Proceedings of the 2007 IEEE
Workshop on Information Assurance
United States Military Academy, West Point, NY, 20-22 June 2007

On the Effort to Create Smartphone Worms in Windows Mobile

Michael Becher, Felix C. Freiling, Boris Leider

Abstract— **Compared to what we know about malware for desktop and server systems, we know almost nothing about malware for smartphones and similar mobile devices. With the growing ubiquity of such devices, they are becoming increasingly popular as attack targets. The ultimate target for an attacker would be to create a *smartphone worm* which autonomously spreads between devices. In this paper we focus on devices running the *Windows Mobile* operating system. In particular we investigate the effort needed to create a smartphone worm for the recent version 5 of Windows Mobile. We measure this effort in work time by a skilled individual using modern tools and software engineering techniques. We found that it takes roughly 600 work hours (14 weeks of full time work) to come very close to the target. Our work highlights the strengths and weaknesses of Windows Mobile version 5 over the previous version 2003 as well as the general difficulties of attacking ARM-based architectures. The insights from our study can be used to estimate lower bounds of cost-to-break metrics for current and future versions of Windows Mobile.**

I. Introduction

The increasing ubiquity of smartphones and other mobile devices has turned them into interesting attack targets. The increasing processing power and the use of desktop-like operating systems on these devices has made them susceptible to the same threats of malware which prevail throughout the desktop and server world. Malware infection for such devices can be categorized according to the user interaction, that is necessary for the malware to infect the system. This results in four distinct classes:

- The most benign interaction is asking the user, whether it is allowed to be installed or to spread, clearly indicating its possible malicious behavior. This is the typical behavior of proof-of-concept malware.
- The next category are the standard questions at installation procedures for (unsigned) software. The user might be accustomed to them because of previous installation procedures, that he performed. This is the standard way, how trojan horses get installed, usually by seducing the user with social engineering techniques, that he really wants to install the offered software (e.g., the "Free Worldcup After-Party Ticket - just install").
- The third category is an action, that is common behavior when using a mobile phone, e.g., reading an incoming message of the multimedia messaging service (MMS). If a virus spreads by sending MMS messages to the contacts of the user, the recipients see an MMS message from a known sender, and it is probable, that they would open it.
- The most dangerous type of malware is a *smartphone worm*, that is able to spread without any user interaction. This would be the worst case concerning mobile phone security, but as of today, no such type of malware is known.

Michael Becher: University of Mannheim, Germany.
Felix C. Freiling: University of Mannheim, Germany.
Boris Leidner: RWTH Aachen University of Technology, Germany.

In this paper, we focus on malware for the popular Windows Mobile family of operating systems for mobile devices. Only few instances of malware for Windows Mobile are known today. *Dust*, which was reported about in July 2004, is the first virus that appeared. Details have been published, including its source code [1]. One month later, in August 2004, the first trojan called *Brador* showed up and was analyzed as well [2]. The *Crossover* virus appeared in February 2006. It is a binary, that executes on Windows PC and on Windows Mobile environments, and — when executed on a PC — waits for an ActiveSync connection to infect the connected device [3]. According to the classification above, none of these programs can be called a smartphone worm. However, since this class of malware is most dangerous, we need to actively research characteristics and countermeasures to learn more about the associated threats.

Research in malware for mobile devices seems not very attractive at first sight, because many standard techniques known from the desktop and server world do not work in the new environment. As an example, the hardware architecture used in smartphones (ARM) is very different from the well-understood Intel x86 family. Operating system concepts like processes, scheduling and virtual memory are also quite different in mobile devices, even between different versions of mobile operating systems. Thus, it is neither clear how easy it is to stage standard attacks from the PC world in mobile environments, nor how easy it is to transfer known attacks for one version of Windows Mobile to the next.

In this paper, we investigate the question of how easy or difficult it is to create a smartphone worm for the recent version 5 of Windows Mobile. We do this by measuring the effort needed by a skilled individual to create such a worm using standard methods of penetration testing, standard security tools, and software engineering methods.

