

Android Post-Exploitation Framework

RIT Computing Security B.S. Capstone

By: Ayobami Adewale , Omar Aljaloud , Enzo DeStephano , Jake McLellan , Rob Olson

Contents

Contents	2
1 Introduction	3
2 Background	4
2.1 Android Security Model	4
2.2 Existing Android Malware	6
2.3 Android and Linux	7
2.4 Malicious Loadable Kernel Modules	8
3 Methodology	9
3.1 Kernel Implant	10
3.1.1 Compilation	10
3.1.2 Installation	11
3.2 Linux Implant	12
3.3 Application Implant	13
3.4 Attack Scenarios	14
4 Results	15
4.1 Kernel Implant	15
4.2 Linux Implant	16
4.3 Application Implant	17
5 Future Work	18
References	20

1 Introduction

With the rapid growth of smartphones, mobile devices have become an essential component for many users. The current number of smartphone users globally is about 6.567 billion, which means more than 82% of the world population has a smartphone (O'Dea, 2022). Smartphones are being used for everyday work through messages, emails, video calls, and social media networking. Thus, as more individuals have begun to use smartphones, the security of these devices has become an increasingly severe concern. Nowadays, smartphones are equipped with operating systems that resemble those found on desktop computers in terms of complexity. As a result of this development, smartphone operating systems are subject to many of the threats that affect desktop operating systems. Malware in the mobile space, specifically at the application and user-space layers, have become a large threat in recent years. McAfee discovered 2.3 million new mobile malware in the first three months of 2021, up 73% over the same period the previous year (Chee, 2021). As these are more commonplace, they are much easier to detect and prevent. As such, adversaries have to adapt and create more tools that will allow them to remain stealthy and maintain their access. For malware adversaries, there are three primary areas of developmental focus, each with their own benefits and downsides. The first is developing for the mobile application layer. Relative to the availability of development resources, This is the easiest and most common way of performing post-exploitation on a mobile device. While simpler to develop and implement, this level of development is highly abstracted when compared to the other options. These abstractions allow for security measures to be put in place which restrict the level of access obtainable by the application layer. The next option is to develop for the system level. This is the middle layer between the kernel and the application levels. While less documented, an increased amount of privileges are given at this level which can be abused by the malware developer. The final choice would be to develop for the kernel level. Interacting with the system at the lowest level is extremely difficult to research and develop, but provides the most access. Generally, the amount of protections in place increase around the areas that provide more access. This has to be a key concern of the malware adversary as they decide how they will perform post exploitation on a mobile device.

This paper will explore the process of development, implementation, and potential use cases of each of these options, specifically on Android mobile devices. As stated above, the relationship between privileged access and developmental complexity run in parallel to each other. For instance, developing a mobile application is extremely well documented. There are a multitude of resources available such as open-source code to guides for developers. While this option may be the most accessible for threat actors, it does not provide a high amount of access to the targeted devices and is extremely user-dependent.

Considering the other hand, a kernel-level rootkit for a mobile Android device is much more difficult to develop. Not only is the topic relatively inaccessible from research resources that exist on the Internet, but the lower-level protections that do exist are well-engineered and achieve their goal of preventing adversaries from obtaining this level of access. Thus, against these high walls of sparse documentation and intensive security protections, this project will aim to develop proof-of-concept modules for each of post-exploitation options that will contain the abilities of a more commonplace application-layer implant, to discuss the potential real-world use cases of this methodology, and to bring to light more information on the developmental processes for these items. Some of the specific desired capabilities of each implant are to provide remote shell access, as well as access certain mobile-specific features of the device like that of the camera and microphone.

With these goals in mind, the rest of this paper will cover a background of numerous items: the Android security architecture and model, the similarities and differences shared amongst Android and Linux, existing types of malware found on Android systems, and more details about rootkits as a technique. Further in this project, the methodology that the team will use when working through this project will be described, including both the development and installation process and requirements of each option. Lastly, the results of this work will be presented, investigating what worked well and what did not, as well as areas that could be explored in future research related to the topic of Android post-exploitation.

2 Background

This section aims to provide a description of the protections that Android implements at each of the three operating system layers. Additionally, the similarities between Linux and Android will be enumerated. With this knowledge in mind, existing threats by malware adversaries in the wild on the mobile platform are described, many of which have been affecting computers for more than a decade. This section will also observe why, in particular, rootkits are such an attractive type of malware and some background difficult development process.

