Natural Intent: The Use and Misuse of Intents in Android Applications

by

William Gibbs

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2020 by the
Graduate Supervisory Committee:

Adam Doupé, Chair
Yan Shoshitaishvilli
Ruoyu Wang

ARIZONA STATE UNIVERSITY

May 2020

ABSTRACT

The Java programing language was implemented in such a way as to limit the amount of possible ways that a program written in Java could be exploited. Unfortunately, all of the protections and safeguards put in place for Java can be circumvented if a program created in Java utilizes internal or external libraries that were created in a separate, insecure language such as C or C++. A secure Java program can then be made insecure and susceptible to even classic vulnerabilities such as stack overflows, string format attacks, and heap overflows and corruption. Through the internal or external libraries included in the Java program, an attacker could potentially hijack the execution flow of the program. Once the Attacker has control of where and how the program executes, the attacker can spread their influence to the rest of the system.

However, since these classic vulnerabilities are known weaknesses, special types of protections have been added to the compilers which create the executable code and the systems that run them. The most common forms of protection include Address Space Layout Randomization (ASLR), Non-eXecutable stack (NX Stack), and stack cookies or canaries. Of course, these protections and their implementations vary depending on the system of. I intend to look specifically at the Android operating system which is used in the daily lives of a significant portion of the planet. Most Android applications execute in a Java context and leave little room for exploitability, however, there are also many applications which utilize external libraries to handle more computationally intensive tasks.

The goal of this thesis is to take a closer look at such applications and the protections surrounding them, especially how the default system protections as mentioned above are implemented and applied to the vulnerable external libraries. However, this is only half of the problem. The attacker must get their payload inside of the application in the first place. Since it is necessary to understand how this is occurring,

I will also be exploring how the Android operating system gives outside information to applications and how developers have chosen to use that information.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Fuzzing is a methodology for creating a large variety of unexpected inputs that a program may not handle well. The purpose of fuzzing is to find the corner cases that were missed by the well intentioned programmers. However, this type of bug finding can be used for exploitive purposes as well. For programmers, in order to find these corner cases, a fuzzer would need to be seeded with inputs that the program would expect and would ideally cover all execution paths that the program could have. However, malicious fuzzing does not take as much heed in fuzzing a program. Having a seed that executes as many code blocks in the program as possible is definitely ideal, but not entirely necessary. The fuzzer does not provide input like a regular user would, all values are equal in the eyes of the fuzzer. This means that the fuzzer will use all values from 0 to 255 for a byte of input instead of just printable characters that users generally use. On top of this, the goal of malicious fuzzing is generally to find a single exploitable bug, whereas a developer might wish to find as many corner cases as possible. With this in mind, fuzzing can easily be scaled to attack a large variety of programs until a single bug is found which the attacker can then work on exploiting.

However, a fuzzer can have several different hurdles that it needs to overcome before it can begin fuzzing. After all, if the fuzzer cannot even give input to the program, then how is it supposed to be able to fuzz. In the case of the Android operating system, fuzzing android applications is exceedingly difficult. It would be possible to fuzz the application in such a way as to replicate a user's interaction, however, this provides little to no benefit for attackers. If an attacker wanted to launch an attack at any reasonable scale, which is true in most cases, then going to

individually insecure devices would significantly slow the progress of their attacks. Thus, attacking as a user is not the ideal scenario.

The attacker must get data into the application from the outside without physical access to launch any attack of scale. This is where the intent system in Android comes into play. Android intents allow applications to communicate with one another. This is how a link in one app knows to launch the link with an available browser on the system. An intent is either targeted to launch a single app, or the intent asks the system if there are any applications available that can handle the type of intent it broadcasts and the user can select the appropriate application. Through this intent communication system, attackers can interact with vulnerable applications on the device. Now the only issue is that the intent must navigate through the Java program space and find itself inside a vulnerable external library.

The purpose of this paper is to determine the scale of vulnerable applications that exist. Saying that there are many applications available for download for the Android operating system would be an understatement. We will take a subset of the current applications available for download and explore their exploitability and how they utilize the protections available to them.

Chapter 2

BACKGROUND

2.1 Android Application Stack

Under the hood, the Android operating system uses the Linux kernel as it's base [3]. The Linux kernel is used because it is free to use and is well established. This allows for many different companies to develop their own flavors of Android without worrying about paying for licenses for the operating system. For application purposes though, the Linux kernel is almost inconsequential, this is because the runtime environment for applications is actually the Dalvik Virtual Machine (DVM) which is a virtual machine for executing Java code. Virtual machines also have the benefit of acting as a sandbox. This gives applications a certain amount of security as their execution context cannot be hijacked by other processes except in the case of a severe kernel vulnerability.

The core Android functions can be categorized and separated and abstracted into several different layers [27]. The Linux kernel forms the foundation of Android and handles low level functionality. As mentioned before, Applications are executed in the Dalvik Virtual Machine with the core C libraries needed to function existing just outside of it. Inside of applications exist *Native Libraries* which are the C and C++ libraries that a developer has added to an Application. However, to save execution time as well as memory, the Android operating system shares necessary core libraries with all processes.

Above the runtime setup is the Application Framework which contains an individual Application's life cycle. It is at this level that intent communication occurs. To expand more on the execution environment of the Android operating system, the

Figure 2.1: Framework for Android Applications

core library that is shared among all libraries is the *Bionic libc*. This is a special minimized version of libc, it does not contain all of the functionality of a normal libc, it is highly optimized for size and performance in embedded environments. It is in this library that the *Stack Canary* protection is offered.

To further isolate the Application, each DVM for each application is run as a separate user on the device [15]. This ensures separate applications cannot interfere with another process's execution environment. In order to do so, a privilege escalation vulnerability would need to exist.

