SAPIENZA
UNIVERSITÀ DI ROMA

FACULTY OF INFORMATION ENGINEERING, INFORMATICS AND STATISTICS

MASTER OF SCIENCE IN ENGINEERING IN COMPUTER SCIENCE

Securing the IDT and the System Call Table from malicious LKMs

Candidate

Pietro Stroia

Supervisor

Prof. Francesco Quaglia

Assistant Supervisor

Ph.D. Alessandro Pellegrini

Academic Year 2012/2013

Timeo Dànaos et dona ferentis.

Laocoön

Abstract

Provided that loadable kernel modules (LKMs) run at ring 0, is it really possible to prevent kernel-level attacks once the evil code has started its execution? Without a hypervisor, or some kind of virtualization technology (e.g. VT-x), results in general have not been proved very successful, as Microsoft's Patchguard technology has shown in the past years[1].

My thesis, which provides a patch for the Linux kernel version 3.2.51, aims at providing a security mechanism against synchronous malicious LKMs, or, in other words, LKMs that execute their attack in their initialization phase, without deferring any work. Although the patch code is for an IA-32 compatible Linux system only, there are no major obstacles in porting the code to other operating systems that support LKMs and run at least on the 80386 architecture. The rationale behind this work is that I personally had the "feeling" I could puzzle simple evil LKMs and let them believe they had successfully executed their attack, without using any kind of virtualization technology. The work, named the "YASI patch", is to be intended as a small "proof-of-concept" running on top of an already secured system.

Acknowledgments

First and foremost, I would like to express my most sincere thanks to my supervisors, Prof. Francesco Quaglia and Ph.D. Alessandro Pellegrini, without whom this thesis would still be buried in my mind, probably dismissed in a few months from now as a vague and unfeasible idea. I greatly appreciated their astounding willingness to help out with any problem I had, from LaTeX to kernel code, and their always encouraging trust in me. I recall with a smile and a deep sense of gratitude the many emails, the long off-hour consulting support and even a Skype call when Prof. Quaglia was too down with the flu to come to the department to hear me ramble confusedly about the x86 architecture.

I would like to thank my parents, Anna and Franco, for their love, infinite support and enthusiasm, not only in these two years but in my entire life. They have been, if not physically, always beside me in the good and, above all, bad times that have put me to a hard test. I would like also to take the chance to give a warm embrace to my cousin Donatella, my grandmothers Laura and Mea and my uncle Gianfranco.

It's also quite difficult to find the right words to properly thank the blue-eyed blonde girl I could always rely on and that made any situation I deemed difficult lighter and easier. Well, thank you Francesca! :)

While I would like to thank each one of my friends, colleagues and people I have met in these years individually, it will be very hard even to name a few here without sparking a fierce debate! I would like to thank you all together, thinking of the good times we have spent at our department, in Rome, Eindhoven or Isernia.

Last, but not least, I would like to dedicate this thesis to myself, hoping that it will lead me to many greater achievements.

That's all folks!

Contents

List of Figures

Foreword

In mid-2010 the whole world was left in awe by the discovery of the first real cyber weapon, the Stuxnet worm. The worm itself, and the Flame, Duqu and Gauss variants appeared in the following years, have probably marked the beginning of an ongoing militarization of the Internet and of a state-sponsored cyber warfare and cyber spying. Yet, I would have known the news only almost one year later, in the sleepy days preceding my Bachelor's degree graduation ceremony. That was the proverbial feeble spark that slowly lit a small fire of curiosity first, and a real thirst for knowledge in the field then.

As a personal thought, I think it is due to say that the last two years have been really intense, with a lot of new information, emotions and experiences flowing in at full pace. While sometimes I wished I could have postponed some deadlines to concentrate more on the stuff I enjoyed studying and working on, that has not always been possible. However, I do sincerely hope that you will enjoy reading my work as I enjoyed working on it, wishing that I will be able to convey my enthusiasm to you about this always intriguing subject.

