Hijacking the Linux Kernel

Boris Procházka[1], Tomáš Vojnar[2], and Martin Drahanský[3]

1 **Faculty of Information Technology, Brno University of Technology**
Božetěchova 2, 61266 Brno, Czech Republic
`iprochaz@fit.vutbr.cz`

2 **Faculty of Information Technology, Brno University of Technology**
Božetěchova 2, 61266 Brno, Czech Republic
`vojnar@fit.vutbr.cz`

3 **Faculty of Information Technology, Brno University of Technology**
Božetěchova 2, 61266 Brno, Czech Republic
`drahan@fit.vutbr.cz`

Abstract

In this paper, a new method of hijacking the Linux kernel is presented. It is based on analysing the Linux system call handler, where a proper set of instructions is subsequently replaced by a jump to a different function. The ability to change the execution flow in the middle of an existing function represents a unique approach in Linux kernel hacking. The attack is applicable to all kernels from the 2.6 series on the Intel architecture. Due to this, rootkits based on this kind of technique represent a high risk for Linux administrators.

Digital Object Identifier 10.4230/OASIcs.MEMICS.2010.85

1 Introduction

We propose a new attack on the Linux kernel based on changing the control flow in the system call handler. The attack is applicable to all members of the 2.6 family on the Intel architecture. The main idea, changing the control flow in the middle of the system call handler, has not been to the best of our knowledge considered before and hence rootkits (tools setting up an environment for an attacker and hiding his/her activities) are not detectable using current detection tools. To compensate for the newly proposed attack, we also provide a new detection tool capable of detecting the new attack.

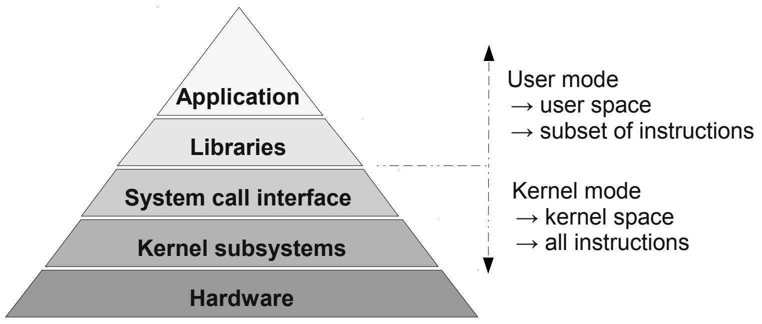


Figure 1 Operating system hierarchy

Basically, attacks can be divided into two main groups according to the operating system hierarchy (Fig. 1)[1, 2]:

© Boris Procházka, Tomáš Vojnar, and Martin Drahanský;
licensed under Creative Commons License NC-ND

1. **Attacks on user mode.** Attacks against user's and system's applications and libraries. In this case, the attack is usually performed by a simple substitution of binaries, where the attacker swaps the originals with the corrupted. Fortunately, these kind of attacks are quite easy to detect thanks to checksums. There is also a possibility to use private, static-compiled binaries.
2. **Attacks on kernel mode.** Most of nowadays attacks are oriented towards kernel space, especially against kernel interfaces like system call interface or virtual file system. The main reason why these kind of attacks are so popular is because the attacker is able to gain control of the whole system with no mercy. We can easily imagine that if we change the behaviour of kernel interfaces, we will change the behaviour of the whole system (because user space programs rely on them). The attacker usually wants to hide his activity in the system so he modifies the interfaces to publish only a subset of real results. In the case of kernel space attacks, there is no reliable method how to reveal the attacker in the system. We can only hope that he was not skilled enough to masquerade all side effects of his activities.

In the rest of the paper, we will focus solely on the system call interface. We will discuss existing types of attacks on this mechanism and later on, an original attack will be revealed. Our idea will be to inject jump code in the middle of the system call handler.

2 System Call Interface

The system call interface forms an interface for switching between user and kernel mode. An application raises a query through the system call interface and the kernel tries to satisfy it. The system call interface is probably the most important interface in the system as it creates an abstract layer between users and the kernel.