However, every process spinning up their own DVM and instance of *Bionic libc* would make the start of every application much slower and consume much more memory. To work around this, on boot, the Android operating system initializes the Zygote [13]. The Zygote is a single process which contains a copy of *Bionic libc* and a DVM. Any new application that starts forks this Zygote process and copies the DVM. This saves time because the copy of the DVM is already initialized and by sharing the location of *Bionic libc* Android can avoid having to duplicate memory efforts of

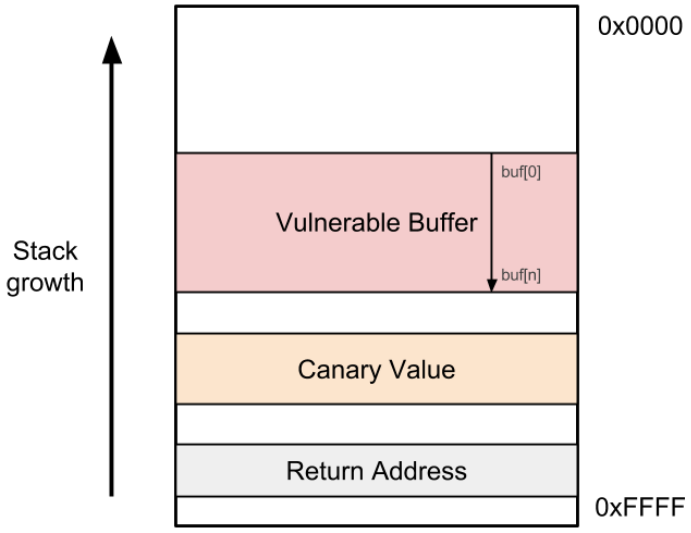

Figure 2.2: Stack Canary Use Case

storing the relatively large library despite it's optimizations.

Finally, when Android wants to start a new application is sends an *Intent* [9] to the Application. The Intent tells the Application which activity it should start from to ensure the user is loaded into the proper screen.

2.2 System Protections

2.2.1 Stack Canaries

The stack contains a program's local variables that are used during execution [10]. This includes everything from pointers, to integers, to arrays. Arrays being stored on the stack are especially important to attacks as they often provided ways for their control flow to be hijacked. The arrays can hijack the control flow by writing out of their allowed bounds, this can give attackers the ability to overwrite the saved instruction pointer which is often stored on the stack [10, 20].

To thwart this type of exploitation, stack canaries were introduced [5]. Figure 2.2 shows an example of what a stack canary looks like in practice. It guards the saved

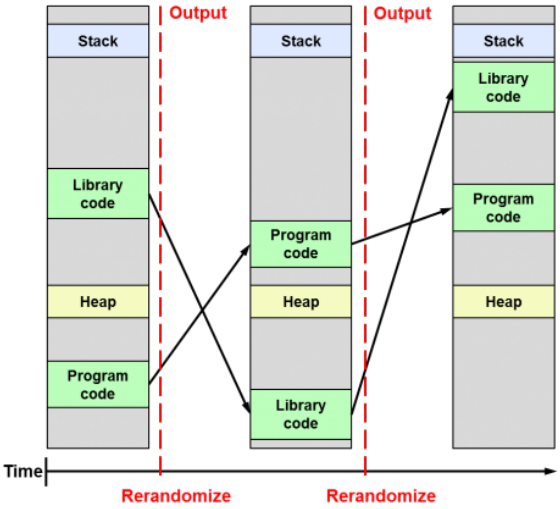

Figure 2.3: ASLR Library Randomization Upon Execution

instruction pointer using a unique value that is either 32-bit or 64-bit depending on the architecture. This value changes with every run of the program so it is difficult to circumvent if there is not something to leak data from the stack. If the program sees that the canary has changed on return to the parent function, then the program will immediately terminate.

2.2.2 ASLR

Address Space Layout Randomization (ASLR) makes guessing addresses to jump to much more difficult for attackers by randomizing the addresses that libraries are loaded at [16].

Every execution of a program uses a new randomized address much like the value of the stack canary. However, this poses the challenge for programs to know where the libraries they need to access are stored at. The Global Offset Table (GOT) is section of data stored in the program which acts as a lookup table for whatever external library function that the program needs to use. The attacker will run into the same problem as before, there is no way to guess the value without some sort of information

leak. The attacker is, however, able to hijack the execution flow of the program by overwriting entries in the GOT since the values are only loaded upon the first call to the external function.

2.2.3 Fortify Source

Fortify Source, unlike stack canaries, rely on the compiler for enhanced fortification. Fortify Source tries to limit functions that could potential cause out of bounds writes much like stack canaries. However, Fortify Source takes a single function such as *gets* and replaces it with a safer alternative like *fgets* which bounds the amount able to be read into a buffer. However, Fortify Source is not enabled by default which makes the amount of libraries that contain it small in number.

2.2.4 RELRO

RElocation Read-Only (RELRO) protects the GOT from the above mentioned hijacking attack2.2.2 which requires the attacker to overwrite entries in the GOT. RELRO will, at runtime, resolve all used external library functions to their correct addresses and remap the GOT to be read-only. After that, the only thing attackers can do is either leak data from the table, or have some other vulnerability that allows the memory to be remapped as writable.

2.2.5 Non-eXecutable

Non-eXecutable (NX), also known as Data Execution Prevention, stops any and all execution of code on the stack. Without this protection, attackers could write binary code to the stack and then jump to it with some form of control flow hijack which would allow them to execute arbitrary code in the process which is very dangerous. NX makes sure that the stack is only mapped to be readable and writable, but

not executable. The program code that must be executed is mapped to be only readable and executable making sure that attackers cannot write their own code in that memory space either.

