**Reihe Informatik. TR-2008-003**

# Kernel-Level Interception and Applications on Mobile Devices

Michael Becher and Ralf Hund

Department of Computer Science, University of Mannheim, Germany

May 2008

**Abstract**

The techniques of kernel-level system call interception are well known today for many different operating systems. This work starts with transferring these technique to the Windows CE type of operating systems. Afterwards, two current problems are solved. The first solution uses the technique for dynamic malware analysis with a sandbox approach, extending previous solutions in terms of effectiveness. The second solution enhances the expressiveness of security policies by implementing the concept of a reference monitor on the operating system level. Windows CE based devices are now enabled to enforce sophisticated security policies without the need to change the underlying operating system.

## 1 Introduction

The world of mobile devices is different from the common personal computer world, especially regarding its security properties [5]. A particular problem for the future is the inherent capability of generating costs for a user. Therefore, it is necessary to explore possibilities to prevent, that users of mobile phones will lose their confidence into their devices and the mobile world in general. This technical report contributes to the solution of this problem. It will describe a solution that enhances the expressiveness of security policies in a way that enables more fine-grained policies on the one hand and more global policies on the other hand. This is achieved by implementing the concept of a reference monitor on the level of the operating system. Another contribution is to the analyzing side of mobile device security, implementing a more general possibility of dynamic malware analysis with a sandbox approach. Both contributions happen to be based on the same foundation, the technique of kernel-level interception that will be described first.

The paper is structured as follows. It gives a short introduction into the world of mobile security in Section 2 and an introduction into the Windows CE type of mobile operating systems in Section 3. Section 4 transfers the technique of kernel-level system call interception (KLI) to Windows CE. The evaluation in Section 5 proves the technique as effective and complete. The sandbox approach is described in Section 6. Section 7 sketches the enhancement of security policies on mobile devices.

The presented results are work in progress, mainly for two reasons. First, we are convinced that the kernel-level sandboxing approach in Section 6 will cover all incarnations of future Windows CE malware, but we cannot prove that claim because of the lack of real malware samples. The virus Dust is the only interesting malware for Windows CE today. And second, we know about the usefulness of a policy enhancement in Section 7 implemented as an add-on, but we are missing some empirical results that will be addressed in the future.

## 2 Mobile Security

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 malicious software (malware) which prevail throughout the desktop and server world.

Malware for mobile phones is a topic of increasing importance since the middle of 2004 when the Dust virus appeared. Since then, some surveys of mobile phone malware has been published, that will be summarized in the next subsection. After that, details of three phases of malware are shown.

### 2.1 Surveys of Malware for Mobile Devices

The earliest of current surveys deals mostly with Palm OS malware [7]. Peikari [25] gives an overview also of Windows Mobile and Symbian OS malware. An extensive overview covering nearly all malware of its writing time was given by Shevchenko [30]. Example trojans have been developed by Emm [9] and Molitor [19]. Bachfeld [3] gives an overview over the filtering activities of German mobile operators and assumes, that the malware threat is currently only relevant from the perspective of the anti-virus companies. It ends with a test of Symbian OS anti-virus programs.

Töyssy and Helenius [31] list infection routes and some examples of malware. But their focus is on countermeasures and media perception.

The latest survey was given by Hypponen [14]. It shows in an illustrative comic cartoon, how many repetitions of an installation attempt (via Bluetooth) could even break down the resistance of a security-conscious user.

### 2.2 Phases of Malware

The main phases of malware are seen here as infection of the device, spreading, and malicious functionality. Making itself permanent on the device is not named here, because it is supposed, that malware can perform most malicious actions directly after infection.

#### 2.2.1 Infection

Infection possibilities, especially of Windows CE, have been investigated in detail by Mulliner [22, 23] and Leidner [16].

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. An example is the MMS buffer overflow found by Mulliner [20], that only requires the user to open the MMS. If a virus spreads by sending MMS messages to the contacts of a user, the recipients see an MMS message from a known sender, and it is probable, that they 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.

#### 2.2.2 Malicious Functionality

Once on the device, the malware can perform its malicious action. The possibilities for these actions under Windows Mobile comprise the entire system functionality [16]. For Symbian OS since version 9 (containing the Platform Security Architecture) it is assumed to be more difficult for malware to perform malicious actions on the device.

