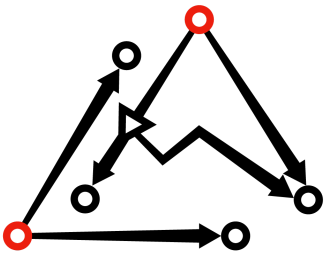

Least Authority

PRIVACY MATTERS

Mina Signer SDK + StakingPower Wallet

Security Audit Report

Mina Foundation

Final Audit Report: 21 September 2021

Table of Contents

Overview

Background

The Mina Foundation requested that Least Authority perform a security audit of the Mina Signer SDK and the StakingPower Wallet.

Project Dates

- **June 28 - July 30**: Code review *(Completed)*
- **August 4**: Delivery of Initial Audit Report *(Completed)*
- **September 16 - 17**: Verification Review *(Completed)*
- **September 21**: Final Audit Report delivered *(Completed)*

Review Team

- Bryan White, Security Researcher and Engineer
- Jan Winkelmann, Cryptography Researcher and Engineer
- Mirco Richter, Cryptography Researcher and Engineer
- JR, Cryptography Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Mina Signer SDK + Staking Power Wallet followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

The following code repositories are considered in-scope for the review:

- Mina Signer SDK: https://github.com/crackerli/ffi_mina_signer
- StakingPower Wallet: https://github.com/crackerli/coda-mobile-wallet.git

Specifically, we examined the Git revisions for our initial review:

Mina Signer SDK: `f92d67d3daa7e6fe376985e62427855a6b305a83`

StakingPower Wallet: `941b04458d6c3a988d988e63b2c4739a6cedb179`

For the verification, we examined the Git revision:

Mina Signer SDK: `bab3b807fee04b43b5a004bb005c59051172f320`

StakingPower Wallet: `1265a03722d8ec2aef749557a13f5857ce85cee3`

For the review, these repositories were cloned for use during the audit and for reference in this report:

Mina Signer SDK: https://github.com/LeastAuthority/Mina-Signer-SDK

StakingPower Wallet: https://github.com/LeastAuthority/Mina-StakingPower-Wallet

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third party code, unless specifically mentioned as in-scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- Mina StakingPower Wallet figma (*shared via email with Least Authority on 23 June 2021*)
- Mina Signer SDK README.md: https://github.com/crackerli/ffi_mina_signer/blob/null-safety/README.md
- StakingPower Wallet README.md: https://github.com/crackerli/coda-mobile-wallet/blob/master/README.md

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation and adherence to best practices;
- Exposure of any critical information during user interactions with the blockchain and external libraries, including authentication mechanisms;
- Adversarial actions and other attacks that impact funds, such as the draining or the manipulation of funds;
- Mismanagement of funds via transactions;
- Vulnerabilities in the code, as well as secure interaction between the related and network components;
- Proper management of encryption and storage of private keys, including the key derivation process;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

The Mina StakingPower Wallet is a mobile wallet application that enables users to generate a key pair for sending, receiving, and staking on the Mina Network. The Mina StakingPower Wallet relies on the Mina Signer SDK for cryptographic functionality. Both the Mina StakingPower Wallet and the Mina Signer SDK were the key components in scope for our security audit.

System Design

Our team conducted a broad and comprehensive review of the StakingPower Wallet and Mina Signer SDK's system design. It is clear that security has been strongly considered and we commend the security due diligence efforts by the development team. In addition to instances of adherence to security best practices, we detail our findings below, including several issues and suggestions.

StakingPower Wallet

The StakingPower Wallet implements basic security safeguards for protecting the user, by which the mnemonic seed is encrypted before being persisted to disk. A helpful and informative User Interface (UI) dialog has been implemented, warning users before the mnemonic phrase is viewed and prompts users to take security precautions.

User Interface Security Considerations

In investigating key pair generation and wallet setup, we found that there are no constraints preventing users from choosing insecure passwords. If the mnemonic seed is compromised, all keys derived from it are compromised, allowing an attacker full control over all accounts derived from that mnemonic seed. As a result, we recommend that secure password constraints be implemented and that users are shown password strength estimations (Issue A).

State Management

We examined the BLoC state management system used in the state management of the wallet. While this is a complex system, it is an industry standard for state management of applications with sophisticated UI state and behavior. Similarly, we examined the handling of network requests and responses and did not identify any issues.

In instances where secrets are available in memory, `Navigator` arguments (see `Navigator.pushNamed`) are used to persist the mnemonic in cleartext across the onboarding routes during initial wallet creation (see recovery_phrase_screen.dart:102 and encrypt_seed_screen.dart:L116). In addition, the mnemonic is sent in cleartext over the event bus. It remains unclear whether this is considered a safe practice since this may be the appropriate way to perform such operations. An attacker would need to be able to monitor memory to observe the bus. However, we searched for instances in which `bloc.listen()` would be called generically and logged, as this would dump all instances of the mnemonic passing over the event bus, but we did not identify any instances of this. More generally, we did not identify any issues in the implementation.