Chapter 3

ANDROID APPLICATION SECURITY

3.1 Application Vulnerabilities

As mentioned in Section 2.1 interacting with and exfiltrating data from another application is difficult due to the sandboxed context in which each application runs in. The protections and vulnerabilities mentioned in Section 2 are for C and C++ contexts only, Java programs do not suffer from the same types of vulnerabilities due to the bounds checking among other protections that exist for Java programs. For example, out of bounds writes will cause a Java program to immediately halt execution preventing any sort of control flow hijacking.

Java applications may be fairly secure, but the same cannot be said about the C and C++ libraries that they are used in conjunction with, these libraries are commonly referred to as *native libraries*.

However, why should these Android applications use C and C++ libraries if Java code is so secure? The answer lies in the execution speed and memory efficiency of Java code or lack thereof [2]. C and C++ require manual memory management which is more cumbersome and error prone, provides a much greater resource control ability compared to the lazy garbage collector of Java. However, poor memory management in C and C++ is much worse than the garbage collector that Java has. C and C++ can also be run natively on Linux whereas Java needs a virtual machine, the DVM, since Java compiles itself into Java byte code which the DVM must then execute by turning the byte code into instructions that the processor will understand. Because both languages have their advantages, the JNI bridge exists in Java to allow for the execution of C and C++ libraries in functions.

Search Type	Physical Amount	All APKs Prevalence	Native Code Prevalence
APKs without Native Code	1753541	80.7%	0.0%
APKs with Native Code	419561	19.3%	100.0%
APKs with Full Canaries	165586	7.3%	39.5%
APKs with Partial Canaries	188469	8.7%	44.9%
APKs with No Canaries	65506	3.0%	15.6%

Table 3.1: Sample Applications Gathered from the Google Play Store

Do all developers care about this? Not every application can benefit from mixing Java and native libraries. To understand the prevalence of native libraries in Android applications, a largest sample of applications were taken from the Google Play store to be investigated.

To break down table 3.1 let's first take a look at the amount of applications with native libraries. Almost 20% of the applications sampled use native libraries in some form or another. The core *Bionic libc* that is shared with every Android application has very little surface area for attacks if there are no native libraries present as there is no direct way to access the libc without them. Though using native libraries can increase the potential vulnerability of an application, it does not by any means make the application exploitable.

The *full canaries* categorization for Table 3.1 are for applications that use native libraries and all of the native libraries that it uses contain stack canaries. The *partial canaries* category similarly includes applications that use native libraries, however only a subset of the libraries contain the stack canary protection. This can be attributed to the way that stack canaries can be included in an application. At compile time, the option to include canaries in only the functions which may pose a risk can

be selected or all functions can be given canaries. If a library is deemed to not be at risk, then it is possible for no canaries to exist. However, it is also possible that native libraries from different sources were used. Using libraries from different sources allows for potentially different compilation options among the Libraries which could be the cause of the partial presence of canaries. The *no canaries* category are applications that have native libraries, but those libraries do not have stack canaries compiled in them.

However, these numbers may be slightly misleading, of the categories, the *partial canaries* is the easiest to satisfy. If an application has a large amount of native libraries, but only one or a small amount contain no canaries, then the condition to be put in this category is satisfied. Likewise, an application that has a large amount of native libraries, but there exists only one or a few libraries that have canaries, then the condition is also satisfied. The most interesting number present is the 15.6% of native libraries that contain no canaries. These native libraries are prime targets for exploitation.

3.2 Additional Security Mechanisms

Protection Type	Libraries Protected	Percent Total
Stack Canary	216,919	79.4%
NX	255,279	93.5%
Fortify Source	171,242	62.7%
RELRO	139,682	48.8%

Table 3.2: 273,083 Libraries from 64,161 APKs Were Sampled for their Countermeasures

Apart from the analysis of the applications as a whole, it's important to look at

the libraries as individuals. The native libraries can also be identified according to the protections afforded to them to determine their exploitability. All of the sampled native libraries had their protection information probed. This includes Fortify Source, RELRO, and NX, but not ASLR. ASLR is a protection at the kernel level, not the application level so it is dependent on the device, not the application.

These protections were gathered from more than 64 thousand Android applications which contained almost 300 thousand native libraries. Of all the libraries gathered, slightly more than 20% did not have canaries. As discussed in the previous section, there are multiple potential causes. However, more than 90% have NX enabled, making almost all native libraries secure against attacks involving shellcode. However, this does leave the question of how vulnerable the non-protected libraries are. Less than 10% seems like a small amount, but that is in fact almost 20 thousand libraries. That's a fairly large surface area. Less than 50% of libraries implemented RELRO which is surprising. That leaves half of all libraries available for exploitation via overwrites to the GOT. The amount of applications that enabled Fortify source is a little more than 60% which is actually fairly surprising as Fortify source is not a default option. But this begs the question, why would a nono-default option be more prevalent than a default option like RELRO.

The percent for each type of protection is wildly different except for the relatively close values for RELRO and Fortify source. One would expect that if a library would have protections, it would have all, or at least most of the offered protections, but the wild variation of protections is certainly interesting. However, these disparate numbers can be partially explained by the occurrence of multiple of the same libraries in different applications. Table 3.3 shows that among all of the applications that were sampled, there is a high value for the most used *SHA1*. *SHA1* guarantees that all data in a blob or document is the exact same, otherwise a different hash will be produced.

This means that many different applications share the same library, and this is not a rare occurrence. Sharing the same library among many applications makes that library a great attack target. If that single library can be exploited, it is likely that all of the other applications using that library can be exploited as well.