#### 2.2.3 Spreading

Spreading in the wireless LAN is simple for devices running the Windows Mobile operating system, as they announce themselves when connecting to the network. For other devices it is necessary to actively scan for new targets. The different characteristics of the spreading process are discussed by Mickens and Noble [18], who propose an extension of the usually used Kephart-White model for modeling spreading between locally related devices (WLAN or Bluetooth).

### 2.3 Windows CE Security

#### 2.3.1 Vulnerability Research

A first low-level publication about Windows CE was at Black Hat Europe 2003 by de Haas [8]. Besides some hardware information, the talk summarizes typical security flaws of that version, and presents the "Wallaby Patch Tool" custom boot loader for the HTC Wallaby, that is able to copy device memory to the SD card and to remove the device PIN.

In the same year Fogie [10] published information about reverse engineering of mobile binaries. The focus is on ARM assembler and the IDA Pro Disassmbler, but the target is a Windows CE executable file.

In Black Hat talk “Pocket PC Abuse”, Fogie [11] presents a keyboard logger, the possibilities of hidden programs, and code to trigger a hard reset.

The topic of shellcode generation is dealt with by Mulliner [21], Hurman [13], and in Phrack magazine [29]. These works use Windows CE version 4.2 as a basis.

The work of Asselineau and Hospital [2] deals with the use of the C API in order to infect a process, and with the limitations of exploitability at the kernel level. It proposes the transfer of the concept of capabilities from Symbian OS as a solution, and mentions a solution based on virtual machines. It concludes with a statement on currently available anti-virus and personal firewall programs. They would not offer sufficient protection, because:

1. “the defensive process has the same rights as the malicious process, therefore it can be terminated by the latter”, and
2. the anti-virus engines would be insufficient. And it would be impossible to deal with a sufficient complete signature database. Even the behavior-based detection is discarded by the authors, because the operating system would already need too much of the limited resources of the device.

Fogie [12] presents vulnerabilities of third-party software for Windows Mobile, subdivided into "Password Exposure Bugs", "Data Protection Programs", and "Miscellaneous Information Disclosure Bugs". Conclusions for increased security are password protection of the device, encryption of data with a proven security scheme, the advice to store as few data as possible on the device, and finally the advice to use computer security common sense also on mobile devices.

Finally, Windows Mobile version 5 has been investigated recently [16, 6]. These works further add a robust framework around the topics of low-level shellcode.

#### 2.3.2 Exploited Vulnerabilities

Only one remote code exploit is known so far for Windows Mobile. It is a buffer overflow in the handler program of MMS messages [20]. It is described in more detail in [22].

The user receives an MMS message with a specially crafted header field, that will trigger a buffer overflow, when the MMS is processed by the message handler process `tmail.exe`. The header field will be read, when the user opens the MMS for reading. The buffer overflow is not executed, when the MMS is unread in the inbox.

The exploit today is the only known example of an infection of the third type (cf. Subsection 2.2). Updated version for the message handler are available and can be downloaded for protecting the device against possible malware, that exploits the vulnerability.

## 2.4 Conclusion

It can be concluded, that the authors disagree on the impact of mobile malware. To this day, there has not been a major mobile malware outbreak. This might be due to the missing homogenous operating system base, as there is no dominant operating system today. Another reason might be the usage patterns of mobile phones, that often does not involve installation of additional software on the phone. And a third reason can be seen in the missing interest of malware authors to write malware for mobile devices. All of today's known malware either is proof-of-concept software or requires user interaction for installation on the mobile phone.

But the future will show, in which direction the field of mobile security will develop.

# 3 The Windows CE Operating System

This section introduces some aspects of the Windows CE operating system, that will be used in the following section to implement kernel-level interception. It will explain the way system calls are implemented in Windows CE, the concept of protected server libraries, and give some details about the operating system's kernel data structures.

## 3.1 Windows CE System Calls

From the user-level perspective, Windows CE provides the well known Win32-API interface with some minor exceptions. Hence, many user space programs written for Windows NT based operating systems can be easily ported to Windows CE. In contrast to user space, the kernel is different from those of other Windows operating systems. Especially the way system calls are processed is different.

