Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Pentest-Report Silence Laboratories Silent Shard Mobile App, Web & Cloud 06.-07.2023

Cure53, Dr.-Ing. M. Heiderich, M. Pedhapati, B. Casaje, A. Kahla

Index

Introduction
Scope
Testing Methodology
WP1: Audits & pentests against Silent Shard Snap Android SDK & app
WP2: Source code audits & pentests against Silent Shard Snap web app UI / JS
WP3: Source code audits & pentests against Silent Shard Firebase Cloud Functions
Identified Vulnerabilities
SIL-03-009 WP1: Seed phrase leakage in mobile application memory (Low)
SIL-03-010 WP1: Seed phrase leakage via dynamic instrumentation (Low)
SIL-03-011 WP1: Sign request screen handles newlines incorrectly (Low)
SIL-03-012 WP3: IDOR in sendMessage's Cloud Functions API (Medium)
SIL-03-013 WP3: Valid JWT forgery containing arbitrary user IDs (High)
Miscellaneous Issues
SIL-03-001 WP1: Android application lacks root detection (Info)
SIL-03-002 WP1: Insecure v1 signature support on Android (Info)
SIL-03-003 WP1: Android config hardening recommendations (Info)
SIL-03-004 WP1: Potential user disruption via exported activity (Low)
SIL-03-005 WP1: Potential leakage via absent security screen (Info)
SIL-03-006 WP1: Potential phishing via StrandHogg 2.0 on Android (Info)
SIL-03-007 WP1: Android binary hardening recommendations (Info)
SIL-03-008 WP2/3: Multiple hardcoded credentials in source code (Low)
Conclusions

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Introduction

"Silent Shard is an MPC-based TSS complemented by cyber-physical proofs for much usable, secure, and truly decentralized support for digital wallets, exchanges and institutional asset enterprises."

From https://silencelaboratories.com/silent-shard/

This report - defined by the acronym *SIL-03* - offers detailed information regarding a Cure53 penetration test and source code audit against the Silent Shard Snap web and Android applications, as well as Firebase Cloud Functions.

Following the proposition from the Silence Laboratories Pte. Ltd. maintainers in May 2023, this audit was scheduled for completion across June and early July 2023. Specifically, a ten workday time frame was allocated and then fulfilled by four senior Cure53 testers between CW24 and CW26.

Considering the variety of components scrutinized for this project, Cure53 deemed grouping the facets into three Work Packages (WPs) apt for efficiency reasons. These were described as follows:

- **WP1:** Source code audits & pentests against Silent Shard Snap Android SDK & app
- **WP2:** Source code audits & pentests against Silent Shard Snap web app UI / JS
- **WP3:** Source code audits & pentests against Silent Shard Firebase Cloud Functions

Notably, due to the division of the originally requested scope, this report complements another corresponding project that focused on the Silent Shard Snap and codebase (see *SIL-02*).

Cure53 leveraged a host of supporting entities that were provided by the client in advance of the examinations, including sources, application builds, relevant documentation, and other assorted items. Similarly to *SIL-02*, this assignment complied with a white-box penetration testing methodology

Any outstanding preliminary procedures were completed in the week before the active analysis phase, as per usual for Cure53 audits (in this case, CW23 2023). These endeavors generally help to gain a complete understanding of the scope and negate any hindering factors that may otherwise affect the project.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

A dedicated and shared Slack channel was established to enable communications between the Silence Laboratories and Cure53 teams. All employees from both organizations that played an active role in this particular task were invited to partake in the ongoing conversations, which were generally smooth and efficient. This medium also facilitated the live reporting process, which essentially entailed divulging a selection of issues at the point of detection for immediate proaction by the Silence Laboratories team.

In relation to the findings, the testers' highly satisfactory coverage over the characteristics outlined in the three work packages yielded a total of thirteen. A lower proportion of five were categorized as security vulnerabilities, whilst the remaining eight were assigned to the *Miscellaneous Issues* section due to their reduced risk of exploitation.

At this point, in order to provide a comprehensive appraisal of the security posture, it is imperative to differentiate between the outcomes encountered for each work package, i.e. the Android application (WP1), web application UI and JS (WP2), and Firebase Cloud Functions (WP3).

Firstly, the WP1-related reviews against the Snap Android application revealed a noteworthy volume of non-impactful flaws that were all assigned a *Low* or *Informational* severity rating. These generally pertained to typical weaknesses or hardening guidance. As such, the Android application garnered a relatively strong impression, though opportunities for bolstering defense-in-depth could (and should) be integrated for enhanced security efficacy.

The lack of tangible vulnerabilities concerning the web app UI and JS attests to the substantial security paradigms already applied in this respect.

Conversely, Cure53's examinations of Firebase Cloud Functions revealed fewer findings in comparison, though these entail prominent threats due to the *Medium* and *High* severity implications. The latter of which is particularly pertinent considering the ability to forge valid JSON Web Tokens (JWTs) with arbitrary user IDs (see ticket SIL-03-013). Thus, one can strongly suggest initiating mitigation actions as soon as possible to nullify the involved risk.

To summarize, one can verify that the aspects scrutinized during this review offer robust security defense on the whole, despite the partial discrepancies observed between each work package. With this in mind, Cure53 believes that a first-rate industry standard can be achieved should the Silence Laboratories team adhere to the guidance offered throughout this report.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

From a structural viewpoint, the report is divided into a number of key chapters moving forward. Firstly, the scope, general setup, and utilized materials are all stipulated in the bullet points below. Following this, comprehensive information concerning the team's breadth of coverage and applied testing techniques is offered in the *Testing Methodology* segment.

Next, the report lists all findings in ticket format and by order of detection, starting with the *Identified Vulnerabilities* and culminating with the *Miscellaneous Issues*. All tickets proffer an advanced technical overview, a Proof-of-Concept (PoC) or steps to reproduce if required, relevant code excerpts and examples, and the suggested remediation approaches for the developer team to implement.

