DARK WOLF
SOLUTIONS

Android Security Research Playbook

Version 1.0

February 2024

Authors:

Hahna Latonick, Director
Jonathan Waterman, Principal
Matthew Bianchi, Senior Associate
Jacob Swinsinski, Associate

Dark Wolf Solutions, LLC

Dark Wolf Solutions operates at the nexus of mission and technology to meet our Nation's most challenging missions. We combine the most innovative emerging technologies with deep federal domain expertise through cutting-edge intelligence services, DevSecOps agile software development, information operations, penetration testing and incident response, applied research and rapid prototyping, machine learning, and engineering services.

We support a diverse portfolio of solutions and services for Defense, Intelligence, and Fortune 500 customers. Our team comprises analysts, support officers, and experienced engineers and integrators with hands-on expertise across many of the most relevant COTS, GOTS, and open source technologies. We also regularly compete in premier competitions with an increasing number of conference wins and placements including IoT, Wireless and OSINT CTFs at DEF CON, Wireless CTFs at BSides, Splunk Boss of the SOC and the Navy's HACKtheMACHINE.

The following were awarded to Dark Wolf Solutions in the 2023 DEF CON 31 Capture the Flag competitions in Las Vegas, Nevada:

IoT	Kubernetes	Embedded Systems
1st Place	1st Place	2nd Place

For more information please visit us at: https://www.darkwolfsolutions.com.

Table of Contents

Android Security Research Playbook

Android security research plays a major role in the world of cybersecurity that we live in today. As of 2024, Android has a 71.74% global market share of mobile operating systems' according to Stat Counter. There are presently 3.3 billion Android OS users in the world according to Business of Apps. With the advent of new tools and frameworks, understanding the landscape of Android-based vulnerabilities and exposures has become more important than ever before.

We've created the Android Security Research Playbook (ASRP) as a getting started guide for security researchers to follow, equipping them with the tools and processes to conduct their research successfully. The ASRP isn't all-inclusive of every tool, approach, or method that can be employed; but, it provides recommendations for each stage of the process to help security researchers progress with their research. The stages of the ASRP are summarized below.

Reconnaissance

The *Reconnaissance* phase consists of gathering information relevant to the target application, understanding its overall purpose and capabilities, and then mapping out its attack surface.

Static Analysis

The *Static Analysis* phase allows us to utilize a wide array of open-source tools to dig deeper into the design and implementation of our target application without the need to execute it. Tools discussed include Android Studio, JADX, MobSF, and more.

Dynamic Analysis

The *Dynamic Analysis* phase lets us examine the runtime behavior of the target application. Tools discussed include Frida, Objection, and more.

Vulnerability Discovery

The *Vulnerability Discovery* phase explores classes of vulnerabilities that can be found in a target application, including both zero-day and N-day vulnerabilities.

Fuzzing

The *Fuzzing* phase enables us to automatically identify vulnerabilities at scale by sending crafted inputs to a target and then monitoring for crashes due to the application making incorrect assumptions about the user-supplied data or not properly accounting for it.

Bypassing Exploit Mitigations and Security Protections

The *Bypassing Mitigations and Security Protections* phase describes mechanisms that may need to be defeated to successfully reverse engineer the target and exploit its vulnerabilities.

Exploitation

The *Exploitation* phase describes possible cyber effects that can be achieved by exploiting the vulnerabilities identified, demonstrating the severity and impact of the vulnerabilities.

Lab Environment Setup

It is crucial to have a proper lab environment to perform Android security research effectively and efficiently. This section describes the recommended hardware and software to use for Android security research. This is not an exhaustive list and should not be considered as mandatory requirements to conduct your research. Use what you are most comfortable with and what is best to meet your needs and goals.

Hardware Recommendations

We recommend the following hardware for conducting Android security research.

Host Computer	Intel-based CPU (i5 or above) with at least 8 cores 32GB RAM 1TB+ SSD NVIDIA GPU (RTX or similar)
Real Android Device (optional)	Samsung or Google Pixel with Android 11+

Rooting Physical Android Devices

Some of the Plays and tools described in the ASRP will require a rooted Android device. Rooting is the process by which users of Android devices can attain privileged control (known as root access) over various subsystems of the mobile device. Since Android is based on a modified version of the Linux kernel, rooting an Android device gives similar access to administrative (superuser) permissions as on Linux or any other Unix-like operating system.

Rooting is often performed to overcome limitations that carriers and hardware manufacturers put on some devices. Thus, rooting gives the ability to alter or replace system applications and settings, run specialized applications that require administrator-level permissions, or perform other operations that are otherwise inaccessible to a standard user. It is worth noting that rooting is distinct from SIM unlocking and bootloader unlocking. The former allows removing the SIM card lock on a phone, while the latter allows rewriting the phone's boot partition.

Gaining root on a device is slightly different for each phone manufacturer and even model. Therefore, when following the ASRP, we recommend using an Android emulator, since their emulated devices are often rooted by default.

Software Recommendations

We recommend the following software for conducting Android security research.

Host Computer	Virtual Machine
Windows 10 or above	Latest Ubuntu LTS Operating System (OS)
Latest Android Studio	Security Research Tools (described in later sections)
Latest VMware Workstation Pro	Git
Emulation Software (see section below)	Latest Android Studio
Emulated Android devices	Burp Suite Community Edition

Emulation Software

For emulation, we recommend using Genymotion, Android Studio, or Corellium. All three of these products provide the most up-to-date and relevant features for Android security research.

Genymotion provides a wide array of emulated sensors and features, as well as a fast startup time for both booting and running applications on the emulated device. It does require setting up an account and a license for professional use after a free trial on Linux and Windows. Genymotion emulated devices are also rooted by default.

Android Studio is the official IDE for Google's Android OS, but it also has an amazing emulation feature built into it. This feature allows the user to create an AVD (Android Virtual Device), and for each AVD the user can specify a hardware profile, system image, and more. AVD images can be rooted using the following article.

Corellium provides a "Virtual Hardware Platform", which allows its users to emulate primarily ARM-based devices with very high-fidelity. The interface is easy to use, and provides additional hardware-specific features and capabilities than Genymotion and Android Studio. Corellium emulated devices are also rooted by default.

Virtualization Software

VMware Workstation Pro lets you run multiple operating systems as virtual machines (VM) on a single Windows or Linux computer. A paid license is required after its free trial. Here is a tutorial on how to create an Ubuntu VM using VMware Workstation Pro.

VirtualBox is a free alternative to VMware Workstation Pro and comes bundled with Genymotion; however, it does not come with nearly as many features. Here is a tutorial on how to create an Ubuntu VM using VirtualBox.

Introduction to Android

Android Architecture

Android is an open-source, Linux-based operating system for a wide array of devices and form factors. Its documentation and source code are available through their Android Open Source Project (AOSP). The major components of the Android software stack are shown in **Figure 1**.

An Android application ("app") is an app created using the Android API. The Google Play Store is often used to find and download Android apps, though there are other alternatives.The *System App*s are pre-installed on Android, providing essential functionality, such as the dialer, messaging, and system settings. These core apps can be used directly and their capabilities can be invoked by third-party apps. The *Java API Framework* is a set of Java APIs that developers can use to build Android apps. The *Native C/C++ Libraries* provide key functionality, such as multimedia support and graphics rendering. The *Android Runtime* provides a runtime environment that executes Android applications. The *Hardware Abstraction Layer (HAL)* is a set of interfaces that allows the Android OS to communicate with the underlying hardware components. The *Linux Kernel* provides the core system services, such as memory management, process management, networking, and security.

