STI Template Version May 2018

# Automating Android Application Security Assessments

*GIAC (GMOB) Gold Certification*

Author: Alexander J. Fry, alexanderfry@student.sans.edu
Advisor: *Bryan Simon*

Accepted: *December 19, 2019*

Abstract

Security assessments of mobile applications on Android devices typically require significant manual preparation, manual tool installation, and manual test execution. Often, there is a limited amount of time allocated to perform the testing, constraining the depth and breadth of the assessment. Additionally, testing may be performed informally using a checklist or other guidance. This paper examines the viability of automating the preparation, tool installation, and execution of security tests for Android applications against a recognized standard to establish a minimum testing baseline and ensure consistency in mobile application security assessments.

# 1. Introduction

Security assessments of mobile apps on Android devices typically involve manual preparation, manual tool installation, and manual test execution. Since there is a limited amount of time available to perform the testing, partially automating these tasks would be a more efficient use of resources and would help free up an analyst's time to focus on tests that definitively require manual analysis. This research focuses on analyzing the security of in-house developed apps. Apps developed in-house are less likely to contain intentionally malicious or privacy-violating functionality than external apps (MITRE, pg. 3). Therefore, testing is primarily geared towards identifying common vulnerabilities and, to a lesser extent, privacy-violating behavior.

Mobile app security assessments are performed to ensure that a mobile app has been developed according to security requirements. These requirements can be grouped into two categories: general and organization-specific. General app security requirements define the intended or unintended behavior of an app that should or should not be present in order to ensure the security of the app. These requirements are considered "general" because they can be applied across most mobile apps, whereas "organization-specific" requirements typically provide additional protections based on the mission of the organization. In order to perform a baseline security assessment that is consistent and repeatable and targets the correct level of protection, it is helpful to test against a recognized standard that describes security requirements. Some of the most well-known standards, best practices, checklists, and other resources are specified by NIAP, MITRE, NIST, Josh Wright's App Report Cards, and OWASP (NIST, 2019).

## 1.1. National Information Assurance Partnership (NIAP)

The NIAP defines Protection Profiles (PPs) that are intended to certify products for use in national security systems to meet an implementation-independent set of security requirements. The PP-certified products are also used by federal organizations and commercial organizations in non-national security systems. The NIAP PPs define security objectives, requirements, and assurance activities that must be met for a product evaluation to be considered International Organization for Standardization (ISO)/ International Electrotechnical Commission (IEC) 15408 certified (NIST, 2019). Although

most mobile apps do not need to be IEC 15408 certified, the NIAP recommends a set of activities and evaluations in the document "Requirements for Vetting Mobile Apps from the Protection Profile for Application Software." The requirements defined in this document are divided into two broad categories: functional requirements and assurance requirements. The functional requirements are similar to the general app security requirements defined previously: the intended or unintended behavior of an app that should or should not be present in order to ensure the security of the app. An example functional requirement is the use of Random Bit Generation Services (DBRG) in cryptographic operations. An analyst must determine if the mobile app uses no DRBG functionality, invokes platform provided DRBG functionality, or implements other DRBG functionality for its cryptographic operations. The assurance requirements are actions that an analyst must take or conditions that must be true in order to satisfy that requirement. For example, an assurance requirement is "the application shall be labeled with a unique reference." One strength of the NIAP PP is that it provides notes and a security control outline to help determine if the mobile app satisfies a particular security control (NIAP, 2016).

## 1.2. MITRE Corporation (MITRE)

In 2016, the MITRE Corporation (MITRE) published an analysis of the effectiveness of mobile app security vetting tools for helping enterprises automate portions of their vetting process. The tools evaluated were Android Lint (Included in Android Studio and Android SDK) and eight other commercial products that were anonymized due to non-disclosure agreements. MITRE developed primary focus areas for app vetting based on who developed the application and the planned use of the app.

According to MITRE, the primary focus for "in-house enterprise use apps" and "in-house developed apps for public distribution" is finding and fixing security vulnerabilities (MITRE, 2016). For "commercial enterprise use apps," both the identification of malicious functionality and security vulnerabilities carry equal weight. And the primary focus for "commercial personal use apps" is malicious functionality (MITRE, 2016).

## 1.3. National Institute of Standards and Technology (NIST)

NIST Special Publication 800-53 (Revision 5, Initial Public Draft) provides a comprehensive set of safeguarding measures for all types of computing platforms, including general-purpose computing systems, cyber-physical systems, cloud and mobile systems, industrial/process control systems, and Internet of Things (IoT) devices. The ultimate objective of the safeguarding measures is to build information systems that are more penetration resistant to attacks, limit the damage from attacks when they occur, and make the systems resilient and survivable (NIST, 2017). Three security control groupings, referred to as "baselines," are provided based on high, medium, and low impact. The controls can be customized to an organization-specific process to manage information security and privacy risk.

However, only a subset of NIST security controls is relevant to software applications like mobile apps. An example of a control that applies to mobile apps is SA-11 Developer Testing and Evaluation. This control requires the developer of the system, system component, or system service to implement a security and privacy assessment plan; perform testing/evaluation at an organization-defined frequency and organization-defined depth and coverage; produce evidence of the execution of the assessment plan and the results of the testing and evaluation; implement a verifiable flaw remediation process, and correct flaws identified during testing and evaluation. The publication also describes how to develop specialized sets of controls, also known as control overlays, that can be tailored for unique or specific types of business functions and technologies. A control overlay could be tailored specifically for mobile apps (NIST, 2017).

## 1.4. App Report Cards

The App Report Cards are two Microsoft Excel spreadsheets developed by Josh Wright. The spreadsheets are intended as guidance to ensure consistent analysis and reporting during Android and iOS security assessments. There is one App Report Card for iOS and another for Android, each containing test items that address specific iOS and Android app security requirements. Test items are each worth a certain number of points and are collectively worth a maximum of 100 points, plus 15 points extra credit. Sources like the SANS SEC575 course, Mobile Device Security and Ethical Hacking, explore

tools and techniques for developing test cases for each test item. However, the analyst may choose alternative tools and techniques that meet personal or organizational requirements. At the conclusion of the assessment, the points for each test item are added up to produce an overall letter grade for the mobile app based on a provided grading scale. There is no guidance for the assignment of points, so a mobile app with the same results may receive different grades from two separate analysts, based on analyst judgment. For example, the results from the test item, "Does the app encrypt sensitive network traffic?", may show that the app encrypts some sensitive network traffic but not all sensitive network traffic. Not only could each analyst have a different idea of what constitutes sensitive information, but one analyst may assign zero points, whereas a second analyst may assign partial credit. The difficulty in determining that a security requirement is satisfied coupled with the lack of standard procedures reduces the overall utility of the App Report Cards (Wright, 2015).

## 1.5. OWASP Mobile Application Security Verification Standard (MASVS)

The OWASP Mobile Application Security Verification Standard (MASVS) v1.1 is envisioned as a security standard for mobile apps. It focuses on how mobile apps must handle, store, and protect sensitive information and attempts to standardize security requirements using verification levels that address different threat scenarios. MASVS offers a baseline for mobile application security (MASVS-L1), the inclusion of defense-in-depth measures (MASVS-L2), and protections against client-side threats (MASVS-R). One of the goals of the MASVS is to define an industry standard that can be tested against in mobile app security assessments (OWASP, 2019).