Dependencies

The StakingPower Wallet uses the HTTP library, dio, in addition to the standard Dart HTTP package, to make TLS encrypted connections to staking provider and market information services (specifically, Figment, Nomics, and Staketab (API)). The `fromMap` methods on each class of response object have been implemented such that only expected and relevant response data are considered by the application.

Furthermore, we checked for instances where the application opens external resources and found that the `url_launcher` Flutter plugin is used to open URLs provided manually by the Staketab vendor. A staking provider website value could open and pass data to other installed applications via Intents on Android and Universal Links on iOS. As a result, we recommend adding a warning for end users notifying that they are navigating to an untrusted resource and the associated risk, or that the development team filter provider website URL values those which use specific, known schemes (Issue B).

Mobile Device Security

Our team considered the general security model of a mobile device and found that physical access and rogue applications represent a substantial threat, particularly in cases where a victim is unaware that an attack has succeeded or been attempted. In the case of a rogue application, it must be installed on the same device on which the wallet application is installed. Once installed, a root exploit is applied for that specific device, if one exists, or once one has been made available. In the case of an attacker having physical access to the device and is able to breach biometric or password user access controls (as a result of poor security configuration or social engineering), the attacker could enable the installation of such a rogue application.

We found that the mnemonic phrase is exposed to other applications on the device through the use of Clipboard. We suggest that the user be required to write down the mnemonic and that Clipboard access to the application/mnemonic be disabled in order to avoid potential exploits (Issue E).

Furthermore, an attacker with a rogue application installed on the device may be able to record the wallet screen, which may contain security-critical information, such as the mnemonic or private information (e.g.

addresses, transaction history, and others). We recommend that the measures outlined in this report be taken in order to mitigate this vulnerability (Issue H).

The `shared_preferences` Flutter plugin is used to persist application data to disk, such as account(s) information, current network ID, and the encrypted mnemonic seed phrase. The account metadata is being stored unencrypted, including the user's account balance, address(es), and staking provider address(es). In addition to encrypting the mnemonic seed, we recommend encrypting all persisted data (except for the salt, which is required for decryption). This would improve security against an attacker gaining control of the filesystem (Issue C). Furthermore, in examining persistence of wallet data on iOS, `NSUserDefaults` (and therefore `shared_preferences`) may not be an optimal choice for persistence on iOS. Thus, we recommend alternatives that provide storage-level encryption (Suggestion 3).

The wallet application implements no Root Detection or Certificate Pinning. Jailbroken or rooted iOS devices weaken the security of the device itself and can lead to excessive application permissions regarding other applications' data. In these situations, any stored secrets on the device would be considered insecure.To prevent the wallet application from opening on a device that has been rooted or jailbroken, we suggest that Root Detection be implemented (Suggestion 7). Furthermore, we suggest Certificate Pinning, which prevents the application from being accessed by proxies that could result in encrypted traffic being decrypted by well-positioned attackers (Suggestion 8).

Finally, in looking at what intent-filters the Android build implements, we found that only the ACTION_MAIN action with the CATEGORY_LAUNCHER category is included. This is standard practice and indicative that no intent-based attack surface is present.

Mina Signer SDK

Key Management

In our review of the Mina Signer SDK Flutter plugin, we examined how private keys are secured and found that a Dart BIP39 implementation is used to generate a mnemonic seed phrase for use in a BIP44 wallet. The generated mnemonic phrase is converted to its seed format and then encrypted using AES-256 in CBC mode. The encryption key that is used is the SHA256 hash of a user-specified password, prepended with a random 64-bit salt.

The key derivation function implemented (PBKDF2) does not sufficiently adhere to recommended best practices. If an attacker can crack the Key Derivation function, they will be able to decrypt the seed key, allowing them to compromise the wallet. As a result, we recommend that the key derivation function be reconfigured to adhere to industry standards (Issue F). Furthermore, we note that PBKDF2 is not a sufficiently secure algorithm for the purpose of encryption key derivation and recommend that a memory-hard key derivation function be implemented (Issue D).

Encryption

The encryption algorithm used in Mina Signer SDK, AES in CBC mode, is not considered to be sufficiently secure. This may result in an attacker modifying the seed and scenarios that could lead to the loss of the key. As a result, we recommend that an alternative authenticated encryption be used (Issue G).

Code Quality

StakingPower Wallet & Mina Signer SDK

The StakingPower Wallet and Mina Signer SDK code bases are generally well organized and adhere to best coding practices. However, our team identified recommended areas for improvement. In the

StakingPower Wallet code base, we found that API keys are stored as constants, whereas best practice recommendations suggest populating them at compile-time (Suggestion 1).

Tests

Test coverage for both the StakingPower Wallet and the Mina Signer SDK is insufficient in that neither repository implements a test suite. We suggest that sufficient test coverage be implemented for success and failure cases, which helps to identify potential edge cases and helps protect against errors and bugs, which may lead to vulnerabilities or exploits. A test suite should include a minimum of unit tests and integration tests. End-to-end testing is also recommended so that it can be determined if the implementation behaves as intended (Suggestion 5).

