Android Rootkits

Adam Zakaria

adam.zakaria@tufts.edu

Ming Chow

Abstract

A rootkit is software designed to help a user maintain “root” privileges through the hiding of processes and the redirection of system calls. Rootkits are particularly insidious because they are often kernel-based, making it very difficult for malware detection programs within an operating system to combat them. Linux rootkits on the x86 architecture are a well-documented topic, but documentation on Android rootkits, which typically run on the ARM architecture, is relatively sparse. My paper will provide an overview of the Android architecture, explore methods of implementing Android rootkits, and briefly discuss how to protect systems against them.

Introduction

To The Community

Given the increasing presence of mobile computing, and the increasing market share of Android, development of malware for the platform has become very popular, and will continue to grow. It is thus imperative that knowledge about Android rootkits is disseminated throughout both the developer and end-user communities. I personally chose this topic because I greatly enjoy low level system hacking and wanted to learn about the ARM architecture. This paper tries to speak in layman’s terms where possible, and tries to provide sufficient explanation of technical concepts such that outside sources of information will not need to be frequently consulted. I hope you enjoy it and please email me if you have any questions or simply want to chat.

Android Architecture

Android is the world’s most popular mobile operating system and is written on top of the Linux kernel. Similar to the Linux kernel, Android is an open source project, but its latest releases are privately developed by Google, and once these releases are made available to the public their source code is made available as well.

The linux kernel is the interface between Android's software and the device's hardware, it is the lowest level software component. On top of the kernel are Android's libraries, application framework, and applications. Applications are written in Java and are run in Dalvik, a virtual machine particularly suited for low resource devices and which uses just-in-time compilation to achieve a performance boost.

Android applications are sandboxed, meaning each application runs in its own environment, isolated from other applications. By default, applications are not privileged processes. Android malware has experienced a huge jump in popularity thanks to the operating system's increased market share, and there is perhaps no juicier target for cybercriminals than the kernel.

Rootkit Primer

Once crackers gain root access to a system, it is often difficult to maintain their privileges because of the "noise" (logs, running processes, etc.) that may have been created during the break in. An effective way of maintaining root access is through the use of a kernel level rootkit. A rootkit is a type of software designed to hide processes, change files, and do anything else necessary to keep root access. Rootkits have several different flavors: user mode, kernel mode, firmware and hypervisor, the most popular flavors being user mode and kernel mode. Kernel mode rootkits are particularly lethal because they have the same privileges as the operating system, making it difficult for the anti-malware systems within the operating system to detect them because they can easily manipulate the data being analyzed.

Attackers that gain privileges on Android will want to maintain those privileges and they may do so with a kernel level rootkit as aforementioned. Linux kernel rootkits are well documented for the x86 architecture, but most Android devices run on ARM processors and rootkits on the ARM architecture are not well documented. Thus, the rest of this paper will be an analysis of the methods and difficulties of installing a kernel rootkit in Android on the ARM architecture.

Rootkits: Installation

Finding the sys_call_table

Rootkits are able to do their dirty work by hooking the kernel. Hooking is the act of intercepting system calls and handling their interception in some way, perhaps by redirecting them elsewhere. In the linux kernel, the addresses of system calls are stored in the sys_call_table. If we can somehow overwrite addresses stored in the sys_call_table we can redirect system calls to arbitrary locations in memory and essentially do whatever we want. However, it is not so simple. First of all, it is often difficult to find where the sys_call_table is because it is often not exported.

The linux kernel maintains the addresses of all of its global symbols, symbols being either variables or functions, in /proc/kallsyms. Symbols can be either static, external, or exported. Static symbols are only available visible within where they are written, external symbols are available from any part of the kernel, and exported symbols are even available to loadable kernel modules. If sys_call_table is not exported, which it is not after kernel version 2.6, it will not be in /proc/kallsyms, making it more difficult for us to find it. Even worse, the sys_call_table location depends on the kernel version in use. Sys_call_table can also be made read only to further deter malicious activity.

One method of finding sys_call_table is by using the Exception Vector Table, which is a table that maintains pointers to all exception handlers. The equivalent to ARM's Exception Vector Table on the x86 architecture is the Interrupt Descriptor Table All ARM systems have an Exception Vector Table and it is usually stored at address 0x0, however, Android is a high-vector implementation so it is stored at 0xffff0000.

