Rootkits on Smart Phones: Attacks and Implications

Jeffrey Bickford, Ryan O'Hare, Arati Baliga, Vinod Ganapathy and Liviu Iftode
Department of Computer Science
Rutgers University

ABSTRACT
Smart phones are increasingly being equipped with operating systems that compare in complexity with those on desktop computers. This trend makes smart phone operating systems vulnerable to many of the same threats as desktop operating systems.

This paper examines the threat posed by *rootkits* to smart phones. Rootkits are malware that stealthily achieve their goals by modifying operating system code and data, and have long been a problem for desktops. However, smart phones expose several unique interfaces, such as voice, GPS and battery, that rootkits can exploit in novel ways. These attacks can have serious social consequences, ranging from loss of privacy to denial of service during emergencies.

This paper demonstrates the threat of smart phone rootkits with three novel attacks. We implemented rootkits that allow a remote attacker to: (1) snoop on a victim's confidential conversations; (2) snoop on a victim's geographical location; and (3) stealthily exhaust the battery on a victim's phone. We also discuss the social implications of each of these attacks.

INTRODUCTION
Over the last several years, mobile phones have evolved from a mere means of communication to general-purpose computing platforms. Such mobile phones—also called *smart phones*—are equipped with a variety of software and hardware mechanisms that let a user better interact with the cyber and physical world. For example, smart phones are often pre-installed with a number of applications, including clients for location-based services and general-purpose web browsers. These applications utilize hardware features such as GPS and enhanced network access via 3G and Wimax. To support the increasing complexity of software and hardware on smart phones, smart phone operating systems have similarly evolved. For example, modern smart phones typically run full-fledged distributions of operating systems, such as Linux, Windows, Android and Symbian OS, that comprise tens of millions of lines of code.

However, the increasing complexity of smart phones has also increased their vulnerability to attacks. Recent years have witnessed the emergence of *mobile malware*, which are viruses and worms that infect smart phones. For instance, F-Secure reported an almost 400% increase in mobile malware within a two year period from 2005-2007 [27]. Mobile malware typically use many of the same attack vectors as do malware for traditional computing infrastructures, but often spread via interfaces and services unique to smart phones, including Bluetooth, SMS and MMS. The Cabir worm, for instance, exploited a vulnerability in the Bluetooth interface and replicated itself to other Bluetooth enabled phones. Recent research has also explored the security implications of connecting smart phones to the Internet—Enck *et al.* [24] demonstrated attacks that could compromise open interfaces for SMS (*e.g.,* web sites that allow users to send SMS messages) to cripple large portions of a cellular network.

This paper explores the threat posed by *kernel-level rootkits* to smart phones. Rootkits are malware that achieve their malicious goals by infecting the operating system. For example, rootkits may be used to hide malicious user space files and processes, install backdoors and Trojan horses, log keystrokes, disable firewalls, virus scanners and intrusion detection systems, and include the infected system into a botnet. Worse, because they affect the operating system, rootkits can achieve their malicious goals stealthily, thereby remaining undetected and retaining longer-term control over infected machines. Stealth techniques adopted by rootkits have become popular among malware writers—a recent study by MacAfee reported a nearly 600% increase in rootkits in the three-year period from 2004-2006 [16].

While rootkits have long been a threat to traditional desktop computers because their operating systems present a large and complex attack surface, the increasing complexity of smart phone operating systems makes them an attractive target for rootkit authors. As a general-purpose computing platform, smart phones are also vulnerable to many of the same threats posed by rootkits to desktop computers. However, the main contribution of this paper is in showing that rootkits can exploit several interfaces and services unique to smart phones to launch novel attacks with serious social consequences. Specifically, this paper demonstrates rootkits that implement three new attacks:

1. *Snooping via the voice subsystem.* We demonstrate a rootkit that infects the GSM subsystem, thereby enabling an attacker to snoop on a victim's confidential conversations.

The rootkit is programmed to stealthily dial the attacker's phone number when certain events of interest are triggered, for instance, when a calendar program sends a reminder to the victim about an impending meeting.

2. *Location-tracking with GPS.* We present a rootkit that compromises privacy of a victim's location. When an attacker sends a command to a rootkit-infected phone (*e.g.,* via an SMS message) the rootkit queries the GPS device and sends the victim's coordinates to the attacker (*e.g.,* as a text message).

3. *Denial of service via battery exhaustion.* Smart phones are battery operated and are hence resource constrained. We demonstrate a rootkit that stealthily exhausts a smart phone's battery. This attack renders the phone unusable when its user needs it the most, *e.g.,* during emergencies.

The social consequences of these attacks are devastating. Smart phones have become ubiquitous to the point that people rely on their phones for day-to-day activities, such as coordinating meetings and dealing with emergencies. As a personal device, users typically trust their phones and do not expect them to misbehave. Smart phone rootkits exploit this trust to achieve their malicious goals while also being extremely difficult to detect. Because the smart phone user demographic is extremely diverse, we suspect that a large fraction of these users are completely unequipped to deal with security mechanisms on their phones. Consequently, such users are easy targets for stealth attacks such as rootkits.

Detecting and recovering from rootkits is challenging, even on desktop systems. Because rootkits affect the operating system, any rootkit detection mechanism must operate outside the operating system, typically on specialized hardware (such as a co-processor [50, 28]) or in a virtual-machine monitor [26, 39, 18]. Although there have been recent efforts to deploy virtual machines on smart phones [14, 8, 10, 15], such support is not widely available yet. Even so, existing rootkit detection techniques [28, 36, 37, 17], which have primarily been developed for desktop systems, employ heavyweight mechanisms that require periodic scans of kernel memory snapshots. Such techniques will likely place substantial energy demands if used on smart phones. We conclude the paper with a discussion of these and other techniques to detect smart phone rootkits.

BACKGROUND

This section presents an overview of rootkit attacks, which have affected desktop systems for nearly a decade. We also discuss why smart phones are a particularly attractive platform for malware, and rootkits in particular.

