix B).

Exploit Scenario
After a crash, an error-reporting service is called, leaking the program's heap along with the user's mnemonic.

Recommendation
Short term, fetch the mnemonic from the keychain only when absolutely needed, e.g., right before signing or when displaying it in settings. Clear out sensitive information from memory when it is no longer needed.

3. Keychain security level is not the strongest available

Severity: Medium
Type: Access Controls
Target: iOS keychain

Difficulty: High
Finding ID: TOB-TZWL-003

Description
iOS wallet's keychain storage relies on the default security level provided by the KeychainSwift library wrapping the native keychain API. The library's defaults are restrictive; however, they are not the strongest option available. Given the sensitivity of the data stored in the wallet's keychain, the highest security mechanisms available should be used. First, require user presence when fetching an item. Second, enforce a password that is set on the user's device; this option automatically prevents an item from being included in backups.

Additionally, the default security level is subject to change in the KeychainSwift library and could remain unnoticed by the wallet's code.

Exploit Scenario
Since there's no user presence checking in the keychain, an unauthorized user is able to sign transactions.

Recommendation
Implement the highest possible security level for keychain items, especially for the mnemonic. Pass the security level explicitly to the KeychainSwift library.

References
- Restricting Keychain Item Accessibility
- KeychainSwift—keychain item access

4. Keychain operations are not checked for errors

Severity: Low
Type: Error Reporting
Target: iOS Keychain

Difficulty: Undetermined
Finding ID: TOB-TZWL-004

Description
Keychain operations can fail for a number of reasons. All keychain operations return error codes which have to be checked to ensure an operation was performed correctly. In the wallet's code, the operations are executed through a wrapper library, KeychainSwift, which is also subject to failure. The code using KeychainSwift assumes that `get`, `set`, and `delete` operations are always successful, and does not check for errors.

Exploit Scenario
A wallet user unpairs a wallet, but the operation to delete the mnemonic from the keychain fails. The mnemonic stays in the keychain after the wallet is removed.

Recommendation
Short term, fix the code to check keychain errors and handle them appropriately.

Long term, always handle returned errors.

References
- Security Framework Result Codes

5. Recovery phrase is displayed without a timeout

Severity: Low
Type: Data Exposure
Target: Show recovery phrase views

Difficulty: Low
Finding ID: TOB-TZWL-005

Description
Wallets are able to show the recovery phrase to the user, and since there's no timeout for this view, once the phrase is shown, it stays uncovered indefinitely. A careful user could partially mitigate this issue by setting a global lockout timeout for the phone; however, the wallet should implement any available means to secure the mnemonic.

Exploit Scenario
A wallet user leaves the phone unattended with the recovery phrase view on. An attacker takes the victim's phone, reads the recovery phrase, and uses it later to take over the victim's wallet.

Recommendation
Add a timeout to the recovery phrase view that exits the view or hides the secret.

6. PIN code is sometimes not requested

Severity: Low
Type: Access Controls
Target: iOS wallet

Difficulty: Low
Finding ID: TOB-TZWL-006

Description
The PIN code mechanism is implemented to further protect the wallet from unauthorized use. However, the PIN code set by the user is not always requested, for instance, when an app returns from being backgrounded. To reproduce the issue:

1. Run the wallet on the iPhone SE (2nd generation) simulator.
2. Create a new wallet.
3. Skip Touch ID during onboarding.

Resetting the PIN code manually fixes the issue.

Exploit Scenario
An attacker with access to the phone is able to steal the wallet when the PIN code is not requested.

Recommendation
Short term, ensure the PIN code is requested. Test the mechanism on all target devices.

7. Wallet can be imported from more than 12 words

Severity: Medium
Type: Data Validation
Target: iOS and Android wallets

Difficulty: Low
Finding ID: TOB-TZWL-007

Description
A wallet can be created with more than 12 words as a mnemonic. The wallet address is generated according to the provided excess words and seems to be fully stored. The problem arises in the “Show recovery phrase” view in Settings. This view supports only up to 12 words, and only the first 12 words are shown, despite the mnemonic being longer. The issue happens in both Android and iOS.

Exploit Scenario
A wallet user creates the wallet with an 18-word mnemonic. Later, she writes down the 12 words shown in the wallet's settings and relies on them as a backup. The wallet is unrecoverable from the 12 words.

Recommendation
Short term, validate the number of words a user provides. Either lock the system to 12-word mnemonics only, or fully support a variable word count.

8. Too much code mocked

Severity: Informational
Type: Testing
Target: iOS MagmaWallet/Services/Mocks

Difficulty: Not applicable
Finding ID: TOB-TZWL-008

Description
Swift code depends on external services called over the Internet. To make tests reliable and reproducible, a separate set of implementations mimics the external functionality. Mocking, however, is done on a level of abstract services (`AccountService`, `ConseilService`, `TokenPriceService`, etc.) defined by the application rather than the actual external services such as `MessariAPI`, `TezosNode`, and `ConseilAPI`. The way current mocks are implemented makes it hard to test for error-handling code, and the additional logic is not present in mocks, but only in the real-world implementation.

Furthermore, some of the mocked functions call back to the real implementation, ensuring that certain preconditions are met so the external call won't happen. There is a risk of invalidating this precondition when developing the codebase.

Exploit Scenario
A mocked service is used for testing that doesn’t cover the production functionality, potentially introducing bugs.

Recommendation
Short term, mock the actual external services rather than abstract ones defined by the application. Re-implement only what's needed right at the edge of the system. Inject faulty scenarios and test error cases in addition to happy path testing.