2.1 Android Security Model

When it comes to Android system security, Android uses SELinux to establish mandatory access control (MAC) on processes (Android, 2022). For Android to ensure application security, Android enforces application sandboxing. Android assigns a unique Linux user id (UID) to each application to implement

application sandboxing, where each application runs in its own process (Android, 2022). Application sandboxing is set up on the kernel level by using the unique user id to set them up. Also, applications within the sandbox will not be able to access user data or system resources except if they had requested the permissions within the application. For all applications, Android uses application code signing to ensure that the application/APK belongs to the application's developers (Android, 2022). Android requires that all the applications are signed before generating Android App Bundle for the application to be installed or updated on the device. Signing the application ensures the integrity of the application to the Android Play Store. Trust-On-First-Use (TOFU) and the same-origin policy are used for signature verification of APK files (Elenkov, 2014). Android checks whether the same signers sign an application as the initiated one to perform signature verification. This signature verification check ensures that the same authors push the updates of the application as the original application. After that, it establishes a trusting relationship between the pushed update and the existing application to merge the updates. When the Android security model is trying to make security decisions such as roles and permissions for an application, it looks for the security attributes within the metadata of the AndroidManifest.xml file to perform the required security decisions for that application. Those Android security decisions are made by Android security models such as SELinux.

When descending to the kernel level, there exists an extended amount of security protection in place, which makes logical sense as privileges are highest at that level. Beyond the existing solutions in place for the other primary layers of an Android system, there are two security implementations acting as the main prevention techniques for kernel-level access abuse. The first protection on Android systems is the collaboration of Verifying Boot and dm-verity. Practically, Verifying Boot works by moving partition by partition and “cryptographically verifying all executable code and data” before it is used when booting (Android, 2022). Included in this is the Android kernel in the boot partition. The partition is hashed and compared to an “expected hash value” (Android, 2022). If they don’t match, the system will not boot. This works because when the kernel is being compiled, a private and public key pair are generated. The private key is burned into the device and thus unchangeable. The public key exists on the boot partition, and is used to verify the signature of the hash that is calculated when the Verifying Boot process is occurring. Now, considering that kernel modules have the ability to modify system behavior at a very low level, and the fact that large data partitions cannot be loaded into memory and easily hashed, dm-verity is utilized. Device-mapper-verity, or dm-verity, checks the integrity of block devices. By viewing the “underlying storage layer of the file system,” it is possible to determine if tampering has occurred (Android, 2022). Dm-verity takes 4096 byte blocks of the partition and works in parallel to utilize all of them to build a SHA256 hash tree. If any of the blocks are unexpected or modified, the dm-verity will fail

and the device will not boot. In the case of a malware adversary who is developing a kernel module, these technologies would prevent kernel modules from being loaded before the boot process occurs. They would either 1) have to be able to get around the bootloader and its protections or 2) load the module after the device has passed this step. The latter option is more feasible, but a different approach to persistent loading would need to be implemented. Now, Android does provide support for loadable kernel modules past version 8.X, but they do have to follow certain requirements to be utilized: modules have to be loaded from read-only and verified partitions, and cannot be in certain partitions like /system (Android, 2022). Since the kernel rootkits being used here are not valid and/or signed by a legitimate manufacturer, they will not follow these rules and must be loaded out-of-tree. The command “insmod,” used to load a loadable kernel module on Linux systems, exists on Android systems and does work. However, root privileges are required to use the command, which is another barrier of entry for the malware adversary.

2.2 Existing Android Malware

While mobile operating systems have been designed in such a way that deters malware from impacting infected devices, that has not deterred malware authors entirely. Commodity malware such as “Joker” are prevalent throughout the Google Play store and can steal SMS data and contact information from infected devices if given correct permissions (Suau, 2022). Often, less sophisticated mobile malware like this is distributed to unsuspecting users in non-targeted attacks via social engineering attacks and is designed to infect a widespread range of users. Some of the more advanced instances of mobile malware come from NSO Group, an Israeli organization responsible for the creation of the Pegasus mobile malware. Pegasus is a form of spyware capable of targeting microphone and camera hardware, along with SMS message history from targeted Android and iOS devices. In one instance, human right activist Ahmed Mansoor was targeted by attackers utilizing an iOS zero-day exploit chain to deploy Pegasus. (Marczak & Wetangula, 2016). A report from Amnesty International’s Security Lab detected forensic evidence of intrusion linked to NSO Group on at least 41 devices (Amnesty International, 2021). From trivial application-level malware to nation state grade surveillance spyware, the Android threat landscape is composed of a wide array of technical capabilities.

Pegasus is one of the few known malware that utilizes multiple layers of the android system to target individuals. The Pegasus malware begins from the application level. At the level, it uses its super user access to perform some surveillance like acquiring the databases of some popular apps like Facebook, WhatsApp, Twitter and others. Live audio is collected under specific situations like locked screen, microphone not in use, music isn’t playing and more. For screen capture, pegasus utilizes a binary called

“screencap” which exists on some installations under the `/system/bin/` path. If that binary exists, it takes screenshots and saves it in PNG file format to `/data/data/com.network.android/bcl4.dat`. For devices that do not have that binary, Pegasus uses a native screen capture implementation that is provided by the `take_screen_shot` binary, located in the `applications/res/raw` directory. Keylogging was achieved by injecting itself into the keyboard process. The binary in use for this is called `libk` which was stored in the `res/raw` directory of the application. During execution, Pegasus writes the binary to another location, `/data/local/tmp/libuml.so`. The binary immediately injects itself into the keyboard process via process id before deleting itself from the device. (Pegasus Analysis, 2017) The capabilities of this malware really emphasize on the advantages and disadvantages of the different layers of the android operating system.