To close proceedings, the *Conclusions* chapter aims to consolidate the varying impressions gained throughout this test and provide a final estimation of the security posture exhibited by the Silent Shard Snap web UI and Android applications, as well as Firebase Cloud Functions.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Scope

- **Source code audits & penetration tests against Silent Shard app, web UI & cloud functions**
  - **WP1:** Source code audits & pentests against Silent Shard Snap Android SDK & app
    - **Sources:**
      - **Repository:**
        - *silentshard-android-for-metamask-snap-integration/-/tree/v1*
      - **Commit:**
        - c1bacc380aa031d7cc9becd6735dbfd62b84b6cc
      - **Documentation:**
        - **Relevant Files:**
          - *silentshard-android-for-metamask-snap-integration/-/blob/v1/README.md*
          - *silentshard-android-for-metamask-snap-integration/-/blob/v1/silentshardsdk/README.md*
        - **Commits:**
          - 3b83054400a443bc36b6f004c90358f1652f0986
          - 394ca0c882465c1e152e1dbed25513f6b05aeed5
      - **Build:**
        - **APK:**
          - *silentshard-android-for-metamask-snap-integration/-/releases*
        - **Commit:**
          - 3bb056dc7dafcb311f5cc2db3101d504cb1d128fc86d
  - **WP2:** Source code audits & pentests against Silent Shard Snap web app UI / JS
    - **Sources:**
      - **Repository:**
        - *shard-metamask-snaps/silentshardnewui/*
      - **Commit:**
        - f462638c29dfc31e0085bcfec92cedf550a12a84
  - **WP3:** Source code audits & pentests against Silent Shard Firebase Cloud Functions
    - **Sources**
      - **Repository:**
        - *shard-metamask-snaps/metamask-snap-backend/*
      - **Commit:**
        - a2559f6e9cfe32e59118787b1b6e6b1fc2ccb82d
      - **Documentation**:
        - *README.md of metamask-snap-backend/main*
  - **Test-supporting material was shared with Cure53**
  - **All relevant sources were shared with Cure53**

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Testing Methodology

The *Testing Methodology* passages break down the myriad techniques applied and consequential coverage achieved by the audit team against the focus targets stipulated in WP1, WP2, and WP3. A dedicated section for each work package is provided for ease of reference, should the client wish to ascertain the exact methods employed against the Shard Snap SDK, mobile app, web app, and Firebase Cloud Functions. In essence, Cure53 hopes that the information outlined forthwith attests to the rigorous efforts instigated by the auditors during the course of this engagement.

WP1: Audits & pentests against Silent Shard Snap Android SDK & app

Cure53 initiated procedural analysis by reviewing all content held in the Android Manifest file, which verified that the application fails to comprehensively mitigate hijacking attacks, permits debugging, and does not explicitly set *hasFragileUserData* to *false*.

Nonetheless, the application disallows backups, as validated by the *android:allowBackup="false"* setting. This prohibits a plethora of potential leak occurrences in edge-case scenarios.

The team also materialized a list of all exported components from the internal Android Manifest file information and invoked these activities with additional intents included and omitted. Here, the ability to crash the app by invoking an exported activity with a uniquely-crafted intent prior to user authentication was confirmed and documented in ticket SIL-03-004.

The application's behavioral traits were subjected to intense scrutiny, which raised a security concern regarding the absence of a security screen to protect data displayed by the app, as detailed in ticket SIL-03-005. Similarly, the app fails to integrate root detection in any form, as stipulated in ticket SIL-03-001. Finally, Cure53 noted that the wallet backup is saved in the SD card.

The auditors' undertakings in this area also revealed that the application can be executed on devices with SDK that are not supported, such as Android 5. Hence, any would-be attacker could leverage the insecure v1 signature support to achieve certain malicious goals, as discussed in ticket SIL-03-002.

The team operated the app similarly to any typical user whilst reviewing altered files in the app folder, which served to demonstrate whether any sensitive information had been written unencrypted to the device.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

These efforts confirmed that the app stores security-related wallet information, such as keys, in an encrypted file with a hardcoded generic key. An unencrypted file with Firebase authentication tokens was also detected.

To complement the aforementioned procedure, the logs generated by the application were stringently reviewed, though Cure53 could not observe any pertinent data in the associated records.

The testing team instigated a number of initiatives that aimed to intercept network traffic between the device and backend servers, including advanced techniques such as self-signed certificates, DNS spoofing, and similar. However, all of these attempts failed. The testing team also attempted to spoof requests to Firestore, though the DNS server was appropriately configured to block this behavior. In this area, Cure53 also sought to verify that requests were performed via an encrypted protocol, such as HTTPS.

Elsewhere, the consultants prioritized assessing Android Keystore content, though in actuality the verification was made that the application does not leverage it altogether.

The *silentshard-android-for-metamask-snap-integration* repository was scanned by applying a variety of static analysis tools and manual audit methods in tandem. In this respect, Cure53 observed that the application does not employ libraries to secure data at rest, such as *EncryptedSharedPreferences* for instance. The team also acknowledged the absence of beneficial code functionalities. Specifically, tapjacking protection (see SIL-03-003) and *FLAG_SECURE* for screenshot protection were both omitted from the configuration. Some related opportunities for hardening were also detected, as highlighted in ticket SIL-03-007.

The team strove to enumerate any instances of insecure error logging, hardcoded secrets in the source code, and vulnerable third-party dependencies. This raised a persistent connected fault, which is outlined in ticket SIL-03-006.

The pairing flow and communication between the dApp and device were evaluated to determine the capability for abuse or presence of logical defects. Positively, Cure53 confirmed that this process is resistant to tampering, considering that encryption protects the communication process between the two devices. Additionally, the recovery phrase and the key shares were generated using a secure random generator.

Cure53's fuzzing procedures against unexpected user input to the Firebase Cloud Functions were unfruitful in identifying any noteworthy findings, which attests to the resilient exception handling capabilities established by the Silence Laboratories team.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

In addition, the utilization of the libsodium library for cryptographic operations proved highly robust.

The export and import backup functionality was meticulously probed. Here, particular focus was placed on both the mechanism in general and the methods by which data synchronization with Google Password Manager was achieved.

By conducting memory dumps of the application in various states, Cure53 successfully uncovered a potential seed phrase leak following the wallet export process. Supporting advice on this shortcoming is offered in ticket SIL-03-009.

Finally, during the application's dynamic instrumentation phase, Cure53 determined that the seed phrase is leakable under certain circumstances, primarily due to the absence of runtime integrity checks. Moreover, the Android Keystore system should be adopted to protect private keys and other pertinent data, as stipulated above. Please refer to ticket SIL-03-010 for associated fix suggestions.

WP2: Source code audits & pentests against Silent Shard Snap web app UI / JS

To commence the WP2 assessment, the shared repositories were subjected to extended inspections to determine any erroneous usage of *dangerouslySetInnerHTML*[1], due to its frequent overuse and high propensity for XSS issues.

