Air Force Institute of Technology
AFIT Scholar

Theses and Dissertations

Student Graduate Works

6-14-2012

Covert Android Rootkit Detection: Evaluating Linux Kernel Level Rootkits on the Android Operating System

Robert C. Brodbeck

Covert Android Rootkit Detection: Evaluating Linux Kernel Level Rootkits on the Android Operating System

THESIS

Robert C. Brodbeck, Civilian, USAF

AFIT/GCO/ENG/12-14

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government. This material is declared a work of the U.S. government and is not subject to copyright protection in the United States.

AFIT/GCO/ENG/12-14

COVERT ANDROID ROOTKIT DETECTION: EVALUATING LINUX KERNEL LEVEL ROOTKITS ON THE ANDROID OPERATING SYSTEM

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science

Robert C. Brodbeck, B.S. Computer Science

Civilian, USAF

June 2012

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

AFIT/GCO/ENG/12-14

COVERT ANDROID ROOTKIT DETECTION: EVALUATING LINUX KERNEL LEVEL ROOTKITS ON THE ANDROID OPERATING SYSTEM

Robert C. Brodbeck, B.S. Computer Science
Civilian, USAF

Approved:

Rusty Baldwin ____________________ 6 Jun 12
Dr. Rusty O. Baldwin (Chairman) Date

Barry Mullins ____________________ 6 Jun 12
Dr. Barry E. Mullins (Member) Date

William Kimball ____________________ Jun 5 2012
Mr. William B. Kimball (Member) Date

Abstract

This research developed kernel level rootkits for Android mobile devices designed to avoid traditional detection methods. The rootkits use system call hooking to insert new handler functions that remove the presence of infection data. The effectiveness of the rootkit is measured with respect to its stealth against detection methods and behavior performance benchmarks. Detection method testing confirms that while detectable with proven tools, system call hooking detection is not built-in or currently available in the Google Play Android App Store. Performance behavior benchmarking showed that the new handler function inserted by the system call hooking affects the average completion time of the targeted system calls. However, this delay's magnitude may not be noticeable by average users.

The covert Android rootkits implemented target the emulator available from the Android Open Source Project (AOSP) and the Samsung Galaxy Nexus running Android 4.0. The rootkits are compiled against both Linux kernel 2.6 and 3.0, respectively. This research shows the Android's Linux kernel is vulnerable to system call hooking and additional measures should be implemented before handling sensitive data with Android.

Acknowledgments

I would like to thank my advisor and committee for their support and contribution to this work. Their expertise and guidance in selecting a research topic and gaining the knowledge needed to complete this endeavor has not gone unnoticed. I am grateful for Dr. Baldwin's always quick and insightful feedback that helped to keep me focused and on task. I'd like to thank Dr. Barry Mullins and Mr. William Kimball's contagious enthusiasm for Reverse Engineering which led me to pursue my topic and gain invaluable knowledge and technical aptitude. Most importantly, I would like to thank my family for their never-ending support during the countless and late hours spent working on the computer. Thank you.

Robert C. Brodbeck

Table of Contents

Page

List of Figures

Page

List of Tables

COVERT ANDROID ROOTKIT DETECTION: EVALUATING LINUX KERNEL LEVEL ROOTKITS ON THE ANDROID OPERATING SYSTEM

I. Introduction

1.1 Research Domain

Smartphones are mobile phones that offer more advanced features and computing power than traditional cellular phones. Beyond simply making calls, a Smartphone can carry multiple connections from cellular networks, wireless Bluetooth, the Internet (via Wi-Fi), USB and other peripherals. With these new connections, smartphone users can access email, social networks, and banking all from their mobile device. Information security, then, becomes an immediate concern with sensitive data being handled on potentially unsecure devices.

The Google Android operating system [Goo12] is currently the most widely used platform for Smartphones and is on about a quarter of Tablet PC devices. The Android operating system is a mobile device operating system for Smartphones and Tablet PCs designed by Google and the Open Handset Alliance. The Android operating system stack runs on top of the Linux kernel, typically, on a 32-bit mobile device ARM processor. Since Android is built on top of the Linux kernel, it inherits the same vulnerabilities and the possibility of exploitation by malware, backdoors, and rootkits to gain control of the system or induce denial-of-service (DoS) attacks. A rootkit is a set of programs and code that allows a permanent or consistent, undetectable presence on an operating system.

With the Android's widespread adoption, research in attacks may spark interest in developing preventive security measures for Android. This research is particularly interested with developing kernel level rootkits that remain undetected by currently available detection methods on the Android operating system.

1.2 Problem Statement

Android dominance of the Smartphone market has made it an inevitable target of malicious attacks. Malware for Android typically targets sensitive information like GPS location, Short Message Service (SMS) billing, bank account credentials, premium phone calls, e-mails, and social network credentials. Understanding how malware remains undetected when it accesses to this information is advantageous to increased development in detection and operating system security measures.

Kernel level rootkits run at the highest privilege by manipulating memory known as kernel space. Malware developers insert rootkits into operating system by exploiting software bugs. The Android operating system is no exception and old software vulnerability attacks become new when targeting its Linux kernel.

1.3 Research Goals

Kernel level rootkits that remain undetected persist longer and increase the capability of an attacker to exfiltrate data from the targeted device. Rootkit effectiveness, then, can be determined by the detectability of the rootkit. Modern rootkits divert the flow of execution at the kernel level to prevent infection detection. Understanding and evaluating these techniques can lead to more effective detection measures. The goal of this research is to determine the effectiveness of various covert techniques implemented in kernel level rootkits on the Android operating system's Linux kernel. The covert techniques are based on traditional implementations of system call hooking used to hide infection data. The rootkits are tested against available and proven detection techniques and benchmarked for behavior performance analysis to determine the rootkit's effectiveness.

1.4 Document Outline

Chapter II introduces the Android operating system, conventional software exploits, and taxonomy of rootkits. Chapter III presents the methodology for evaluating the rootkits developed for this research. The rootkits are designed to evade the currently available detection methods. Chapter IV presents the design and implementation of the rootkits. The chapter also presents the results and analysis of the rootkits against detection methods and the delay induced by the covert techniques. Chapter V highlights the accomplishments of this research and proposes future research in both offensive and defensive techniques against the Android operating system.

II. Literature Review

This chapter reviews kernel exploits and rootkit techniques that target the Linux kernel component of the Android operating system. The first section introduces the focus of this research, the Android operating system and its kernel. The second provides an overview of exploits that obtain initial access to the Linux kernel. The third section provides an overview of rootkits focusing on those whose objective is to remain undetectable and persistent.

2.1 Introduction to Android

The Android operating system is a mobile device operating system for smartphones and tablets designed by Google and the Open Handset Alliance. When released in October 2008, Google also publically released the source code as the Android Open Source Project (AOSP) under Apache's open source license [Goo12]. This made the code readily available for analysis and compilation. Android runs primarily on the popular mobile device 32-bit processor, ARM. ARM is a RISC (Reduced Instruction Set Computer) architecture which means that it uses simpler instructions compared to x86 processor's CISC (Complicated Instruction Set Computer) architecture. However, the operating system concepts for a Linux kernel running on ARM are the same. Android is currently the best-selling smartphone platform worldwide. This widespread adoption has led to increased targeting by malware writers.

2.1.1 The Android Software Stack

The Android "software stack" includes an operating system, middleware, and key applications [God12]. The software stack is composed of five abstract layers shown in

Figure 2.1. From top to bottom the layers are Applications, Application Framework, Android Runtime, Native Libraries, and the Linux Kernel. This section describes the features from the top to the bottom layer.

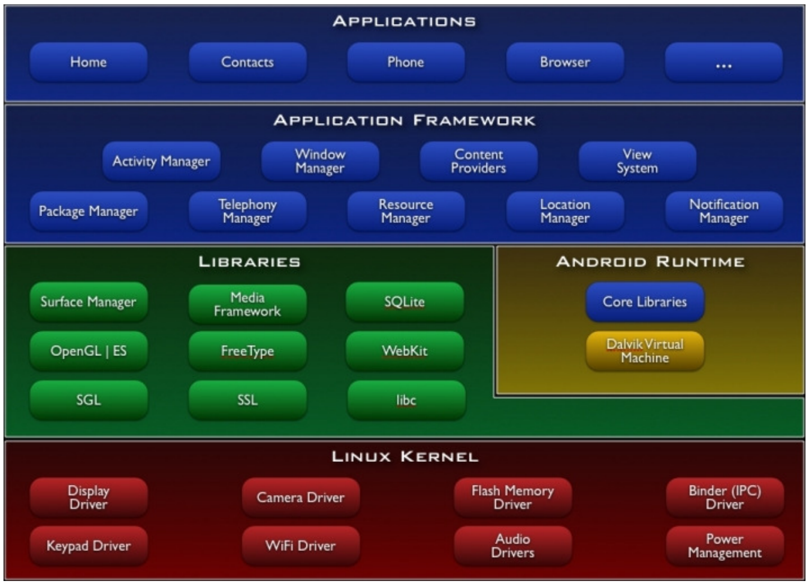


Figure 2.1 The Android Software Stack

Android comes with a set of core applications that include an email client, Short Message Service (SMS) program, calendar, maps, browser, and contacts. Other applications can be downloaded from the Android Application Market or from a Universal Serial Bus (USB) connected computer to a mobile device running Android. Developers can take advantage of the Android Software Development Kit (SDK) and design applications for public release. Applications developed in the Android SDK are written in the Java programming language but can also be written in C/C++ using the Native Developer Kit (NDK). Even though the applications are designed in Java, they run in a Dalvik Virtual Machine (DVM) rather than the Java Virtual Machine (JVM) in PC environments. The Android application framework promotes efficiency and security with its DVM application sandboxing and permission model interfaces that provide access to the lower layers.

The Android application framework sits above the system libraries, core libraries, and DVM. Since Android is an open development platform, developers can build extremely rich and innovative applications. Through the application framework, developers can utilize device hardware, access location information, run background services, set alarms, add notifications to the status bar, and much more. The application architecture is also designed to simplify the reuse and replacement components of applications securely. All applications have access to a set of services and systems that includes a rich and extensible set of Views, Content Providers, a Resource Manager, a Notification Manager, and an Activity Manager.

The Android Runtime layer is above the system libraries and kernel providing the DVM and core libraries. DVM is specifically designed for embedded environments such as mobile devices, tablet computers, and netbooks to support application portability and runtime consistency. With these features in mind, Dalvik supports multiple virtual machine processes (i.e., instances) per device and ensures runtime memory is used efficiently. Android runs every application in its own DVM instance. The virtual machine has a registered-based architecture that runs classes compiled by a Java language compiler. These classes are transformed into the optimized .dex (Dalvik Executable) file format to be more compact and memory efficient than Java class files. Java code can also be reused by converting Java .class and .jar files to .dex files at build time. The core libraries are written in Java and provide a substantial subset of the Java 5 SE packages as well as some Android-specific libraries. These libraries access the capabilities provided by the hardware (storage, network access), operating system (utilities), and native libraries (data structures).

The next layer in the Android software stack includes a collection of native libraries implemented in C/C++ used by various components. The capabilities derived from these libraries are available to developers through the Android application framework. These libraries include the System C library, media libraries, Surface Manager, LibWebCore, SGL, 3D libraries, FreeType, and SQLite.

Lastly, the Linux kernel layer acts as an abstraction layer between the hardware and the rest of the software stack. The kernel handles system services such as security, memory management, process management, network stack, and driver model. The components of the Linux kernel include display driver, camera driver, flash memory drive, binder (ipc) driver, keypad driver, Wi-Fi driver, audio drivers, and power management. The Linux kernel supports programs written in the C programming language. Since Android is built on top of the Linux kernel, it inherits its vulnerabilities and the possibility of exploitation by malware, backdoors, and rootkits to gain control of the system or cause denial of service.

2.1.2 Summary

This section gives a brief overview of the Android operating system structure and features. The abstraction of the platform’s kernel from the end-user is both an advantage from a usability standpoint and a disadvantage from a security awareness standpoint [Pap10]. Executing code below the application framework layer discreetly can easily and completely subvert a user. If this execution is malicious, attackers can perform malicious activities such as exfiltrate sensitive or personal data without detection. Exploiting software vulnerabilities allow attackers to install tools to perform malicious activity. The following section discusses these vulnerabilities and exploitation of Android's Linux kernel.

2.2 Software Exploitation in the Linux Kernel

Vulnerabilities in software are often due to programming errors known as bugs. Bugs are defined as a malfunction in a program that makes the program produce incorrect results, behave in an undesired way, or simply crash [Per10]. Security issues arise from vulnerabilities that are exploited. Exploits that can be reused on similar vulnerabilities can be generalized into vulnerability classes. Classes discussed in the following sections include uninitialized pointer dereferences, memory corruption vulnerabilities, integer overflows, and race conditions.

2.2.1 Uninitialized Pointer Dereferences

A pointer is a variable that holds the address of another variable in memory. When dereferencing a point, the object pointed to is accessed. Static, uninitialized pointers will always contain a NULL (0x0) value and a NULL return value indicates a failure in memory allocation. NULL pointer dereferences occur when a kernel path dereferences a NULL pointer causing a kernel panic. The kernel will try to use the memory address 0x0 which usually is not mapped. Exploitation can occur when an attacker can predict or force a pointer dereference to an uninitialized, unvalidated, or corrupted pointer resulting in a read or write to an arbitrary location by the kernel [Per10] [Sqr07].

2.2.2 Memory Corruption Vulnerabilities

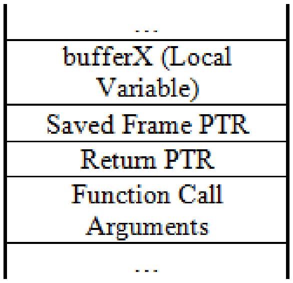
Memory corruption vulnerabilities classes include cases in which kernel memory is corrupted as a consequence of poorly written code that overwrites the kernel's contents. Kernel memory consists of the kernel stack and the kernel heap [Per10]. The kernel stack is associated with each process whenever it runs at the kernel level. The kernel heap is used each time a kernel needs to allocate memory. The fundamentals of exploiting these structures translate to the user space as buffer overflows. Buffer overflows are explained below to introduce kernel memory exploitation.

Buffer overflow exploits are a common avenue for an attacker to gain access and control of a machine. The vulnerability is typically exploited by sending more data to a program than the developer intended [How09]. The memory a program uses to store instances of the same data type is known as a buffer which stores things like character arrays or strings. Strings are primarily used for input and output to the user. The structure of how this data is handled in a program must be understood to take advantage of a buffer overflow.

A program process is organized into three memory sections: text, data, and stack [Ale96]. The text section is fixed by the program and includes code and read-only data. As this section contains executable code, it normally has read-only permissions; attempts to write to this section will result in a memory segmentation fault. The data section contains initialized and uninitialized data. Data variables for the program are stored here. This section corresponds to the data-bss section of the executable file. While these two sections are important to executing a program, the stack section is the target of a buffer overflow.

The stack section is where dynamic data is stored. It is used for local variables, to pass parameter values, and to return values from functions and procedures. The stack section is a Last-In-First-Out (LIFO) data structure which means that elements are added or removed from only one end or top of the structure. The stack grows into the higher memory addresses as elements are added. If the data and stack section grow into each other, the process is blocked and run again with more memory allocated.

A pointer to the top of the stack in memory is stored in a CPU register called the stack pointer (SP). The stack consists of logical stack frames that are allocated when calling a function and unallocated when returning. The base of the current stack frame is pointed to by the stack frame pointer (FP). When a function is called, the previous FP is pushed (i.e., saved) onto the stack. The SP is copied into the FP to allocate the new frame and the SP is incremented to allocate space for local variables. These actions are called the prolog of the function while the actions of a returning function are called an epilog. At the epilog, the actions of the prolog must be “reversed” and cleaned up. An attacker can develop an exploit with a process’ stack structure via buffer overflows.



...
bufferX (Local Variable)
Saved Frame PTR
Return PTR
Function Call Arguments
...

Figure 2.2 Program Stack

Consider a vulnerable program stack as shown in Figure 2.2. The variable bufferX has 8 bytes allocated and the saved frame pointer and return pointer have 4 bytes allocated, respectively. If bufferX is controlled by user input and the bounds of the buffer are not checked within the program, a malicious user could exploit this. To test for buffer overflow vulnerabilities, the user may enter a large amount of input. Based on the example presented earlier, this will change the memory state from Table 2.1 to Table 2.2 supposing the user entered hexadecimal A's.

Table 2.1 The Initial State of Memory

	bufferX	**Saved Frame PTR**	**Return PTR**
Address	0x00 – 0x07	0x08 – 0x0B	0xC – 0x0F
Data	0x0	0x1234	0x600D

Table 2.2 The Post State of Testing Memory

	bufferX	**Saved Frame PTR**	**Return PTR**
Address	0x00 – 0x07	0x08 – 0x0B	0xC – 0x0F
Data	AAAA AAAA	AAAA	AAAA

A segmentation fault will occur but the user now knows a buffer overflow is present. Therefore, the user will probe the program until the location of the return pointer is located. The objective of this exploit is for the malicious user to run shellcode [Sko06]. Suppose it is determined that the return pointer starts 12 bytes from buffer; the process can be forced to return to an address where desired malicious code resides by overwriting the 4 bytes after that. If the input from Table 2.3 is entered the instruction pointer will point to the malicious code in bufferX and the malicious user has successfully compromised the program and the machine running it.

Table 2.3 Exploiting Memory

	bufferX	**Saved Frame PTR**	**Return PTR**
Address	0x00 – 0x07	0x08 – 0x0B	0xC – 0x0F
Data	/bin/sh	SSSS	0x00

The heap data structure can also be targeted by a malicious user if the buffer bounds are not properly checked when memory is allocated. These exploits are known as heap overflows. As in buffer overflow, heap overflow can allow an attacker to redirect the flow of program execution or change other variables in vulnerable programs. The buffer overflow example is simplified but the same technique is applied to stack and heap structures. This technique can be leveraged by attackers to run shell code and gain access from remote machines, exfiltrate information, or install software such as rootkits.

2.2.3 Integer Overflows

Integer overflow occurs when the value of an integer is increased beyond the maximum value it can represent given the number of bytes allocated. The result can include ignoring the overflow or aborting the program. However, most compilers store the incorrect value and continue executing causing sometimes disastrous and unexpected results.