We managed to build a prototype worm that can spread autonomously in case a vulnerability is given in a Windows

ISBN 9-9999-9999-9/99/ $20.00

Mobile service that is accessible over the network. The working effort encompasses the following tasks:
1. Building a worm toolkit, i.e., a software library that contains shellcode and worm spreading functionality. This is what we call the *constant part* of the attack. It is developed once and can be used (possibly in a slightly adapted form) for any future vulnerability. The effort required for achieving this was about 13 weeks of full time work.
2. Finding a vulnerability in the network interfaces of Windows Mobile 5. This task included attempts to execute known attacks for Windows Mobile 2003 as well as applying the technique of *fuzz testing* (fuzzing) to the WLAN network stack of Windows Mobile 5 to find buffer overflows. This is what we call the *variable part* of the attack. The effort necessary for the second task can only be approximated because we terminated our efforts after one week of full time work without finding a new vulnerability that could be exploited by a worm. We however found two denial-of-service vulnerabilities which we report upon.

To summarize, we found that it takes roughly 600 work hours (14 weeks of full time work) to come very close to the target of building a smartphone worm. The benefit of our work is that it helps to understand the difficulties adversaries face when attacking Windows Mobile. It also highlights the strengths and weaknesses of Windows Mobile version 5 over version 2003 as well as the general difficulties of attacking ARM-based architectures. Most importantly, however, the insights from our study can be used to estimate lower bounds of cost-to-break metrics [4] for current and future versions of Windows Mobile. We are unaware of any other work that has measured attack effort as precisely as we have.

The paper is structured as follows: Section II lists previous work in Windows Mobile vulnerability research. The development of a building block as the constant part of a smartphone worm is presented in Section III. The variable part of development, restricted to the search for buffer overflows by employing fuzzing, is presented in Section IV.

II. Related Work

An extended documentation of this work is available [5], and there is a recent article describing smartphone malware and worms [6].

Shellcode development in ARM assembler is essential for a smartphone worm for Windows Mobile. This was introduced by Hurman [7], San [8], Mulliner [9], and Fogie [10]. The results of these works can not all be transferred to Windows Mobile 5 devices, because of some changes in the operating system. Furthermore, smartphone worms need more reliable shellcode. These problems are worked out in this paper.

Fuzz testing or *fuzzing* means automatically searching for vulnerabilities in software. The attacker generates input, that complies with the structure of the expected input, but contains random data, that is intended to make the targeted process fail. If this happens, the next step is analyzing the possibility of using this failure for injecting malicious code into the process.

Recently, a buffer overflow in the MMS handling process was found using fuzzing [11]. This work focused on Windows Mobile 2003. The user must read an incoming malicious MMS message for triggering the exploit. So the vulnerability is of the third type, whereas this work focused on the fourth type of malware, i.e., on smartphone worms.

This paper aims at measuring the effort, that is needed to find a vulnerability. The intrusion process of breaking a system can be seen as subdivided into three phases [12]. The first phase is a *learning phase*, where information about the targeted system is collected. The *standard attack phase* consists of a number of fairly straightforward actions and standard tools for breaking a system, with a rather high probability of success. The *innovative phase* consists of inventing new methods for breaking a system. The time between successful breaches increases, because it can never be said, after how much time the ideas of an attacker will lead to an effective attack. This model can be applied to our situation. When starting our efforts, we were in the innovative phase of the model. The situation changes with the publication of our results, because parts of them can now be taken as successful (i.e., standard) attacks on the system.

III. Building Blocks for a Smartphone Worm

This Section describes the development of a proof-of-concept smartphone worm that is able to spread on Windows Mobile 5 devices over WLAN. The problems will be solved, that arise due to the nature of Windows Mobile 5 and the ARM processor architecture. The result will show that the threat of smartphone worms is real.

A. Windows Mobile 5 and ARM basics

Windows Mobile 5 is different from other operating systems in some parts. It runs on processors that implement the ARM architecture. All processes in Windows Mobile reside in one large 4 GB virtual address space. Therein, every process has its own process slot which is represented by the most significant byte of the memory addresses of the process. The process slot is set by the operating system at the start of the process. Additionally, a process can refer to its own slot by using the process slot zero. This will be useful later on.

Windows Mobile 5 runs either in user or kernel mode. Some memory areas are protected, so that only processes running in kernel mode can access these areas. Older devices with older versions of Windows Mobile were compiled for the *full-kernel mode*. Full-kernel mode means that all processes run in kernel mode for the benefit of higher execution speed.

ISBN 9-9999-9999-9/99/ $20.00 ©2007 IEEE

The interface between processes and the operating system are system calls. Applications do not use system calls directly, but use the Windows API functions that encapsulate or extend system calls. The Windows API comes in libraries like `coredll.dll` or `winsock.dll`.