2.3 Android and Linux

There exist a large amount of similarities between Android and Linux operating systems. Android itself is actually built and maintained by Google as a modified version of the Linux kernel (Android, 2022), hence why there exists much overlap between the two. In fact, as of 2019, Android switched from the system of grabbing new LTS releases of the Linux kernel and making specific merges and modifications. Now, they maintain a branch of the LTS kernel called “android-mainline” (Android, 2022). As the new Linux LTS stable kernel branch is updated, changes are merged into a sub-branch of android-mainline for testing. After a considerable amount of time and testing on the sub-branch, the changes are merged into the mainline for the Android kernel, thus saving time and resources (Android, 2022). Considering this point, this overlap provides both pros and cons to all sorts of engineers, whether for developing applications or security researchers in their testing. As previously mentioned, both Linux and Android are kept open source, which means that all of the source code for the operating system is made public.

In regards to this project and these similarities, there are a couple main points to enumerate. When considering the development of modules for the Android kernel, there is some intersection between the two operating systems. This will be expanded on later in this paper, but at a high-level most of the actual coding of the module is the same for each OS. For example, a hello world module for both is programmed in C, utilizes similar headers and formatting, and is installed in the same way. The process of developing and compiling is where the actual differences lie. Another important item to note is at the system layer, the similarities increase. While the CPU architecture might differ from device to an emulated system, all native binaries on a system are ELF (Executable and Linkable Format). Just like on a pure Linux system, these ELF binaries used shared object libraries (.so) like `glibc` for dynamic function linking. This comes

into play when choosing the development environment for an Android system binary being created and compiled.

2.4 Malicious Loadable Kernel Modules

Loadable Kernel Modules (LKMs) are object files that can be loaded into an already-operating kernel. They are mainly used to add new features to the kernel, such as device drivers, filesystem drivers, and system calls. However, maliciously, these assume the name “rootkit” and are used to maintain the highest level of access from within the kernel. When an adversary can get the Linux administrator to load a new malicious module to their kernel, the adversary owns full control over their system since the malicious module will run at the kernel level of the Linux administrator’s (victim’s) operating system, granting the adversary complete control over the victim's system. An example of a Linux Loadable Kernel Module is Drovorub. Drovorub is a Linux malware tool set consisting of various malware toolsets, and one of them is an implant coupled with a kernel module rootkit (National Security Agency, 2020). The kernel module rootkit hides itself and the implant on infected devices via various techniques. It persists even after a reboot unless UEFI secure boot is configured in "Full" or "Thorough" mode. Moreover, a recent technique is adversaries use Linux Loadable Kernel Module to create cryptocurrency-mining malware. For instance, Skidmap is a Linux malware that installs malicious kernel modules in order to keep its cryptocurrency mining processes hidden (Remillano II & Urbanec & Luy, 2019). Another malicious kernel module used in the wild is the “snd_floppy” rootkit discovered by Sophos as part of the “Cloud Snooper” campaign which allowed the attackers to bypass firewall restrictions when communicating with infected devices (Shevchenko, 2020). Each of these malware families, though unique in their goals and functionality, demonstrate the capabilities of malicious Linux Loadable Kernel Modules.

The term "rootkit" was first used to describe a set of tactics used by attackers to hide the existence of malicious software on a compromised machine. Even though rootkits have been a threat to desktop machines for a long time because of their large attack surface, similarly, smartphone operating systems have become attractive for rootkit authors, making smartphones as vulnerable as desktop rootkits. A rootkit is often installed after an attacker gains elevated privileges on a system. Furthermore, one of the widespread techniques for an attacker to gain a foothold into the system is by exploiting software vulnerabilities. A rootkit can be used to open the door to a variety of attacks once the system is infected. Rootkits, for instance, are frequently used to hide keyloggers, which secretly monitor keystrokes and record sensitive user data such as passwords and credit card details. They might even install backdoor applications on the machine, allowing a remote attacker to obtain access in the future. A rootkit may and

should hide many things to be stealthy, including processes and files, and three of the most common hiding techniques are hooking, patching, and data structure manipulation (Nerenberg, 2007). Modern rootkits (kernel-level rootkits) have been developed to manipulate the operating system's code and data structures since rootkits that impact user-level files are easily identified. The operating system's data structures that store control data, such as the system call table, and other function pointers, which determine control flow in the kernel, are among the most common targets of such kernel-level rootkits. Rootkits use a hooking technique to interpose themselves in the kernel's control chain and hide malicious functionality and objects, including files and processes.

More threateningly, since rootkits can achieve their malicious objectives stealthily, they remain undetected and enjoy long-term persistence over the infected system. Due to the increased levels of user freedom that exist within a desktop environment, rootkits are much more prominent on these types of operating systems. With the increased difficulty and protections that exist on mobile devices, developing and implementing kernel-level rootkits turns into a much more arduous task.