Rootkits: Threats and Evolution

The term "rootkit" was originally coined to refer to a toolkit of techniques developed by attackers to conceal the presence of malicious software on a compromised system. A rootkit is typically installed after an attacker obtains elevated privileges on the system via other means. Two popular methods for a remote attacker to gain entry into a system are (1) exploiting software vulnerabilities; and (2) drive-by-download attacks. In either case, remote attackers compromise software vulnerabilities, such as buffer overflows, either in network-facing server applications or in browser code to download rootkits onto the system. Rootkits can also be delivered via spam, peer-to-peer sharing applications or through other attacks, such as worms or bots.

Once infected, a rootkit can be used to open the door to several future attacks. For example, rootkits are commonly used to conceal keyloggers, which steal sensitive user data, such as passwords and credit card numbers, by silently logging keystrokes. They might also install backdoor programs on the system that allow a remote attacker to gain entry into the system in the future. Rootkits can also perform other stealthy activities, such as disabling the firewall/antivirus tools or affecting the output quality of the system's pseudo random number generator, thereby causing the generation of weak cryptographic keys [19]. None of these activities are directly visible to the user because the rootkit conceals their presence. Indeed, rootkits are characterized by such stealthy behavior. Their stealthy nature enables rootkits to stay undetected, and therefore retain long-term control over infected systems.

Stealth techniques used by rootkits have evolved significantly over the past decade, as have techniques to detect stealthy behavior. Early rootkits operated by replacing critical system binaries and shared libraries with Trojan horses that contained malicious functionality. These rootkits were typically installed after the attacker had gained control over the system via other means. For instance, an attacker could first acquire root privileges by compromising a network-facing setuid application via a buffer overflow exploit, following which he could replace system binaries with Trojan horses. A popular example of such *user-level rootkits* is the t0rn rootkit, which replaced the system binary of the Linux `ps` utility, thereby concealing malicious processes from the system user. User-level rootkits remained stealthy because they infected *existing* system binaries rather than downloading new files on the system. However, detecting such rootkits is also relatively easy. For example, tools such as Tripwire and AIDE [29, 2] detected such rootkits by checking the integrity of system binaries and shared libraries, *e.g.,* by comparing a SHA-1 hash of these binaries against known values.

Because rootkits that affect user-level files are easily detected, modern rootkits have evolved to modifying the code and data structures of the operating system. Among the most common targets of such *kernel-level rootkits* are the operating system's data structures that store *control data*, such as system call table, interrupt descriptor table and other function pointers, such as those in the virtual file system layer, which determine control flow in the kernel. Rootkits modify these data structures (using a technique called *hooking*) to interpose upon the kernel's control path and hide malicious functionality and objects, such as files and processes. For example, the Adore rootkit modifies several entries in the system call table of an infected Linux system. Each of these modified entries points to malicious code that is inserted into the operating system as a loadable kernel module. When a user-level program invokes the system call, the modified entries in the system call table cause the malicious code to be invoked, which in turn invokes the code of the system call. The malicious code in Adore, for instance, hides a root shell backdoor program that it installs on the system for the attacker to re-enter [1].

There are two key reasons why kernel-level rootkits are stealthy and difficult to detect. First, kernel-level rootkits operate by modifying the operating system. Consequently, they can easily hide themselves from user-level antivirus tools, which typically rely on information supplied by the operating system to detect malicious software. For example, an infected operating system can modify the view of the file system visible to an antivirus tool, thereby effectively preventing the antivirus from scanning malicious files. Existing solutions to detect kernel-level rootkits use external mechanisms, such as a co-processor or a virtual-machine monitor, to externally scan the contents of kernel memory to ensure integrity of its data structures, *e.g.,* by checking that entries in the system call table point to code that implements system calls [28, 36, 37, 17].

The second reason why rootkits are difficult to detect is that the kernel manages several thousand heterogeneous data structures, most of which are critical to its correct operation. As a result, the attack surface of the kernel is huge. Recent work demonstrated rootkits that modify several kernel data structures that store *non-control data* to subvert the kernel [19, 36]. For example, the Fu rootkit hides a malicious user-space process by modifying linked lists maintained by the Windows kernel for process accounting [6].

Recent work has also explored rootkits that use other stealth mechanisms. These include virtual-machine based rootkits that install a hypervisor below an existing OS [31, 4] and those that exploit hardware-specific features [23, 42]. These rootkits do not affect the operating system and are therefore even more difficult to detect than kernel-level rootkits. Nevertheless, kernel-level rootkits still remain by far the easiest and most popular option for attackers to achieve stealth because of the large and complex attack surface that the kernel presents. Therefore, the focus of this paper is on the threat of kernel-level rootkits to smart phones.

Smart Phones and Mobile Malware

The decreasing cost of advanced computing and communication hardware has made such hardware affordable for adoption by smart phone vendors. In turn, smart phones equipped with such advanced features have gained tremendous popularity. Over 115 million smart phones were sold worldwide in 2007 [12]. The iPhone alone, which incorporates several hardware features and mobile applications, sold one million units within 74 days of its launch [3]. To support a rich set of hardware interfaces and application programs, smart phones are typically equipped with full-fledged operating systems, thereby making smart phones a general-purpose computing environment. However, smart phones offer several unique services, such as telephony, location awareness, and instant messaging via SMS and MMS, that are not typically available on desktop computers.

Smart phones are an attractive target for attackers, both in the kinds of attacks that are possible and in the social implications of these attacks. Smart phones have access to both telephony and the Internet. Malware that can attack a smart phone has the unique advantage of being able to affect the cell phone infrastructure as well as other phones on the cellular network. These abilities have driven malware authors to focus on smart phones, with a recent report from MacAfee [9] stating that nearly 14% of mobile users worldwide have been directly infected or have known someone infected by mobile malware. Nearly 72% of the users surveyed in the MacAfee study expressed concerns regarding the safety of using emerging mobile services and more than 86% were concerned about receiving inappropriate or unsolicited content, fraudulent bill increases, or information loss and theft.