9. Sometimes all exception types are handled

Severity: Informational
Type: Error Reporting
Target: Swift and Kotlin code

Difficulty: Undetermined
Finding ID: TOB-TZWL-009

Description
Swift and Kotlin code sometimes handle all exception types, but only the exceptions expected to be thrown should be caught. Catching all exception types can silence important errors and state corruption.

Exploit Scenario
An unrelated exception is handled, preventing the program from recovering from failure and making it crash in an unrelated place.

Recommendation
Short term, rethink and redesign error handling in the cases where the catch-all pattern occurs.

Long term, don't handle unrelated exceptions unless absolutely necessary.

10. encryptWithSalt uses weaker hashing on exception

Severity: Low | Difficulty: Hard
Type: Cryptography | Finding ID: TOB-TZWL-010
Target: `data/src/kotlin/.../smartwallet/data/core/security/CryptoUtils.kt`

Description
The `encryptWithSalt` function uses a `hash()` function when an exception occurs inside the `try` block. The `hash()` function is implemented as `SHA256` and provides weaker security for password hashing than `PBKDF2`.

```
fun encryptWithSalt(string: String, salt: String): String {
    try {
        val spec: KeySpec = PBEKeySpec(string.toCharArray(), salt.toByteArray(), STRENGHT,
128)
        val factory: SecretKeyFactory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1")
        val hash = factory.generateSecret(spec).encoded
        return hash.toHexString()
    } catch (e: Exception) {
        val ciphered = string.addSalt(salt)
        return ciphered.toByteArray().hash().toHexString()
    }
}
```

Figure 10.1: data/src/main/kotlin/io/camlcase/smartwallet/data/core/security/CryptoUtils.kt#L41-51.

Exploit Scenario
A wallet developer uses `encryptWithSalt`. An exception occurs, and the caller doesn't know that a weaker version of the hashing function was used.

Recommendation
Short term, change `encryptWithSalt` to signal failure through the return type or an exception instead of falling back to a weaker hash function.

11. `generateMnemonic` function delays error handling

Severity: Informational | Difficulty: Undetermined
Type: Error Reporting | Finding ID: TOB-TZWL-011
Target: `JavaTezos/src/main/.../javatezos/crypto/mnemonic/MnemonicFacade.kt`

Description
The `generateMnemonic` function from `MnemonicFacade` returns `null` instead of throwing an exception due to an invalid argument (Figure 11.1). This is not recommended, as a caller might propagate the `null` return value to other parts of the program and the nullability has to be dealt with, possibly in multiple remote places, even though the root cause was already known before calling the function.

Additionally, the function loses error information by catching `TezosError` without details. The `toMnemonic` function can throw two distinct exceptions that would be collapsed into a `null` value inside the `catch` block (Figure 11.1). The function's documentation suggests that returning `null` can occur only if the strength is not valid, which is true when it is not a multiple of 4. This description also doesn't include the value 0 for which the second exception is thrown (Figure 11.2). While in this example it is unlikely that someone will pass 0 to the function, and debugging won't be hard, the general pattern should be avoided.

```
/**
 * Generate a mnemonic of the given strength in english.
 * Process is heavyweight, recommended to launch this in a computation thread.
 * @param strength This must be a multiple of 4.
 * @return list of mnemonics, null if strength is not valid
 */
fun generateMnemonic(strength: Int): List<String>? {
    val array = ByteArray(strength)
    Random.Default.nextBytes(array)
    return try {
        mnemonicCode.toMnemonic(array)
    } catch (e: TezosError) {
        null
    }
}
```

Figure 11.1: JavaTezos/src/main/kotlin/io/camlcase/javatezos/crypto/mnemonic/MnemonicFacade.kt.

```
public List<String> toMnemonic(byte[] entropy) throws TezosError {
    if (entropy.length % 4 > 0)
        throw new TezosError(TezosErrorType.MNEMONIC_GENERATION, null, null,
                new IllegalArgumentException("Entropy length not multiple of 32 bits."));

    if (entropy.length == 0)
        throw new TezosError(TezosErrorType.MNEMONIC_GENERATION, null, null,
                new IllegalArgumentException("Entropy is empty."));
```

Figure 11.2:
JavaTezos/src/main/java/io/camlcase/javatezos/crypto/mnemonic/MnemonicCode.java#L151.

Exploit Scenario
A wallet developer calls `generateMnemonic`, which returns `null`. The value is propagated to other parts of the program, and the root cause of the issue is lost, which makes it harder to debug issues related to the mnemonic being `null`.

Recommendation
Short term, remove the `try ... catch` block from the `generateMnemonic` function and change the return type to `List<String>`.

Long term, avoid collapsing multiple errors into one value.

12. Wallet structure permits inconsistent state

Severity: Informational | Difficulty: Undetermined
Type: Data Validation | Finding ID: TOB-TZWL-012
Target: `JavaTezos/src/main/kotlin/io/camlcase/javatezos/model/Wallet.kt`

Description
The `Wallet` structure (Figure 12.1) from Kotlin Tezos SDK permits the mnemonic to be nullable; however, in practice it is not possible because constructors ensure it is always present. The Android wallet also expects the mnemonic to be present and does redundant checks to ensure it.

```
data class Wallet(
    override val publicKey: PublicKey,
    val secretKey: SecretKey,
    val address: Address,
    val mnemonic: List<String>? = null
) : SignatureProvider {
```

Figure 12.1:
JavaTezos/src/main/kotlin/io/camlcase/javatezos/model/Wallet.kt#L38-43.

```
fun getMnemonics(): Single<List<String>> {
    return walletLocalSource.getWalletOrDie()
        .map {
            val mnemonics = it.mnemonic
            if (mnemonics == null || mnemonics.size < 12) {
                throw MagmaError(ErrorType.NO_WALLET)
            }
            mnemonics
        }
}
```

Figure 12.2:
data/src/main/kotlin/io/camlcase/smartwallet/data/usecase/user/GetSeedPhraseUseCase.kt#L34-43.

