

# **Android Reverse Engineering**

**Nicolas Janis (NDJSec)**

**March 14, 2024**

### 0.1 Preface

This course, **Android Reverse Engineering**, is meant to be a free comprehensive guide to Android Reverse Engineering (RE). I am writing this guide to help bring new users into the space, as well as share what I have learned while learning reverse engineering. At the time of this writing, March 2024, I have been learning reverse engineering for almost 2 years and recently began Android RE. During this journey, my good friend Kevin Thomas (@mytechnotalnet) recommended that I share my skills.

Before I start, I would like to thank a few of the people who helped me organize and write this information. First is Kevin, for encouraging me and giving me the idea to create this guide. Secondly, I would like to thank my research advisor Dr. Adwait Nadkarni for his mentorship and providing me the resources to learn these skills and bring them to you. Thirdly, I would like to thank my parents for all their love and support while writing this series. They have done so much not only helping me to edit this guide but also helping to create the opportunity for me to learn what I am presenting in this guide. I would also like to thank Taylor Perry for her help in proofreading this guide and for being my moral support in this journey. And finally, I would especially like to thank all my friends and coworkers who introduced me to reverse engineering and helped me learn the skill that I always wanted to learn but was too afraid to.

With that said, let's get to the fun stuff. Join me on this journey while we learn a new skill and solve the ever growing challenges of this amazing field together. This is just the starting point to train the next generation of Cyber Security Professionals.

0.1 Preface .....	2
<b>Chapter 1 Introduction to Reverse Engineering</b>	<b>5</b>
1.1 Goals .....	5
1.2 Techniques.....	6
1.3 Tools: Overview .....	6
1.4 Tools: Linux.....	7
1.5 Tools: VIM.....	12
1.6 Tools: APKTool .....	14
1.7 Tools: Jadx.....	15
1.8 Tools: Android Debug Bridge .....	16
1.9 Tools: Ghidra .....	17
<b>Chapter 2 Android Basics</b>	<b>19</b>
2.1 Android Overview: Why should we care? .....	19
2.2 Component Model .....	20
2.3 Android Permissions .....	21
2.4 Android Package: APK.....	22
2.5 Manifest File.....	23



# Chapter 1 Introduction to Reverse Engineering

## 1.1 Goals

Before we can begin to discuss the intriguing and interesting world that is Reverse Engineering, we must first outline what is Reverse Engineering, also known as RE.

Reverse Engineering, as defined by the Merriam-Webster dictionary is:

### Definition 1.1

"To disassemble and examine or analyze in detail (a product or device) to discover the concepts involved in manufacture usually to produce something similar"

But what does this mean to us? In short, reverse engineering is applying the scientific process to a piece of code, in such a way that we can take an application and begin to try and understand what it does and how it does it. For us, that means beginning to understand Android and its inner workings. From there, we will begin to look into Java and its underlying Java Virtual Machine (JVM). Then we will explore how Android changes this through their use of the Dalvik Virtual Machine, Dalvik bytecode, and the more recent Android Runtime (ART). Lastly, we will begin our descent into the deep world of Native code (C, C++, Assembly) through the use of the Java Native Interface (JNI). This is where the worlds of Native code reverse engineering and Java reverse engineering begin to collide.

At this point, you might be asking yourself one of two questions, "Cool! Where do we start?", or "Why should I care and why should I learn RE?". To the first question I say don't fret, we will begin in due time. To those of you who are still skeptical and might think "why should I care", let's talk about all the great things that reverse engineering has been used for and all the great things that it can still do. Hopefully, by the end of this section I will have convinced you to join the tip of the spear that is security research and reverse engineering.

The first famous example of reverse engineering comes from Phoenix Technologies. In the Mid-1980s, Phoenix implored reverse engineering to create a compatible BIOS (Basic Input Output System) with the IBM BIOS that existed at the time. In their efforts, they were able to create a system that allowed more users to interact with the software of the tech giants of the time IBM. Because of this, you are now able to enjoy various choices when building your systems without having to worry about not being able to use software. If that's not enough, does anyone remember WannaCry or other large malware attacks? These were stopped by reverse engineers.

The truth is that many of you are performing reverse engineering already and don't even know it. Every time you enter a new code base and start poking around to understand the code so you can build your next great invention, you too are performing reverse engineering.

Are you sold yet? I hope so. Join me in learning about the amazing world of reverse engineering and learn to reverse engineer on a system owned and used by millions around the world. Let's become the world's next generation of Cyber Professionals. The choice is yours, so let's embark on this journey together, are you ready?

## 1.2 Techniques

In the world of reverse engineering, there are two main techniques for analysis. These include Static Analysis and Dynamic Analysis. As the name implies, static analysis is the concept of statically looking at the code to perform analysis. In other words, you are analyzing the code without running it. You are simply reading through the code in order to identify important details and gain an understanding of the code. In comparison, dynamic analysis is the concept of looking at the code dynamically. In other words, you are performing your analysis through the act of running the code and analyzing the program's behavior at run-time.

In order to perform this analysis, we as reverse engineers use a variety of tools. In the case of dynamic analysis, we often use a disassembler and/or a decompiler to convert our program into some kind of code that we can understand. For this series, we will use 3 main tools for static analysis but know that there are many out there and it ultimately comes down to personal preference and comfort. The three static analysis tools that we will be using throughout the series, are *Apktool*, *Jadx*, and *Ghidra*. However, I will be sure to link to many other tools at the end of this series for those interested.

We will also use a multitude of different dynamic analysis tools throughout this series. Some of these tools include Frida, which is a dynamic instrumentation framework. Unicorn, which is a great emulation framework. And ADB, or known as the Android Debug Bridge, which is Android's developer interface for interacting with their phones directly for debugging purposes.

We will spend the early parts of this series focusing static analysis through the use of disassemblers and decompilers. Later on in the course, we will look into dynamic analysis.

A disassembler is a program that will allow a reverse engineer to extract the assembler instructions from the machine instructions or the bytecode of the program. This will allow us, as reverse engineers, to more easily understand what is going on. Disassemblers are often paired with decompilers, programs that can convert this assembler into a pseudo-representation of what the original code might have looked like. In our case this will be Java code, and later this will be C/C++.

In the next section, we will look at some of the tools that we will be using throughout this series, and walk through how to download some of these tools.

## 1.3 Tools: Overview

In this section, and the subsequent sections, we will look at and install a few of the tools that we will be using throughout the course of this series. The first tool we will look at is APKTool. APKTool is an Android reverse engineering tool designed for pulling apart Android apps, also known as Android Packages (APKs). Don't worry too much about APKs, as we will talk about them in greater detail later in this course. The second tool that we will look at is Jadx. Jadx is a Java decompilation tool that is mainly focused on Android APKs. While many Java decompilation tools exist out there, I mainly choose Jadx for its built-in support of Smali, Android's bytecode that we will discuss more later, extraction and decoding of the Android manifest file, as well as support for easily creating Frida scripts. Don't worry if you aren't sure what a lot of this means, as we will describe all of it in much greater detail later on in this course. I simply wanted to outline some of the reasons I choose Jadx, and why you might want to as well.

After Jadx, we will look at the Android Debug Bridge (ADB). This tool directly from Android, will allow us to easily interact with our Android phone, whether it be an emulator or a real phone. Lastly, we will

discuss Ghidra, the open-source native code reverse engineering tool released by the National Security Agency (NSA). This tool will be great for when we get into our native code analysis towards the end of this course. But, don't worry about that now, that is a long ways away and by then you will be well on your way to becoming a strong security professional.

However, before we begin to install any of these tools let's look at Linux. We will look at how to install it and talk about some of the reasons we might want to consider Linux as our operating system of choice when performing our analysis.

In the next section, we are going to go through how to install Linux in a virtual machine using VirtualBox and some of the reasons, and why you might want to consider using Linux for your analysis.

## 1.4 Tools: Linux

In this section, we will be looking at Linux and walking through a simple guide on how to install Linux on your computer using a virtual machine (VM). For this we are going to use VirtualBox, a free virtualization software for Window, Mac, and Linux.

Some of you might be wondering why we would want to use Linux, over say Windows or Mac. The short answer is because of some of the tools and scripts we will use later on in this series are written for UNIX-based operating systems and work best on Linux. However, it is worth noting that for a lot of what we will be doing MacOS works perfectly fine and you can choose to skip this section if you'd like. However, for Windows, I suggest you continue with this section using a VM. You may also use the Windows Subsystem for Linux (WSL). While I will not be going over how to install WSL, it should be able to handle many of the analysis tools we will use. With that being said, let us begin.