The pervasive nature of smart phones and a large, unsophisticated user base also makes smart phones an attractive target for malware writers. Because phone usage revolves largely around day-to-day user activities, important personal and financial information can likely be compromised by mobile malware. For example, smart phones are increasingly being used for text messaging, email, storing personal data, including financial data, pictures and videos. Because users rely critically on their phone for daily conversations, espionage of such voice conversations is likely to have serious social implications. As a second example, users typically tend to carry their smart phones (and keep them powered on) wherever they go; therefore, an attack that compromises the GPS subsystem will compromise privacy of the victim's location.

Traditional threats to desktop systems, such as worms and viruses, have already begun infecting mobile platforms. According to F-Secure [5], there are already more than 400 mobile viruses in circulation. Several existing mobile malware result in simple annoyances—for example, the Skull.D virus locks the phone and flashes an image of a skull and crossbones on the screen—however, others are more dangerous and can cause financial damage to the user by sending text messages to "premium" numbers. Malware such as spyware and Trojan horses have also started affecting smart phones.

The threats posed by mobile malware can readily be countered using many of the same tools available for desktop machines. For example, an antivirus tool equipped with an appropriate virus signature database can detect the presence of viruses on a smart phone. As antivirus tools begin to get deployed on mobile platforms, we envisage that attackers will also move toward using stealth techniques to maintain long-term control over infected smart phones by maliciously modifying smart phone operating systems. Rootkit detection techniques are based upon the use of external mechanisms (such as a co-processor or a virtual-machine monitor) that are not readily available for smart phones, thereby providing attackers further incentive to implement malicious functionality using rootkits.

Attack	LOC	Size of kernel module
GSM	116	92.8 KB
GPS	428	101.7 KB
Battery	134	87.2 KB

Figure 1. Lines of code and size of the kernel modules that implement each of the three attacks.

In the following sections, we present three novel rootkits that exploit interfaces and services unique to smart phones, discuss their social implications as well as the challenges that they pose to detection.

ROOTKITS ON SMART PHONES

In this section, we present three proof-of-concept rootkits that we developed to illustrate the threat that they pose to smart phones. Our test platform was a Neo Freerunner smart phone running the OpenMoko Linux distribution [11]. We chose this platform because (a) Linux source code is freely available, thereby allowing us to study and modify its data structures at will; and (b) the Neo Freerunner allows for easy experimentation, *e.g.,* it allows end-users to re-flash the phone with newer versions of the operating system.

All our rootkits were developed as Linux kernel modules (LKM) that we installed into the operating system. However, during a real attack, we expect that these LKMs will be delivered via other mechanisms, *e.g.,* after an attacker has compromised a network-facing application or via a drive-by-download attack. Figure 1 presents the lines of code needed to implement each attack, and the size of the corresponding kernel module. This figure shows the relative ease with which rootkits can be developed. It also shows that the small size of kernel modules allows for easy delivery, even on bandwidth-constrained smart phones.

Although our implementation and discussion in this section are restricted to the Neo Freerunner platform, the attacks are broadly applicable, even to phones running microkernel operating systems, such as the Symbian OS. However, since the attacks modify OS-specific data structures, they must be re-implemented for each platform—we expect that doing so will be relatively easy.

Attack 1: Spying on Conversations via GSM

Goal.

The goal of this attack is to allow a remote attacker to stealthily listen into or record confidential conversations using a victim's rootkit infected smart phone. The rootkit activates itself whenever certain events of interest happen. For example, the rootkit could trigger its malicious operation when a calendar program on the victim's phone displays a notification about an upcoming event, such as a meeting. When the rootkit triggers, it stealthily dials the attacker's phone number (which can either be pre-programmed in the rootkit, or delivered to the rootkit via an SMS message from the attacker), who can then listen into or remotely record ongoing conversations. Alternatively, the rootkit could trigger when the victim dials a number. The rootkit could then stealthily

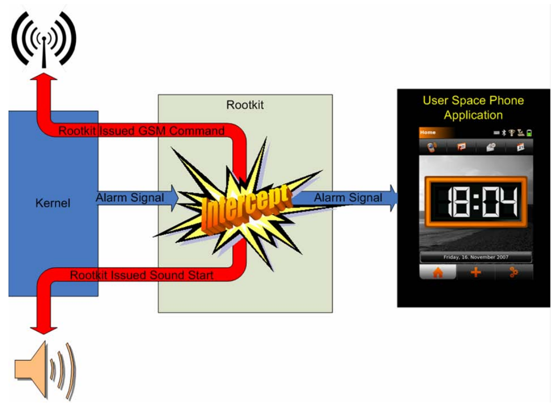
Rootkit
User Space Phone Application
Rootkit Issued GSM Command
Kernel
Alarm Signal
Intercept
Alarm Signal
18:04
Friday, 16. November 2007
Rootkit Issued Sound Start

Figure 2. The GSM rootkit intercepts an alarm signal, *e.g.,* a meeting notification, and stealthily dials the attacker, thereby allowing him to snoop on confidential conversations.

```
 1.   char *atcommand1 = "AT command1";
 2.   char *atcommand2 = "AT command2";
 3.   ...
 4.   mm_segment_t saved_fs = get_fs();
 5.   set_fs(KERNEL_DS);
 6.   fd = sys_open("/dev/ttySAC0", O_RDWR | O_NONBLOCK, 0);
 7.   sys_write(fd, atcommand1, sizeof(atcommand1));
 8.   sys_write(fd, atcommand2, sizeof(atcommand2));
 9.   ...
10.   sys_close(fd);
11.   set_fs(saved_fs);
```

Figure 3. Pseudocode of the GSM rootkit. The rootkit first prepares a set of AT commands (lines 1-3), modifies the boundaries of the data segment (lines 4-5), and writes AT commands to the GSM device (lines 6-10).

place a three-way call to the attacker's number, thereby allowing the attacker to record the phone conversation.

Background.