The MASVS-L1 was chosen as the principal standard for this research because it is public and open; comprehensive, mature, developed by experts, and highly testable. A mobile app that achieves MASVS-L1 adheres to mobile application security best practices. It fulfills baseline requirements in terms of code quality, handling of sensitive data, and interaction with the mobile environment. Additionally, other mobile app guidance and checklists can be compared against MASVS-L1, helping to meet organizational assurance and compliance requirements (OWASP, 2019).

The MASVS-L1 contains 43 security requirements across seven areas: Architecture, Design, and Threat Modeling; Data Storage and Privacy; Cryptography; Authentication and Session Management; Network Communication; Platform Interaction; and Code Quality and Build Setting. The Architecture, Design, and Threat Modeling Requirements is the only category that does not map to technical test cases in the OWASP Mobile Testing Guide. It covers topics such as threat modeling, secure SDLC, and key management (OWASP, 2019).

Appendix A contains a mapping of the MASVS-L1 security requirements to tools for testing each requirement. The tools listed come from the OWASP Mobile Testing Guide, SANS SEC575 class, and those used by this author in performing security assessments.

For each technical test case illustrated in this research, the primary criteria for selecting each tool was the ability to be scripted and the accuracy of the results.

## 2. Android Device Ecosystem

The Android marketplace is worldwide and consists of millions of devices, each running different versions of the Android operating system (OS) as new as Android 10, released in September 2019 and as old as Android Gingerbread 2.3, released in February 2011 (Google, 2019). In addition to the wide variety in the age of devices and OS, the marketplace is also characterized by inconsistency in the security posture of devices. The significant differences among Android devices in the marketplace is also referred to as "fragmentation." This contrasts with the Apple iOS marketplace in which the majority of end-user devices are less than four years old and running a recent version of iOS with a consistent security posture (Horwitz, 2019). The reason for the Android device "fragmentation" is due to the operating model and business drivers in the Android ecosystem.

The Android OS is maintained by the Open Handset Alliance (OHA), which includes Google, the original creator of the OS. A vendor such as Samsung or Huawei then customizes the Android OS, often bundling third-party apps. A mobile operator (MO) such as T-Mobile bundles additional third-party apps before the device is delivered

to an end-user. If a security flaw is identified in Android, vendors and MOs need to go through a complicated process to deliver the security update to an end-user (Wright, 2019).

The process typically begins with the flaw being disclosed to the OHA, either publicly or privately. For many Android functions, the flaw can be present in code that is outside the control of the OHA. This requires communications with the vulnerable code owner, who may be an independent software vendor (ISV) or a member of the OHA. When the remediation for a vulnerability, typically consisting of a software update, is made available, notice is returned to the OHA, which subsequently shares information about the remediation with multiple vendors under a nondisclosure agreement. The vendor must test and incorporate the software update into the Android OS, taking into account any negative interaction with third-party apps bundled with the device. After the update is prepared, it must be shared with the MOs, who must repeat the process of testing the software update to ensure there is no negative interaction with third-party apps the MO has bundled with the device. After the MO completes integration and testing of the remediation, it can be made available to end-users via an Over-the-Air (OTA) update, as a download on a website, or via another distribution method (Wright, 2019).

If the vendor or MO chooses not to remediate a flaw, Android users have no access to a software patch that would protect the device and end-user from the exploitation of the flaw. Often, these are business-driven decisions. For example, the vendor makes money from selling new devices every two to three years, instead of maintaining them, so they may decide not to remediate a flaw because it is close to the release date of new devices. Similarly, the OHA may choose not to produce the remediation for a specific version of the Android OS because there is an upcoming release of the Android OS that would not contain the flaw. On the other hand, ISVs and other producers of apps are incentivized to maintain backward compatibility with old devices and Android OS because more app use results in more money. The apps maintain backward compatibility by specifying a minimum API version that is necessary to support the core features of the app. The apps running on older devices and operating systems are not able to take advantage of security features that require a more recent

release of the API. This results in many users running vulnerable software on potentially vulnerable devices with little support or significant delay for security resolution (Wright, 2019).

Due to this fragmentation, performing security testing on a wide variety of mobile apps would require several different Android physical devices with differing OS versions and patch levels. An alternative to a physical device is an Android Virtual Device (AVD) run in an Android emulator. An AVD is a configuration that defines the characteristics of an Android phone, tablet, Wear OS, Android TV, or Automotive OS device that are simulated in an Android Emulator. Both physical devices and AVDs that run in an emulator have advantages and disadvantages (Figure 1).

| Emulator/Simulator | Physical Device |
|---|---|
| Cheap | Expensive |
| Snapshots | No easy snapshots |
| Easily detected by emulator detection | A real device - bypasses emulator detection |
| API calls may be simulated | Real APIs |
| Rooted by default | Needs custom rooting |
| Can be extremely slow | Very fast |
| Easily start a new emulator | The device can be bricked |

Figure 1. Emulator/Simulator vs. Physical Device. From “SANS SEC575” (Wright, 2019).

Ultimately, in order to automate the testing process at a reduced cost, it makes sense for most organizations to use an emulator. The emulator can also be configured to support the minimum API necessary to test each mobile app.

# 3. Lab Setup

The lab environment consisted of a host system and an AVD for running the mobile app. The host system was a MacBook Pro (15-inch, 2017) with a 2.9 GHz Intel Core i7 processor, 16 GB 2133 MHz LPDDR3 memory on macOS Mojave, version 10.14.6. Security testing was performed on the mobile app using technical test cases developed for four of the MASVS-L1 security requirements, a subset of the security requirements whose technical test cases could be partially automated. The technical test cases, scripts, and other tools can be found in the supplemental material GitHub repository: *https://github.com/alexanderfry/android-research-paper*.

## 3.1. Virtual Testbed

Two of the most popular emulators are Android Studio and the Android-x86 Project. AVDs that run in these emulators need to be tailored to the API levels used by the target mobile apps.

### 3.1.1. Android Studio

The Android Studio Integrated Development Environment (IDE) provides the Android emulator. The Android emulator uses the open-source QEMU virtualization software to emulate an ARM or x86/64 processor, behaving similarly to a stock Android release before the inclusion of MO or vendor third-party apps. The Android Studio also provides the AVD Manager interface that is used to create and manage AVDs and the Software Development Kit (SDK) Manager interface that is used to download different versions of the Android platform (Google, 2019).

The availability of a command-line interface makes it possible to script the creation, management, and launching of AVDs. The *avdmanager* command is used to create and manage Android Virtual Devices (AVDs) while the *emulator* command is used to list and launch available AVDs. Additionally, the command line offers features such as proxy, packet capture, network throttling, and GNU debugger (GDB) access. Unlike VMware and other emulators, the QEMU emulator uses the local IP stack of the host instead of creating an additional IP stack interface. This means that the IP address of the AVD is the same as the host IP address, complicating communication between the

AVD and the host. In order to communicate on the command line with the AVD via the Android Debug Bridge (ADB) tool, it is necessary to forward a listening port in the AVD to a port on the host system (Wright, 2019).

Although the Android Studio provides the facility to create, manage, and launch AVDs from the command line, it has disadvantages that include lack of performance; problems with peripherals such as sound cards, video cards, and cameras; and compatibility issues with personal security products. An alternative is the Android-x86 Project (Wright, 2019).

### 3.1.2. Android-x86 Project

The Android-x86 project is a port of the Android OS to Intel-based processors using 32-bit (x86) and 64-bit processors. The advantages of the Android-x86 Project are that it can be run in virtual machines (VM) from VMware or other hypervisors to create AVDs. These VMs offer the same level of access and ability to install, debug, and evaluate Android applications (Wright, 2019).