In Linux (on the IA-32[1]), system calls are identified by numbers and their invocation is realized by a software interrupt. Parameters are passed through CPU's registers in a strict order: `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi` and `ebp`. The `eax` register is used to hold the system call number. Let us see how system calls work on the case of the `setuid()`[2] call (Fig. 2).

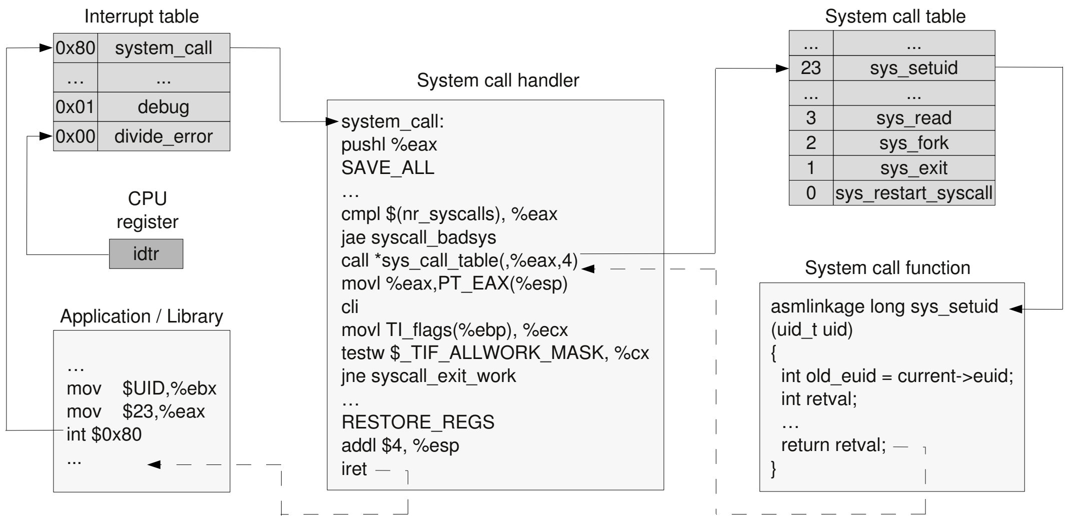
First of all, an application (or a wrapped routine in a library) has to fill CPU's registers with expected values. Then, an exception is risen by the `int $0x80` instruction which will cause the switch-over to the kernel mode and a system call handler activation. The address of the system call handler is saved in the interrupt table and is determined by an index into this table (`0x80` in this case).

The system call handler (implemented in `system_call()`) first saves the number of the system call and then the contents of all CPU's registers (`SAVE_ALL` macro). All passed arguments are saved and the CPU is released for further computing. Then, some basic tests are performed (we will cover this in detail in Subsection 4.1). If everything is alright, the system call number saved in the `eax` register is used as an index into the system call table to invoke a system call function (`sys_setuid()` in the described example).

System call functions perform the executive code. Their purpose is to change the system state or return system values. All system call functions are implemented as `asmlinkage`, which means that their arguments are saved exclusively on the stack.

When the system call function finishes, the return value is saved on the stack in the place of the `$eax` register. The system call handler continues with disabling interrupts

[1] Intel architecture, 32bit. Known also as i386 or x86.
[2] `Setuid()` sets the effective user ID of the current process.



```
...
mov   $UID,%ebx
mov   $23,%eax
int $0x80
...
```

```
system_call:
pushl %eax
SAVE_ALL
...
cmpl $(nr_syscalls), %eax
jae syscall_badsys
call *sys_call_table(,%eax,4)
movl %eax,PT_EAX(%esp)
cli
movl TI_flags(%ebp), %ecx
testw $_TIF_ALLWORK_MASK, %cx
jne syscall_exit_work
...
RESTORE_REGS
addl $4, %esp
iret
```

```
asmlinkage long sys_setuid
(uid_t uid)
{
  int old_euid = current->euid;
  int retval;
  ...
  return retval;
}
```

Figure 2 The Linux system call interface

and performing additional tests. On success, all CPU's registers are restored from the stack (`RESTORE_REGS` macro) and the function is finished with the `iret` instruction. This instruction causes a controlled switch-over back to the user mode and a continuation of the application.