Documentation

StakingPower Wallet

Our team was provided with a `README.md` and a graphic design document, which provided a helpful but limited overview of the StakingPower Wallet. We recommend that the wallet documentation be improved to include better details of the system architecture, the interaction between its subcomponents, interaction with third-party API's, setting up and running an Android simulator, and setting up a keystore (Suggestion 4).

Code Comments

We found the code comments to be sufficient in explaining the intended functionality of each component.

Mina Signer SDK

Our team found the Mina Signer SDK project documentation to be insufficient. We recommend comprehensive documentation providing a high-level description of the system, each of the components, and interactions between those components. This can include developer documentation, new developer onboarding documentation, and architectural diagrams. This allows an auditing team to assess the in-scope components and understand the expected behavior of the system being audited.

Code Comments

The code comments in the `mina_native_signer/crypto.c` file were particularly helpful in reviewing the Mina Signer SDK library. However, we identified an instance of inaccurate code comments and suggested scanning the code base for inaccurate code comments, which should be corrected or removed (Suggestion 6).

Scope and Dependencies

The scope of this security review of the StakingPower Wallet and Mina Signer SDK was generally sufficient. However, it was necessary to examine `shared_preferences` as well as its alternatives (Suggestion 3), in order to better evaluate security vulnerabilities related to data persistence, which was outside the scope of this review.

We note that the dependencies BLoC state management system and dio HTTP client library used, while generally known to be secure, increase the surface area for a potential vulnerability. However, these design choices might be driven by performance trade-offs. Furthermore, we note concerns with the use of the event bus to pass critical secrets, such as the mnemonic and passwords, however, we did not identify instances where this data ended up in the filesystem itself.

The dio HTTP client library is used to make API requests to external services for querying blockchain state. Specifically, the services Figment, Nomics, and Staketab are used. As such, the wallet falls in the

"API Wallet" category. While Mina technology could allow a wallet to validate the correctness of responses by these services, this functionality is not currently implemented. Therefore, at present, it is assumed that these parties can be trusted with supplying the correct information on the state of the chain.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: Mnemonic Encryption Password Has No Constraints [StakingPower Wallet]	Partially Resolved
Issue B: Opening Arbitrary URLs of Unknown Origin [StakingPower Wallet]	Resolved
Issue C: Account Metadata is Not Encrypted at Rest [StakingPower]	Resolved
Issue D: Key Derivation Function is Insecure [Mina Signer SDK]	Resolved
Issue E: Mnemonic Exposed Through Clipboard [StakingPower Wallet]	Unresolved
Issue F: Weak PBKDF2 Parameters [Mina Signer SDK]	Resolved
Issue G: Substandard Encryption Algorithm [Mina Signer SDK]	Resolved
Issue H: Wallet Screen Not Protected Against Screen Recording [StakingPower Wallet]	Resolved
Suggestion 1: Load API Keys at Compile-Time [StakingPower Wallet]	Resolved
Suggestion 2: Make Key and IV Argument to Encrypt Required [Mina Signer SDK]	Resolved
Suggestion 3: Reconsider Shared_Preferences for Persistence [StakingPower Wallet]	Resolved
Suggestion 4: Improve Documentation [StakingPower Wallet + Mina Signer SDK]	Unresolved
Suggestion 5: Increase Test Coverage [StakingPower Wallet + Mina Signer SDK]	Unresolved
Suggestion 6: Correct Inaccurate Code Comment [Mina Signer SDK]	Resolved

Suggestion 7: Implement Root Detection [StakingPower Wallet]	Partially Resolved
Suggestion 8: Implement Certificate Pinning [StakingPower Wallet]	Unresolved

Issue A: Mnemonic Encryption Password Has No Constraints [StakingPower Wallet]

Location

The following code execution path demonstrates the password confirmation text being passed directly through to the key derivation function without any validation:

`Mina-StakingPower-Wallet/lib/new_user_onboard/screen/encrypt_seed_screen.dart:74`

`Mina-Signer-SDK/lib/sdk/mina_signer_sdk.dart:33`

`Mina-Signer-SDK/lib/encrypt/crypter.dart:36`

Synopsis

There are no constraints on the user-chosen password (e.g., length, character diversity, non-dictionary words, etc.). This makes it possible for a user to choose a weak password, increasing the feasibility of compromise of the data encrypted with its derived key.

Impact

If the mnemonic seed is compromised, all keys derived from it are compromised. This means an attacker would have full control over (i.e., able to sign transactions on behalf of) all accounts derived from that mnemonic seed.

Preconditions

An attacker would have to obtain the `shared_preferences` `.xml` or NSUserDefaults `.plist` file.

On Android, SharedPreferences data is stored on the filesystem, which could be exfiltrated by a rogue application or an unauthorized user, given physical access to the device. On iOS, `NSUserDefaults` and `.plist` files are not accessible to rogue applications, except in the cases where the device has been jailbroken or rooted.