Integers are typically the same size as a pointer on the system they are compiled on. The Android operating system runs on 32-bit ARM processors therefore the pointers and the integers will be 32-bit. Suppose a program sets an unsigned short integer variable x to its maximum value of 65,535 (represented in hexadecimal as 0xffffffff). Suppose the program is overwritten so that it will add 1 to x until it reaches 70,000 where it will terminate. The program will actually never terminate. If overflow is ignored when 1 is added to 65,535, the result is truncated to a size that can be stored into the 32-bit length of x, in this case is 0x00000000. Thus, the program will continue in an infinite loop. Signed integers also have this problem because the range for a 32-bit signed integer is from –32,768 to 32,767. When 1 is added to 32,767, the result is –32,768. Integer overflow affecting the sign of the value is classified as a signedness bug. These examples are clearly programming errors and cannot be exploited by a malicious user. If, however, there was user input and a more complex control structure, a malicious user could alter the execution flow by entering input targeting the integer overflow and thereby exploit the program.

Frequent targets of integer overflow exploits include network daemons and operating system kernels. However, all integer overflows are not exploitable because memory is not being directly overwritten [Ble02]. They do not allow direct execution flow control but the subtle effect of an erroneous value can lead to problems later in the code which may enable the exploitation of bugs such as buffer or heap overflows [How09]. Although, integer overflows may not cause direct compromise of an application, the application will not detect that a calculation was performed incorrectly and will continue to execute. The unexpected behavior of the application continuing after an integer overflow introduces a vulnerability into the system.

2.2.4 Race Conditions

A race condition occurs when two actors (i.e., processes or threads) compete within the same time interval for the same resource. The integrity of the resulting data or the correctness of computing tasks may be affected. Race conditions primarily occur in operating systems but can also occur in multithreaded or cooperating processes [Pfl11].

The type of system can complicate exploiting a race condition. Symmetric multiprocessing (SMP) systems are easier to exploit because multiple kernel paths can be concurrently executing on multiple processors increasing the likelihood of a kernel race condition. On uniprocessor (UP) systems, however, it is more difficult to set up a situation where race condition will occur. For example, suppose two kernel tasks with a possible race condition are concurrently executing on an UP system. The first task must somehow be preempted. The scheduler then must be forced into selecting the second 'racing' thread. Finally, the race condition is exploited if the second task modifies the same kernel memory as the first. Race conditions can be prevented using synchronization primitives (e.g., locks, semaphores, conditional variables, or monitors) but these reduce performance and often induce deadlocks in a system [Per10].

A race condition is possible at a relatively high level in Linux using files and other objects [How09]. Suppose an application needs to create a temporary file. It first checks to see if the file already exists and if not the application creates the file. An attacker deduces the naming scheme of the temporary file and creates a link back to a file of the attacker's choice. If the temporary file's suid bit is set to root, the file executes as root causing a privilege escalation for the attacker. Thus, this race condition causes a privilege escalation.

2.2.5 Summary

This section reviewed software exploitation of a Linux kernel. There are protection technologies built to defend against these exploits but attacks continue to overcome these protection mechanisms. The following section discusses how rootkits avoid detection and maintain persistence within a compromised system.

2.3 Introduction to Rootkits

Suppose a user with malicious intent has an exploit for an Android phone. The exploit runs shellcode but the delivery mechanism may be detected. Therefore, the attacker wants to deliver one payload that maintains access and is not detectable by the end user or an administrator. A backdoor that bypasses authentication mechanisms is a good solution, but it is not the best because backdoors are noisy and detectable by an Intrusion Detection System (IDS). The best solution to avoid detection would be a rootkit. A rootkit allows an attacker to permanently or consistently maintain undetectable access to the root, that is, the most powerful user on a system. Rootkits can enable an attacker to install a backdoor to remote control a compromised system and exfiltrate information.

Rootkits are typically organized into two classes, user level and kernel level. A typical user level rootkit is designed to gain full control of the memory space of a targeted application, while kernel level rootkits run at the highest privilege by controlling the memory known as kernel space. This section explains the techniques used in these classes of rootkits.

2.3.1 User Level Rootkits

User level rootkits are unprivileged and are stored outside of kernel memory space. They are user space code that patches or replaces existing applications to provide cover for malicious activities. User level rootkits replace system binaries, add malicious utilities, change configuration files, delete files, or launch malicious processes [Gri06]. For example, the Linux system program 'ls' could be changed so as to not reveal the presence of a malicious file in a directory. These rootkits, while effective, are easily detected by file system integrity and signature checking tools [Dav08], therefore, most modern IDS software prevent these rootkits from being installed or can detect an active intruder.

2.3.2 Kernel Level Rootkits

Kernel level rootkits defeat such tools by directly modifying the operating system kernel. These rootkits modify the execution flow of kernel code to run their own payload. However, modifying the kernel in this way can drastically affect the stability of the system causing a kernel panic.

The simplest way to introduce code into a running kernel is through a Loadable Kernel Module (LKM) [Kon07]. LKMs add flexibility to an operating system by providing a means to add functionality without recompiling the entire kernel. Added functionality might include device drivers, filesystem drivers, system calls, network drivers, TTY line disciplines, and executable interpreters [Hen06]. Most modern UNIX-like systems, including Solaris, Linux, and FreeBSD, use or support LKMs [Zov01]. However, the kernel packaged with Android does not support LKMs by default. The kernel can be recompiled and installed on Android to add LKM support if physical access to the mobile is available. LKMs are very useful, but they also allow maliciously written kernel modules to subvert the entire operating system which can lead to a loss of control of the Linux kernel and consequently all the layers above the kernel [Pap10]. Kernel level rootkits typically subvert the kernel to hide processes, modules, connections and more to avoid detection. Particular techniques include hooking system calls, direct kernel object manipulation (DKOM), run-time kernel memory patching, interrupt descriptor table hooking, and intercepting calls handled by Virtual File System (VFS). These techniques are discussed at a high level in this section.

2.3.2.1 Hooking System Calls

The kernel provides a set of interfaces or system calls by which processes running in user space can interact with the system. The applications in user space send requests through this interface and the kernel fulfills requests or returns an error. The execution flow of a system call can be seen in Figure 2.3 [Bov05] [Lov10]. A process invokes a system call to jump from user space to the assembly language function called the system call handler in kernel space. The system call number passes from the wrapper routine to the handler function via the EAX register. The system call table calls the appropriate system call service routine location based on the number passed and returns a number indicating success or error. Not allowing user space applications to access or run in kernel space provides stability and security to the entire operating system. This arbitration prevents applications from incorrectly using hardware, stealing other processes' resources, or otherwise doing harm to the system inadvertently or otherwise. Even so, system calls can be hooked to exploit the power of the kernel.

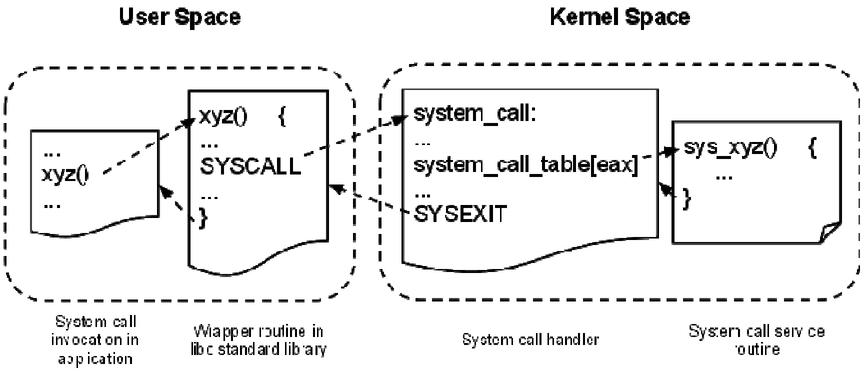


Figure 2.3 Normal System Call [Bov05] [Lov10]

Hooking is a technique that employs handler function, called hooks, to modify control flow [Kon07]. A new hook registers its address as the location for a specific function, so when that function is called the hook runs instead. Typically, a hook will call the original function at some point to preserve the original behavior. System calls can be hooked using a maliciously designed LKM to alter the structure of the system call table.

To hook the system call table, the original targeted system call pointer to the function must be saved. The original system call is also called to preserve the original behavior because the objective of a hooked call is to modify the I/O of the function, not destroy it. A pointer to the hooked system call handler, typically located in the LKM, is saved in the system call table location of the target. At any point the target system call is made, it will move through the hooked system call handler providing a system-wide hook stored in kernel space. Figure 2.4 [Bov05] [Lov10] shows the inserted hooked system call and how it maintains the execution flow by calling the proper system call service routine. The hooked system call returns control to user space after completion.

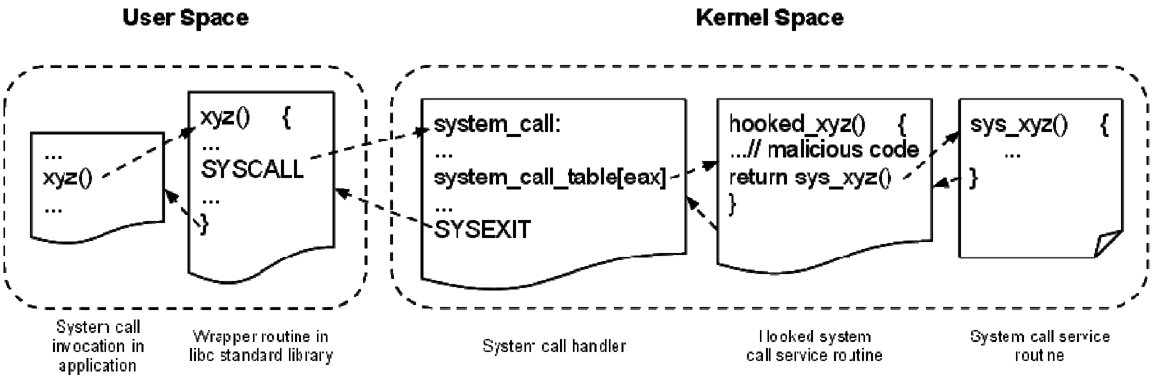


Figure 2.4 Hooked System Call [Bov05] [Lov10]

2.3.2.2 Direct Kernel Object Manipulation (DKOM)

Hooking the system write call allows a rootkit to hide from system binaries like ls, lsmod, and ps; even so, robust IDSs can still detect the existence by following the kernel structures. All operating systems store internal record-keeping data in main memory [Kon07]. The Linux kernel is no exception and provides generic data structures and primitives to encourage code reuse by developers [Bov05]. These structures, that all programmers are familiar with, include linked lists, queues, maps, and binary trees. Altering the data in these structures to hide an attacker's activity is called Direct Kernel Object Manipulation (DKOM)

For example, the Linux kernel contains a process list that links together all existing process descriptors in a doubly linked list. Each process is contained in a task_struct structure as in Figure 2.5 [But04]. The task_struct contains the pointers to the prev_task and next_task. Removing the malicious process from the list of prev_task and next_task will hide the malicious process from the system as shown in Figure 2.6 [But04]. This technique can change depending on the kernel version but the technique is essentially the same in each implementation.

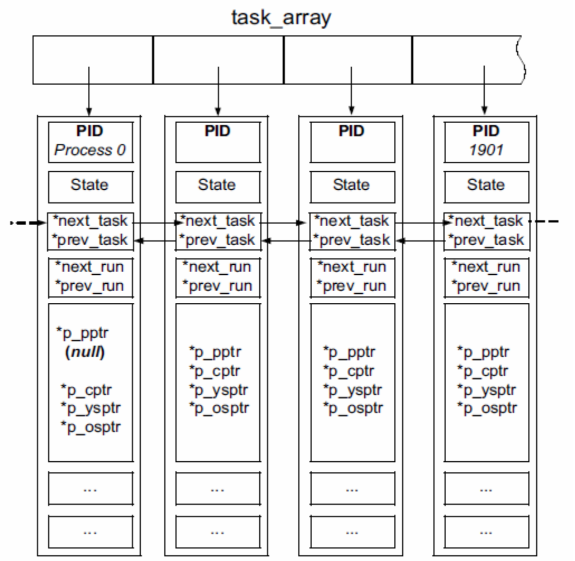


Figure 2.5 Normal Kernel Object Linking [But04]

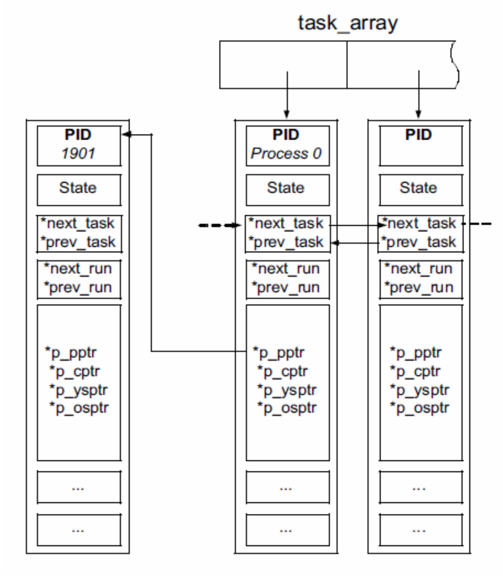


Figure 2.6 Direct Kernel Object Manipulation [But04]

2.3.2.3 Run-Time Kernel Memory Patching

The classic and arguably easiest way to introduce code into the Linux kernel is through a LKM. Another technique patches a running kernel with user space code, also known as Run-Time Kernel Memory Patching (RKP) [Kon07][Pra99]. Interacting with /dev/kmem device allows reading and writing to kernel virtual memory. Note that root permissions must be present. RKP has been used to install LKMs without LKM support [Ces98] and cloak system call hooks [Sdd01].

2.3.2.4 Interrupt Descriptor Table (IDT) Hooking

An interrupt is an event that alters the sequence of instructions executed by the processor. When an interrupt occurs a system table called the Interrupt Descriptor Table (IDT) associates each interrupt or exception with the address of the corresponding handler [Bov05]. System calls use software interrupts to switch from user mode to kernel mode. The interrupt handler invokes the system call handler from the address stored in the system call table. A rootkit can hook the IDT by modifying the interrupt handler address in the IDT or by patching the first few instructions of the interrupt handler [Sha08]. These modifications would put the rootkit code in the flow of execution while still letting the system handle interrupts properly [Kad02].

2.3.2.5 Intercepting Calls Handled by VFS

The virtual file system (VFS) can also be targeted to compromise the kernel and hide the attacker's presence. The VFS is a software layer in the Linux kernel that handles all system calls related to the standard UNIX file system. VFS can handle several different types of file systems [Lev06]. The adore-ng kernel-level rootkit targets the VFS by replacing the VFS handler routines with its own routines. These handler routines provide directory listings to /proc file systems. Therefore, modifying these handles can hide specified files and processes from the user mode programs.

2.3.3 Firmware Level Rootkits

Firmware-based rootkits (also known as bootkits) can ensure persistence against removal. A firmware-based rootkit hides by modifying the software on devices such as the Advanced Configuration and Power Interface (ACPI) BIOS and Peripheral Component Interconnect (PCI) BIOS. The firmware is modified to contain malicious ACPI Machine Language (AML) instructions that interact with system memory and the I/O space thereby allowing the rootkit to bootstrap code that overwrites kernel memory as a means of infection [Hea06]. Although an effective technique, firmware rootkits are easily detected by Trusted Computing Group's Trusted Platform Module (TPM). TPM checks the integrity of the operating system image and firmware and is in widespread use today [Dav08].

2.3.4 Virtual Machine Based Rootkits (VMBR)

Rootkit activity can be also hidden using a Virtual Machine Based Rootkit (VMBR). VMBR operates without modifying anything on the system while monitoring an operating system's activity. Therefore, any IDS integrity checks will not detect any presence of a rootkit because it is actually handling the virtualization of the entire operating system. VMBRs can be either software or hardware based.

2.3.4.1 Software Virtual Machine Based Rootkit

Software VMBRs virtualize the target operating system by executing the rootkit within a separate operating system hosting a virtual machine monitor (VMM). Figure 2.7 shows how the subverted system is stacked after a software VMBR is installed. The rootkit code remains hidden from the subverted operating system because it executes in a separate operating system context [Kim08]. μBeR, a proof of concept VMBR rootkit, uses this technique by installing a microkernel to subvert the Android operating system to provide malicious services [Tri10]. Although the microkernel has a small footprint, the rootkit induces performance overhead which can be detected based on discrepancies between virtual and physical hardware [Dav08].

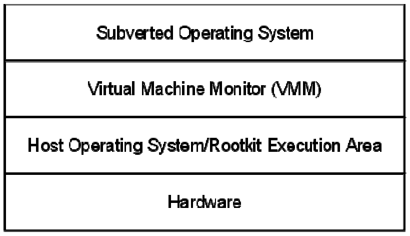

Subverted Operating System
Virtual Machine Monitor (VMM)
Host Operating System/Rootkit Execution Area
Hardware

Figure 2.7 Software Virtual Machine Based Rootkit

2.3.4.2 Hardware Virtual Machine Based Rootkits

Hardware VMBRs are a specialized rootkit that uses specific instruction sets to switch contexts between VMM and the guest operating system. The rootkit virtualizes an operating system by gaining root access and installing the rootkit hypervisor. It carves out memory for the hypervisor and migrates the running operating system into a virtual machine. The rootkit then intercepts access to hypervisor memory and selected hardware devices. Figure 2.8 illustrates the structure of the system after the hypervisor is installed [Zov06]. Hardware VMBRs are specialized because they only target specific technologies such as AMD SVM (Bluepill) and Intel VT (Vitriol) processors [Rut06] [Zov06]. Hardware VMBRs are detectable because hypervisors must use cache, memory bandwidth, and TLB entries in the course of multiplexing a CPU. A guest operating system can be made intentionally sensitive to these resources to detect an attempted hypervisor install [Kim08].

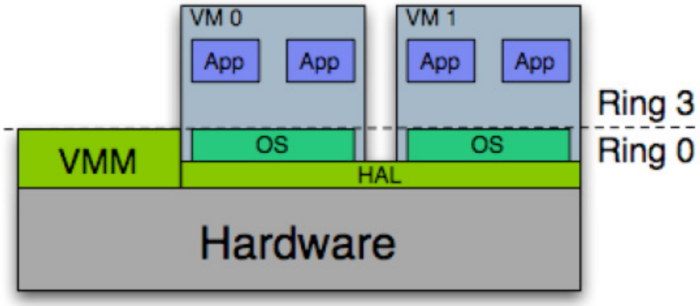


Figure 2.8 Hardware Virtual Machine Based Rootkit [Zov06]

2.3.5 Summary

This section provides a taxonomy of rootkits and an overview of techniques that can be employed by a kernel level rootkit in Linux. The rootkits described include user level rootkits, kernel level rootkits, firmware level rootkits, and virtual machine based rootkits. The kernel level rootkit techniques covered include hooking system calls, direct kernel object manipulation (DKOM), run-time kernel memory patching, interrupt descriptor table (IDT) hooking, and intercepting calls handled by the virtual file system (VFS). This section reviews the attack target, initial compromise through exploits, and maintaining persistence and stealth within the Linux kernel with rootkits.