System calls are typically implemented by executing special software interrupts like `int2e` in Windows NT. Some versions also use the special `sysenter` instruction that is provided by the x86 instruction set. Subsequently, a handler function is executed in the kernel, the requested system call is processed and finally the kernel gives execution back to the initiator of the system call in user space. The requested function and the parameters are given by the parameters of the interrupt call and the user space stack. Windows CE uses a slighty different approach. Although the ARM processor architecture provides an interrupt instruction `SWI`, the transition from user space to kernel space is achieved by jumping to a specially crafted invalid memory address, consisting of a architecture-dependent fixed offset, an APISet number and a method number. Consequently, the exception dispatcher will be executed and check whether the address is assigned to a certain system call. Therefore, a special area of the memory is reserved for such system call traps (called the *kernel trap area*). On ARM processors this area is located between the memory addresses `0xF0008000` and `0xF0010000`, and kernel trap adresses can be obtained by the formula

$$\text{0xF0010000} - ((\text{ApiSetID} \ll 8) \,|\, \text{MethodID}) * 4$$

## 3.2 Protected Server Libraries

Windows CE loads device drivers as non-privileged user-mode processes [24]. As a consequence, system calls are processed in separate processes, whose execution must take place in the kernel, so the parameters must be passed to kernel space.

Each device driver process which exports system call APIs has to register its own APISet first by calling the special functions `CreateAPISet` and `RegisterAPISet`.

The parameters consist of an arbitrary name with a length of 4 bytes, the number of exported functions, a method pointer table to the corresponding handler functions, and a pointer to a signature table being a bitmask of 32 bits, where the various bits indicate whether a certain argument is a pointer or not. The number of different APISets is limited to 32, where the lower 16 identifiers are reserved for the kernel. In a traditional client/server model the caller and the server run in separate threads. Windows CE differs and lets threads migrate between both processes in a system call for the sake of performance. Therefore, the current process of a thread does not necessarily have to be the thread's owner. This information can be obtained by calling `GetCurrentProcess`, `GetOwnerProcess` and `GetCallerProcess`. The latter returns the caller process of the current protected server library (PSL) API, while `GetOwnerProcess` obtains the process which really owns the thread making the function call.

As shown in figure 1, a system call in its original form goes through the following stages:

1. The program initiates an API call by invoking the designated export in a DLL (usually `CoreDLL`).
2. The DLL jumps to the corresponding kernel trap address. This step is omitted, if the program performs the jump itself.
3. The kernel exception dispatcher extracts the APISet and method number, switches to the process belonging to the APISet and jumps to the requested method by checking the method pointer table. At this stage, arguments which are tagged as pointers in the signature table will be adjusted from addresses relative to slot zero to global adresses.
4. After the method has finished, it returns to the exception handler.
5. A context switch to the caller process takes place and execution continues. To understand how it is possible to hook API calls on a kernel-mode level, one has to know which relevant data structures are maintained by the kernel, that can be altered.

## 3.3 Internal Kernel Data Structures

Each APISet contains all its information in a `CINFO` structure. This includes all the parameters that were passed to `CreateAPISet` as well as the dispatch type. Currently, Windows CE distinguishes handle-based from implicit APISets, the former ones being direct system calls while the latter ones are attached to handles such as files, sockets, and so on. An implicit API is identified by its APISet identifier and method identifier. In contrast, a handle-based API is given by its handle and the method identifier. In order to access each implicit APISet's data, the kernel maintains an array that holds all `CINFO` structures. A pointer to this array can be found in the `UserKInfo` array which is always located at the fixed offset `0xFFFFCB00` on the ARM architecure. Since even the kernel mode APISets are being registered when the system boots, all the relevant pointers are contained in writable pages. Thus, they can simply be altered and redirected to different functions. On the other hand, for each handle there is a `CINFO` structure that is allocated when the handle is created and deallocated when it is being closed.

For the pupose of completely intercepting system calls, the attached `CINFO` pointer must be changed after its creation. As every handle is being created in an implicit API call (such as `CreateFile`, `socket`) those functions will need some special handling in order to hook the method of the handle they return. But this special handling does not prevent the hooking of all system calls.

## 4 Kernel-Level System Call Interception

This section will give the implementation details of our solution. It starts with the environment that a solution has to respect and introduces afterwards two of the main aspects of our solution, the prolog/epilog methods and design decisions for positioning the sandbox within operating system and user programs.

### 4.1 Environment

Methods to hook into system calls in the kernel space have been widely studied and documented for Linux and Windows NT based operating systems. Those techniques often involve hooking a variety of tables which are used in the process of a system call such as interrupt tables and system service tables. Another approach is the so called direct kernel object manipulation. While those common methods generally also apply to the Windows CE operating system, they are heavily constricted by the wide presence of read only memory (ROM) on a mobile device. But these problems have been solved for mobile devices [17], defining the basis for our solution.