First, download the VirtualBox software. To do that you can go to the following link below and install your appropriate version.

[VirtualBox Download](#)

### Note

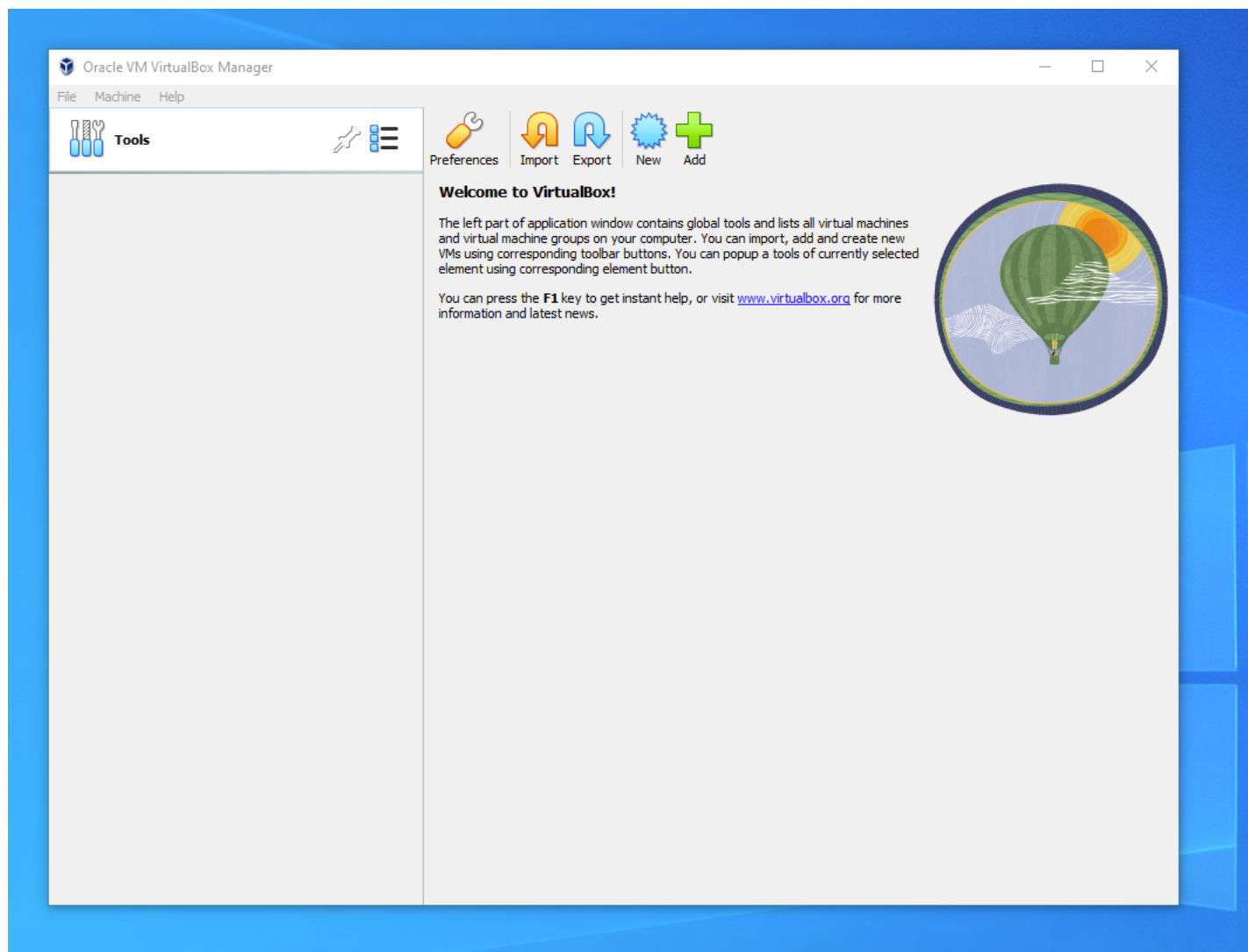
At the time of writing this, if you are on an M1 or M2 Mac, VirtualBox does not work entirely correctly. If you are on one of these platforms, feel free to skip this section as many of the tools here will work without issue on Mac.

You will also need to install a Linux ISO file. An ISO file is the file type used on operating systems installers. We will use this ISO file in order to install our Linux distribution of choice, in this case, Ubuntu.

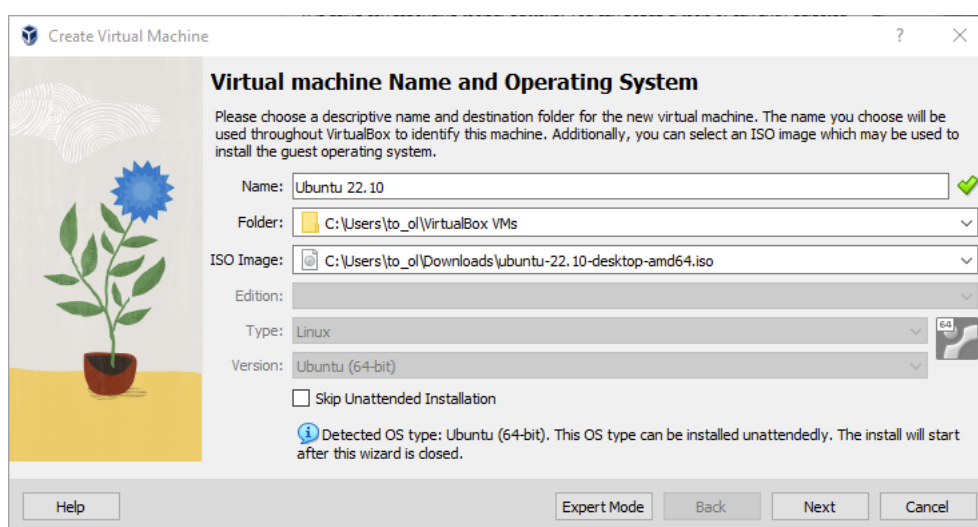
[Ubuntu Install Linux](#)

In this next part, we will be following Ubuntu's guide for installing an Ubuntu VM in VirtualBox. The link to their guide can be found [here](#).

Once, you have the installer downloaded, run the installer and follow the install instructions provided by the VirtualBox installer. All the defaults are fine for what we will be doing. Once you have VirtualBox installed, you should see a screen that looks like the following:



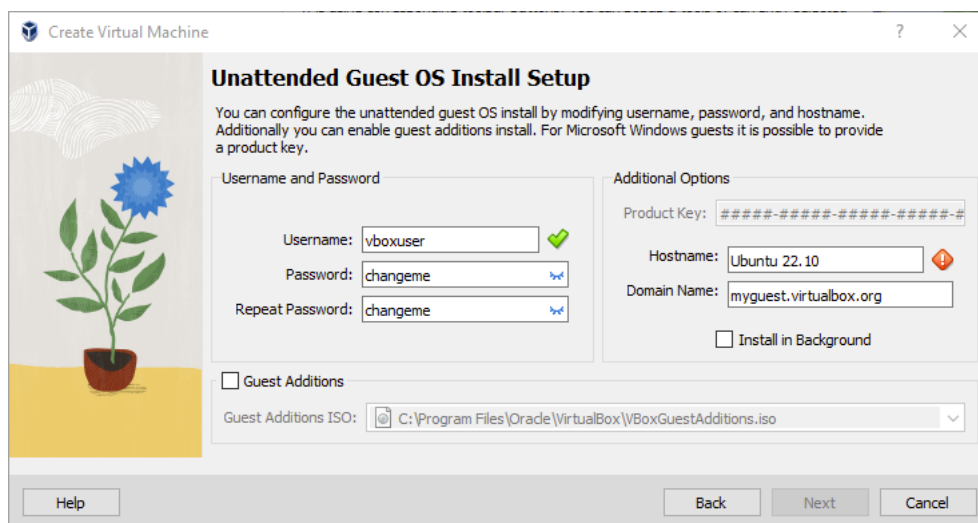
Next, we will create a new VM. To do this, we want to click **New** and fill in the appropriate details. For the name, you can choose whatever you like. For the machine folder, the default choice is fine. Then for the ISO image, select the Ubuntu ISO we downloaded earlier. Lastly, hit next.