Since the ReactJS framework does not handle URLs assigned to the HTML anchor tags' *href* property, the source code was reviewed for any instances of this nature. Ultimately, no assignment was found and the URL was confirmed comprehensively user-controlled.

Forthwith, the provided source code was audited for DOM XSS-related weaknesses, including usage of *location.href*, *window.open*, and user-controlled URL parameters.

Notably, the team's *npm audit* of the React application did not reveal any significant findings, with the caveat of the minor vulnerability documented in the accompanying *SIL-02* report (see ticket *SIL-02-002*).

Furthermore, the audit team examined network requests - including requests to third-party services - and error logs for any erroneous wallet information exposure, though no correlating risk circumstances were identified.

[1] https://reactjs.org/docs/dom-elements.html#dangerouslysetinnerhtml

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Lastly for WP2, the *silentshardnewui* repository was scanned by leveraging a host of static analysis tools and manual audit approaches simultaneously. This yielded a detrimental behavior concerning the location of a hardcoded API key, as described in ticket SIL-03-008.

WP3: Source code audits & pentests against Silent Shard Firebase Cloud Functions

In relation to Cure53's efforts against the scope items defined in WP3, the *metamask-snap-backend* repository was subjected to a scanning process similar to other aforementioned repositories. Here, the audit team observed a hardcoded JWT token and a private key, as documented in ticket SIL-03-008.

The testing team's code analysis further corroborated that the application implements authorization security controls in a safe manner. Moreover, all requests were achieved via an encrypted protocol such as HTTPS, whilst all associated endpoints correctly utilize an auth middleware for authentication purposes.

All endpoints exposed by the Cloud Functions API - specifically *getToken, refreshToken*, and *sendMessage* - were systematically studied. This facilitated the discovery of two issues pertaining to an authentication bypass and partial read/write access to the Firebase database that permits valid JWT token forgery, as underlined in tickets SIL-03-012 and SIL-03-013 respectively.

Despite stringent undertakings, Cure53 was unable to detect any Google sign-in authentication flaws during the course of this exercise.

Access control configurations and security rules for Firestore were also appraised. Despite the lack of associated weaknesses, Cure53 did observe pertinent flaws related to usage of the Admin SDK that enabled full database privileges.

Cure53 finalized WP3 testing by reviewing JWT token handling and verifying whether only optimal, state-of-the-art algorithms were permitted. The component's propensity for trivial weaknesses, such as omitting verification of the signing algorithms, was also subjected to a validation process.

To close, the team positively acknowledged the JWT's limited expiration date, which was deemed appropriate and adhered to sound security paradigms. Generally speaking, best practice stipulates avoiding the implementation of enduring tokens in order to neutralize the impact of potential leakages.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Identified Vulnerabilities

The following section lists all vulnerabilities and implementation issues identified during the testing period. Notably, findings are cited in chronological order rather than by degree of impact, with the severity rank offered in brackets following the title heading for each vulnerability. Furthermore, all tickets are given a unique identifier (e.g., *SIL-03-001*) to facilitate any future follow-up correspondence.

SIL-03-009 WP1: Seed phrase leakage in mobile application memory (*Low*)

Cure53 confirmed that the seed phrase and key shares persist in memory following requests to export the wallet. This data may be preserved in memory for an extended duration, due to Android's behavior of retaining applications in memory until reclaimed. Subsequently, an attacker with physical access to the device may exploit this vulnerability by dumping the application memory. Additionally, the prevalence of publicly-known Android kernel faults and the high likelihood of users operating on unpatched Android devices expands the risk of malicious applications escalating their privileges to access sensitive information in the memory.

To mitigate this issue, Cure53 discourages retaining sensitive data in memory longer than absolutely required. The developer team should nullify any variables that hold the key shares and seed phrase post-utilization. Furthermore, usage of immutable objects such as strings for the purpose of storing sensitive information is generally suboptimal and unadvisable. Alternatively, a char array could be applied to store sensitive data since these can be explicitly overwritten, thereby minimizing the risk of data leakage. Here, one must stipulate that references to immutable objects may remain in memory until garbage collection occurs, even if they are removed or nulled. With this in mind, Cure53 recommends implementing application mechanisms to enforce garbage collection[2], which will ensure prompt removal of sensitive data from memory.

[2] https://kotlinlang.org/docs/native-memory-manager.html#enable-garbage-collection-manually

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

SIL-03-010 WP1: Seed phrase leakage via dynamic instrumentation *(Low)*

Testing verified a potential seed phrase leakage via dynamic instrumentation techniques, whereby an attacker with physical access to the device can intercept and manipulate the execution of the application at runtime, enabling them to retrieve the seed phrase. To provide one example, this behavior could occur by hooking the *com.silencelaboratories.silentshardsdk.internal.Utils.generateSeedPhrase* function and monitoring its return value.

To mitigate this issue, Cure53 recommends implementing runtime integrity checks[3] within the application to guarantee the integrity of the app's memory space. Additionally, one can advise leveraging the security features provided by Android Keystore[4] to encrypt the seed phrase upon generation and decrypt only in instances deemed absolutely fundamental for essential operations. Minimizing the attack surface via these approaches will substantially reduce the risk of seed phrase exposure.

One effective solution in this respect represents the freely available *DetectFrida*[5] library, which will enhance the efficacy of the application's anti-tampering scheme. Nonetheless, Cure53 must stipulate that suitably skilled and dedicated attackers may identify bypass methods, in spite of the proposed defensive measures.

SIL-03-011 WP1: Sign request screen handles newlines incorrectly *(Low)*

The observation was made that the Silent Shard mobile app lacks optimal handling for newlines that appear in the sign message request confirmation popup. Generally, a transaction sign request can contain a custom message to be signed. When this custom message is displayed on the mobile app, newline characters in the message shift the remaining confirmation dialog down.

However, the popup on the mobile app does not allow users to scroll down and peruse the rest of the content. Consequently, an attacker could send a transaction sign request with a malicious message to obfuscate the *reject* and *approve* buttons.

To mitigate this issue, Cure53 recommends ensuring that both extended and newline-inclusive messages are correctly handled in the app. This can be achieved by guaranteeing that messages containing a multitude of newlines do not hide the action buttons, as well as enforcing a definitive separation between the custom message and the remaining transaction information.