In Windows Mobile 5 code signing is used to determine if an application (or another executable module) is trusted or not. Signatures can be obtained by participating in Microsoft's Mobile2Market program [13]. There are two versions of Windows Mobile 5: for Pocket PCs and for Smartphones. *Windows Mobile for Smartphones* has a tighter security model than *Windows Mobile for Pocket PCs*. On Pocket PCs, all applications run in privileged mode. If there is no valid signature, the user is asked to confirm the execution. Smartphones distinguish between applications that are signed for privileged execution, application that are signed for normal execution, and applications that are not signed or invalid signed. Applications signed for privileged execution may additionally write to all parts of the Windows registry, can read and write system files, and may access a privileged part of the Windows API, the *Trusted API* [13].

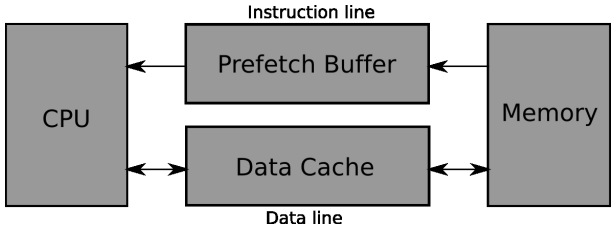

Fig. 1. Separation of memory access

ARM processors have separated memory buses for instructions and data (see Fig. 1). Instructions are fetched over the instruction bus, but data reading and writing is done over the data bus. Both buses have their own caches and buffers, and they work independently from each other [14].

B. Infection

The infection routine is the first part of the building block. It injects code into the mobile device by exploiting a security vulnerability, and gains access to the operating system and its resources. The injected code is the *shellcode* and its development is the main task in this part. The GNU Assembler `GAS` [15] is used for compilation.

For the development of shellcode for a smartphone worm, a program is needed that can be exploited over wireless network, e.g. WLAN. The program contains a stack-based buffer overflow, a security vulnerability, which is a very common programming error in programming languages like C/C++. The vulnerability is used to create a proof-of-concept for exploitation of this class of vulnerabilities.

B.1 Stack-based buffer overflows

Exploiting stack-based buffer overflows in Windows Mobile 5 works in principle the same way as it does in other operating systems. A local buffer in a function is filled with user data that exceeds the buffer boundary. This way, memory behind the buffer can be overwritten. That is also true for the memory address that contains the saved return address of the function. If this address can be manipulated, the program will continue its execution at this address after the running function has returned. The return address may now point to an address that is controlled by the user. Usually, this is the address of the buffer that was overwritten. Therefore, machine code is put at the beginning of the buffer. The machine code (or shellcode) can then be executed by the CPU.

B.2 Return address

The first problem is, that the buffer address for the running process of the vulnerable program must be known. The process resides with all its code, stack and heap memory in a process slot. This slot was assigned by the operating system and cannot be known by remote attackers. The most significant byte of the buffer address represents the slot number of the process. Because the memory of a running process can be addressed by the slot number zero, the real slot number is not needed. The most significant byte of the address can be set to zero.

B.3 Shellcode stages

The user input that fills the buffer is often limited to a defined size. Therefore, shellcode must be as small as possible. Very small shellcodes exist for operating systems like Linux and UNIX. 40 bytes are sufficient to take over such a system [16]. Shellcode for Windows Mobile is bigger for several reasons. All processor instructions in the ARM architecture are four bytes long. Note, that in the x86 architecture some simple instructions just need two bytes. Another reason is the load-store architecture used by ARM processors. Memory can only be altered if it is first transferred to a register, then manipulated, and finally written back to memory. That is why all modifications on memory take at least three instructions.

The interface between user processes and the operating system is the last reason for bigger shellcode. In Linux, a shell interpreter for the execution of arbitrary commands exists. Unfortunately, in Windows Mobile (and other Microsoft systems) the command interpreter is not as powerful and useful as a shell interpreter in Linux [17]. Access to the Windows API is absolutely essential and many instructions are required to use it.

If there is not enough space in the buffer, then the size and complexity of payloads is limited. Because a worm is a complex program, a solution for that problem is needed. The solution is to divide the shellcode in two stages. The

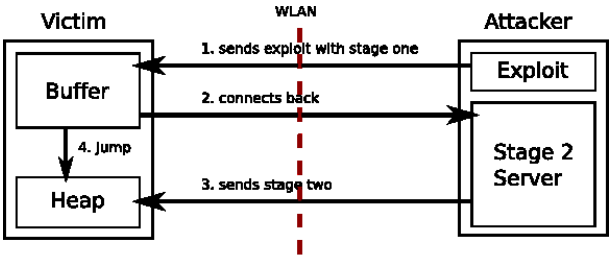

Fig. 2. Exploitation with staged shellcode

first stage contains the smaller part of the shellcode. Its only purpose is to download the second stage from the attacking host, save it to heap memory, and execute it. The second stage is practically not limited in size, and may hold even large worm programs. Fig. 2 depicts the functioning of the staged exploitation.