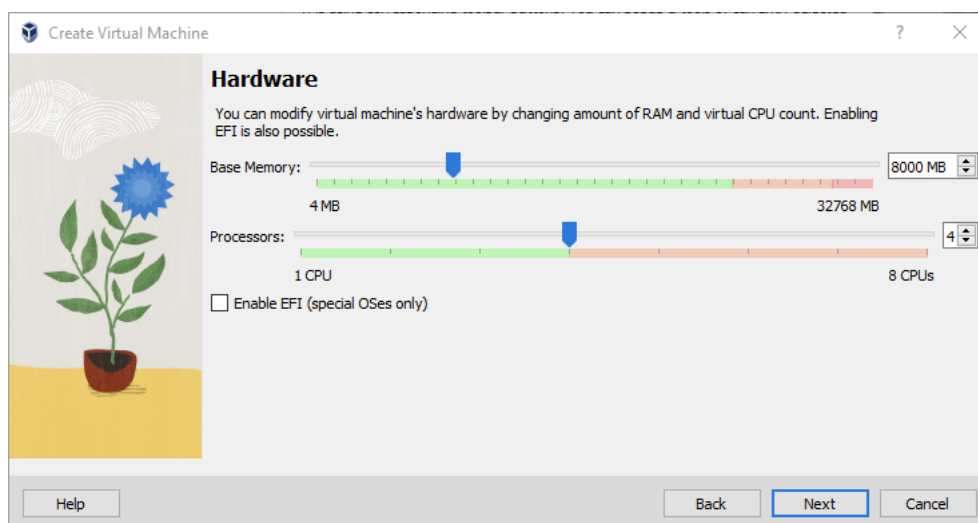
Next, we will create a user profile that we can use when using this machine. For this, enter in any values for your username and password. The default credentials are username: **vboxuser** and password: **changeme**. However, be sure to change these defaults as they will create a user without sudo access, which we will need over the course of this series. Lastly, enter a hostname of anything, ensuring



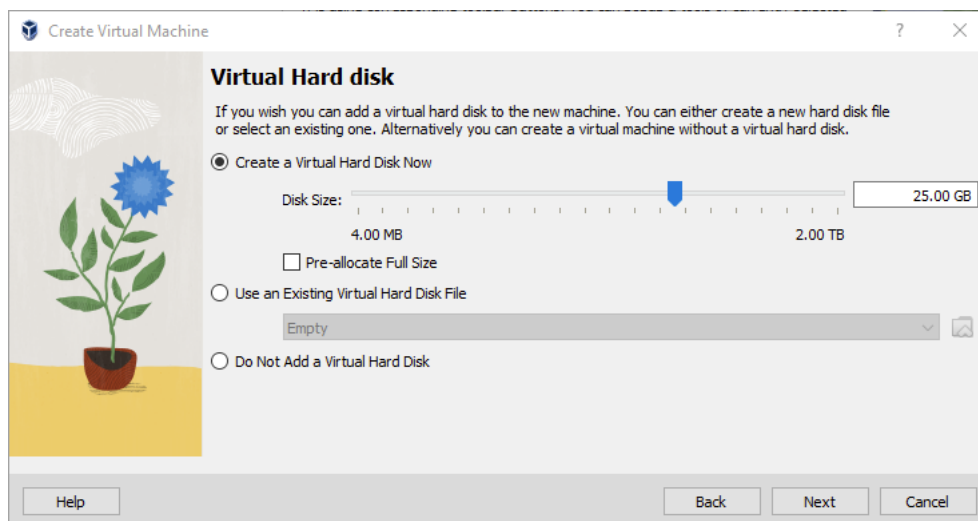
no spaces proceed it and you can leave the domain name blank. Also, select Install Guest Additions as this will allow us to copy and paste to and from the VM, along with other very useful functionality. Once done, click **Next** to continue.



Next, we must define the VM's resources such as memory and processors. For this, around 8GB of RAM is recommended by Ubuntu, and 4 CPUs are also recommended, however, this will depend on your system. A good note is to keep the sliders in the green as this will help prevent issues. Once done, click **Next** to continue.



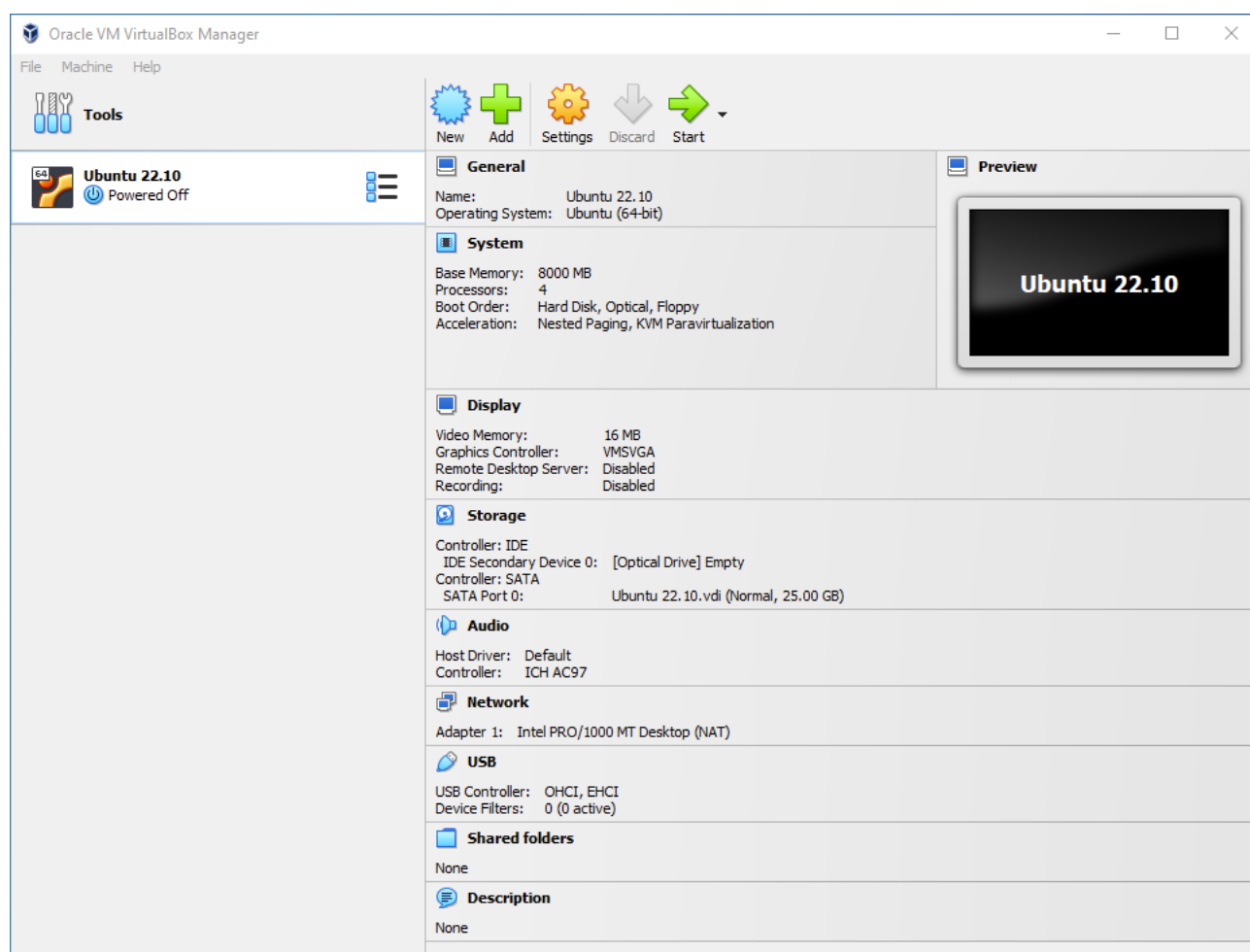
Lastly, we have to specify the amount of disk space, or storage that this VM will have. For this, I would not go under 25GB, but know that more is better. If you plan to have other tools, such as Ghidra and Frida in your VM, I would make this number larger. Remember, whatever number you use now, you can not easily resize later, so erring on the side of larger is better. Once done, click **Next** to continue.