Chapter 1

A few operating systems security aspects and issues

Security, in general, is a very elusive goal. As it is usually stated in the field, with an enlightening analogy, security is "a chain as strong as its weakest link". Unlike other fields in Computer Science which can easily count on the rigorousness of abstract mathematics to define what is formally safe and what is not, such boundaries are slightly more blurred when talking about operating system security and development. While, generally speaking, we are still very far from solving all the possible security issues at application-level development, at least it has the much sought privilege of being confined in a very precise environment by a lower layer, which is "the operating system". Instead kernel programming is completely free of such a restrictive environment, giving developers the highest level of freedom they can strive for. Unfortunately, as also seen in the news, such freedom can be easily misused or abused, leading to serious and advanced threats like kernel-level malware. However, this does not imply that secure operating systems do not exist in general, but rather that they are part of a niche market, given their industrial, military or purpose-driven nature. While not being an evaluation of the overall security of the final product, but rather a certification of the amount and level of testing they received, such niche operating systems usually receive a very high grade on the Evaluation Assurance Level (EAL) scale[21]. Taking with a grain of salt such grades, commercial and universally known operating systems, for instance Microsoft Windows or Linux, do not go higher on that scale than a EAL4+ certification[21].

Leaving aside the EAL certification, the very first thing that comes to mind is that embedded operating systems do not usually allow, or probably even support, the execution of third-party code on demand, having some fixed tasks that routinely ask for execution. However, such a feature is probably mandatory on any self-respectable, general-purpose commercial product. Allowing the execution of such binaries, potentially coming from untrusted locations, is obviously a clear security issue. The problem however is not only confined to this external code, but rather the issue is about which modifications to the system an application is allowed to perform. Even without considering intentional malicious behavior, is it possible for a simple but gone astray program to corrupt kernel memory, other programs' memory or execute privileged instruction? Where is exactly the difference between kernel-level code and user-space (less privileged) code? Putting aside the first questions for a while, it is clear that the processor needs to be able to discriminate between different levels of code privilege and, unless there is some kind of hardware-assisted mechanism, this is clearly not possible. It is now necessary to start talking about what is commonly referred to as "rings", and this is where the aforementioned boundaries of responsibility start blurring soon.

"Rings" are essentially a synonym for privilege levels. The first Intel processor to have such capability, and thus being able to run in Protected Mode, was the 80286 CPU[5], supporting 4 privilege levels:

- Ring 0 is the highest privilege level, reserved for what will be become known as kernel code.
- Ring 1 is usually unused on traditional operating systems.
- Ring 2 is usually unused as well.
- Ring 3 is the lowest privilege level, reserved for what will be become known as user space.

Would it be right to quickly blame any operating system as being "insecure" if tailored for the 80186 architecture? Wouldn't be fairer to be a little concerned about the choice of running everything at ring 0, even on the 80286 and later architectures, as MS-DOS did? While the examples might seem very outdated or a bit extreme for a modern reader, they serve the purpose to stress the fact that any operating system's evaluation cannot be really parted from the underlying architecture. For an interested reader, a more modern example of an OS-level security issue, caused by some hardware design, might be the controversial and recently introduced Intel `RDRAND` assembly instruction that has sparked quite a heating debate[20][7].

Going back to our pending questions, the 80286 architecture, along with the Protected Mode, introduced also a mechanism of memory protection called segmentation, which, very basically, provided a way to divide the main memory in so called "segments"[5]. Process isolation was guaranteed by assigning one or more contiguous memory segments to each program, effectively separating their address space as a result. In addition to that, each segment expressed its actual level of privilege in a field called DPL. While being almost forgotten as a memory protection mechanism, in favor of paging on modern systems, it made heavy use of the `cs`, `ds`, `es`, `fs`, `gs` and `ss` segment registers and of the `gdtr` and `ldtr` registers, which stored respectively the Global Descriptor Table (GDT) and the Local Descriptor Table (LDT) addresses. Manipulation of such structures and registers obviously required executing some privileged instructions, only available at code running at ring zero[15].

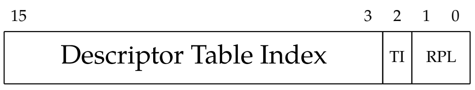


Figure 1.1: 16-bit Segment Selector

Recalling that the 80286 was a 16-bit processor, segmentation worked by setting each segment register with a 16-bit segment selector. The TI field in the segment selector was used to choose between the two descriptor tables, the GDT or, alternatively, the LDT. The RPL field instead indicated the requester's level of privilege, used by the hardware when performing a privilege level checking. As seen in Figure 1.1, the segment selector has an index that points to an entry in the GDT or the LDT, entry known as a segment descriptor. Figure 1.2 shows a segment descriptor specifying the base address of the segment, its length, its type and the descriptor privilege level (DPL) along with some other fields.