B.4 Zero-free shellcode

If a buffer overflow occurs, the user input is copied to the buffer. The buffer will be overflowed when the user input is too long. Often the user input is a string, and the string is copied by functions like `strcpy` or `sprintf`. These functions assume, that strings are zero-terminated, that means their end is marked by a trailing zero-byte. If they see a zero-byte, they will stop the copy process. But the shellcode may contain many zero-bytes. Instructions like `b` or `bl` (for branching) often contain zero-bytes and almost all instructions that use the register `r0` have a zero-byte. It is hard to write zero-free shellcode by hand, so a generic way to produce such code must be found.

It is best practice to encode the shellcode [17]. For this, the first stage is again divided into two parts. The first part is a decoder that must be implemented zero-free once. The second part is a payload containing encoded shellcode. The encoded shellcode is also zero-free, but has to be decoded before it can be executed. The XOR function and a four byte key is used to encode and decode the payload. The shellcode payload can be developed without watching out for zero-freeness, because the encoding will do all the work. The key must be chosen wisely, so that encoded instructions do not lead to new zero-bytes. When the first stage is executed, the decoder decodes the encoded payload of the first stage in place and executes it.

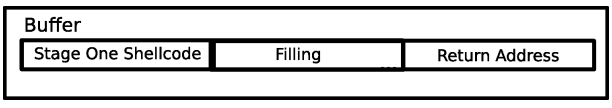

Fig. 3. Exploit string

Not only the shellcode must be zero-free, but the whole exploit string that is passed to the buffer (see Fig. 3). Above was said, that the return address will contain a zero-byte. This is necessary, if the true process slot shall not be guessed. Fortunately, the zero-byte is the most significant and therefore last byte in the address. Furthermore, the address is the last unit of our exploit string, so the zero-byte is the last byte in the exploit string. Because all strings have to end with one zero-byte, the string functions will copy it to the buffer.

The second stage of the shellcode does not need to be zero-free. It will not be copied by string functions.

B.5 Selfmodifying code on ARM

The first stage downloads the seconds stage from network, writes it to memory and executes it. This is called *selfmodifying code*. As mentioned before, ARM processors have two memory buses for instruction and data access (Fig. 1). When the first stage decodes the encoded payload, instructions are written via the data bus. On execution of the decoded payload, instructions are fetched via the instruction bus. This may lead to inconsistencies for some reasons. The instruction bus has a prefetch cache that will fetch instructions in advance. If instructions are prefetched before the payload is decoded, the CPU might execute the instructions from the prefetch buffer, even if the payload in memory is already decoded. That is why the prefetch cache has to be invalidated.

The second reason is bound to write-back-caches. When the first stage payload is decoded, it may happen, that the decoded instructions were written to the data cache, but are not yet copied to memory. On this occasion, the unaltered instructions will be executed, because the instruction cache does not know that data in the data cache is still to be written. To solve this problem, the data caches must be flushed by hand.

Finally, there may be a data write buffer in addition to the data cache. The write buffer will be filled whenever data has to be written to memory. Filling the write buffer is faster than writing to memory, so the CPU may do other things while the write buffer is still flushing its data to memory. If not all data is written back to memory, again undecoded instructions are executed.

The solution to all of this problems are *Instruction Memory Barriers* (short *IMBs*). IMBs are sets of instructions that flush buffers, invalidate caches, and do anything that is necessary for execution of selfmodifying code. These instructions may differ from ARM processor to ARM processor.

Hurman [7] uses the following instructions for the invalidation and flushing of caches and buffers:

```
mcr p15, 0, r0, c7, c10, 4
mrc p15, 0, r1, c2, c0, 0
mov r1, r1
```

However, these instructions are not sufficient. One test device failed while executing the second stage, because not all

instructions were written to memory yet. Also, the processor needs to enter system mode for this kind of instructions. That means Windows Mobile has to run in kernel mode.