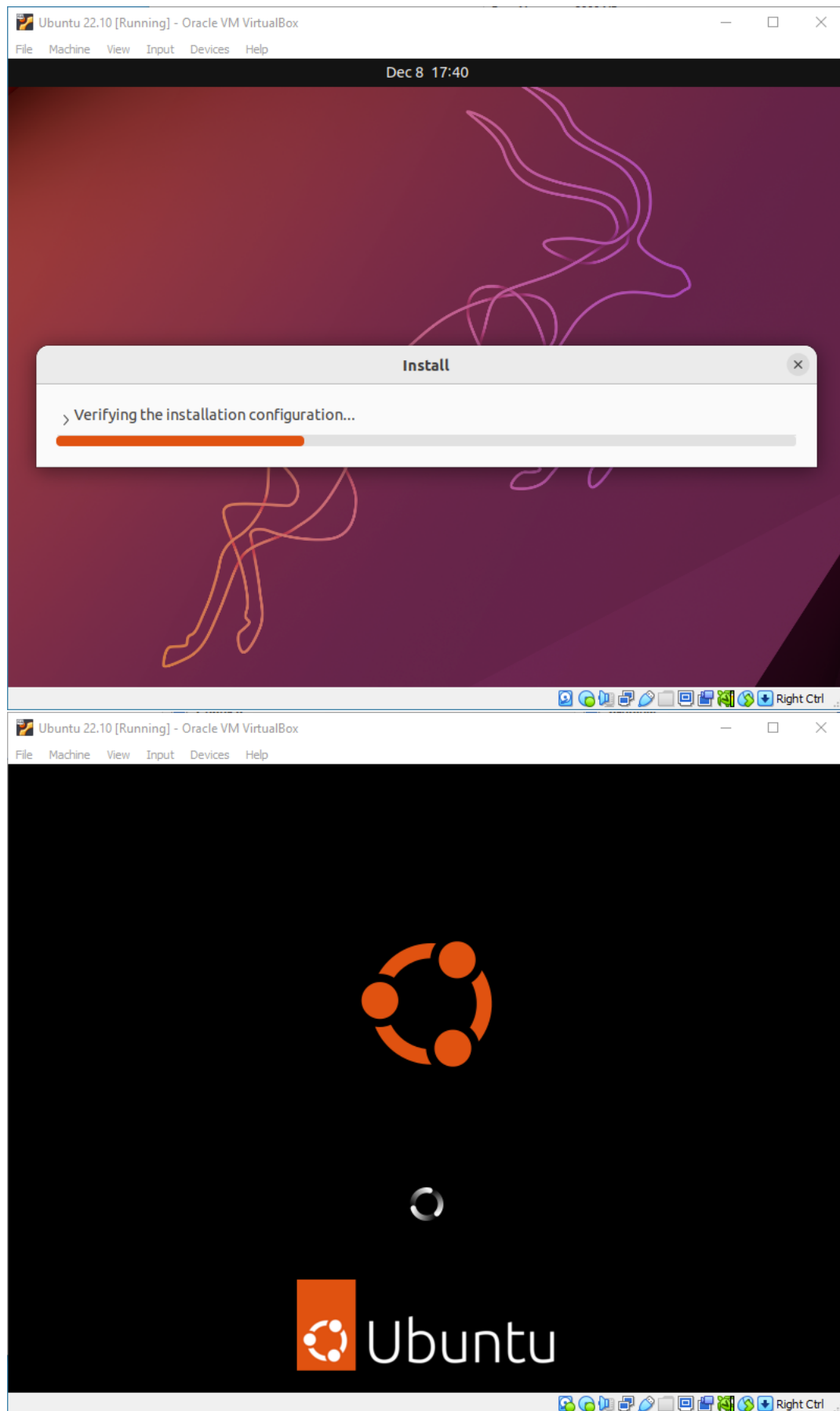


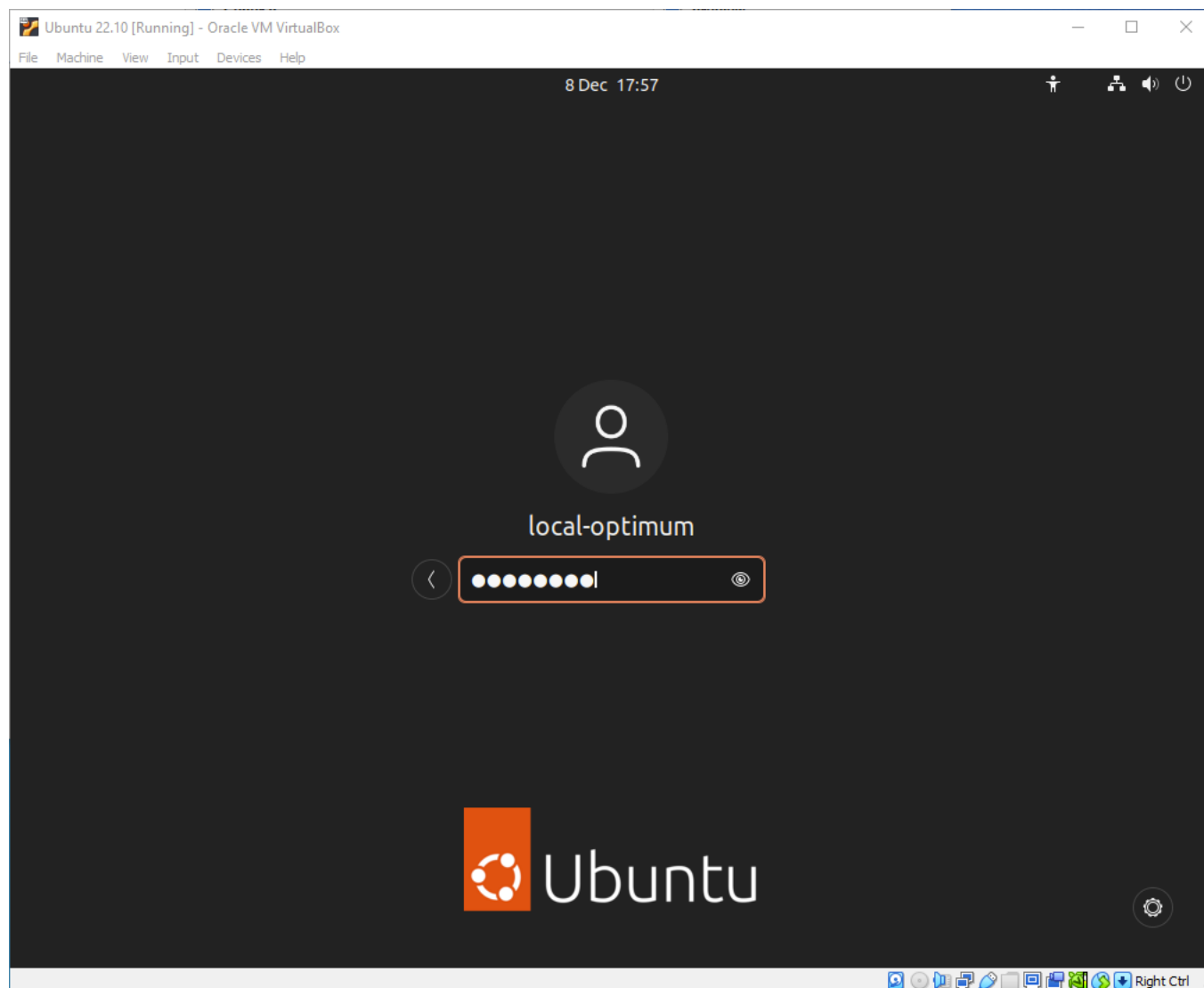
Finally, click **Finish** on the summary page to finish initializing your VM.

Congrats! You have just created your first Linux VM and are one step closer to becoming the next generation of security professionals.

Now, let's boot up the system and let the unattended installation do its thing. To do this click your newly created VM and click start. This will prompt you with a message saying "Powering VM up...". Don't touch the machine and just let it do its thing. When it is done, you will be greeted with a login screen where you can enter your username and password, which you set earlier in the installation. If everything went correctly, you will have officially set up your first Ubuntu Linux VM and are ready to start your RE journey.







In the next section, we will talk about installing VIM, a basic command line text editor in Linux.

## 1.5 Tools: VIM

Now that we have a working Linux setup, it's best that we install some kind of text editor. For this tutorial we will be using VI Improved, more commonly known as VIM.

VIM is a text editor that works in the command line. VIM is what is known as a modal editor. This means that you have different modes of operation and, depending on your mode, you can do different things. This may sound daunting, but we can do it together. And yes, we will be answering the age old question of "How do you quit VIM?"

At the end of this section, you will be able to not only quit VIM, but also use a text editor available to any and every UNIX-based system. This means that if you are on any UNIX computer, Linux, Android, Mac, etc., you will be able to work with and edit files. This is our true reason for learning to use VIM. Because who wants to download a file from a phone, modify it, and then push it back to the phone. If you know VIM, you don't have to.