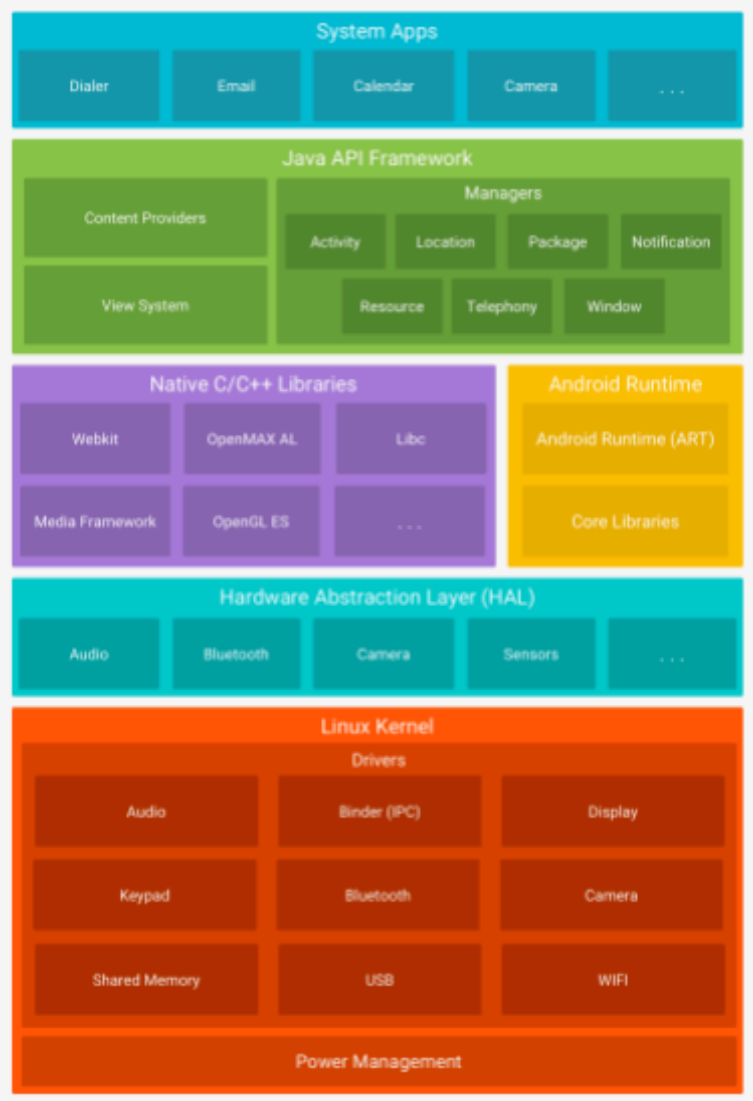


Figure 1. Android Software Stack.

Android Applications

Android applications are commonly developed in Kotlin, Java, and C++ languages. Android provides an SDK to compile the app's code, data, and resources into an Android Package Kit (APK) or an Android App Bundle (AAB). Each Android app operates in its own security sandbox and only has access to the components and resources that it needs for its operations.

Android apps consist of four component types: *Activities*, *Services*, *Broadcast Receivers*, and *Content Providers*. Activities represent the user interface screens and handle user interactions. Services perform background tasks and can run even when the user is not interacting with the app. Broadcast receivers listen for and respond to system-wide events. Content providers manage and share data between different applications.

Android uses *Intents* to start activities, services, and broadcast receivers. It serves as a mechanism for communication between components within an Android application or between different applications. For activities and services, an intent defines the action to perform. For broadcast receivers, the intent defines the broadcast announcement. To learn more about how intents work, we recommend reviewing Android's Intents and Intent Filters documentation.

An APK includes an *AndroidManifest.xml* file, which declares the app's components, permissions, hardware requirements, and API libraries. The APK also includes resources, such as images, audio files, and anything relating to the app's visual presentation. Resources are stored separately from the app's source code; thus, developers can specify alternative resources to optimize their app for different device configurations without code modifications.

Android strives to be the most secure and usable operating system for mobile platforms. They often customize traditional operating system security controls to protect app and user data; protect system and network resources; and, provide app isolation from the system, other apps, and from the user. To that end, Android provides several key security features, including robust security at the OS level through the Linux kernel, mandatory app sandbox for all apps, secure interprocess communication, app signing, app-defined and user-granted permissions, and more. They also provide a security bulletin describing issues affecting Android devices and possible fixes. We recommend reviewing their security documentation to learn more about its security model and features as it will be crucial to your Android security research. In addition, it is worthwhile to become familiar with the OWASP Mobile Top 10, described in Appendix A, which further elaborates on the most critical security risks in mobile applications. This concludes the introduction to Android. Let's dive into the first stage of the Playbook: Reconnaissance.

Reconnaissance

Reconnaissance is the first stage of the process and involves researching your target before carrying out any analysis. This step will enable us to understand our target, its history, its capabilities, and any known vulnerabilities. The collection of this information will help define our target's attack surface and inform our security research strategy.

PLAY 00: Gather Open-Source Resources

Researchers will conduct **Open-Source Intelligence** (**OSINT**) on the target and document any public information found. This will help us understand how our target works. We want to gather as much information as possible in order to help craft a target strategy, using publicly available and relevant research to our advantage. This play is a quick method to curate a multitude of resources. We do not actually need to dive into the information until later plays.

Prerequisites	• Any operating system with a browser • Internet access
Process	• Use available search engines to gather open-source resources. • Explore relevant areas of the OSINT framework, if necessary, to collect additional information.
Result	• Curated list of information and resources pertaining to our target.

PLAY 01: Contextualize the History of the Target

It is important to understand the history of our target from why our target exists in the first place to how it has evolved to the present day. This will help illuminate design, implementation, and security decisions made by the developers of the target software. This information will reveal how often the target is updated, its complexity, and what security mitigations may need to be bypassed. In addition, the collection of this information will start to reveal the target's attack surface and potentially new areas to investigate for vulnerabilities.

Prerequisites	• Internet access • Utilize the resources found during the OSINT stage
Process	• Begin going through the list of compiled resources and summarize the application's history • Make note of why the changes were made or introduced
Result	• An outline of the target's history and evolution. This can be captured in a text document, spreadsheet, or wiki page.

PLAY 02: Discover Any Known CVEs

By utilizing previous security research, it will show us areas of the target that have been actively exploited and gives us an opportunity to see what can be possibly demonstrated against the target. Relevant CVEs will be evaluated in later plays to ensure that proper mitigations were implemented to address the reported vulnerabilities.

Prerequisites	• Internet access
Process	• Utilize search engines and explore the **Exploits & Advisories** section within the OSINT Framework • Investigate the target's bug/security tracking system, if applicable • Document your findings
Result	• List of relevant CVEs

PLAY 03: Curate List of Capabilities

It is important to understand the target's features and capabilities. For example, what is the target capable of? What does it offer the user? How does it transmit information, if any?
This will give us an idea of the target's attack surface and shape our security research strategy.

Prerequisites	• Internet access
Process	• Research documentation on the target's main website. • Explore 3rd party sites or product reviews (if applicable)
Result	• Curated list of capabilities

PLAY 04: Find Sources for the APK

We will obtain our target's Android Package Kit (APK) file, which will enable us to perform static and dynamic analysis, as described in later sections.

Prerequisites	• Internet access to visit APKMirror, APKPure or APKCombo
Process	• Search for the target APK on either one of the platforms above. This will allow you to instantly download the latest version of the application that is available on the Google Play Store.
Result	• Downloaded APK on system

Static Analysis

Static analysis is a method of examining source code or binary code without executing the program. This can be performed manually or by using automated analysis tools. There are different objectives for static analysis, including understanding the design and implementation of a target, and identifying possible security weaknesses or vulnerabilities. This section details several static analysis techniques to facilitate your security research.

PLAY 05: Understand the APK's Design and Compiler: APKiD

APKiD is like PEiD, but for APK files, outputting what compilers, packers, obfuscators, and security mechanisms are used by the APK. This play illuminates the APK's design and any potential issues that may need to be addressed based on its design elements. To learn more about APKiD, we recommend this official presentation or its OWASP resource page.

Prerequisites	• Follow the APKiD installation instructions.

Process	• Run APKiD on the APK file:
	Unset
	`$ apkid -r --scan-depth 4 target.apk`
	• For each classes.dex file, record any compiler, packers, and exploit mitigations detected by APKiD, including:
	  ○ Compiler Info - "R8 without marker"
	  ○ Anti-Debug - `detectDebugger(), isDebuggerConnected()`
	  ○ Anti-VM - Build checks, SIM checks, etc.
	  ○ Packers - AppGuard, Dexprotector, etc.

Results	• A list of design decisions that could potentially affect target analysis.

⚠️Important⚠️

Obfuscators and packers used by the APK can prevent or hinder the static analysis process, including reverse engineering (RE); therefore, we recommend defeating these protections by completing the Anti-RE bypass play.