III. Methodology

3.1 Background

The Android operating system is a mobile device operating system for Smartphones and Tablet PCs designed by Google and the Open Handset Alliance. The code is open source licensed and available under the Android Open Source Project (AOSP) [Goo12]. Releasing the code under open source license makes it readily available for analysis and compilation. The Android operating system stack runs on top of the Linux kernel typically on the 32-bit mobile device ARM processor. Since Android is built on top of the Linux kernel, it inherits its vulnerabilities and the possibility of exploitation by malware, backdoors, and rootkits to gain control of the system or induce denial of service (DoS). Widespread adoption of Android has led to increased targeting by malware writers. Android attacks have naturally sparked interest in researching protections for Android. This research is particularly interested in developing kernel level rootkits; a set of tools consisting of small programs that allow an attacker to permanently or reliably maintain undetectable access to the root user, that is, the most powerful user on a system. Kernel level rootkits run at the highest privilege by manipulating memory known as kernel space. Attackers insert rootkits into operating system by exploiting software bugs. This research examines the detectability of system call hooking rootkits for the Android operating system by examining subversion techniques inherited from the underlying Linux kernel. The rootkit's detectability is measured against currently available security mechanisms and anomaly detection methods.

3.2 Problem Definition

This section describes the specific goals of the research along with the research hypothesis. The approach describes how the hypothesis is tested against the research goal.

3.2.1 Goals and Hypothesis

The goal of this research is to determine the effectiveness of traditional system call hooking techniques implemented by a kernel level rootkit against the Android operating system. The effectiveness of the rootkit is measured with respect to its stealth against currently available security mechanisms and anomaly detection methods.

The hypothesis of this research is that the Android's Linux kernel cannot be trusted and additional measures should be implemented before handling sensitive data with Android. The rootkit is expected to be effective without the end user noticing, thereby preventing any indication of infection.

3.2.2 Approach

Rootkits maintain access to a system by hiding their presence from the end user. An administrator may use the conventional Linux tools to look for suspicious files, processes, modules, or ports. These tools include ls, netstat, ps, and lsmod, respectively. The cat command is also used to probe files containing system information. A rootkit designed to hide from these commands would then be an effective way to remain undetected by an end user or administrator. Modifying the behavior of these tools' processes can be done by hooking system call functions with code at the kernel level to remove signs of an access breach or infection.

The most straightforward way to introduce code into the kernel is by using a loadable module in Linux. The operating system allows these extensions to be loaded so manufacturers of third party hardware can add support for their products. Therefore, any code can be loaded into kernel space via a Loadable Kernel Module (LKM). Code running at the kernel level has full access to all privileged memory of the kernel and system processes. This research leverages LKMs to introduce covert techniques by hooking system calls to control the Linux kernel component of the Android operating system.

The effectiveness of each rootkit is evaluated against rootkit detection methods on both an emulator and device. These detection methods are probe-based, integrity-based, behavior-based, heuristic-based, and signature-based. Each method's implementation is discussed in detail in Appendix A.

3.3 System Boundaries

The System Under Test (SUT) is the Covert Android Rootkit Detection System (CARDS). CARDS includes the Android mobile device, the Android operating system (OS), an Android LKM rootkit, and the infection data hidden by the covert techniques. The Android operating system platform is built for the ARM processor architecture. No other operating system or processors are considered. The SUT does not initially include any Android applications that provide system protection. The workload is the covert techniques employed by the LKM rootkit and the detection methods used against the rootkit.

The Component Under Test (CUT) is the LKM rootkit inserted into the Android operating system on the emulator or device. Figure 3.1 shows CARDS complete with input, outputs, and internal components.

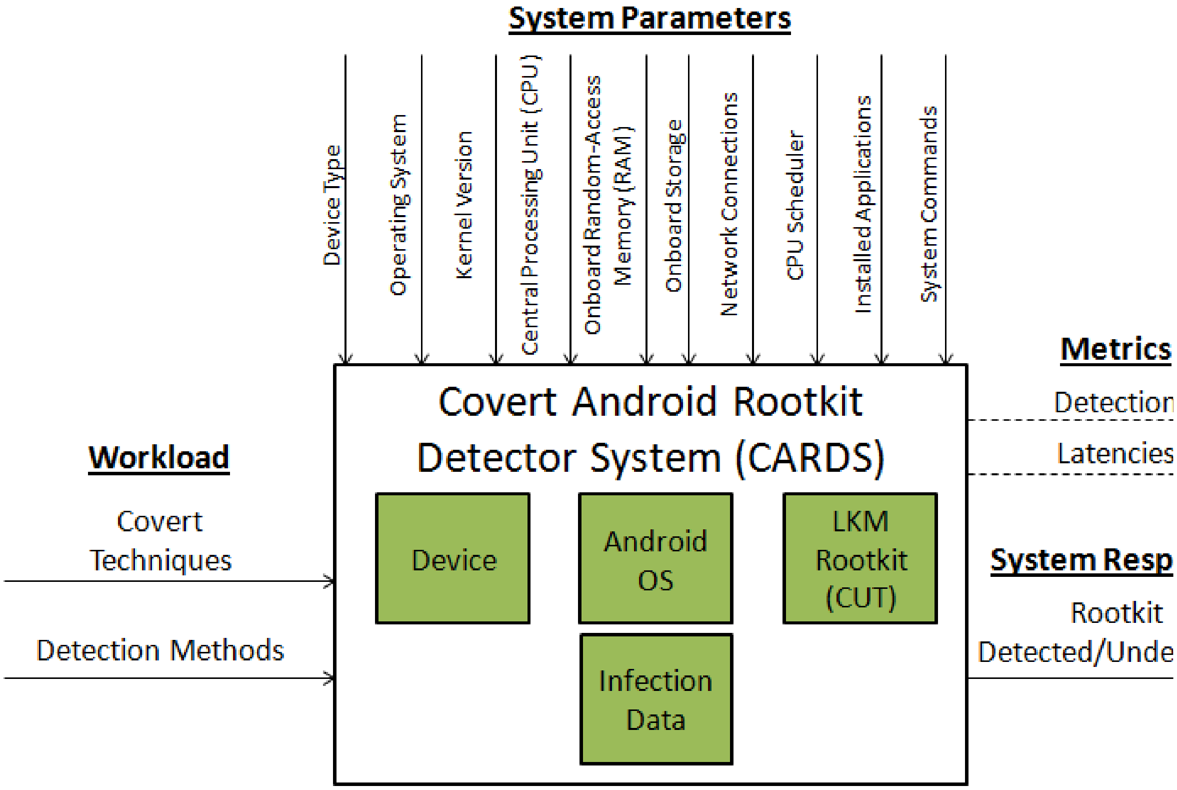


Figure 3.1 Covert Android Rootkit Detection System (CARDS)

3.4 System Services

The service that CARDS provides is stealth from detection methods. Since the rootkit is hiding from both the operating system and user, no functionality or service should be restricted in any way unless it is an outcome of the subversion. For example, the command netstat will not print out information about port 31337 because it is required to provide a backdoor to the operating system and end user. The rootkit should not cause denial of service such as loss of network communications, application crashes, or operating system crashes.

CARDS stealth service has two outcomes: the rootkit is detected or is undetected on the device. The desired outcome for the stealth service is to be undetected, but even if it is, functionality degradation can prompt a user to wipe the devices internal memory which will result in the removal of the rootkit and infection data. Therefore, the performance latency of an infection is evaluated with the behavioral-based detection method to account for possible functionality degradation.

3.5 Workload

The workload is the rootkit employing varying system call hooks to achieve covert operations and the detection methods used to detect an infection or anomalies. Detection methods use system commands, forensics tools, and Android applications. System commands include Linux system binaries used by administrators to determine a rootkit's presence. The forensics tools are open source programs compiled for the ARM processor. The Android application is publicly available in the Google Play Store. The end user initializes system commands, forensics tools, and Android applications; however, some tools are automated using scripts for streamlining data collection.

Each workload specified has separate parameters. The different covert techniques are employed interchangeably by the LKM rootkit. The Android applications, the system commands and forensic tools are included in the detection methods workload. Varying the workload will determine the stealth effectiveness against the detection methods.

3.6 Performance Metrics

The first metric is a binary response of detection methods. Successful scan detection is classified by a "yes" and an unsuccessful scan is classified by a "no". This metric determines if the rootkit remains undetectable against probe-based, signature-based, integrity-based, and heuristic-based detection methods. The covert technique performance is best when it is not detected by any method.

The second metric, latencies, compares the measured time for an uninfected or clean system call execution with an infected system call execution. The system calls measured are unique to each rootkit. System call hooking can potentially add delay to the time for a system call to be completed because the additional code added by the LKM is executed before returning to user space. This delay may alert the presence of a rootkit to the end user or a system administrator. The measurement of this metric is the difference between the beginning and the end of the system call.

3.7 System Parameters

The parameters listed below affect the performance of the rootkit:

Device Type – The device type specifies the particular mobile device platform being tested. The device determines the targeted operating system configuration and therefore the techniques that can be employed by the rootkit. This research uses an emulator from AOSP and GSM Samsung Galaxy Nexus (GN).

Operating System (OS) – The operating system determines the kernel that the LKM rootkit can be compiled against. The operating system version used in this research is Android 4.0 (Ice Cream Sandwich).

Kernel Version – The kernel version changes how the rootkit covert techniques are implemented. The emulator uses Linux kernel 2.6 and GN uses Linux kernel 3.0.

Central Processing Unit (CPU) – The processor determines how fast the system can execute instructions in which how the operating system performs. The rootkit may become more detectable if the processor cannot handle the overhead of the rootkit. The emulator has an ARMEABI-v7A ARM Cortex-A8 processor and the GN has a 1.2 GHz TI OMAP 4460 ARM Cortex-A9 dual-core processor.

Onboard Random-Access Memory (RAM) – The RAM available determines how quickly the operating system can read and write to memory thus affecting overall performance. After a power cycle, RAM is wiped and the operating system is reloaded at boot. Consequently, the LKM rootkit will also be wiped at this time. The emulator has a 1024MB allocated RAM and the GN has 1GB of onboard RAM.

Onboard Storage Space – The storage space available in the system determines how much data and programs can be loaded on the device at a time. More executable code can be loaded with higher capacities. The emulator has a 496MB storage and 4GB SD card allocated and the GN has 16GB of onboard storage.

Network Connections – Network connections allow the phone to communicate with other devices and can be leveraged by the rootkit for remote command and control or to exfiltrate data. The emulator has only an Internet network connection shared from the host machine. The network connections the GN has are GSM 850/900/1800/1900 MHz, HSPDA 850/1700/1900/2100 MHz, and Wi-Fi: IEEE

802.11 a/b/g/n (2.4/5 GHz). The netstat command can be used to view the open ports; the rootkit hides its open connection.

CPU Scheduler – The CPU scheduler allocates CPU time efficiently while providing responsive user feedback. By allocating CPU time to a process, the current running process is preempted until it runs again by the scheduler. Preemption can lead to artificial execution completion delays in performance.

Installed Applications – Installed applications indicate the type of data that may be covertly exfiltrated from the target. Default applications on Android 4.0 include: Browser, Calculator, Calendar, Camera, Clock, Email, Gallery, Messaging, Movie Studio, Music, People, Phone, Search, Settings, and Voice Dialer.

System Commands – These programs are installed with the kernel and can be used over the Android Debug Bridge (ADB) command line. These tools are in the directory /system/bin and are included in the PATH environment variable by default. Commands typically loaded on a Linux device can also be used by compiling compatible open source code for ARM. These missing programs are included in the side-loaded BusyBox toolkit.

3.8 Factors

The following experimental factors are used at the indicated levels. Table 3.1 contains all the factors and levels.

Table 3.1 Experimental Factors

Factors	Levels
Covert Techniques	hide_file
	hide_proc
	hide_mod
	hide_port
Platforms	Emulator
	Device
Detection Methods	Probe
	Integrity
	Signature
	Heuristic
	Behavioral

- Covert Techniques
  - hide_file – a technique that hooks system calls to hide files that contain a specified magic string. The rootkit is designed to completely hide the specified file from the command ls and cat.
  - hide_proc – a technique that hooks system calls to hide running processes that contain a specified magic string. The rootkit is designed to completely hide the specified running process from the command ps, ls /proc, and kill.
  - hide_mod – a technique that hooks system calls to hide modules that contain a specified magic string. The rootkit completely hides such modules from the command lsmod, cat /proc/modules, and rmmod.
  - hide_port – a technique that hooks system calls to hide a specified open port from the command netstat and cat /proc/net/tcp6.
- Platforms
  - Emulator – a virtual mobile device that runs a full Android system stack on a computer. The emulator allows a simulation of how the

rootkits are expected to perform. The emulator is compiled with AOSP and runs the codenamed Goldfish Linux kernel.

  - Device – a physical mobile device compatible with the Android OS. The device allows a real world performance analysis for the rootkits. The device is the Samsung Galaxy Nexus and runs the codenamed Maguro Linux kernel.

- Detection Methods
  - Probe – Rootkit detection using the system commands to find an unusual presence of a file, open port, process, or module. This method is the base detection method that all the covert techniques implemented should circumvent.
  - Integrity – Rootkit detection by comparing files and memory with a trusted source. An example of a trusted source is a baseline system or a previous snapshot of the files and memory.
  - Signature – Android applications installed on a mobile device to scan for signatures of known malware. The rootkits implemented are not expected to be detected because they have not been publicly released.
  - Heuristic – Rootkit detection by recognizing any deviations in a computer's expected output.
  - Behavioral – Rootkit detection by deducing a rootkit infection by monitoring normal system execution to identify anomalies in performance.

3.9 Evaluation Technique

A combination of simulation and measurement is used to evaluate the system. Each rootkit's objective functionality is validated by the probe-based detection and then tested against the test harness that includes the signature, integrity, and heuristic detection methods. Performance latencies are measured via system call completion times using the strace command. The completion times are measured 5 times before and after infection for a total of 10 measurements for each system call hooked by the rootkit. Repeating capture of system call completion times 5 times was determined to be sufficient to distinguish a difference between the uninfected and infected state. The evaluation is performed on the emulator and then performed on the mobile device to show real-world performance.

This evaluation is performed on both the emulator and a mobile device loaded with Android 4.0 Ice Cream Sandwich (ICS). A Dell Latitude E6510 laptop is used to compile code, run the emulator, and communicate via ADB. AOSP provides an emulator for the Android environment [Goo12]. The mobile device tested is an unlocked GSM-version of Samsung Galaxy Nexus, which is one of few phones recommended for building Android from AOSP [Goo12]. The kernel is configured to enable LKM installation and loaded to the emulator and device via ADB. Tools installed on to Android include: BusyBox [Vla12], unhide-tcp [Lin12], skdet [Gev12], Lookout Security & Antivirus [Loo12], and strace v4.5.18 [Kra12]. Scripts used to automate testing are included in Appendix A.

3.10 Experimental Design

A full factorial design is used to evaluate the interaction between the factors. The factors include covert techniques employed by each rootkit, platforms used for evaluation, and detection methods, with 4, 2, and 5 levels, respectively. This results in 4 x 2 x 5 = 40 experiments. Each experiment is run until the output can be determined. The kernel versions, CPU, onboard RAM, onboard storage, network connections depend on the device type. The operating system, CPU scheduler, installed applications and system commands are consistent throughout all the experiments while factors vary.

The variance in the results in this research should be low or zero because the results directly depend on a successful detection and the latency of repeated system call code. The latency of the system calls will be reported with 95% confidence. System overhead is expected to be higher for most of the system calls infected by each rootkit.

3.11 Methodology Summary

As mobile devices become more widespread, they continue to become targets for malicious attackers. Unfortunately, mobile operating systems have inherited the same vulnerabilities as their PC counterparts. This chapter describes the methodology for evaluating the detectability of a kernel level rootkit against the Google Android operating system on an emulator and Samsung Galaxy Nexus. The goal of the research is to determine whether a novel kernel level rootkit is undetectable to current security mechanisms.

The SUT and CUT are identified along with accompanying parameters. Factors are selected from the system and workload parameters. The methodology tests these factors during experimentation to produce results to determine the effectiveness of the rootkit. The metrics used for evaluation are detectability and system call latencies.

The methodology consists of both simulation and measurement evaluations. AOSP supplies an emulator to use for simulations and Samsung Galaxy Nexus connected to a Dell Latitude E6510 laptop is used for the measurement evaluations. A full factorial design is implemented with 40 experiments.

IV. Covert Android Rootkit Detection Experimentation Results

4.1 Introduction

This chapter presents the covert techniques that utilize system call hooking implemented in the tested rootkits. The evaluation technique used to determine the effectiveness of the rootkits is presented. Finally, the results for detection method testing and behavior latency benchmarking for covert technique rootkits are reported.

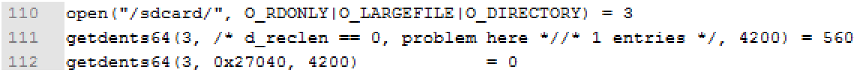
4.2 Rootkit Technical Design and Implementations

The Android operating system is fundamentally an application framework built on top of a Linux kernel. The applications that execute within the framework are written in Java and run in individual Dalvik Virtual Machine (DVM) instances [God12]. However, the system below that framework is written in a combination of C/C++, therefore C executables can be compiled and executed over a command line shell via Android Debug Bridge (ADB). Loadable kernel modules (LKM) can also be compiled for extending the kernel without recompiling. Modules operate within kernel space allowing a degree of control over the operating system. Therefore, they can be used to deliver rootkit code that can hide an intrusion in the operating system.

4.2.1 System Call Hook Development

System call hooking can be used to modify control flow of a system call in an operating system by employing an intercepting handler function. This technique is commonly installed via LKMs because the modules give direct control over memory in kernel space. This software is commonly referred to as a rootkit with the objective of hiding and maintaining privileged access to a system. To understand how this is implemented, this section will inspect targeting and hooking individual system calls.

One of the goals of a rootkit is to hide from the user without disrupting the execution flow of the operating system. However, in most cases, one cannot know which code is being executed by a process unless it is open source. The simplest way to determine which system calls are invoked by the process is the strace command. The strace command intercepts and records the system calls used by a specified process and the signals (or return values) received by that process [Cla05]. For instance, an attacker wants to hide a file in the directory from the ls command. A report of the system calls for the ls command can be generated using the command ‘strace –o ls.out ls’. The grep command can then search the output for the directory of the intended hidden file to identify that the open() operates on the directory. By targeting that open() in the output, it can be seen that the getdents64() is invoked with the pointer returned by open() as seen in Figure 4.1.