And I know, I know, VIM may seem weird but it's the fastest way to edit text on any UNIX machine. And to paraphrase a friend of mine, "It beats the caveman way of doing it. Using the echo command to edit files."



So now that I've hopefully got you hooked on VIM, and you are ready to ask "how do we get it?". Well in some cases, you might already have it installed. If you open a terminal, and type `vim .`, if VIM is installed it will open and put you into NetRW, which will let you browse your files. However, if this command fails, let's install VIM so we can get started. To do this, use whatever package manager is on your operating system. For our Linux VM, the command will be **`sudo apt install vim -y`**. This command will install VIM for us so that we can get started.

Once VIM is installed, run the same command from earlier to ensure that VIM is installed correctly.

```
[No Name]
=====
" Netrw Directory Listing                               (netrw v171)
" /Users/nicjanis/Documents/Android-Reverse-Engineering
" Sorted by      name
" Sort sequence: [\/!$, \<core\%(\\d\\+\\)=\\>, \.h$, \.c$, \.cpp$, \~\\=\\*$, *, \.o$, \.obj$, \.info$, \.swp$, \.bak$, \~$
" Quick Help: <F1>:help  -:go up dir  D:delete  R:rename  s:sort-by  x:special
" =====
./
./
.git/
images/
Android_Reverse_Engineering_DRAFT.pdf
LICENSE
README.md
```

Now to answer the age old question of, "How do I exit VIM?"

To answer this question, first hit the **escape** key which will enter you into **Normal mode**. Once you are in normal mode, press the following keys in order, `:q`. Now you too know the answer to the mystical secret that is closing VIM.

So let's break down what that command means.

1. `:` - The colon enters us into command mode, where we can enter commands to execute.
2. `q` - This is the quit command, so we can quit the current VIM instance.

Now that you know how to close files with VIM, let's learn how to edit some files. To open a file, run the following command `vim ~/.vimrc`. This will create a new file called **.vim** in the `~` directory. Once this file is open, press the `i` key, this will enter you into insert mode. This will allow you to enter text. Type the following lines into your file:

### Example 1.1

```
set number
set smartindent
set tabstop=4
set shiftwidth=4
set expandtab
```

Then press **"escape"** to enter back into normal mode. Then press **:wq** to run the write command (**w**) and then the quit command (**q**). This will save and close your file.

Congrats! You have officially just created and edited your first file with VIM.

The file you just created will help format anything you write in order to make everything in the tutorial look more uniform.

Lastly, I will leave you with some of the most common commands you will use when using VIM.

#### Note

**j** or **down-arrow** [move cursor down one line]  
**k** or **up-arrow** [move cursor up one line]  
**h** or **left-arrow** [move cursor left one character]  
**l** or **right-arrow** [move cursor right one character]  
**O** [move cursor to the start of the current line]  
**\$** [move cursor to the end of the current line]  
**b** [move cursor back to the beginning of preceding word]  
**dd** [deletes the line the cursor is on]  
**D** [deletes from the cursor position to the end of the line]  
**yy** [copies the current line]  
**p** [puts the copied text after the cursor]  
**u** [undo the last change to the file]  
**:w** [save file]  
**:wq** [save file and exit text editor]  
**:q!** [quit text editor and do not save any changes]

With this you can now go read and modify text on any UNIX-based system no matter how big or small the system is. Plus, you can also say you know the answer to the question of, "How does one exit VIM?"

Join me in the next section where we continue our journey to set up our analysis environment by downloading tools. The next tool on our list will be APKTool, a small but powerful tool in any Android Reverse Engineer's toolkit.

## 1.6 Tools: APKTool

So we have a Linux environment and know how to use VIM, but what else do we need to be effective Android reverse engineers? To answer that question, I will first introduce you to APKTool. Created by iBotPeaches, APKTool is a lightweight but powerful Android reverse engineering tool designed for the command line. While APKTool is capable of many powerful things such as disassembling, assembling, and analyzing Android apps known as Android Packages or APKs for short, right now we will only install APKTool. Later on in this series, we will begin to dive into APKTool and all that it is capable of.

APKTool works on Linux, Windows, and Mac, but for this tutorial we will install it on Linux. However, installation instructions for all 3 OSes can be found [here](#).

The first thing that we have to install is the APKTool wrapper script, which can be found at the link above. We can install it using the following commands:

```
1 cd ~
2 mkdir Tools
3 cd Tools
```

```

4     mkdir APKTool
5     cd APKTool
6     wget https://raw.githubusercontent.com/iBotPeaches/Apktool/master/scripts/linux/
    apktool

```

These commands will create a directory in our home directory called **Tools**. This is something that I like to do when installing custom tools that I want access to system-wide.

The next thing that we need to do is install APKTool itself. For that, download the latest version [here](#). When downloading the file, make sure to place it in the newly created **Tools/APKTools/**. Once that is done, run the following commands to finish the installation.

```

1     mv apktool_2.9.3.jar apktool.jar
2     sudo chmod +x apktool.jar
3     sudo chmod +x apktool
4     sudo cp apktool.jar /usr/local/bin/
5     sudo cp apktool /usr/local/bin/
6     apktool

```

If everything was done correctly, you should see the following. If you do, congrats! You have officially installed APKTool and are one step closer to becoming an Android reverse engineer.

```

> apktool
Apktool 2.9.3 - a tool for reengineering Android apk files
with smali v3.0.3 and baksmali v3.0.3
Copyright 2010 Ryszard Wiśniewski <brut.all@gmail.com>
Copyright 2010 Connor Tumbleson <connor.tumbleson@gmail.com>

usage: apktool
    -advance,--advanced    Print advanced information.
    -version,--version     Print the version.
usage: apktool if|install-framework [options] <framework.apk>
    -p,--frame-path <dir>  Store framework files into <dir>.
    -t,--tag <tag>         Tag frameworks using <tag>.
usage: apktool d[ecode] [options] <file_apk>
    -f,--force             Force delete destination directory.
    -o,--output <dir>     The name of folder that gets written. (default: apk.out)
    -p,--frame-path <dir> Use framework files located in <dir>.
    -r,--no-res            Do not decode resources.
    -s,--no-src            Do not decode sources.
    -t,--frame-tag <tag>  Use framework files tagged by <tag>.
usage: apktool b[uild] [options] <app_path>
    -f,--force-all        Skip changes detection and build all files.
    -o,--output <dir>     The name of apk that gets written. (default: dist/name.apk)
    -p,--frame-path <dir> Use framework files located in <dir>.

For additional info, see: https://apktool.org
For smali/baksmali info, see: https://github.com/google/smali

```

Now that we have APKTool installed, we are one step closer to having a proper analysis environment for doing Android reverse engineering.

Join me in the next installment, where we will look at Jadx, a Java decompilation tool, and walk through how to install it.

## 1.7 Tools: Jadx

As some of you may or may not know a majority of Android code is written in Java or Kotlin, both of which run on top of the Java Virtual Machine (JVM). Don't worry if you do know what the JVM is, we will talk about that later. But to be good Android reverse engineers, we need a tool that can read Java code right? For that, I introduce you to Jadx, a Java decompilation tool that can do it all. Reading Java bytecode or Smali Android bytecode? Jadx has you covered. Need to read the Android manifest? Jadx has you covered. Want to easily create Frida scripts? Jadx can do that for you too.

Some of you might at this point be thinking "What does any of that mean?". And to that I say, don't worry about that right now. I promise by the end of this series you will see why all of that is amazing



and why we might want the functionality. For now, just understand that Jadx is a powerful tool that can help us endlessly. Also, did I mention that you can get all this for the low, low price of free?

That's right, you can get all this amazing functionality completely for free. And to make things better, Jadx is completely open-source. So why wait? Let's go install Jadx right now!

First go to the Jadx GitHub, which can be found [here](#), and download the latest version of Jadx. When downloading the file, place it in the **Tools** directory we created earlier. Once that has been done, run the following commands:

```
1 cd ~/Tools
2 mkdir Jadx
3 unzip jadx-1.4.7.zip -d Jadx
4 mkdir bin
5 cd bin
6 ln -s ~/Tools/Jadx/bin/jadx jadx
7 ln -s ~/Tools/Jadx/bin/jadx-gui jadx-gui
8 export PATH=$PATH:~/Tools/bin/
```

Once you run these commands, you will have installed Jadx. However, let's break down what those commands are doing so it doesn't just look like black magic. The first 5 lines unzip the jadx program and place them into a directory for storage. The real magic then happens in the last 3 lines. The first 2 are creating what is called a symbolic link—for the Windows people out there think of this as a shortcut. The last line is placing the directory that has these links in our **PATH** environment variable. This is the place where the OS goes to look when we run commands. By doing this, we can now run the **jadx-gui** or **jadx** commands from anywhere on our system and launch the program.