A mobile device typically contains a flash ROM memory holding the operating system along with the manufacturer's drivers and additional software. This portion of the memory is not writable and thus cannot be altered in order to hook certain events. It should be noted, that it is possible to flash the ROM. However, this is beyond the scope of our work and might be researched as future work. Besides the read-only memory, a device also contains writable memory such as SDRAM or writable flash memory (writable in the sense that it can be modified at runtime).

Windows CE does not copy executable code from ROM to RAM. Instead, it executes code directly in the ROM (called "execute in place"), making it impossible to alter that code. As the instuctions are copied to the CPU cache while being executed, there might be a possibility to change the code in the cache and thus bypass the read-only memory. But this is highly dependent on the device's hardware and may result in a unstable solution.

### 4.2 Prolog and Epilog

Our approach substitutes the method table pointer with a pointer of our own. This can be seen in the lower part of figure 1, where the original call and the hooked call take different ways beginning at the method table pointer. We generated a stub for every system call, each stub consisting of about 30 assembler instructions and some additional data. Each system call is redirected to its individual stub function with our specially crafted method pointer. The stub prepares the entry of a common prolog function that is the same for all methods.

Subsequently, the prolog decides whether the API call is to be hooked. If that is the case, then all the relevant information of the system call (such as method identifier and parameters) are passed to a special thread which resides in the target applications

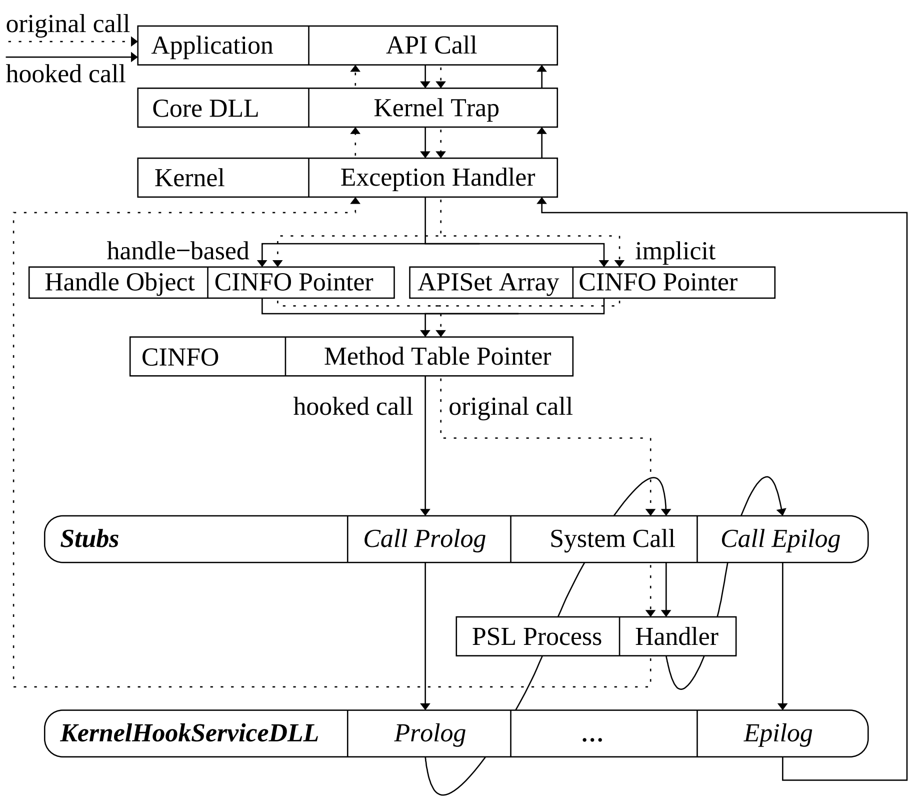

Figure 1: Kernel-Level System Call Interception

address space where they are further processed. It has to be noted, that this approach was chosen because of simplicity and the fact, that we also try to hook API calls in userpace via import address patching. A kernel-mode only sandbox would handle all of the hooking in kernel space instead. Additionally, there is no kernel debugger available which makes it rather hard and time-consuming to find errors in kernel-mode code. Therefore, some parts of the hook handling are processed directly in the application process.

The advantage of having a generic hook handler is its compact and portable nature as opposed to individual handlers, which tend to be a rather bloated solution. More code ultimately leads to more bugs and of course, the main disadvantage is simply the high number of system calls, which makes development very time-consuming. However, it has to be known which method is to be called when executing the generic prolog. This is exactly the task of the stub, whose code is also always the same but replicated for each API function with its individual data such as APISet identifier and method identifier.