Feasibility

With the availability of powerful hardware like CPUs, GPUs, and FPGAs on the cloud, it is not difficult to brute force the key for various classes of weak passwords (e.g., low entropy, dictionary words, and others), particularly since SHA-256 is inexpensive on CPU resources (see Issue D).

Obtaining the shared preferences file, however, likely requires that the device has been successfully attacked previously. For example, as previously described, this could be done by means of physical access or a rogue application. Rogue applications have been historically identified in multiple application stores.

Technical Details

Mina uses hierarchical deterministic wallets according to BIP44 to structure their wallet data in a client-agnostic way.

`shared_preferences` data is protected by filesystem user access control only and is stored in plain-text XML.

Mitigation

We recommend adding messaging to the `EncryptSeedScreen` widget that informs users of password best practices, the risk of choosing a weak password, and the risks of using a rooted device.

Remediation

We recommend adding password strength estimation to the form validation and prevent users from choosing weak passwords.

Status

The StakingPower Wallet team has added the `password_strength` package as a dependency and integrated it into the UI to show the user a password strength estimation based on length, characters used, and a “top 10,000” dictionary.

Although this estimation is being done and presented to the user, it is not preventing the submission of weak passwords. In order to fully resolve the issue, we recommend that the StakingPower Wallet team prevent users from choosing weak passwords.

Verification

Partially Resolved.

Issue B: Opening Arbitrary URLs of Unknown Origin [StakingPower Wallet]

Location

The following code execution path demonstrates how these values are used from when they are received from Staketab to when they are passed into `url_launcher`:

`Mina-StakingPower-Wallet/lib/stake_provider/blocs/stake_providers_bloc.dart:42,47,48`

`Mina-StakingPower-Wallet/lib/stake_provider/blocs/stake_providers_entity.dart:6,60`

`Mina-StakingPower-Wallet/lib/stake_provider/screen/stake_providers_screen.dart:104,163,176,233`

`Mina-StakingPower-Wallet/lib/global/global.dart:63`

Synopsis

The Staketab API is used to retrieve a list of staking providers. The URLs from the website field are used as-is in the application's provider list as link destinations.

Impact

A staking provider website value could open and pass data to other installed applications via Intents on Android and Universal Links on iOS.

Preconditions

One or more provider website values returned from the Staketab APIs use a scheme, which is supported by an installed application's intent-filters or universal links, on Android or iOS, respectively.

Feasibility

Staketab claims to receive URLs from provider representatives, which they manually verify internally before committing to the database, making this unlikely.

Technical Details

The `url_launcher` Flutter plugin is used by the application to open these URLs.

On Android, `url_launcher` uses the Intent system. By default, it uses an implicit intent with the `Intent.ACTION_VIEW` action, depending on intent resolution to open the appropriate application (see intents-filters guide).

On iOS, `url_launcher` uses `UIApplication#openURL` at FLTURLLauncherPlugin.m:118. This means that URLs can open and pass data to other installed applications via universal links.

Mitigation

We recommend adding a warning for end users, which explains that they are navigating to an untrusted resource and the associated risks.

Remediation

We recommend filtering provider website URL values to, at most, those which use specific, known schemes (e.g. HTTPS).

Alternatively, we recommend setting the `forceWebView` and `forceSafariVC` options to `true`.

Status

The StakingPower Wallet team has implemented a warning dialog, which includes a warning message and the URL itself. This dialog makes it so that users must confirm that they wish to navigate to a given URL.

Verification

Resolved.

Issue C: Account Metadata is Not Encrypted at Rest [StakingPower Wallet]

Location

`my_accounts/screen/create_account_screen.dart:183`

`my_accounts/screen/edit_account_screen.dart:166`

`new_user_onboard/screen/encrypt_seed_screen.dart:82`

`wallet_home/screen/wallet_home_screen.dart:168`

Synopsis

With the exception of the mnemonic seed, data stored by the application using the `shared_preferences` plugin is not encrypted. This unnecessarily exposes sensitive metadata in the event that a device is compromised.

Impact

Account metadata may be leaked, resulting in de-anonymization of the target user as well as disclosing the number of accounts managed by the wallet, their addresses and balances, and with which staking provider(s) (if any) each is staked.

Preconditions

An attacker would have to obtain the `shared_preferences .xml` or NSUserDefaults `.plist` file.

On Android, SharedPreferences data is stored on the filesystem which could be exfiltrated by a rogue application or an unauthorized user given physical access to the device.

On iOS, `NSUserDefaults` and `.plist` files are not accessible to rogue applications, except in the cases where the device has been jailbroken or rooted.

Feasibility

Obtaining the `shared_preferences` file likely requires that the device has been successfully attacked previously (e.g. by means of physical access or a rogue application). Rogue applications have historically been identified in multiple application stores.

Technical Details

The `shared_preferences` Flutter plugin uses Android's SharedPreferences on Android and NSUserDefaults on iOS to persist data to disk on a per application basis. The wallet is storing all persisted data as JSON-encoded strings in an `.xml` or `.plist` file on Android or iOS, respectively.

Remediation

We recommend encrypting all persisted data (except for the salt).