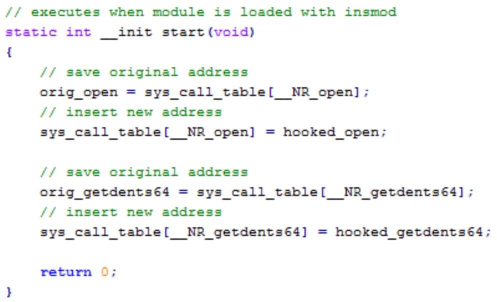
```
open("/sdcard/", O_RDONLY|O_LARGEFILE|O_DIRECTORY) = 3
getdents64(3, /* d_reclen == 0, problem here *//* 1 entries */, 4200) = 560
getdents64(3, 0x27040, 4200)            = 0
```

Figure 4.1 Opening the directory of the intended hidden file

The sequence of system calls reveals that getdents64() is the pivotal function to the execution of ls and should be the target of the rootkit. Note that the grep command may not always be available on the Android image. BusyBox, a toolkit of common UNIX utilities optimized for embedded systems, can be installed on the system to access the missing tool [Vla12].

To intercept the execution of the system call, a new function needs to be inserted in place of the targeted system call table entry. The system call table can be modified by code delivered by LKM. Prior to 2.5 Linux kernels, the system call table structure (sys_call_table) was exported to the entire kernel memory space. This new feature is an obstacle since Android runs on 2.6 or higher Linux kernels. However, there are other ways to determine the address of system call table. The first method brute forces the address by starting at a location in kernel space and incrementing the address [Cla05]. After each increment, the address is compared to the known locations of exported system calls, such as sys_read() and sys_write(), to determine the actual system call table location. The second, simpler way is by searching (using the grep utility) for 'sys_call_table' against the System.map file or, if that is unavailable, /proc/kallsyms on the Android image.

Once the system call table address is determined, it is simple to change the system call table entry to the location of a new handler function. The system call table entry is changed when the LKM is loaded using the insmod command. Figure 4.2 contains code that saves the original address of the open() and getdents64() system call table entries and changes them to new function handlers prefixed with "hooked_".



```
// executes when module is loaded with insmod
static int __init start(void)
{
    // save original address
    orig_open = sys_call_table[__NR_open];
    // insert new address
    sys_call_table[__NR_open] = hooked_open;

    // save original address
    orig_getdents64 = sys_call_table[__NR_getdents64];
    // insert new address
    sys_call_table[__NR_getdents64] = hooked_getdents64;

    return 0;
}
```

Figure 4.2 System Call Hook LKM Initialization

In Figure 4.2, the symbol representing the offset of the system call found in unistd.h header (AOSP: kernel/arch/arm/include/asm/unistd.h) refers to the system call table entry. That entry is saved into a global variable and replaced with a new handler function defined in the LKM. The new handler function is called instead of the original system call when it is invoked. Therefore, the handler function must match the declaration of the original function to handle the intercepted parameters properly. The system call table entries can be restored by assigning the saved original addresses. Restoring the system call table to its original state is typically performed when the module is removed with the rmmod command.

This method of targeting system calls and writing functions to alter the return values of system calls can also be used to hide files, processes, ports and modules. The next section describes the implementations of the LKM rootkits that hide these targets. The full source code implementation and description can be obtained available from Dr. Rusty O. Baldwin at the Air Force Institute of Technology (rusty.baldwin@afit.edu).

4.2.2 Hiding a File or Directory with hide_file.ko

The hide_file.ko rootkit hooks the system calls lstat64(), open(), and getdents64() to hide files or directories that are named with a specified substring. The new lstat64() handler function simply checks if the path contains the hidden file constant string. If the string is present, the system call returns a "No such file or directory" signal. Otherwise, the original lstat64() returns. The new open() handler function compares the inode of a specified path to hide and the inode of the requested file path to open. If the two inodes are equal, open() returns "No such file or directory" signal. Otherwise, the original open() is returned. The new getdents64() handler function calls the original system call and copies the returned dirent buffer into kernel space and then iterates through the entries. If an entry contains the specified substring of the intended hidden file, the entry is removed from the buffer and size are changed to reflect the removal. The buffer is then copied back to user space and the size is returned. All these operations hide the files or directories so that the infection is covert from probe-based detection methods.

4.2.3 Hiding a Process with hide_proc.ko

The hide_proc.ko rootkit hides processes that are named with a specified substring by hooking the system calls getdents64() and kill(). The new getdents64() handler function calls the original system call and copies the returned dirent buffer into kernel space and then iterates through the entries. The PID (process id) of the iterated entry is compared to every running task. Once the matching running task is found, the task name is extracted and compared to the specified substring. The matching record is removed from the buffer and size is changed to reflect the removal. The buffer then is copied back to user space and the size is returned. The new kill() handler function compares the passed PID value to every running process to extract the name of the task. If the task name contains the specified substring, the function returns a "No process or process group can be found corresponding to that specified by PID" signal. Otherwise, the original kill() system call returns. All these operations hide processes so that the infection is covert from probe-based detection methods.

4.2.4 Hiding a Module with hide_mod.ko

The hide_mod.ko rootkit hides modules that are named with a specified substring by hooking the system calls delete_module() and read(). The new delete_module() handler function simply checks if the module name passed to the function contains the specified substring. If the substring is present, a “No module by that name exists” signal is returned. Otherwise, the original delete_module() system call returns. The new read() handler function calls the original read to obtain the buffer that will be returned to the user. The function then determines the inode of the current open file and compares that inode to the /proc/modules inode obtained when the module was loaded. If the inodes match and the current running task is lsmod or cat, the buffer is examined. Since the structure of the data in /proc/modules is consistent, the data in the buffer can be modified to make a search easier. Each new line is first changed to a terminating null. This technique allows the code to iterate through the data like an array of character strings. If one of those strings contains the substring of the intended hidden module, the string is removed from the buffer and the size is changed to reflect the removal. The newlines are inserted back into the null terminator positions after iterating through the buffer. The buffer is then copied back to user space and the size is returned. All these operations hide modules so that the infection is covert from probe-based detection methods.

4.2.5 Hiding a Port with hide_port.ko

The hide_port.ko rootkit hooks the read() system call to hide a specified port number. /proc/net/tcp6 is targeted because the open backdoor connection is created over an IPv6 TCP port. The new read() handler function calls the original read to obtain the data that will be returned to the user. The function determines the inode of the current open file and compares that inode to the /proc/net/tcp6 inode obtained when the module was loaded. If the inodes match and the current running task is netstat or cat, the buffer is examined. Since the structure of the data in /proc/net/tcp6 is consistent, the data in the buffer can be modified to make a search easier. Each new line is first changed to a terminating null. This technique allows the code to iterate through the data like an array of character strings. The first string is skipped because that is the line containing the column headers for the data. Each line after that begins with a line number beginning at line 0. The line number and local port number are extracted from each string. If the local port number matches the intended hidden port, the string is removed from the data and the size is changed to reflect the removal. The iteration then begins from the beginning of the data again and corrects the line number if the cat command is used on /proc/net/tcp6. Once the data has been iterated through without any removals, the newlines are inserted back into the null terminator positions. The buffer is then copied back to user space and the size is returned. All these operations hide ports so that the infection is covert from probe-based detection methods.

4.3 Rootkit Evaluation by Detection

The detectability of the covert Android rootkits effectiveness is determined by its effectiveness in hiding the presence of infection data from different detection methods. The covert techniques used are hide files, hide open ports, hide processes, and hide modules.

Two metrics measure the effectiveness of rootkits' stealth on both the emulator and device. The first metric is detection to determine the detectability by traditional detection methods. Four scans using methods discussed in Chapter 3 are used for this metric. The scans are probe-based, integrity-based, signature-based, and heuristic-based method detection. A successful detection by any of these scans is indicated by a "yes" and an infection not detected is indicated by a "no".

The second metric is system behavior when rootkits are present (infected) and not present (clean). The system call hook is intercepting the system call and running more instructions before returning. It is likely that this adds a delay to the execution of the system call. These delays, also described in Appendix A.5, are measured using the strace tool.

4.4 Detection Method Testing Results

Table 4.1 contains the detection method testing results for each of the covert techniques tested on the emulator. The first column lists the configurations tested and each row represents a covert technique implemented in its respective rootkit. The remaining columns to the right indicate the type of detection method tool used against each covert technique rootkit. Probe-based and signature-based did not detect the rootkit infection and thereby did not limit the effectiveness of the rootkits. These results were expected because the rootkits were designed for commands used in the probe-based detection. Signature-based detection methods failed because they only search application folders, SD card files, SMS and contacts. Root privileges cannot be provided to the scanner to search for infections in the system area and the implemented covert techniques rootkits have not been publicly released. The Integrity-based scanner detected 100% of the covert technique rootkits; however, it is dependent on having a trusted source to find infections. The heuristic-based scanner limits the effectiveness of each rootkit configuration except hide_file. The results are not surprising because the tools used to determine an infection for a heuristic scan perform exhaustive collection of system data.

Table 4.1 Emulator Rootkit Detection

	Detection Method			
Configuration	Probe	Integrity	Signature	Hueristic
hide_file	no	yes	no	no
hide_proc	no	yes	no	yes
hide_mod	no	yes	no	yes
hide_port	no	yes	no	yes

The detection method results for each of the covert techniques tested on the device are shown in Table 4.2. The results are similar to the emulator except for the Lookout Security & Antivirus scan against the hide_proc rootkit and the infection data not being detected by the heuristic scanner for the hide_mod rootkit. The freezing during Lookout Security & Antivirus is because the hide_proc rootkit must have unintentionally corrupted the execution of the application. Although this is not helpful for the test results, such a crash could lead a user to wipe the phone to fix the unintended behavior. The wipe would remove the rootkit infection and lead to an ineffective infection. The heuristic scan was not able to find the hidden module on the device because the buffer from the /proc/modules read() system call is constructed differently than that on the emulator. Overall, the integrity and heuristic-based detection methods best limit the effectiveness of the covert rootkits.

Table 4.2 Device Rootkit Detection

	Detection Method			
Configuration	Probe	Integrity	Signature	Hueristic
hide_file	no	yes	no	no
hide_proc	no	yes	no*	yes
hide_mod	no	yes	no	no
hide_port	no	yes	no	yes

**** phone became unresponsive when starting app and rebooted if the rootkit was removed via ADB***

4.5 Behavior Latency Benchmark Results

The measurements of the system call completion times with the rootkit installed (infected) and not installed (clean) on the tested emulator are shown in Table 4.3. The system calls targeted by the rootkit are the only ones timed since the others will not be affected by the system call hooking. The table is organized by the specific test configurations: type of the intended target, the rootkit infection, and the system calls being measured. The clean latencies are placed above the infected latencies to make it easy to compare and do not illustrate the order in which the data was collected. The system call completion times measured five times and the sample mean are listed to the right of each test configuration. All the clean latency means were less than infected except the lstat64() system call.

Table 4.3 Emulator System Call Latencies (in microseconds)

Target	Rootkit	System Call	Run 1	Run 2	Run 3	Run 4	Run 5	Mean
file	none	getdents64	350	435	362	412	483	408.4
	hide_file	getdents64	511	434	392	479	458	454.8
	none	lstat64	167	232	187	202	155	188.6
	hide_file	lstat64	108	147	125	173	167	144
	none	open	232	167	215	174	163	190.2
	hide_file	open	190	257	203	243	233	225.2
proc	none	getdents64	1093	1310	1193	1387	1107	1218
	hide_proc	getdents64	1788	1314	1370	1400	1625	1499.4
	none	kill	109	132	106	124	131	120.4
	hide_proc	kill	180	162	176	156	173	169.4
mod	none	delete_module	142	131	113	126	112	124.8
	hide_mod	delete_module	207	201	194	212	188	200.4
	none	read	151	147	157	148	153	151.2
	hide_mod	read	255	215	226	221	268	237
port	none	read	15597	17248	17013	17960	16536	16870.8
	hide_port	read	18762	18492	18054	17064	17666	18007.6

The time elapsed is calculated from the difference between the beginning and the end wall-clock timestamps. If the process executing the system call is preempted by the scheduler for a longer than average time, the elapsed time can be quite large. For example, Run 5 of clean getdents64() had a large outlier of 2602 microseconds which put the average latency for the test configuration at 832 microseconds. This latency is 313% higher than the average latency indicating it was preempted longer than the other runs. It is also well past the 1.5 IQR from the second quartile, a further indication that it is indeed an outlier. Therefore, it was replaced by collecting a new latency using the command from the Perl script. There were 6 outliers total in the data collected from the emulator. These were all handled the same way.

Figure 4.3 is a box plot comparing the clean verses infected system call completion times on the emulator. As seen in the collected data and calculated means, the infected getdents64() and open() system calls for hide_file tend to take longer to complete than the clean system call while lstat64() opposes that trend. Box plots for all the clean verses infected system call completion times for the tested emulator are available in Appendix B.

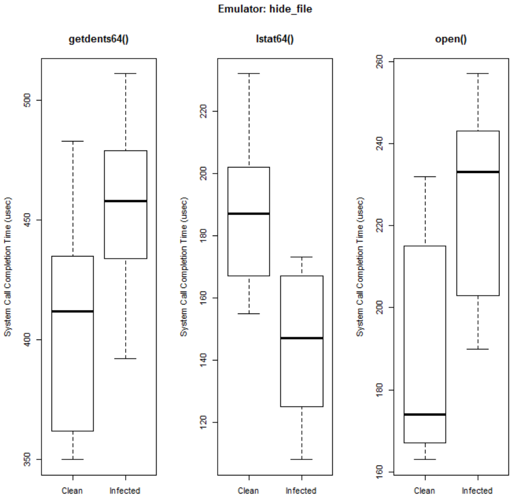


Figure 4.3 Emulator hide_file System Call Latencies Box Plots

Table 4.4 shows the upper and lower bounds of the 95% confidence intervals for the results from the emulator. The table is organized by the specific test configurations: type of the intended target, the rootkit infection, and the system calls being measured. The clean CI are placed above the infected CI to make it easy to compare but do not indicate the order the data was collected. The true population mean is 95% certain to lie between the upper and lower bounds of the confidence interval for each test configuration. As seen previously, the confidence intervals trend higher for infected system call completion times except, again, for lstat64().

Table 4.4 Emulator System Call Latency 95% t-Confidence Interval Bounds (in microseconds)

Target	Rootkit	System Call	Lower	Mean	Upper
file	none	getdents64	340.820	408.4	475.980
	hide_file	getdents64	398.819	454.8	510.781
	none	lstat64	151.035	188.6	226.165
	hide_file	lstat64	109.792	144	178.208
	none	open	151.415	190.2	228.985
	hide_file	open	190.518	225.2	259.882
proc	none	getdents64	1058.994	1218	1377.006
	hide_proc	getdents64	1251.018	1499.4	1747.782
	none	kill	105.228	120.4	135.572
	hide_proc	kill	156.934	169.4	181.866
mod	none	delete_module	109.109	124.8	140.491
	hide_mod	delete_module	188.407	200.4	212.393
	none	read	146.202	151.2	156.198
	hide_mod	read	208.240	237	265.760
port	none	read	15780.623	16870.8	17960.977
	hide_port	read	17171.667	18007.6	18843.533

Table 4.5 shows the results of a single-sided t-test to determine if the difference in the means of the clean verses infected system calls on the emulator. The left column is the target while the remaining right columns are the system calls affected by each rootkit. The null hypothesis is that the clean was less than the infected system call. As expected,

all the system calls except lstat64() were statistically significant meaning that the means of the clean latencies were less than the infected latencies. The clean lstat64() system call was greater than the infected. The actual function invoked by that system call is inspected to determine what causes this difference.

Table 4.5 Emulator Clean vs. Infected System Call Completion Times

	System Call					
Target	open()	lstat64()	getdents64()	kill()	delete_module()	read()
files	LESS	GREATER	LESS	-	-	-
procs	-	-	LESS	LESS	-	-
modules	-	-	-	-	LESS	LESS
ports	-	-	-	-	-	LESS

Table 4.6 shows the difference between the means of the clean and infected system call latencies on the emulator. All show an increase in the infected system call completion times except lstat64() under hide_file. This exception can be attributed to the short amount of code the hooked function runs. In the source code in /kernel/common/fs/stat.c, lstat64() executes more code since it handles more initializing and flag checking. Although, hooking lstat64() results in a shorter execution time, the existence of the difference in means indicates that an infection can still be detected by a specially designed algorithm. Since all the differences are less than 0.1 second (or 100,000 microseconds), the threshold proposed by Nielsen for a user to notice a difference in system performance [Nie93], the latency induced by the covert technique would likely remained unnoticed by the average user.

Table 4.6 Emulator Behavior Difference in Latency Means (in microseconds)

Rootkit	System Call	Clean Mean Latency	Infected Mean Latency	Difference in Latency Means
hide_file	getdents64	408.4	454.8	46.4
	lstat64	188.6	144	-44.6
	open	190.2	225.2	35
hide_proc	getdents64	1218	1499.4	281.4
	kill	120.4	169.4	49
hide_mod	delete_module	124.8	200.4	75.6
	read	151.2	237	85.8
hide_port	read	16870.8	18007.6	1136.8

Table 4.7 lists the measurement of the system call completion times with the rootkit installed (infected) and not installed (clean) on the device tested, Samsung Galaxy Nexus. The table is organized identically to the emulator data in Table 4.3. The measurements were taken on the device to show the real-world performance of the covert technique rootkits. The latencies measured for the device do not reflect the same range or magnitude as the emulator. However, the difference in means caused by the additional instructions added by the system call hook is evident as it was on the emulator.

Table 4.7 Device Behavior Latency Data (in microseconds)

Target	Rootkit	System Call	Run 1	Run 2	Run 3	Run 4	Run 5	Mean
file	none	getdents64	2346	2503	2165	2471	2230	2343
	hide_file	getdents64	2319	2382	2442	2563	2748	2490.8
	none	lstat64	274	428	305	335	458	360
	hide_file	lstat64	152	184	214	244	275	213.8
	none	open	396	214	244	275	305	286.8
	hide_file	open	763	793	824	916	1099	879
proc	none	getdents64	672	701	763	793	824	750.6
	hide_proc	getdents64	1373	1402	1465	1494	1556	1458
	none	kill	30	31	61	91	92	61
	hide_proc	kill	152	183	213	244	274	213.2
mod	none	delete_module	91	61	31	30	92	61
	hide_mod	delete_module	152	183	214	244	275	213.6
	none	read	91	61	31	30	92	61
	hide_mod	read	244	275	335	306	336	299.2
port	none	read	946	976	1007	1037	1068	1006.8
	hide_port	read	2258	2289	2319	2350	2411	2325.4

The measured system call completion times also suffered from outliers. As with the emulator, outliers were replaced by collecting a new latency measurement using the specific command from the Perl script. There were 4 outliers total in the data collected from the device. These were handled the same way as described when replacing outliers with the emulator data.