The system calls are used very frequently and the implementation by software interrupt is not very efficient. Due to this, processors Intel Pentium II and older contain an additional instruction called a "fast system call" (the `sysenter` instruction). Although this instruction calls another handler (`sysenter_entry()`), the results (and even the body of the function) is almost the same as in the case of a system call handler (`system_call()`). So we do not have to distinguish between these two methods in the rest of the paper—the impact will be the same.

3 State of the Art

Attacks against the system call interface are relatively old and widespread. In this section, we will briefly describe existing types of such attacks:

1. **Attacks on the system call table.** The oldest and the most widely used way of intrusion. Its aim is to change an original record in the system call table with another version of the system call function [3]. This function is then used instead of the original. In most cases it acts like a wrapper for the original function filtering its results. The system call table is usually checked by administrators nowadays, so attackers had to develop more sophisticated ways of attacks.
2. **Attacks on the system call function.** If we do not want to change records in the system call table, we can move one step forward and change the prologue of the system call function [4]. The basic idea is to rewrite the entry point of the original function with a jump to a different function. The usage is the same as in the previous with one exceptions—if we want to use the original function, we have to repair its prologue or an infinite loop threatens.

3. **Attacks on the system call handler.** Another method how to redirect the execution flow without touching the system call table is to leave off using it. To do so, we have to copy the original system call table to a new location and change the pointer in the system call handler [5]. When it is done, the old system call table is not used anymore and we can modify our private one, just like in the first case.
4. **Attacks on the interrupt table.** If we take a closer look at Fig. 2, we can reveal that a second table is used in the subsystem—the interrupt table. The attacker can change records even in this table and forge the handler routine [6]. This attack is not trivial as the attacker needs to build up his own handler function and the interrupt subsystem is closely associated with the computer architecture.
5. **Attacks on the idtr register.** The interrupt table is located thanks to the `idtr` register. The value of the register can be modified by the `sidt` instruction. The attacker can do the same trick as in the attack on the system call handler—make a copy of the table and change the value in the `idtr` register to pointer on it [7].

4 The New Approach

In this section, we will focus on changing an execution flow in the middle of an existing function. The idea is motivated by attacks on the system call function, where the prologue is rewritten by a jump code. We will try to generalize this technique to be applicable even in the middle of functions.

If we want to hijack an execution flow in the middle of an existing function, we have to rewrite its code. This is quite easy. The biggest problem is to ensure the original behaviour of the corrupted function. If we write down all the problems we have, we will get these three issues:

1. **Seven bytes of space.** For hijacking the execution flow, we have to rewrite the existing code with a jump or call instruction. The easiest way is to fill one of the CPU's registers with the destination address and then perform an absolute jump[3]. If we write it down in assembly, we will get something like this:

   ```
   movl $0,%eax --> \xb8\x00\x00\x00\x00
   jmp  *%eax   --> \xff\xe0
   ```

   The code is compiled as shown on the right side. The result is seven bytes long machine code. This means that we will need to rewrite at least two instructions as the Intel architecture uses a variable instruction length.
2. **Keep valid code.** We have to keep the code valid after overwriting it. If we produce an invalid instruction, the CPU rises an exception and immediately terminates the process. Due to this, we have to respect the beginnings and ends of instructions and do not overwrite code containing the labels.
3. **Keep the original semantics.** We will change the structure of the considered function by injecting some code. As we want to keep the original behaviour, we have to compensate the rewritten code to sustain the original semantics. This is the most difficult condition and requires a data analysis because when the hijacking is completed, the function must continue in its execution with no restrictions.

[3] It is not possible to do an absolute jump by `jmp $address`.

We will now demonstrate how to solve the above problems for the particular case of the system call handler.

4.1 Where to Hijack Control Flow in the System Call Handler

In this subsection, we will study the code of the system call handler and try to determine best places for hijacking. The handler is a low-level subsystem thus it is completely written in assembler[4]:

```
system_call:
  pushl %eax                          //Storing of system call number
  SAVE_ALL                           //Storing of all CPU's registers
  movl $0xffffe000, %ebx              //Calculation of the pointer to
  andl %esp, %ebx                                   //current process
```

The function starts its activity by storing the system call number and all CPU's registers in the stack. Afterwards, a pointer to the current process is calculated and saved in the `ebx` register.