PLAY 06: Decompress the APK: APKTool

APKTool lets you quickly familiarize yourself with the general structure of an APK file. Android applications are usually compressed into an APK file, so we will decompress it to review its files.

Prerequisites	• Follow the APKTool installation guide
Process	• Run the following command on the target APK: Unset `$ apktool d target.apk` • To see which `classes.dex` files are present, use the `-s` flag.
Results	• A decompressed version of the target APK

For additional information on using APKTool, the official APKTool documentation.

PLAY 07: Decompile the APK: JADX

JADX is a Dex-to-Java decompiler that comes equipped with automatic decoding and deobfuscation features. Its tools can be run on the command line or via its GUI.

Prerequisites	• Install JDK 11+ on your system (check with `java -version`).
	  ○ Linux: `$ sudo apt-get install openjdk-11-jdk`
	  ○ Windows: Download here
	• Follow the JADX Build From Source instructions.
	• Navigate to the `/build/jadx/bin` directory

Process	• Run `/build/jadx/bin/jadx-gui` to open up the JADX GUI • Select the option to *Open up a File* • Go to *File->Preferences*, and make sure under *Decompilation* that *Code Comments Level* is set to DEBUG • Go to *Tools->Decompile All Classes* • Once the APK is decompiled, save this project onto your system as a **Gradle project folder** by going to *File->Save As Gradle Project*
Results	• A decompiled version of the target APK saved to your system as a Gradle project folder

PLAY 08: Examine the APK File Structure: Android Studio

Android Studio is an IDE created by Google for Android application development. It easily displays an APK's file structure, allowing us to quickly understand its organization and content. We will use Android Studio to load the Gradle project from the previous play in an environment that is best suited for Android development and testing.

Prerequisites	• Saved Gradle Project • Install Android Studio

Process	• Upload the Gradle project as a new project in Android Studio. Make sure to account for which language it's made in (Kotlin or Java), and download any pertinent SDK files beforehand
	• Verify that all of the original APK files are present by comparing the file structure on the left with the file structure displayed in JADX.
	• Open the drop down menu for `main/`
	• Observe the file structure, and then document any useful locations or patterns that appear to be present:
	  ◦ `AndroidManifest.xml` will likely be placed outside of any folder structures. Document the activities listed in the file, including their permissions and related source code folders.
	  ◦ `res/` contains resource files in a variety of formats like .xml, .webp, .gif, .json, etc.
	  ◦ `assets/` can sometimes have the most variance depending on the app, but will likely contain the visual content displayed by the app
	  ◦ `java/` or `kotlin/` is where your source code will be, and JADX will have attempted to deobfuscate any renaming due to identifier remapping.
	  ◦ `lib/` will contain any shared object library files, make note of any file names that stand out or seem unnecessary for the purposes of the app
	  ◦ Look for any `classes.dex` files floating in the `main/` directory folder.

Results	• An understanding and familiarity with the APK file structure. • A list of interesting target locations for further analysis.

For supplemental analysis, you can use JADX or Ghidra to further review the APK's file content.

PLAY 09: Analyze the APK's Library Files: MobSF

Mobile Security Framework (MobSF) is a security research platform for mobile applications. One of its best features is statically analyzing library files and reporting potential vulnerabilities.

Prerequisites	• Install Docker. For example, run the following Linux commands:
	Unset
	`$ sudo apt install -y docker.io`
	`$ sudo gpasswd -a $USER docker`
	`$ sudo usermod -aG docker $USER`
	`$ sudo reboot`
	• Pull down the MobSF Docker image per the repo's instructions.

Process	• Initialize MobSF by running the following Docker command: Unset `$ docker run -it --rm -p 8000:8000 opensecurity/mobile-security-framework-mobsf:latest` • Wait for the output to slow down and then open up the MobSF's web interface at `http://127.0.0.1:8000` in a web browser. • Drag the original APK file over to the "Upload & Analyze" screen on the web page. • Once MobSF finishes analyzing the application, save a copy of the PDF report detailing its static analysis results.
Results	Report of static analysis results, including: • File Information, Certificate Analysis, and App Components • Manifest and Code Analysis • Shared Object Library File Analysis • Communication Info: OFAC Sanctioned Countries, Domain Malware Checks, and General Network Security • Hardcoded Emails and Secrets

PLAY 10: Reverse Engineer Library Files: Ghidra

Ghidra is a multi-purpose, open-source reverse engineering tool that can help researchers statically analyze APK files with ease. Android APKs use shared libraries for several reasons, including code reuse, efficiency, and reducing the size of the APK itself. Shared libraries are compiled binary files that contain reusable code and resources that can be used by multiple applications. They can include native code written in languages like C or C++, as well as resources like images, fonts, or data files. Unfortunately, shared libraries may not be updated often, leaving the APK vulnerable to exploitation.

Prerequisites	• Install JDK 17 64-bit by either going here or running the commands:
	Unset
	`$ sudo apt update`
	`$ sudo apt upgrade`
	`$ sudo apt install openjdk-17-jdk openjdk-17-jre`
	• Download the latest Ghidra software.
	• Unzip the Ghidra ZIP archive
	• Change into the Ghidra software directory
	• (Linux) Make the *ghidraRun* script executable: `$ chmod +x ghidraRun`

Process	• Launch Ghidra using `./ghidrarun` • Create a new project folder for the APK • Load the APK file into your project • Navigate to the lib/ directory • Open up a shared library file inside the Code Browser tool • Run the automatic analysis on the file using its default selections • Review its Strings, Function Calls, Imports, Exports, Symbols, Structures, and Enumerations listings • When analyzing functions, analyze them in the disassembler, decompiler, and in graph view. Make note of cross-references. • Identify any known insecure or banned LIBC functions being called • Analyze any potentially vulnerable memory allocations. • Verify results from other static analysis tools
Results	• Areas of interest and potential vulnerabilities found within the APK's library files to further reverse engineer

Dynamic Analysis

Dynamic analysis is a method of examining source code or binary code while executing the program. There are different objectives for dynamic analysis, including understanding the target's runtime behavior, debugging, and verifying security weaknesses or vulnerabilities. This section details several dynamic analysis techniques to assist your security research.

⚠️Important⚠️

An APK may include security protections that can prevent or hinder the dynamic analysis process; therefore, we recommend defeating these protections by completing the following Plays first: APKiD, Anti-VM, Anti-Debugging, Root Detection, and Anti-RE.

PLAY 11: Establish Communication with the Android Device

Android Debug Bridge (ADB) is a command-line tool that lets you communicate with an Android device. It is included in the Android SDK Platform Tools package or can be installed individually.

Prerequisites	Select one of the methods below to successfully install ADB. *Method #1: Package Manager* ● In Linux, run the command: `$ sudo apt install adb` *Method #2: SDK Manager* ● Use the `sdkmanager` CLI tool to install ADB. *Method #3: SDK Platform Tools* ● Download and install the latest SDK Platform Tools. Method #4: Android Studio (AS) ● Use the AS SDK Manager to install the Android SDK Platform Tools.
Process	**Physical Device:** ● Enable USB debugging in your device's developer settings. ● Plug in the device via USB cable. ● Accept any pop ups and authenticate any required access ● Verify that you are connected to the target device: `$ adb devices` **GenyMotion Device:** ● Obtain the IP address of your virtual device while it's running, ● Connect to the emulated device: `$ adb connect ip_address:port` ● Verify that you are connected to the target device: `$ adb devices`
Results	● Can successfully connect to and communicate with the device.

PLAY 12: Download and Install Frida

Frida is a dynamic code instrumentation toolkit that lets you inject snippets of JavaScript or your own library into native apps running on a variety of operating systems. Many popular dynamic analysis tools for Android utilize Frida and require the Frida Server to be installed on the device. Frida will provide great introspection throughout the dynamic analysis process.