Additionally, the project provides ISO files of various Android OS releases and supports ext3, ext2, NTFS, and FAT32 file systems. An ISO can be installed in various ways, e.g., on its own hard disk; on an NTFS file system to co-exist with Microsoft Windows; and on a hypervisor such as VMware, XEN, VirtualBox, or Hyper-V. Once an Android OS is installed, the Android Debug Bridge (ADB) can be used to control and automate interaction (Android-x86, 2019).

### 3.1.3. Android Debug Bridge

The Android Debug Bridge (ADB) is a utility used to interact with both Android physical devices and virtual devices. The ADB can accomplish common tasks such as listing attached and running Android devices, copying files to and from the filesystem, installing and uninstalling Android apps using Android Package (APK) files, accessing a Linux shell on the device, and viewing log files (Wright, 2019). Common commands are summarized in the table below (Figure 2).

| Description | Command |
|---|---|
| List running (or attached) Android devices by serial number and description. Other adb commands can specify -s <serial> to connect to a specific device. | adb devices |
| Connect to remote ADB Server (Android-x86) | adb connect [IP]:5555 |
| Restart ADB server as root | adb root |
| Copy the local file to Android filesystem | adb push [local] [remote] |
| Retrieve a file from Android to the local filesystem | adb pull [remote] [local] |
| Access a Linux shell | adb shell |
| Install an Android app | adb install [filename].apk |
| Uninstall an Android app | adb uninstall [package-name] |
| View Android device log | adb logcat |
| Reboot Android device | adb reboot |
| Use shell to screenshot directly from the command line, saving the screenshot on the local file system (ex. In macOS Mojave) | adb shell screencap -p \| perl -pe 's#\x0D\x0A#\x0A#g' > screenshot.png |

Figure 2. ADB Commands. From Author and “SANS SEC575” (Wright, 2019).

### 3.1.4. Choice of Android Emulator

Android Studio was chosen as the emulator to use for this research. The primary reason was due to its ability to script the full lifecycle of the Android mobile app from installation to disposal. Additionally, two of the disadvantages cited previously have no bearing on the security testing. One, the security testing will not interact with peripherals

such as sound cards, video cards, and cameras. Two, there will be no compatibility issues with personal security products because there are none installed. The performance issues should be minimized as well because the host machine supports the Intel Hardware Accelerated Execution Manager (Intel® HAXM). Intel® HAXM is a hardware-assisted virtualization engine (hypervisor) that uses Intel Virtualization Technology (Intel® VT) to speed up Android app emulation on a host machine (Google, 2019). Additionally, much of the testing will be headless, not requiring interaction with the Android UI.

## 3.2. Intentionally Vulnerable Mobile App

The security testing was performed on InsecureBankV2, an intentionally vulnerable mobile app developed primarily by Dinesh Shetty and made available for download on GitHub. It is an Android native mobile app developed in Java that comes with a backend server component developed in Python2. It was chosen from among the apps in the marketplace because it is designed to teach mobile application security best practices and contains a large number of known vulnerabilities that can be identified through security testing (Figure 3):

| | |
|---|---|
| Flawed Broadcast Receivers | Insecure Logging mechanism |
| Intent Sniffing and Injection | Android Pasteboard vulnerability |
| Weak Authorization mechanism | Application Debuggable |
| Local Encryption issues | Android keyboard cache issues |
| Vulnerable Activity Components | Android Backup vulnerability |
| Root Detection and Bypass | Runtime Manipulation |
| Emulator Detection and Bypass | Insecure SD Card storage |
| Insecure Content Provider access | Insecure HTTP connections |
| Insecure WebView implementation | Parameter Manipulation |
| Weak Cryptography implementation | Hardcoded secrets |
| Weak change password implementation | Username Enumeration issue |
| Sensitive Information in Memory | Developer Backdoors |

Figure 3. InsecureBankV2 Known Vulnerabilities (Shetty, 2019).

Before testing InsecureBankV2, a suitable AVD needs to be created and rooted; and the InsecureBankV2 server component needs to be installed and run.

### 3.2.1. Creating the AVD

The InsecureBankV2 API levels must be identified in order to configure a compatible emulator environment for InsecureBankV2. The API levels are defined in the binary encoded Android manifest file, *AndroidManifest.xml,* along with other information a (virtual) device needs in order to run the app.

First, in order to retrieve this information, the manifest file was extracted from the InsecureBankV2 APK: *unzip InsecureBankv2.apk AndroidManifest.xml.*

Then, the AndroidManifest.xml file was converted to a text format with AXMLPrinter2.jar, using grep to retrieve the minimum API level: *java -jar AXMLPrinter2.jar AndroidManifest.xml | grep minSdkVersion*.

The resulting output *android:minSdkVersion="15"* indicated that the minimum API level was 15. However, there are other API levels that have a bearing on emulator selection, e.g., targetSdkVersion and maxSdkVersion. The targetSdkVersion is the API level of the Android device where the app expects to run. Android uses this setting to determine whether to enable any compatibility behaviors so that the app continues to operate as intended. The target API version for InsecureBankV2 is 22. The maxSdkVersion sets a maximum API level that applies to specific permissions. For example, the Android permission, *READ_EXTERNAL_STORAGE* has a maximum API level of 18.

Based on these API levels, the “Jelly Bean” release (Android 4.3) with API level 18 was selected as the emulator version that supports the least common denominator of API levels to support the core app functionality.

First, the *sdkmanager --list --verbose | grep platforms/android-18* command was run to check if Jelly Bean was already installed. The command did not return any results indicating that Jelly Bean was not installed.

Next, the platform, platform-tools (which includes adb and fastboot), and Google APIs Intel x86 Atom System Image was installed by running the command: *sdkmanager "platform-tools" "platforms;android-18" "system-images;android-18;google_apis;x86”*. The SDK installation command is captured in the *install_android_sdk.sh* script.

Finally, the AVD was created by running the following command: *avdmanager create avd --name app_testing --abi google_apis/x86 --package 'system-images;android-18;google_apis;x86' --device "Nexus 6P" -c 1000M.* This creates a Nexus 6P AVD named app_testing with a 1 GB virtual SD Card. The AVD creation steps were captured in the *create_android_avd.sh* script.

Unfortunately, not all of the options for AVD creation can be specified via the command line. For example, options for choosing hardware graphics rendering, having the device power as from a cold boot, using more than one core of the CPU, and enabling keyboard input, need to be enabled in the Android Studio GUI. These settings can be found by loading the AVD Manager, selecting "Edit this AVD" next to the newly created AVD, and then clicking on the "Show Advanced Settings" button in the lower left-hand corner (Figure 4).