To go into detail, a hooked system call goes through the following stages:

1. When system call is dispatched by the exception handler, the corresponding function address is extraced from the method pointer, which was previously patched by the sandbox. Thus our individual stub instructions are being executed rather than the real function. As explained above, the kernel switches to the address space of process of the PSL, this has to be taken into account when dealing with pointers.

2. The task of the stub is to prepare and call our generic prolog function. Special attention has to be paid to the fact that we must not alter certain registers as they might hold some of system call's arguments or might be used later on. Therefore the first step is to save all registers to the stack, followed by setting up the arguments of the prolog, which are the APISet identifier and method identifier as well as a pointer to the current stack where the register values were stored. On ARM processors, the first four arguments are passed in the registers R0-R3, whereas the rest is stored on the stack. Additionally, the registers R4-R12 have to be preserved through function calls.

3. First of all the prolog checks which process has initiated the system call. If this process is not sandboxed, then it returns immediately. Furthermore, kernel-hooking might be deactivated for single threads under certain circumstances. In this case it also returns. For instance we only hook the first level of system calls, because system calls within system calls are not of interest. We only care about the sandboxed application and not the way system calls are implemented in a PSL, so this is ignored. Moreover, a system call might already have been hooked in user space by IAT. Generally speaking, the way the generic handler is implemented has to be well thought through, otherwise the kernel might quickly hang in an endless loop when a hooked system call performs system calls itself. In case the prolog has decided to hook the call, it writes the parameters to a shared memory region and indicates, that there is a system call to be executed by triggering a special event, causing the special thread in the application address space to further process the hook. This includes extracting and logging the parameter information of the call. When finished, a second event is triggered that awakes the sleeping kernel-mode hook. Events are indicated using standard interprocess communication functions such as global mutexes or global events. Eventually the prolog returns, register values are restored from the stack and the original system call is performed.

4. In case the system call was hooked, the stub also prepares the entry of a generic epilog hook function after the call was performed. The epilog goes through the same

stages as the prolog. In some situations, it might also modify the return value of a system call. For instance, this could be necessary when the sandbox wants to hide its presence.

## 4.3 Sandbox Positioning

Our sandbox solution consists of two different DLLs, one being responsible for user-level hooking (IAT) and the other taking care of kernel-level hooking. This separation is a consequence of the layout of Windows CE system calls and the fact that there might be several processes being sandboxed at one time. In this case, there has to be a consistent interface which is exactly what the kernel hook DLL provides. The kernel-level DLL is loaded on initialization of a sandboxed process by the user-level library. Subsequently, both parties initialize themselves.

It is a vital point where the kernel-level DLL is positioned in memory. As previously explained, system calls are executed in many different processes. Therefore, our generic hooking code has to be accessible from every such process, because the kernel switches to its address space before performing the call. One solution is to inject the DLL into every PSL process. However, we have chosen to rather inject into the `nk.exe` process only and use global addresses instead. Because the kernel switches the thread into kernel-mode before performing the system call, our code will always be accessible. One just has to take into account, that the prolog and the epilog may only use local stack variables because global variables are relative to slot zero and hence not correctly mapped since a different address space is active. In order to inject into `nk.exe`, the sandbox uses the undocumented `PerformCallback4` function which executes code in another process just like in a system call. Therefore, we execute the `LoadLibrary` function in the process of `nk.exe` with a global pointer which points to the name of our kernel-hooking DLL. The well known `CreateRemoteThread` API is not available on Windows CE.

## 4.4 Preventing Kernel Mode

It might be important to prevent other programs from entering kernel mode. This is especially true for the two applications in the following sections. The sandbox wants to hide its presence from other programs, so that investigated malware does not alter its behavior because of the sandbox. And the reference monitor is only effective, if it is the only process besides system processes, that has superior access to the operating system.

Fortunately, there are only a limited number of ways for doing this. The separation between user mode and kernel mode is effective in Windows CE, so the only way to enter kernel mode is to use a system call. And all system calls are hooked by our solution, so we are always able to prevent a program from entering kernel mode, if all ways into kernel mode are intercepted. It can simply be returning an appropriate error code for an unsuccessful system call. This is some kind of suspicious behavior, but a program in user mode cannot distinguish any further between the presence of a sandbox and the possibility that the device just does not allow kernel mode.