The Freerunner phone is equipped with GSM radio, which is connected via the serial bus and is therefore available to applications as a serial device. During normal operation of the phone, user-space applications issue system calls to the kernel requesting access to the GSM device. The kernel services the request and the application in turn is able to access telephony functionality provided by the device. GSM devices use a series of commands, called AT (attention) commands, that let the kernel and user-space applications invoke specific GSM functions. For example, GSM devices typically support AT commands to dial a number, fetch SMS messages, and so on. To maliciously operate the GSM device, *e.g.,* to place a phone call to a remote attacker, the rootkit must therefore issue AT commands from within the kernel.

Attack Description.

Our prototype rootkit operates by intercepting alarms set by the user. For example, as shown in Figure 2, the alarm could be associated with the keyword "meeting" that is displayed in a user-space calendar program to notify the user of an impending meeting. The rootkit intercepts this alarm and activates its malicious functionality when the alarm is triggered.

The attack code stealthily dials a phone number belonging to a remote attacker, who can then snoop or record confidential conversations of the victim. The phone number dialed by the rootkit can either be hard-coded into the rootkit, or delivered via an SMS message, which the rootkit intercepts to obtain the attacker's phone number (Attack 2 describes SMS-based rootkit control in further detail).

• *Triggering the rootkit.* To operate, the rootkit must have the ability to intercept an alarm signal, such as an alarm from the calendar program.[1] Our prototype rootkit achieves this goal by hooking the system call table and replacing the address of the `write` system call with the address of a malicious `write` function implemented in the rootkit. When an alarm is signaled, a specific message is written to the screen. The goal of the malicious `write` function in our prototype rootkit is to intercept the message to be written to the screen and check for the existence of three substrings. First, it checks that the substring "`Window Prop`" appears in the message, which indicates that the current message is a notification message. Second, it checks for the presence of the substring "`Clock`," which verifies that the message originated from the clock program (more substrings can be used to check for alarms from other user-space programs). Last, the substring "`NETWMType: 6`" must also occur, which indicates that an alarm signal is generated. We obtained these substrings by studying how alarm notifications are delivered to the user in an uninfected kernel.

• *Placing a phone call.* When triggered, the rootkit places a phone call by emulating the functionality of user-space telephony applications. In particular, user-space applications, such as the Qtopia software stack, which ships with the OpenMoko Linux distribution on the Freerunner phone, issue a sequence of system calls to the kernel. These system calls probe and initiate the GSM device and instruct the device to place a call to a particular number. Specifically, applications such as Qtopia use `write` system calls to issue AT commands to the GSM device (these commands are supplied as arguments to the `write` system call). The number to be dialed is also supplied as a system call argument.

Our prototype rootkit achieves the same goal by issuing the same sequence of AT commands from within the kernel. We obtained the sequence of AT calls that must be issued to place a phone call by studying the Qtopia software stack. In particular, we used the `strace` utility to trace the arguments to `write` system calls issued by Qtopia when dialing a number. We then designed a rootkit that would issue the same set of AT commands from kernel space when triggered by an alarm.

To issue the above sequence of AT calls from kernel mode, the rootkit first modifies the boundaries of the data segment to point to kernel-addressable space instead of user-addressable space using the `get_fs`/`set_fs` call sequence, as shown in Figure 3. This sequence allows the kernel to issue system calls (such as `sys_open`, `sys_write` and `sys_close`, shown in Figure 3) from kernel mode. When a system call is issued, the Linux kernel first checks that the arguments to the call are within the virtual address-space of a user-space application. While this check is important when system calls are issued by user-space applications (*e.g.,* to ensure that an application cannot maliciously refer to kernel data in a system call argument), it will cause system calls issued by the kernel to fail. The `get_fs`/`set_fs` call sequence modifies the data segment so that the checks on system call arguments succeed, thereby allowing system calls to be issued from kernel space. The rootkit simply opens the GSM device, places a sequence of `write` system calls with appropriate arguments (*e.g.,* AT commands) and closes the device.

The AT commands issued by the code in Figure 3 activate the telephony subsystem and successfully establish a connection to the attacker's phone. However, AT commands do not activate the microphone on the smart phone, which is controlled by the sound subsystem. To activate the microphone, we programmed the rootkit to issue the following command from kernel mode using the `usermodehelper` function, which allows the kernel to run user-mode commands: `alsactl -f /usr/share/OpenMoko/scenarios/gsmhandset.staterestore`.

Social Impact.

Snooping on confidential conversations has severe social impact because most users tend keep their mobile phones in their proximity and powered-on most of the time. Rootkits operate stealthily, and as a result, end users may not even be aware that their phones are infected. Consequently, an attacker can listen-in on several conversations that violate user privacy, ranging from those that result in embarrassing social situations to leaks of sensitive information. For example, an attack that records the conversations at a corporate board meeting can potentially compromise corporate trade secrets and business reports to competitors. Similarly, several automated phone-based services often require a user to enter (via voice or key presses) PIN numbers or passwords before routing the call to a human operator; an attacker snooping on such calls may financially benefit from such information.

Attack 2: Compromising Location Privacy using GPS

Goal.

The goal of this attack is to compromise a victim's location privacy. The victim's rootkit-infected smart phone receives a text message (SMS) from the attacker, querying the victim's current location. The rootkit intercepts (and suppresses) this message and sends the remote attacker a text message with the user's current location, obtained via GPS.

Background.

As with the GSM device, the GPS device is also a serial device. The kernel maintains a list of all serial devices installed on the system using a linked list of `struct tty_driver` elements. A rootkit can easily locate the GPS device using the `name` field of this structure. Every `tty_driver` structure also contains a buffer (`read_buf`) where the corresponding device stores all its data until it is read by a user-space application. Our prototype rootkit reads information from this buffer before it is accessed by user-space applications.

[1]Because the OpenMoko phone does not have any released calendar programs, we used the `alarm` command on the Linux platform to simulate a calendar application generating an alarm.