<table>
<tr><td>Prerequisites</td><td>

A rooted Android device (physical or Genymotion emulated device).
Install Frida's CLI tools: <code>$ pip install frida-tools</code>
Install the Frida-Server version for your specific Android platform (e.g., <code>frida-server-16.1.10-android-x86.xz)</code>
Rename the downloaded file to <code>frida-server</code>.
Make <code>frida-server</code> executable: <code>$ chmod +x frida-server</code>
Connect to the Android device (see this Play)
Use ADB to push the <code>frida-server</code> file into the <code>/data/local/tmp/</code> directory of your Android device:

<pre>Unset
$ adb push frida-server /data/local/tmp</pre>
</td></tr>
<tr><td>Process</td><td>

For a physical device, execute these commands to run frida-server:

<pre>Unset
$ adb shell
$ su
$ setenforce 0
$ /data/local/tmp/frida-server -D &
$ setenforce 1</pre>

For an emulated device, run frida-server using these commands:

<pre>Unset
$ adb shell
$ /data/local/tmp/frida-server &</pre>

Verify that frida-server is running: <code>$ adb shell ps | grep frida</code>

</td></tr>
<tr><td>Results</td><td>

An Android device with Frida-Server running

</td></tr>
</table>

PLAY 13: Install Target App on the Android Device

Android applications typically release an official version of their APK on the Google Play Store, but websites like APKMirror and APKPure also provide a resource to download APK files. To perform dynamic analysis of the application, we need to install it on the Android device.

Prerequisites	*Method #1 Prerequisites: Install App via Google Play Store* ● For Genymotion emulated devices, install GAPPS on your device first and reboot it to gain access to the Google Play Store. ● A valid Google account on your Android device to use the Google Play Store app. ● Ability to open and use the Google Play Store on your device. *Method #2 Prerequisites: Install App via ADB* ● ADB ● Locate the previously downloaded APK file (see this play). *Method #3 Prerequisites: Install App via Drag and Drop* ● Genymotion emulated device ● Locate the previously downloaded APK file (see this play).
Process	*Method #1: Install App via Google Play Store* ● Launch the Google Play Store and sign in with your Google account. ● Search for the target app, install it, and verify that it loads properly. *Method #2: Install App via ADB* ● Navigate to the previously downloaded APK file. ● Connect to the Android device. ● Install it using ADB: `$ adb install file.apk` *Method #3 Prerequisites: Install App via Drag and Drop* ● Drag and drop the previously downloaded APK file onto the home screen of your Genymotion device.
Results	● Successfully installed the target application on the Android device.

PLAY 14: Explore APK Runtime Behavior: Objection

Objection is a runtime mobile exploration toolkit, powered by Frida, that helps you assess the security posture of mobile applications without needing a jailbreak. It enables users to easily navigate through different activities, invoke them and even pass in arguments to them. Some of the more impressive capabilities include adding in Frida scripts, dumping memory-related information, installing additional plugins, monitoring activities, and hooking classes or methods.

Prerequisites	• Frida server installed and running on target device (see this play) • ADB installed and connected to the target device • Install dependencies: `$ sudo apt install python3 python3-pip` • Install Objection: `$ pip3 install objection`

Process	• Connect to the device using ADB
	• Grab the app identifier by running `frida-ps -Uai`
	• Run `objection --gadget {TARGET_APP} explore`
	• In Explore mode, here are additional options to execute:
	  ○ `env` lists any environment related file paths for the application
	  ○ `Import` brings in Frida scripts and loads them as jobs that run in the background
	  ○ `jobs` lets you list and kill any running scripts or “jobs”
	  ○ `android` lets you do many things:
	  ■ `hooking` lists out activities, receivers, services, classes, methods, and their parameters, watch specific function calls, and set return values for any given function
	  ■ `heap` shows you live instances of a Java class
	  ■ `keystore` or `intent` lists keystore information and use intents to launch activities or services.
	  ○ `memory` lets you dump, write, and search memory

Results	• Additional information pertaining to memory, activities, environment, keystores, or intents • Validate any suspected vulnerabilities by running Frida scripts or hooking functions manually

PLAY 15: Perform Real-time Monitoring: RMS

The Runtime Mobile Security (RMS), powered by Frida, is a powerful web interface that enables users to manipulate Android applications at runtime. With RMS, you can easily dump all loaded classes and relative methods, hook everything on the fly, trace method arguments and return values, load custom scripts, and more.

<table>
<tr><td>Prerequisites</td><td>

Frida CLI tools installed
Frida server installed and running on the target device
ADB installed and connected to the target device
Install software dependencies:

<pre>Unset
$ sudo apt install npm
$ wget -qO-
https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.1/in
stall.sh | bash
$ source ~/.bashrc
$ nvm install v20.5.0
$ nvm use v20.5.0
$ npm install -g rms-runtime-mobile-security</pre>

Install RMS: <code>$ npm install -g rms-runtime-mobile-security</code>

</td></tr>
<tr><td>Process</td><td>

Connect to the device using ADB.
Run <code>rms</code> at the command line to launch the tool.
Using a web browser, navigate to <code>http://127.0.0.1:5491/</code>
Select <code>Android</code> from the dropdown for the OS.
Type in the package name and it should try to autocomplete it.
Select <code>Spawn</code> in the event that the app is not running. If you already have the app running, you can <code>Attach</code> to its running instance.
Next load or write any Frida scripts you want the application to be initiated with (ex: if you need to do root bypass to open the app)
Lastly, click <code>Start RMS</code>

</td></tr>
<tr><td>Results</td><td>

Successfully observe, hook, and modify the apps runtime behavior.

</td></tr>
</table>

PLAY 16: Monitor and Intercept Web Traffic: Burp Suite Proxy

It is common for Android applications to interact with the Internet or web services, generating web traffic when requesting or receiving the desired data. To better understand what data is being communicated and how, we will utilize the Burp Suite Proxy, an intercepting proxy server that sits between a user's web browser and a target web server. Its primary purpose is to intercept, inspect, and manipulate HTTP and HTTPS traffic between the client (browser) and the server, allowing security researchers to analyze, test, and modify web requests and responses.

Prerequisites	• Install Burp Suite Community Edition in Linux VM
	• Configure Burp Suite
	  ○ Click the **Settings** tab and navigate to **Tools** > **Proxy**,
	  ○ Within Proxy Listeners, ensure that you are listening on the **localhost address (127.0.0.1)** and your port matches the one within your device's or your web browser's proxy settings (**default is 8080**).
	  ○ Go to the **Proxy** tab and toggle on "**Intercept is off".** This will enable you to begin intercepting traffic
	• Install and start FoxyProxy extension for your browser.
	• Obtain Burpsuite CA Certificate
	  ○ This will be the certificate that you will use to bypass pinned certificates on the target app to analyze outbound requests.
	• Wappalyzer

Process	• In Burp Suite, go to the **Proxy/Intercept** tab. This will allow you to intercept, modify, and pause any inbound/outbound requests. • Interact with the target's web server, its web applications, or a general webpage to observe the target's behavior. ○ For example, if the Android app is Chrome, open Chrome and navigate to a web page. • Confirm that the Burp Proxy works properly.
Results	• Successfully monitor and intercept web traffic using Burp Proxy

For more information on how to use Burp Proxy, we recommend reviewing its documentation.

PLAY 17: Enumerate Web Server and Web Applications

Many Android applications use a web server or incorporate web server functionality as part of their features or architecture. Web server integrations allow for various functionalities, including communication with remote services, data synchronization, real-time updates, and more. Enumerating web servers and web applications is an essential step in the process of assessing their security. Enumeration involves systematically gathering information about the target web server and its applications. This information can be crucial for identifying vulnerabilities, potential attack vectors, and the overall attack surface.

Prerequisites	• Complete the prerequisites of this play.
Process	• In Burp Suite, go to the **Proxy/Intercept** tab. • View the site's source code and identify any useful information (passwords, keys, version numbers, email addresses, links, subdomains, vhosts, etc.). • Utilize Wappalyzer to identify what technology is running on target-related websites. • Interact with the web server and its web applications in normal and unexpected ways. • Perform a Directory Bruteforce using Dirsearch with a wordlist from the SecLists repo in an attempt to uncover hidden directories. Check if the site has a **robots.txt** page to uncover other hidden directories. • Crawl the website using web crawling tools (e.g., wget, Burp Suite Spider, or web application scanners) to discover additional pages. • Perform DNS Enumeration using *dnsrecon*, *Sublist3r*, and *Amass.* • Perform Port Scanning using tools like Nmap. • Perform Banner Grabbing using curl, telnet, or automated scripts. • Perform HTTP Method Enumeration for supported HTTP methods. • Perform Authentication and Session Enumeration (e.g., login pages). • Enumerate content types like scripts, stylesheets, images, and APIs. • Determine SSL/TLS configurations and certificate details.
Results	• Note what data is communicated between sources and destinations

The following article further describes how to enumerate applications on web servers.