3 Methodology

One of the first decisions one must make when writing a post-exploitation framework for Android devices is the permission level at which it is expected to run. A lower privileged implant, commonly distributed as a standard Android application (APK file), would be able to infect a larger quantity of devices in a less substantial way. A medium privileged implant distributed in the form of an ELF binary would allow for code execution on the infected device, though it would be difficult to produce in a scalable way. A higher privileged implant distributed in the form of a Loadable Kernel Module would need to be compiled with the same kernel version as the intended target, making it substantially more difficult to write in a scalable, impactful way. With this in mind, the decision was initially made to proceed with a highly privileged implant since it would provide a greater level of access, but eventually due to the developmental difficulties and the inherent lack of scalability, the primary goals of the project shifted to focus on the investigation of each of the three layers.

3.1 Kernel Implant

3.1.1 Compilation

Depending on your target, the kernel would have to be obtained from the developers' website. For this research, the team planned on targeting the latest android kernel for a larger audience. Since the kernel has been developed over many years, the codebase is quite large. To perform our experiment in a safe place, we utilized a virtual machine that has at least 100gb of storage and 8gb of RAM. After the creation of the environment, three main things (find a better word) are needed. The kernel source code of your target device, software development kit (SDK), and native development kit (NDK) from the Android developer website.

It is important to note that depending on the kernel version, the option to create a kernel module might be turned off. In order to fix that, search this line, "CONFIG_MODULES=n" in the config file and change it to "CONFIG_MODULES=y".

With those changes, building the kernel was straight forward. Simply executed the build script as such, "`bash ~/<kernel source>/build/build.sh`". With a successful compilation, a .img file was created. Next is writing a simple kernel module to load onto the system. This is considered one of the easier processes of this research. The team's proof of code was to print out "Hello world" from the kernel space shown in Figure 1.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. The terminal displays the source code for a kernel module. The code includes headers for linux/module.h and linux/kernel.h. It defines two static functions: sample_init, which prints "Hello Kernel World...\n" and returns 0, and sample_exit, which prints "Goodbye Kernel World\n". Finally, it calls module_init(sample_init) and module_exit(sample_exit) to register the module with the kernel.

```
#include "linux/module.h"
#include "linux/kernel.h"

static int sample_init(void) {
    printk("Hello Kernel World...\n");
    return 0;
}

static void sample_exit(void) {
    printk("Goodbye Kernel World\n");
}

module_init(sample_init);
module_exit(sample_exit);
```

Figure 1 (Carbon, 2022)

In addition to the module code, a makefile is needed to instruct the system how to properly compile. The team's Makefile looks like Figure 2:



```
obj-m += sample.o

KERNEL_DIR=~/Desktop/pixel6/out/android-gs-pixel-5.10/private/gs-google
CROSS_COMPILE=~/Desktop/ndk/toolchains/llvm/prebuilt/linux-x86_64/bin/x86_64-linux-android31-

all:
    make -C $(KERNEL_DIR) M=$(PWD) ARCH=x86 CROSS_COMPILE=$(CROSS_COMPILE) modules
clean:
    make -C $(KERNEL_DIR) M=$(PWD) clean
```

Figure 2 (Carbon, 2022)

To explain some of the components of this Makefile, the `KERNEL_DIR` variable points to the source code of the kernel. When compiling, there are a multitude of libraries and headers being utilized and their location is not automatically known by the system. Moving forward, the `CROSS_COMPILE` value points to the directory where the specific compiler itself lies. In this case, the `x86_64-linux-android31` is utilized as it matches the correct Android version and the CPU architecture being utilized. Lastly, the `PWD` variable directs the compiler to the current directory where the kernel module source code is and where it should be built.

Once compilation is done, an ELF file with a `.ko` (kernel object) file extension should be created. Once the object is prepared, the next step is installing the module on the system.

3.1.2 Installation

As for the actual installation of the module itself, there are two main points to focus on here. The first is that all of the symbols and functions are versioned correctly during the compilation of the module. If anything does not line up during either the development process (ex. Using outdated functions or libraries) or compilation process (ex. Using the wrong Android NDK compiler), the module will either cause a crash or the system will prevent its installation. The second is the actual method of inserting this module into a working device. There exists a default command on Android systems called “`insmod`”. As

the name suggests, this command will take a given file and install a kernel module into the kernel. Recalling from earlier, due to the fact that protections like dm-verity exist on Android systems, persistent loading of kernel modules is extremely difficult. However, the team's aim was not to provide a solution or bypass to these protections, but instead provide a proof-of-concept. If properly working, the module would just have to be installed after the Android boot process has finished.