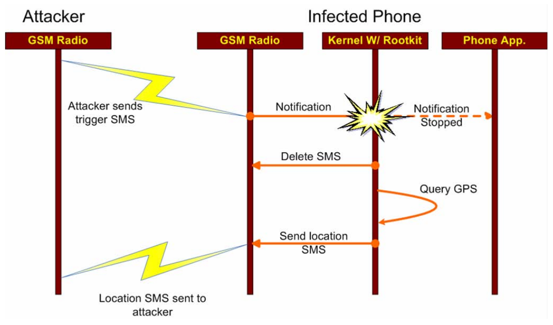

Figure 4. Sequence of steps followed in attack on location privacy.

Attack Description.
A rootkit that compromises location privacy as described above must implement three mechanisms. First, it must be able to intercept incoming text messages, and determine whether a text message is a query from a remote attacker on the victim's current location. Second, the rootkit must be able to extract location information from the GPS receiver. Last, it must generate a text message with the victim's current location, and send this information to the attacker. An overview of this attack is shown in Figure 4. Note that the first of these mechanisms is also useful in Attack 1—instead of hard-coding the number that the rootkit must dial, an attacker can transmit the number that the infected phone must dial via a text message. The attacker can also enable/disable the rootkit's malicious functionality via text messages, in effect allowing the attacker to remotely control the rootkit.

Our prototype rootkit intercepts text messages by hooking the kernel's `read` and `write` system calls. This is achieved by modifying the corresponding entries in the system call table to point to rootkit code. Consequently, each read/write by a user-space application is intercepted by the rootkit. Text messages are received and processed by the GSM device. User-space SMS applications constantly poll the GSM device to check for new messages. When the SMS application attempts to `read` from the GSM device, the rootkit's code intercepts this operation and reads data from the `read_buf` buffer of the GSM device. Text message notifications are stored in the buffer contain a string "`+CMTI=simindex`," where "`simindex`" is the serial number of the text message. Upon finding this substring, the rootkit parses the message to determine whether it contains an instruction from the attacker (in some pre-determined format). Each message is also associated with a status bit that determines whether the message is new or has already been read. Our prototype rootkit carefully ensures that any new messages that are not from the attacker will remain marked as unread, and can subsequently be processed by user-space SMS applications, thereby allowing the rootkit to operate stealthily.

If the rootkit intercepts a message from the remote attacker, it attempts to obtain location information from the GPS device. As before, the rootkit can easily obtain location information from the `read_buf` buffer of the `tty_driver` structure associated with the GPS device. Note that the rootkit can obtain location information even if the user has disabled GPS. This is because the rootkit operates in kernel mode, and can therefore enable the device to obtain location information, and disable the device once it has this information.

Having obtained location information, the rootkit constructs a text message and sends the message by configuring the GSM modem to text mode, and issuing a `AT+CMGS` command (to send the message).

Social Impact.
Protecting location privacy is an important problem that has received considerable recent attention in the research community. By compromising the kernel to obtain user location via GPS, this rootkit defeats most existing defenses to protect location privacy. Further, the attack is stealthy. Text messages received from and sent to the attacker are not displayed immediately to the victim. The only visible trace of the attack is the record of text messages sent by the victim's phone, which is recorded by the service provider. The victim may detect the attack only when he receives his monthly statement and notices text messages sent to an unknown number. However, many users rarely notice small increases to their monthly statement, thereby allowing occasional attacks to proceed undetected for long periods of time.

Attack 3: Denial of Service via Battery Exhaustion

Goal.
This attack exploits power-intensive smart phone services, such as GPS and Bluetooth, to exhaust the battery on the phone. The rootkit operates by powering the GPS and Bluetooth devices. To operate stealthily, the rootkit reverts the GPS and Bluetooth devices back to their original states when the user queries the status of these devices. For example, if both devices were initially powered off, then the rootkit powers the devices off again. Consequently, these devices will appear to be powered off to a user who queries their status, thereby allowing the attack to proceed stealthily.

This rootkit was motivated by and is similar in its intent to a recently proposed attack that stealthily drains a smart phone's battery by exploiting bugs in the MMS interface [40]. However, the key difference is that the rootkit achieves this goal by directly modifying the smart phone's operating system.

Background.
The GPS and Bluetooth devices can be toggled on and off by writing a "1" or a "0," respectively, to their corresponding power device files. For example, the GPS can be turned on from user space using the following shell command:
```
echo 1 >/sys/class/i2c-adapter/i2c-0/0-0073/
neo1973-pm-gps.0/pwron.
```
User-space applications typically determine the status of these devices by querying this file.

Attack Description.
The rootkit interposes upon user-space commands to toggle the GPS and Bluetooth devices. To remain stealthy, the rootkit restores the original state of these devices when a user

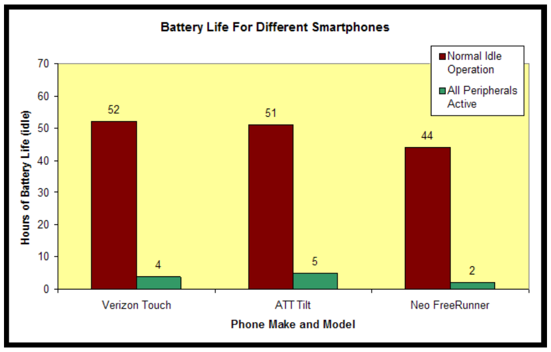

Figure 5. Denial of service via battery exhaustion. This figure shows how the battery life degrades in different phone models when the GPS and Bluetooth devices are powered on.

attempts to view their status. Most users typically turn these devices off when they are not in active use because they are power-intensive.

The rootkit operates by overwriting kernel function pointers for the `open` and `close` system calls in the system call table, making them point to rootkit code. Each time a file is opened or closed in user space, rootkit code is executed. When an `open` system call is executed, the rootkit examines if the file being opened corresponds to the power device files of the GPS or Bluetooth devices. If so, it restores the state of these devices to their previously saved states. Upon invocation of the `close` system call, the devices are powered on again. Consequently, the devices are always on, except when the user actively queries the status of these devices.