Signals and System Calls

Exceptions are the result of software interrupt signals sent to the processor and tell the processor something has happened that requires a change of control flow. System calls are often made by libc (glib on GNU / Linux) wrapper functions, so named because they wrap up system calls in a way that makes them more portable by putting the system call number and arguments in the registers they need to be, regardless of architecture. When a system call is made by a user mode program, an interrupt signal is sent to the processor, telling it to switch to kernel mode to execute the system call, at which point the processor decides whether or not to permit the action. The underlying details of how this is done are processor dependent; on the x86 architecture, the number of the system call you want to execute is stored in the EAX register, with the system call parameters stored in the rest of the registers (if there are fewer available registers than parameters, arguments can be stored on the stack and pointed to by EBX). Once the registers are all set up, the syscall call is made by executing the 0x80 interrupt instruction. The C programming language also provides a way to directly make the system calls through the syscall function, or by using inline assembly. The process is relatively similar on ARM processors. The system call number is typically loaded into the r7 (r0 on older processors that do not support the thumb user space) register, with the parameters loaded into the other available registers (typically r0,..,r6) as necessary. The SVC instruction (supervisor call, formerly SWI (software interrupt)) is then executed to switch to kernel mode, the equivalent of the 0x80 instruction on the x86 architecture. Additionally, software interrupt signals are used to signal programming errors that arise during execution, such as a divide by zero error. There are also hardware interrupt signals that allow interrupt signals to be sent to the processor from external devices. However, neither of these are important in the context of this paper.

Finding sys_call_table, continued

Eight bytes past the start of the Exception Vector Table are instructions to call the just mentioned Software Interrupt Handler, whose address is stored 420 bytes past the start of the exception vector table. The Software Interrupt Handler references the sys_call_table, so by examining its code we can deduce the address of the sys_call_table. By looking at the assembly instructions we can obtain the opcode that references the sys_call_table, and from there we can search through memory for the location of the opcode and obtain the offset to sys_call_table. You may have noticed the peculiarity of looking for offsets rather than for memory addresses, which is a difference between ARM and x86. In order to make ARM assembly position independent, programs access data relative to the PC, the program counter, rather than referencing addresses, thus allowing a program to run regardless of where it is loaded in memory. This is desirable because ARM programs are loaded into RAM in unpredictable locations, so the programmer cannot know where data will located. Thus, containing references to absolute addresses would break the program, and thus offsets are used instead.

Finding sys_call_table with sys_close

An even easier way of finding the sys_call_table address uses the sys_close system call, and the fact that it is always 6 words from the start of the sys_call_table (it is the sixth system call). We can simply increment the software interrupt handler address until we compare the address we are at and it matches the address of the sys_close call. Since the start of sys_call_table will be 6 words before that address, simply subtracting 6 words from the location of sys_close will give us the address we need.

Writing to /dev/kmem

Now that we know where sys_call_table is located, we can start trying to write to it to redirect system calls. To do this, we use something called the /dev/kmem access technique. Kmem is a linux device file that allows the user to access kernel memory and is often used to examine and patch the system. Kmem allows arbitrary overwriting of the kernel. The kernel has been patched to disallow writing to kmem by file input / output, but it is still possible by using mmap, which creates a new memory mapping in the calling process's virtual address space. We simply call mmap on the kmem device to share the kernel memory space with the user space and then we can step through and write to the memory using the lseek, read, and write functions. Since we know the location of the sys_call_table we can write to it and redirect system calls to handler functions that we design ourselves and are able to do whatever we want.

Forging a copy of sys_call_table

This method of writing to sys_call_table directly is easily detected, so more stealthy hooking techniques have been developed. Perhaps the best technique currently available is the forged sys_call_table technique. This involves making a copy of the sys_call_table in heap memory and then writing to the Exception Vector Table to have the Software Interrupt Handler reference the forged sys_call_table. To do this, we must again find the point where the Software Interrupt handler references the sys_call_table, find the offset to it, and then go to that point in memory so we can copy it. However, we must know the size of the sys_call_table ahead of time in order to copy it, but this proves to be very tricky. Not only is sys_call_table's size platform dependent, its size can also vary depending on compile environments and config options. This means that even on two kernels of the same version, the sys_call_table may not be the same size! We can find the sys_call_table size by examining the assembly of the Software Interrupt Handler, finding the opcode that checks the upper bound of the sys_call_table, and then searching for that opcode within memory. We then copy the sys_call_table into memory, and save the offset between the original sys_call_table and the forged sys_call_table. Finally, we go back to the Software Interrupt Handler code that references the sys_call_table and overwrite the offset it uses to reference the table with the offset for the forged table. Now, anytime a system call is made the Exception Vector Table will delegate to the Software Interrupt Handler, which will now go to the forged sys_call_table instead of the original. With this forged table we can arbitrarily redirect system calls, and in a stealthy manner, because the original sys_call_table is unmodified.