---

[3] https://bit.ly/runtime-checks
[4] https://developer.android.com/training/articles/keystore
[5] https://github.com/darvincisec/DetectFrida

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

SIL-03-012 WP3: IDOR in sendMessage's Cloud Functions API (*Medium*)

Cure53 verified that the */sendMessage* endpoint allows attackers to control the *collection name*, *docId,* and *data* of objects prior to insertion into Silent Shard's Firestore database. This behavior enables the ability to insert or modify arbitrary documents, including (but not limited to) the *users* and *backup* collections.

The audit team's procedures also confirmed that an adversary can read the entire data for an object that matches the *uid* included in their authorization header, by simply leveraging the check for the existence of the *backup_data* key in the document's data.

Affected file:
/metamask-snap-backend-master/functions/src/sendMessage.ts

Affected code:
```
const db = admin.firestore();

if (collection === "sign" && data.message && data.message.round==1) {
  await sendNotificationToUser(db,payload.uid,{notification: {
      title: "Transaction request",
      body: `Please approve this or deny`,
    },
  });
}

await db.collection(collection).doc(docId ?? payload.uid).set(data);
```

Steps to reproduce:

1. Ensure that the Silent Shard Browser Add-on is installed and paired with any dApp.
2. Retrieve the *Authorization* token from any authenticated request initiated to *us-central1-mobile-wallet-mm-snap.cloudfunctions.net* using a proxy of choice.
3. Perform the following cURL request. Note that **{{token}}** must be replaced with the value extracted in Step 1.