In a above code fragment, we now try to find a candidate place for hijacking. We cannot rewrite the beginning instructions which are saving data from the user space (we would probably loose some data). However, when all data from the user space is saved, the CPU is released and there is an ideal opportunity for hijacking the control flow. If we measure the number of bytes of two instructions calculating the pointer to the current process (`movl`, `andl`), we will get seven bytes. The calculation of the pointer is also standalone and independent and it can be easily reproduced.

```
  testw $_TIF_WORK_SYSCALL_ENTRY,TI_flags(%ebp)  //Process traced?
  jnz syscall_trace_entry             //If so, jump to trace function
  cmpl $(nr_syscalls), %eax          //eax >= number of system calls?
  jae syscall_badsys                                   //If so, abort
```

The system call handler continues with two tests. The first one checks whether the running process is being traced. If the trace flag is set, the process is stopped and made available to the debugger. The second test checks the validty of system call number in `eax` register. As the number in `eax` register represents an index into the table, it cannot be greater than total number of system calls in the system.

We can leave studying of the fragment above very briefly. We would break code containing relative jumps (`jnz`, `jae`) which would be very difficult to compensate.

```
  call *sys_call_table(0, %eax, 4)   //Calling sys_call_table[eax]
  movl %eax,PT_EAX(%esp)                   //Storing of return value
```

The core of the system call handler. The `eax` register is used as an index into the system call table to call the system function. The return value is saved into the stack in the position of the `eax` register for the user space.

Despite the core of the system call handler is suitable for hijacking, we will leave it off. The reason is that the pointer to the system call table is also modified by attacks on the system call handler and the system scanners generally test this value.

[4] This code can be found in Linux kernel source: `arch/x86/kernel/entry_32.S`.

```
cli                                           //Clear Interrupts
movl TI_flags(%ebp), %ecx           //Copy process flags in ecx
testw $_TIF_ALLWORK_MASK, %cx           //Is needed extra work?
jne syscall_exit_work                    //If so, do extra work
```

When the system function returns, all interrupts are masked and the process is tested against additional work requirements (unserved signal, process is traced).

The code fragment above offers another opportunity for hijacking. The length of the `movl` and `testw` instructions is eight bytes, which is enough for jumping out. These two instructions are also standalone so we can reproduce them.

```
RESTORE_REGS                        //CPU's registers restoration
addl $4, %esp          //Clearing up system call number from stack
iret                                      //Return from interrupt
```

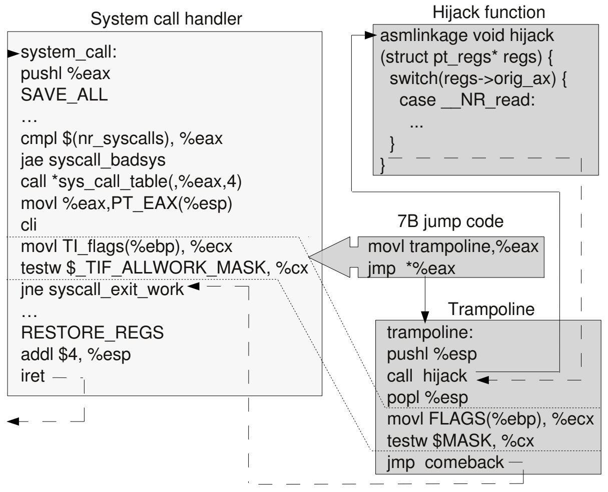
The end of the system call handler prepares the system to switch back to the user mode. All CPU's registers are restored from the stack (filled with results of the system call), the system call number is cleared and the `iret` instruction is triggered.

The hijacking of this terminating fragment is possible too. The main problem is when the process is traced and so does not use this code fragment.

4.2 Changing the Control Flow in the System Call Handler

So far, we have located two suitable places for hijacking the control flow in the system call handler. Both of them are occurring in the whole 2.6 kernel and thus offering a very good portability and usability.