PLAY 18: Debugging Programs with GDB

The GNU Project Debugger (GDB) is an industry standard debugging tool for C/C++ programs. It allows you to see what is happening under the hood of a program while it executes, manipulate its runtime behavior, and perform crash analysis when it terminates unexpectedly. GEF and Pwndbg are GDB plugins that provide additional features for security research. In addition, GDB Multiarch is a version of GDB which supports multiple target architectures.

Prerequisites	● Install GDB Multiarch: ○ Linux: `$ sudo apt-get install gdb-multiarch` ○ Windows: Use Cygwin to install the gdb-multiarch package ● Install GEF or PwnDbg
Process	● Pick a target binary from your APK's file structure ● Run GDB on the binary: `$ gdb-multiarch ./binary` ● Gather information using the *info* command ● Set breakpoints on target functions or memory addresses: *b main* ● Start the program with *run*, can give it user input like "*run AAAA"* ● Analyze runtime behavior: ○ Familiarize yourself with the control-flow of the binary ○ Observe its function calls. ○ Monitor its register state. ○ Examine memory locations or register values with *x*, and use modifiers like */x*, */o*, */u*, */t*, and */d* to switch the formatting
Results	● Successful dynamic analysis of the target binary using GDB.

For more information on using GDB, check out this tutorial.

PLAY 19: Setup Remote Debugging: GDB Server

GDB Server makes it possible to remotely debug other programs. Running on the same target system as the program to be debugged, it allows GDB to connect from another system (“host”). The connection can be either TCP or a serial line.

<table>
<tr><td>Prerequisites</td><td>

Internet access
GDB installed in Linux VM

</td></tr>
<tr><td>Process</td><td>

Download a prebuilt GDB Server from gdb-static or gdb-static-cross.
Make <code>gdbserver</code> executable: <code>$ chmod +x gdbserver</code>
Use ADB to push <code>gdbserver</code> onto the target Android device
Get the PID of your running Android app: <code>$ ps -A | grep <app></code>
Attach <code>gdbserver</code> to the Android app, listening on an arbitrary port:

<pre>Unset
$ ./gdbserver :1234 --attach <pid of app> &</pre>

Run GDB: <code>$ ./gdb-multiarch</code>
Run the GDB command: <code>$ (gdb) target remote :1234</code>
Observe the connection to the GDB server
Proceed with debugging the target Android application

</td></tr>
<tr><td>Results</td><td>

Successfully perform remote debugging using GDB and GDB Server.

</td></tr>
</table>

Vulnerability Discovery

Vulnerability discovery is the process of analyzing software, hardware, protocols, or algorithms for the purpose of identifying one or more security vulnerabilities. Vulnerabilities occur due to improper design, insecure coding, misconfigurations, and assumptions made about system/software usage. A vulnerability is a bug that renders the system/software vulnerable to attack; therefore, not all bugs (defects) are vulnerabilities.

Vulnerability research can be performed manually or by using automated analysis tools. There are different objectives for vulnerability discovery, such as to help make systems and software more secure, or to develop capabilities in support of offensive cyber operations. This section details several vulnerability classes to investigate during your security research. While running through these plays, it is strongly suggested to analyze code for any banned functions.

PLAY 20: Verify Buffer Overflow Vulnerabilities

A buffer overflow occurs when the amount of data in the buffer exceeds its allocated size, overflowing into adjacent memory regions. If the attacker can control the return address as a result of the buffer overflow, then they will be able to divert the flow of code execution to something of their choosing. This can potentially lead to remote code execution, arbitrary data read/write, and other effects.

Prerequisites	• Reverse Engineering Tool (e.g., Ghidra) • Access to source code or decompiled code, if possible
Process	• Investigate memory allocations • Investigate LIBC function calls that utilize memory-allocated variables, arrays, or structures • Determine if the length of the source data being copied exceeds the size of the destination buffer.
Result	• Confirmed or ruled out possible buffer overflow vulnerabilities

The following article illustrates buffer overflow vulnerabilities and how they can be exploited.

PLAY 21: Verify Integer Overflow Vulnerabilities

Integer overflows occur when the result of an arithmetic operation on integers exceeds the maximum representable value for the data type being used. In most programming languages, integers are represented using a fixed number of bits, and there is a limit to the maximum value that can be stored. When the result of an operation exceeds this limit, the overflow occurs, and the value "wraps around" to the minimum representable value. This can lead to unexpected and potentially erroneous behavior in a program. Integer overflows can also lead to buffer overflows if not properly handled.

Prerequisites	• Reverse Engineering Tool (e.g., Ghidra) • Access to source code or decompiled code, if possible
Process	• Investigate arithmetic operations like addition, subtraction, multiplication, or division on integers. • Investigate if integers are properly calculated and utilized for memory allocations
Result	• Confirmed or ruled out possible integer overflow vulnerabilities

The following article gives an example of an integer overflow and how it can be exploited.

PLAY 22: Verify Write-What-Where Vulnerabilities

A write-what-where vulnerability enables an attacker to perform arbitrary writes to an attacker-controlled memory location. This type of vulnerability can arise due to programming errors, such as buffer overflows or other memory corruption issues. When a program fails to properly validate or sanitize input, an attacker might be able to provide specially crafted data that can overwrite memory locations with arbitrary values. If the attacker can control both the content and the destination of the write operation, it can lead to the exploitation of the system.

Prerequisites	• Reversing Engineering Tool (e.g., Ghidra) • Access to source code or decompiled code, if possible
Process	• Investigate memory allocations • Investigate LIBC function calls that utilize memory-allocated variables, arrays, or structures • Identify if memory locations are being overwritten. • Determine if you can write data to arbitrary memory locations.
Result	• Confirmed or ruled out possible write-what-where vulnerabilities

The following article provides an example of exploiting a write-what-where vulnerability.

PLAY 23: Verify Use-After-Free Vulnerabilities

A use-after-free (UAF) is a class of vulnerabilities that occurs when a program attempts to access memory after it has been freed. A common occurrence of this is dereferencing a pointer that points to a freed chunk in the heap. The consequences of a UAF vulnerability can result in a program crash, printing unexpected data, or even arbitrary code execution.

Prerequisites	• Reverse Engineering Tool (e.g., Ghidra) • Access to source code or decompiled code, if possible
Process	• Investigate memory allocations • Investigate freeing memory • Observe if any dangling pointers exist • Observe if pointers or data structures are used after being freed.
Result	• Confirmed or ruled out possible UAF vulnerabilities

The following article provides examples of UAF vulnerabilities and how to exploit them.

PLAY 24: Verify Logic Error Vulnerabilities

Logic bugs are errors in a program's logic that cause it to operate incorrectly, but not to terminate abnormally. They can produce unintended or undesired output or other behavior, although it may not immediately be recognized as such. Logic errors can enable attackers to gain unauthorized access to systems, steal sensitive data, or take control of devices.

Prerequisites	• Reverse Engineering Tool (e.g., Ghidra) • Access to source code or decompiled code, if possible
Process	• Analyze the code's logic, flow, and how it handles user interactions. • Verify that authentication and authorization checks are correctly implemented and consistently applied throughout the code.Look for where access controls can be bypassed or incorrectly enforced. • Check how user inputs are validated and sanitized to prevent injection attacks. Examine how the code handles untrusted data. • Analyze how the code validates and parses data from external sources, such as files or network requests. Look for improper data type conversions or assumptions. • Examine how the application manages state transitions and handles concurrency. Look for potential race conditions or inconsistencies in the application's state.
Result	• Confirmed or ruled out the possibility of a logic bug vulnerability.

The following article provides examples of logic bugs and their resulting consequences.

PLAY 25: Verify Format String Vulnerabilities

A format string vulnerability is a type of security vulnerability that occurs in software when a program uses user-supplied input as the format string for a function that performs formatted output, such as printf() or sprintf(), without proper validation or sanitization. This can lead to unintended and potentially dangerous behavior, including code injection, information disclosure, memory corruption, or even remote code execution.