Status

The StakingPower Wallet team has replaced their usage of the `shared_preferences` plugin with `flutter_secure_storage`, which ensures that all data stored by the wallet is encrypted at rest. Additionally, automated migration has been included for existing installations.

Verification

Resolved.

Issue D: Key Derivation Function is Insecure [Mina Signer SDK]

Location

`Mina-Signer-SDK/lib/encrypt/crypter.dart#L37`

`Mina-Signer-SDK/lib/encrypt/kdf/sha256_kdf.dart`

Synopsis

An AES-256 encryption key is derived from a user password for the purpose of encrypting the mnemonic seed. This key derivation function utilizes the SHA-256 hashing function, which is not a sufficiently secure algorithm for this purpose.

Impact

Leakage of all plaintext of an encrypted wallet is possible, including the wallet seed. This leads to the attacker being able to author transactions and steal funds.

Preconditions

The attack requires access to the encrypted wallet data (e.g. by reading the memory of a stolen device).

Feasibility

Cracking weak password hashes is a process that can be radically optimized (e.g. using FPGAs). While these require some know-how and upfront investment, they allow very high guess rates compared to their costs.

Technical Details

The SHA-256 hash function is, for multiple reasons, not an appropriate algorithm for deriving keys from passwords. Firstly, hash functions are not key derivation functions. They serve different purposes, even though key derivation functions usually are constructed from hash functions. Secondly, when deriving keys from passwords, memory-hard functions should be used, in order to make brute force attacks infeasible. Note that SHA-256 can be executed and parallelized not only on computers, but also on FPGAs, resulting in a significantly lower power per guess ratio, which in turn leads to lower operating cost per guess ratio. Memory-hard functions help because they can not be parallelized well on FPGAs, GPUs, or specialized hardware.

Moving the burden of choosing a password that is secure with a simple key derivation function like HKDF to the user ignores that many users will not follow best practices when selecting a password. Entering passwords on phones can be tricky and prone to mistyping, which incentivizes users not to choose good passwords, which may in turn lead to insecure wallets. Additionally, the time needed to correctly enter a sufficiently secure password for a regular key derivation function is much longer than the time needed for a more secure password hashing function to run, leading to a net slowdown of the user. A memory-hard function key derivation function is more secure if the user chooses a weak password.

Remediation

We recommend using the Argon2id key derivation function, with the memory parameter set to 64MB, parallelism set to 4, and iteration count to 3.

Status

The StakingPower Wallet team has updated the `MinaCryptor` class's `encrypt` and `decrypt` methods to use Argon2id for key derivation and XChaCha20Poly1305 for encryption by default. It falls back to the original method when decrypting for backwards compatibility.

Verification

Resolved.

Issue E: Mnemonic Exposed Through Clipboard [StakingPower Wallet]

Location

new_user_onboard/screen/recovery_phrase_screen.dart#L87

Synopsis

The mnemonic phrase used to derive the wallet keys is accessible to the rest of the applications on the device via Clipboard. Clipboard is a global object that is accessible across application security boundaries. Any applications that are watching Clipboard will be able to see the mnemonic when the user copies it to Clipboard.

Impact

With access to the mnemonic, an attacker would be able to clone the wallet and take over all of its assets.

Preconditions

An application monitoring Clipboard when the mnemonic is copied. This is not an unlikely scenario, as many applications are known to do this by design. As of iOS 14, iOS notifies users whenever another application accesses Clipboard. For an application to be targeting the mnemonics specifically, a user's phone would need to already have a malicious application targeting the StakingPower Wallet installed.

Feasibility

Clear APIs for viewing Clipboard are available for iOS and Android developers. Creating a Clipboard monitoring application would not be difficult. However, a more difficult task would be to compromise the phone of a Mina user. Additionally, if the user is using an iPhone with version of iOS >= 14, they would be notified when another application accessed Clipboard, alerting them that the mnemonic had been accessed.

Remediation

We recommend not allowing the mnemonic to be saved to Clipboard. Instead, we suggest requiring users to write down the mnemonic in a place off of the filesystem of their device.

Status

The StakingPower Wallet team has created an intermediary step that warns the user of the dangers involved with using Clipboard. In order to remediate this issue, we recommend preventing the ability for the user to copy to Clipboard, as the current approach transfers the risk to the user and leaves the issue unresolved.

Verification

Unresolved.

Issue F: Weak PBKDF2 Parameters [Mina Signer SDK]

Location

lib/encrypt/kdf/pbkdf2_kdf.dart

Synopsis

The parameter configuration of the PBKDF2 does not adhere to accepted standards. A SHA-1 HMAC is used with 100 iterations and returns a 32 byte key with 16 byte salt. Current OWASP recommendations

specify over 300,000 iterations using SHA-256 HMAC for FIPS-140 compliance. As of 2016, NIST recommends a minimum of 10,000 iterations. For reference:

- 5 Authenticator and Verifier Requirements: https://pages.nist.gov/800-63-3/sp800-63b.html#sec5
- Password Storage Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html
- Parameter Choice for PBKDF2: https://cryptosense.com/blog/parameter-choice-for-pbkdf2

However, even with sufficient iterations, PBKDF2 can be efficiently parallelized and therefore does not provide a strong protection against brute-force attacks.