3.2 Linux Implant

Due to the high amount of similarities between Linux and Android systems, research had to be done in several different areas for a successful implementation to occur. The explored areas are a compilation of system-level binaries, file migration and execution, the differences between Linux and Android native binaries, and the advantages and disadvantages of implementing an implant at the Android Linux level. Considering the first stage of exploring the compilation of system binaries, the team has focused on what target headers and libraries are needed to perform a successful compilation. Alongside these versioning differences, the CPU architecture is extremely important for compilation. So, the identification of x86 architectures versus ARM architectures is simple, but the most important aspect of the development process. Additionally, the limitations of dynamic and static linking when compiling were explored. Within the file migration and execution, the team has looked into which filesystem sections are writable and which filesystems are immutable. This would be helpful to identify which area of the Android file system is optimal to store and execute the Linux implant. While identifying the differences between Linux and Android native binaries, the team has observed what Android native binaries could be used to initiate and build an implant. With this information, it would lead to either having to develop more custom code for the implant if the features did not already exist at the system layer (in the form of a native binary). The last area of the research focused on the advantages and disadvantages of implementing an implant at the System level. This was an important consideration as to determining if future work should be done in this area and to decide if this level is considered implementing - especially since there were a lack of known benefits, drawbacks, or just general knowledge when it comes to the system layer of an Android device.

Moreover, aside from the research, the team plans on constructing two simple proof of concept implants on the Android system. The first proof of concept is writing a simple C code, where the code only prints "Hello world!". This simple proof of concept aims to test the compilation process and observe the behavior of running a simple binary. The second proof of concept is writing a C code that establishes a reverse shell and bind shell on the Android system. This code utilizes the networking aspect of the

implant on the Android system. This proof-of-concept aims to observe the possibility of constructing a universal binary that could run on both Linux and Android systems.

3.3 Application Implant

Finally, the team looked into the feasibility and usefulness of implementing an Android application level implant. The goal of this implant was to determine the level of functionality that can be implemented at the application level within the confines of Android's standard application permissions. Primarily, a focus was placed on developing a proof of concept application capable of remotely executing commands in the form of a bind shell. All test applications were developed in Java using Android Studio with the target API version set to Android API level 31 (Android 12), the most recently released version at the time of development.

Initially, a control application was developed in Java which uses the `Log.i()` function to print an arbitrary string to the device's system information log. Once this application was confirmed to function within the testing environment, work could begin on a simple proof of concept for the Android application level. The first stage of this proof of concept is to get basic command execution working. This is accomplished in Java using the function `Runtime.getRuntime().exec()`, the typical approach to execute commands and receive the output. The function below demonstrates the implementation of this capability.



```
private String execute(String command){
    StringBuffer output = new StringBuffer();
    Process process;
    try {

        process = runtime.getRuntime().exec(command);
        process.waitFor();
        BufferedReader reader = new BufferedReader(new InputStreamReader(process.getInputStream()));

        String line = "";
        while ((line = reader.readLine()) != null){
            output.append(line + "\n");
        }

    } catch (Exception e){
        Log.e("EXEC", e.toString());
        return e.toString();
    }

    return output.toString();
}
```

Figure 3 (Carbon, 2022)

Next, a TCP server is required which allows for remote connections to be made to the implant. This server code should receive commands from incoming connections, process the command, and return the output of the command execution. It is worth noting that this approach requires the normal permission *android.permission.INTERNET* to function properly. Once both parts of the bind shell are operational, they can be combined to complete the proof of concept. This implant can then be used by running the application on the victim device and connecting to the specified TCP port (9001) via the Netcat utility.

3.4 Attack Scenarios

The team has also developed several real world attack scenarios regarding the installation of each implant type. In these attack scenarios, the details of each hypothetical attack is dependent on the level of access desired as well as the perceived/expected skill level of an adversary. Another consideration worth considering for each attack scenario is the scalability of implant distribution. Although highly targeted attacks will not benefit from scalability, adversaries looking to infect a high quantity of victims will have to consider this when choosing the distribution method that best fits their needs.

The simplest attack scenario is to use a benign-looking application to mask malicious functionality and distribute the APK file to the desired target with an enticing phishing message. Similarly, it may also be possible to abuse the Google Play Store for widespread, non-targeted distribution of a benign-looking application level implant. This method of distribution is one of the most scalable, allowing for a high infection rate. In 2020, researchers claimed that the Google Play Store was the primary method of distribution for Android malware analyzed, accounting for 87% of all Android malware installations, while additional marketplaces accounted for an additional 5.7% (Kotzias et al., 2020).

At the system level, the Pegasus malware is a great example of this. Originally spreading via the Play Store, malware was able to propagate and install itself at the system level via a CVE. Thus, malware adversaries are able to follow this model in order to get their implant on a phone at the system level. If a vulnerability is not known, it is also possible to use ADB to move the implant onto the device (either via cable or TCP/IP), although this is less commonplace. This method of distribution would not be highly scalable, though it is simple to successfully deploy to devices within physical access of the adversary.

Persistently loading a kernel module in Android requires a custom-compiled kernel. The most sophisticated attack vector involves the interception of devices in shipment to allow for the custom Android version to be installed. From there, the devices could be loaded with the kernel module implant.