Social Impact.
This attack quickly depletes the battery on the smart phone. In our experiments, the rootkit depleted the battery of a fully charged and infected Neo Freerunner phone in approximately two hours (the phone was not in active use for the duration of this experiment). In contrast, the battery life of an uninfected phone running the same services as the infected phone was approximately 44 hours (see Figure 5). We also simulated the effect of such a rootkit on the Verizon Touch and ATT Tilt phones by powering their GPS and Bluetooth devices. In both cases, battery lifetime reduced almost ten-fold. Because users have come to rely on their phones in emergency situations, this attack results in denial of service when a user needs his/her phone the most.

Although our prototype rootkit employs mechanisms to hide itself from an end user, this attack is less stealthy than Attacks 1 and 2. For example, a user with access to other Bluetooth-enabled devices may notice his smart phone is "discoverable," causing him to suspect foul play. Nevertheless, we hypothesize that the vast majority of users will suspect that their phone's battery is defective and replace the phone or its battery.

DISCUSSION

In this section, we discuss various ways in which rootkits can be delivered and installed on a victim's phone and the persistence of rootkits on the compromised phone. We then discuss the possibility of rootkits in microkernel operating systems, such as Symbian. We conclude with a discussion of mechanisms that can help detect rootkits on a smart phone.

Rootkit Delivery and Persistence

Rootkits can be delivered to smart phones using many of the same techniques as used for malware delivery on desktop machines. A study by F-Secure showed that nearly 79.8% of mobile phones infections in 2007 were as a result of content downloaded from malicious websites. Among the other major contributors to malware delivery were Bluetooth and text messages [27]. Rootkits can also be delivered to smart phones via email attachments, spam, illegal content obtained from peer-to-peer applications or by exploiting vulnerabilities in existing applications.

Post delivery, rootkits require privileged access (*e.g.,* root privileges) to infect the operating system by modifying its code and data. The Neo Freerunner phone used in our experiments ran the OpenMoko Linux distribution, which directly executed applications with root privileges. Therefore, unsafe content downloaded on this phone automatically obtains root privileges. Even on operating systems that do not run applications with root privileges, an attacker may exploit vulnerabilities in application programs and the operating system to obtain elevated privileges to install rootkits. Such vulnerabilities are not uncommon even in carefully engineered systems; for example, a vulnerability in Google's Android platform allowed command-line instructions to execute with root privileges [7].

Once installed, the goal of a rootkit is to maintain long-term control over the compromised system. Rootkits typically do so by installing themselves as kernel modules that are loaded each time the operating system is booted. Smart phone rootkits can also employ the same mechanism to retain long-term control over infected phones. However, because this approach leaves a disk footprint (*i.e.,* the kernel module containing the rootkit), it may possibly be detected by antivirus tools. Sophisticated rootkits avoid this problem by directly modifying code and data in kernel memory and do not leave a disk footprint. Although such rootkits only persist until the system is rebooted, they are effective on desktop computers, which are often not rebooted for several days or months at a time. Because smart phones are powered off more often (or may die because the battery runs out of charge), rootkits that directly modify kernel memory may only persist on smart phones for a few days. In such cases, an attacker can re-infect the phone (or, for rootkits that spread via Bluetooth, infected phones in the vicinity of a victim may re-infect the victim). Nevertheless, the social consequences of smart phone rootkits mean that they can seriously affect end user security even if effective only for short periods of time.

Rootkits on Microkernel Operating Systems

The prototype rootkits discussed in this paper were implemented on a macrokernel operating system (Linux). Such operating systems execute large amounts of code in kernel mode. For example, device drivers on Linux, which constitute about 3.1 million lines of code (approximately 60% of the entire code base), execute in kernel mode. Rootkits often masquerade as device drivers or other kernel modules. Once installed, they have ready access to kernel code and data.

In contrast, microkernel operating systems execute only a small amount of code in kernel mode. Device drivers and other modules, such as the networking subsystem and file system, are executed as user-mode processes. Because microkernels limit the amount of code running in kernel mode, they present a much smaller attack surface for kernel-level rootkits than macrokernel operating systems. As such, rootkits have historically targeted only macrokernel operating systems, such as Linux, Windows and OS X, that dominate the market; to our knowledge, rootkits have not been developed for microkernel operating systems.

The Symbian operating system, which has approximately 50% of the market share for smart phone operating systems, borrows features from both microkernel and macrokernel operating systems. Device drivers and memory management code execute in kernel-mode, as in macrokernels; however, file systems and the networking subsystem are implemented as user-space processes, as in microkernels [38, 13].

Rootkits that are distributed as device drivers can infect a Symbian smart phone in the same way as they do other macrokernel operating systems. However, rootkits can also easily target the microkernel-like components of Symbian, including the file and networking subsystem by infecting the user-space processes that implement these subsystems. For example, antivirus tools rely on the view provided by the file system to scan files on disk. A rootkit on a could maliciously modify the file system process to hide itself from antivirus tools.

Techniques to Detect Smart Phone Rootkits

Intrusion detection tools, such as antivirus software, are effective at detecting and eliminating viruses, worms and spyware. Such tools are also effective at detecting mobile malware on smart phones. However, these tools typically rely on the operating system to provide critical services, such as access to files, and thereby implicitly trust the operating system. In rootkit-infected systems, this trust is misplaced. Because the rootkit runs in kernel mode, it can easily evade detection by antivirus tools, which run as user-mode applications. Consequently, rootkit detection tools must be isolated from the operating system that is being monitored. Rootkit detectors that have been developed for desktop computers therefore execute on a secure coprocessor [50, 28] or are isolated using virtualization [26, 39].

Smart phones available on the market today are not equipped with secure co-processors. Consequently, virtualization offers the only practical alternative to implement rootkit detection tools on smart phones. A number of commercial efforts are currently underway to build virtual machine monitors for smart phones [14, 8, 10, 15]. The main goal of virtualizing smart phones is to enable users to have multiple personalities on the phone. For example, the same phone can be used with multiple accounts and providers, such as a corporate account and a personal account. Rootkit detection tools can leverage these virtual machine monitors to isolate themselves from the smart phone's operating system. For example, a rootkit detector can execute in a separate virtual machine and monitor the memory of the smart phone's operating system. However, most existing rootkit detection tools [28, 36, 37, 17] employ compute-intensive algorithms that require them to periodically fetch and scan kernel memory snapshots of the operating system being monitored. Such monitoring can potentially drain the battery of the phone, thereby bringing into question the practicality of such rootkit detection tools, and underscoring the need for new, energy-efficient techniques to detect rootkits on smart phones.