Figure 4.4 shows a box plot that illustrates the comparison for the clean verses infected system call completion times on the tested device. As expected, the infected getdents64() is shown to tend to take longer to complete the clean system call. Box plots for all the clean verses infected system call completion times on the tested device are available in Appendix B.

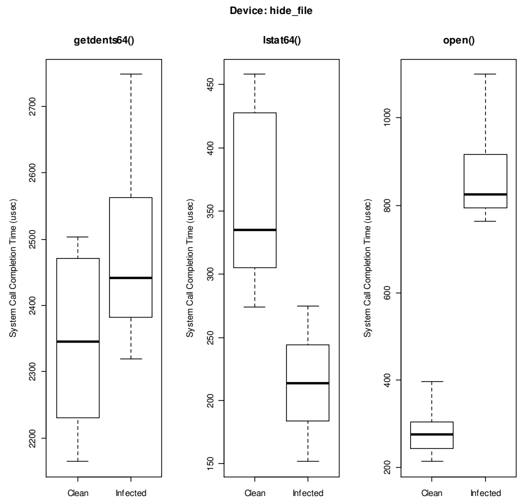


Figure 4.4 Device hide_file System Call Latencies Box Plots

Table 4.8 shows the upper and lower bounds of the 95% confidence intervals on the tested device. The table is organized identically to the emulator data in Table 4.4. The true population mean is 95% certain to lie between the upper and lower bounds of the confidence interval for each test configuration. Again, all the clean latency means were less than infected except for the lstat64() system call.

Table 4.8 Device Behavior Latency 95% t-Confidence Interval Bounds (in microseconds)

Target	Rootkit	System Call	Lower	Mean	Upper
file	none	getdents64	2160.465	2343	2525.535
	hide_file	getdents64	2280.175	2490.8	2701.425
	none	lstat64	261.301	360	458.699
	hide_file	lstat64	153.721	213.8	273.879
	none	open	200.046	286.8	373.554
	hide_file	open	710.520	879	1047.480
proc	none	getdents64	672.126	750.6	829.074
	hide_proc	getdents64	1367.371	1458	1548.629
	none	kill	23.124	61	98.876
	hide_proc	kill	153.320	213.2	273.080
mod	none	delete_module	23.124	61	98.876
	hide_mod	delete_module	153.327	213.6	273.873
	none	read	23.124	61	98.876
	hide_mod	read	249.865	299.2	348.535
port	none	read	946.920	1006.8	1066.680
	hide_port	read	2252.360	2325.4	2398.440

Table 4.9 shows the results of a single-sided t-test to determine if the difference in the means of the clean verses infected system calls on the tested device. The table is organized identically to the emulator data in Table 4.5. The null hypothesis tested that the clean was again less than the infected system call. As expected, all the system calls except lstat64() were statistically significant meaning that the means of the clean latencies were less than the infected latencies. The clean lstat64() system call was again determined to be greater than the infected.

Table 4.9 Device Clean vs. Infected System Call Completion Times

	System Call					
Target	open()	lstat64()	getdents64()	kill()	delete_module()	read()
files	LESS	GREATER	LESS	-	-	-
procs	-	-	LESS	LESS	-	-
modules	-	-	-	-	LESS	LESS
ports	-	-	-	-	-	LESS

Table 4.10 shows the difference between the means of the clean and infected system call latencies. The table is organized identically to the emulator data in Table 4.6. As seen on the tested emulator, all the cases show an increase in the infected system call completion times except lstat64() under hide_file. This exception is attributed to the short amount of code the new handler function runs compared to the original system call as explained previously on the emulator. As before, the existence of the difference in means indicates that an infection can still be detected by an algorithm but may be unnoticed by a user because the differences are less than 0.1 second.

Table 4.10 Device Behavior Difference in Latency Means (in microseconds)

Rootkit	System Call	Clean Mean Latency	Infected Mean Latency	Difference in Latency Means
hide_file	getdents64	2343	2490.8	147.8
	lstat64	360	213.8	-146.2
	open	286.8	879	592.2
hide_proc	getdents64	750.6	1458	707.4
	kill	61	213.2	152.2
hide_mod	delete_module	61	213.6	152.6
	read	61	299.2	238.2
hide_port	read	1006.8	2325.4	1318.6

4.6 Summary

This chapter presents the evaluation technique used to determine the effectiveness of the rootkits. The results of the covert techniques tested against detection methods are presented and statistical analysis is performed on the data provided for the behavioral-based detection. Both the evaluations were performed on the emulator and Samsung Galaxy Nexus device.

The results of the detection method testing showed that while conventional probe-based and signature based-detection did not limit the effectiveness of the tested rootkits, integrity-based and heuristic-based were able to detect the presence of an infection thereby limiting the stealth and the effectiveness of the rootkit. The results of behavior latency benchmark showed that the difference in means is statistically significant and could be used as a basis for detecting system call hooking rootkit infections. As expected, the results were similar for both the emulator and the device. Chapter V addresses accomplishments of this research and proposes future work for kernel level rootkit design and detection on Android.

V. Conclusions

5.1 Research Accomplishments

This research defined software attacks from a technical perspective, designed kernel level rootkits with covert functionality on Android mobile devices, and evaluated the effectiveness of the rootkits by scanning an infected system with detection methods and benchmarking the behavior of the covert techniques used.

A mobile device, such as a Smartphone, can carry multiple connections from cellular networks, wireless Bluetooth, the Internet (via Wi-Fi), USB and other peripherals. Smartphone users can access email, social networks, and banking, all from their mobile device. Information security becomes an immediate concern with this amount of sensitive data being handled on these potentially unsecured devices. This research demonstrates that the Android software stack is not robust enough to be fully trusted since the kernel can be directly manipulated to hide an attacker's presence. The kernel level rootkits tested against detection methods were designed for the Linux kernel used by the latest release, Android 4.0 Ice Cream Sandwich. These rootkits focused on covert techniques to hide the presence of data used by an attacker to infect a mobile device. Detection methods were used to measure the effectiveness of the kernel level rootkits. The effect of hooking system calls have on performance behavior was analyzed in depth.

A rootkit is most effective when its presence cannot be detected. The most popular free Antivirus, Lookout Security & Antivirus, and the built-in system commands for Android were not able to detect the presence of an infection hidden by the covert rootkits. The Lookout Security & Antivirus has signatures for attacks that take advantage of the native Linux kernel but are ineffective since they only look for a signature of the data unpacked on to the device from the application.

Integrity-based detection methods were 100% effective against the covert techniques but rely on installing prior to the infection or have a trusted source. Heuristic-based detection also shows signs of weakening the effectiveness of the kernel level rootkits. However, the strace command can be used to target the processes of both these methods to implement new hooks to block detection.

The behavioral-based detection method analysis showed that the rootkits tested can have a noticeable impact on the latency of a system call completion time. The magnitude is too small for a user to notice but the differences could be discovered by another algorithm. However, this method again requires an uncompromised state to identify anomalies [Sha12].

5.2 Research Impact

Rootkits are a real threat on mobile devices, especially ones that use the Android platform. While the installation method used to test this requires a custom setup not common to all phones, it may be possible to exploit and install this rootkit using exploits [Ces98]. Keeping the device updated also might not be possible because of fragmentation in Android, therefore, the user is left to depend on the manufacturer for security. This security model is unsustainable and more research into Android rootkits and detection needs to be performed because the problem is only going to get worse.

5.3 Future Research Areas

Kernel level rootkits are a real threat on the Android operating system even though as yet there are not many reported cases. Sophisticated attackers can use rootkits to maintain long periods of undetected presence on an Android mobile device. The proof of concept rootkits and the detection methods presented can be extended for new research in both attack and defense.

Integrity and heuristic detection methods limited the effectiveness of the kernel level rootkits during testing. The rootkit could be more effective by evading integrity detection using techniques presented in [You11] and [You12] since the system call table structure is not directly modified. Heuristic detection can be evaded by designing the covert technique to thwart detection tools like unhide-tcp and skdet. Research into exploring different kernel level rootkits is always expanding and designing rootkits for the post-PC era devices should not be ignored.

Behavioral detection also limited a rootkit's covert effectiveness by detecting differences in the mean of the system call latencies. However, its reliance on having an uncompromised state to compare the measured latencies needs to be removed. Algorithms designed to monitor system behavior for performance anomalies should be investigated further. This additional detection measure could increase the effectiveness of an IDS for Android or other operating systems.

Beyond the stealth and detection, research methods to hook into the Android framework from the kernel level rootkit should be developed. Although the personal data such as contacts, text messages, and application data can be found by searching known locations, new ways of getting to this data in a more abstract and Android framework friendly way would make data exfiltration and remote control of an Android mobile device much easier. Removing the need to reverse engineer a target application for that application's data would have a long lasting impact on the security research for Android.

The rootkits were designed for the Android operating system but should also be easily portable by compiling for a targeted Linux kernel. As Android and Linux continue to be used across more devices and platforms, the need for security research increases since the inherent vulnerabilities are not going away. The success of implementing a kernel level rootkit in a mobile device environment demonstrates that additional security measures should be implemented on Android and its Linux kernel.

Appendix A. Detection Method Implementations

This appendix contains the detection method implementations including description and scripts.

A.1 Probe-based Detection

Probe-based detection identifies signs of infection by forensic analysis using the system commands: ls, cat, ps, kill, lsmod, and rmmod. The function of the system commands are described below:

- List Files (ls) – lists the files in a current or specified directory.
- Catenate File (cat) – writes the contents of each file specified in the standard output.
- List Processes (ps) – lists the processes currently running.
- Kill Process (kill) – sends a signal to a process. The default signal sent is the termination signal but the command can also send other specified signals. This extra functionality is not used in this research.
- List Modules (lsmod) – lists the modules currently installed.
- Remove Module (rmmod) – removes a specified module.
- List Network Connections (netstat) – lists all incoming and outgoing network connections, routing tables, and network interface statistics.

Table A.1 is a list of the system commands used to search for infection traces during probe-based detection.

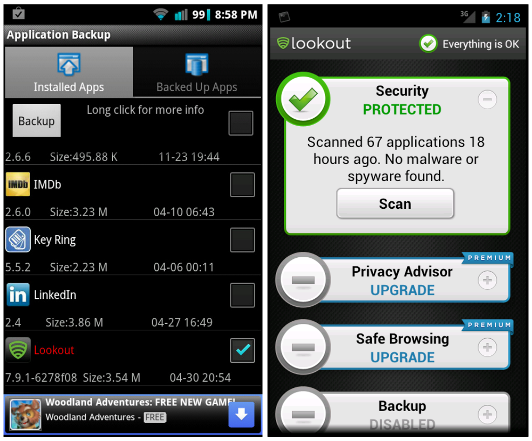
Table A.1 Probe-based Detection Method Commands

Infection Traces	System Commands Used
Files	ls, ls <filename>, ls –l, ls –l <filename>, cat <filename>
Processes	ps, ps <process_name>, ls /proc/, kill <pid>
Modules	lsmod, cat /proc/modules, rmmod <module_name>
Ports	netstat, cat /proc/net/tcp6

A.2 Signature-based Detection

The signature-based malware detection tool Lookout Security & Antivirus [Loo12] is scanned against each covert technique on both Android platforms. Anti-virus (AV) tools like Lookout Security & Antivirus are ineffective in detecting all attacks because they typically only scan application folders, SD card files, SMS and contacts [Far11]. Since a typical Android device does not have root privileges by default, the AV applications cannot search for infections in the system area that is the most targeted and vulnerable.

The Lookout Security & Antivirus application is available in the Google Play Store. The application was downloaded from the Google Play Store on another device and the .apk installer file was copied using Astro File Manager [Gop12] from that device. The .apk is then installed over the ADB with the command ‘adb install Lookout-70800.apk’ [God12]. The security scan was then initialized when running Lookout Security & Antivirus. Figure A.1 shows the interface of Astro File Manager and Lookout Security & Antivirus.



8:58 PM

Application Backup

Installed Apps | Backed Up Apps

Backup Long click for more info

2.6.6 Size:495.88 K 11-23 19:44

IMDb

2.6.0 Size:3.23 M 04-10 06:43

Key Ring

5.5.2 Size:2.23 M 04-06 00:11

LinkedIn

2.4 Size:3.86 M 04-27 16:49

Lookout ☑

7.9.1-6278f08 Size:3.54 M 04-30 20:54

Woodland Adventures: FREE NEW GAME!

Woodland Adventures - FREE

2:18

lookout Everything is OK

Security
PROTECTED

Scanned 67 applications 18 hours ago. No malware or spyware found.

Scan

PREMIUM
Privacy Advisor
UPGRADE

PREMIUM
Safe Browsing
UPGRADE

Backup
DISABLED

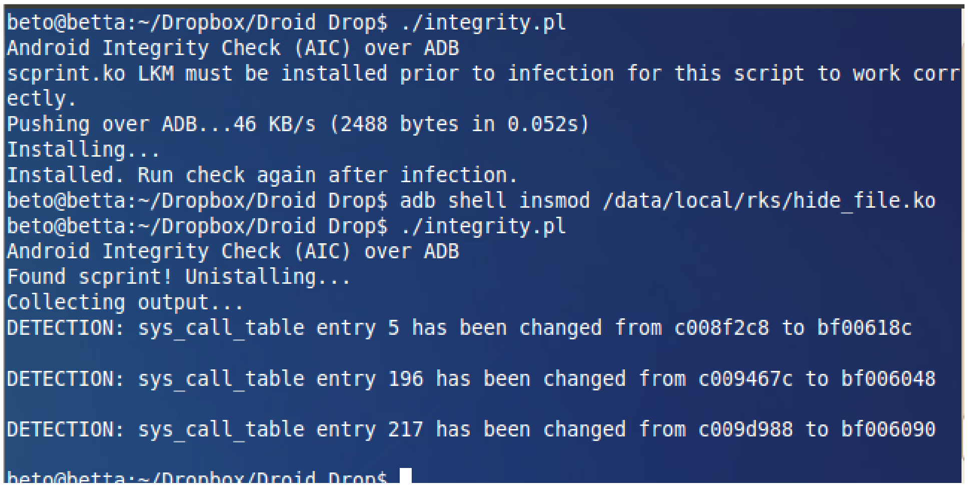
Figure A.1 Astro File Manager Backup, Lookout Security & Antivirus Menu

A.3 Integrity-based Detection

Rootkits often take control over of a kernel by modifying static system structures. Integrity-based detection compares data of a trusted source with potentially infected data to find differences that identify a possible attack. In the case of the system call hooking technique, used by the rootkits in this research, the rootkit overwrites the address of the system call table entry so that the malicious code in a new handler function is executed. The addresses in the system call entries are supposed to be permanent and should not change even after a reboot of the operating system. Therefore, the system structures need to be validated in a trusted state before searching for changes from an attacker. The trusted source for a system call table can be found in the System.map file if it is available.

Android devices typically do not have the System.map file available. Therefore, the addresses of the system call table entries can be determined by scanning kernel memory using an LKM [Bur10]. The system call table address can be found the same way as described in Section 4.2.1. When the LKM is installed, each system call table entry is saved into a copy array. Upon removal of the LKM, each system call table entry is compared to the copy array. A detection message prints to the message buffer of the kernel if a difference is found and can be read using the dmesg command.

During testing, the LKM used to test integrity was named scprint. The scprint module was compiled for the specific kernel being tested, the same as the rootkits. Prior to the rootkit infection, scprint was installed. This way the system call table is in a trusted state. The specific covert technique rootkit is installed which hooks its respective system calls. The scprint module is then uninstalled and the proper detection messages are printed to the message buffer of the kernel. The process of downloading, installing, and uninstalling the scprint module was automated using a Perl script named integrity.pl. Figure A.2 shows a successful detection by the integrity-based detection.



```
beto@betta:~/Dropbox/Droid Drop$ ./integrity.pl
Android Integrity Check (AIC) over ADB
scprint.ko LKM must be installed prior to infection for this script to work corr
ectly.
Pushing over ADB...46 KB/s (2488 bytes in 0.052s)
Installing...
Installed. Run check again after infection.
beto@betta:~/Dropbox/Droid Drop$ adb shell insmod /data/local/rks/hide_file.ko
beto@betta:~/Dropbox/Droid Drop$ ./integrity.pl
Android Integrity Check (AIC) over ADB
Found scprint! Unistalling...
Collecting output...
DETECTION: sys_call_table entry 5 has been changed from c008f2c8 to bf00618c

DETECTION: sys_call_table entry 196 has been changed from c009467c to bf006048

DETECTION: sys_call_table entry 217 has been changed from c009d988 to bf006090

beto@betta:~/Dropbox/Droid Drop$
```

Figure A.2 Integrity Check Output Pre and Post Infection

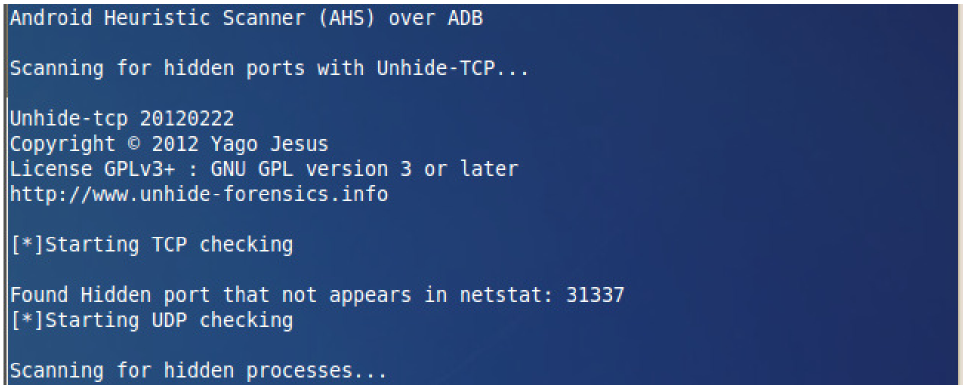
The code for integrity.pl:

```
#!/usr/bin/perl -w
# Android System Call Table Inegrity Check
# Bobby Brodbeck, AFIT, June 2012

use 5.10.0;

print "Android Integrity Check (AIC) over ADB \n";

my $lsmod_out = `adb shell lsmod`;

if ($lsmod_out =~ /scprint/)
{
        print "Found scprint! Unistalling...\n";
        `adb shell rmmod scprint`;
        print "Collecting output...\n";
        my $dmesg_out = `adb shell dmesg | tail -10`;
        # print "Output:\n$dmesg_out\n";
        my @detects = ($dmesg_out =~ /(DETECTION:(.)+)\n/g);
        for($i = 0; $i < @detects; $i++)
        {
                print "$detects[$i]\n";
        }
}
else
{
        print "scprint.ko LKM must be installed prior to infection for this script to work
correctly.\n";
        print "Pushing over ADB...";
        `adb push ~/mods/scprint/scprint.ko /data/local/`;
        print "Installing...\n";
        `adb shell insmod /data/local/scprint.ko`;
        print "Installed. Run check again after infection.\n";
}

############## EOF: integrity.pl ##############
```

A.4 Heuristic-based Detection

Rootkits take control of execution flow to hide data; however, there are other ways to find data that is hidden. Heuristic-based detection identifies anomalies by comparing the returned data from different functions targeting the same source. The heuristic scan was designed custom for detecting the rootkits tested in this research. They used a combination of tools created to detect Linux rootkits and commands already available on the Android kernel. The heuristic scan searches for anomalies to identify hidden files, processes, modules, and ports.