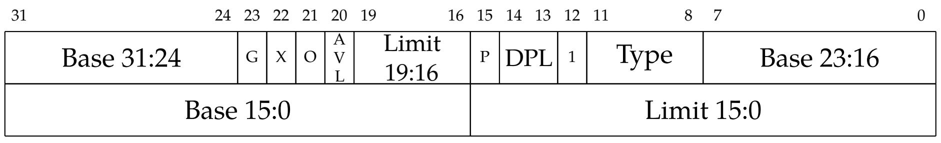


Figure 1.2: Data and Code Segment Descriptors

In 1985, three years after the debut of the 80286, the first 32-bit processor was introduced to the market, the 80386 processor, which was also the first one to support the paging mechanism[5]. Paging divides the main memory in so called “pages” of variable length, but usually of 4KB each. The translation mechanism of paging associates to each 4KB page a 4KB “frame” of physical memory. While different levels of pagination are possible, paging uses only two components, a Page Directory and multiple Page Tables, whose entries are respectively called PDE (Page Directory Entry) and PTE (Page Table Entry). While hiding some complexity away and simplifying a little bit, we can say that the `cr3` control register contains the starting address of the Page Directory. Entries in the Page Directory (PDE) point to a Page Table, while entries in a Page Table (PTE) point to the actual physical “frames”. A virtual address contains the necessary indexes to address first a PDE and then a PTE, so that a virtual to physical mapping is always possible. While any operating system textbook will provide all the insights required to coherently discuss about paging, like its actual mechanisms, the possible levels of paging and why translation look-aside buffers are needed, it is worth to stress out again that paging provides a more fine-grained control over memory than segmentation, with the additional benefit of a reduced fragmentation[13][24].

31 12	11 10 9	8	7	6	5	4	3	2	1	0
Page Table Base Address	Avail	G	PS	0	A	PCD	PWT	US	RW	P

Figure 1.3: Page-Directory Entry (4KB Page Table)

31 12	11 10 9	8	7	6	5	4	3	2	1	0
Page Base Address	Avail	G	PAT	D	A	PCD	PWT	US	RW	P

Figure 1.4: Page-Table Entry (4KB Page)

That ended our very brief introduction to the memory protection problem, answering all the previous pending questions. Nevertheless, that wasn't at all an extensive list of the security problems that operating systems face. Apart from classic, but not trivial problems like authentication and access or resource control, if we would like to scratch just the surface of the myriad of other potential issues, we should dig in really more advanced topics like heap and stack canaries, shellcode generation, address space layout randomization, "return to libc" attacks and so on and so forth.

As already stated before, the main part of my thesis is a security-focused patch for the Linux kernel. It is interesting to know better the operating system this thesis is targeted at. Linux is an open-source, modular and monolithic kernel that runs on a wide range of architectures but started in 1991 as 80386-only and was then greatly inspired by the Minix operating system[22]. During the years, portability was probably the key factor for the success and popularity of the Linux kernel, but that did not come without a price in terms of trade-offs. For instance, one of them is that Linux does not use segmentation, because some architectures, on which Linux is able to run, do not support it at all. Even when running on the x86 architecture, Linux employs a "flat-memory model", which basically "turns off" segmentation, leaving the virtual memory implementation to the paging mechanism only. When defining Linux as a monolithic kernel, it means that the whole operating system is running at the highest level of privilege, ring 0, confining the user-space applications to the lowest level and not using the eventual levels of privilege in between, even when provided by the underlying architecture. This is an additional example of these trade-offs, as Linux runs, for instance, on both the x86 architecture that provides 4 levels of privilege and on the PowerPC architecture that has only two[14].