Exploit Scenario
A developer using the SDK incorrectly assumes the mnemonic is present and adds more redundant code to handle the `null` possibility.

Recommendation
Short term, change the `Wallet` structure to always include the mnemonic field.

Long term, choose types that describe data semantics as closely as possible. This helps turn runtime errors into compilation errors.

13. Server lacks OCSP stapling

Severity: Medium | Difficulty: High
Type: Cryptography | Finding ID: TOB-TZWL-013
Target: `https://api.exchangeratesapi.io/ & https://data.messari.io/`

Description
The affected servers do not return its SSL certificate's revocation status via *OCSP Stapling*. This feature provides clients with the ability to detect whether the server's SSL certificate has been revoked.

Apple recommends that OCSP Stapling should be implemented on all mobile endpoints. This implies that OCSP Stapling will become a requirement for iOS Apps on the App Store.

Exploit Scenario
Alice gains access to a server using its shared wildcard SSL certificate, compromising the certificate's private key. Even if Tezos revokes the compromised certificate, clients will continue to allow connections to any server with the certificate—even ones hosted by Alice—because there is no OCSP Stapling.

Recommendation
Short term, update all Magma servers and mobile clients to enable support for OCSP Stapling.

Long term, perform certificate revocation exercises to ensure that the protections are sufficient, as well as to train staff on how to react to a compromised SSL credential.

References
- Apple WWDC 2017: Your Apps and Evolving Network Security Standards
- Apache SSL/TLS Strong Encryption: OCSP Stapling

14. Sensitive information exposed to 3rd Party SDK (Sentry)

Severity: Medium
Type: Data Exposure
Target: Sentry SDK in iOS and Android

Difficulty: High
Finding ID: TOB-TZWL-014

Description
The Sentry SDK on iOS and Android collects a substantial amount of, possibly, sensitive information about the user, the app, and their interactions with it. This may include Nonpublic Personal Information (NPI), Personally Identifiable Information (PII), hardware trackers, geolocation, and other data types which are covered by GDPR and CCPA regulatory regimes.

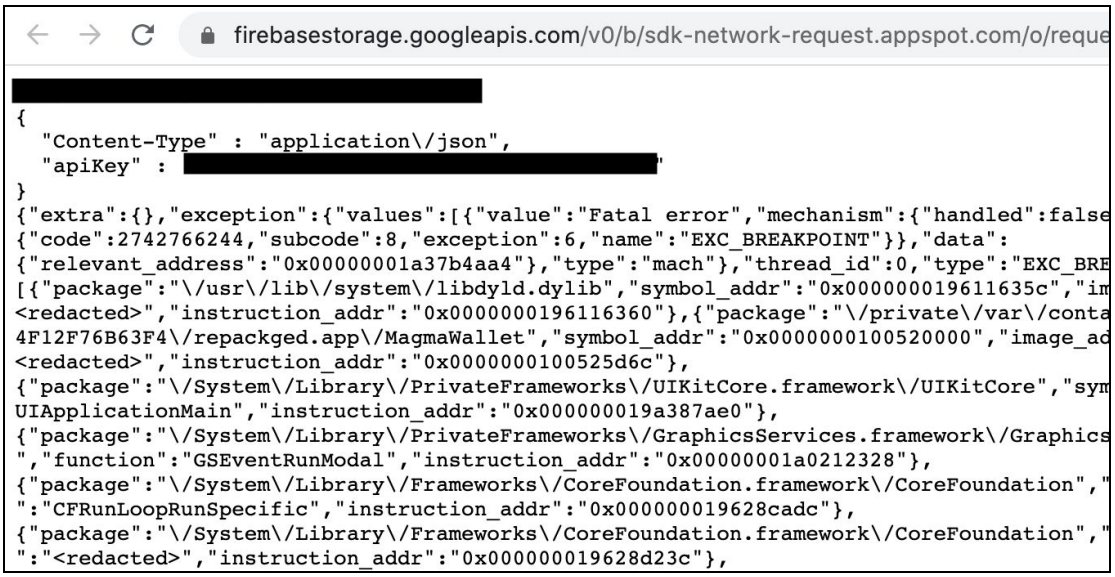

Figure 14.1: An example of the raw data collected by Sentry from an app user

Furthermore, the Sentry SDK collects three additional classes of data:

- On iOS, the Carrier ID, including the device's cellular network, carrier name, and country were sent to Sentry.
- On iOS, the Device Manufacturer, OS Version, Board ID, Device Platform, Device Type, and Device Architecture were sent to Sentry.
- On Android, the `Android_ID` was sent to Sentry.

Notably, the Android_ID is a 64-bit number that is randomly generated when the user first sets up their device. It remains constant for the lifetime of the device. Collecting this ID has significant privacy implications since it uniquely identifies the device and does not allow users to opt-out in any reasonable manner

Data sent to 3rd parties that may include user NPI (Nonpublic Personal Information) or PII (Personally Identifiable Information) must be documented in the app's privacy policy.

Exploit Scenario
GDPR complaints result in a fine levied against the app. In cases of less severe infringement, this could result in a fine of up to €10 million or 2% of the firm's worldwide annual revenue from the preceding year, whichever amount is higher.

Recommendation
Re-evaluate whether this SDK is necessary for production use of the application. If so, take steps to limit its data collection through configuration and then document the data sharing activity in the app privacy policy. If the SDK is not required, remove the SDK.