Impact

If an attacker can crack the Key Derivation function, they will be able to decrypt the seed key, allowing them to compromise the wallet.

Preconditions

An attacker would need to have access to the encrypted seed key, as well as specialized hardware to launch the attack.

Mitigation

We recommend vastly increasing the iterations. Since the NIST recommendations are also old by modern standards, the current OWASP recommendation of 300,000 iterations is a more robust solution if PBKDF2 is used.

Remediation

We recommend replacing PBKDF2 with a memory-hard function (see Issue D).

Status

The StakingPower Wallet team has deprecated the use of PBKDF2 and replaced it with a memory hard key derivation function using `flutter_sodium`, using `flutter ffi`, Argon2id (v13) and XChaCha20Poly1305ietf. The parameters for Argon2id password hashing are a memory limit of 128M, 3 iterations, and an output length of 32, and parallelism set to 1.

Verification

Resolved.

Issue G: Substandard Encryption Algorithm [Mina Signer SDK]

Location

`encrypt/aes/aes_cbcpkcs7.dart`

Synopsis

The CBC mode of operation does not provide any authentication on its own. That means that it is malleable in that it is possible to change the ciphertext in such a way that it can decrypt to a different, valid-looking plaintext. A better approach would be to use authenticated encryption (e.g. libsodium's secretbox or AES with the GCM mode or operation).

Impact

The attacker could modify the seed and there are scenarios where this may lead to loss of the key. For example, if the attacker has physical access to the phone but is unable to crack the password, they may replace the parts of seed. The attacher then returns the phone and the owner transfers money to the

account. This results in the attacker knowing parts of the seed, which may result in making a successful guess of the rest of the seed.

Preconditions

An attacker would need to have access to the encrypted seed and the ability to change it.

Remediation

We recommend using an authenticated encryption algorithm like libsodium's secretbox or AES-GCM.

Status

The StakingPower Wallet team has replaced the AES-CBC encryption algorithm with XChaCha20Poly1305ietf, which is an authenticated cipher.

Verification

Resolved.

Issue H: Wallet Screen Not Protected Against Screen Recording [StakingPower Wallet]

Synopsis

On Android and some versions of iOS, the user or an application may record videos or individual frames from the StakingPower Wallet application.

The data displayed by the StakingPower Wallet is critical to both the security of the application and the privacy of the user. This data includes the mnemonic phrase, which is displayed during wallet key pair generation, and at any time after that in the backup screen. A leak of the mnemonic passphrase is a critical breach of the security of the wallet. Additionally, the wallet displays user account balance and transaction history, of which a malicious screenshot could leak private user data.

Private user data must be secured and protected from access by other applications running on the mobile device.

Impact

A malicious application accessing an image of the mnemonic phrase could result in loss of all wallet funds. In addition, account balance or transaction history leaks are a breach of user privacy.

Preconditions

The user must make a screenshot, and grant other applications access to the screenshot. Alternatively an application that has access to the contents of the screen (e.g. a screen recording application) would be required.

Feasibility

The feasibility of such an attack depends on the specific phone and operating system (OS).

On iOS 10 or newer, there is no API for an application to use to access the screen of another application, so this attack vector is not applicable. However, if the attacker plants a malicious application with access to the photo gallery on the device and the user makes a screenshot of the seed phrase or other private information, it is possible for the application to access it.

On Android, the MediaProjection API can be used to record the contents of the screen. The user has to explicitly consent to the access, and during the duration of the screen recording, an icon is shown. This

icon represents wireless transmission of the screen contents to a projector or ChromeCast and may not be immediately identifiable as a screen recording icon. Additionally, any application with access to the files of the user can access all screenshots.

Mitigation

We recommend that measures to prevent or mitigate the impact of screenshots be taken, as detailed below.

Android

On Android, we recommend setting FLAG_SECURE on the application window and ensuring that no private content is shown in other windows. For more information, we suggest referring to this blog post about vulnerabilities of weak screenshot protection. It appears in the past the techniques used to prevent screen capture in Flutter have occasionally failed, however, the easiest way to enable it may be the `flutter_windowmanager` package. However, we do recommend testing the setup on multiple devices and Android versions to verify that the packages function correctly.

iOS

On iOS, there is no API to prevent screenshots, which makes mitigating this issue more difficult. In some instances, the only thing that can be done is to remind the user that taking screenshots of the seed means making it available to other applications. In other cases, such as the seed generation screen, stronger measures could be taken. In this case, the iOS event UIApplicationUserDidTakeScreenshotNotification can be handled to display a notice to the user that the action is not secure, and to generate and display a new seed. With this approach, instead of preventing the screenshot of a seed that is used, using a seed that is screenshot is prevented. ScreenShieldKit also warrants mention in this case as a potential resource for preventing screenshots.