Criterion	Library Name	Total
Most Unique ROP Gadgets	libTennisApplication.so	692,753 gadgets
Largest Size	lib1cem.so	57.8 Mb
Most Used Name	libstlport_shared.so	1402 libraries
Most Used SHA1	libysshared.so	5043 libraries

Table 3.3: Native Library Data

Chapter 4

APPLICATION INVESTIGATION

4.1 Taint Tracking

With all the data gathered about the protection on the native libraries, it should be easy to figure out how to exploit them. However, that is not quite the case. The traditional methods for binary exploitation must be adapted to work on an Android operating system compared to it's cousin Linux. A standard binary in Linux will take input either, from a file, from the commandline via standard input, or from a socket that you can write to. These methods do not translate well to an Android context. Android applications do not have a standard in that is easily written to. Nor do Android devices readily expose ports that anyone can write to. It may be possible to get a user to download a file and then later use that unwittingly only to have their device exploited, but that is fairly farfetched. By far the easiest way to get information into an application is with Android's own intent system. Getting a user to download and use a malicious application is not an impossibility. Scams like this happen every day. Also, malicious applications cannot directly write files in another process's space and it cannot directly interact with the process, only through intents can any exchange take place.

Intents can be used either internally or externally. Internal intents are used to navigate between activities, much like a navigation bar in a website is used to navigate between pages on the same website. External or broadcast intents are received by other applications and contain data that they can act on. However, this type of intent requires the user to select which application the intent can go to given a list of possible recipients. We are also limited to only applications that are willing to

accept our intent. A completely vulnerable application could exist, but without it being willing to accept a broadcast intent, we are stuck. We can pre-determine which applications will be willing recipients by taking a look at their Android manifest files which explicitly controls the types of intents the application is looking for.

However, even if we can get an intent into the application, how do we know where and when the intent data is used? This is where Flowdroid, a *taint tracker* comes in. Taint tracking, as a broad concept, tracks data as it flows through a program [8]. This can allow you to see how your data is being used by the program. However, there are two types of taint tracking, static and dynamic. Dynamic taint trackers rely on user interaction and track how the data is used as the program executes. Static taint tracking takes in whatever type of file it is configured for and deduces how the data would flow to whichever point was marked as the stop, or sink. Flowdroid is a static taint tracker. Static taint tracking does suffer from the lack of certainly that dynamic taint tracking has, as it cannot determine with 100% accuracy the content or type of data flowing through the program. However, it is much faster and more comprehensive than dynamic taint tracking.

For example, Figure 4.1, we have marked which function is the *source* of the data that we wish to track and which function is the end point or *sink*. It can plainly be seen that the data transforms as it moves through functions, but despite the change in data type, it is tracked regardless. This can pose a problem, which is, as our data moves through a program, it can transform and be manipulated into no longer being useful for exploitive purposes.

Determining which function should be your source and sink are paramount when taint tracking. Incorrect selections can return results that are dead ends or entirely useless. So how does this taint tracking help in an Android context? We can mark the functions that turn the intent into useful data as sources, and then mark all of

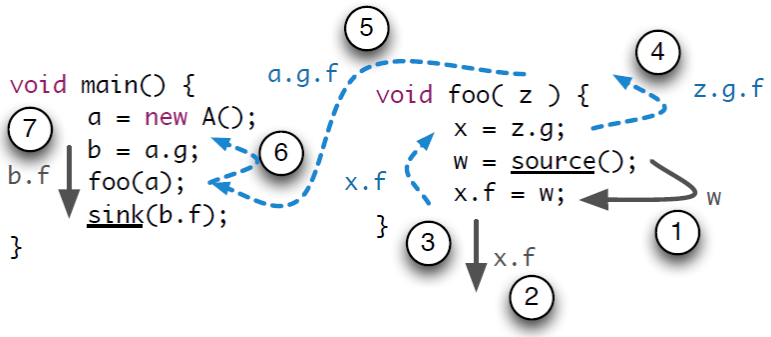

Figure 4.1: Taint Tracking Flow.

the exposed native library functions as the sinks. Since we only care about how to exploit applications from intents, everything else is entirely irrelevant. As mentioned above, the contents of the data are not guaranteed to persist from the intake of the intent all the way until the native library function marked as the sink. Since there are a vast amount of available applications and limited time, I chose to limit my sinks to native functions that have strings as parameters. This means that overflows may be possible with a long enough string. This is exemplified by Figure 4.2.

```
[main] INFO soot.jimple.infoflow.Infoflow -
The sink specialinvoke
$r0.<org.appcelerator.kroll.runtime.v8.V8Runtime:
        void nativeRunModule(
        java.lang.String,java.lang.String,
        org.appcelerator.kroll.KrollProxySupport)>
```

Figure 4.2: Flowdroid Flow

4.2 Application Instrumentation

Once the taint tracking analysis by Flowdroid has finished, it's time to ship it off to Friday. Frida is a tool which injects a full V8 Javascript engine into Android

applications. This Javascript engine can fully interact with the target program and instrument it [12]. We can take the analysis returned by Flowdroid combined with data from the Android Manifest of an application, to form an intent and identify which native function the intent may end up at. Through Frida we can then instrument the native function to return what data was passed to it and even execute the native function with whatever parameters we deem necessary. Figure 4.3 shows and example of the instrumentation through frida. This function is causing the onResume function to be called and inserting a crafted intent into the program's defined onNewIntent function.

```
Java.perform(function x() {
  var my_class = Java.use("org.qtproject.qt5.android.bindings.StartActivity");
  console.log("HOOK: " + my_class);

  my_class.onResume.overload().implementation = function () {
    this.onNewIntent(this.getIntent().getDataString())
  };
});
```

Figure 4.3: Frida Injection Script

Chapter 5

INTENT ANALYSIS

Further investigation of the Intents and how they are used can be difficult in Android. Combining the output generated by Frida and Flowdroid provides an impressive and comprehensive view of how Intents are used throughout Android applications. It was first investigate how many Applications that could be targeted using outside intents. This pool of applications was narrowed from the already smaller pool that include native libraries since some applications do not accept intents from the outside. Furthermore, using the analysis provided by Flowdroid, it was found that 64% of applications that contain native libraries have a path from an outside intent into the native library via a native function of some sort.