Modularity allows to extend kernel functionalities, without having to recompile the kernel, using loadable kernel modules (LKMs) which are basically relocatable ELF objects containing kernel-level code and data. However, the ability to extend the kernel not only at maintenance time, but in reality "on-demand", poses not entirely negligible security risks. It is not hard to realize that, once loaded, the external code will be part of the operating system to all intents and purposes and, as stated before, will run at the same, highest level of privilege as the rest of the kernel. Interesting questions, unfortunately still unresolved, arise when thinking if it is possible to protect the kernel from itself, if we need further support in hardware than what we currently have, if the world should switch to a "microkernel" layout in the future or if we should start working only with virtualized kernels. While my thesis humbly provides, on very stringent hypothesis, a randomization-based security mechanism against malicious LKMs, it might be interesting to point out that this problem is not entirely unknown in literature. Basically, only two different approaches are available:

- A white-list approach, based on signatures, which forces users and administrators to load only trusted, digitally signed LKMs. Support for this feature has been finally mainlined with Linux kernel version 3.7, requiring only to set the appropriate configuration flags, `CONFIG_MODULE_SIG` and optionally `CONFIG_MODULE_SIG_FORCE`, which does not allow mounting a module lacking a digital signature[4]. However, expecting that a large majority of Linux users are using a distro-packaged kernel, this approach, depending on how it is implemented, may hurt the ability and freedom to start developing or running out-of-tree modules[12].

- The other approach, introduced since 2005, is basically the one used by Microsoft in their Patchguard Kernel Patch Protection technology for x86-64 architectures. It provides a mechanism that routinely checks if some kernel structures have been modified by some device drivers, which are to be intended here as the Windows-equivalents of Linux loadable modules. Similarly to the YASI patch, its goals are to detect modifications to the Interrupt Descriptor Table (IDT) and to the System Call Table (SCT). However, Microsoft Patchguard technology is an order of magnitude more complex and complete, guarding also against modifications of the Global Descriptor Table (GDT) and other vital kernel structures[23]. The major difference between this approach and the one implemented by the YASI "proof-of-concept", is that the main objective of the latter is also to let an attacker believe he has successfully patched the kernel and not just to crash abruptly.

Chapter 2

Attacking with a synchronous LKM

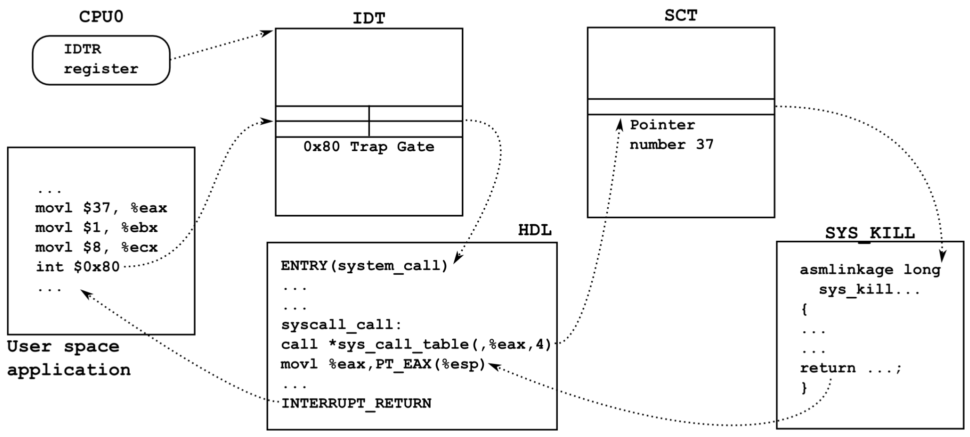
A necessary preliminary step, before discussing in greater detail how malicious kernel-level rootkits accomplish their work, is to recall how system calls get executed with the traditional software-interrupt scheme on the x86 architecture, or, in other words, what happens when software calls the `int 0x80` assembly instruction. From a simplified point of view of a user-space program or library, a system call is executed by storing the system call number in the `eax` general purpose register and storing up to six parameters in `ebx`, `ecx`, `edx`, `ebp`, `esi`, and `edi` registers before executing the `int 0x80` instruction. System call numbers, for the x86 architecture and at least up until Linux kernel version 3.2.x, are available for consultation at `<arch/x86/kernel/syscall_table_32.S>`[9].