Rootkits: Defense

Defending your Android device against rootkits is similar to defending it from any other malware, there really is not a specific formula just for rootkits. First and foremost, install software from trusted sources. But be wary, users often think that installing software from the Google Play store guarantees safety, but there have been many recorded cases of malicious users bypassing Bouncer, Android's malware detection bot for the Play store, and successfully uploading malware. Thus it is also very important to insure that all apps you install must follow the principle of least privilege, that is, they only require the privileges they absolutely need to function properly. Finally, be wary of misinformation. Android is a relatively new and misunderstood platform, so education and critical thinking on the part of the user is imperative to a secure system.

The best thing to do to get rid of a rootkit is to format your drive and reinstall all of your software. Trying to delete a rootkit does not make much sense because rootkits can potentially change the behavior of every program on your system; you cannot expect to find a rootkit when the tools you are using to detect it are potentially affected by the rootkit.

Conclusion

As Android rootkits are relatively few in number, tools that try to detect their presence have yet to be developed. There are programs for Linux like Tripwire and AIDE that use MD5 to check the integrity of files so that if the kernel or user space programs are modified, the modifications will be detected. However, these tools can be rendered useless by sophisticated rootkits. One possibly effective method of detection is monitoring API calls and CPU usage, because rootkits will often run background daemons, though the tools used to detect these daemons may be affected by the rootkit and cannot be trusted. If one is to take anything away from this paper, take away the fact that you cannot trust anything about a computer infected with a rootkit. Rootkits are extremely sophisticated, stealthy, and malicious pieces of software, and it does not seem like there will be any good ways to stop them in the near future.

References:

[1] You, Dong-Hoon. "Android platform based linux kernel rootkit ." *Phrack Magazine*. 04 04 2011:

[2] Rainer, Wichmann. "Linux Kernel Rootkits." *Linux Kernel Rootkits*. la-samhna labs. Web. 13 Dec 2013. <http://www.la-samhna.de/library/rootkits/index.html>.

[3] Cockerell, Peter. "Assembly Programming Principles." *ARM Assembly Language Programming*.

Computer Concepts, n.d. Web. 13 Dec 2013. <http://www.peter-cockerell.net/aalp/html/ch-5.html>.

[4] "Android LKM Rootkit." *wiki::rootkit*. upche, 23 12 2009. Web. 13 Dec 2013. <http://upche.org/doku.php?id=wiki:rootkit>.

[5] Percoco, Nicholas J., and Christian Papathanasiou. "“This is not the droid you’re looking for…”." *DEFCON*. Defcon. Web. 13 Dec 2013. <http://www.defcon.org/images/defcon-18/dc-18-presentations/Trustwave-Spiderlabs/DEFCON-18-Trustwave-Spiderlabs-Android-Rootkit.pdf>.

[6] "When Malware Goes Mobile." *SOPHOS*. Sophos. Web. 13 Dec 2013. <http://www.sophos.com/en-us/security-news-trends/security-trends/malware-goes-mobile/10-tips-to-prevent-mobile-malware.asp&xgt;.

[7] Wang, Zhaohui. "Google Android Platform Introduction to the Android API, HAL and SDK." . George Mason University. Web. 13 Dec 2013. <http://cs.gmu.edu/~astavrou/courses/ISA_673_S12/Android_Platform_Extended.pdf>.

Supporting Material

Listed below is code for the two methods of finding the elusive sys_call_table, as described in the paper. I have provided annotations within the code.

Sys_close method

//Initialize pointers to traverse code section where sys_close should be contained

```
static unsigned long* sys_call_table;
static int get_sys_call_table(void) {
  unsigned long *ptr;
  unsigned long *ptr_save = NULL;
  long delta;
  ptr=(unsigned long *)init_mm.start_code;
```

//Find sys_close, we subtract 6 words from address to take into account 6th position in sys_call_table (6th system call)

```
  while((unsigned long )ptr < (unsigned long)init_mm.end_code) {
                if((unsigned long *)*ptr == (unsigned long *)sys_close) {
                ptr_save = ptr-6;
                printk (KERN_INFO" -> matching detected at %p\n", ptr_save);                   }
```