The simplest way to gain kernel mode privileges is to call the `SetKMode` function which is provided by Windows CE. Apart from that, an application might also register its own APISet and perform a system call. Because system calls are always executed in kernel mode, the application temporarily has full privileges. Both examples have to be handled and the remaining approaches must be taken into account for a dependable solution.

## 5 Evaluation: Completeness

There are two aspects when considering completeness: interception of every system call and recognition of the system call's signature (i.e., its parameters). We describe the solution for both aspects in the following.

### 5.1 Interception

The most important part is to see every system call. This is achieved through the technique depicted in Figure 1. We change the central pointer for the data structures to point to our own data structures, and there is no other way for a program to enter kernel mode when using system calls. However, there are several special cases to consider: handle-based system calls and our own services.

Handle-based system calls load the kernel space addresses at the handle's creation time. Therefore, it is necessary to change the addresses there, so that these system calls do not circumvent our system. An example system call is `CreateFile`, where pointers to handle-based system calls (such as `ReadFile`, `WriteFile`) are maintained in an individual `CINFO` structure which is connected to the handle object. Hence, one has to patch the handle right after it was created.

Another special case is our own `KernelHookServiceDLL`. It provides some services that are necessary for the system, but that are not intercepted.

### 5.2 Signature Recognition

The signatures of the system calls can be found in the header files of the shared Windows CE source code that is distributed with the Platform Builder. These header files can be parsed. The system calls are grouped into different API sets. These are documented as comments in the header files. The sourcecode can be parsed with a tool like doxygen  and the actual signatures can be assigned to the system call in its corresponding APISet.

There are some undocumented system calls that are not present in the shared source header files. A typical examples are the GWES (graphics, window and event subsystem) API functions. All of these are intercepted, but it might happen that their signature is unknown. This case requires manual effort to locate the signature. This can be solved by using a debugger (like IDA Pro) and decompiling the library file.

## 6 Dynamic Malware Analysis

This section shows the first application of the kernel-level interception technique to help analyzing mobile malware. It will embed the result into the current state of dynamic analysis, analyze the mobile malware sample Dust, and show how an automatic analysis is useful even with a limited number of malware samples.

```
mov lr, pc
ldr pc, [r11, #-24] ; find first file
```

Figure 2: Excerpt of Dust Source Code

## 6.1 Related Work

This application of kernel-level interception contributes to the field of dynamic malware analysis. It manifests itself as an analysis module for a dynamic malware analysis system for mobile phone networks [5]. This previous work describes an analysis module that uses the technique of import address table (IAT) patching. It logs a system call when a program uses the techniques that are supposed to be used. This is comparable to the DLL injection techniqe that CWSandbox uses [32]. Both approaches are not able to see direct jumps into the kernel that are necessary to analyze some malware samples. The malware Dust is nearly half of the currently known Windows CE malware and it uses this technique of direct jumps.

TTAnalyze [4] uses a different approach to analyze a sample. They use a processor emulator with a defined interface for intercepting system calls and are able to see every system call, regardless what the sample did to call it. Our solution is different in terms of applicability on real devices. It can be used on any real device where the investigated sample is not able to detect the presence of an emulator.

## 6.2 Analysis of Mobile Malware "Dust"

At the current time there is only few malware known for the Windows CE operating system. One trojan is called Brador[1] that opens a backdoor on the device and sends the device's IP address to the malware author [28]. The virus Crossover[2] is an example of cross-platform malware that are able to run on different platforms. Crossover uses the .NET application framework and can be executed on Windows CE and Win32 platforms because of the executable's binary compatibility. It checks for the platform during startup and executes different code depending on the platform [26].

A third malware example is Dust[3]. Its source code was published [27], so we could use this malware to validate our analyzer. Dust does not use the import address table to access the system calls, because it cannot be sure that the host program uses the same system calls as itself. Instead, it calculates the addresses with the formula of Subsection 3.1 and directly jumps to these addresses. An example can be seen in Figure 2, where the program counter is set to a value of the stack that was previously set to the address of `FindFirstFileW`.

We used a device with Windows CE version 5. The original Dust sample did nothing. It assumed to run in kernel mode, as it was common in previous versions of Windows CE. So we added a call to `SetKMode` to make it work. An excerpt of the analysis can be seen in Figure 3. The system call #7 is the log of the source code excerpt in Figure 2. The succeeding system calls exactly reflect the source code.