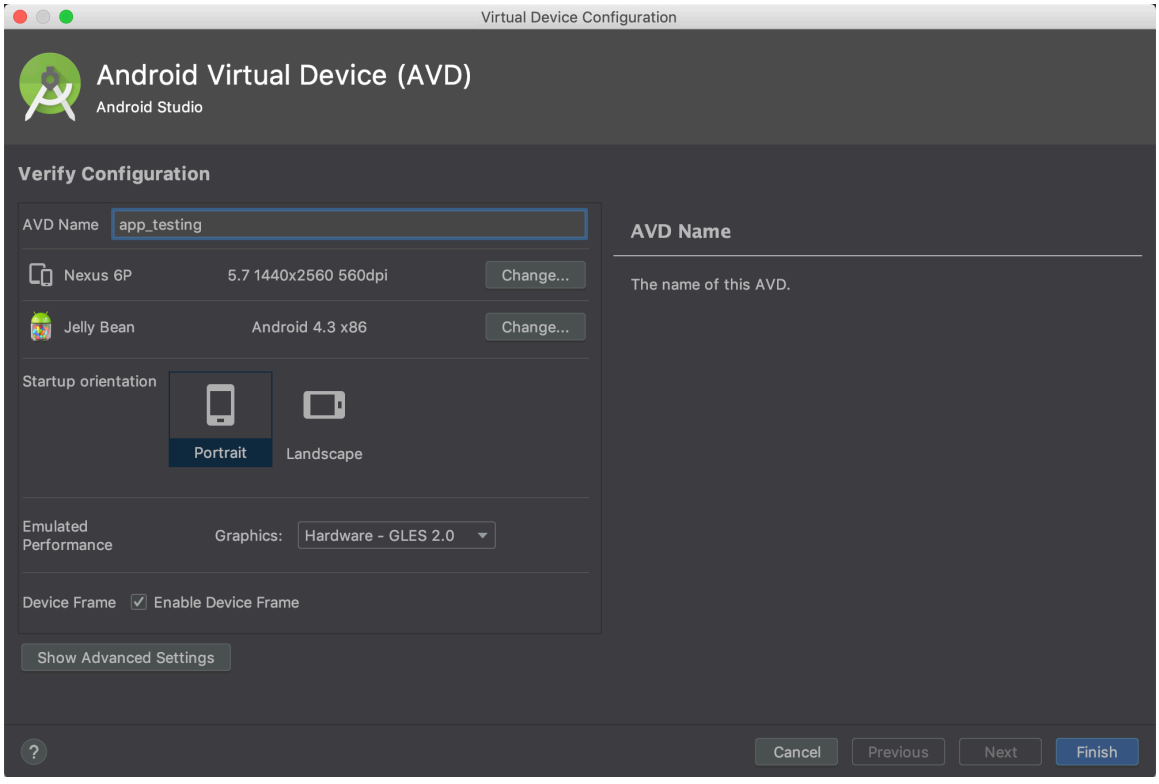


Figure 4. AVD Show Advanced Settings in Android Studio

### 3.2.2. Rooting the AVD

After the AVD advanced settings were configured, the AVD was "rooted." Rooting is a process to enable users and security analysts to obtain privileged control (known as **root** access) over the Android OS. The AVD was rooted using a third-party app, SuperSU. The SuperSU Android Package can be downloaded from a GitHub repository maintained by a user known as 0xFireball.

First, the AVD was launched with the command: *emulator -avd "app_testing" -writable-system -selinux disabled*, which makes the OS writable and disables Security-Enhanced Linux (SELinux).

Next, the ADB daemon (adbd) was restarted as root and the system was remounted as writable by running the command: *adb root && adb remount*.

Then, after changing to the root_avd directory, the SuperSu APK was installed on the AVD with the command: *cd root_avd/;adb install SuperSU/common/Superuser.apk*.

Finally, the remaining steps consisted of copying the superuser (su) binary executable to the AVD, modifying permissions on the executable, which would ensure that the rooting would persist from a reboot, and opening the SuperSU app on the emulator.

Upon opening, the SuperSu app displayed the following message: "The SU binary needs to be updated. Continue?". It was necessary to click the "Continue" button, choose the "Normal" installation, and then press "OK" after the installation completed. If the AVD does not remain rooted after a reboot, it may be necessary to run the command: *su --daemon&* from a root shell on the AVD (0xFireball, 2017). The scriptable rooting steps were captured in the *root_android_avd.sh* script, located in the supplemental material GitHub repository.

After rooting, the AVD was launched by running the command: *emulator -avd "app_testing" -writable-system -selinux disabled*. At various times when this command was run, the emulator launched but complained about not being able to connect to the adb daemon on port 5037. The connection error was resolved by shutting down the emulator

manually and stopping the adb daemon with the command: *adb kill-server* and then starting it again with: *adb start-server*.

After the AVD launched successfully in the emulator, the *adb devices* command was run to identify the app_testing AVD device id. Then, the InsecureBankV2 mobile app was installed on the AVD: *adb install InsecureBankv2.apk <device id>*.

### 3.2.3. Running InsecureBankV2 Server

The InsecureBankV2 app back-end server component is written in Python2 and requires a number of third-party libraries: flask, flask-sqlalchemy, simplejson, cherrypie, and web.py. A Python virtual environment was configured to install the app and its dependencies. A Python virtual environment is a self-contained directory tree that contains a Python installation for a particular version of Python, plus a number of additional packages. Using a virtual environment makes it possible to run multiple applications on the same host, each with conflicting requirements. Otherwise, installing dependencies as globally accessible would risk breaking system tools or other projects. Once the virtual environment is set up, the back-end server is run on the host operating system on default TCP port 8888. The setup of the virtual environment and command to run the server can be executed by running the BASH script: *start_server.sh*.

## 4. Testing

There are two broad classes of security testing for mobile apps: Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST). SAST consists of examining an app's components without executing them by analyzing the source code either manually or automatically. DAST involves examining the operation of the app during runtime. This type of analysis can be manual or automatic. It can be performed from outside the application, e.g., via scanning or inside the application via instrumentation or proxy of network traffic. DAST is an effective way to view the interaction of components, entry points, and data flow (OWASP, 2018).

In most cases, an app needs to be run, and all exposed functionality exercised to facilitate both static and dynamic testing. For example, if an app stores sensitive data

such as PII, user credentials, or cryptographic keys, that data needs to be entered and stored to statically test if system credential storage facilities are used appropriately to safeguard said data. With mobile apps, the user typically interacts with screens that are the user interface (UI) to the mobile app. While UI testing could be performed manually, it is more efficient to have the user interaction automated with a suite of tests that are typically developed by quality assurance to confirm that the functionality of an app is complete and behaving as expected.

## 4.1. User Interface Test Automation

There are a number of popular test suites for Android UI test automation, including Android Studio Espresso, Appium, and Selendroid. Android Studio also comes with the Espresso Test Recorder that is used to generate Espresso tests during user interaction (Google, 2019). While it is possible to generate some tests for user interaction, it is often necessary to write or edit test code to ensure full coverage of the user interface.

Unfortunately, the InsecureBankV2 GitHub repository does not come with UI tests, and developing a UI test automation suite is outside the scope of this research. If a UI test suite is not available, a security analyst has other options that include employing tools such as Burp Suite and Drozer. While Burp Suite and Drozer are typically used for security testing, they can also be used for functional testing.

### 4.1.1. Burp Suite

Burp Suite can be used to automate functional tests by manually executing app functionality, capturing the traffic between the app and the server, and replaying the traffic. It contains a Proxy feature that can capture both HTTP and HTTPS traffic and a Macro feature that can be used to replay the traffic. Setting up Burp Suite to proxy traffic from the AVD involves several steps that include downloading the Burp Suite certificate and installing it in the AVD. Fortunately, Jake Miller developed a Python3 script, *install_burp_cert.py*, that automates these steps (Miller, 2019). The Python3 script requires only one additional library, pyOpenSSL. If pyOpenSSL is installed as a global dependency, it could break system tools or other projects. This can be avoided by installing pyOpenSSL as a local dependency in a Python virtual environment.