Verbatim from the `INT` assembly instruction page of the Instruction Set Reference of the Intel IA-32 Architecture Software Developer's Manual, Volume 2[5]:

> The INT *n* instruction generates a call to the interrupt or exception handler specified with the destination operand. . . . The destination operand specifies an interrupt vector number from 0 to 255, encoded as an 8-bit unsigned intermediate value.
>
> . . .
>
> The interrupt vector number specifies an interrupt descriptor in the interrupt descriptor table (IDT); that is, it provides index into the IDT. The selected interrupt descriptor in turn contains a pointer to an interrupt or exception handler procedure. In protected mode, the IDT contains an array of 8-byte descriptors, each of which is an interrupt gate, trap gate, or task gate.

At this point, the current thread enters kernel mode and, after passing some needed checks in such interrupt handler procedure, executes the requested system call, depending on the value of the `eax` register. Indeed, the `eax` register is used in a `call` assembly instruction as an index to the System Call Table (SCT), which is a table that contains pointers to the actual system calls. Upon leaving kernel mode, the return value of the requested system call will be available to the user-space program or library in the `eax` register. If needed, Figure 2.1 provides a visual explanation of what mentioned so far.

Kernel-level rootkits offer almost infinite possibilities: malicious software can leverage kernel functionalities to hide files and network connections, gain root privileges and much more. However, how is it possible for a malicious LKM to take control of what gets executed at ring 0? One of the simplest ways is to "steal" a system call. It basically amounts to changing one or more addresses in the SCT, redirecting the execution flow to an external function



```
...
movl $37, %eax
movl $1, %ebx
movl $8, %ecx
int $0x80
...
```

```
ENTRY(system_call)
...
...
syscall_call:
call *sys_call_table(,%eax,4)
movl %eax,PT_EAX(%esp)
...
INTERRUPT_RETURN
```

```
asmlinkage long
  sys_kill...
{
...
...
return ...;
}
```

Figure 2.1: Execution flow of a system call invocation (x86 architecture)

specified by the evil module itself. Until around Linux kernel version 2.4.18[6] the SCT symbol `sys_call_table` used to be exported, and thus visible to any kernel module. As still seen in many historical online resources, stealing a system call was as easy as doing:

```
extern void ** sys_call_table;
...
orig_sys_kill = sys_call_table[__NR_kill];
sys_call_table[__NR_kill] = evil_sys_kill; /* steal! */
```

In addition to that, since around kernel version 2.6.12[10], the SCT is in a read-only protected memory. And trying to steal the `sys_kill` system call with the same method, just to make an example, would result in a saddening kernel OOPS:

```
[ 72.035098] BUG: unable to handle kernel paging request at c12cbf24
[ 72.036219] IP: [<f86f50ea>] init_module+0xb6/0xd0 [evil]
[ 72.037422] *pdpt = 0000000001485001 *pde = 00000000360f5063 *pte =
     80000000012cb161
[ 72.038119] Oops: 0003 [#1] SMP
```

The hexadecimal error value in the last line, `0x0003`, according to the page fault error code bits in `<arch/x86/mm/fault.c>`, signals that there was a write access protection fault in kernel mode. Things used to be easier indeed.

However, by running at ring 0, in principle, an evil module could easily overcome this protection, by flagging again such memory pages as writable or by, interestingly, setting to zero the Write Protect (WP) bit in the control register `CR0`.

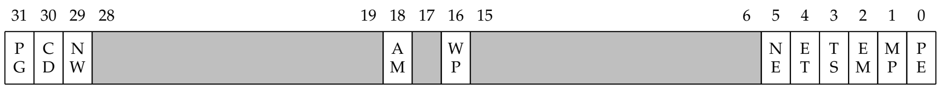


Figure 2.2: Control Register CR0

Again, verbatim from section 2.5 of the Intel manual, Volume 3[5]:

> **WP Write Protect (bit 16 of CR0)** — When set, inhibits supervisor-level procedures from writing into read-only pages; when clear, allows supervisor-level procedures to write into read-only pages (regardless of the U/S bit setting; see Section 4.1.3 and Section 4.6). This flag facilitates implementation of the copy-on-write method of creating a new process (forking) used by operating systems such as UNIX.