cURL request:
```
curl -i -s -k -X $'POST' -H $'Host: us-central1-mobile-wallet-mm-
snap.cloudfunctions.net' -H $'Authorization: Bearer {{token}}' -H
$'Content-Type: application/json' --data-binary
$'{\"docId\":\"1337\",\"collection\":\"arbitrary_collection\",\"data\":
{\"arbritary_data1\":\"arbritary_data2\"},\"expectResponse\":false}'
$'https://us-central1-mobile-wallet-mm-snap.cloudfunctions.net/sendMessag
e'
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

4. Observe that the object will be created in the collection specified.

To mitigate this issue, Cure53 recommends checking the user-controlled collection against a blocklist and disallowing the creation of new documents to collections such as *users*. To prevent collection rewriting, one can advise storing the document creating user’s *uid* in the document's data and validating whether the user's *uid* matches it. This would prevent a malicious attacker from rewriting other users' existing documents.

SIL-03-013 WP3: Valid JWT forgery containing arbitrary user IDs *(High)*

The observation was made that one can forge a valid pairing document containing an arbitrary *user_id* via the */getToken* endpoint by leveraging the vulnerability described in ticket SIL-02-002*.* This can be accomplished by adopting a known and valid *sign_public_key* with its associated *signature* during the creation of the malicious pairing document.

This document will subsequently be utilized by the backend to generate a valid JWT token, which an attacker could in turn leverage to impersonate the user with the given *uid*. To successfully instigate this attack, the prerequisite to possess the targeted victim's *uid* is needed, which a malicious dApp may be able to extract given that its value is static and associated with a user's device.

A number of plausible actions will be facilitated via this flaw, including reading and editing a victim's FCM token, reading the encrypted backup data, as well as triggering arbitrary *sendNotificationToUser* requests to the user's device.

Affected file:
/metamask-snap-backend-master/functions/src/getToken.ts

Affected code:
```
const db = admin.firestore();
const unSub = db
  .collection("pairing")
  .doc(pairing_id)
  .onSnapshot(
    async (snap) => {
      const pairingData = snap.data() as Pairing | undefined;
      if (pairingData) {
        [...]
        const signPublicKey = pairingData.sign_public_key;
        await _sodium.ready;
        try {
          _sodium.crypto_sign_verify_detached(
            _sodium.from_hex(signature),
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
    pairing_id,
    _sodium.from_hex(signPublicKey)
  );
} catch {
  [...]
}
const uid = pairingData.user_id;
const token_expiration = Date.now() + 2 * 60 * 60 * 1000;
const token = signJwt({
  uid,
  pairing_id,
  web_sign_public_key: signPublicKey,
  expiry: token_expiration,
});
```

Steps to reproduce:

1. Ensure that the Silent Shard Browser Add-on is installed and paired with any dApp.
2. Retrieve the *Authorization* token from any authenticated request initiated to *us-central1-mobile-wallet-mm-snap.cloudfunctions.net* using a proxy of choice.
3. Perform the following cURL request. Note that **{{token}}** must be replaced with the value extracted in Step 1.

   cURL request:
   ```
   curl -i -s -k -X $'POST' -H $'Authorization: Bearer {{token}}' -H
   $'Content-Type: application/json' --data-binary
   $'{\"docId\":\"3y38ibmODR1GgmZ9N7q\",\"collection\":\"pairing\",\"data\":
   {\"is_backed_up\":true,\"backup_data\":\"true\",  \"created_at\":
   1688347519574,  \"expiry\":
   100000000000,  \"sign_public_key\": \"df56361304c2d4552c9533f3f2d92b18c87
   3adb57530537bd5ed96ecb194dafb\",  \"user_id\": \"arbritary_userID\",  \"p
   hone_enc_public_key\": \"1\",  \"device_name\": \"1\",  \"backup_data\":
   \"1\"},\"expectResponse\":true}' $'https://us-central1-mobile-wallet-mm-
   snap.cloudfunctions.net/sendMessage'
   ```

4. Perform the following cURL command, then check the JWT that will be generated in the response, which will be valid and assigned to the *arbritary_userID uid*.

   cURL command:
   ```
   curl -i -s -k -X $'POST' -H $'Content-Type: application/json' --data-
   binary
   $'{\"pairing_id\":\"3y38ibmODR1GgmZ9N7q\",\"signature\":\"e19a257453927e1
   083856b83d9f1a8caad0565fd3e2590e261bc95b97330e6e64d13ee85bf93fee545f8df3e
   b460905d32844d469160d7c7e553877d4ab78d03\"}' $'https://us-central1-
   mobile-wallet-mm-snap.cloudfunctions.net/getToken'
   ```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

To mitigate this issue, Cure53 recommends storing the *uid* of the user that created the pairing in the object's data. Furthermore, the developer team could restrict the ability to directly set the *user_id* attribute with user-controlled values.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Miscellaneous Issues

This section covers any and all noteworthy findings that did not incur an exploit but may assist an attacker in successfully achieving malicious objectives in the future. Most of these results are vulnerable code snippets that did not provide an easy method by which to be called. Conclusively, whilst a vulnerability is present, an exploit may not always be possible.

SIL-03-001 WP1: Android application lacks root detection (*Info*)

Cure53 verified that the Silent Shard Android app and Android SDK lack root detection implementation at the time of writing. Consequently, the application fails to alert users regarding the myriad security implications of operating the app in an environment of this nature[6]. To validate this limitation, simply install the application on a rooted device and note the complete lack of application warning.

To mitigate this issue, Cure53 recommends implementing a comprehensive root detection solution. The free-to-download *rootbeer* library[7] for Android could be installed to alert users concerning the risk of operating on the app on a rooted device.

However, one must stipulate that the aforementioned measure would not comprehensively eliminate the bypass potential, particularly if the adversary in question is highly proficient. Moreover, considering that the user holds root access and the application does not, the application will always be disadvantaged in this context.

SIL-03-002 WP1: Insecure v1 signature support on Android (*Info*)

The test team verified that the Android build is signed with an insecure v1 APK signature, which increases the app's susceptibility to the known Janus[8] vulnerability on devices operating Android versions older than 7.

Specifically, this fault grants attackers the opportunity to smuggle malicious code into the APK without breaking the signature. At the time of writing, the app supports a minimum SDK of 21 (Android 5), which also utilizes the v1 signature. Furthermore, Android 5 devices no longer receive updates and are vulnerable to a plethora of security weaknesses.

[6] https://www.bankinfosecurity.com/jailbreaking-ios-devices-risks-to-users-enterprises-a-8515
[7] https://github.com/scottyab/rootbeer
[8] https://www.guardsquare.com/en/blog/new-android-vulnerability-allows-atta[...]affecting-their-signatures

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Thus, one should assume that any installed malicious app can trivially obtain root privileges on those devices using public exploits[9] [10] [11].

This erroneous behavior enables adversaries to manipulate users into installing a malicious attacker-controlled APK that matches the v1 APK signature of the legitimate Android application. As a result, a transparent update would be possible without any ensuing warnings appearing in Android, effectively taking over the existing application and all data held within.

To mitigate this issue, Cure53 recommends increasing the minimum supported SDK level to at least 24 (Android 7) to ensure that this known vulnerability cannot be exploited on devices running deprecated Android versions. In addition, future production builds should only be signed with APK signatures representing v2 and above.

SIL-03-003 WP1: Android config hardening recommendations (*Info*)

Cure53's analysis verified that the Silent Shard Android app fails to leverage optimal values for a number of security-related configurations, which unnecessarily weakens the application's security posture on the whole. To provide one example, the application neglects to mitigate potential tapjacking and screen capture attacks. The associated deficiencies are outlined below.

Issue #1: Lack of tapjacking protection
The Android app accepts user taps whilst alternative apps render any arbitrary overlay. Malicious attackers may leverage this weakness to impersonate users using a crafted app, which launches the victim app in the background whilst another entity is rendered on top. Please note that this attack vector is mitigated from Android 12[12] onward. Since the app supports Android 5, this renders all users operating Android versions lower than 12 vulnerable to this attack. The following command confirms that tapjacking protections are absent both on the provided source code and decompiled app.

Command:
```
grep -r 'filterTouchesWhenObscured' * | wc -l
```

Output:
```
0
```

[9] https://www.exploit-db.com/exploits/35711
[10] https://github.com/davidqphan/DirtyCow
[11] https://en.wikipedia.org/wiki/Dirty_COW
[12] https://developer.android.com/topic/security/risks/tapjacking#mitigations

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

To mitigate this issue, Cure53 advises implement the *filterTouchesWhenObscured*[13][14] attribute at the Android WebView level[15], which will ensure that taps are ignored in the event the Android app is not displayed on top.

Issue #2: Lack of *FLAG_SECURE* for screenshot protection

The Android app allows applications to capture all information displayed on-screen. Malicious apps without any special permissions may accomplish this by simply prompting the user for screen capture access, which is common on Android for screenshot and video recording apps. Notably, a malicious app can accomplish this without any user warnings or prompts if it possesses root privileges, which is achievable by simply prompting the user for them, adopting a rooted device, or exploiting any number of publicly known Android vulnerabilities[16] on unpatched devices (common). To compound this risk assessment, the University of Cambridge's *Security Metrics for the Android Ecosystem*[17] paper demonstrated that root privileges can in fact be gained on 87.7% of Android phones via a security vulnerability.

This issue can be verified on a physical device or emulator with the following commands, which - utilizing a non-root adb session - will capture all screen content whilst the Android app is open and subsequently download it to the computer.

Commands:
```
adb shell screencap -p /sdcard/screenshot1.png
adb pull /sdcard/screenshot1.png
```

To mitigate this issue, Cure53 advises ensuring that all WebViews set the Android *FLAG_SECURE* flag[18], which will guarantee that even apps running with root privileges cannot directly capture the information displayed by the app. A centralized security control would be optimal for this implementation, such as a base activity's *onCreate* event that all other activities inherit.

Proposed fix:
```
public class BaseActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

[13] http://developer.android.com/reference/[...]/View.html#setFilterTouchesWhenObscured(boolean)
[14] http://developer.android.com/reference/[...]/View.html#attr_android:filterTouchesWhenObscured
[15] https://developer.android.com/reference/android/view/View#security
[16] https://www.cvedetails.com/vulnerability-list.php?vendor_id=1224&product_id=19997&[...]
[17] https://www.cl.cam.ac.uk/~drt24/papers/spsm-scoring.pdf
[18] http://developer.android.com/reference/android/view/Display.html#FLAG_SECURE

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
        getWindow().setFlags(LayoutParams.FLAG_SECURE,
    LayoutParams.FLAG_SECURE);
    }
```

Issue #3: Usage of *android:debuggable="true"*

The application allows debugging under the current implementation, which may allow local attackers with access to an unlocked device to enable USB debugging and access application secrets.

Affected file:
AndroidManifest.xml

Affected code:

```
<application android:theme="@style/Theme.SilentShard"
android:label="@string/app_name" android:icon="@mipmap/ic_launcher"
android:name="com.silencelaboratories.silentshard.base.Application"
android:debuggable="true" android:allowBackup="false" android:supportsRtl="true"
android:appComponentFactory="androidx.core.app.CoreComponentFactory">>
```

To mitigate this issue, Cure53 recommends applying the *false* value for *android:debuggable*.

Issue #4: Undefined *android:hasFragileUserData*

Since Android 10, one can specify whether application data should survive when apps are uninstalled via the *android:hasFragileUserData* attribute. When set to *true*, the user will be prompted to retain app information despite uninstallation.

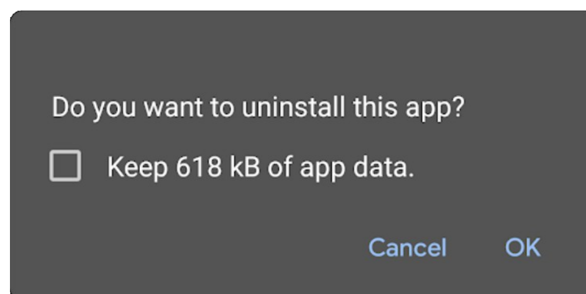

Fig.: Uninstall prompt with checkbox for app data retention.

Since the default value is *false*, this behavior does not incur any direct security impact. However, Cure53 strongly suggests setting this to *false* explicitly to define the app's intention to protect user information and ensure all data is deleted post-app uninstallation. Notably, this option is only applicable in the event the user attempts to

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

uninstall the app from the native settings. If the user uninstalls the app from Google Play, a prompt will not be offered to request whether data should be preserved.

SIL-03-004 WP1: Potential user disruption via exported activity *(Low)*

The assessment confirmed the ability to crash the Silent Shard Android app by invoking an exported activity with specifically-crafted intents. Malicious apps installed on the same Android device could leverage this weakness to continuously crash the app, effectively discouraging users from utilizing it.

Nonetheless, this fault's impact is drastically lowered in this context, due to the fact that the application only crashes when the user is logged out. Furthermore, initiating activities from apps sent via the background is only possible on API level 28 and below. On newer Android versions, intents can only be sent if the app is in the foreground[19].

Affected exported activity:
com.silencelaboratories.silentshard.presentations.main.MainActivity

Steps to reproduce:

1. Open the Silent Shard app and push it to the background whilst running.
2. Record the Android logs locally via:
   `adb logcat > log.txt`
3. Open the *IntentFuzzer*[20] app.
4. Select *NonSystemApps* → *com.silencelaboratories.silentshard*.
5. Scroll down in the activities.
6. Long-press a *MainActivity* until a serializable intent is sent.
7. Confirm that a serializable intent has caused a fatal crash in *com.silencelaboratories.silentshard.presentations.main.MainActivity* by perusing the Logcat output.

Resulting crash output in Logcat:

```
E  FATAL EXCEPTION: main
E  Process: com.silencelaboratories.silentshard, PID: 8118
E  java.lang.RuntimeException: Unable to start activity
ComponentInfo{com.silencelaboratories.silentshard/com.silencelaboratories.silent
shard.presentations.main.MainActivity}: android.view.InflateException: Binary
XML file line #18 in com.silencelaboratories.silentshard:layout/activity_main:
Binary XML file line #18 in
com.silencelaboratories.silentshard:layout/activity_main: Error inflating class
androidx.fragment.app.FragmentContainerView
```

19 https://developer.android.com/guide/components/activities/background-starts
20 https://github.com/MindMac/IntentFuzzer

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
E       at
android.app.ActivityThread.performLaunchActivity(ActivityThread.java:3374)
E       at
android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:3513)
[...]
```

To mitigate this issue, Cure53 recommends exporting the minimum possible volume of activities for the application to function. Following this, the Silence Laboratories team can fortify some of the remaining exported activities with appropriate Android permissions. For additional mitigation guidance and contextual information regarding the protection of Android activities with permissions, please refer to the slides entitled *An In-Depth Introduction to the Android Permission Model*[21], as well as the Stack Overflow discussion concerning *How to use custom permissions in Android*[22]. For activities that must be exported and cannot be protected, adequate input validation and exception handling should be established to definitively eliminate this attack vector.

SIL-03-005 WP1: Potential leakage via absent security screen *(Info)*

Cure53 observed that the Silent Shard Android app and SDK fail to render a security screen when the app is backgrounded. This allows attackers with physical access to an unlocked device to peruse data displayed by the app before it disappears into the background. A malicious application or attacker with physical access to the device could exploit these weaknesses to gain access to user information, such as wallet balances and recent transactions. Notably, whilst the potentially disclosed data is also publicly accessible to attackers with knowledge of the crypto address[23], an attacker with physical access but without the aforementioned information could gain insight regarding a victim user's overall financial status via this attack vector.

To replicate this issue in Android, simply navigate to a sensitive screen and send the application to the background. Subsequently, observe the open app and note that the input text is now user-legible. This text will remain readable even post-device reboot.

[21] https://www.owasp.org/...How_to_Secure_MultiComponent_Applications.pdf
[22] https://stackoverflow.com/questions/8816623/how-to-use-custom-permissions-in-android
[23] https://sepolia.etherscan.io/tx/0xe22[...]

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

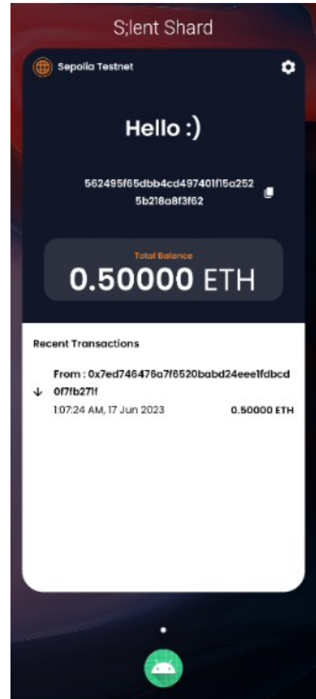

Fig.: Side-channel leak via absent security screen on Android.

This shortcoming's root cause can be verified in the Android application's relevant files, which do not currently capture the relevant events to display a security screen when the application is backgrounded.