Selfmodifying code is used by all operating systems. When an operating system executes a user program, its process code, stack and data is copied to memory. After that, the process code is executed. Thereby, in the overall context the system modifies itself. So selfmodification is a feature of operating systems. This means, that operating systems need IMBs for all processors they are supposed to run on. In Windows Mobile this is the Windows API function `CacheRangeFlush` [13].

```
void CacheRangeFlush(
  LPVOID pAddr,
  DWORD dwLength,
  DWORD dwFlags
);
```

If the first two arguments are set to zero and the third is set to `CACHE_SYNC_ALL`, all caches and buffers will be invalidated or flushed. Shellcode can now be executed reliably on all ARM processors. This is a major advantage compared to the approach of Hurman.

B.6 Accessing the Windows API

The shellcode needs Windows API functions to fulfill some tasks. It needs functions for networking, allocation of heap memory, and—last but not least—the function `CacheRangeFlush` for flushing of caches and buffers. The easiest way to use this API functions is to find out their addresses in the device memory and use them directly in the shellcode. There is just one problem with that: the function addresses may be different for each Windows Mobile device. Because a smartphone worm is supposed to work on as many devices as possible, it is more suitable to find out the addresses at runtime.

The approach of Hurman [7] and San [8] uses a structure that resides in kernel memory. This structure contains the relation between function names and function addresses. To access this structure it is necessary for the system to run in kernel mode. This worked in older Windows Mobile devices, where the full-kernel mode was used. The test devices running Windows Mobile 5 are not compiled for full-kernel mode, so this approach does not work on newer devices.

A better solution can be achieved if system calls are used. Not all functions that are used by the shellcode have an equivalent system call. For example, the heap management functions `malloc` and `free` from the Windows API use system calls to implement their functionality, but there are no system calls `malloc` and `free`. We still need access to the Windows API. This can be done, by using the three system calls `LoadLibrary`, `GetModuleHandle` and `GetProcAddress`. `LoadLibrary` loads a module into memory and returns a handle for a given module name, e.g. `winsock.dll`. `GetModuleHandle` assumes, that the given module is already loaded and just returns the module handle. With the module handle and a function name `GetProcAddress` can retrieve the address of the given function.

By using these three system calls, the addresses of all Windows API functions in all modules can be reached. Kernel mode is no longer needed. This is the main advantage compared to the approaches of Hurman and San.

B.7 Second stage shellcode

Now any payload of worm code can be appended to the second stage. To simplify the development of the worm payload, the high-level language C and the GNU C compiler `gcc` for the target `arm-wince-pe` [18] are used. The result is an executable file (EXE).

The second stage is now separated into an executable starter and the executable payload (the EXE file). On execution of the second stage, the starter saves the payload to the filesystem of the mobile device. After that, the executable file is executed in a new process by using the API function `CreateProcess` [13]. Finally, the process of the exploited program is killed.

Whenever an executable is started, the operating system validates the signature of the executable. If the executable is not signed or the signature is invalid, the operating system prompts the user for a confirmation. This behavior is not desired by a smartphone worm, because now the infection depends on user interaction. However, it is possible to circumvent this by manipulating the Windows registry. The registry features a security policy named "Unsigned Prompt Policy" that defines the behavior when loading an unsigned executable.

```
; Unsigned Prompt Policy
[HKEY_LOCAL_MACHINE\Security\Policies\Policies]
    "0000101a"=dword:0
```

The default policy for Pocket PCs is the user prompt (value 0). If the value is changed to 1, the prompt will be disabled. This can be done with the registry functions from the Windows API (`RegOpenKeyEx`, `RegSetValueEx` and `RegCloseKey`). Note, that these functions may be restricted, when the process runs in normal execution mode on Smartphones. For Pocket PCs, there is no limitation, because all processes run in privileged execution mode.

C. Spreading

At this point any worm payload can be developed in a high level language and ARM assembler is no longer needed. The focus is now on the spreading function of the worm.

C.1 Algorithms for spreading

The medium for spreading is the same as for the infection: the WLAN network. The task of the worm's spread-

ing function is to find mobile devices on the network and to start the infection. For the infection over WLAN an IP address of the victim device is needed. How can IP addresses of other mobile devices be found?

The simplest algorithm would be to canonically enumerate all IP addresses of connected networks and start infection for every IP address. Of course, this is a time consuming task. In a test, the infection of a whole class C network (254 hosts) took about 30 seconds to finish. For a class B network (65534 hosts) the algorithm would take more than two hours. This is a strong limitation for the spreading of a smartphone worm.