//Compute the offset by matching against the relevant ARM opcodes

```
                if( (unsigned long *)*ptr             == (unsigned long *)0xe3c07609 &&
                (unsigned long *)*(ptr+1) == (unsigned long *)0xe3570071 &&
                (unsigned long *)*(ptr+2) == (unsigned long *)0x13570f59 &&
                (unsigned long *)*(ptr+3) == (unsigned long *)0x388d0060 &&
                (unsigned long *)*(ptr+4) == (unsigned long *)0x31a00001 &&
                (unsigned long *)*(ptr+5) == (unsigned long *)0x31a01002 &&
                (unsigned long *)*(ptr+6) == (unsigned long *)0x31a02003 &&
                (unsigned long *)*(ptr+7) == (unsigned long *)0x31a03004 ){

     delta = ptr - ptr_save;
                printk (KERN_INFO" -> opcode detected at %p, probable stc %p, delta = %ld\n", ptr, ptr_save, delta);
```

//We know that if we get an offset outside of this range we messed something up

```
if(delta > 0x154 && delta < 0x174){
            sys_call_table = (unsigned long *)ptr_save;             return 0;   }             }
            ptr++;  }  return -1;}
```

Doing Things The Hard Way

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/mm.h>
#include <linux/init_task.h>
#include <linux/syscalls.h>

unsigned long* find_sys_call_table();
```

//Nice little greeting message for when the Linux Kernel Module is loaded

```
int init_module() {
  printk("<1> Hello, kernel!\n");
  unsigned long *sys_call_table = find_sys_call_table();
```

```
return 0;

}

//Nice little exit message

void cleanup_module() {

   printk("<1>I'm not offended that you"

                "unloaded me.  Have a pleasant day!\n");

}

unsigned long* find_sys_call_table()

{

//Software Interrupt Table handler address (remember, in some ARM systems this would be stored at

0x8, but Android is a vector implementation and is thus stored at this address)

  const void *swi_addr = 0xFFFF0008;


  unsigned long* sys_call_table = NULL;

  unsigned long* ptr = NULL;

  unsigned long vector_swi_offset = 0;

  unsigned long vector_swi_instruction = 0;

  unsigned long *vector_swi_addr_ptr = NULL;


 //Get the instructions for vector_swi

  memcpy(&vector_swi_instruction, swi_addr, sizeof(vector_swi_instruction));

  printk(KERN_INFO "-> vector_swi instruction %lx\n", vector_swi_instruction);


 //Get vector_swi offset

  vector_swi_offset = vector_swi_instruction & (unsigned long)0x00000fff;

  printk (KERN_INFO "-> vector_swi offset %lx\n", vector_swi_offset);


//Calculate vector_swi address

                vector_swi_addr_ptr = (unsigned long *)((unsigned long)swi_addr + vector_swi_offset + 8);

  printk (KERN_INFO "-> vector_swi_addr_ptr %p, value %lx\n", vector_swi_addr_ptr, *vector_swi_addr_ptr);
```

```
//Here we start to try to find the reference to sys_call_table from the SWI handler
 ptr=*vector_swi_addr_ptr;
 bool foundFirstLighthouse = false;
 unsigned long sys_call_table_offset = 0;

 printk (KERN_INFO "-> ptr %p, init_mm.end_code %lx\n", ptr, init_mm.end_code);

 while ((unsigned long)ptr < init_mm.end_code && sys_call_table == NULL)
 {
         //Lighthouse values to insure we're in the right spot in memory (lighthouses guide, duh)
         if ((*ptr & (unsigned long)0xffff0fff) == 0xe3a00000)
         {
         foundFirstLighthouse = true;
         printk (KERN_INFO "-> found first lighthouse at %p, value %lx\n", ptr, *ptr);
         }

         //Looking for the load sys_call instruction
         if (foundFirstLighthouse && ((*ptr & (unsigned long)0xffff0000) == 0xe28f0000))
         {
         // Get syscall table offset
         sys_call_table_offset = *ptr & (unsigned long)0x00000fff;
         printk (KERN_INFO "-> sys_call_table reference found at %p, value %lx, offset %lx\n", ptr, *ptr,

sys_call_table_offset);

         // VOILA!  Address obtained!
         sys_call_table = (unsigned long)ptr + 8 + sys_call_table_offset;
         printk (KERN_INFO "-> sys_call_table found at %p\n", sys_call_table);
         break;
         }
```

```
                ptr++;
  }

  return sys_call_table;
}
```