	Total	Unique Content	Unique Name
APKs	27,581	27,581	27,581
Activities	90,613	27,294	1,945
Methods	1,840	1,840	1,676

Table 5.1: 30,000 APKs were analyzed to determine their shared properties.

A sample of 30,000 APKs were analyzed to determine how intents were used throughout their respective applications. From these 30,000 applications, as per Table 5.1, there were more than 90,000 activities that had flows from intent to native code. However, these 90,000 activities can be effectively reduced to little more than 27,000. The cause of this reduction is the removal of default Activities. That is, activities that are pre-included in Android applications such as the Android.util.Log activity.

Table 5.2 is a list of the most common activities that have native sinks from intent

Name	Count
android.util.Log	13,565
android.os.Bundle	12,695
java.io.OutputStream	7,717
com.unity3d.player.UnityPlayer	6,349
android.content.SharedPreferences$Editor	5,838
android.content.ContentResolver	5,400
java.net.URL	5,107
android.os.Handler	3,490
android.content.Context	3,341
android.content.Intent	1,934

Table 5.2: Top 10 Most Common Activities Among Android Applications

sources. It is clear that with exception to one activity, the most common activities are all default Android and Java activities. These are not of interest because they offer little possibility from exploitation. However, when we remove all of the default activities, we get a list of the most common non-default activities. Table 5.3 shows just this. We can see that there is a great mix of non-default activities that have native sinks. Also of interest is the amount of times they reappear in other applications. There is a sharp difference between first and second place, but this list makes up more than 15,000 applications. Since these activities are tied to their native functions, it is very likely that a single exploit generated for a native function contained in a single activity will be able to exploit the same activity in other Android applications.

Table 5.4 is a list of the most common native methods found in activities. Looking at the table, most of the methods are custom methods instead of default methods

Name	Count
com.unity3d.player.UnityPlayer	5,349
com.facebook.LoginActivity	1,757
com.unity3d.player.UnityPlayerProxyActivity	1,732
com.facebook.CustomTabActivity	1,352
org.apache.http.message.BasicNameValuePair	1,254
org.appcelerator.kroll.runtime.v8.V8Runtime	1,047
org.appcelerator.kroll.runtime.v8.V8Object	1,047
com.ansca.corona.purchasing.storeactivity	885
com.facebook.react.bridge.writableNativeMap	881
org.apache.http.client.httpclient	760

Table 5.3: Top 10 Most Common Non-Default Activities Among Android Applications

included in default libraries. The default methods include putString, Write, and putInt. Besides these three methods, all of the other methods are from developer created native libraries. In fact, almost all of the most common methods listed here are prefixed with the word "native". This is due to the naming convention the developer used as all of these methods come from the same library or same set of libraries. This is very encouraging, most of the common methods, not only are from the same library, but they have almost the same frequency of occurrence. In addition, some of these native methods take a java.lang.String as the parameter, which is exactly what we would hope would happen since strings are one of the most common ways to cause overflows in a program. The method "D" is the only other method that is unique, but is not part of the same library as the methods prefixed by "native". "D" is actually an obfuscated method name of a method. However, for the

purposes of this experiment, all obfuscated methods with the same parameter inputs were grouped together. For the purposes of instrumentation, the method name and the parameters are the only thing that matters, what the method itself does is not relevant.

Name	Count	Parameters
putString	15,531	java.lang.String, java.lang.String
D	9,236	java.lang.String, java.lang.String
Write	6,607	byte
putInt	6,279	java.lang.String, int
nativeInitWWW	6,243	java.lang.Class
nativeSetLocationStatus	6,229	int
nativeforwardEventsToDalvik	6,211	boolean
nativeFile	6,093	java.lang.String
nativePause	5,785	None
nativeDone	5,783	None
nativeSoftInputClosed	5,783	None
nativeSetInputCanceled	5,783	boolean
nativeSetInputString	5,783	java.lang.String

Table 5.4: Top 13 Most Common Native Methods Among Android Application Activities

Chapter 6

APPLICATION EXPLOITATION

Through using both Frida and Flowdroid, a potential vulnerability could be unearthed, but it has not been mentioned how the exploitation method would differ from a normal attack. This begins with having a malicious application installed on the device which is loaded with the ability to send crafted intents to specified applications. Then the attacker must bypass the protections mentioned before. Fortify Source is an unfortunate roadblock that cannot be overcome, however, since any function that was fortified would not be vulnerable, all Fortify Source does is limit the amount of vulnerable applications, it would not interfere with exploitation directly. However, the same cannot be said about ASLR. Since ASLR is system dependent, it's safe to say that most modern systems implement it. It would be incredibly difficult to get any kind of leak with an intent as it would require data to be sent somewhere accessible by the attacker. However, this does not mean that all hope is lost, in fact, far from it.

As mentioned in Section 2.1 it was mentioned that part of the start of every application is forking the Zygote and making a copy of the Zygote's DVM. This is where a fatal error of the Android operating system presents itself. The *Bionic libc* is never re-initialized. Therefore, it's position and address are always static (per boot of the device). This makes it incredibly trivial to obtain addresses to jump to in libc. All the malicious program has to do is take note of where the functions it wants from it's own libc is and use the exact same addresses when exploiting the target application.