A virtual environment was created to install the library and run the script. This consists of only three commands: *python3 -m venv install_burp_cert*, *source install_burp_cert/bin/activate*, and *pip3 install pyopenssl*.

After running the script, the emulator was shut down and restarted with an additional argument to send all traffic through Burp Suite Proxy: *emulator -avd "app_testing" -writable-system -selinux disabled -http-proxy 127.0.0.1:8080*.

Next, the “jack” user was authenticated through the InsecureBankV2 app with the correct password. The authentication step consists of a single HTTP POST Request and corresponding HTTP 200 Response that is captured in the Burp Suite Proxy->HTTP history tab. Then, it is a straightforward process to create a Macro from the Project options->Sessions tab.

Finally, When the Macro is run, the traffic is replayed against the backend server component, resulting in a successful authentication. The Macro can be edited, and the username and password changed to incorrect values and then run again, resulting in failed authentication.

This test also partially confirms MASVS 4.1 security requirement, which states, “If the app provides users access to a remote service, some form of authentication, such as username/password authentication, is performed at the remote endpoint” (OWASP, 2019).

A disadvantage of using Burp for functional test automation is that the Burp Macro does not interact with the UI; it only replays traffic against the backend server component. As an alternative to or in conjunction with Burp, Drozer can be used to interact with the mobile app.

### 4.1.2. Drozer

Drozer consists of a Console that runs on the host and an Agent that runs on the Android physical or virtual device. Before using Drozer, the components were installed on the host and AVD.

First, the Agent was downloaded from the Internet and installed on the AVD. These steps are captured in the script: *install_drozer_agent.sh*.

Next, it was necessary to open the Drozer Agent app on the device and start the embedded server on the default port 31415. Then, in the same terminal window, the following command was run to forward the local TCP port 31415 to the remote TCP port 31415: *adb forward tcp:31415 tcp:31415*.

Afterward, the Console was installed. The Console is developed in Python2 and requires a number of third-party libraries in order to run, including: pyOpenSSL, protobuf, twisted, and service_identity. A Python virtual environment was set up to install the console by executing the following BASH script: *install_drozer_console.sh*.

Finally, after executing the Drozer Console installation script, the command: *drozer console connect* was run to connect to the Agent. A successful connection returns a Drozer command prompt "dz>".

Drozer interacts with exposed activities to automate functional testing of the app via the user interface. An activity represents a single screen in a mobile app and needs to be exposed (as opposed to hidden) for Drozer to be able to interact with it. Drozer communicates with the exposed activity by sending it an Intent, which is a data structure that tells the activity to start and sends it any necessary data.

First, it was necessary to enumerate the exposed activities by running the command, *run app.activity.info -a com.android.insecurebankv2* that produces the following output: *com.android.insecurebankv2.LoginActivity*, *com.android.insecurebankv2.PostLogin*, *com.android.insecurebankv2.DoTransfer*, *com.android.insecurebankv2.ViewStatement*, and *com.android.insecurebankv2.ChangePassword*.

Next, the InsecureBankV2 source code was reviewed to determine which of these activities referred to the login screen. However, it was determined that the login screen corresponded to a Java Class named "DoLogin" and a corresponding *com.android.insecurebankv2.DoLogin* activity that was not listed as an exposed activity by Drozer. A subsequent review of the InsecureBankV2 AndroidManifest.xml file confirmed that the activity was defined but was not exposed – this is referred to as "exported" in the manifest file. A decision was made to modify the AndroidManifest.xml

file in order to export the *com.android.insecurebankv2.DoLogin* activity. This would involve decoding the InsecureBankV2 app, modifying the AndroidManifest.xml file to export the activity; and then repacking, re-signing, and re-installing the InsecureBankV2 app as described in the following steps.

First, the InsecureBankV2 app was decoded with the *apktool* by running the command: *apktool d --no-src --frame-path InsecureBankV2MobileFrameworks InsecureBankv2.apk*.

Next, the *android:exported="true"* line was prepended to the *com.android.insecurebankv2.DoLogin* activity in AndroidManifest.xml.

Then, the app was repacked and re-signed. The app was signed with the Android Studio debug code-signing certificate. If a project has been built in Android Studio previously, the IDE already created a debug keystore and a certificate in $HOME/.android/debug.keystore. The default password for this keystore is "android," and the key is called "androiddebugkey."

Finally, the InsecureBankV2 app was uninstalled from the AVD, and the re-signed app was installed. These steps are captured in the script: *repack_re-sign_re-install.sh*, and must be run from within the unpacked app directory.

After the Drozer Console command to enumerate activities was run again, the list included the *com.android.insecurebankv2.DoLogin* activity.

Based on a review of the source code, the DoLogin activity accepts an Intent with extra parameters for the username and password that are required to authenticate to the InsecureBankV2 app. After some trial and error, the following command allowed Drozer to authenticate to the InsecureBankV2 app: *run app.activity.start --component com.android.insecurebankv2 com.android.insecurebankv2.DoLogin --extra string passed_username "jack" --extra string passed_password "Jack@123$"*. The following section demonstrates how this UI test can be used in conjunction with a security test to verify MASVS security requirements.

## 4.2. Technical Test Case for MASVS 2.3

MASVS 2.3 states, “No sensitive data is written to application logs” (OWASP, 2019). The technical test case for MASVS 2.3 consists of running a command that dumps a log of system messages from the attached AVD, confirming that no sensitive data is written to the log. Ideally, all app functionality should be exercised before analyzing the log messages. However, since InsecureBankV2 is an intentionally vulnerable mobile app, it was known beforehand that the app discloses the authentication credentials in the system log.

The test sequence began by opening a Terminal window and running the command: *adb logcat | egrep -i ‘http|https|cookie|login|md5|sha1|auth|pass|ssn|ein’*, that uses *adb logcat* to continuously dump the log of system messages, piping the output to *egrep* to filter for any sensitive data. Then the “jack” user was authenticated through the InsecureBankV2 app by running the authentication sequence in the Drozer Console: *run app.activity.start --component com.android.insecurebankv2 com.android.insecurebankv2.DoLogin --extra string passed_username "jack" --extra string passed_password "Jack@123$"*. The below screenshot of the command output shows the successful authentication and the plaintext authentication credentials that were written to the log (Figure 5).

```
I/ActivityManager( 1524): START u0 {flg=0x4000000 cmp=com.android.insecurebankv2/.LoginActivity} from pid 2914
I/ActivityManager( 1524): Displayed com.android.insecurebankv2/.LoginActivity: +93ms
I/ActivityManager( 1524): START u0 {flg=0x10000000 cmp=com.android.insecurebankv2/.DoLogin (has extras)} from pi
d 3118
D/Successful Login:( 2914): , account=jack:Jack@123$
I/ActivityManager( 1524): START u0 {cmp=com.android.insecurebankv2/.PostLogin (has extras)} from pid 2914
I/ActivityManager( 1524): Displayed com.android.insecurebankv2/.PostLogin: +172ms (total +226ms)
```

Figure 5. Successful Authentication to DoLogin Activity

## 4.3. Technical Test Case for MASVS 4.1

MASVS 4.1 states, “If the app provides users access to a remote service, some form of authentication, such as username/password authentication, is performed at the remote endpoint” (OWASP, 2019). The technical test cases for MASVS 4.1 are performed with both Burp Suite and Drozer. Burp Suite was used to capture the traffic from the authentication step, as described in Section 4.1 on User Interface Test Automation. The Burp Suite testing confirms that the app performs authentication before

granting users access to a remote service, and the authentication is performed at the remote endpoint. However, there is a vulnerability in InsecureBankV2 that allows the login screen to be bypassed, resulting in the user being granted authenticated access to the post-login screens. One way to remediate this vulnerability would be to prevent the PostLogin activity from being exported.