Prerequisites	• Reverse Engineering Tool (e.g., Ghidra) • Access to source code or decompiled code, if possible
Process	• Search for the use of insecure formatting functions (e.g., sprintf) • Identify user-controlled format strings (e.g., printf(user_input)) • Look for improper validation or sanitization • Experiment with including format strings in user input (%s, %x, %n) • Populate buffers with the maximum length of data to prevent null characters from being appended.
Result	• Confirmed or ruled out the possibility of format string vulnerabilities.

The following article illustrates exploiting format string vulnerabilities in an Android app.

PLAY 26: Verify Information Leak Vulnerabilities

An information leak or information disclosure is the unintended or unauthorized exposure of data to individuals, entities, or processes that should not have access to it. Such data may include arbitrary, sensitive, or confidential information. Information leaks can occur due to various factors, including vulnerabilities in software or misconfigurations. Information leaks can lead to bypassing exploit mitigations, privacy violations, identity theft, or financial loss.

Prerequisites	• Reverse Engineering Tool (e.g., Ghidra) • Access to source code or decompiled code, if possible
Process	• Identify format string vulnerabilities • Identify unencrypted data transmitted over the network, stored on a server, or printed out. • Identify weak access controls, such as weak passwords or broad permissions. • Explore SQL injection attacks to execute arbitrary SQL queries.
Result	• Confirmed or ruled out the possibility of information leaks.

The following article illustrates exploiting information leaks in an Android app.

PLAY 27: Identify Vulnerabilities in Shared Libraries

Android APK files use system libraries for system-level functionality and third-party libraries to integrate specific functionalities into their apps. These shared libraries, however, can impact the security posture of Android applications in many ways, including dependency on library security, data leakage, improper access controls, malicious dynamic loading, and more. Thus, shared libraries can expand the attack surface of APKs, especially if best practices are not followed.

Prerequisites	• Reverse Engineering Tool (e.g., Ghidra) • Review the following Ghidra Reversing Play • Access to the library file • Access to the library's source code or decompiled code, if possible • OSINT Framework (Exploits & Advisories)
Process	• Research previously known vulnerabilities of the library. • Confirm if those vulnerabilities have been properly mitigated • Reverse engineer the library for vulnerabilities previously discussed. • Look for banned functions. • Map vulnerable library calls to APK Activities defined in its Java code
Result	• Confirmed or ruled out possible vulnerabilities in shared libraries.

The following article describes abusing dynamic code loading in the Google Play Core Library.

PLAY 28: Verify Input Validation Vulnerabilities

Input validation vulnerabilities occur when a program does not properly validate user input, allowing an attacker to enter malicious data that can cause the program to behave in an unexpected way. This can include data that is too long, contains invalid characters, or is otherwise unexpected. Lack of input validation can lead to buffer overflows, format string exploitation, SQL injection, Cross-Site Scripting (XSS), and more. Improper input handling is one of the most common weaknesses identified across applications and a leading cause behind critical vulnerabilities that exist in systems and applications.

Prerequisites	• Reverse Engineering Tool (e.g., Ghidra) • Access to source code or decompiled code, if possible
Process	• Analyze functions that contain implementations for ingesting user data. Confirm if the implementations are insufficient.
Result	• Confirmed or ruled out poor or lack of input validation

The following article demonstrates improper input handling resulting in a buffer overflow.

PLAY 29: Verify any Web Server-based Vulnerabilities

When Android applications use a web server or incorporate web server functionality, it inherently expands its attack surface. If they are not designed, developed, or configured properly, attackers can compromise the server, steal data, or disrupt services. These vulnerabilities can exist in various components of a web server stack, including the web server software itself, server-side applications, and associated plugins or modules. A wide range of web server vulnerabilities exist, including Cross-Site Scripting (XSS), SQL Injection (SQLi), Broken Access Controls, Cryptographic failures, Directory/Path Traversal, Server-Side Request Forgery (SSRF), and more. This Play discusses common attacks that can be explored to confirm such vulnerabilities.

Prerequisites	• Complete the prerequisites of this Play • Any collected Enumeration results from this Play
Process	• Verify if any of the Enumeration results reveal any vulnerabilities • Attempt to perform a Directory Traversal attack • Attempt to perform an Authentication Bypass • Attempt to perform an Indirect Object Reference (IDOR) • Attempt to perform Server-Side Request Forgery (SSRF) • Attempt to perform a Local or Remote File Inclusion • Attempt to perform a File Upload Bypass • Attempt to perform SQL Injection • Payloads that may be useful include XSS, SSTI, SQL injection.
Result	• Confirmed or ruled out the possibility of web server vulnerabilities

Fuzzing

Fuzzing is an automated software testing method that identifies possible software defects, weaknesses, and vulnerabilities by sending invalid, malformed, or unexpected inputs to a target system. The fuzzer then monitors for exceptions such as crashes, overflows, or information leaks. When a crash does occur, a crash file is generated detailing the inputs that caused the exception for reproducibility and further investigation. There are different types of fuzzing techniques, such as dumb fuzzing, smart fuzzing, mutation fuzzing, generation fuzzing, in-memory fuzzing, snapshot fuzzing, and more. This section details certain fuzzing tools that can assist your Android security research.

PLAY 30: Run a Mutational Fuzzer

American Fuzzy Lop (AFL) is a brute-force fuzzer that employs a combination of innovative techniques, including instrumentation and genetic algorithms, to guide the fuzzing process towards exploring different code paths. AFL is also a mutation-based, coverage-guided fuzzer. AFL++ is an extended and community-driven fork of the original AFL project. AFL++ includes various improvements, bug fixes, and additional features to provide an enhanced and extended fuzzing experience for security researchers. Users can choose between AFL and AFL++ based on their specific requirements and the features offered by each tool.

Prerequisites	• Download AFL or AFL++ • Install AFL/AFL++ software dependencies • Install QEMU (Optional)
Process	• Instrument programs for use with AFL/AFL++ (see their instructions) • Create a corpus directory that holds the initial “seed” sample inputs • Run the AFL/AFL++ fuzzer. For example: Unset `$ ./afl-fuzz -i testcase_dir -o findings_dir -- /path/to/program @@` • Monitor fuzzer’s performance. • Analyze crash results, if any. • Tweak the fuzzer as necessary to improve its performance.
Results	• AFL/AFL++ produces a crash file detailing a legitimate vulnerability.

When source code is not available, AFL/AFL++ provides QEMU support for emulation. To utilize this feature, run their *build_qemu_support.sh* script and make sure to export the necessary

QEMU_LD_PREFIX and QEMU_SET_ENV environment variables. If you're interested in using AFL, here is a detailed walkthrough explaining how to use the tool. If you want to further explore AFL++, check out its extensive collection of tutorials.

PLAY 31: Run a Coverage-Guided Fuzzer with Sanitizers

LibFuzzer is an in-process, coverage-guided, evolutionary fuzzing engine. It is linked with the library under test, and feeds fuzzed inputs to the library via a specific fuzzing entrypoint. The fuzzer then tracks which areas of the code are reached, and generates mutations on the corpus of input data in order to maximize the code coverage. LibFuzzer helps security researchers find software vulnerabilities by automatically generating and executing a large number of test cases with the goal of exploring different code paths and identifying potential security issues. Libfuzzer can also be coupled with sanitizers to find specific vulnerabilities.

Prerequisites	• Install Clang compiler • Install QEMU (optional)
Process	• Select your target function or library that you want to fuzz. • Compile the vulnerable library program using clang and the flag options *–fsanitize=fuzzer,address.* • Develop your fuzzing harness using LLVMFuzzerTestOneInput(). • Compile your fuzzing harness with clang and link it to the object file of the vulnerable program. • (Optional) If using QEMU for cross-architecture support, export the environment variables QEMU_LD_PREFIX and QEMU_SET_ENV. • (Optional) Create a corpus directory that holds the initial "seed" sample inputs • Execute your fuzzer directly or by using QEMU • Monitor fuzzer's performance. • Analyze crash results, if any. • Tweak the fuzzer as necessary to improve its performance.
Results	• Libfuzzer produces a crash file detailing a legitimate vulnerability.

If you're interested in using LibFuzzer, here is a detailed tutorial explaining how to use the tool.