[1]also known as Backdoor.WinCE.Brador.a

[2]also known as MSIL/Cxover.A

[3]also known as Virus.WinCE.Duts.a with different character order in its name

| ID | System Call | Arguments |
|---|---|---|
| 7 | `FindFirstFileW (direct)` | `lpFileName=*.exe` |
| 8 | `SC_CreateFileForMapping (direct)` | `lpFileName=Dust.exe ...` |
| 9 | `FindNextFileW (direct)` | `hFindFile=420704 ...` |
| 10 | `SC_CreateFileForMapping (direct)` | `lpFileName=Sample.exe ...` |

Figure 3: Excerpt of Dust Analysis

## 7 Security Policy Enhancement

This section applies the kernel-level interception technique to the field of security policies. It shows, how the technique can be used to enhance the expressiveness of current security policies and additionally how to implement recent related work in a more general way.

### 7.1 Problem Description and Related Work

Current security policy implementations on real mobile devices only enable coarse-grained policies. Especially, there are only few possibilities to restrict data network usage. The current expressiveness is to disallow an access to a certain API completely, to allow it unrestricted (for every run of the program or for the current session), and to ask every time that an API is called. Especially the last option can be very tedious for the user when the program uses many consecutive calls to a certain API. It would be useful to enhance the expressiveness of security policies to allow at once a certain number of events, e.g., a number `n` of messages or amount `x kBytes` of data.

Recent work [15] solves the problem for the Java 2 Micro Edition (J2ME) that is present on almost all of today's mobile phones. They implement an extension of an open-source J2ME virtual machine. The extension must be present at compile time of the virtual machine and must afterwards be incorporated into the device.

Our solution is different in two ways. First, it is applied at the deeper level of the device's operating system, not on a virtual machine level. Therefore, it can be used for native programs, enabling a broader application. Second, it can be applied to existing devices without changing the operating system. That means, we implement the concept of a reference monitor [1] as a flexible solution that can be applied to existing devices as an add-on in contrast to a major change of the operating system. This solution will be described in the next subsection.

Currently out of scope is the question of how the security policy finds its way from the specification into our system. At the moment, we simply assume that it is present.

### 7.2 Problem Solution

We are able to use the previous parts of this paper as building blocks for our solution. And without loss of generality we use the example of data network access in the following.

Subsection 5.1 proved that we are able to intercept all system calls. Of course, this includes any system calls to access the data network. And every access to the data network must use a system call. Therefore, our solution will see every access to the data network and is able to apply access restrictions by a security policy. Embedded into our solution of Figure 1, the reference monitor would be implemented in the prolog function of `KernelHookServiceDLL`.

Our solution has superior rights compared to the monitored programs, because it works in kernel mode. Therefore, we must prevent that other programs switch into kernel mode. Normal programs do not need to switch into kernel mode, therefore it is no restriction of functionality when disallowing kernel mode. As we proved in Subsection 4.4, it is possible to completely disallow access to kernel mode with our solution.

When the policy is active, it is necessary to match the calling process with its corresponding security policy that the reference monitor holds in a list for every policy of the system. As described in Subsection 3.2, this is done with `GetCallerProcess` in a reliable way.

We now have a reference monitor in place, that can be used to effectively enforce security policies concerning data network usage. But the system is able to perform more functions. A fundamental problem in the Windows CE world is the lack of any reasonable type of security policy. Other systems like Symbian OS or J2ME implement concepts like private persistent data spaces for every program, API grouping and several trust levels, with selective access to API groups depeding on the trust level. Our system is able to add these concepts to Windows CE. With the same techniques as restricing data network usage, it is possible to restrict access to certain parts of the persistent storage to certain processes and therefore programs.

Additionally, it is possible to restrict the access to cost-generating functions like messaging or phone functions. So, it is useful to globally restrict access to these functions. For example, the Internet browser on mobile devices is likely to be an increasingly important attack vector with the increasing prevalence of data network usage. But the Internet browser does not need to access messaging (or phone) functions. And likely does no other program, with the notable exception of signed - and therefore trusted - programs. So it is useful to restrict the possibility to send messages to the messaging process and the possibility to initiate phone calls to the phone handler process. This way, many of the expected problems of upcoming mobile malware can be solved, and our solution is able to accomplish this.

## 8 Conclusion and Future Work

We introduced the technique of kernel-level system call interception for the Windows CE type of operating systems and explained, how this technique is able to solve two current problems, one in the area of dynamic malware analysis with a sandbox approach, the other one in the area of enhaning the expressiveness of security policies.