C.2 NetBIOS port stealing

A better algorithm for spreading can be developed when looking at some features of Windows Mobile. Every Windows Mobile device that connects to local network, initially sends a series of NetBIOS datagrams on UDP port 137 to the network broadcast address. This is used by the devices to notify their participation in the *NetBIOS Name Service* [19]. The behavior can be exploited to find out the IP addresses of devices on the network. To do this, it is necessary to listen on UDP port 137 for incoming datagrams. The port is used by the system process `device.exe`, which is responsible for the implementation of the NetBIOS protocol, as well as for some device drivers. For this reason, it is not possible to kill the process. The mobile device would crash.

One solution would be to sniff on the wireless interface for the desired UDP datagrams. This would require *packet capturing*, the receiving of packets at the link layer of the OSI Reference Model. Windows Mobile 5 has no built-in capability for packet capturing. However, packet capturing can be done with the WinPcap library [20].

An easier solution exists due to an unfavorable behavior of the network stack in Windows Mobile. It is possible to *steal* a port from a listening process. The socket option `SO_REUSEADDR` must be used for binding the port. At the first try, this will not succeed. But if a second thread does the same binding attempt afterwards, the port will be bound by that thread. This is not a feature, but an undesired behavior of the network stack implementation. In other Microsoft operating systems, a socket option named `SO_EXCLUSIVEADDRUSE` is available to prevent port stealing. Windows Mobile, however, does not provide such an option.

Now it is possible to receive the NetBIOS events, and mobile devices can be infected at the moment they enter the WLAN network. This is used for the implementation of the proof-of-concept worm.

D. Assembling the smartphone worm

The worm program is a Windows PE executable for ARM processors. It was developed with the `gcc` compiler for the target system `arm-wince-pe`. It uses three threads:
1. a main thread,
2. a thread to provide the second stage shellcode (see III-B),
3. a thread for spreading (see III-C).

The main thread does the following:
1. show a confirmation dialog (see Fig. 4) at startup,
2. use portstealing to be able to receive NetBIOS datagrams,
3. launch a thread for the providing of the second stage shellcode and thread for spreading.

Fig. 4. Infected device

When the worm program is executed on a Windows Mobile 5 device, it first shows a confirmation dialog and asks the user if it should proceed (see Fig. 4). If the user confirms, portstealing is used to listen on the NetBIOS port 137. Then the program launches the thread for providing the second stage shellcode. After that, the spreading thread is launched. The spreading thread now waits for incoming NetBIOS datagrams from victim devices and sends, on event, the exploit with the first stage shellcode back to the victim device. The victim device executes the first stage shellcode and connects back to receive the second stage. Finally, the second stage shellcode is executed on the victim device, and again a confirmation dialog (Fig. 4) is shown.

E. Conclusion

The research showed, that it is possible to develop a smartphone worm (the fourth malware category of Section I) for devices running Windows Mobile 5. Infection and spreading work reliably and effectively. The exploited security vulnerability is not a real vulnerability, and does not exist in default software for Windows Mobile devices. The results can be applied to the other malware categories according to user interaction as well. The third category (common behavior) can use the infection part (Section III-B), because the staged execution and the shellcode work is useful there. The second category (trojan horse) can use the spreading part, e.g., to send an installation version of itself to the other device.

The research and development of the worm program was done in a time about thirteen weeks. About nine weeks

were spent for developing and testing the shellcode, the remaining four weeks for the spreading algorithms and for assembling the worm. This is just the constant part needed by smartphone worm. A real security vulnerability is still needed for the worm to become a real threat.

IV. Measuring Resistance

The variable part of an autonomously spreading worm is finding an actual vulnerability of the investigated system. Finding vulnerabilities in an operating system like Windows Mobile 5 can be subdivided into several stages. The focus on a worm restricts the possible attack vectors to interfaces, that have connections to other devices, e.g., Bluetooth, Infrared, IP over WLAN or the mobile network, of which IP over WLAN was chosen. The possible vulnerabilities include buffer overflows, heap overflows, and race conditions. Buffer overflows are chosen here as a vulnerability, that was seen as the most dangerous programming error of the recent years [21], and that has successfully been exploited in Windows Mobile 2003. So the focus was on searching for buffer overflows, that can be exploited through the WLAN interface.

A. Finding the attack vectors

The first step is port scanning the device (one hour of work). This resulted in five open UDP ports: 68 (DHCP), 137, 138 (both NetBIOS), 1034 ("activesync-notify"), and 2948 (WAP Push). Then, research on the structure of protocol messages is necessary (resulting in 17 hours of work). The string variables in the protocol messages are the attack vectors, that are used during the fuzzing sessions. Some protocols are sufficiently documented, others (like "activesync-notify") lack a documentation, implying that the structure can only be found by reverse engineering sniffed protocol messages, or by giving input without any structure at all.