Therefore, this attack vector requires coordination with the organization handling the shipment making it infeasible to successfully pull off for most non-government entities. In the past, it has been reported that the NSA has utilized this attack vector, which it refers to as “interdiction”, against multiple electronic device types (Appelbaum et al., 2013). This method of distributing the implant is the least scalable as it requires the most effort per infected device. It would likely require Android to be compiled multiple times over to work across different devices and Android versions. Therefore, this method of distribution would allow for the adversary to gain a high level of access into the Android devices of a small number of targets.

4 Results

The results section will cover the results of research and development for this project. For each of the three layers, the simple proof-of-concept code can be viewed within this document or can be referenced at: <https://github.com/O72/Android-Post-Exploitation>. Each sample will be described and explained in detail below.

4.1 Kernel Implant

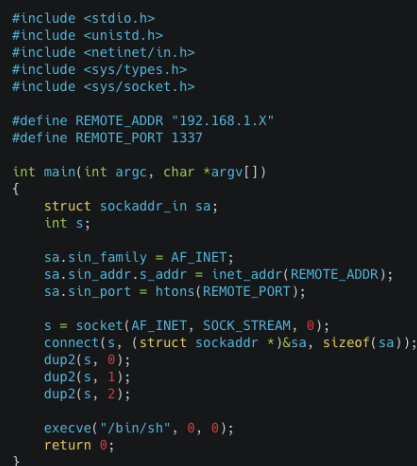
At a high level, the goal of compiling a kernel module for an Android device was unsuccessful. However, certain characteristics of the development process were brought to light while attempting to solve this issue. First off, since it is extremely difficult to expand on the kernel to a non-custom Android via a module, the team obtained the Google Pixel 6 Android source code and compiled the kernel. The VM was deployed in a dedicated virtual machine. Usually, the manufacturers remove the build headers and symbols from a distributed device, but by compiling the kernel ourselves, we were able to obtain some of the missing files. However, there were still issues when trying to compile the module itself. The source code was valid, but once again the build headers were either missing or named incorrectly across the board. Any attempts to fix these issues uncovered more similar problems. Additionally, when using the NDK’s provided compilers, there were issues concerning the Makefiles contained within the kernel source code and the pointers in the Makefile our team developed. With these problems along with others, a kernel module was unable to be compiled. Related to this, more information will be provided in the future work section of this paper.

4.2 Linux Implant

The research in this section has revealed various results. During the compilation stage, the team has identified several limitations with dynamic and static linking. Dynamic linking would be sufficient when the target system version is known; however, static linking is better to be used otherwise. The team has used static linking to perform the proof of concept. To successfully compile the proof of concepts, the team had to compile the implants on an external Linux system and compile them statically using gcc. After the compilation process, the binaries were transferred to the Android system using ADB. At this stage, the team has identified the limitations of certain Android systems, where many of the sections of the filesystem have noexec permissions that are immutable, such as /sdcard and /system. However, the team has identified that /data/local and /data/local/tmp are two of the best filesystem directories to migrate and execute the proof of concept samples.

Furthermore, the team developed and executed the planned proof-of-concepts. All the “Hello world!”, reverse shell, and the bind shell executables were compiled using a Kali Linux machine using GCC and then transferred to the Android system using ADB, where they were placed in /data/local/tmp and executed. The bind shell and reverse shell both initiate a TCP connection on port 1337.

While the PoCs might be simple in terms of implant capabilities, these binaries provide a working example of a system-level binary that was able to be successfully developed and deployed. A snippet of the code is shown below as reference in Figure 4.



```
#include <stdio.h>
#include <unistd.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>

#define REMOTE_ADDR "192.168.1.X"
#define REMOTE_PORT 1337

int main(int argc, char *argv[])
{
    struct sockaddr_in sa;
    int s;

    sa.sin_family = AF_INET;
    sa.sin_addr.s_addr = inet_addr(REMOTE_ADDR);
    sa.sin_port = htons(REMOTE_PORT);

    s = socket(AF_INET, SOCK_STREAM, 0);
    connect(s, (struct sockaddr *)&sa, sizeof(sa));
    dup2(s, 0);
    dup2(s, 1);
    dup2(s, 2);

    execve("/bin/sh", 0, 0);
    return 0;
}
```


Figure 4 (Carbon, 2022)

With this system access, there are some important considerations to be made - especially when comparing this implant to that of the application implant. Returning to Android's security model, their implementation of application sandboxing is extremely well-designed. To quickly recap, it prevents applications from accessing the data and resources of other applications. It will be further described in the next section, but since this implementation was developed for and functions on the Android system layer, it does not run into the same issues that an application implant does. A great example of this is its ability to access the confidential data of most processes. An example of this would be a SQLite database that an application might use in order to store cached login information. Now, it was enumerated that this does not contain the highest level of privileges, and is unable to perform certain actions (like making partitions immutable), it is still miles above an application implant in terms of usability, all while providing a simpler development process than that of a kernel-layer implant.