For future work, we will address the completion of our policy enhancement system in order to be a versatile solution. Some performance evaluation will quantitatively prove our intuitive impression, that our solution works efficiently. Additional future work will be the application of our dynamic analysis solution to analyze real-world malware samples for Windows CE, as soon as they appear.

## Acknowledgements

This research was kindly supported by T-Mobile Deutschland.

## References

[1] James Anderson. Computer Security Technology Planning Study, 1972.

[2] Roderick Asselineau and Jean-Marc Hospital. Bewertung der Sicherheit von mobilen Endgeräten unter Windows CE. *Misc*, 1:17–23, 2006.

[3] Daniel Bachfeld. Wurmflug. *c't*, 13:156–163, 2006.

[4] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic Analysis of Malicious Code - TTAnalyze. *Journal of Computer Virology*, 2006.

[5] Michael Becher and Felix C. Freiling. Towards Dynamic Malware Analysis to Increase Mobile Device Security. In *Proc. of SICHERHEIT*, 2008.

[6] Michael Becher, Felix C. Freiling, and Boris Leidner. On the Effort to Create Smartphone Worms in Windows Mobile. In *Information Assurance and Security Workshop, 2007*, pages 199–206, 20-22 June 2007.

[7] Claudio Beltrametti and Reto Calonder. Malware on Mobile Devices. Master's thesis, FH Chur, October 2004.

[8] Job de Haas. The phone in the PDA - Pocket PC Phone edition security. In *Black Hat Briefings Europe*, May 2003.

[9] David Emm. The Changing Face of Malware. In *Proceedings of the IWWST*, 2005.

[10] Seth Fogie. Embedded Reverse Engineering: Cracking Mobile Binaries. In *DEF CON*, August 2003.

[11] Seth Fogie. Pocket PC Abuse. In *Black Hat*, July 2004. Black Hat.

[12] Seth Fogie. Airscanner vulnerability summary: Windows Mobile fails the test. *(IN)SECURE Magazine*, 8:41–51, September 2006.

[13] Tim Hurman. Exploring Windows CE Shellcode, June 2005.

[14] Mikko Hypponen. Malware goes Mobile. *Scientific American*, pages 70–77, 2006.

[15] Iulia Ion, Boris Dragovic, and Bruno Crispo. Extending the Java Virtual Machine to Enforce Fine-Grained Security Policies in Mobile Devices. *ACSAC*, pages 233–242, 2007.

[16] Boris Leidner. Voraussetzungen für die Entwicklung von Malware unter Windows Mobile 5. Master's thesis, RWTH Aachen, February 2007. (in German).

[17] Dmitri Leman. Spy: A Windows CE API Interceptor, October 2003.

[18] James W. Mickens and Brian D. Noble. Modeling epidemic spreading in mobile environments. In *WiSe '05: Proceedings of the 4th ACM workshop on Wireless security*, pages 77–86, New York, NY, USA, 2005. ACM Press.

[19] Sören Molitor. Mobiles unter Beschuss - Viren und Verwandte auf Handys, June 2004.

[20] Colin Mulliner. Advanced Attacks Against PocketPC Phones. In *DEF CON*, August 2006.

[21] Collin Mulliner. Exploiting PocketPC. In *What The Hack*, Jul 2005.

[22] Collin Mulliner. Security Analysis of Smart Phones. Master thesis, UCSB, 2006.

[23] Collin Mulliner and Giovanni 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*, pages 77–88, Washington, DC, USA, 2006. IEEE Computer Society.

[24] John Murray. *Inside Microsoft Windows CE*. Microsoft Press, Redmond, WA, USA, 1998.

[25] Cyrus Peikari. PDA attacks, part 2: airborne viruses - evolution of the latest threats. *(IN)SECURE Magazine*, 4:32–41, October 2005.

[26] Cyrus Peikari. Analyzing the Crossover Virus: The First PC to Windows Handheld Cross-Infector, March 2006.

[27] Cyrus Peikari, Seth Fogie, and Ratter/29A. Details Emerge on the First Windows Mobile Virus, September 2004.

[28] Cyrus Peikari, Seth Fogie, Ratter/29A, and Jonathan Read. Reverse-Engineering the First Pocket PC Trojan, October 2004.

[29] san. Hacking Windows CE. *Phrack Magazin*, 6(63), July 2005.

[30] Alisa Shevchenko. An overview of mobile device security, September 2005.

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

[32] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy*, 5(2):32–39, 2007.