Bypass Exploit Mitigations and Security Protections

Exploit mitigations are measures and techniques implemented to reduce the risk and impact of software vulnerabilities being exploited by malicious actors. These mitigations aim to make it more challenging for attackers to successfully leverage vulnerabilities for unauthorized access or manipulation of a system. Binary exploit mitigations are often implemented by the compiler and/or the linker in cooperation with the operating system. The presence of exploit mitigations in a binary file can be statically checked without running the executable. Exploit mitigations for a system may also be enforced via security policies or access controls. Additional security protections may also be integrated to prevent reverse engineering, low-level analyses, or the use of virtual environments. This section describes a variety of exploit mitigations and security protections that may need to be bypassed for successful target exploitation.

PLAY 32: Bypass Stack Canaries

Stack canaries, also known as stack cookies, are values placed on the stack to detect buffer overflow attacks. If a buffer overflow occurs and overwrites the canary value, it serves as an indicator of a potential exploit, triggering an exception.

Prerequisites	• Location of the canary on the stack
Process	There are two popular methods for bypassing stack canaries: 1. Leak the stack canary value 2. Bruteforce the stack canary value Once obtained, the stack canary value can be incorporated into your exploit to prevent an exception from occurring.
Results	• Reliably obtain stack canary value for exploitation.

The following article provides an example of successfully bypassing a stack canary.

PLAY 33: Bypass Execute Never

The Execute Never (XN) exploit mitigation marks certain areas of memory as non-executable, preventing the execution of code in those memory regions. This mitigates the risk of buffer overflow and similar attacks that involve injecting and executing malicious code in areas intended for data storage. This mitigation is bypassed by Return-oriented programming (ROP).

Prerequisites	• Ability to inject code • Ability to control code execution (e.g., PC, LR registers) • Install Ropper
Process	• Identify available ROP gadgets in LIBC using Ropper • Build ROP chain • Overwrite the return address with the address of the first ROP gadget • Groom the stack as necessary for the entire ROP chain. This includes any arguments for function calls made. • The final ROP gadget executes the desired payload or system call
Results	• Can successfully return to and execute the desired code.

The following article provides an example of using ROP to bypass the XN mitigation on ARM.

PLAY 34: Bypass Address Space Layout Randomization

Address Space Layout Randomization (ASLR) randomizes the memory addresses used by a process, making it difficult for attackers to predict the location of specific functions or data. Randomized elements include the stack, heap, libraries, or the base address of an executable if compiled with position independence. ASLR, however, can be bypassed using information leaks. An example of bypassing ASLR against an Android app can be found here.

Prerequisites	• Information leak • GDB, GDB Server, or other debugging capabilities
Process	• Use the information leak to collect information about the memory layout, such as the address of a LIBC function call (e.g., exit). • Calculate the LIBC offset for the function pointer (e.g., exit) • Subtract the offset from the function pointer to get the base address of LIBC • Identify the offset of another of LIBC function call (e.g., system) • In your exploit, add this offset to the LIBC base address to automatically reference the LIBC function call (e.g., system) • Craft the rest of your exploit accordingly to achieve desired effect(s)
Results	• Successful exploitation of the ASLR-enabled target.

PLAY 35: Bypass Anti-VM Mechanisms

Anti-VM (Virtual Machine) mechanisms are techniques employed by software to detect the presence of a virtualized environment, such as a virtual machine or sandbox. Anti-VM mechanisms are often implemented to hinder analysis or reverse engineering.

Prerequisites	• List of Anti-VM techniques determined by APKiD (see this play).
Process	Below are different ways to bypass Anti-VM checks: • Most BUILD checks can potentially be mitigated by modifying your device's build.prop file • Checks that look for "sim_operator", "device_id", or "line1_number" all relate to Android API references under the TelephonyManager class. • You can attempt to hook API calls like `getSimOperator()` and `getDeviceId()` by running Objection on the live application, or by utilizing Frida scripts in a similar fashion.
Results	• Can successfully run the target app in a virtual environment

The following article provides an example of how to bypass Anti-VM checks for an Android app.

PLAY 36: Bypass Anti-Debugging Mechanisms

Anti-debugging mechanisms are techniques employed in software to detect and hinder the process of debugging or reverse engineering. This can prevent security researchers from examining and understanding the runtime behavior of a program.

Prerequisites	• List of Anti-Debugging techniques determined by APKiD in this play.
Process	• You can attempt to hook functions like `detectDebugger()` or `isDebuggerConnected()` by running Objection on the live application, or by utilizing Frida scripts in a similar fashion.
Results	• Can successfully debug the target app

The following resource provides more information on how to bypass anti-debugging mechanisms on Android with additional examples and solutions.

PLAY 37: Bypass Anti-RE Mechanisms

Anti-RE (Reverse Engineering) mechanisms are techniques employed in software to impede or deter the process of reverse engineering. Examples of Anti-RE methods include obfuscation, encryption, packing, or polymorphism. This Play will focus on performing code deobfuscation.

Prerequisites	• File editing software (An IDE or code editor like VS Code) • APKiD output (see this play) • Ability to communicate with the device via ADB (see this play) • Ability to perform static analysis (e.g., JADX, Ghidra) • Access to a debugger (e.g., GDB, GDB Server) • Software that can hook functions running inside of a live Android app (e.g., Objection, RMS, or Frida)
Process	• Use the obfuscation information provided by APKiD to guide your deobfuscation efforts • Open the APK in JADX • Observe any possible obfuscation, such as no strings, scrambled strings, and obfuscated JNI function calls • Identify possible deobfuscating function for the codebase • Use a debugger to examine the runtime behavior of the deobfuscation process • Use automated tools and scripts like Frida that are designed for deobfuscation • Perform manual refactoring where necessary • Continue reverse engineering until the app is properly deobfuscated
Results	• Deobfuscated APK

The following resources provide additional information and examples for deobfuscating and unpacking Android applications:

- Android Deobfuscation – Part I
- Android Deobfuscation – Part II
- N Ways to Unpack Android Apps

PLAY 38: Bypass Root Detection

Root detection in mobile applications is a security mechanism that identifies whether a mobile device has undergone the process of "rooting" on Android devices or "jailbreaking" on iOS devices. Rooting is the practice of removing software restrictions imposed by the device manufacturer or operating system (OS) to gain privileged control. This detection method determines if the user is running the app on a rooted or jailbroken device.

Prerequisites	• Access to Frida • Access to JADX • Access to APKTool
Process	Method #1: • Run automated tools and scripts like Frida that are designed for root detection bypass Method #2: • Analyze the APK in JADX and identify the root detection function. • Then decompress the APK using APKTool and modify the SMALI code corresponding to the root detection function accordingly.
Results	• Successfully bypassed root detection mechanisms

The following article walks through different examples for bypassing root detections on Android.

PLAY 39: Bypass SELinux

Security-Enhanced Linux, or SELinux, is a set of kernel modifications and user-space tools designed to add mandatory access controls (MAC) to the Linux operating system. It was originally developed by the National Security Agency (NSA) in collaboration with Red Hat and other open-source contributors. SELinux provides a flexible and fine-grained security framework that goes beyond the traditional discretionary access controls (DAC) commonly used in Unix-like operating systems.

Prerequisites	• GDB, GDB Server (optional)
Process	• Attach GDB to the target process • Use the command *getenforce* to get the current SELinux mode • To switch to Permissive mode, run *setenforce 0* • To switch back to Enforcing mode, run *setenforce 1*
Results	• Target app running in Permissive mode

To learn more about SELinux internals and policies, check out the following article.

PLAY 40: Bypass Code Signing

Code signing for mobile applications is a security practice that involves digitally signing the executable code and other relevant files of a mobile application with a cryptographic signature. This signature provides a way to verify the authenticity and integrity of the app's code before it is installed and executed on a mobile device. During the security research process, a target application may need to be modified and rebuilt; therefore, it is important to have the ability to perform code signing and signature verification or bypass these checks entirely.