Regarding the Android_ID, Google recommends using the Android Advertising ID, available via Google Play Services, to identify the device. More information about the Advertising ID and possible approaches to identifiers is available in the Android "Best Practices for Unique Identifiers" guide. These approaches provide developers with a simpler, future-proof API that affords users greater control, including the ability to reset the ID or opt-out of using it.

15. Multiple servers support weak TLS protocols and ciphers

Severity: Low
Type: Cryptography
Target:

Difficulty: High
Finding ID: TOB-TZWL-015

- `https://conseil-prod.cryptonomic-infra.tech:443/`
- `https://api.exchangeratesapi.io/`
- `https://mytezosbaker.com/`
- `https://data.messari.io/`
- `https://tezos-prod.cryptonomic-infra.tech:443/`

Description
The indicated servers support TLS 1.0, a legacy protocol found vulnerable to numerous attacks, and weak ciphers with key lengths fewer than 56 bits, a key strength vulnerable to brute force attacks.

TLS 1.0 has been found vulnerable to a litany of issues since its release, most notably the POODLE vulnerability discovered by Google in 2014. Flaws have affected the protocol's key exchange and encryption schemes that it supports. The vast majority of these attacks are summarized in RFC 7457.

Most mobile apps negotiate a stronger version of the protocol (such as TLS 1.2) and stronger ciphers, making this issue difficult to exploit. However, vulnerable endpoints may be exposed through protocol downgrade attacks or attacks such as FREAK.

Furthermore, TLS v1.0 is no longer acceptable for PCI compliance as stated in Appendix A2 of the PCI DSS Requirements.

Exploit Scenario
An attacker with access to a compromised internet routing infrastructure performs a protocol downgrade attack on users of the Magma wallet. They force connections to various servers into weakened TLS protocols and ciphers, which allow them to brute force and ultimately decrypt these communications offline. This facilitates stealing cryptocurrency from the affected users.

Recommendation
Disable TLS v1.0 across all the App's endpoints, and require TLS 1.2 or later. Note, TLS v1.0 will prevent legacy clients from connecting to the endpoints.

Use the Mozilla SSL Configuration Generator in Intermediate mode or higher to reconfigure the affected servers.

Additionally, regularly monitor and validate the deployed configurations using tools such as SSLyze or SSL Labs.

References

- RFC 7457: Summarizing Known Attacks on Transport Layer Security (TLS)
- FREAK (or 'factoring the NSA for fun and profit')
- Mozilla SSL Configuration Generator
- SSL Server Test

16. Servers do not restrict access over HTTP

Severity: Low
Type: Cryptography
Target:

Difficulty: High
Finding ID: TOB-TZWL-016

- `https://api.exchangeratesapi.io`
- `https://mytezosbaker.com/`
- `https://conseil-prod.cryptonomic-infra.tech:443/`
- `https://tezos-prod.cryptonomic-infra.tech:443/`
- `https://data.messari.io/`

Description
The HTTP Strict Transport Security (HSTS) header allows the server to inform clients, such as browsers or mobile apps, that they should never access the server over HTTP. All requests to access the site using HTTP should be converted to HTTPS requests instead.

HSTS is supported by all recent browsers and most mobile networking APIs (such as NSURLSession on iOS). Setting the HSTS header will ensure that Magma is always accessing these endpoints over HTTPS.

Exploit Scenario
An attacker performs a downgrade attack on Magma users that forces connections over an unencrypted network communication. The Magma app does not detect this downgrade and connects to the unencrypted endpoint. The attacker collects information that facilitates stealing cryptocurrency.

Recommendation
Set the HTTP Strict Transport Security header in server responses to further protect the Magma's network traffic, and ensure that Magma will connect to the APIs over HTTPS only.

The following HTTP header can be added to the endpoint's responses for this purpose:

```
Strict-Transport-Security: max-age=10886400
```

Consider securing connections to the server endpoints further by:

- Enabling HSTS for all subdomains using the `includeSubDomains` directive.
- Adding the endpoint to the HSTS preload list, which will automatically enable HSTS for the endpoint in most recent browsers.

This can be achieved by deploying the following HSTS header:

```
Strict-Transport-Security: max-age=10886400; includeSubDomains; preload
```

References

- HSTS Preload List Submission

- MDN: Strict-Transport-Security

iOS Platform Findings

#	Title	Type	Severity
17	Application data remains exposed after first unlock	Data Exposure	Medium
18	Disable third-party keyboards	Data Exposure	Medium
19	TLS certificates are not pinned for multiple domains	Cryptography	Medium
20	Sensitive data is archived in iCloud and iTunes backups	Data Exposure	Medium
21	iOS pasteboard data does not have an expiration date	Data Exposure	Low

17. Application data remains exposed after first unlock

Severity: Medium
Type: Data Exposure
Target: Magma Wallet iOS Client

Difficulty: High
Finding ID: TOB-TZWL-017

Description
The Data Protection entitlement is not enabled in the Xcode project for the App. As a result, the default Data Protection class is used for files which only encrypts data until the user unlocks their device for the first time since booting it. After unlocking their device, application data will remain unprotected and exposed regardless of whether the user locks their phone.

Exploit Scenario
An attacker steals a user's phone. They use one of the following methods to extract sensitive data from the locked device:

- Exploit a vulnerability that executes code on the device or within the app.
- Use a lock screen bypass, of which there have been several in iOS.
- Use a pairing record from a computer that was previously paired via iTunes.

Files that are not protected can be recovered even if the device is locked.

Recommendation
Enable the data protection entitlement in Xcode. It is configurable under the "Capabilities" tab. The Data Protection capability will automatically protect all files created by the App with the strongest Data Protection class, NSFileProtectionComplete. This protection class ensures that the file is encrypted with the user's passcode whenever the device is locked.