To summarize, a Linux implant provides much more access than an application implant. Some advantages are that it does not require system permissions to use some functionality compared to application level. It is able to use existing Linux tactics like defense evasion more easily. Since Linux is open-source software, there is a significant amount of documentation and resources available for development which makes the whole process quicker. The downside to developing a Linux implant is the amount of prior knowledge necessary. Without an understanding of the Linux kernel and it works, it's possible to create an unstable implant. Even though the Android operating system is a fork of the Linux kernel, some functionality has been stripped off to make it much smaller and compatible for smaller devices. So when developing for Android at the system layer, it is worth noting that there is limited documentation on the variations and differences between Linux and the target Android version. Despite these downsides, the team believes that Linux implants present significant potential for implant development as they provide a middle ground within the implant types in terms of difficulty to implement and access granted to the attacker.

4.3 Application Implant

While the team was able to successfully implement a functioning bind shell at the application layer, its limitations were quickly revealed. Testing the application level implant revealed the effectiveness of Android's security architecture and application sandboxing. In its current form, the proof of concept is able to handle basic remote command execution. However, the number of commands that an adversary

can successfully execute is highly limited with this approach. One command that functions as intended is “id,” which shows that the command is being executed from the context of the user “u0_a157,” the user account for the proof of concept app. The screenshot below shows the resulting output of the “id” command when executed from the context of the bind shell.

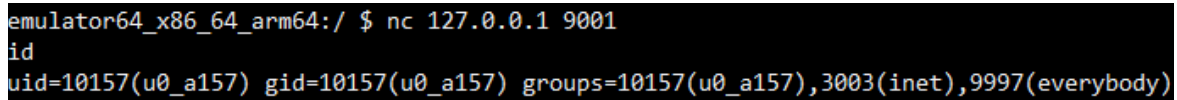
A screenshot of a terminal window with a black background and white text. The prompt is 'emulator64_x86_64_arm64:/ \$'. The user has entered 'nc 127.0.0.1 9001' and then 'id'. The output of the 'id' command is 'uid=10157(u0_a157) gid=10157(u0_a157) groups=10157(u0_a157),3003(inet),9997(everybody)'.

Figure 5

Commands which depend on file-system access such as “ls” or “cat” do not function as they would from an account with “system” or “root” level permissions. Interestingly, other file-system related commands including “mount” and “df” work as expected. For these reasons, this approach would not appear to be of much use to an attacker as the limited command execution greatly impedes post-exploitation activities.

5 Future Work

There are numerous directions that future work in this area of research can take. Each implant type researched provides potential for future development. In its current state, the team believes that the Linux system level implant has the highest potential to develop further capabilities useful for a post-exploitation framework. The level of access provided by the implant, the relative simplicity of development at this level, and the lack of prior research and development makes this level ideal for further research. Future capabilities worth researching and developing further at this level include remote file transfer, information stealing, and potential camera/microphone access. These functionalities are typically associated with spyware. However, at the application layer, they are restricted by permissioning. It is worth investigating if these restrictions can be bypassed by deploying an implant at the Linux level.

Additionally, when considering the kernel layer of an Android system, the lack of resources that exist in terms of modern documentation and guides for kernel module development mean that there is much work that can be done. Successfully compiling a functional, basic “Hello world!” kernel module specifically for the team’s Android environment would be a step in the right direction that would allow for the further exploration of Android kernel level implants and other future work in this highly-privileged space. With the issues that were overcome with compiling the custom kernel and the information gained through this

work, successfully compiling a proof-of-concept kernel module for an Android device is closer to being possible.

Another consideration for future work exists within the application level. The main obstacles encountered at this level were restrictions due to application sandboxing and SELinux. Therefore, future work would be required to bypass these restrictions in some way. Privilege escalation could be achieved by exploiting either known vulnerabilities or using newly-discovered vulnerabilities to execute code from the context of a user with higher privileges.

A long-term, more ambitious goal within this project would be to implement the implant levels together to form a true framework. This framework would give adversaries the opportunity to determine which level of access they feel is necessary for a given scenario and deploy the implant they feel fits best in a simple, scalable solution. Ideally, the framework would remove the overhead of up-front technical development that makes Android post-exploitation difficult to begin with. This would help to make mobile penetration testing and security research more accessible.

This research has been very knowledgeable to the team from a technical perspective. Much was learned; however, the team would emphasize some essential takeaways. First, the Android security model has come a very long way in a very short time. Measures like Application Sandboxing and Verified Boot can severely restrict the level of access to an adversary. Moreover, the team learned that, unfortunately, kernel development is extremely challenging simply because the amount of public research on the area is very limited, or mostly non-helpful information. Therefore, this restricted our research on kernel development. Second, the team was stunned by the potential of the system level. Development was not substantially more complex than the application layer; however, the level of access provided and the versatility of system implants makes the system level reasonably practical. Therefore, the team believes there is unrecognized potential in this area of research and that adversaries will eventually notice this in the future. Finally, the team was able to investigate the strengths of Android's security model at all levels and research the development process for each layer. The team was able to successfully develop PoCs for two of the three layers and, with the kernel layer, able to compile our own kernel for both R&D and future work. Overall, with the knowledge gained in this project, the amount of information pertaining to Android post-exploitation and its developmental difficulties and processes has increased considerably. As a result, achieving the final goal of creating an overarching framework that simplifies this work is one step closer.