Status

The StakingPower Wallet team has implemented a mix of prevention and notification strategies to prevent unnoticed capturing of sensitive information.

Android

On Android, the client protects the contents of the screen showing sensitive information using FLAG_SECURE. While the screen can still be recorded using `adb` on some devices, this vector is usually not available to attackers.

iOS

On iOS, screenshots on such screens are detected and the user is warned.

Verification

Resolved.

Suggestions

Suggestion 1: Load API Keys at Compile-Time [StakingPower Wallet]

Location

`Mina-StakingPower-Wallet/lib/constant/constants.dart:6`

`Mina-StakingPower-Wallet/lib/constant/constants.dart:15`

Synopsis

`FIGMENT_KEY` and `NOMICS_KEY` are constant literals committed into version control. This appears like it could easily result in one or more contributors accidentally committing and publishing their API keys. This also leaves room for this to happen in production depending on the rigorousness of the release process.

Mitigation

We recommend loading API keys via environment to the Dart/Flutter ecosystem features such as compile-time variables.

Status

The StakingPower Wallet team has moved API keys into build scripts, which are excluded from version control using command `options –dart-define` to pass these compile parameters and `String.fromEnvironment` in Dart to read from a corresponding environment variable.

Verification

Resolved.

Suggestion 2: Make Key and `IV` Argument to Encrypt Required [Mina Signer SDK]

Location

`Mina-Signer-SDK/lib/encrypt/aes/aes_cbcpkcs7.dart:8`

Synopsis

The signature for `AesCbcPkcs7#encrypt` allows for the `key` and `iv` arguments to be null. In the case that either is `null`, the implementation assigns a `Uint8List` of length 1 containing the value zero to it. In the case where either key or iv is `null`, the security of the resulting value is silently weak (as opposed to throwing an exception). In the case where both are `null`, the resulting value can simply be reversed, exposing the plaintext input.

Mitigation

We recommend making `key` and `iv` arguments required for encryption.

Status

The StakingPower Wallet team has deprecated the AesCbcPkcs class's encrypt and decrypt methods as encryption and decryption are now performed using XChaCha20Poly1305 provided by the `flutter_sodium` plugin.

Verification

Resolved.

Suggestion 3: Reconsider `shared_preferences` for Persistence [StakingPower Wallet]

Synopsis

The `shared_preferences` documentation warns not to use it for storing critical data:

> "Data may be persisted to disk asynchronously, and there is no guarantee that writes will be persisted to disk after returning, so this plugin must not be used for storing critical data."

We suspect this warning has more to do with the behavior of NSUserDefaults than `Shared_Preferences` as the NSUserDefaults documentation states:

> "When you set a default value, it's changed synchronously within your process, and asynchronously to persistent storage and other processes."

Impact

Since the user is expected to keep a copy of the mnemonic, the impact of any data loss should be one of only temporary inconvenience for the user.

If the application is relaunched after a failed write of either the encrypted mnemonic seed or the account metadata, it will ask the user to create a new wallet or import from mnemonic.

Mitigation

We recommend considering alternatives which do not depend on NSUserDefaults. `Flutter_secure_storage`, for example, uses Keychain Services instead of NSUserDefaults on iOS. This would also improve the security of shared preferences data at rest (see Issue C).

Status

The StakingPower Wallet team has replaced direct usage of `shared_preferences` with `flutter_secure_storage` in response to Issue C.

Verification

Resolved.

Suggestion 4: Improve Documentation [StakingPower Wallet + Mina Signer SDK]

Location

`Mina-Signer-SDK/blob/master/README.md`

`Mina-StakingPower-Wallet/blob/master/README.md`

Synopsis

The existing project documentation was helpful in getting the application running, however, we found a number of areas where the documentation was insufficient and would benefit from improvement.

StakingPower Wallet

Our team was provided with a `README.md` and a graphic design document for the wallet application. Although helpful, we found this documentation insufficient at describing the details of the system architecture and interaction between its subcomponents. We found no documentation describing interaction with third-party API's, requiring assumptions to be made about creating accounts with third-parties to obtain API keys. Additionally, our team found no documentation on setting up and running an Android simulator, or on how to set up a keystore.

Mina Signer SDK

The Signer SDK contains code copied and modified from the official Mina C Reference Signer, but the documentation does not state which version of the commit was copied. In addition, there is no documentation of how the encryption is performed (i.e. how the keys are derived from passwords and what encryption scheme is used).

Mitigation

We recommend creating additional documentation detailing the StakingPower Wallet and Mina Signer SDK architecture. Additionally, we suggest adding documentation outlining how to set up a keystore and how to run the application in the Android simulator.

Status

The StakingPower Wallet team has responded that they intend to make documentation available in conjunction with publishing the Signer SDK package to the Dart software repository, pub.dev. As a result, the suggestion remains unresolved at the time of this verification.

Verification

Unresolved.