In this test case, *adb logcat* is used to capture the system log output as evidence that the login bypass was successful. The test sequence began by opening a Terminal window and running the command: *adb logcat | egrep -i 'login'*. Then a call was made to the "PostLogin" activity from the Drozer Console without any additional arguments: *run app.activity.start --component com.android.insecurebankv2 com.android.insecurebankv2.PostLogin*. The below screenshot of the *adb logcat* Terminal window shows that the "PostLogin" screen was displayed without any prior authentication (Figure 6).

```
I/ActivityManager( 1524): START u0 {flg=0x4000000 cmp=com.android.insecurebankv2/.LoginActivity} from pid 2914
I/ActivityManager( 1524): Displayed com.android.insecurebankv2/.LoginActivity: +127ms
I/ActivityManager( 1524): START u0 {flg=0x10000000 cmp=com.android.insecurebankv2/.PostLogin (has extras)} from
pid 3118
I/ActivityManager( 1524): Displayed com.android.insecurebankv2/.PostLogin: +117ms
```

Figure 6. Direct to PostLogin Screen

## 4.4. Technical Test Case for MASVS 6.1

MASVS 6.1 states, "The app only requests the minimum set of permissions necessary" (OWASP, 2019). The permissions for an Android mobile app are declared in the Android manifest file, AndroidManifest.xml. Given an Android APK, the manifest file must be extracted and decoded to plain text before the permissions can be analyzed. Another approach is to use Drozer, which interacts with the installed mobile app and prints the app permissions as output to the console. An added benefit is that the output can be further parsed to determine which permissions may be unnecessary or dangerous.

Beginning with Android 6 (API 23 and later), apps are explicitly granted all the Android "normal" permissions that previously required authorization from the user. The Android documentation states, "there is no great risk to the user's privacy or security in letting the apps have those permissions" (Google, 2019). At the same time, Android designated nine permissions as "dangerous." The dangerous permissions are classified

into nine permission groups: Calendar, Camera, Contacts, Location, Microphone, Phone, Sensors, SMS, and Storage. Since the InsecureBankV2 AVD was built using API level 18, it does not take advantage of this permission management model. However, the classification is useful for analyzing the permissions, as will be shown in the following steps.

First, the Drozer Console was connected to the AVD as described in previous technical test cases. Then the following command was run from the Drozer Console: *run app.package.info -a com.android.insecurebankv2*. The output from this command is shown below in Figure 7.

```
dz> run app.package.info -a com.android.insecurebankv2
Package: com.android.insecurebankv2
  Application Label: InsecureBankv2
  Process Name: com.android.insecurebankv2
  Version: 1.0
  Data Directory: /data/data/com.android.insecurebankv2
  APK Path: /data/app/com.android.insecurebankv2-1.apk
  UID: 10050
  GID: [3003, 1015, 1028]
  Shared Libraries: null
  Shared User ID: null
  Uses Permissions:
  - android.permission.INTERNET
  - android.permission.WRITE_EXTERNAL_STORAGE
  - android.permission.SEND_SMS
  - android.permission.USE_CREDENTIALS
  - android.permission.GET_ACCOUNTS
  - android.permission.READ_PROFILE
  - android.permission.READ_CONTACTS
  - android.permission.READ_PHONE_STATE
  - android.permission.READ_EXTERNAL_STORAGE
  - android.permission.READ_CALL_LOG
  - android.permission.ACCESS_NETWORK_STATE
  - android.permission.ACCESS_COARSE_LOCATION
  Defines Permissions:
  - None
```

Figure 7. InsecureBankV2 Permissions

By appending " > output.txt" to the command, the output can be saved to a local text file. Then, after exiting the Drozer Console, the output can be further refined to identify permissions that are classified as "dangerous": *cat output.txt | egrep -i 'calendar|camera|contacts|location|microphone|phone|sensors|sms|storage'* (Figure 8).

```
(dzr) Chaac:InsecureBankv2_working alex$ cat output.txt | egrep -i 'calendar|camera|contacts|location|microphone
|phone|sensors|sms|storage'
  - android.permission.WRITE_EXTERNAL_STORAGE
  - android.permission.SEND_SMS
  - android.permission.READ_CONTACTS
  - android.permission.READ_PHONE_STATE
  - android.permission.READ_EXTERNAL_STORAGE
  - android.permission.ACCESS_COARSE_LOCATION
```

Figure 8. InsecureBankV2 Dangerous Permissions

At this point, the analyst must determine if any of the "dangerous" permissions are unnecessary or used in a malicious fashion. This typically involves a manual exploration of mobile app functionality.

## 4.5. Technical Test Case for MASVS 7.1

MASVS 7.1 states that "The app is signed and provisioned with a valid certificate" (OWASP, 2019). The technical test case for MASVS 7.1 uses command-line utilities to confirm that the app is digitally signed and provisioned with a valid certificate. When an APK is signed, a public-key certificate is attached to it. This certificate uniquely associates the APK with the publisher and the publisher's private key. When an app is being built in debug mode, the Android SDK signs the app with a debug key created specifically for debugging purposes. An app signed with a debug key is not meant to be distributed (OWASP, 2018).

Android APKs have a directory called META-INF, which includes certificates and signature information files. The file with the "RSA" extension is the public key used to validate the app signature. First, the "RSA" file was identified by running the unzip (list) command: *unzip -l InsecureBankV2.apk | grep META-INF*. Then the RSA file was uncompressed with the command: *unzip -qq InsecureBankV2.apk META-INF/CERT.RSA*. The certificate information was then checked by running the command: *openssl pkcs7 -inform DER -in META-INF/CERT.RSA -noout -print_certs*. The screenshot below shows the output from this command (Figure 9).

```
subject=/ST=MA/L=Boston/O=SI/OU=Services/CN=Dinesh Shetty
issuer=/ST=MA/L=Boston/O=SI/OU=Services/CN=Dinesh Shetty
```

Figure 9. InsecureBankV2 Certificate Details

The certificate details reflect that someone who purports to be Dinesh Shetty, the name of the developer of InsecureBankV2, signed the app. While this is sufficient information to identify the publisher, further verification is possible. The *apksigner* tool can be used to ensure that the build has been signed via both the v1 and v2 schemes for Android 7.0 (API level 24+), via all the three schemes for Android 9 (API level 28+), and that the code-signing certificate in the APK belongs to the developer. The following command verifies the APK signature: *apksigner verify --verbose InsecureBankv2.apk*, and provides the output captured in the below screenshot (Figure 10).

```
Verifies
Verified using v1 scheme (JAR signing): true
Verified using v2 scheme (APK Signature Scheme v2): false
Number of signers: 1
```

Figure 10. InsecureBankV2 Apksigner details

It makes sense that the APK cannot be verified with the v2 scheme because the target API level for InsecureBankV2 is 22, below API level 24 required for the v2 scheme. The *jarsigner* tool can be used to examine the contents of the signing certificate by running the command: *jarsigner -verify -verbose -certs InsecureBankv2.apk*, which provides the output captured in the below screenshot (Figure 11).

```
sm     7588 Fri Sep 18 12:20:42 EDT 2015 AndroidManifest.xml

      >>> Signer
      X.509, CN=Dinesh Shetty, OU=Services, O=SI, L=Boston, ST=MA
      [certificate is valid from 7/24/15, 4:37 PM to 7/17/40, 4:37 PM]
      [Invalid certificate chain: PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderEx
ception: unable to find valid certification path to requested target]
```

Figure 11. InsecureBankV2 Signing Certificate Details

If this was a debug certificate from Android Studio, the Common Name (CN) attribute would be set to "Android Debug". The “Invalid certification chain” error typically results from the use of a self-signed certificate.