For example, the Android app apparently does not offer any code that captures backgrounding events to implement a security screen. This can be confirmed by searching globally for Android events in the source code provided, as well as the decompiled Android APK:

Command:
```
egrep -Ir '(onActivityPause|ON_PAUSE)' * |egrep -v
"(androidx|google|android/support)" |wc -l
```

Output:
```
0
```

To mitigate this issue, Cure53 recommends rendering a security screen overlay when the app is due to be backgrounded. For Android apps, this can be achieved by capturing the relevant backgrounding events; typically *onActivityPause*[24] or the *ON_PAUSE*

[24] https://developer.android.com/.../Application.ActivityLifecycleCallbacks#onActivityPaused[...]

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Lifecycle event[25] are leveraged for this purpose. Following this, the developer team should ensure that all views set the Android FLAG_SECURE flag[26] if feasible. This will guarantee that even apps running with root privileges are unable to directly capture information displayed by the app on screen. Alternatively, a base activity file could be amended to always set this flag, regardless of the focus.

In addition to the measures stipulated above, some apps can implement an app-specific PIN or password to unlock it, thereby bolstering defense-in-depth. However, solutions such as Face ID or Touch ID offer greater versatility in this context, considering that user-friendliness and security resilience are simultaneously provided.

SIL-03-006 WP1: Potential phishing via StrandHogg 2.0 on Android *(Info)*

Testing confirmed that the Android app and SDK are currently vulnerable to a number of task hijacking attacks. The *launchMode* for the app-launcher activity is currently not set and hence defaults to *standard*[27], which mitigates task hijacking via StrandHogg[28] and other deprecated techniques documented since 2015[29], whilst rendering the app vulnerable to StrandHogg 2.0[30]. Notably, this vulnerability affects Android versions 3-9.x[31] but was only patched by Google on Android 8-9[32]. Since the app supports devices operating Android 5 (API level 21), this renders all users running Android 5-7.x vulnerable, as well as users running unpatched Android 8-9.x devices (which is commonly encountered in the modern era).

As a result, a malicious app could leverage this limitation to manipulate the way in which users interact with the app. Specifically, this would be implemented by relocating a malicious attacker-controlled activity in the user's screen flow, which may prove useful toward instigating phishing, Denial-of-Service (DoS), or user credential theft. This fault has been exploited by banking malware Trojans in the past, as confirmed by a number of publicly documented cases[33].

For StrandHogg and regular task hijacking attacks, malicious applications typically adopt one or a selection of the following techniques:

[25] https://developer.android.com/reference/androidx/lifecycle/Lifecycle.Event
[26] http://developer.android.com/reference/android/view/Display.html#FLAG_SECURE
[27] https://developer.android.com/guide/topics/manifest/activity-element#lmode
[28] https://www.helpnetsecurity.com/2019/12/03/strandhogg-vulnerability/
[29] https://s2.ist.psu.edu/paper/usenix15-final-ren.pdf
[30] https://www.helpnetsecurity.com/2020/05/28/cve-2020-0096/
[31] https://www.xda-developers.com/strandhogg-2-0-android-vulnerability-explained-developer-mitigation/
[32] https://source.android.com/security/bulletin/2020-05-01
[33] https://arstechnica.com/[...]/[...]fully-patched-android-phones-under-active-attack-by-bank-thieves/

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

- ***Task Affinity Manipulation***: The malicious application offers two activities, M1 and M2, wherein *M2.taskAffinity = com.victim.app* and *M2.allowTaskReparenting = true*. In the event the malicious app is opened on M2, once the victim application has initiated, M2 will be relocated to the front and the user will interact with the malicious application.

- ***Single Task Mode***: In the event the victim application has set *launchMode* to *singleTask*, malicious applications can apply *M2.taskAffinity = com.victim.app* to hijack the victim application's task stack.

- ***Task Reparenting***: In the event the victim application has set *taskReparenting* to *true,* malicious applications can transfer the victim application task to the malicious application stack.

However, in relation to StrandHogg 2.0, all exported activities lacking a *launchMode* of *singleTask* or *singleInstance* are affected on vulnerable Android versions[34]. This deficiency can be confirmed by reviewing the Android application's *AndroidManifest*.

Affected file:
AndroidManifest.xml

Affected code:
```
<activity
android:name="com.silencelaboratories.silentshard.presentations.main.MainActivit
y" android:exported="true"/>
```

As one can deduce, *launchMode* is not set and hence defaults to *standard*.

To elucidate this area of concern and demonstrate its exploitation potential, an example of a malicious app was created and offered below.

PoC demo:
https://7as.es/SilentShard_3rA3QbO8v/Task_Hijacking_PoC.mp4

To mitigate this issue, Cure53 advises implementing as many of the following countermeasures as deemed feasible by the development team

[34] https://www.xda-developers.com/strandhogg-2-0[...]/

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

- Firstly, the task affinity should be set to an empty string. This is optimally implemented in the Android Manifest at the application level, which will protect all activities and ensure the fix functions even after launcher activity amendments. The application should adopt a randomly generated task affinity rather than the package name to prevent task hijacking, since malicious apps will not be offered a predictable task affinity to target.

- Subsequently, the *launchMode* should be altered to *singleInstance* (rather than *singleTask*). This will ensure continuous mitigation in *StrandHogg 2.0*[35] whilst enhancing security strength against older task hijacking techniques[36].

- A custom *onBackPressed*() function could be implemented to override the default behavior.

- Lastly, the *FLAG_ACTIVITY_NEW_TASK* should not be set in *activity launch* intents. However, if this configuration is deemed absolutely necessary, one should apply it in tandem with the *FLAG_ACTIVITY_CLEAR_TASK* flag[37].

Affected file:
AndroidManifest.xml

Proposed fix:
```
<application android:theme="@style/Theme.SilentShard"
android:label="@string/app_name" android:icon="@mipmap/ic_launcher"
android:name="com.silencelaboratories.silentshard.base.Application"
android:debuggable="true" android:allowBackup="false" android:supportsRtl="true"
android:appComponentFactory="androidx.core.app.CoreComponentFactory"
android:taskAffinity="">
[...]
<activity android:theme="@style/Theme.SilentShard"
android:name="com.silencelaboratories.silentshard.presentations.splash.SplashAct
ivity" android:exported="true" android:launchMode="singleInstance">
```

[35] https://www.xda-developers.com/strandhogg-2-0-android-vulnerability-explained[...]/
[36] http://blog.takemyhand.xyz/2021/02/android-task-hijacking-with.html
[37] https://www.slideshare.net/phdays/android-task-hijacking

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

SIL-03-007 WP1: Android binary hardening recommendations *(Info)*

Cure53 acknowledged that a plethora of binaries embedded into the Silent Shard Android application do not currently leverage the available compiler flags to neutralize potential memory corruption vulnerabilities, which superfluously elevates the application's susceptibility to issues of this nature.