Prerequisites	• Access to *keytool* and *jarsigner* (included with JDK) • Modified APK

Process	• Open a command prompt and use keytool to generate a keystore. Replace ***your_key_alias*** with a unique alias for your key and ***your_keystore_name.keystore*** for the name for your keystore file.
	Unset
	`$ keytool -genkeypair -v -keystore your_keystore_name.keystore -keyalg RSA -keysize 2048 -validity 10000 -alias your_key_alias`
	• Use jarsigner to sign the APK. This command will prompt you to enter the keystore password.
	Unset
	`$ jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore your_keystore_name.keystore your_app.apk your_key_alias`
	• Use jarsigner to verify the signature of the APK. If the APK is signed correctly, you will see a successful verification output.
	Unset
	`$ jarsigner -verify -verbose -certs your_app.apk`
	• Note, if you are using Android Studio or another build tool, you may have built-in options to sign your APK during the build process.

Results	• Successful signing and execution of modified APK

Exploitation

Exploitation is the process of creating an exploit that takes advantage of a security vulnerability to perform an unexpected action which results in a cyber effect. It confirms that the discovered vulnerability is exploitable while also demonstrating the severity of the identified vulnerability. This section describes different types of cyber effects that can be achieved during exploitation.

PLAY 41: Explore Local Privilege Escalation Attacks

Local Privilege Escalation (LPE) is the exploitation of vulnerabilities in a system or software to gain higher levels of access or privileges than originally intended by the system's security policy. The term "local" indicates that this type of privilege escalation occurs when an attacker already has some level of access to the system with limited privileges, typically as a regular user. Upon successful exploitation, they may escalate from a regular user to an administrator or root user. With elevated privileges, the attacker may be able to execute arbitrary code, install malicious software, manipulate system configurations, or perform other actions that were not permitted with their initial level of access.

Prerequisites	● Identified vulnerability or misconfiguration that can be exploited. ● Ability to bypass security mechanisms, if necessary.
Process	Since there are different ways to obtain an LPE effect, below are common techniques and scenarios to consider for your target: ● DLL Hijacking ● Unpatched system/software ● Abusing named pipes ● Insecure configurations ● Kernel vulnerabilities ● Driver vulnerabilities ● Manipulating services running with higher privileges ● Insecure file permissions ● User input validation vulnerabilities ● Weak authentication mechanisms ● Weak, plaintext, or hard-coded passwords ● Use of Legacy or End-of-Life software ● Other software vulnerabilities
Results	● User with escalated privileges

For more information on exploiting an Android vulnerability that can lead to an LPE, check out this article. In addition, ARM provides documentation regarding Exception Levels that can also be escalated for consideration.

PLAY 42: Explore Remote Code Execution Attacks

Remote Code Execution (RCE) allows an attacker to execute arbitrary code or commands on a target system from a location external to the target system. The code could be designed to perform various malicious actions, such as taking control of the system, stealing sensitive information, or disrupting normal operations. This can occur over the internet, across a network, or through some other remote communication channel.

Prerequisites	• Ability to perform code injection • Ability to control or redirect program execution • Ability to bypass exploit mitigations, if necessary • Ability to execute shellcode • Shellcode for a reverse shell (or desired effect)
Process	• Start a listener (e.g., `netcat`) on the attacker machine, if applicable. • Send malicious input or data to exploit the target • Observe an established reverse shell connection or desired effect.
Results	• An established reverse shell connection or desired effect executed

The following article provides an example of achieving an RCE effect against an Android app.

PLAY 43: Explore Pin Lock Defeats

A "PIN lock defeat" is the unauthorized bypass or circumvention of the Personal Identification Number (PIN) lock on a mobile device. The PIN lock is a security feature commonly used on mobile devices to restrict access and protect the device's data from unauthorized users. Mobile applications may also use a PIN to restrict access to the app's data or features. Biometric authentication (such as fingerprint or facial recognition) may be used in addition to or instead of a PIN, requiring spoofing or bypassing the biometric authentication mechanism to gain access.

Prerequisites	• Access to the target device or application
Process	Since there are different ways to obtain to defeat a PIN lock, below are common techniques and scenarios to consider for your target: • Brute Force attack • PIN implementation weaknesses • Weak, plaintext, or hard-coded PINs • Directly invoking Activities that don't require a PIN
Results	• Access to the device or mobile application without the need for a PIN

The following article provides an example of bypassing the lock screen on an Android device.

PLAY 44: Explore Zero-Click Exploitation

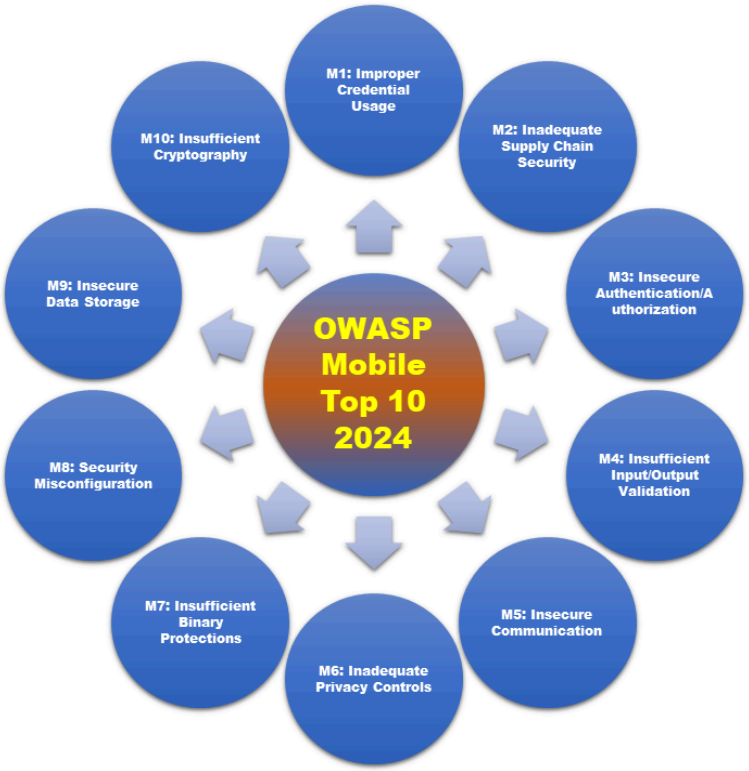
Zero-click exploitation is a type of cyberattack in which an attacker can compromise a target system without any user interaction or involvement. Unlike traditional exploits that may require the victim to click on a malicious link or open a malicious attachment, zero-click exploits do not rely on user actions. Zero-click exploits can be delivered in different ways and often result in remote code execution on the targeted device. Since there is no user interaction, the victim may not be aware that their device has been exploited and may go unnoticed for an extended period.

Prerequisites	• Identified attack vector and delivery mechanism • Ability to perform code injection • Ability to control or redirect program execution • Ability to bypass exploit mitigations, if necessary • Ability to execute shellcode • Shellcode for a reverse shell (or desired effect) • One or more exploitable vulnerabilities
Process	• Start a listener (e.g., `netcat`) on the attacker machine, if applicable. • Send malicious input or data to exploit the target • Observe an established reverse shell connection or desired effect.
Results	• Successful exploitation without user interaction

The following writeup walks through developing a zero-click exploit against an Android app.

Appendix A: OWASP Mobile Top 10

The OWASP Mobile Top 10 is a list of the ten most critical security risks facing mobile applications. OWASP, which stands for the Open Web Application Security Project, is a non-profit organization that focuses on improving the security of software. The list is updated periodically to reflect the evolving landscape of mobile application security threats. The OWASP Mobile Top 10 for 2024 includes the following risks.



Insecure credential management can occur when mobile apps use hardcoded credentials or when credentials are misused. Supply chain vulnerabilities may arise if mobile applications are developed by third-party developers or rely on third-party libraries. Insecure authorization can occur when an organization fails to authenticate an individual before executing a requested API endpoint from a mobile device. Insufficient validation and sanitization of data from external sources can introduce severe security vulnerabilities. Insecure communication can allow attackers to possibly intercept and modify data if it is transmitted in plaintext or using a deprecated encryption protocol. Insecure privacy controls can lead to the disclosure of personally identifiable information (PII). Insufficient binary protections make it easier for attackers to reverse engineer or exploit a mobile app. Improper security configurations, permissions, and controls can lead to vulnerabilities and unauthorized access. Insecure data storage can result in potential privacy breaches and unauthorized access to sensitive data. Insecure cryptography can undermine the confidentiality, integrity, and authenticity of sensitive information. Keep in mind the OWASP Mobile Top 10 risks throughout the Android security research process as they will likely be encountered while following and executing the ASRP.