In conclusion, the intent of this requirement is to ensure that a valid build release certificate exists for the app and sufficiently identifies the app publisher. While the lack of certificate details does not necessarily make the app less secure, the digital signature is used to verify the publisher's identity for application updates. Additionally, the signing

process can prevent an app from being tampered with or modified to include malicious code. Although the commands for this technical test case can be scripted, the output requires review by an analyst to determine if the app meets the security requirements.

Although few technical test cases were captured during this research, additional test cases can be created based on the MASVS Security Requirement to Tools Mapping in Appendix A. These could then be contributed to the public GitHub repository.

# 5. Areas for Further Research

There are additional areas of research that would help further the development of technical test cases for the MASVS and Android security assessment automation. Two of these are Applying Machine Learning to Technical Test Cases; and Testing Against Resilient Mobile Apps.

## 5.1. Applying Machine Learning to Technical Test Cases

Most of the technical test cases provided in this research produce output that must be manually reviewed to determine if the MASVS-L1 security requirement was met. It would be better if the output could be automatically analyzed and given a score that reflected the confidence that the MASVS-L1 security requirement was met. This problem could benefit from supervised machine learning. Supervised machine learning is when there are input variables and an output variable, and an algorithm is used to learn the mapping function from the input to the output. It is called supervised learning because the process of an algorithm learning from the training dataset can be compared to a teacher supervising the learning process. When using a training dataset, the correct answers are known beforehand, and the algorithm iteratively makes predictions on the training data and is corrected by the teacher. Learning stops when the algorithm achieves an acceptable level of performance. In the context of our problem, intentionally vulnerable mobile apps are the training dataset. We know beforehand which technical test cases will fail and why, and we can correct the algorithm until we achieve an acceptable baseline. Both a human subject matter expert and the algorithm would analyze the output, and if the algorithm makes an incorrect prediction, the human subject matter expert will correct it (Brownlee, 2019).

### 5.2. Testing Against Resilient Mobile Apps

Intentionally vulnerable mobile apps like InsecureBankV2 are not protected against reverse engineering and specific client-side attacks such as those specified in MASVS V8: Resilience Requirements. For example, MASVS 8.1 states that "The app detects, and responds to, the presence of a rooted or jailbroken device either by alerting the user or terminating the app" (OWASP, 2019). InsecureBankV2 was run in a compromised environment, a rooted emulator, and did not notify the user or terminate itself. Many real-world apps, especially those that process sensitive data, are developed with protections that impede dynamic analysis and tampering, device binding, and comprehension. A more realistic advancement of the technical test cases would be to run them against a range of resilient mobile apps to analyze how well they function and then address shortcomings in operation by incorporating conditional logic that accounts for resiliency protections (OWASP, 2019).

## 6. Conclusion

Security assessments of mobile apps on Android devices typically involve manual preparation, manual tool installation, and manual test execution. Since there is often a limited amount of time available to perform the testing, it is helpful to partially automate these tasks in order to free up analyst time to focus on tests that definitively require manual analysis. In order to perform a baseline security assessment that is consistent and repeatable and targets the correct level of protection, it would be helpful to test against a recognized standard that describes security requirements.

It was found that the use of an emulator makes it possible to programmatically create and destroy AVDs, which is a foundational requirement for building a solution that can scale when testing more than one mobile app at a time. Additionally, apps and tools were able to be installed and uninstalled from the command line, helping to create a security assessment process that is automated and repeatable.

In the absence of automated test scripts to drive the functionality of a mobile app, security tools such as Burp Suite and Drozer could be used to develop functional tests as well as automate the security assessment process. Mobile app security assessments are

performed to ensure that a mobile app has been developed according to security requirements. The MASVS-L1 standard is public and open; comprehensive, mature, developed by experts, and highly testable. Both tools and technical test cases are available for the majority of the MASVS-L1 security requirements, as illustrated in Appendix A, MASVS Security Requirement to Tools Mapping.

While working on this research, a few technical test cases for the MASVS-L1 requirements were partially or fully automated, and the bulk of the research is available on a public GitHub repository to be shared and contributed to by the wider information security community. As work continues in this research area, the repository is structured in such a way that scripts and security test cases can be continuously added and improved upon, leading to greater automation of the security assessment process. Additionally, a future application of a test harness to the MASVS-L1 scripts, the employment of machine learning, and incorporating improvements from testing against resilient mobile apps would help establish a minimum testing baseline and ensure consistency in mobile app security assessments.

# References

0xFireball. (2017). Rooting the Android Studio AVDs. GitHub repository, https://github.com/0xFireball/root_avd

Android-x86. (2019). Android-x86 Documentation. Retrieved November 6, 2019, from https://www.android-x86.org/documentation.html

Brownlee, J. (2019, August 12). Supervised and Unsupervised Machine Learning Algorithms. Retrieved November 16, 2019, from https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/

Google. (2019). Android Studio User Guide. Retrieved November 6, 2019, from https://developer.android.com/studio/intro

Horwitz, J. (2019, October 17). As iOS 13 hits 50% adoption, Android fragmentation keeps getting worse. Retrieved from https://venturebeat.com/2019/10/17/as-ios-13-hits-50-adoption-android-fragmentation-keeps-getting-worse/

Miller, J., (2019). Android App Testing. GitHub repository, https://github.com/laconicwolf/Android-App-Testing

MITRE Corporation (MITRE). (2016). Analyzing the Effectiveness of App Vetting Tools in the Enterprise. Retrieved from https://www.mitre.org/sites/default/files/publications/pr-16-4772-analyzing-effectiveness-mobile-app-vetting-tools-report.pdf

National Information Assurance Partnership (NIAP). (2016). Requirements for Vetting Mobile Apps from the Protection Profile for Application Software v1.2. Retrieved from https://www.niap-ccevs.org/MMO/PP/394.R/pp_app_v1.2_table-reqs.htm

Author Name, email@address

National Institute of Standards and Technology (NIST). (2019). Special Publication 800-163 Revision 1, Vetting the Security of Mobile Applications. Retrieved from https://doi.org/10.6028/NIST.SP.800-163r1

National Institute of Standards and Technology (NIST). (2017). Special Publication 800-53, Revision 5, Initial Public Draft, Security and Privacy Controls for Information Systems and Organizations. Retrieved from https://csrc.nist.gov/publications/draft-pubs

Open Web Application Security Project (OWASP). (2019). Mobile Application Security Verification Standard. GitHub repository, https://github.com/OWASP/owasp-masvs

Open Web Application Security Project (OWASP). (2018). Mobile Security Testing Guide v1.0. GitHub repository, https://github.com/OWASP/owasp-mstg/

Shetty, D., (2019). InsecureBankv2. https://github.com/dineshshetty/Android-InsecureBankv2

Wright, J., (2019). SANS SEC575. Mobile Device Security and Ethical Hacking. Books 575.1-575.5.