Our goal is to modify the return values from the system call functions which are available after their invocations. Due to this, we will focus on the second place suitable for hijacking the control that we have identified in the system call function.



```
system_call:
pushl %eax
SAVE_ALL
...
cmpl $(nr_syscalls), %eax
jae syscall_badsys
call *sys_call_table(,%eax,4)
movl %eax,PT_EAX(%esp)
cli
movl TI_flags(%ebp), %ecx
testw $_TIF_ALLWORK_MASK, %cx
jne syscall_exit_work
...
RESTORE_REGS
addl $4, %esp
iret
```

```
asmlinkage void hijack
(struct pt_regs* regs) {
  switch(regs->orig_ax) {
    case __NR_read:
      ...
  }
}
```

```
movl trampoline,%eax
jmp  *%eax
```

```
trampoline:
pushl %esp
call  hijack
popl %esp
movl FLAGS(%ebp), %ecx
testw $MASK, %cx
jmp  comeback
```

Figure 3 Hijack implementation

We will rewrite selected part of the code by an unconditional jump into the trampoline function (`trampoline()`). The trampoline has several tasks. First, it saves the top of the stack to ensure a convenient access to the system call results. Subsequently, it calls the hijack function (`hijack()`), which modifies these results on the basis of the system call number (deleting records about hidden directories in the `ls` program, for example). When the hijack function returns the execution back to the trampoline, the system has to be set back to the original state. To do so, the stack is cleared, instructions used for the hijacked jump are compensated and the execution is returned just after the kidnapped place. The attack is over.

5 Experiments

To verify all the presented facts and ideas, two rootkits based on this new technique were implemented. The first one, called *MoleKit*, is able to infiltrate the system through writing into the `/dev/mem` file[5]. Molekit provides only basic services like process and directory hiding. It is because attacking the system through `/dev/mem` file is quite complicated due to many heuristics needed (finding code patterns in the memory), but it represents a way how to infiltrate a system even without a loadable kernel module support (see [8] for more information).

The second rootkit is called *Powerkit* and infiltrates the system using a kernel module. The main advantage of kernel modules is the access to the kernel API. This makes it possible to implement advanced features such as keylogging or escalation of privileges.

As these two rookits use the new method of hijacking, no current anti-rootkit or validity scanner can detect them. Because of that, we implemented *Sentinel* scanner [9]. Sentinel is a tool which periodically checks integrity of the interrupt subsystem, the system call interface (including the new method presented in this paper) and the virtual file system. The detection is based on testing key values of these subsystems (tables, pointers to tables, system call handler code, function prologues, ...) against their reference values obtained after system instalation.

6 Conclusion

In this paper, a new method of hijacking the Linux kernel was presented. The attack was successfully verified on the whole 2.6 kernel series and two rootkits based on this new technique were implemented. Because these two rookits would represent a serious security risk for Linux administrators, a tool for their detection was published.

Acknowledgements This work was supported in part by the the Czech Ministry of Education (projects MSM 0021630528) and the internal BUT FIT grant FIT-10-1.

References

1 R. Love. *Linux Kernel Development*. Novell Press, Indiana 46240, USA, 2006, ISBN 0-672-32720-1

2 P.D. Bovet, M. Cesati. *Understanding the Linux Kernel.* O'Reilly, USA, 2005, ISBN 0-596-00565-2

3 P. Sobolewski. *Hakin9 Nr 2/2005*. Software-Wydawnictwo Sp. z o.o., Warszawa, Poland, 2005, ISBN 1214-7710

4 S. Cesare. *SYSCALL REDIRECTION WITHOUT MODIFYING THE SYSCALL TABLE.* http://www.ouah.org/stealth-syscall.txt

[5] It is a character device providing access to the main memory

5 Devik, Sd. *Linux on-the-fly kernel patching without LKM.* http://www.phrack.org/issues.html?issue=58&id=7#article

6 Kad. *Handling Interrupt Descriptor Table for fun and profit.* http://www.phrack.org/issues.html?issue=59&id=4#article

7 B. Prochazka. *Methods of Linux Kernel Hacking.* FIT BUT, Brno, 2008, bachelor's thesis

8 A. Lineberry. *Malicious Code Injection via /dev/mem.* http://www.blackhat.com/presentations/bh-europe-09/Lineberry/BlackHat-Europe-2009-Lineberry-code-injection-via-dev-mem.pdf

9 B. Prochazka. *Program Sentinel.* http://www.stud.fit.vutbr.cz/ xproch63/conference/memics2010/sentinel.zip