ASLR is now inconsequential as a security feature in Android systems, but we still have to overcome the stack canary which is so prevalent as mentioned by Figure

3.1 and Figure 3.2. Bruteforcing the canary value is inadvisable as the amount of time it would take, especially on an embedded system like Android, would make large scale attacks difficult. However, that would be true if the canary value were not stagnant. In normal processes the canary is taken from libc upon execution. Libc generates and stores the canary when it is linked to whichever program is using it. This canary value is sufficiently random as it is sourced from the device's pseudo-random number generator. There are several attacks that exist that might allow the canary's randomness to be limited, however, that is wholely unnecessary. The exact same vulnerability which removes ASLR as a roadblock will also remove the canary as an issue. Since libc is never re-initialized, the canary value it holds stays constant. If the attack vector would happen to be outside the device, the canary can easily be bruteforced. Instead of bruteforcing the canary all at once, a single byte overwrite at a time and checking if the application crashed is enough to bruteforce the canary [22]. This significantly reduces the attempts required for a bruteforce. Since the canary is dependent on libc which is never re-initialized, the application can crash as many times as it needs to as long as the phone itself is not rebooted. There is, in fact, an even easier solution to this issue

```
extern "C"
JNIEXPORT jstring JNICALL
Java_com_android_overflow_canary(JNIEnv *env, jstring mystr, jobject /* this */) {
uint32_t canary;

#if defined(__i386__)
asm("movl 0xd8(%%esp),%0;" : "=r"(canary) :);
#endif

char canaryBuffer[16] = {0};
sprintf(canaryBuffer, "0x%" PRIx32, canary);
return env->NewStringUTF(canaryBuffer);;
}
```

Figure 6.1: Reading the Canary in a Native Context

```
7db:   8b 94 24 d8 00 00 00    mov    edx,DWORD PTR [esp+0xd8]
7e2:   39 d3                   cmp    ebx,edx
7e4:   89 44 24 18             mov    DWORD PTR [esp+0x18],eax
7e8:   0f 85 0c 00 00 00       jne    7fa
7f9:   c3                      ret
7fa:   8b 5c 24 64             mov    ebx,DWORD PTR [esp+0x64]
7fe:   e8 9d fc ff ff          call   4a0 <__stack_chk_fail@plt>
```

Figure 6.2: Canaries in x86 Systems on Android

Because the canary value is stagnant for all applications, malicious application can use it's own canary value as the canary value for the target application and safely overwrite the foreign canary, something that normally would never be possible.

The paper SSPFA describes a very similar technique as well as a proposition for the solution to this static canary value [17]. However, no such solution has been implemented on widely available Android operating systems.

Figure 6.1 gives an example of exactly how a malicious application can read it's own canary. This canary value detection is unfortunately not automated as the position of the canary can shift depending on the variables in the function and the parameters passed to the compiler. This is, of course, native code written in a small native library. To find the canary offset, one must first disassemble the compiled native library and read the assembly. Figure 6.2 shows what the end of a disassembled function may look like. In the disassembled function we can see a call to __stack_chk_fail which is libc's function for handling overwritten canaries. This code is shows in the x86 architecture for ease of reading, but this applies to all architectures.

Chapter 7

FUTURE WORK

We have discussed how certain Android protections fail to function properly, as well as how to exploit those failures. We have also discussed how to discover routes that lead from outside of the application down into the application's native libraries and how to verify the integrity of the input being fed into the native functions to determine their exploitability. However, this was only discussed relative to the Android system and the protections that it provides specifically for stack-based overflows. There is further work to be done in exploring how these techniques can be manipulated and applied to analysis other systems as well as other classes of vulnerabilities.

7.1 Vulnerability Analysis

One major area left that this thesis did not explore are other types of memory corruption bugs that can be leveraged for exploitation [29]. Large strings were sent in as intent data in an attempt to cause a stack overflow and trigger the canary protections on the Android device. This was done as stack overflows are fairly easy to trigger compared to other types of control-flow hijacking that may require much more program specific input. As such, only functions that could cause these types of vulnerabilities were look at as potentially vulnerable applications. However, expanding the search to include heap based attacks and overflows would increase the attack surface. Heap based attacks are more difficult to detect on a surface level as the overflow or overwrite may not directly cause the program to crash. More sophisticated taint analysis on programs could reveal ways to trigger common heap vulnerabilities such as use after free which are useful in hijacking the control flow of a program [30]. Introducing heap based attacks would significantly increase the amount of applica-

tions that could be exploited. To fully explore this path, the protections afforded to native library heaps must be investigated. Android devices before version 5.1 used the dlmalloc implementation of malloc whereas newer devices have switched to the jemalloc implementation [31]. Android's jemalloc suffers from the ability to create large chunks within a small address space making chunks somewhat predictable. This leaves rooms for tools like shadow which analyses the heap in an Android application looking for a vulnerability [26]. The Canary protections covered in this thesis only guard against stack-based attacks. Adding this new class of vulnerability would significantly increase the amount of potentially vulnerable applications. In turn, the amount of exploitable applications would increase as well. Conducting this research would broaden our understanding of the Android system and showcase just how vulnerable the average Android device is. It would give exposure to previously unknown flaws that could prove disastrous if left unchecked.