An alternative approach to detect rootkits is to use trusted hardware, such as a TPM chip [44]. TPM chips can compute and securely store *integrity measurements* of a software stack. When used in combination with an integrity measurement protocol [25, 41], these measurements can detect the presence of unwanted code (such as rootkits) in the software stack. The Trusted Computing Group has recently announced the specification of trusted hardware for mobile phones (MTM—the Mobile Trusted Module), and an increasing number of smart phone vendors are beginning to deploy such hardware on their phones [43]. However, trusted hardware can only detect rootkits that modify immutable code and data. Recent research has demonstrated rootkits that can achieve malicious goals by modifying mutable data in the operating system [19, 36, 17]. Such rootkits can remain undetected even on MTM-enabled smart phones.

RELATED WORK

We focus our description of related work to two main areas: (1) rootkits and rootkit defenses for desktop computers; and (2) mobile malware and detection tools.

Rootkits and Detection Tools for Desktops

Rootkit detection tools on desktop computer systems are largely centered around known techniques used by rootkits to hide their presence. Early rootkits operated by replacing system binaries and shared libraries on disk with Trojaned versions, which would hide malicious objects owned by the attacker. Tools such as Tripwire [29] and AIDE [2] detected such rootkits by checking the integrity of system files. Other tools such as Strider Ghostbuster [20] used a cross-view based approach to detect user-mode rootkits. Cross-view based techniques operate by using different system interfaces to obtain answers for a query, and comparing the query results obtained from these interfaces. For example, a cross-view based detector can identify hidden processes by observing differences between the output of the `ps` utility and the kernel's internal representation of running processes.

More recently, rootkits have evolved to modify the kernel. Such rootkits either modify kernel code or data structures that store control data, such as the system call table or function pointers. Recent work has shown that non-control data in the kernel can be modified as well to carry out highly stealthy attacks on the system [19]. Several tools have been developed to detect such rootkits—they typically operate by ensuring invariants on kernel code and data [28, 37, 36, 17].

An alternative to the above techniques are the ones that scan kernel modules offline and determine whether they are malicious. These include both signature-matching techniques as employed by most commercial malware detection tools, and symbolic execution tools [48, 32], which statically approximate the behavior of a kernel module to determine whether it likely affects key kernel data structures.

Mobile Malware and Defenses

As phones have evolved to become full-fledged computing devices, they have also become attractive targets for malware such as viruses, worms and Trojans [22, 33, 45]. Malware typically exploits vulnerabilities on interfaces unique to the smart phone such as Bluetooth [49] or vulnerabilities in applications running on the phone.

Tools to detect mobile malware have adapted well-known techniques used on desktops, such as signature and behavior-based detection algorithms, to operate in a resource-constrained environment. These algorithms use lesser memory, run faster, and consume lesser battery power than their desktop counterparts [47, 21]. Other approaches to detect mobile malware monitor and analyze untrusted software for anomalous behaviors that deplete energy [30].

Mobile phones are likely targets of cross-service attacks. For example, an attack may be perpetrated and downloaded through the Internet onto a smart phone via its data plan, and may access its telephony subsystem. Such threats are possible because of lax security mechanisms currently employed by mobile phones. For example, an arbitrary user-space process on the phone can issue AT commands to the GSM device [46]. Such cross-service attacks can be prevented by labeling user space files and resources and enhancing access control mechanisms on smart phone operating systems [34, 35]. However, these solutions are ineffective against the rootkits that we propose in this paper. This is because rootkits achieve their malicious goals by directly modifying kernel data.

CONCLUSIONS

The evolution of malware has historically been an arms race between attackers and defenders. As defenders improve the ability of their tools to detect malicious activities, attackers employ new techniques to evade detection. Rootkits evade detection by compromising the operating system, thereby allowing them to defeat user-space detection tools and operate stealthily for extended periods of time. This paper demonstrated that kernel-level rootkits can exploit smart phone operating systems, often with serious social consequences.

The popularity of the mobile platform has already caught the attention of attackers, who have increasingly begun to develop and deploy viruses and worms that target these platforms. As these threats gain notoriety, so will the power of tools to detect these threats. We believe that this trend, combined with the increasing complexity of operating systems on modern smart phones, will push attackers to employing rootkits to achieve their malicious goals. We therefore conclude with a call for research on tools and techniques to effectively and efficiently detect rootkits on smart phones.

REFERENCES

1. Adore-0.42 rootkit. `http://packetstormsecurity.org/UNIX/penetration/rootkits/`.
2. Advanced intrusion detection environment. `http://sourceforge.net/projects/aide`.
3. Apple: one million iPhones sold. `http://www.cnet.com.au/apple-1-million-iphones-sold-339281969.htm`.
4. The bluepill project. `http://bluepillproject.org/`.
5. F-secure warns of mobile malware growth. `http://www.vnunet.com/vnunet/news/2230481/f-secure-launches-mobile`.
6. Fu rootkit. `http://www.rootkit.com/project.php?id=12`.
7. Google fixes android root-access flaw. `http://www.zdnetasia.com/news/security/0,39044215,62048148,00.htm`.
8. Green hills platform for secure mobile devices. `http://www.ghs.com/mobile/index.html`.
9. Mcafee mobile security report 2008. `http://www.mcafee.com/us/research/mobile_security_report_2008.html`.
10. Open kernel labs. `http://www.ok-labs.com/`.
11. Openmoko Neo FreeRunner. `http://wiki.openmoko.org/wiki/Neo_FreeRunner`.
12. Smartphones will soon turn computing on its head. `http://news.cnet.com/8301-13579_3-9906697-37.html`.
13. Symbian OS device driver model. `http://www.symbian.com/Developer/techlib/v70docs/SDL_v7.0/doc_source/BasePorting/DeviceDrivers/SymbianOSDeviceDriverModel.guide.html`.
14. Vmware mobile virtualization platform. `http://www.vmware.com/technology/mobile/`.
15. Wind river systems. `http://www.windriver.com/`.
16. Rootkits, part 1 of 3: A growing threat, April 2006. MacAfee AVERT Labs Whitepaper.
17. Vinod Ganapathy Arati Baliga and Liviu Iftode. Automatic inference and enforcement of kernel data structure invariants. In *ACSAC '08: Proceedings of the Annual Computer Security and Applications Conference*, 2008.
18. Arati Baliga, Liviu Iftode, and Xiaoxin Chen. Automated containment of rootkits attacks. *Computers & Security*, 27(7-8):323 – 334, 2008.
19. Arati Baliga, Pandurang Kamat, and Liviu Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2007.
20. Doug Beck, Binh Vo, and Chad Verbowski. Detecting stealth software with strider ghostbuster. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, 2005.
21. Abhijit Bose, Xin Hu, Kang G. Shin, and Taejoon Park. Behavioral detection of malware on mobile handsets. In *MobiSys '08: Proceeding of the 6th international conference on Mobile systems, applications, and services*, 2008.