Absent flag:
-D_FORTIFY_SOURCE=2

The omission of this flag means that typical *libc* functions lack buffer overflow checks, which increases the application's proneness to memory corruption defects. Pertinently, the vast majority of binaries are affected, though the following list is reduced for concision reasons.

Example binaries (from decompiled build app):
- *lib/x86/libsodiumjni.so*
- *lib/armeabi-v7a/libsodiumjni.so*

To mitigate this issue, Cure53 suggests compiling all binaries via the *-D_FORTIFY_SOURCE=2* argument to guarantee that common and insecure *glibc* functions, such as *memcpy*, are automatically safeguarded with buffer overflow checks.

SIL-03-008 WP2/3: Multiple hardcoded credentials in source code *(Low)*

During the code audit, Cure53 noted that the source code provided contains an abundant volume of hardcoded credentials. In the event of a source code leak from a developer laptop or source control server, this weakness may allow an attacker to gain access to sensitive information, such as the Etherscan API key and private keys. The following examples illustrate the present fault.

Affected project:
shard-metamask-snaps/silentshardnewui/

Affected file:
src/utils/web3Utils.ts

Affected code:
```
apikey=3JS[...]
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Affected project:
shard-metamask-snaps/metamask-snap-backend/

Affected file:
functions/src/utils/jwt.ts

Affected code:
```
const JWT_TOKEN_KEY ="cc5[...]";
```

Affected project:
shard-metamask-snaps/metamask-snap-backend/

Affected file:
functions/src/index.ts

Affected code:
```
credential: admin.credential.cert({
    clientEmail: "firebase-admin[...]",
    privateKey: "-----BEGIN PRIVATE KEY-----\nMII[...]
}),
```

To mitigate this issue, Cure53 recommends completely removing all credentials, tokens, and secrets from the source code. Alternatively, the applications should retrieve these at runtime using a robust approach, for instance via access to a secure password vault such as *AWS Secrets Manager*[38]. Supplementary guidance regarding this concern and proposed mitigation methods is offered in the Common Weakness Enumeration (CWE) definition's *CWE-798: Use of Hard-coded Credentials* page[39], as well as the *OWASP Cryptographic Storage Cheat Sheet*[40].

[38] https://docs.aws.amazon.com/secretsmanager/latest/userguide/intro.html
[39] https://cwe.mitre.org/data/definitions/798.html
[40] https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Conclusions

The following passages serve to extrapolate the coverage achieved and findings encountered by the pentesters during this assignment, thereby materializing a conclusive overview of the security efficacy offered by the components in scope. Cure53 concludes this project with a mixed impression of the work packages in scope, which all yielded varying impressions from a security perspective. Nonetheless, the general opinion of the grouped characteristics is positive and one can confirm that a decent security standard has already been achieved, though the multitude of tickets documented herein should be internally addressed at the earliest opportunity.

During Cure53's stringent evaluation procedures to detect any potential exposure of confidential data, the likelihood of insecure data logging and storage was estimated in depth. Positively, despite meticulous efforts, the test team was unable to locate any associated risk scenarios.

The source code review phase pertaining to the Android SDK and app similarly proved unfruitful, considering the lack of identified faults in this area. Likewise, the pairing flow and communication between the dApp and device were subjected to a deep dive appraisal process to locate any potential abuse opportunities or logical flaws, though this ultimately verified the framework's sufficient safeguarding.

In addition, the export/import backup functionality withstood rigorous compromise attempts, with the general objective of pinpointing any potential abusable vectors in the synchronization with Google Password Manager.

Concerning some of the pertinent findings encountered, Cure53 acknowledged that the seed phrase is leakable in the mobile application memory, as outlined in ticket SIL-03-009.

Furthermore, the audit team observed that the current implementation does not leverage the security features provided by Android Keystore and lacks anti-instrumentation detection mechanisms. This deficiency raises concern regarding the possibility of seed phrase leakage, as highlighted in ticket SIL-03-010.

During the assessment, the Cure53 consultants also acknowledged that the implementation lacks root detection procedures, which would otherwise render the debugging process easier to achieve (see ticket SIL-03-001). Additionally, auxiliary source code reviews indicated that the application supports insecure v1 signatures on Android and lacks essential hardening recommendations, as documented in tickets SIL-

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

03-002 and SIL-03-007 respectively. These effective measures are deemed essential toward enhancing the application's overall security posture, particularly in relation to sensitive information handling.

The careful inspection of the Android Manifest file disclosed misconfigurations that facilitated a plethora of detrimental security traits, including potential phishing via StrandHogg 2.0 - as addressed in ticket SIL-03-006 - and the capability to instigate user disruption via an exported activity, as detailed in ticket SIL-03-004.

The communication between the Snap and mobile application via the Cloud Functions API evoked problematic tendencies, with issues related to the arbitrary manipulation of the data stored in the Firestore database (see ticket SIL-02-012) and valid authorization token forgery (see ticket SIL-02-013).

Cure53 also painstakingly evaluated the cryptography leveraged by the various systems in scope to determine the propensity for implementational flaws. Here, the team verified that the majority of systems integrated encryption, decryption, and other cryptographic primitives via the cryptographic library *libsodium*. All locations wherein this library was adopted were scoured to guarantee none exhibited any security-related threats, which ultimately proved correct.

An array of general hardening improvements can be incorporated to bolster the Android app’s security posture. Whilst these miscellaneous findings do not directly incur exploitation or damage potential, one can nevertheless recommend resolving all tickets to minimize the application’s attack surface.

In conclusion, this Cure53 security assessment against the diverse range of components in focus revealed a notably resilient security posture for the first two work packages, considering that these areas repelled any major risk circumstances and yielded only miscellaneous pitfalls on the whole. Resolving the aforementioned findings will undoubtedly imbue the application with a performant and commendable security foundation. However, Cloud Functions exhibited ample opportunities for hardening - as corroborated by the numerous connected *Medium* and *High* rated tickets - which must be installed to raise security robustness to a similar level.

Cure53 would like to thank Andrei Bytes, Jay Prakash and Daksh Garg from the Silence Laboratories Pte. Ltd. team for their excellent project coordination, support, and assistance, both before and during this assignment.