Wright, J., (2015). App Report Cards. GitHub repository, https://github.com/joswr1ght/MobileAppReportCard

Author Name, email@address

# Appendix A
# MASVS Security Requirement to Tools Mapping

| # | Security Requirement | Tools |
|---|---|---|
| 1.1 | All app components are identified and known to be needed. | N/A |
| 1.2 | Security controls are never enforced only on the client-side, but on the respective remote endpoints. | DAST: Burp Suite, Zap, FRIDA, or BRIDA in conjunction with manual or automated tests on client-side |
| 1.3 | High-level architecture for the mobile app and all connected remote services has been defined, and security has been addressed in that architecture. | N/A |
| 1.4 | Data considered sensitive in the context of the mobile app is clearly identified. | N/A |
| 2.1 | System credential storage facilities are used appropriately to store sensitive data, such as PII, user credentials or cryptographic keys. | SAST: JadX (Android Key Store use) |
| 2.2 | No sensitive data should be stored outside of the app container or system credential storage facilities. | SAST: Unzip & strings (classes.dex) |
| 2.3 | No sensitive data is written to application logs. | DAST: adb logcat w/ egrep |
| 2.4 | No sensitive data is shared with third parties unless it is a necessary part of the architecture. | SAST: AXMLPrinter2.jar (AndroidManifest.xml); JadX w/ strings, grep. DAST: Burp Suite or Zap |
| 2.5 | The keyboard cache is disabled on text inputs that process sensitive data. | SAST: JadX w/ strings, grep. DAST: manual input |

| # | Security Requirement | Tools |
|---|---|---|
| 2.6 | No sensitive data is exposed via IPC mechanisms. | SAST: AXMLPrinter2.jar (AndroidManifest.xml); JadX w/ strings, grep. DAST: Drozer |
| 2.7 | No sensitive data, such as passwords or pins, is exposed through the user interface. | SAST: JadX w/ strings, grep. DAST: manual input |
| 3.1 | The app does not rely on symmetric cryptography with hardcoded keys as the sole method of encryption. | SAST: JadX w/ strings, grep. |
| 3.2 | The app uses proven implementations of cryptographic primitives. | SAST: JadX w/ strings, grep. |
| 3.3 | The app uses cryptographic primitives that are appropriate for the particular use-case, configured with parameters that adhere to industry best practices. | SAST: JadX w/ strings, grep. |
| 3.4 | The app does not use cryptographic protocols or algorithms that are widely considered depreciated for security purposes. | SAST: JadX w/ strings, grep. |
| 3.5 | The app doesn't re-use the same cryptographic key for multiple purposes. | SAST: JadX w/ strings, grep. |
| 3.6 | All random values are generated using a sufficiently secure random number generator. | SAST: JadX w/ strings, grep. DAST: Burp Sequencer |
| 4.1 | If the app provides users access to a remote service, some form of authentication, such as username/password authentication, is performed at the remote endpoint. | DAST: Burp Suite |
| 4.2 | If stateful session management is used, the remote endpoint uses randomly generated session identifiers to authenticate client requests without sending the user's credentials. | DAST: Burp Sequencer |
| 4.3 | If stateless token-based authentication is used, the server provides a token that has been signed using a secure algorithm. | DAST: Burp Suite |

| # | Security Requirement | Tools |
|---|---|---|
| 4.4 | The remote endpoint terminates the existing session when the user logs out. | DAST: Burp Suite |
| 4.5 | A password policy exists and is enforced at the remote endpoint | DAST: Burp Suite |
| 4.6 | The remote endpoint implements a mechanism to protect against the submission of credentials an excessive number of times. | DAST: Burp Suite |
| 5.1 | Data is encrypted on the network using TLS. The secure channel is used consistently throughout the app. | DAST: Wireshark |
| 5.2 | The TLS settings are in line with current best practices, or as close as possible if the mobile operating system does not support the recommended standards. | DAST: Wireshark |
| 5.3 | The app verifies the X.509 certificate of the remote endpoint when the secure channel is established. Only certificates signed by a trusted CA are accepted. | SAST: JadX w/ strings, grep.<br>DAST: Burp Suite |
| 6.1 | The app only requests the minimum set of permissions necessary. | SAST: AXMLPrinter2.jar (AndroidManifest.xml); AAPT; ADB & dumpsys<br>DAST: Drozer |
| 6.2 | All inputs from external sources and the user are validated and if necessary sanitized. This includes data received via the UI, IPC mechanisms such as intents, custom URLs, and network sources. | SAST: JadX w/ strings, grep.<br>DAST: Content Query, Drozer, Burp Suite |
| 6.3 | The app does not export sensitive functionality via custom URL schemes, unless these mechanisms are properly protected. | SAST: AXMLPrinter2.jar (AndroidManifest.xml); JadX w/ strings, grep.<br>DAST: Drozer |
| 6.4 | The app does not export sensitive functionality through IPC facilities, unless these mechanisms are properly protected. | SAST: AXMLPrinter2.jar (AndroidManifest.xml); JadX w/ strings, grep.<br>DAST: Drozer |

| # | Security Requirement | Tools |
|---|---|---|
| 6.5 | JavaScript is disabled in WebViews unless explicitly required. | SAST: JadX w/ strings, grep.<br>DAST: Burp Suite |
| 6.6 | WebViews are configured to allow only the minimum set of protocol handlers required ( ideally, only https is supported ). Potentially dangerous handlers, such as file, tel and app-id, are disabled. | SAST: JadX w/ strings, grep.<br>DAST: Manual input |
| 6.7 | If native methods of the app are exposed to a WebView, verify that the WebView only renders JavaScript contained within the app package. | SAST: AXMLPrinter2.jar (AndroidManifest.xml); JadX w/ strings, grep.<br>DAST: Drozer w/ weasel |
| 6.8 | Object deserialization, if any, is implemented using safe serialization APIs. | SAST: JadX w/ strings, grep.<br>DAST: Xposed |
| 7.1 | The app is signed and provisioned with a valid certificate. | SAST: unzip, openssl, jarsigner, apksigner |
| 7.2 | The app has been built in release mode, with settings appropriate for a release build (e.g., non-debuggable). | SAST: AXMLPrinter2.jar (AndroidManifest.xml)<br>DAST: Drozer, Android Studio, jdb |
| 7.3 | Debugging symbols have been removed from native binaries. | SAST: nm, Hopper |
| 7.4 | Debugging code has been removed, and the app does not log verbose errors or debugging messages. | SAST: JadX w/ strings, grep.<br>DAST: ADB Logcat |
| 7.5 | All third-party components used by the mobile app, such as libraries and frameworks, are identified and checked for known vulnerabilities. | SAST: OWASP Dependency Check |
| 7.6 | The app catches and handles possible exceptions. | SAST: JadX w/ strings, grep.<br>DAST: Xposed |
| 7.7 | Error handling logic in security controls denies access by default. | SAST: JadX w/ strings, grep.<br>DAST: Xposed |
| 7.8 | In unmanaged code, memory is allocated, freed, and used securely. | SAST: JadX w/ strings, grep. |

| # | Security Requirement | Tools |
|---|---|---|
| | | DAST: Valgrind or Mempatrol (Native Code); Squares leak canary (Java/Kotlin); Android Studio Memory Profiler; Android Java Deserialization Vulnerability Tester. |
| 7.9 | Free security features offered by the toolchain, such as bytecode minification, stack protection, PIE support, and automatic reference counting, are activated. | SAST: JadX, JD-GUI (Identify obfuscation) |