Note that this entitlement will cause iOS to encrypt all the app's files when locked, which may cause issues if the app expects to access its files in the background. If this is necessary, use the `NSFileProtectionComplete-UntilFirstUserAuthentication` class for only the specific files that must be accessed in the background.

References

- Apple: Data Protection Entitlement
- Apple: Encrypting Your App's Files

18. Disable third-party keyboards

Severity: Medium
Type: Data Exposure
Target: Magma Wallet iOS Client

Difficulty: Low
Finding ID: TOB-TZWL-018

Description
The App does not disable custom keyboards. Such keyboards were introduced in iOS 8 and allow users to install a custom keyboard that can be used in any App, in order to replace the system's default keyboard.

Custom keyboards are not leveraged when the user types into a "Secure" field (such as password fields) but can potentially log all the user's keystrokes for regular fields. If the App requires the user to enter sensitive data (such as credit card numbers) in a field not marked as "Secure", allowing a custom keyboard to be used may increase the risk of such data being leaked.

Exploit Scenario
Alice creates a custom keyboard that Bob uses. Alice's keyboard can silently exfiltrate all of Bob's keystrokes in the Magma app.

Recommendation
Short term, disable third-party keyboards within the Magma iOS client to prevent leakage of sensitive data entered by the user. This can be achieved by implementing the `application:shouldAllowExtensionPointIdentifier:` method within the client's `UIApplicationDelegate`.

Long term, stay abreast of changes to iOS that might permit data exfiltration from the client.

References
- Apple Developer Documentation: UIApplicationDelegate

19. TLS certificates are not pinned for multiple domains

Severity: Medium
Type: Cryptography
Targets:
Magma Wallet when connecting to:

- `https://data.messari.io/`
- `https://api.mytezosbaker.com/`
- `https://api.exchangeratesapi.io/`

TezosKit when connecting to:

- `https://tezos-prod.cryptonomic-infra.tech:443/`
- `https://conseil-prod.cryptonomic-infra.tech:443/`
- `https://mytezosbaker.com/`

NSErrorFromSentryError when connecting to:

- `https://sentry.io/`

Difficulty: High
Finding ID: TOB-TZWL-019

Description
Magma will accept SSL certificates signed by any Certificate Authority (CA) installed within the device's trust store for the indicated domain names.

As part of an attack, this could allow an entity in control of a CA private key (such as a government or anyone that compromises a certificate authority) to perform a man-in-the-middle attack against any HTTPS traffic initiated by the target code locations.

Furthermore, users can be tricked or required (e.g., by their employer's policies) to install new CA certificates in the device trust store, increasing the potential for compromise.

Exploit Scenario
An attacker compromises the Indian Controller of Certifying Authorities (Indian CCA) and mints a new certificate for mytezorsbaker.com. They use a privileged network position to intercept and modify network traffic to the Magma mobile app and exploit users that rely on authenticated communications with this server.

Recommendation
Implement SSL pinning by embedding the server (or CA) certificate or public key for the App's server endpoints inside the app. The app should ensure that the server's certificate chain contains the pinned key or certificate.

Consider using TrustKit to pin certificates. If TrustKit cannot work, modify the code the validate the SSL certificate sent by the server is the same as the certificate hard-coded in the app. Consider using the URLSession:didReceiveChallenge:completionHandler method as part of the NSURLSessionDelegate protocol.

Alternatively, SSL pinning can be implemented by modifying the identified code locations to validate that the SSL certificate sent by the server matches a certificate hard-coded within the App. Consider using the URLSession:didReceiveChallenge:completionHandler method as part of the NSURLSessionDelegate protocol to validate certificates.

References

- TrustKit makes it easy to deploy SSL public key pinning and reporting
- Apple: URLSessionDelegate

20. Sensitive data is archived in iCloud and iTunes backups

Severity: Medium
Type: Data Exposure
Target accounts:

- `io.camlcase.tezos.wallet.mnemonic`
- `io.camlcase.tezos.onboarding.status`
- `io.camlcase.tezos.pincode`
- `io.camlcase.tezos.biometrics.type`

Difficulty: Low
Finding ID: TOB-TZWL-020

Description
Magma does not explicitly exclude sensitive data from device backups, which will allow sensitive data to get exported to iTunes and iCloud backups. Apple and any attackers with access to a user's iTunes or iCloud backup will have access to a user's private data.

Exploit Scenario
Alice gains physical access to Bob's phone and knows his passcode. She initiates a backup of Bob's phone to iTunes from which she is able to extract all sensitive keychain data.

Alternatively, Mallory identifies Magma user email addresses, then uses a previously disclosed password database to guess their current iCloud passwords. She retrieves iCloud backups that contain sensitive Magma keychain data from a large number of users.

Recommendation
Short term, explicitly set a `ThisDeviceOnly` accessibility class (such as `kSecAttrAccessibleWhenUnlockedThisDeviceOnly`) for all keychain items. This should prevent keychain data from being migrated to iTunes and iCloud backups.

Long term, empirically validate that no sensitive data is stored to a backup of the Magma Wallet iOS application. Consider uniform usage of a wrapper, such as Square's Valet, to simplify storage and retrieval of data from the keychain.

References

- Apple Developer Documentation: Keychain Services
- Square Valet

21. iOS pasteboard data does not have an expiration date

Severity: Low
Type: Data Exposure
Target:

- `MagmaWallet.ReceiveViewController`
- `MagmaWallet.RecoveryPhraseDisplayViewController`

Difficulty: Low
Finding ID: TOB-TZWL-021

Description
The vulnerable target code locations use the UIPasteBoard API to write data into the device's global pasteboard without configuring an expiration date for the data.