Congrats, you have officially installed Jadx and learned a bit about symbolic links. You truly are on your way to becoming a true security professional.

Join me next, as we dive into the Android Debug Bridge (ADB), looking at what it is and how to install it.

## 1.8 Tools: Android Debug Bridge

Our next, and arguably one of the most useful tools, is the Android Debug Bridge (ADB). ADB is a tool that comes directly from Android itself. Designed for developers, ADB gives us a simple interface for interacting with our Android phone. While the tool is very powerful, and has many capabilities, the 3 we are most interested in are the shell, push/pull, and logcat. The shell command will let us search the file system. The push/pull commands will let us download files to and from the device, and the logcat command will allow us to see debug logs from the phone easily. As you can guess a very powerful tool when working with Android applications.

So how do we get started? Let's find out together and download ADB for ourselves.

If you are downloading ADB, on the Linux VM we setup in a previous section this process is very easy. If you are using a different OS, the process is still rather straightforward. However, for this course, we will only be installing ADB on the Linux VM. To do this, simply run the following command "**sudo apt install adb**". This will use Ubuntu's built-in package manager to install ADB and set everything up for you. Once that is done, you should now have ADB installed. Run "**adb**" if you see the following output ADB installed with no issues.



```

> adb
Android Debug Bridge version 1.0.41
Version 35.0.0-11411520
Installed as /opt/homebrew/bin/adb
Running on Darwin 23.3.0 (arm64)

global options:
-a          listen on all network interfaces, not just localhost
-d          use USB device (error if multiple devices connected)
-e          use TCP/IP device (error if multiple TCP/IP devices available)
-s SERIAL   use device with given serial (overrides $ANDROID_SERIAL)
-t ID       use device with given transport id
-H          name of adb server host [default=localhost]
-P          port of adb server [default=5037]
-L SOCKET   listen on given socket for adb server [default=tcp:localhost:5037]
--one-device SERIAL|USB only allowed with 'start-server' or 'server nodaemon', server will only connect to one USB device, specified by a serial number or USB
--exit-on-write-error exit if stdout is closed

general commands:
devices [-l] list connected devices (-l for long output)
help        show this help message
version     show version num

networking:
connect HOST[:PORT] connect to a device via TCP/IP [default port=5555]
disconnect [HOST[:PORT]] disconnect from given TCP/IP device [default port=5555], or all
pair HOST[:PORT] [PAIRING CODE] pair with a device for secure TCP/IP communication
forward --list list all forward socket connections
forward [--no-rebind] LOCAL REMOTE forward socket connection using:
tcp:<port> (<local> may be "tcp:0" to pick any open port)
localabstract:<unix domain socket name>
localreserved:<unix domain socket name>
localfilesystem:<unix domain socket name>
dev:<character device name>
jdwp:<process pid> (remote only)
vsock:<CID>:<port> (remote only)
acceptfd:<fd> (listen only)
forward --remove LOCAL remove specific forward socket connection
forward --remove-all remove all forward socket connections
reverse --list list all reverse socket connections from device
reverse [--no-rebind] REMOTE LOCAL reverse socket connection using:
tcp:<port> (<remote> may be "tcp:0" to pick any open port)
localabstract:<unix domain socket name>
localreserved:<unix domain socket name>
localfilesystem:<unix domain socket name>
reverse --remove REMOTE remove specific reverse socket connection
reverse --remove-all remove all reverse socket connections from device
mdns check check if mdns discovery is available
mdns services list all discovered services

```

### Note

If you are installing on a different OS, check out these two links for install instructions.

[Linux Installation](#)

[Windows and MacOS Installation](#)

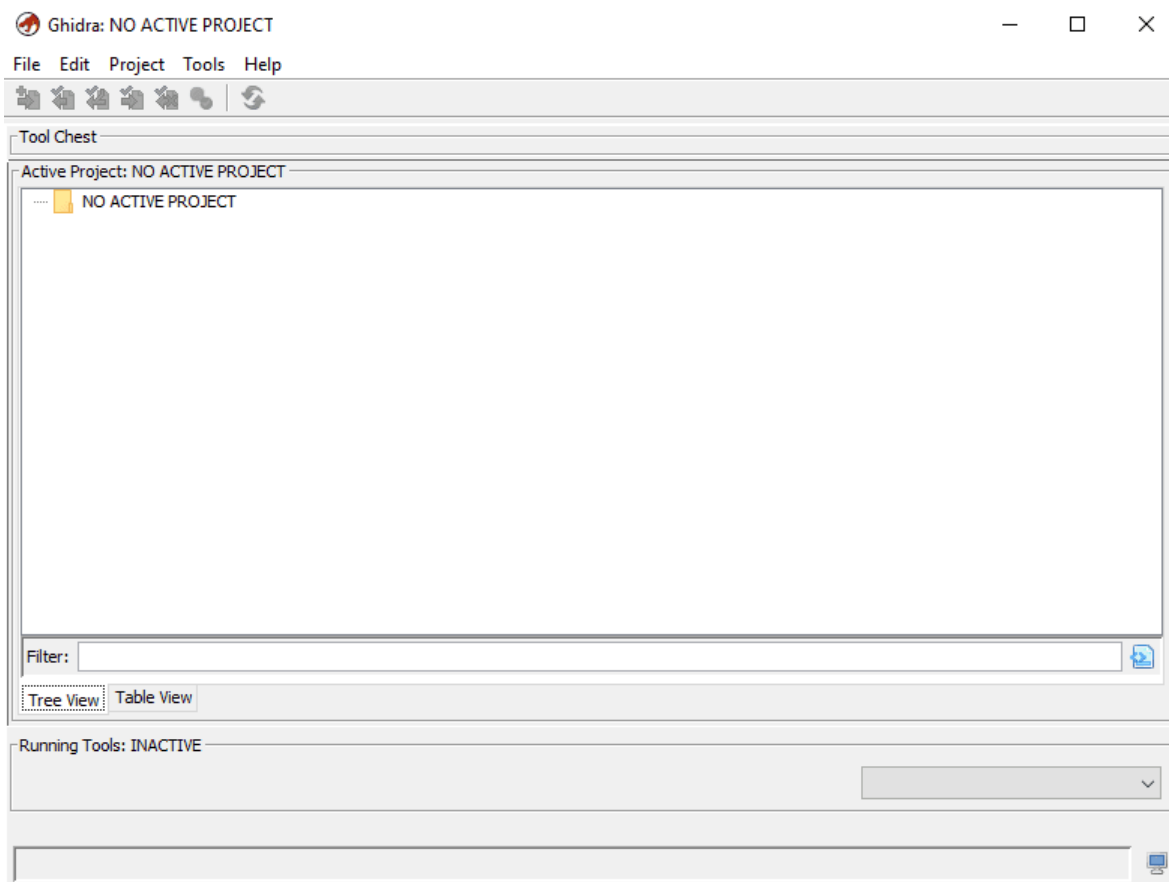
Congratulations, you have just successfully installed ADB and can now be interacting with Android phones. Join me in the next section, as we install our final main tool for this course Ghidra.

## 1.9 Tools: Ghidra

Our last, and arguably most powerful tool is Ghidra. Created by the National Security Agency (NSA) Ghidra is a reverse engineering tool and framework designed for working with native code. With many powerful capabilities, such as the ability to decompile almost any architecture, emulation, disassembly, and debugging, Ghidra can mostly do it all. For this reason, Ghidra is extremely useful for us to know as reverse engineers. Not only can it help us with Android reverse engineering, but it can help with other forms of reverse engineering we might explore in the future. With that being said, let us look into how to install this amazing tool.

The first step in installing Ghidra is installing the latest Java JDK version. To do this on the Linux VM we installed earlier, run the following command, "**sudo apt install openjdk-17-jdk -y**". This command will install the Java JDK for you and take care of setting that up, so we don't have to. Once that is done, we have to install Ghidra itself. To do this, go to the following [GitHub page](#) and download the most recent zip file. Once the file has been downloaded, unzip the file, and navigate to that directory. Next,

run the following command, **./ghidraRun**. If everything is installed correctly, you should see something similar to the image below.