The last preparation step is incorporating the protocol message structure into the fuzzing framework (work time included in the fuzzing time). Now the actual fuzzing process can take place.

B. Fuzzing

The fuzzing part includes varying the input to the attack vectors (i.e., the strings of protocol messages). Any unusual behavior of the investigated device, especially a crash of the input handling process, can be a sign, that a candidate input for a buffer overflow has been found. After fuzzing sessions with each of the attack vectors (21 hours of work), no candidate unusual behavior has been found, but two denial of service (DoS) possibilities and the insight, that security cookies are used in the investigated processes (five hours of work).

One DoS possibility is in the MMS handling process. For inputs in a wrong format, the device displays a warning message, that the user must confirm. If these input are sent repeatedly to the attacked device, it is rendered unusable. Another DoS possibility is in the DHCP handling process. When unsolicited DHCP ACK messages are sent to the device in a fast sequence, the reaction times of the user interface lowers, possibly rendering the device unusable.

C. Security Cookies

Another finding was, that security cookies are used in the processes. Security cookies are part of a stack protection that is enforced when compiling a program, e.g., in Visual Studio with the compiler option `/GS`. They prevent exploitation of stack-based buffer overflows.

In a function where a buffer overflow can happen, a four-byte security cookie is placed between local stack variables and the return address of the function. Its value is random and it is set in the prolog of the function. A reference value of the security cookie is stored in the data segment of the process. If a buffer overflow occurs on the stack and the return address is overwritten, the security cookie is overwritten, too. A comparison of the security cookie with its reference value in the epilog of the function notices the alteration and terminates the process without using the return address. A remote attacker does not know the value of the security cookie. This makes it difficult to exploit an ordinary stack-based buffer overflow [22].

The use of security cookies can be verified by inspecting the device's machine code. The system programs are located in the ROM part of the device and cannot be read on the file system level. Therefore they must be executed and a debugger must be attached to the process.

```
tmail.exe:00014658 LDR     R3, =0x4D714
tmail.exe:0001465C LDR     R3, [R3]
tmail.exe:00014660 STR     R3, [SP,#0x2A8]
[...]
          // function body
[...]
tmail.exe:00014678 LDR     R0, [SP,#0x2A8]
tmail.exe:0001467C BL      loc_418A4
```

Fig. 5. Security cookies in tmail.exe

The investigation has been done for `tmail.exe`, the MMS handling process. It contains instructions, that are shown in Fig. 5. The reference of the cookie's value is here at memory location `0x4d714`. The local value is at offset `0x2a8` of the stack frame. Then a checking function is called.

Because security cookies are used on Windows Mobile 5 devices, it was tried to find ways around them. Security cookies are used on other architectures, too. Litchfield [23] investigates an approach to circumvent the stack protection used in Windows 2003 Server. He finds out, that Structured Exception Handling (SEH) can be used to bypass

security cookies in a meaningful manner. SEH is applied in software development to handle errors and exceptions, therefore the approach is a very generic one. SEH uses data structures on the stack which can be overwritten by an ordinary stack-based buffer overflow. If it is possible to raise an exception or trigger a handled error, the manipulated data structure could be exploited to execute arbitrary code without using the return address.

After looking deeper into the implementations of SEH, it was found, that SEH does not use data structures on the stack, when applied in software for ARM architecture. On ARM architecture the relevant structures are stored in the data segment and in other segments of the process. Because of this, the approach of Litchfield does not work on Windows Mobile devices. Therefore, security cookies are harder to circumvent than on the x86 architecture.

D. Results

Searching for a buffer overflow adds another week (exactly 44 hours) of full-time work to the efforts. After that time, searching for buffer overflows in the protocol fields of WLAN visible interfaces of applications in Windows Mobile 5 was terminated.

The search was done with standard techniques for finding buffer overflows. This time was needed to find two possible denial of service attacks and to verify, that the system was compiled with security cookies.

V. Conclusions

We investigated the resistance of Windows Mobile 5 against the threat of an autonomously spreading smartphone worm as the innovative part of the three phases model of breaking a system. Our results are, that the constant part can be implemented as a reliable basis. The search for a buffer overflow by using fuzzing was not successful. The entire needed effort for creating the constant part (building block) and the variable part (fuzzing) without documenting the results was approximately fourteen weeks of full-time work.