Suggestion 5: Increase Test Coverage [StakingPower Wallet + Mina Signer SDK]

Location

`Mina-Signer-SDK/tree/master/test`

`Mina-StakingPower-Wallet/tree/master/test`

Synopsis

The StakingPower Wallet and the Mina Signer SDK do not implement a test suite. As a result, testing is insufficient and should be expanded to cover both implementations.

A sufficient test suite that tests for success and failure cases helps to protect against errors and bugs. In addition, tests help to identify potential edge cases, which may lead to vulnerabilities or exploits. A test suite should include a minimum of unit tests and integration tests. End-to-end testing is also recommended so that it can be determined if the implementation behaves as intended.

Mitigation

We recommend implementing a test suite for the StakingPower Wallet and the Mina Signer SDK that sufficiently covers both implementations. In particular, testing for the StakingPower Wallet must include integration tests for an application with non-trivial presentation logic and multiple backend service integrations. Testing for Mina Signer SDK must check that native functions are being called correctly by their respective Dart wrapper functions, thus testing success and error cases.

Status

The StakingPower Wallet team has responded that they plan to implement additional unit tests after the initial security issues and suggestions are fully resolved and published. As a result, the suggestion remains unresolved at the time of this verification.

Verification

Unresolved.

Suggestion 6: Correct Inaccurate Code Comment [Mina Signer SDK]

Location

`encrypt/kdf/pbkdf2_kdf.dart#L12-L15`

Synopsis

In the `pbkdf2_kdf.dart` file, we identified an inaccurate comment that states, "If salt is not provided, a random 8-byte one will be generated". However, the code does not reflect this intention. Instead, only a single null byte is used for the salt. This comment is misleading as it indicates that a random salt will be provided, when instead a predictable one will be. This may prompt user's of the SDK to unknowingly build insecure applications.

Mitigation

We recommend correcting the comment to either indicate that a single null byte will be used for the salt if none is provided, or alter the code so that an 8 byte salt is generated using a Cryptographically Secure Pseudo Random Number Generator, such as the `SecureRandom` class from the pointycastle API. Alternatively, the salt could be made a required parameter and the comment removed entirely.

We also recommend checking that all comments are accurate and relevant.

Status

The StakingPower Wallet team has updated the comment to be accurate.

Verification

Resolved.

Suggestion 7: Implement Root Detection [StakingPower Wallet]

Synopsis

Rooted mobile devices compromise the security model of the device and expose user's data to exposure from malicious applications. For instance, on iOS, `NSUserDefaults` and `.plist` files are not accessible to rogue applications, except in the cases where the device has been jailbroken or rooted. If an Android device has an unlocked bootloader, they could boot the device into an arbitrary OS which would permit full access to all filesystems. Without any root detection functionality, the application would continue to function under these circumstances.

A common practice for applications that implement root detection is to perform checks on application startup, and then if any of the checks fail, then the application refuses to load. This prevents users from possibly exposing their wallets to malicious parties. While root detection bypasses do exist, they often require additional effort by the attacker.

Mitigation

We recommend implementing Root Detection and closing the application if the device is rooted.

Status

The StakingPower Wallet team has implemented the `flutter_jailbreak_protection` Flutter package that uses RootBeer for Android and DTTJailbreakProtection for iOS. The development team chose to allow the application to continue to work on a rooted device, but with a warning to the user of the risks involved. As a result, we consider the suggestion partially resolved at the time of this verification.

Verification

Partially Resolved.

Suggestion 8: Implement Certificate Pinning [StakingPower Wallet]

Synopsis

Certificate Pinning locks (or pins) a predefined public key to a host and prevents the application from being used with any proxies that could result in encrypted traffic being decrypted by well positioned attackers. This technique is often used in mobile applications to prevent eavesdropping or the use of intercepting proxies.

In situations where an intercepting proxy is between the application and the host, the proxy will provide a Certificate that is valid and an unsuspecting application will send HTTPS traffic to it. However, this intercepting proxy can then decrypt the messages before re-encrypting them and forwarding them to the target host. Certificate Pinning is a technique to prevent such attacks.

Note that there are Certificate Pinning bypasses that can turn this protection off, but they require that the application binary to be modified, or the device to be rooted.

Mitigation

Certificate Pinning is considered a best security practice for mobile applications. However, because Mina does not control the Figment infrastructure, this creates uncertainty for the best approach. Because Figment could rotate their keys at any time without notifying the development team, users of the Staking Power Wallet could be locked out of the application unexpectedly until the development team pins to a new certificate.

Because Figment's infrastructure is hosted in AWS and certificates are signed by the Amazon Root Certificate Authority (CA), one solution could be to verify that the certificate is for the Figment domain and has been signed by the Amazon Root CA. However, if Figment switches CAs unexpectedly, the same Denial of Service issue will occur.

We feel the solution to this problem will involve the development team assessing the pros and cons of potential issues against following best security practices

Status

The StakingPower Wallet team has responded that they will coordinate with the Figment team on finding a way to support some form of certificate pinning. Until then, certificate pinning will remain disabled. As a result, the suggested mitigation remains unresolved at the time of this verification.

Verification

Unresolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. Although we are a small team, we believe that we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.