With that, you have successfully installed Ghidra the last tool in this chapter. With Ghidra we can do many cool things such as emulation, decompilation, disassembly, and much more. However, we will get there much later. For now, let us continue learning the many great things there are to know about Android reverse engineering. Continue along, as in the next section, we will begin to look into Android and its security systems.

## Chapter 2 Android Basics

### 2.1 Android Overview: Why should we care?

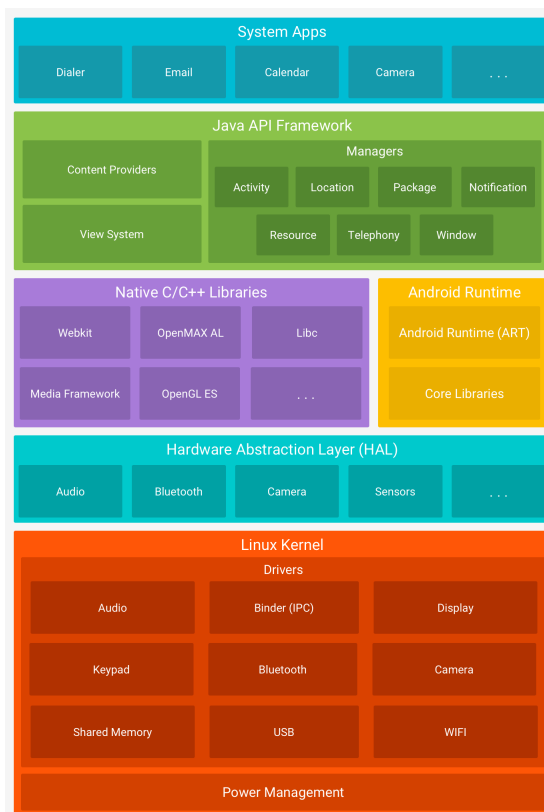
Before we begin, you might be thinking to yourself "Why Android?" And you would be right, why would we want to learn about Android reverse engineering? What if I were to tell you that Android has 73% of the global market share. Not convinced yet? How about the fact that all you need is \$25 and a phone to start developing on Android. This means that the barrier to entry is very low. Furthermore, with 73% of the market share, it proves to be a lucrative place for malware authors to create their malware or for shady businesses to steal your data.

However, you might be asking yourself "Doesn't the app store prevent this?". While they can, they are also human and like you or I, they can miss things. Furthermore, in Android you can side-load apps. This means that a user can self-install apps, without the use of an app store. This makes it that much more important for us to be educated and vigilant security professionals.

Hopefully, you are now beginning to see why the Android platform is important to understand, especially as security researchers.

With that being said, how does Android differ from your computer? While Android runs Linux, its approach to security makes the Android's external surface vastly different from your computer. But how is this done?

One way in which Android differs from Linux is through the use of its middleware. This middleware consists of the Java API Framework, Native C/C++ libraries, the Android runtime, and the Hardware Abstraction Layer (HAL). This middleware layer combined with Android's choice to not focus on UNIX processes makes the Android landscape rather different.



Furthermore, Android uses its API for common resources and inter-app communication. Android also has a strong use of components.

But what does this mean for us? Why should we care?

Understanding Android and its security model is crucial for us to later understand the code that we will be looking at. It's great if we can understand how to read Java code and learn to reverse engineer the Java bytecode, but what does that matter if we can't understand what it means in the bigger picture?

It might not make sense now, but hopefully throughout this chapter we will begin to see the importance of not only understanding Java, but also understanding Android itself.

In the next section, we will look at a brief overview of the Component Model in Android.

---

## 2.2 Component Model

In Android, the component model is one of the main drivers of how apps work. A solid understanding of the component model is crucial to a strong understanding of Android and will allow us to become even better Android reverse engineers.

But what is the component model and why is it so important,?

The component model is the underlying model of how parts of an Android app, or components, interact with each other. The component model consists of a few important parts.

1. Activities
2. Services
3. Content Providers
4. Broadcast Receiver

**Activities** are the parts of the app that take user input. The most important activity is the **Main Activity**. The main activity is often the entry point of an Android application and is often the window displayed to the user. **Services** are background actions that the app can perform. **Content Providers** are interfaces to data, for example an application might have to use a content provider to access the phone's local storage, a cloud server, or even program memory. **Broadcast Receivers** are components that wait for and react to events. For example, an application can have a broadcast receiver for SMS messages. This means that a broadcast receiver will sit there and wait to respond to an event that has been launched. However, these events can be invoked by either the app itself or an external app, but we will get more into this later.

At this point, you might be thinking to yourself "Why does this guy care so much about Android's internals? I just want to do RE." And to that I say don't worry, the importance of this will reveal itself soon. But for the impatient, let me give you some examples of why we should care about the component model and how this differs from your normal Linux computer.

What if I were to tell you that an Android application could have multiple entry points. Or about the fact that an app could accept requests from other apps, or even send data to other apps at their request. Or even that just because you close an app, it doesn't mean that it's not running. Or even that another app could start an app in the background without even telling you.

This is all thanks to the component model, which hopefully you can now begin to see that understanding the component model will help the rest of the code that we REs later make sense. For the

more astute of you out there, you might be wondering "Nic I thought you said that each program runs in its own user and often its own process, how can an app request information?" This comes from the component model and broadcast receivers.

So some of you might be thinking, this sounds very unsafe and why does Android allow this? This is because many of these functionalities are perfectly normal for most apps. Therefore, it is up to us, as security professionals, to understand what is bad and what isn't. To do that, we must first understand Android's component model.

So how do we use this model? For that we have to start with the Android Manifest file. But before we start with tha, let's talk about permissions. Android permissions will be our gateway into this journey. Are you ready?

In the next section, we will dive into Android Permissions. What they are and what they are used for.

---

## 2.3 Android Permissions

In this section we will look at Android Permissions, one of the critical backbones to Android and its security model.

So some of you might be wondering "What are Android permissions?". I'm so glad you asked. Android permissions are exactly what they sound like, they are the permissions users grant to an application. So why are they important and why should we care? Well, understanding Android permissions will not only allow you to understand what an app is doing, but also how to identify points of interest that will help make our later analysis easier.

So how do you use and interact with these permissions? You do this through the use of the Android Manifest; there will be more on that in the next section. For the time being, know that a developer simply declares the permissions that they want in their manifest file and the operating system (OS) does the rest.

We can declare permissions, so what? But what can they do and are they good or are they bad?

At their core permissions are not all bad and they allow developers to bring us many of the great features that we've come to know and love. However, Android themselves gives us some guidance here on how to discern these permissions. Android defines 3 permission types for us:

1. Normal: Very little risk, default value
2. Dangerous: For high-risk protected operations
3. Signature: When the requesting and declaring apps are signed with the same signature

**Normal permissions** are just that, they are normal. They involve very little risk and therefore are granted automatically when requested. Such permissions include Internet, Bluetooth, Receive boot complete, and Access WiFi state. While some of these might be worth looking at during static analysis, Android defines them as "permissions that allow access to data and actions that extend beyond your app's sandbox but present very little risk to the user's privacy and the operation of other apps."

In comparison, **Dangerous permissions** are permissions that are defined to perform high-risk protected operations that could pose a risk to a user's privacy. At this point some of you might be asking "If they pose a risk to user's privacy why do they exist?". You might be right, but first consider some of the functionalities defined as dangerous permissions. These include Camera, Read contacts, Write contacts, Send SMS, Read SMS, Call phone, Access external storage, and more.

Lastly, we have **Signature permissions**, which are permissions that are automatically granted across

applications that have been signed by the same key. In other words, apps that have been created by the same developer. The idea here is that apps from the same developer might want to be able to inter-op and so when a permission is granted to one application, if it is a signature permission, it is then automatically granted to all other permissions from the same developer assuming they request it.

However, some of you might be wondering how does one grant permissions to an app. This is through the use of 2 mechanisms. The first is the manifest—more on that later. The second mechanism that we are going to focus on a bit is Android's runtime permission scheme. The runtime permission scheme is the current way that Android grants permissions to the phone. As the name suggests, this is done by prompting the user at runtime when this permission is wanting to be used.

With that being said, why does this matter? Okay, so we just spent the better part of a page discussing Android permissions, but what does this have to do with Android Reverse Engineering? Why does this crazy guy care so much about the underlying Android system?