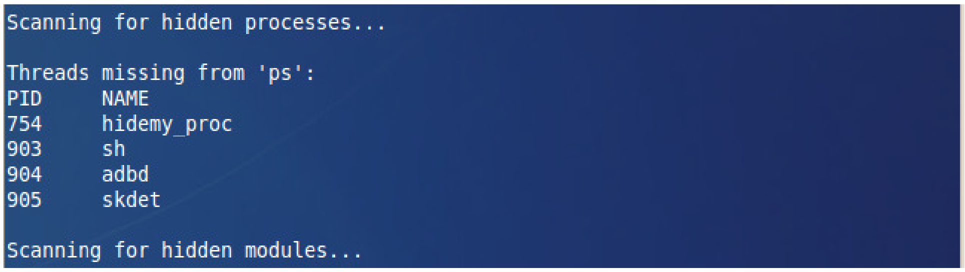
The heuristic scan is implemented by a Perl script that runs commands over ADB and analyzes the output. The analysis is printed to the terminal of the machine running ADB. The heuristic scan starts by running unhide-tcp to find hidden ports. The Rootkit Hunter Project [Boe12] uses unhide-tcp to enhance its detection capabilities. Unhide-TCP identifies TCP/UDP ports that are listening but not listed by the netstat command by brute forcing all TCP/UDP ports available [http://sourceforge.net/projects/unhide/files/]. If any hidden ports are detected, the results are printed to the output. Figure A.3 shows the output when a hidden port is detected by unhide-tcp.



```
Android Heuristic Scanner (AHS) over ADB

Scanning for hidden ports with Unhide-TCP...

Unhide-tcp 20120222
Copyright © 2012 Yago Jesus
License GPLv3+ : GNU GPL version 3 or later
http://www.unhide-forensics.info

[*]Starting TCP checking

Found Hidden port that not appears in netstat: 31337
[*]Starting UDP checking

Scanning for hidden processes...
```

Figure A.3 Port Heuristic Detection Output

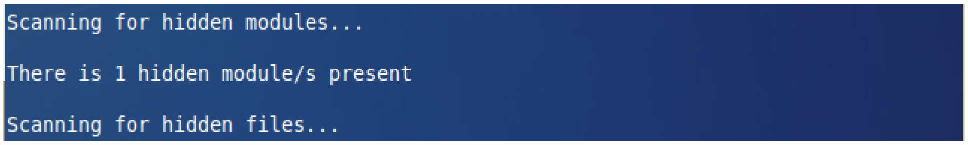
The heuristic scanner identifies hidden processes by comparing the output of the ps –T command and skdet tool. The ps –T command returns all the running task threads on the device. The ps command from the BusyBox toolkit is used. Skdet is designed to detect rootkits such as SucKIT, adore-ng, trojaned files, and more. The Rootkit Hunter Project can also take advantage of skdet to enhance its detection capabilities. The heuristic scanner uses skdet for an alternative source of running task threads by using the –c option. The difference of the two sets of data PIDs is then printed out. The operator running the heuristic scanner must have some extra knowledge to understand the printed difference. Three of the processes are unique to the ps command and skdet tool because of the shell over ADB. Therefore they can be ignored in the results. These processes found during testing were sh (shell), adbd (ADB daemon), and the skdet tool. Figure A.4 shows the output of the heuristic scanner when a hidden process is detected (hidemy_proc).



```
Scanning for hidden processes...

Threads missing from 'ps':
PID     NAME
754     hidemy_proc
903     sh
904     adbd
905     skdet

Scanning for hidden modules...
```

Figure A.4 Process Heuristic Detection Output

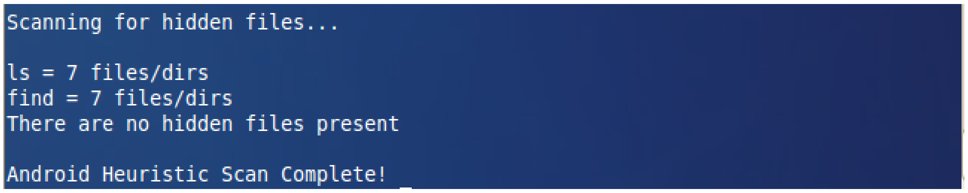
Hidden modules are identified by comparing the number of lines in the lsmod command output verses the line count of /proc/modules. The lsmod command is built into the Android Linux kernel but the wc command is found with the BusyBox toolkit. If the line counts are not equal, the difference is printed out. The lsmod command is expected to have the smaller amount of lines since it is the command targeted by the system call hook. Figure A.5 shows the output of the heuristic scanner when a hidden module is detected.



```
Scanning for hidden modules...

There is 1 hidden module/s present

Scanning for hidden files...
```

Figure A.5 Module Heuristic Detection Output

Hidden files are identified by comparing the number of lines in the ls command output verses the number of lines in the output from ‘find <directory_name> -maxdepth 1’. The ls command is built into the Android Linux kernel but the find command is found with the BusyBox toolkit. The ‘maxdepth’ option dictates the amount of levels to descend from the directory specified in the command line arguments. One line is subtracted from the results of the find command because it will include a line for the target directory. If the line counts are not equal, the difference is printed out. The ls command is expected to have the smaller amount of lines since it is the command targeted by the system call hook. Figure A.6 shows the output of the heuristic scanner when a hidden file or directory is detected.



```
Scanning for hidden files...

ls = 7 files/dirs
find = 7 files/dirs
There are no hidden files present

Android Heuristic Scan Complete! _
```

Figure A.6 File/Directory Heuristic Detection Output

The code for heuristic.pl

```
#!/usr/bin/perl -w
# Android heuristic scanning
# Bobby Brodbeck, AFIT, June 2012

use 5.10.0;

print "Android Heuristic Scanner (AHS) over ADB \n\n";
```

```
my $prog;

# HIDDEN PORT SCANNING ################
print "Scanning for hidden ports with Unhide-TCP...\n\n";

# RUN UNHIDE-TCP AND PRINT OUTPUT
$prog = "/data/local/unhide-tcp";
my $find_ports = `adb shell $prog`;
printf("%s\n", $find_ports);

# HIDDEN PROCESS SCANNING #############
print "Scanning for hidden processes...\n\n";

# GET ALL THREADS PIDS PROVIDED TO PS
$prog = "/data/local/busybox ps -T";
my $ps_out = `adb shell $prog`;

# remove first line
$ps_out = substr($ps_out, (index($ps_out,"\n")+1));
my @ps_pids = split /\s+[0-9]+\s+[0-9]+[:][0-9]+\s+.*\R/, $ps_out;

# trim whitespace
foreach (@ps_pids)
{
        $_ =~ s/^\s+//;
        $_ =~ s/\s+$//;
}

# for($i = 0; $i < 5; $i++)
# {
        # printf ("ps[%i] = %s\n", $i, $ps_pids[$i]); #Prints ith first element
# }

# GET ALL THREADS PIDS ATTAINED BY SKDET
$prog = "/data/local/skdet -c";
my $skdet_out = `adb shell $prog`;

my @skdet_pids = split /\s+[\b]+\s.*\R/, $skdet_out;
my @skdet_pnames = split /.*[\b]+\s+/, $skdet_out;
chomp(@skdet_pnames);

# for($i = 0; $i < 5; $i++)
# {
        # printf ("skdet[%i] = %s\t%s\n", $i,
                # $skdet_pids[$i], $skdet_pnames[$i+1]); #Prints ith first element
# }

# GET THE DIFFERENCE OF THE TWO SETS
%ps_pids = map {$_=>1} @ps_pids;
my @absent_pids = grep(!defined $ps_pids{$_}, @skdet_pids);

# foreach(@absent_pids)
# {
        # print $_."\n";
# }

# GET INDEX OF THREADS MISSING FROM PS
my @absent_index;
for($i = 0; $i < @absent_pids; $i++)
{
        @absent_index[$i] = grep{ $skdet_pids[$_] == $absent_pids[$i]} 0 .. $#skdet_pids;
}

# PRINT OUT THE NAMES OF THE THREADS MISSING FROM PS
print "Threads missing from 'ps':\nPID\tNAME\n";
for($i = 0; $i < @absent_index; $i++)
{
        printf("%s\t%s\n",                                   $skdet_pids[$absent_index[$i]],
$skdet_pnames[$absent_index[$i]+1]);
}
print "\n";

# HIDDEN MODULE SCANNING ##############
print "Scanning for hidden modules...\n\n";

# GET LSMOD OUTPUT AND COUNT LINES
$prog = "lsmod";
my $lsmod_out = `adb shell $prog`;
my $lsmod_lines = $lsmod_out =~ tr/\n//;

# printf("lsmod = %i modules\n", $lsmod_lines);

# GET LINE COUNT FROM MODULES FILE
$prog = "/data/local/busybox wc -l /proc/modules";
my $wcl_out = `adb shell $prog`;
$wcl_out =~ s/[^0-9]//g;

# printf("wc -l = %i modules\n", $wcl_out);

# COMPARE VALUES
if($wcl_out != $lsmod_lines)
{
        my $hidden_mods = $wcl_out - $lsmod_lines;
        printf("There is %i hidden module/s present\n\n", $hidden_mods);
}
else
{
        printf("There are no hidden modules present\n\n");
}

# HIDDEN FILE SCANNING ################
print "Scanning for hidden files...\n\n";

$prog = "ls /sdcard/ -a";
my $ls_out = `adb shell $prog`;
my $ls_lines = $ls_out =~ tr/\n//;

printf("ls = %i files/dirs\n", $ls_lines);

$prog = "/data/local/busybox find /sdcard/ -maxdepth 1";
my $find_out = `adb shell $prog`;
my $find_lines = $find_out =~ tr/\n//;
$find_lines--; # for the parent dir

printf("find = %i files/dirs\n", $find_lines);

if($find_lines != $ls_lines)
{
        my $hidden_files = $find_lines - $ls_lines;
        printf("There is %i hidden file/s present\n\n", $hidden_files);
}
else
{
        printf("There are no hidden files present\n\n");
}

# COMPLETE ############################
print "Android Heuristic Scan Complete!\n"

############## EOF: heuristic.pl ##############
```

A.5 Behavioral-based Detection

Behavioral-based detection deduces a rootkit infection by monitoring kernel level execution to identify anomalies in performance. A rootkit employing system call hooking can potentially add a delay to a system call to complete execution because the additional hook code is executed before returning to the user space program. This delay is the most noticeable to the end user because slow execution hinders productivity. The annoyance caused by the delay may lead the user to wipe the mobile device regardless if there is a rootkit presence detected.

Behavioral-based detection measures latencies of uninfected or clean and infected system calls using the strace command. The system calls measured are only those that have their execution intercepted by a system call. Table A.2 lists these infected system calls for each rootkit. Each rootkit has a Perl script that captures the system call completion time of the target strace output. These scripts are named sc_file.pl, sc_proc.pl, sc_mod.pl, and sc_port.pl.

Table A.2 System Calls Infected by Rootkit

Rootkit	Infected System Calls
hide_file.ko	getdents64(), lstat64(), open()
hide_proc.ko	getdents64(), kill()
hide_mod.ko	read(), delete_module()
hide_port.ko	read()

The strace command shows the time spent in system calls when the –T option is set as an argument to the command. The time elapsed is calculated from the difference between the beginning and the end wall-clock timestamps. The –e option is set to print only relevant system calls by using the expression “trace=”. Following the strace command and options, the actual targeted process command is stated. These commands were chosen as direct targets of the rootkit and the most straightforward output to parse with the Perl script. Table A.3 lists all the strace commands used by the Perl scripts.

Table A.3 strace Commands Used to Measure System Call Completions

Rootkit	Measured System Call	strace Command
hide_file.ko	getdents64()	strace -T -e trace=getdents64 ls -a /sdcard/
hide_file.ko	lstat64()	strace -T -e trace=lstat64 ls -l /sdcard/hidemy.txt
hide_file.ko	open()	strace -T -e trace=open cat /sdcard/hidemy.txt
hide_proc.ko	getdents64()	strace -T -e trace=getdents64 ps hidemy_proc
hide_proc.ko	kill()	strace -T -e trace=kill kill <PID>
hide_mod.ko	read()	strace -T -e trace=read lsmod
hide_mod.ko	delete_module()	strace -T -e trace=delete_module rmmod hidemy_mod
hide_port.ko	read()	strace -T -e trace=open,read netstat

The code for sc_file.pl:

```
#!/usr/bin/perl -w
# Android system call traces for hide_file.ko
# Bobby Brodbeck, AFIT, June 2012

use 5.10.0;

# open output file
$fout = "strace_file.txt";
open(OUT, ">>$fout") || die("This file will not open!\n");


print OUT "Run\tgetdents64\tlstat64\topen\n";

for($i = 0; $i < 5; )
{
        $i++;
        print OUT "$i\t";

        # ls -a <dir of hidden file> system call times
        my $lsa_out = `adb shell /data/local/strace -T -e trace=getdents64 ls -a
/sdcard/`;
        # print OUT "$lsa_out\n";
        my @lsa_times = ($lsa_out =~ /<(\d+.\d+)>/g);
        # print "getdents64($i): @lsa_times\n";
        # sum getdents calls
        $getdents_total = 0;
        $getdents_total += $_ for @lsa_times;
        printf(OUT "%f\t", $getdents_total);

        # ls -l <dir of hidden file> system call times
        my $lsl_out = `adb shell /data/local/strace -T -e trace=lstat64 ls -l
/sdcard/hidemy.txt`;
        # print OUT "$lsl_out\n";
        my @lsl_times = ($lsl_out =~ /<(\d+.\d+)>/g);
        # print "lstat64($i): @lsl_times\n";
        # sum lstat calls
        $lstat_total = 0;
        $lstat_total += $_ for @lsl_times;
        printf OUT "%f\t", $lstat_total;

        # cat <hidden file> system call times
        # print OUT "STRACE OUTPUT FOR 'cat'\n";
        my   $cat_out   =   `adb   shell   /data/local/strace   -T   -e   trace=open   cat
/sdcard/hidemy.txt`;
        # print OUT "$cat_out\n";
        $cat_out =~ /open\("\/sdcard\/hidemy.txt".+<(\d+.\d+)>/;
        printf OUT "%f\n", $1;

        sleep(1);
}

close OUT;

############## EOF: sc_file.pl ##############
```

The code for sc_proc.pl:

```
#!/usr/bin/perl -w
# Android system call traces for hide_proc.ko
# Bobby Brodbeck, AFIT, June 2012

use 5.10.0;

# open output file
$fout = "strace_file.txt";
open(OUT, ">>$fout") || die("This file will not open!\n");

# ps <proc name> system call times
my $ps_out = `adb shell /data/local/strace -T -e trace=getdents64 ps hidemy_proc`;
# print "$ps_out\n";
my @ps_times = ($ps_out =~ /<(\d+.\d+)>/g);
#print "@ps_times\n";
$getdents_total = 0;
$getdents_total += $_ for @ps_times;
printf(OUT "%f\t", $getdents_total);

# use skdet to get pid
my $skdet_out = `adb shell /data/local/skdet -c`;
# print "$skdet_out\n";

# get thread names and pids
my @skdet_pids = split /\s+[\b]+\s.*\R/, $skdet_out;
my @skdet_pnames = split /.*[\b]+\s+/, $skdet_out;
chomp(@skdet_pnames);

# remove carriage return
foreach(@skdet_pnames)
{
        $_=~ s/\r|\n//g;       # remove /r or /n
        #$_ =~ s/(.)/sprintf("%x",ord($1))/eg;
        # print "$_\n";
        # print "$string\n";
}

# get pid of hidden proc
my $pid = -1;
for($i = 0; $i <@skdet_pnames; $i++)
{
        # print "$string = $skdet_pnames[$i]\n"; #Prints ith first element
        if("hidemy_proc" eq $skdet_pnames[$i] )
        {
                $pid = $skdet_pids[$i - 1];
                last;
        }
}
```

```
# kill <pid> system call times
if ($pid > 0)
{
        # $pid = $skdet_pids[$found_index-1];
        my $kill_out = `adb shell /data/local/strace -T -e trace=kill kill $pid`;
        # print "$kill_out\n";
        $kill_out =~ /<(\d+.\d+)>/;
        printf OUT "%f\n", $1;
}
else
{
        print "Could not test kill()\n";
}

close OUT;

############## EOF: sc_proc.pl ##############
```

The code for sc_mod.pl:

```
#!/usr/bin/perl -w
# Android system call traces for hide_mod.ko
# Bobby Brodbeck, AFIT, June 2012

use 5.10.0;

# open output file
$fout = "strace_file.txt";
open(OUT, ">>$fout") || die("This file will not open!\n");

print OUT "Run\tread\tdelete_module\n";

for($i = 0; $i < 5; )
{
        $i++;
        print OUT "$i\t";

        # lsmod system call times
        my $lsmod_out = `adb shell /data/local/strace -T -e trace=read lsmod`;
        # print "$lsmod_out\n";
        # uncomment line below for clean: hello always first
        #$lsmod_out =~ /read\(3, "hello.+<(\d+.\d+)>/;
        # uncomment line below for infected: hide_mod always first
        $lsmod_out =~ /read\(3, "hide_mod.+<(\d+.\d+)>/;
        printf OUT "%f\t", $1;

        # rmmod <hidden mod> system call times
        my  $rmmod_out  =  `adb  shell  /data/local/strace  -T  -e  trace=delete_module  rmmod
hidemy_mod`;
        # print  "$rmmod_out\n";
        $rmmod_out =~ /<(\d+.\d+)>/;
        printf OUT "%f\n", $1;
}

close OUT;

############## EOF: sc_mod.pl ##############
```

The code for sc_port.pl:

```
#!/usr/bin/perl -w
# Android system call traces for hide_port.ko
# Bobby Brodbeck, AFIT, June 2012

use 5.10.0;

# open output file
$fout = "strace_file.txt";
open(OUT, ">>$fout") || die("This file will not open!\n");

print OUT "Run\tread\n";

for($i = 0; $i < 5; )
{
        $i++;
        print OUT "$i\t";

        # netstat system call times
        my $netstat_out = `adb shell /data/local/strace -T -e trace=open,read netstat`;
        # print "$netstat_out\n\n";

        # separate lines for parsing
        @nslines = split(m/\R/,$netstat_out);
        chomp(@nslines);

        # find boundaries of read calls
        my $start = 0;
        my $end = scalar(@nslines);
        my $tcp6_flag = 0;
        for($j = 0; $j < $end; $j++)
        {
                # print "$j: $nslines[$j]\n";

                if($nslines[$j] =~ /open\("\/proc\/net\/tcp6"/)
                {
                        $start = $j+1;
                        $tcp6_flag = 1;        # stop looking for tcp6
                }

                if($tcp6_flag && $j > $start && ($nslines[$j] =~ /open/))
                {
                        $end = $j;             # grab and go on
                }
        }


        # collect read completion times
        my @reads;
        for($j = $start; $j < $end; $j++)
        {
                if($nslines[$j] =~ /<(\d+.\d+)>/g)
                {
                        push(@reads, $1);
                        # print "$j: $nslines[$j]\n";
                }
        }

        # sum completion times
        # print "@reads\n";
        my $read_total = 0;
        $read_total += $_ for @reads;
        printf(OUT "%f\n", $read_total);
}

close OUT;

############## EOF: sc_port.pl ##############
```

Appendix B. System Call Latency Box Plots

This appendix contains all the clean verses infected system call latencies box plots for the emulator and device. These box plots are discussed in detail in Section 4.5.