22. David Dagon, Tom Martin, and Thad Starner. Mobile phones as computing devices: The viruses are coming! *IEEE Pervasive Computing*, 3(4):11–15, 2004.

23. Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell. Cloaker: Hardware Supported Rootkit Concealment. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

24. William Enck, Patrick Traynor, Patrick Mcdaniel, and Thomas La Porta. Exploiting open functionality in sms-capable cellular networks. In *CCS '05: In Proceedings of the ACM Conference on Computer and Communication Security*, 2005.

25. Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *SOSP03: ACM Symposium on Operating System Principles*, October 2003.

26. Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, 2003.

27. M. Hypponen. The state of cell phone malware in 2007. `http://www.usenix.org/events/sec07/tech/hypponen.pdf`.

28. Nick L. Petroni Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Security '04: Proceedings of the USENIX Security Symposium*, 2004.

29. Gene H. Kim and Eugene H. Spafford. The design and implementation of tripwire: a file system integrity checker. In *CCS '94: Proceedings of the 2nd ACM Conference on Computer and communications security*, 1994.

30. Hahnsang Kim, Joshua Smith, and Kang G. Shin. Detecting energy-greedy anomalies and mobile malware variants. In *MobiSys '08: Proceeding of the 6th international conference on Mobile systems, applications, and services*, 2008.

31. Samuel King, Peter Chen, Yi-Min Wang, Chad Verblowski, Helen J. Wang, and Jacob R. Lorch. Subvirt: Implementing malware with virtual machines. In *SP06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, May 2006.

32. Christopher Kruegel, William Robertson, and Giovanni Vigna. Detecting kernel-level rootkits through binary analysis. In *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference*, 2004.

33. Neal Leavitt. Mobile phones: The next frontier for hackers? *Computer*, 38(4):20–23, 2005.

34. C. Mulliner, G. Vigna, D. Dagon, and W. Lee. Using Labeling to Prevent Cross-Service Attacks Against Smart Phones. In *DIMVA '06: Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, July 2006.

35. D. Muthukumaran, M. Hassan, V. Rao, and T. Jaeger. Protecting telephony services in mobile phones. Technical Report NAS-TR-0096-2008, September 2008.

36. Jr. Nick L. Petroni, Timothy Fraser, AAron Walters, and William A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Security '06: Proceedings of the USENIX Security Symposium*, 2006.

37. Jr. Nick L. Petroni and Michael Hicks. Automated detection of persistent kernel control-flow attacks. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

38. J. Pagonis. Overview of Symbian OS hardware interrupt handling. `https://developer.symbian.com/wiki/download/attachments/50987025/HwInterrupt.pdf`.

39. Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *IEEE S&P '08: Security and Privacy, IEEE Symposium on*, Los Alamitos, CA, USA, 2008.

40. Radmilo Racic, Denys Ma, , and Hao Chen. Exploiting MMS Vulnerabilities to Stealthily Exhaust Mobile Phone's Battery. In *SecureComm '06: Proceedings of the Second International Conference on Security and Privacy in Communication Networks*, August 2006.

41. Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *Security04: Proceedings of the 2004 USENIX Security Symposium*, August 2004.

42. Sherri Sparks Shawn Embleton and Cliff Zou. Smm rootkits: a new breed of os independent malware. In *SecureComm '08: Proceedings of the 4th International Conference on Security and Privacy in Communication Networks*, 2008.

43. TCG. Trusted computing group: Mobile trusted computing platform. `https://www.trustedcomputinggroup.org/groups/mobile`.

44. TCG. Trusted Computing Group: Trusted Platform Module (TPM) Specifications. `https://www.trustedcomputinggroup.org/specs/TPM/`.

45. Sampo Töyssy and Marko Helenius. About malicious software in smartphones. *Journal in Computer Virology*, 2(2):109–119, 2006.

46. P. Traynor, V. Rao, T. Jaeger, P. McDaniel, and T. La Porta. From mobile phones to responsible devices. Technical Report NAS-TR-0059-2007, January 2007.

47. Deepak Venugopal and Guoning Hu. Efficient signature based malware detection on mobile devices. *Mobile Information Systems*, 4(1):33–49, 2008.

48. Jeffrey Wilhelm and Tzi cker Chiueh. A forced sampled execution approach to kernel rootkit identification. In *RAID*, 2007.

49. Guanhua Yan, Hector D. Flores, Leticia Cuellar, Nicolas Hengartner, Stephan Eidenbenz, and Vincent Vu. Bluetooth worm propagation: mobility pattern matters! In *ASIACCS '07: Proceedings of the 2nd ACM symposium on Information, computer and communications security*, 2007.

50. Xiaolan Zhang, Leendert van Doorn, Trent Jaeger, Ronald Perez, and Reiner Sailer. Secure coprocessor-based intrusion detection. In *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop: beyond the PC*, 2002.