Well, don't worry we will get to the fun stuff very soon. One of the biggest questions I have been asked when teaching reverse engineering is "What are we looking for?". Unlike a gamified challenge where we are looking for a well-defined flag in the real world, we often have to come up with the thing we are looking for while we are looking for it. Therefore, in order to do this we must start with Android permissions. A strong understanding of Android's permissions will allow us to put what we look at into perspective and to become even better security professionals. At the end of the day it's great if we can read the code, but if we can't put that into the bigger picture and think critically about what we are looking at, we will have a hard time placing our analysis.

In the next, section we will begin to get our hands dirty with some real reverse engineering. To do this, we will look at the Android Packages and learn how to pull them apart.

## 2.4 Android Package: APK

Welcome Ladies and Gentlemen, if you have made it this far you are about to get our first taste of Android reverse engineering.

In order to continue this magical journey of understanding, we must first begin to understand how Android apps or APKs are built. By understanding the components of an APK, it will teach us where to look and help guide us in our early analysis. With that being said, what are the parts of an APK? What even is an APK?

### Definition 2.1

An Android APK is an archive file format for Android applications. Similar to zip files, APKs compress all of the app's content into a single file that can be read by the OS.

For this reason, we could use a tool like unzip to extract the contents of an APK. However, this wouldn't be a great idea since some files are compressed in additional ways that unzip can't handle. Such files include the manifest file. So with that being said, what are the parts of an APK?

An APK consists of 4 sometimes 5 parts:

1. **AndroidManifest.xml**
2. **META-INF** Directory
3. **res** Directory
4. **smali** Directory



The first component is the `AndroidManifest.xml` file. We will dive into this much more in our next section, but in short the manifest file is the sort of control file used for the entirety of the application. It defines the entry points, permissions, activities, receivers, and everything else the app wants or needs. From this file you can learn everything an app needs to function and where it will go to get those functions.

The next component is the `META-INF` directory. The `META-INF` directory contains all the information needed by the app to ensure that none of its internals have been tampered with. Given much of this directory just contains hashes of the assets included in the app. We won't spend time looking at this in detail. Many times we will never even look into this directory how it is good to understand that it exists and what the purpose of this directory is.

The next component is the `res` directory. The `res` directory many of the XML files for various assets within an app. For example, it contains a file named **`strings.xml`** containing all of the hard-coded strings used by an application. The data in this directory can be useful given that sometimes we can find things such as encryption keys used by apps should their developer hard-code and leave them in the app.

The last and arguably most important directory is the `smali` directory. This directory contains all the `smali` code used to make the application. In other words, the `smali` directory contains all the used by the application that we wish to analyze. From the `smali` directory, we can get every class, method, and underlying implementation. We will dive heavily into this directory and how to read and analyze its contents. However, for now, it is important to understand that this directory exists and is the location we will get all of our information from.

In short, these are all of the parts that an APK consists of. While some of these parts can change, these are the core ones to understand. From these parts, we can derive all the information we wish to know and reverse engineer any Android application.

In the next section, we will take a deep dive into the Android Manifest file and understand what it is and some key things to look out for when first looking at it. We will explore how to unpack an APK, find the manifest, and lastly how to find the entry point of the APK.

---

## 2.5 Manifest File

Like any good scientific process, we need to start with background or early analysis. In the case of Android reverse engineering, this comes from the manifest file. The manifest file is the driver for all of the later code that we will see and learn how to reverse engineer. This is our main driver, the first thing Android will look at to get information about an application. For example, the manifest will define the application's name as seen by the system, the minimum Android SDK version, the targeted SDK version, and so much more. The manifest file can be a treasure trove of information for static and dynamic analysis, and it all starts with understanding how to read it.

So let's go tackle this beast together. Are you ready?

The first question we must answer before we can fully understand the manifest file is what kind of file is the manifest file. The manifest file is a control file for an Android application and is written in the form of an Extensible Markup Language (XML) file. XML is a file format that is used to define data in the form of tags and their corresponding information. For example, an XML file could look like the following:

```
1 <note>
2   <to>Tove</to>
```

```

3     <from>Jani</from>
4     <heading>Reminder</heading>
5     <body>Don't forget me this weekend!</body>
6 </note>

```

The main thing to know about XML is that it's only responsible for carrying data and nothing else. This means, that at its core, the manifest file doesn't do anything special other than hold the information about the APK it is referencing so that the phone knows what the app can and cannot do. At this point, some of you might be thinking, "Great, let's just go grab it and get started." I agree with that but hold on a minute. You can't simply grab the manifest file. While it is an XML file, it's not just a file you can download from the Play Store and just read. It is compressed into an Android app's final binary, more commonly known as an Android Package (APK). As discussed in the previous section, the APK is compressed into a binary format, this includes the manifest file, therefore we cannot simply unzip the APK and read the manifest file. However, thanks to the work of many smart people, we can read APKs normally using tools like APKTool or Jadx. For right now, we will use APKTool. Later we will use Jadx, as it is one of the tools you are more likely to see when working on Android reverse engineering in the wild.

To analyze the manifest, we need to unpack the APK using APKTool. We can run the following command in the command line to do this. 'apktool d <apk>' This command takes the APK file you want to analyze and creates a directory with all the output information. Give the command a few seconds and when you are finished you should see a new directory in your current directory that new directory has the manifest we are looking for along with a lot of other information we will get into later. If you open this directory in your favorite text editor you can open the **AndroidManifest.xml** file. This file is the manifest that we want to explore.

```

1 <?xml version="1.0" encoding="utf-8" standalone="no"?>
2   <manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.
   example.vulnerableapp">
3     <uses-permission android:name="android.permission.INTERNET"/>
4     <uses-permission android:name="android.permission.READ_PHONE_STATE"/>
5     <application android:allowBackup="true" android:debuggable="true" android:icon="@mipmap/
   ic_launcher" android:label="@string/app_name" android:roundIcon="@mipmap/
   ic_launcher_round" android:supportsRtl="true" android:theme="@style/AppTheme">
6       <activity android:name="com.example.vulnerableapp.LauncherActivity">
7         <intent-filter>
8           <action android:name="android.intent.action.MAIN"/>
9           <category android:name="android.intent.category.LAUNCHER"/>
10        </intent-filter>android:exported=true</activity>
11      <activity android:configChanges="keyboardHidden|orientation|screenSize" android:
   exported="true" android:label="@string/title_activity_google" android:name="com.
   example.vulnerableapp.GoogleActivity" android:theme="@style/FullscreenTheme"/>
12      <meta-data android:name="android.support.VERSION" android:value="26.1.0"/>
13      <meta-data android:name="android.arch.lifecycle.VERSION" android:value="27.0.0-
   SNAPSHOT"/>
14    </application>
15 </manifest>

```

The manifest file should look something like this. From this file, we can learn a lot about the APK before even looking at the code. Before you begin to try and understand what is in this file it is important to know that XML follows a simple pattern. For each piece of data you have the tag (the label), and the data (the information being described). For example, **<uses-permission>** is the tag, and **an-**



**droid:name="android.permission.INTERNET"** is the data. So from this, there are 2 important pieces of data that you want to look at to begin your analysis. These 2 pieces of information are the permissions and the activity tags. The permissions are important because they give you insight into the kinds of capabilities that an application has or is requesting. This can indicate if an application might be doing something malicious. For example, you have a flashlight app that requests location permissions, does that make any sense? No, it does not. Therefore, that might be one piece of functionality you explore when looking at the code to determine whether or not an app is malicious. The second tag you want to look at is the activity tag. This tag is important because it will indicate the entry point of the application. In other words, it will give you the starting location of the application, and the place where the app will go when it first starts up. This can be extremely helpful especially in large applications as sometimes it is difficult to find the starting point. While lots of other information can be found within the manifest, these are the 2 most critical you must understand to begin reverse engineering Android effectively.

So in the example above, the permissions would be INTERNET and READ\_PHONE\_STATE and the entry point would be com.example.vulnerableapp.LauncherActivity. From this information, we can tell that this application can do 2 things: access the internet and internet-based components, and read the phone state this includes values like the IMEI, get Cell information and the Network Access Identifier. Some of this information is used to allow your phone to make calls while others are unique identification numbers attached to your phone. Therefore, this permission and the methods associated with it can be an important piece of information to track with reverse engineering an APK.

Congratulations, you now have the core knowledge needed to begin looking at code and diving into the heart of Android reverse engineering. Join me in the next section as we take a step away from Android for a bit and begin to learn how Java works beginning to learn to reverse engineer Java code before diving head-first into full Android reverse engineering.