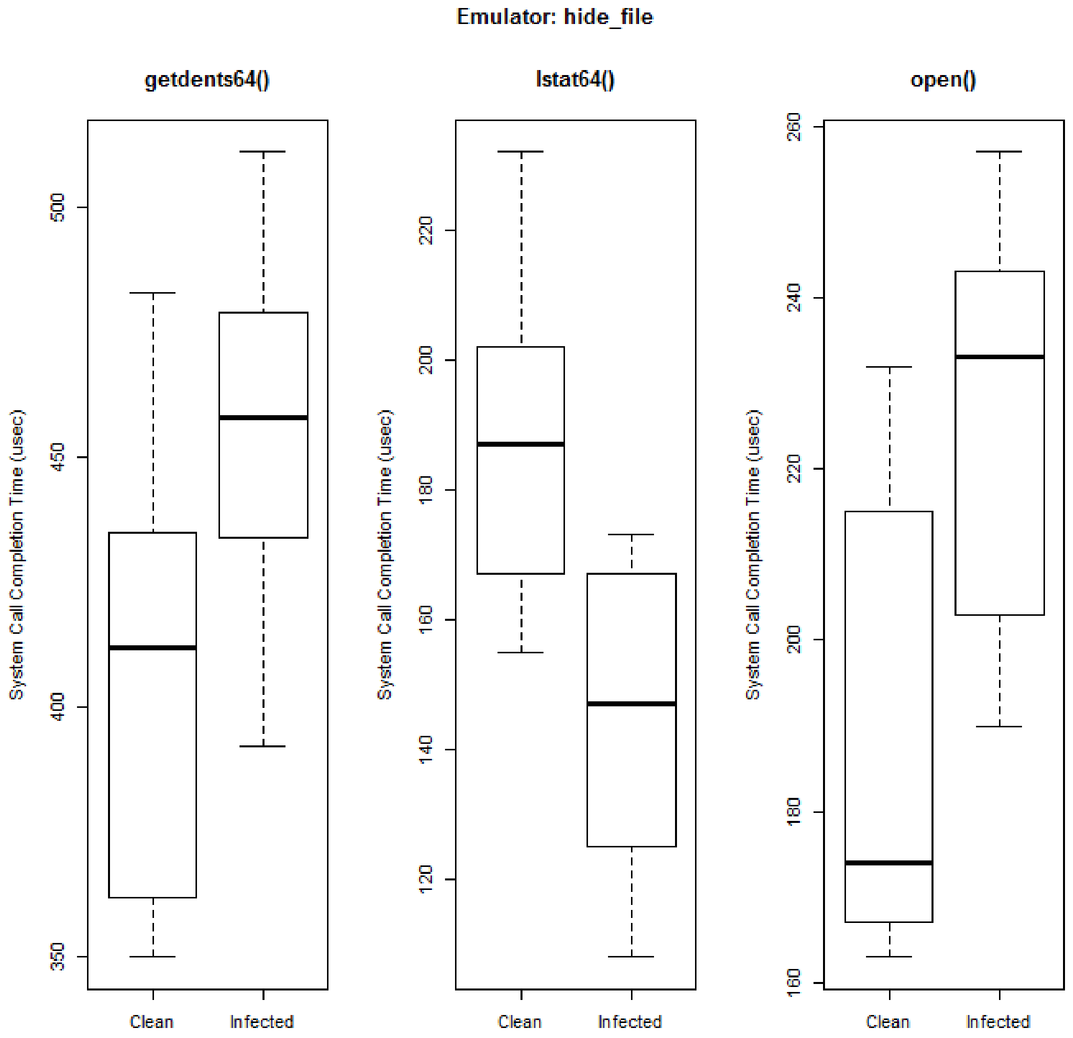


Figure B.1 Emulator: File System Call Latencies

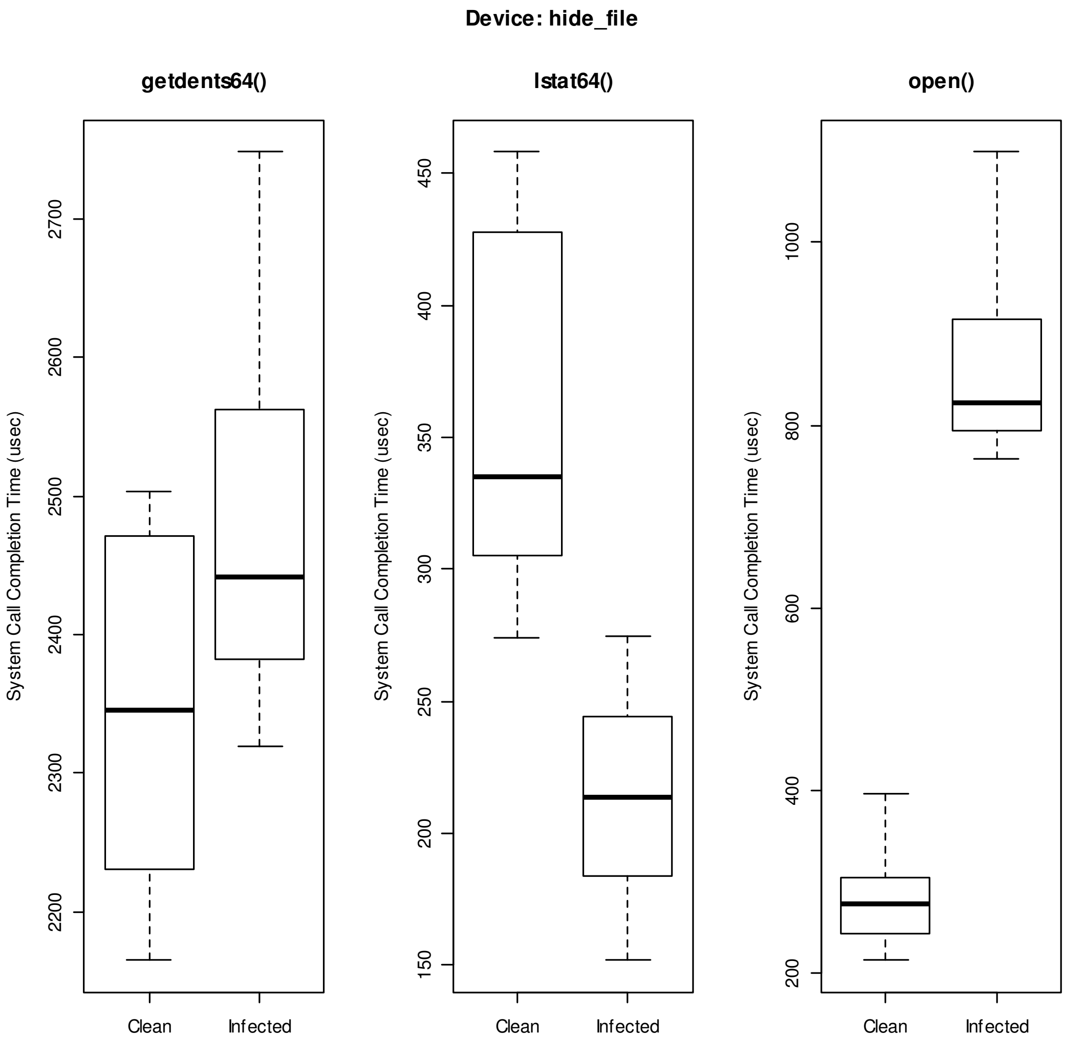


Figure B.2 Device: File System Call Latencies

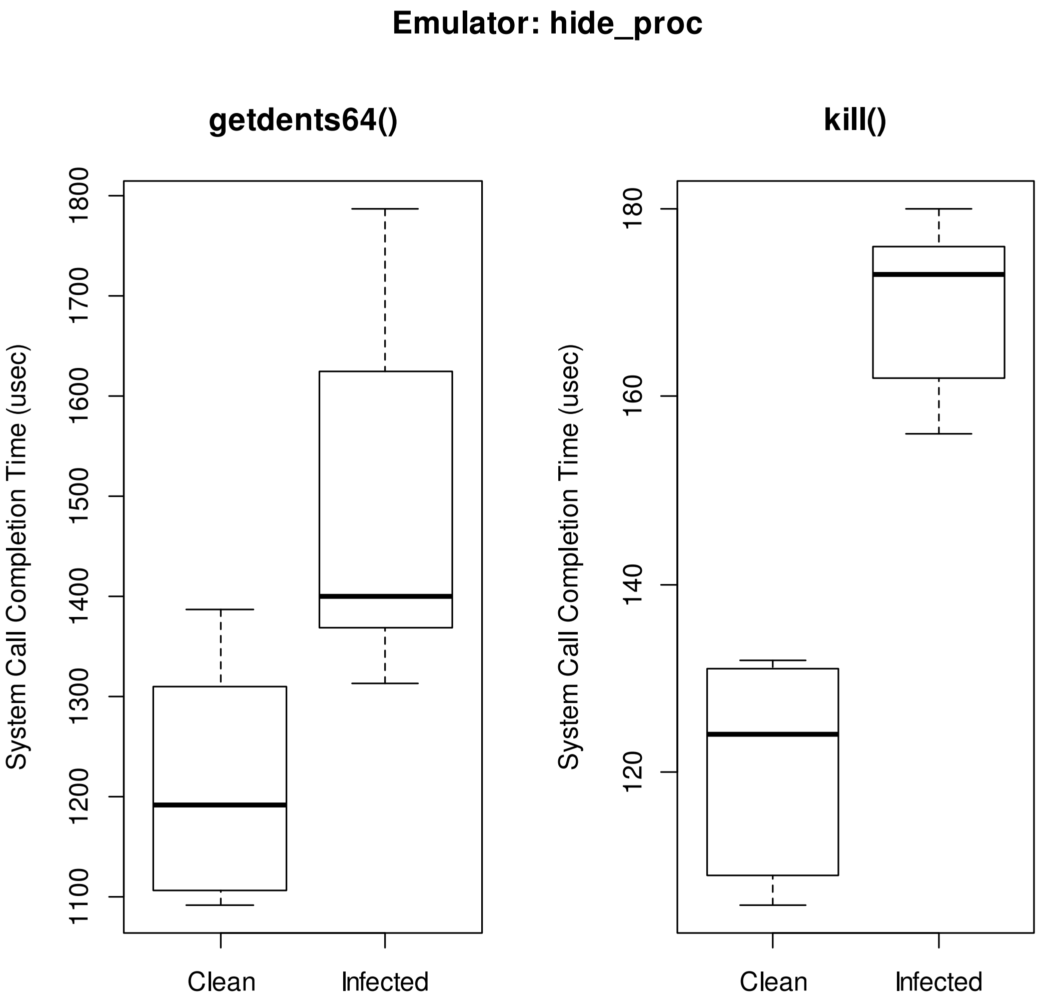


Figure B.3 Emulator: Process System Call Latencies

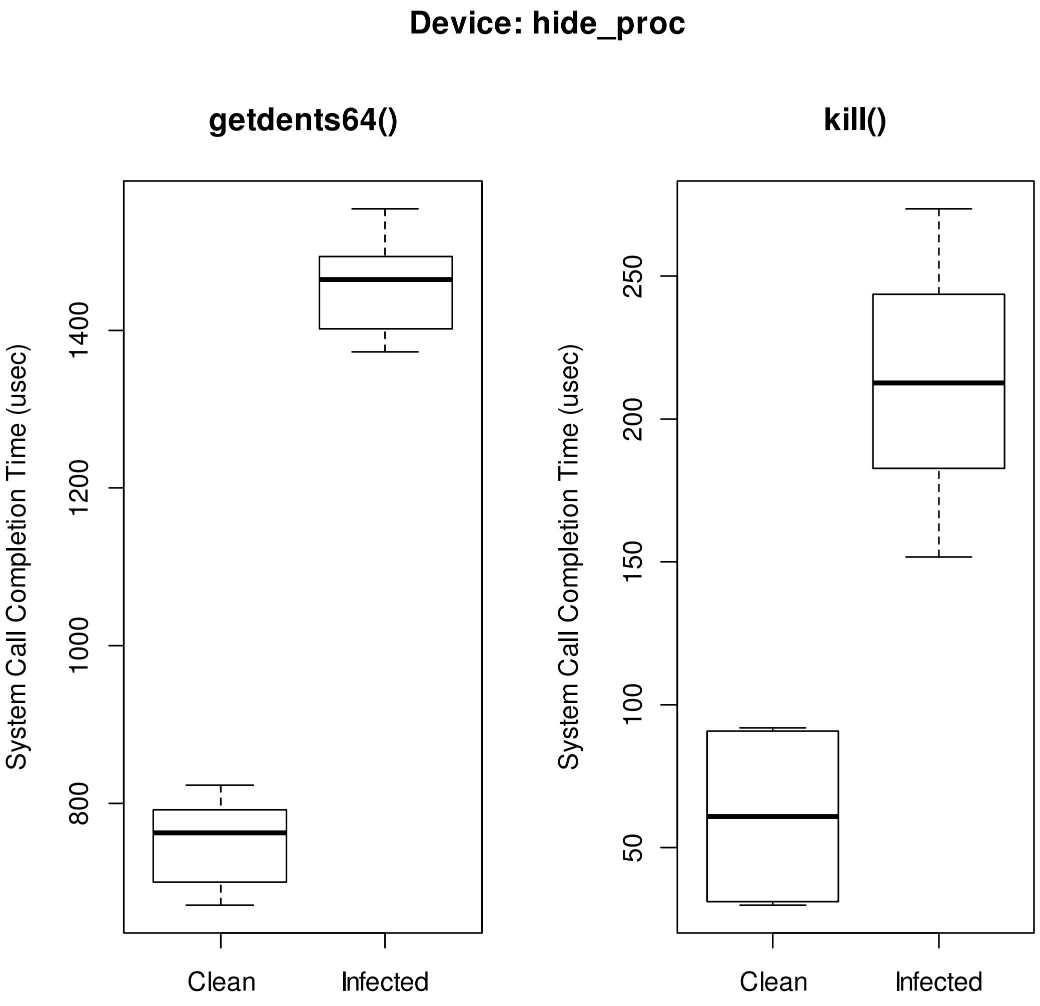


Figure B.4 Device: Process System Call Latencies

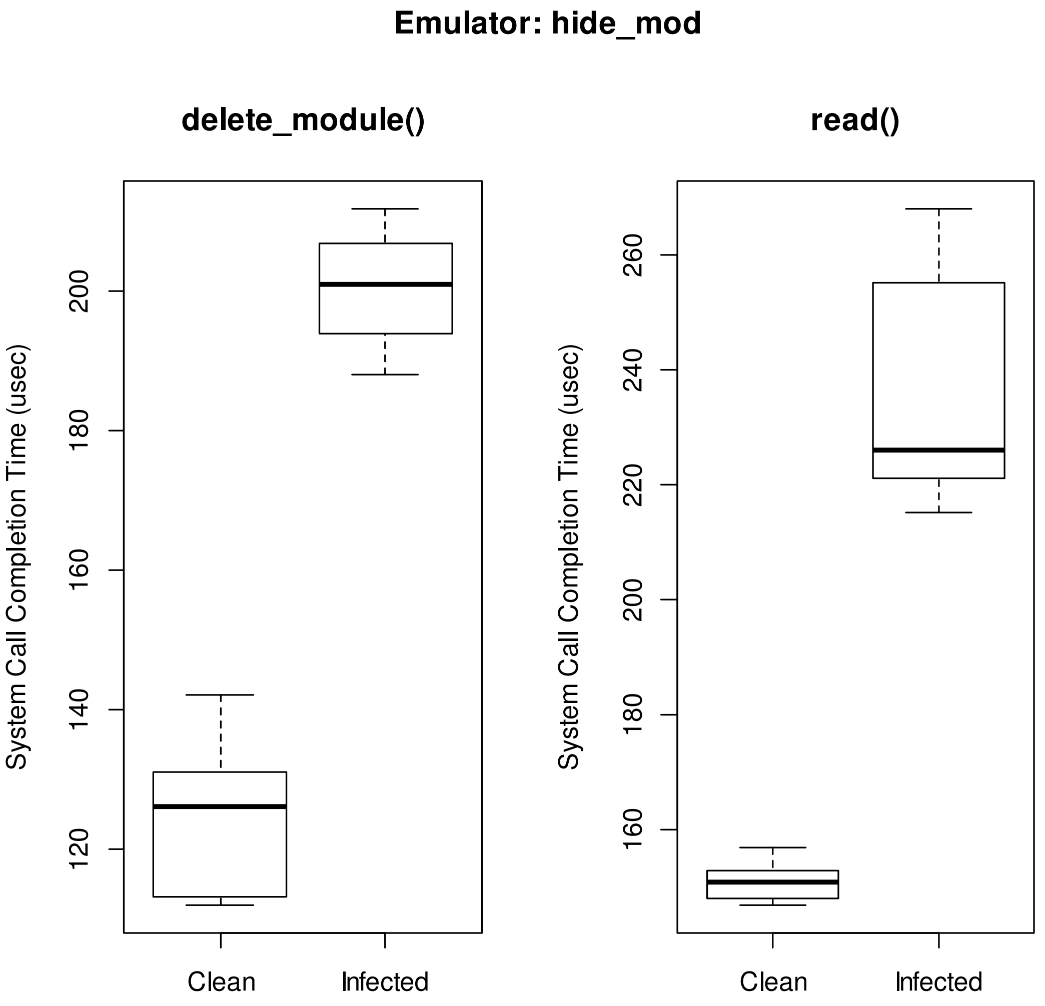


Figure B.5 Emulator: Module System Call Latencies

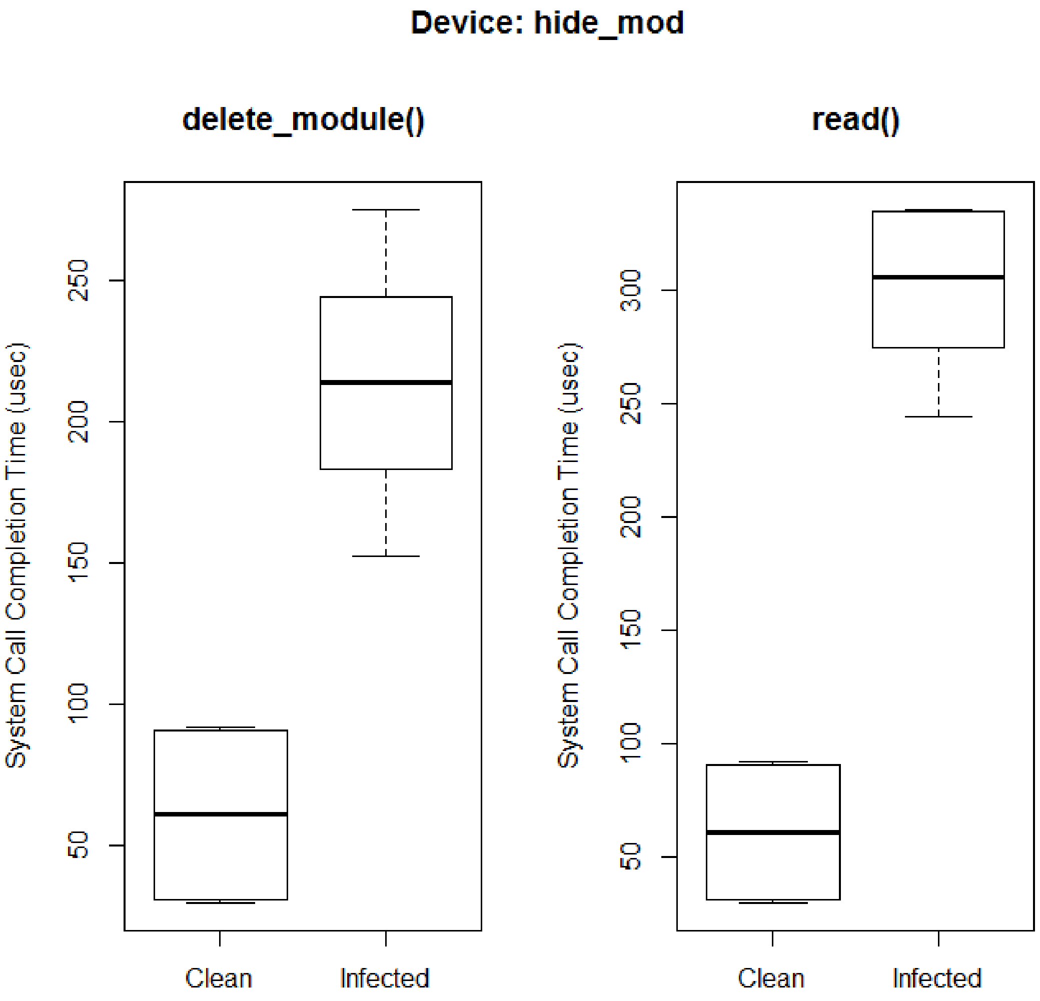


Figure B.6 Device: Module System Call Latencies

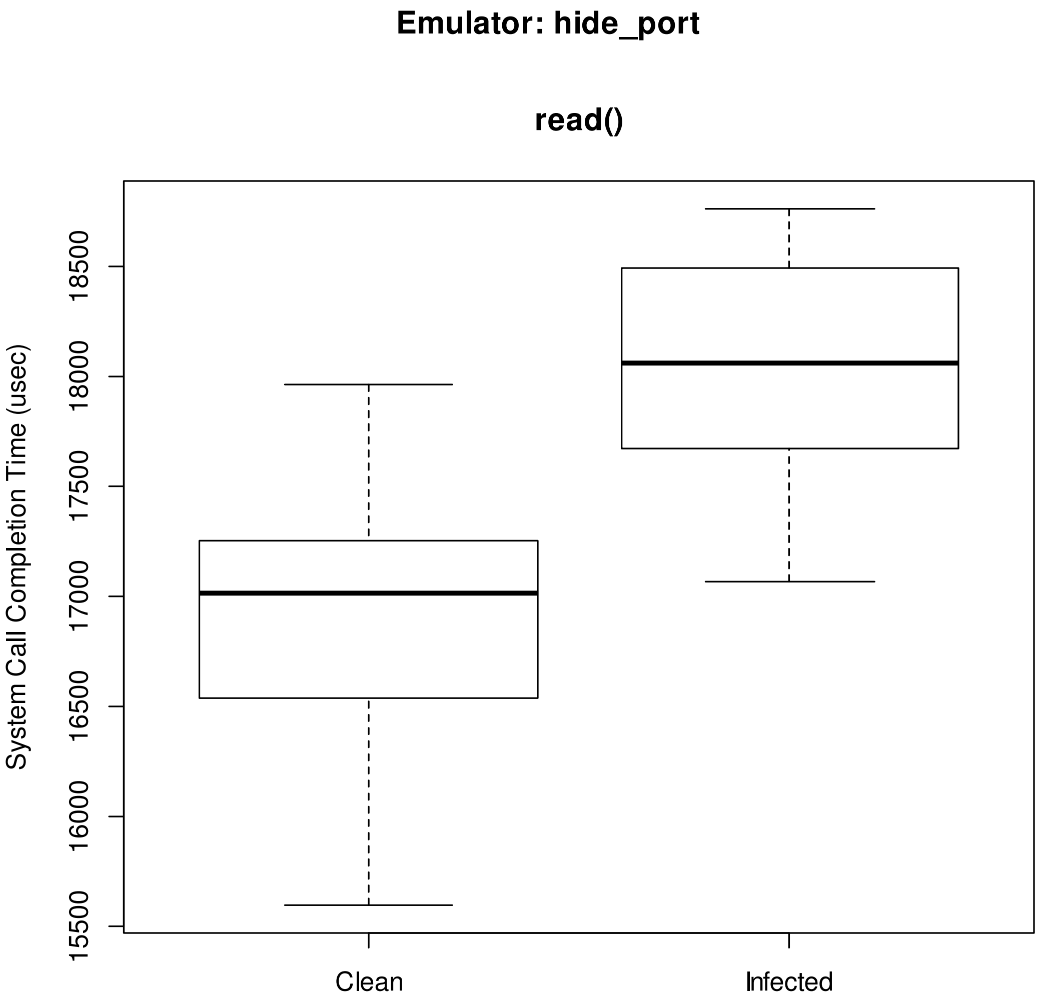


Figure B.7 Emulator: Port System Call Latencies

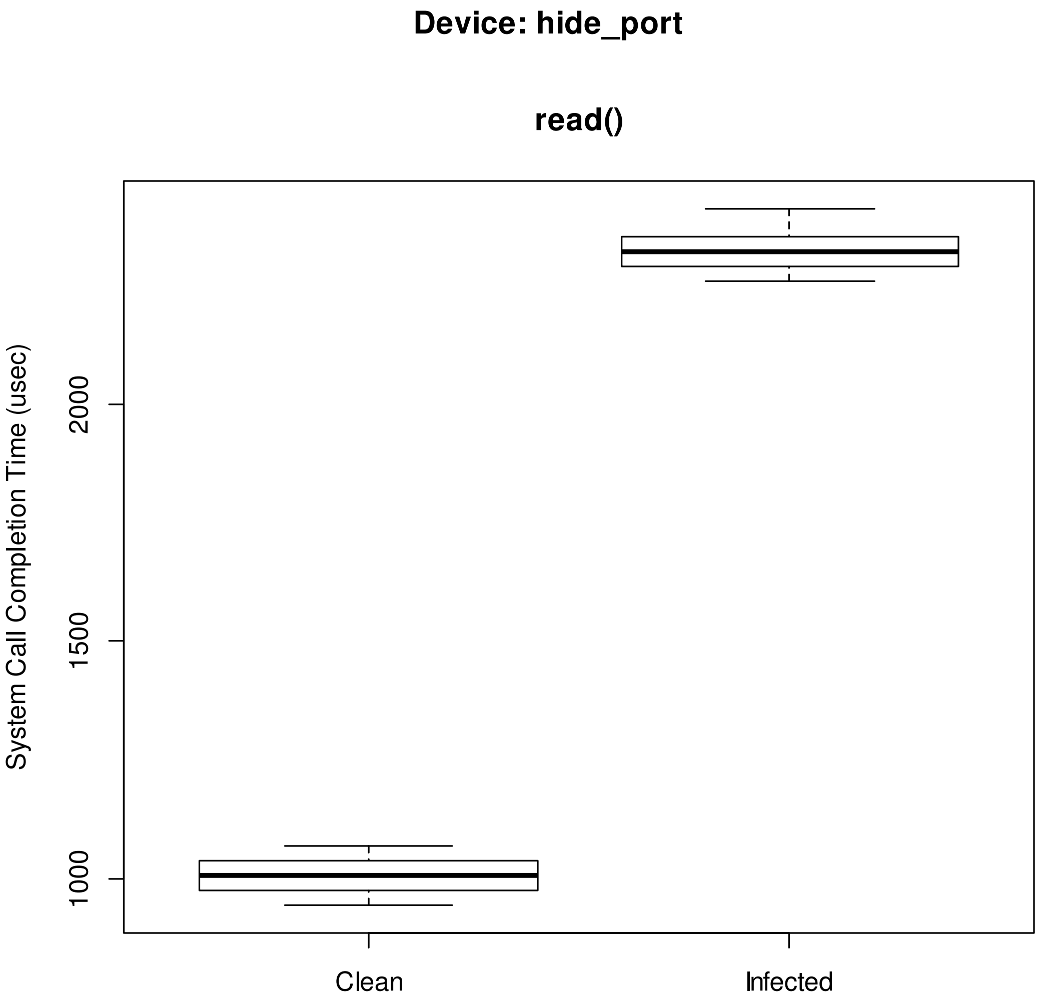


Figure B.8 Device: Port System Call Latencies

Bibliography

[Ale96] Aleph One (pseudo.). "Smashing the Stack For Fun and Profit," *Phrack,* vol. 7, no. 49, 1996.

[Ble02] blexim (pseudo.). "Basic Integer Overflows," *Phrack,* vol. 11, no. 60, 2002.

[Boe12] M. Boelen, 2012. "The Rootkit Hunter Project." [Online]. http://rkhunter.sourceforge.net/, 2012.

[Bov05] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel, Third Edition*. Sebastopol, CA: O'Reilly Media, 2005.

[Bur10] M. Burdach, "Detecting Rootkits and Kernel-level Compromises in Linux," *Symantec Connect*. [Online]. Available: http://www.symantec.com/connect/articles/detecting-rootkits-and-kernel-level-compromises-linux, November 2010.

[But04] J. Butler. "DKOM (Direct Kernel Object Manipulation)," *Black Hat USA 2004*, 2004.

[Ces98] S. Cesare. "Runtime Kernel KMEM Patching," [Online]. http://biblio.l0t3k.net/kernel/en/runtime-kernel-kmem-patching.txt, 1998

[Cla05] J. Clarke and N. Dhanjani. *Network Security Tools: Writing, Hacking, and Modifying Security Tools.* O'Reilly Media, 2005.

[Dav08] F. M. David, E. M. Chan, J. C. Carlyle and R. H. Campbell. "Cloaker: Hardware Supported Rootkit Concealment," *2008 IEEE Symposium on Security and Privacy*, 2008.

[Far11] R. Farmer. "A Brief Guide to Android Security." White Paper, Acumin Consulting, 2011.

[Gev12] D. Gevers. "skdet." Personal Website Hosting Software Download. [Online]. http://dvgevers.home.xs4all.nl/skdet/

[God12] Google Inc. "Android Developers." [Online]. http://developer.android.com/, 2012.

[Goo12] Google Inc. "Android Open Source Project." [Online]. http://source.android.com/, 2012.

[Gop12] Google Inc. "ASTRO File Manager/Browser - Android Apps on Google Play." [Online]. http://play.google.com/store/apps/details?id=com.metago.astro&hl=en, 2012.

[Gri06] J. B. Grizzard. *Towards Self-Healing Systems: Re-establishing Trust in Compromised Systems*. PhD dissertation. Georgia Institute of Technology, 2006.

[Hea06] J. Heasman. "Implementing and Detecting a PCI Rootkit." White Paper, NGSSoftware Insight Security Research (NISR), 2006.

[Hen06] B. Henderson. "Linux Loadable Kernel Module HOWTO." [Online]. http://tldp.org/HOWTO/Module-HOWTO/, 2006.

[How09] M. Howard, D. LeBlanc and J. Viega. *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them.* McGraw-Hill, 2009.

[Kad02] kad (pseudo.). "Handling Interrupt Descriptor Table for fun and profit," *Phrack,* vol. 11, no. 59, 2002.

[Kim08] W. B. Kimball. *SecureQEMU: Emulation-based Software Protection Providing Encrypted Code Execution and Page Granularity Code Signing*. MS Thesis, AFIT/GCO/ENG/09-03. School of Electrical and Computer Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 2008.

[Kon07] J. Kong. *Designing BSD Rootkits: An Introduction to Kernel Hacking*. San Francisco, CA: No Starch Press, Inc., 2007.

[Kra12] P. Kranenburg, B. Lankester and R. Sladkey. "strace." SourceForge. [Online]. http://sourceforge.net/projects/strace/, 2012.

[Lev06] J. Levine, J. Grizzard and H. Owen. "Detecting and categorizing kernel-level rootkits to aid future detection." *IEEE Security & Privacy,* vol. 4, no. 1, pp. 24-32, Jan-Feb 2006.

[Lin12] "Linux Documentation." [Online]. http://linux.die.net/man/, 2012.

[Loo12] Lookout, Inc. "Lookout Mobile Security." [Online]. https://www.mylookout.com/, 2012

[Lov10] R. Love. *Linux Kernel Development*. Addison-Wesley Professional, 2010.

[Nie93] J. Nielsen. *Usability Engineering*. San Francisco, CA: Morgan Kaufmann, 1993.