Data stored in the pasteboard is accessible to any app on the device that is brought to the foreground. Pasteboard data does not expire by default. Data will be accessible in the pasteboard until it is overwritten.

Exploit Scenario
Magma writes sensitive data into the pasteboard. The user opens their TikTok application, which steals the contents of the pasteboard and sends it to locations unknown.

Recommendation
As of iOS 10, `UIPasteBoard` items can be assigned an expiration date: at the configured time and date, iOS will remove the pasteboard items from the pasteboard. Hence, it is recommended that any item saved into the pasteboard be given an expiration date.

References

- Apple: UIPasteboard

Android Findings

#	Title	Type	Severity
22	App does not restrict the use of unencrypted HTTP	Cryptography	Medium
23	Enable verification on Android Security Provider	Cryptography	Medium
24	Enable Android Verify Apps	Configuration	Low
25	App views are vulnerable to tapJacking	Authentication	Low
26	Enable SafetyNet Attestation API	Configuration	Low
27	App susceptible to App Links hijacking	Authentication	Low

22. App does not restrict the use of unencrypted HTTP

Severity: Medium
Type: Cryptography
Target: Magma Wallet Android Client

Difficulty: Low
Finding ID: TOB-TZWL-022

Description
Magma has not enabled Android N's Network Security Configuration feature, which forces the use of encryption (HTTPS) for all the app's connections. Cleartext HTTP traffic is not blocked and it cannot be ensured that all data sent by Magma is encrypted.

Connecting to a server over cleartext exposes user data to attackers on the network, logging appliances, ISPs, etc. Furthermore, browsers have started aggressively banning plaintext traffic altogether to force servers to support HTTPS.

Configuring Magma to opt-out of cleartext traffic ensures that connections initiated by the app, and any embedded SDK, will be protected at all times. This also ensures there are no regressions back to plaintext traffic when new versions are released. For example, if a new SDK is added that uses plaintext HTTP to transport sensitive data, the Network Security feature will prevent the data exposure before it happens.

Additionally, the Network Security feature may be required by the Google Play Store in the near future, similar to how Apple is requiring all iOS Apps to disable cleartext HTTP traffic via App Transport Security.

Exploit Scenario
Magma decides against using the Sentry SDK and replaces it with a new debugging helper, “Warden.” The Warden SDK is configured to send all data about the user’s device over HTTP, exposing it to many intermediate routing devices. Magma does not notice this condition and ships the app in this state.

Recommendation
Enable and configure a Network Security Configuration to prevent the app or any embedded SDK from initiating dangerous cleartext connections.

As of API level 23, add the following attribute to the `<application>` element in the manifest:

```
android:networkSecurityConfig="@xml/network_sec_config"
```

The `network_sec_config.xml` file must then contain the app’s Network Security Configuration. Add the following line to the configuration file to opt-out of cleartext traffic for all domains. This policy will disable cleartext communication by default. The policy will

apply to all the App's connections, except for WebViews and select connections initiated using the java.net.Socket API which do not honor the `cleartexttrafficpermitted` setting:

```
<base-config cleartextTrafficPermitted="false"></base-config>
```

If the app tries to open a plaintext connection with such policy, the following exception will be returned:

```
java.io.IOException: Cleartext HTTP traffic to domain.com not permitted
```

If a domain accessed by the app is only accessible over HTTP, add an exemption:

```
<domain-config cleartextTrafficPermitted=”true”>
<domain>example.com</domain>
</domain-config>
```

References

- Google: Network security configuration

23. Enable verification of Android Security Provider

Severity: Medium
Type: Cryptography
Target: Magma Wallet Android Client

Difficulty: Medium
Finding ID: TOB-TZWL-023

Description
Android client does not explicitly check whether it is running on a device that has an up-to-date Android Security Provider.

The Security Provider is responsible for providing secure network communications, such as SSL/TLS. Running Magma Wallet on a device with an outdated Security Provider exposes it to network attacks. For example, it can allow an attacker on the network to decrypt and compromise SSL/TLS traffic.

Exploit Scenario
A new vulnerability discovered in Android can be exploited to produce a man-in-the-middle attack (similar to CVE-2014-0224). Bob has not upgraded his phone to include the latest version of the Android Security Provider to mitigate this vulnerability, and his SSL traffic to the Magma API server can be snooped and modified.

Recommendation
Short term, on every app startup, run `ProviderInstaller.installIfNeeded()` supplied by Google Play services. This method will ensure that the Android Security Provider is up to date. If the Security Provider remains out of date or an error occurs, this method will throw an exception and Magma should decline to run.

Long term, continue adding anti-tamper and security update protections to the Magma Wallet clients.

References
- Android Developer Documentation: ProviderInstaller.installIfNeeded()
- Update your security provider to protect against SSL exploits

24. Enable Android Verify Apps

Severity: Low
Type: Configuration
Target: Magma Wallet Android Client

Difficulty: Medium
Finding ID: TOB-TZWL-024

Description
Magma does not check that Android's "Verify Apps" mechanism is enabled on the device.

Verify Apps checks for Potentially Harmful Apps (aka malware) on the device. It will review each App during installation and monitor App behavior afterward to block harmful activity.

Android users commonly downloaded and sideload apps outside of Google Play. They may be downloaded from the internet, or installed via third-party App Stores, particularly in countries where Google Play is not available (for example, in China).

Apps installed via such methods have not been vetted by Google. Verify Apps fills this gap by providing a similar level of security review for third-party applications. Verify Apps ensures that the third-party apps are not harmful and will not attack Magma.

Exploit Scenario
The user has unknowingly installed malware (that would be detected by Google) on their device. This malware app attacks the Magma wallet to steal cryptocurrency from the user.