7.2 Systems Applications

The body of this work focuses only on the Android operating system and how given an application, various analyses can be applied in an attempt to find an exploitable path into the application. The core of these analyses can be applied to almost any system, not simply Android. The closest relative would be the iPhone ecosystem, which contains similar concepts making the application of techniques require less effort to execute [11]. However, what would be interesting is to apply this type of analysis to third-party libraries in different Linux applications. It would be much easier working on a standard Linux platform than Android as it would not have any of the extra hurdles to clear that makes analyzing Android applications tedious. Android can only be fuzzed with limited information of how input reaches native libraries, there have been several tools developed to fuzz Android application intents

but none are easy to use or readily scalable [23, 28]. If a vulnerable function is known in a 3rd party library, the vulnerable function could be defined as the sink to eliminate any programs that use the library, yet fail to send any data to the function itself. This combined with symbolic execution may allow for a greater amount of vulnerabilities to be discovered as it could be determined if data making it to the potentially vulnerable function would be of any use for exploitation [25]. Expanding this analysis to other platforms may reveal correlations that are platform specific which may not present themselves in small data sets. Running a multi-platform analysis could potentially reveal common problems or vulnerabilities that are platform agnostic or partially agnostic. Discovering and determining the feasibility of these attacks is valuable. The more knowledgeable we are on the types of vulnerabilities that exist, the more we can limit and protect against them.

7.3 Input Sources

In Android systems, intents are the main method used to communicate between applications, but they are not the only way to get information into other applications. There are many instances of applications either reading files from the system, or retrieving data by calling out to external apis or databases. An in-depth analysis of many applications and their different sources of input. It could be interesting if a large amount of applications read from editable files without user intervention. Since each process is sandboxed [18], the files they write are typically unavailable for outside applications, unless written in a place that other applications can access such as the downloads folder in many systems. Reading from files that are editable by other applications opens a whole host of other potential vulnerabilities. Expanding the sources from intents to other forms such as files or urls increases the attack surface significantly as well as the likelihood of finding an exploitable vulnerability. Much

like studying heap vulnerabilities in an Android context, increasing the sources that a vulnerability can come from increases the amount of devices and applications that can be exploited. The value in this approach is creating a tool or adding functionality to an existing tool which makes detecting these routes and determining their exploitability easier. Limiting human effort as much as possible can only increase the coverage of devices and applications that can be tested, thus creating an overall safer and protected ecosystem.

Chapter 8

RELATED WORK

As described in this thesis, canaries in Android native libraries are entirely broken. However, there are others that are working towards a solution for the lack of proper initialization for canary values by using the DynaGuard technique which changes how and when the canary values are stored [17].

Recent versions of Android have implemented the use of a shadow stack to strengthen native libraries against stack-based control flow hijacking. However, shadow stacks require some amount of overhead on top of the application which can have consequences in an embedded environment [4, 6, 7]. This thesis has contributed to exposing how increasing performance without regard for security can create huge vulnerabilities in a system. Decreasing the overhead of new security implementations such as the shadow stack is key in having them adopted by embedded systems [14, 21].

VAnDroid is a tool which was developed to analyze the potential vulnerability of an Android application [19]. However, unlike the approach in this thesis, VAnDroid is not equipped to detect data flowing into native libraries where control flow hijacking can occur. The purpose of VAnDroid is to detect data injection via intents as well as data exfiltration. VAnDroid could definitely be used in conjunction with the approach described in this thesis to further limit the pool of exploitable applications. Likewise, the tool Blue Seal can automatically determine flow permissions which can further reduce the pool of applications making the discovery of a truly vulnerable application more likely [24].

LibRARIAN is an approach developed to fingerprint vulnerable and out of date libraries used as native libraries in Android applications [1]. LibRARIAN can discover if an Application houses a vulnerable library, but it cannot determine if there is any

way to exploit said library. This approach is truly helpful in identifying highly likely targets for exploitation, but since it only identifies end-day vulnerabilities in libraries, it excludes all unknown or undocumented third party libraries, leaving out a large group of potentially vulnerable applications.

Chapter 9

CONCLUSION

Some of the protections that exist to defend against classic vulnerabilities such as stack overflows can be completely negated if an attacker is able to place a malicious application on a target device. There are numerous protections that have been employed on the Android system, such as DEP, ASLR, RELRO, and Fortify Source to act as obstacles for certain types of vulnerabilities. However, RELRO and Fortify Source are only applied to half of all libraries library where as DEP is applied to almost every library. The trademark memory and speed maximization strategy of the Android Operating System creates a flaw within these protections. Since processes fork from the Zygote process, stack canaries and ASLR addresses are retained throughout multiple processes until a device restart occurs. Using flowdroid for static taint analysis and Frida for instrumenting Android applications, valid paths can be found that carry data from outside 3rd party applications into the native libraries of the vulnerable application. It has also been shown that many Android applications contain native libraries that are commonly used among other applications. Therefore, by discovering a single vulnerability in a widely used native library, one can exploit many different applications giving the attacker a much larger surface area for their attack. Since the canary and ASLR values are static across multiple applications, it is trivial to create an application that can harvest these values which can be used alter to overwrite other applications stack and jump to known locations making exploitation less tedious.

REFERENCES

[1] Almanee, S., M. Payer and J. Garcia, "Too Quiet in the Library: A Study of Native Third-Party Libraries in Android", arXiv e-prints p. arXiv:1911.09716 (2019).

[2] Blackburn, S. M., P. Cheng and K. S. McKinley, "Oil and water? high performance garbage collection in java with mmtk", in "Proceedings. 26th International Conference on Software Engineering", pp. 137–146 (IEEE, 2004).

[3] Bovet, D. P. and M. Cesati, *Understanding the Linux Kernel: from I/O ports to process management* (" O'Reilly Media, Inc.", 2005).

[4] Burow, N., X. Zhang and M. Payer, "Shining Light On Shadow Stacks", arXiv e-prints p. arXiv:1811.03165 (2018).