[Pap10] C. Papathanasiou and N. J. Percoco. "This is not the droid you're looking for…," *Defcon 18,* July 2010.

[Per10] E. Perla and M. Oldani. *A Guide to Kernel Exploitation: Attacking the Core.*Syngress Publishing, 2010.

[Pfl11] C. P. Pfleeger and S. L. Pfleeger. *Analyzing Computer Security: A Threat/Vulnerability/Countermeasure Approach*. Prentice Hall, 2011.

[Pra99] pragmatic (pseudo.). "(nearly) Complete Linux Loadable Kernel Modules," *The Hacker's Choice 2011,* 1999.

[Rut06] J. Rutkowska. "Subverting Vista Kernel for Fun and Profit," *Black Hat USA 2006*, 2006.

[Sdd01] sd (pseudo.) and devik (pseudo). "Linux on-the-fly kernel patching without LKM," *Phrack,* vol. 11, no. 58, 2001.

[Sha08] A. Shah. "Analysis of Rootkits: Attack Approaches and Detection Mechanisms." Research Report, Georgia Institute of Technology, 2008.

[Sha12] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer and Y. Weiss. "“Andromaly”: a behavioral malware detection framework for android devices," *Journal of Intelligent Information Systems,* vol. 38, no. 1, pp. 169-190, 2012.

[Sko06] E. Skoudis. *Counter Hack Reloaded*. Pearson Education, 2006.

[Sqr07] sqrkkyu (pseudo.) and twiz (pseudo.). "Attacking the Core: Kernel Exploiting Notes," *Phrack,* 2007. [Online]. http://www.phrack.org/issues.html?issue=64&id=6#article

[Tri10] J. M. F. d. Trindade, C. Pham and N. Dautenhahn, "µBeR: A Microkernel Based Rootkit for Android Smartphones," *IEEE Symposium on Security and Privacy 2010*, 2010.

[Vla12] D. Vlasenko. "BusyBox." [Online]. http://busybox.net/, April 2012.

[You11] D.-H. You and B.-N. Noh. "Android platform based linux kernel rootkit," *6th International Conference on Malicious and Unwanted Software,* 2011.

[You12] D.-H. You. "Android platform based linux kernel rootkit," *Phrack,* vol. 14, no. 68, 2012.

[Zov01] D. D. Zovi, "Kernel Rootkits." *SANS: Information Security Reading Room.* [Online] http://www.sans.org/reading_room/whitepapers/threats/kernel-rootkits_449, 2001.

[Zov06] D. D. Zovi. "Hardware Virtualization-based Rootkits," *Black Hat USA 2006*, 2006.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 14 June 2012	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From - To)* 20 Sept 2012 - 14 June 2012

4. TITLE AND SUBTITLE Covert Android Rootkit Detection: Evaluating Linux Kernel Level Rootkits on the Android Operating System	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Brodbeck, Robert C., Civilian, USAF	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way Wright-Patterson AFB OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT/GCO/ENG/12-14
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Intentionally Left Blank	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

This research developed kernel level rootkits for Android mobile devices designed to avoid traditional detection methods. The rootkits use system call hooking to insert new handler functions that remove the presence of infection data. The effectiveness of the rootkit is measured with respect to its stealth against detection methods and behavior performance benchmarks. Detection method testing confirms that while detectable with proven tools, system call hooking detection is not built-in or currently available in the Google Play Android App Store. Performance behavior benchmarking showed that system call hooking affects the completion time of the targeted system calls. However, this delay's magnitude may not be noticeable by users. The rootkits implemented targets Android 4.0 on the emulator available from the Android Open Source Project (AOSP) and the Samsung Galaxy Nexus. The rootkits are compiled against both Linux kernel 2.6 and 3.0, respectively. This research shows the Android's Linux kernel is vulnerable to system call hooking and additional measures should be implemented before handling sensitive data with Android.

15. SUBJECT TERMS

Operating Systems, Cellular Phones, Information Security, Mobile Computing

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Dr. Rusty O. Baldwin, ENG
U	U	U	UU	98	**19b. TELEPHONE NUMBER** *(Include area code)* (937) 255-3636, x 4445 Rusty.Baldwin@afit.edu