Recommendation
Ensure that the user's device has enabled Verify Apps when Magma starts. If Verify Apps has not been enabled, redirect the user to the Security settings of the device to enable Verify Apps.

References
- SafetyNet Verify Apps API

25. App Views are Vulnerable to TapJacking

Severity: Low
Type: Authentication
Target: Magma Wallet Android Client

Difficulty: Hard
Finding ID: TOB-TZWL-025

Description
Magma does not defend against TapJacking attacks. In a TapJacking attack, the attacker hijacks the user's taps and tricks the user into performing an action the user did not intend.

Exploit Scenario
A malicious third party App installed on the device creates an overlay on top of Magma. When the naive user taps on the overlay, which can show any user interface to trick the user, the taps are passed on to Magma.

Recommendation
Use Android's touch filter to defend against TapJacking. There are multiple ways to enable touch filtering:

1. Call setFilterTouchesWhenObscured(True) to discard touches when another visible window obscures the view's window.
2. Set the `android:filterTouchesWhenObscured` layout attribute to "true" to discard touches when another visible window obscures the view's window.
3. Implement `onFilterTouchEventForSecurity()` to filter the touch event to apply security policies.
4. Check for the `MotionEvent.FLAG_WINDOW_IS_OBSCURED` flag in the `onTouch()` handler of the view.

References
- Secure Code Warrior: TapJacking

26. Enable SafetyNet Attestation API

Severity: Low
Type: Configuration
Target: Magma Wallet Android Client

Difficulty: High
Finding ID: TOB-TZWL-026

Description
The SafetyNet Attestation API is not checked by the Magma Android client.

Google Play provides the SafetyNet Attestation API for assessing the safety of the device that apps are running on. The API uses software and hardware information to provide a cryptographically signed attestation about the overall integrity of the device.

The SafetyNet Attestation API is capable of handling devices that have passed Compatibility Test Suite (CTS) certification and devices that have not (via the `basicIntegrity` parameter).

Exploit Scenario
Alice is using a phone that has been rooted and has malware on it. The modifications to her device would be detected by Google through the SafetyNet Attestation API, however, the Magma app does not check it before allowing Alice to transact in cryptocurrency.

Recommendation
Short term, use the SafetyNet Attestation API to assess the integrity and safety of the user's device. Configure the Attestation API to use the `basicIntegrity` parameter to support devices that have not passed CTS certification.

Long term, require an affirmative `ctsProfileMatch` result which indicates that the user is in possession of a device that passed CTS certification. Devices without a CTS certification possess unknown security risks and increase the likelihood that the device has been compromised.

References
- Android Developer Documentation: SafetyNet Attestation API
- Inside Android's SafetyNet Attestation: Attack and Defense

27. App susceptible to App Links hijacking

Severity: Low
Type: Authentication
Target: `camlcase.io`

Difficulty: Hard
Finding ID: TOB-TZWL-027

Description
Magma allows redirecting users from a web link to the app (e.g., a link opened in the browser). However, automatic verification is not enabled and domains claimed by the App can be hijacked by any other App on the device.

Exploit Scenario
Malicious apps can trick users into loading the wrong app. If sensitive information is passed via App Links, the malicious App could get access to this data.

Recommendation
Android M (6.x) introduced automatic verification for App Links, which allows an App to prove ownership of specific domains in order to configure Android to always route those URI requests to the App, preventing other Apps from hijacking the domain names.

Enable automatic verification of App Links by doing the following:

1. On each domain, the domain's relationship with the App must be proven by hosting a Digital Asset Links JSON file with the name `assetlinks.json` at a specific URL.
2. The `android:autoVerify="true"` attribute must be added to the intent filter that claims the domains within the App's Manifest.

Once automatic verification has been configured, it will be enabled on any device running Android 6.x or above. Automatic verification will not be enforced on older devices.

References

- Handling Android App Links

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking, or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important
Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal implications for client

High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses in order to exploit this issue

B. Android Heap Dump

This appendix shows the heap dump commands from an Android device using Android SDK Platform-Tools. This can be used to check if secrets still reside in memory.

```
# find a PID of the app
$ ./adb shell ps | grep <fragment of app's identifier>

$ ./adb shell am dumpheap <PID> /data/local/tmp/heapdump
File: /data/local/tmp/heapdump
Waiting for dump to finish...

$ ./adb pull /data/local/tmp/heapdump
/data/local/tmp/heapdump: 1 file pulled, 0 skipped. 80.1 MB/s (33728243 bytes in 0.401s)

$ strings heapdump | grep elephant
p{"passphrase":"","mnemonic":"wet, flat, thumb, glue, bike, trade, marine, laugh, shop,
prevent, elephant, melt"}#
{"passphrase":"","mnemonic":"wet, flat, thumb, glue, bike, trade, marine, laugh, shop,
prevent, elephant, melt"}!
wet, flat, thumb, glue, bike, trade, marine, laugh, shop, prevent, elephant, melt!
wet, flat, thumb, glue, bike, trade, marine, laugh, shop, prevent, elephant, melt!
elephant"
elephant"
wet flat thumb glue bike trade marine laugh shop prevent elephant melt!
wet flat thumb glue bike trade marine laugh shop prevent elephant melt#
wet flat thumb glue bike trade marine laugh shop prevent elephant melt!
wet flat thumb glue bike trade marine laugh shop prevent elephant melt!
11 elephant!
wet flat thumb glue bike trade marine laugh shop prevent elephant melt#
wet flat thumb glue bike trade marine laugh shop prevent elephant melt!
wet flat thumb glue bike trade marine laugh shop prevent elephant melt!
wet flat thumb glue bike trade marine laugh shop prevent elephant melt#
wet flat thumb glue bike trade marine laugh shop prevent elephant melt!
wet flat thumb glue bike trade marine laugh shop prevent elephant melt!
wet flat thumb glue bike trade marine laugh shop prevent elephant melt#
wet flat thumb glue bike trade marine laugh shop prevent elephant melt!
wet flat thumb glue bike trade marine laugh shop prevent elephant melt!
wet flat thumb glue bike trade marine laugh shop prevent elephant melt!
elephant!
elephant!
```

Figure B.1: Dumping the Android application heap.