[5] Cowan, C., C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.", in "USENIX Security Symposium", vol. 98, pp. 63–78 (San Antonio, TX, 1998).

[6] Dang, T., P. Maniatis and D. Wagner, "The performance cost of shadow stacks and stack canaries", ASIACCS 2015 - Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security pp. 555–566 (2015).

[7] Dang, T. H., P. Maniatis and D. Wagner, "The performance cost of shadow stacks and stack canaries", in "Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security", ASIA CCS 15, p. 555566 (Association for Computing Machinery, New York, NY, USA, 2015), URL `https://doi.org/10.1145/2714576.2714635`.

[8] Edward J. Schwartz, D. B., Thanassis Avgerinos, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)", `https://users.ece.cmu.edu/~aavgerin/papers/Oakland10.pdf` (2015).

[9] Elenkov, N., *Android security internals: An in-depth guide to Android's security architecture* (No Starch Press, 2014).

[10] Irvine, K. R. *et al.*, *Assembly language for x86 processors* (2015).

[11] Kaczmarek, S., B. Lees and G. Bennett, *A Swift iPhone App*, pp. 299–324 (Apress, Berkeley, CA, 2019), URL `https://doi.org/10.1007/978-1-4842-4868-3_14`.

[12] Karami, M., M. Elsabagh, P. Najafiborazjani and A. Stavrou, "Behavioral analysis of android applications using automated instrumentation", in "2013 IEEE Seventh International Conference on Software Security and Reliability Companion", pp. 182–187 (IEEE, 2013).

[13] Latifa, E. and E. K. M. Ahmed, "Android: Deep look into dalvik vm", in "2015 5th World Congress on Information and Communication Technologies (WICT)", pp. 35–40 (2015).

[14] Li, J., L. Chen, Q. Xu, L. Tian, G. Shi, K. Chen and D. Meng, "Zipper Stack: Shadow Stacks Without Shadow", arXiv e-prints p. arXiv:1902.00888 (2019).

[15] Lorch, J. R. and A. J. Smith, "Proceedings of the 2nd annual international conference on mobile computing and networking", http://projects.webappsec.org/w/page/13246926/Format%20String (1996).

[16] Marco-Gisbert, H. and I. Ripoll, "On the effectiveness of full-aslr on 64-bit linux", in "Proceedings of the In-Depth Security Conference", (2014).

[17] Marco-Gisbert, H. and I. Ripoll-Ripoll, "Sspfa: effective stack smashing protection for android os", https://doi.org/10.1007/s10207-018-00425-8 (2019).

[18] Neuner, S., V. van der Veen, M. Lindorfer, M. Huber, G. Merzdovnik, M. Mulazzani and E. Weippl, "Enter Sandbox: Android Sandbox Comparison", arXiv e-prints p. arXiv:1410.7749 (2014).

[19] Nirumand, A., B. Zamani and B. Ladani, "Vandroid: A framework for vulnerability analysis of android applications using a model-driven reverse engineering technique", Software: Practice and Experience (2018).

[20] One, A., "Smashing the stack for fun and profit (1996)", See http://www. phrack. org/show. php (2007).

[21] Park, Y.-J. and G. Lee, "Repairing return address stack for buffer overflow protection", in "Proceedings of the 1st Conference on Computing Frontiers", CF 04, p. 335342 (Association for Computing Machinery, New York, NY, USA, 2004), URL https://doi.org/10.1145/977091.977139.

[22] Petsios, T., V. P. Kemerlis, M. Polychronakis and A. D. Keromytis, "Dynaguard: Armoring canary-based protections against brute-force attacks", https://cs.brown.edu/~vpk/papers/dynaguard.acsac15.pdf (2015).

[23] Sasnauskas, R. and J. Regehr, "Intent fuzzer: crafting intents of death", in "Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)", pp. 1–5 (2014).

[24] Shen, F., N. Vishnubhotla, C. Todarka, M. Arora, B. Dhandapani, E. J. Lehner, S. Y. Ko and L. Ziarek, "Information flows as a permission mechanism", in "Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering", ASE 14, p. 515526 (Association for Computing Machinery, New York, NY, USA, 2014), URL https://doi.org/10.1145/2642937.2643018.

[25] Shoshitaishvili, Y., R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis", (2016).

[26] V. Tsaousoglou, P. A., "The shadow over android heap exploitation assistance for androids libc allocator", Infiltrate (2017).

[27] Wang, C., W. Duan, J. Ma and C. Wang, "The research of android system architecture and application programming", in "Proceedings of 2011 International Conference on Computer Science and Network Technology", vol. 2, pp. 785–790 (IEEE, 2011).

[28] Ye, H., S. Cheng, L. Zhang and F. Jiang, "Droidfuzzer: Fuzzing the android apps with intent-filter tag", in "Proceedings of International Conference on Advances in Mobile Computing & Multimedia", MoMM 13, p. 6874 (Association for Computing Machinery, New York, NY, USA, 2013), URL `https://doi.org/10.1145/2536853.2536881`.

[29] Zhang, B., B. Wu, C. Feng and C. Tang, "Memory corruption vulnerabilities detection for android binary software", in "2015 IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC)", pp. 1–5 (2015).

[30] Zhang, M. and S. Zonouz, "Use-after-free mitigation via protected heap allocation", in "2018 IEEE Conference on Dependable and Secure Computing (DSC)", pp. 1–8 (IEEE, 2018).

[31] Zhu, D., Y. Li, N. Pang and W. Feng, "An android system vulnerability risk evaluation method for heap overflow", in "2016 4th International Conference on Enterprise Systems (ES)", pp. 89–96 (2016).