References

- Amnesty International. (2021, July 18). *Forensic Methodology Report: How to Catch NSO Group's Pegasus*. Amnesty International. Retrieved February 20, 2022, from <https://www.amnesty.org/en/documents/doc10/4487/2021/en/>
- Android. (2020, September 1). *System and kernel security*. Android Open Source Project. Retrieved April 10, 2022, from <https://source.android.com/security/overview/kernel-security>
- Android. (2020, September 1). *Verifying Boot*. Android Open Source Project. Retrieved April 10, 2022, from <https://source.android.com/security/verifiedboot/verified-boot>
- Android. (2022, February 15). *Security-Enhanced Linux in Android*. Android Open Source Project. Retrieved April 10, 2022, from <https://source.android.com/security/selinux>
- Android. (2022, March 18). *Application Sandbox*. Android Open Source Project. Retrieved April 10, 2022, from <https://source.android.com/security/app-sandbox>
- Android. (2022, March 18). *Application Signing*. Android Open Source Project. Retrieved April 10, 2022, from <https://source.android.com/security/apksigning>
- Android. (2022, March 18). *Loadable Kernel Modules*. Android Open Source Project. Retrieved April 10, 2022, from <https://source.android.com/devices/architecture/kernel/loadable-kernel-modules>
- Appelbaum, J., Poitras, L., Rosenbach, M., Stocker, C., Schindler, J., & Stark, H. (2013, December 29). *The NSA Uses Powerful Toolbox in Effort to Spy on Global Networks*. Spiegel. Retrieved May 1, 2022, from <https://www.spiegel.de/international/world/the-nsa-uses-powerful-toolbox-in-effort-to-spy-on-global-networks-a-940969.html>
- Carbon. (n.d.). *Carbon Code Diagrams*. Retrieved May 1, 2022, from <https://carbon.now.sh/>
- Chee, K. (2021, July 5). *Global increase in mobile malware but smartphone security lax in S'pore*. The Straits Times. Retrieved February 20, 2022, from

- <https://www.straitstimes.com/tech/tech-news/global-increase-in-mobile-malware-but-smartphone-security-lax-here>
- Elenkov, N. (2014). *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. No Starch Press.
- Kotzias, P., Caballero, J., & Bilge, L. (2020, October 20). *How Did That Get In My Phone? Unwanted App Distribution on Android Devices*. arXiv. <https://arxiv.org/abs/2010.10088>
- Lookout. (2017, April <https://info.lookout.com/rs/051-ESQ-475/images/lookout-pegasus-android-technical-analysis.pdf>) . *Pegasus for Android: Technical Analysis and Findings of Chrysaorw*.
- Marczak, B., Scott, J., & Wetangula, M. (2016, August 24). *The Million Dollar Dissident: NSO Group's iPhone Zero-Days used against a UAE Human Rights Defender - The Citizen Lab*. Citizen Lab. Retrieved February 20, 2022, from <https://citizenlab.ca/2016/08/million-dollar-dissident-iphone-zero-day-nso-group-uae/>
- National Security Administration. (2020, August 13). *Russian GRU 85th GTSs Deploys Previously Undisclosed Drovorub Malware*. Department of Defense. Retrieved February 20, 2022, from https://media.defense.gov/2020/Aug/13/2002476465/-1/-1/0/CSA_DROVORUB_RUSSIAN_GRU_MALWARE_AUG_2020.PDF
- Nerenberg, D. D. (2007, March 5). A Study of Rootkit Stealth Techniques and Associated Detection Methods. *Theses and Dissertations*. <https://scholar.afit.edu/cgi/viewcontent.cgi?article=4107&context=etd>
- O'Dea, S. (2022, February 17). *Number of smartphone subscriptions worldwide from 2016 to 2027*. Statista. Retrieved February 20, 2022, from <https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/>
- Remillano, A., Urbanec, J., & Luy, W. (2019, September 16). *Skidmap Malware Uses Rootkit to Hide Mining Payload*. Trend Micro. Retrieved February 20, 2022, from

https://www.trendmicro.com/en_us/research/19/i/skidmap-linux-malware-uses-rootkit-capabilities-to-hide-cryptocurrency-mining-payload.html

Shevchenko, S. (2020, March 2). *Cloud Snooper Attack Bypasses AWS Security Measures*. Sophos.

Retrieved February 20, 2022, from

<https://www.sophos.com/en-us/medialibrary/PDFs/technical-papers/sophoslabs-cloud-snooper-report.pdf>

Suau, R. (2022, January 20). *New Joker malware detected on Google Play, 500.000+ users affected*.

Mobile Security Blog | Pradeo. Retrieved February 20, 2022, from

<https://blog.pradeo.com/pradeo-identifies-app-joker-malware-google-play>