C. Non–Security-Related Findings

JavaTezos/src/main/kotlin/io/camlcase/javatezos/crypto/CryptoUtils.kt#L29-31
The name "`EllipticalCurve`" is not a correct cryptographic term. "`EllipticCurve`" should be used.

```
enum class EllipticalCurve {
    ED25519
}
```

JavaTezos/src/main/kotlin/io/camlcase/javatezos/crypto/mnemonic/EnglishMnemonic.kt#L49
There is a formatting issue in the mnemonics list: Two of them are written in a single line.

```
"adapt",
"add", "addict",
"address",
```

G. Fix Log

Tezos addressed the following issues in their codebase as a result of our assessment, and each of the fixes was verified by Trail of Bits. The reviewed code is available in PR 67 and PR 69 for iOS, and commits with the label "audit" for Android.

ID	Title	Severity	Status
01	Mnemonic is copied to clipboard	Medium	Fixed
02	Mnemonic stays in memory for too long	Medium	Fixed
05	Recovery phrase is displayed without a timeout	Low	Fixed
06	PIN code is sometimes not requested	Low	Fixed
07	Wallet can be imported from more than 12 words	Low	Fixed
09	Sometimes all exception types are handled	Informational	Not fixed
10	`encryptWithSalt` uses weaker hashing on exception	Low	Fixed
14	Sensitive information exposed to third-party SDK (Sentry)	Medium	Fixed
17	Application data remains exposed after first unlock	High	Fixed
18	Disable third-party keyboards	Low	Fixed
20	Sensitive data is archived in iCloud and iTunes backups	Low	Fixed
21	iOS pasteboard data does not have an expiration date	Low	Fixed
22	App does not restrict the use of unencrypted HTTP	Medium	Fixed
23	Enable verification of Android Security Provider	Medium	Fixed
24	Enable Android Verify apps	Medium	Fixed
25	App views are vulnerable to tapJacking	Low	Fixed
27	App susceptible to app links hijacking	Low	Fixed

Detailed Fix Log

This section briefly describes fixes made post-assessment and reviewed by Trail of Bits.

Finding 1: Mnemonic is copied to clipboard
Resolved. In the Android source code, this finding was resolved by removing the functionality that allowed users to copy their mnemonic to the clipboard. In iOS, it is not possible to copy the mnemonic from the recovery screen.

Finding 2: Mnemonic stays in memory for too long
Rosolved. In the Android source code, this finding was resolved by using `RxJava` to clear objects containing the mnemonic right after use. This finding was also resolved for iOS by sanitizing variables and labels that contained the mnemonics when leaving the recovery screen.

Finding 5: Recovery phrase is displayed without a timeout
Resolved by adding a 30-second timeout to the passphrase recovery screen.

Finding 6: PIN code is sometimes not requested
Resolved by setting the onboarding status on the last screen of the onboarding process, forcing the application to ask for a PIN after a user successfully is onboarded to the application.

Finding 7: Wallet can be imported from more than 12 words
Resolved for iOS and Android by allowing the mnemonic display logic to load more than 12 words.

Finding 9: Sometimes all exception types are handled
Not fixed. In the Android source code, efforts were made to reduce the number of exception types handled, and logging logic was added when errors are thrown.

No changes were made to the iOS source code to remediate this issue.

Finding 10: `encryptWithSalt` uses weaker hashing on exception
Resolved by throwing an error in the catch block of the `encryptWithSalt` function rather than falling back on the insecure hashing function.

Finding 14: Sensitive information exposed to third-party SDK (Sentry)
Resolved by adding a function that removed the sensitive data. The solution also removes the `Android_ID` value as discussed by the developer in this issue submitted to the Sentry repository.

Finding 17: Application data remains exposed after first unlock
Resolved by adding `NSFileProtectionComplete` protections in `MagmaWallet.entitlements`.

Finding 18: Disable third-party keyboards
Resolved by disabling third-party keyboards in the `AppDelegate.swift` file.

Finding 20: Sensitive data is archived in iCloud and iTunes backups
Resolved by setting the relevant keys with the `accessibleWhenUnlockedThisDeviceOnly` access property.

Finding 21: iOS pasteboard data does not have an expiration date
Resolved by setting an expiry date of 60 seconds for the pasteboard in `MagmaWallet/Modules/Home/ReceiveViewController.swift` for the function `copyButtonTapped`.

Finding 22: App does not restrict the use of unencrypted HTTP
Resolved by adding `network_security_config.xml` with the setting `<base-config cleartextTrafficPermitted="false" />`, which prevents the application from communicating without encrypted HTTP.

Finding 23: Enable verification of Android Security Provider
Resolved by calling `ProviderInstaller.installIfNeededAsync` in the balances screen.

Finding 24: Enable Android Verify Apps
Resolved by adding checks for "Verify Apps" and enabling this feature if not already enabled.

Finding 25: App views are vulnerable to tapJacking
Resolved by setting `android:filterTouchesWhenObscured = true` to all screen layouts.

Finding 27: App susceptible to App Links hijacking
Resolved by disabling App Links.