As already said, the situation for an attacker changes with the publication of our results. So the measured effort will be invalid for the investigation of Windows Mobile 5 beginning with their publication time. Their value is, that they can be user to predict the resistance of other operating systems.

Our results can serve as a basis for further work in Windows Mobile malware research, where it will be — as the results are known now — in the second phase of the three phases model, the standard attack phase.

Future work encompasses checking, whether the results can be transferred to the upcoming version Windows Mobile 6. Additionally, it is worthwhile to investigate, how the security cookie mechanism can be circumvented on the ARM architecture.

References

[1] C. Peikari, S. Fogie, and Ratter/29A, "Details Emerge on the First Windows Mobile Virus," September 2004. `http://www.informit.com/articles/article.asp?p=337071`.

[2] C. Peikari, S. Fogie, Ratter/29A, and J. Read, "Reverse-Engineering the First Pocket PC Trojan," Oct. 2004. `http://www.informit.com/articles/article.asp?p=340544`.

[3] C. Peikari, "Analyzing the Crossover Virus: The First PC to Windows Handheld Cross-infector," Mar. 2006. `http://www.informit.com/articles/article.asp?p=458169`.

[4] S. E. Schechter, *Computer Security Strength & Risk: A Quantitative Approach*. PhD thesis, Harvard, 2004. `http://citeseer.ist.psu.edu/schechter04computer.html`.

[5] B. Leidner, "Voraussetzungen für die Entwicklung von Malware unter Windows Mobile 5," Diploma thesis, RWTH Aachen, Feb. 2007. (in German); `http://pi1.informatik.uni-mannheim.de/filepool/theses/diplomarbeit-2007-leidner.pdf`.

[6] S. Töyssy and M. Helenius, "About malicious software in smartphones," *Journal in Computer Virology*, vol. 2, no. 2, pp. 109–119, 2006.

[7] T. Hurman, "Exploring Windows CE Shellcode," June 2005. `http://www.pentest.co.uk/documents/exploringwce/exploring_wce_shellcode.html`.

[8] san@xfocus.org, "Hacking Windows CE," *Phrack Magazin*, vol. 6, July 2005. `http://www.phrack.org/archives/63/p63-0x06_Hacking_WindowsCE.txt`.

[9] C. Mulliner, "Exploiting PocketPC." What The Hack 2005, Netherlands, July 2005. Slides at `http://wiki.whatthehack.org/images/c/c0/Collinmulliner_wth2005_exploiting_pocketpc.pdf`.

[10] S. Fogie, "Reverse Engineering Mobile Binaries." DEF CON 11, Las Vegas, USA, Aug. 2003. Slides at `http://www.airscanner.com/pubs/fogieDC11.pdf`.

[11] C. Mulliner and G. Vigna, "Vulnerability Analysis of MMS User Agents," in *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, (Washington, DC, USA), pp. 77–88, IEEE Computer Society, 2006.

[12] E. Jonsson and T. Olovsson, "A Quantitative Model of the Security Intrusion Process Based on Attacker Behavior," *IEEE Trans. Softw. Eng.*, vol. 23, no. 4, pp. 235–245, 1997.

[13] "Microsoft Developer Network Library," Aug. 2006. `http://msdn.microsoft.com/library`.

[14] D. Seal, *ARM Architecture Reference Manual*. Addison-Wesley Professional, second edition ed., June 2000.

[15] "GNU.org homepage." `http://www.gnu.org`.

[16] "The Metasploit Project." `http://www.metasploit.org`.

[17] J. Koziol, *The Shellcoder's Handbook*. Wiley Publishing, first ed., Mar. 2004.

[18] "CeGCC's web page." `http://cegcc.berlios.de/`.

[19] "Protocol Standard for a NetBIOS Service on a TCP/UDP Transport: Concepts and Methods, RFC 1001," in *Request for Comments*, Network Working Group, Mar. 1987. `http://www.faqs.org/rfcs/rfc1001.html`.

[20] "WinPcap, The Packet Capture and Network Monitoring Library for Windows," 2006. `http://www.winpcap.org`.

[21] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: attacks and defenses for the vulnerability of the decade," *Foundations of Intrusion Tolerant Systems*, 2003.

[22] B. Bray, "Compiler security checks in depth," Feb. 2002. `http://msdn2.microsoft.com/en-us/library/aa290051(VS.71).aspx`.

[23] D. Litchfield, "Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server," Sept. 2003. `http://www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf`.