In the end, the real question was (and still is): How can LKMs get the SCT address? At first sight, only few rough solutions seems available:

- Recompile the kernel to export again the `sys_call_table` symbol and to remove the read-only protection. Unfortunately, this makes sense only if you want to hack your own machine.
- Search the SCT address inside the `/boot/System.map` file, looking for

the `sys_call_table` symbol. It is possible to search for such address directly from kernel mode or using an user space "loader" program, which, after completing the search, will request a LKM insertion and pass the SCT address to the evil module as a loading parameter. The major drawback of this approach is that it relies on a file that could have been easily removed by system administrators.

- Search for the SCT address in memory with brute force, knowing that is for sure between two exported symbols. The address is retrieved by checking if each guessed address, after applying a specific displacement, returns the address of a known and externally visible system call. However, since such symbols are hard-coded in the malicious executable, this technique might be in general ineffective when dealing with a different kernel version than the one the executable was targeted for. For instance, in Linux kernel versions 2.6.x, the SCT used to lie between the exported `boot_cpu_data` and `loops_per_jiffy` symbols[18], but that it is not the case anymore.

However an alternative, elegant and and always applicable solution does exist. The earliest reference I have found to such technique is an article[11] in the Phrack magazine, issue 58 of December 2001, which is quite astonishing when considering the fact that it was published almost exactly 12 years ago as of writing! The proposed solution, which is the building block of my thesis, uses the `sidt` assembly instruction, introduced with the 80286 processor[5] in the x86 instruction set. The `sidt` instruction is used in order to read the `IDTR` register, which stores the linear address of the Interrupt Descriptor Table (IDT) and its length in bytes. The `IDTR` register is 48 bits long and its related C struct is defined in the `<arch/x86/include/asm/desc_defs.h>`

file:

```
struct desc_ptr {
        unsigned short size;
        unsigned long address;
} __attribute__((packed)) ;
```

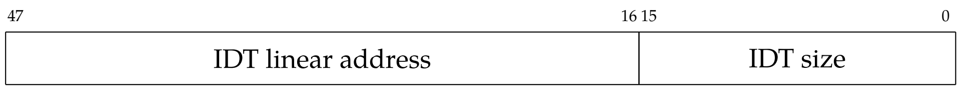


Figure 2.3: IDTR register

Furthermore, the `IDTR` register can be easily read or written in C with the help of some inline extended assembly using, respectively, the already known `sidt` instruction and the `lidt` instruction:

```
struct desc_ptr idtr;
...
__asm__ __volatile__ ("sidt %0" : "=m" (idtr)); /* read */
...
__asm__ __volatile__ ("lidt %0" : : "m" (idtr)); /* write */
```

where the "__volatile__" keyword is used to tell GCC not to do any kind of optimization on this statement[3]. Intel Developer's manual[5] states that while the `sidt` instruction can still be used by user-space programs without generating an exception, both the instructions are really meant to be executed at ring 0.

Finally, it is possible to get the SCT address following these simple steps:

- Read the `IDTR` register with the `sidt` instruction, in order to find the linear address of the IDT.

- Go to the `0x80`-th entry of the IDT (recall the original `int 0x80` instruction), finding a Trap Gate Descriptor there.

- As shown in Figure 2.4, the offset is split in two parts. After building it again, you will find the entry point of the System Call Handler (HDL), defined in `<arch/x86/kernel/entry_32.S>`.

```
ENTRY(system_call)
        RINGO_INT_FRAME   # can’t unwind into user space anyway
        pushl_cfi %eax    # save orig_eax
        SAVE_ALL
        GET_THREAD_INFO(%ebp)
                        # system call tracing in operation /
                            emulation
        testl $_TIF_WORK_SYSCALL_ENTRY,TI_flags(%ebp)
        jnz syscall_trace_entry
        cmpl $(nr_syscalls), %eax
        jae syscall_badsys
syscall_call:
        call *sys_call_table(,%eax,4)
        movl %eax,PT_EAX(%esp)  # store the return value
syscall_exit:
...
```

- From this address, search for the `FF 14 85 xx xx xx xx` byte pattern. The unknown 4 bytes, `xx xx xx xx`, form the address of the SCT.

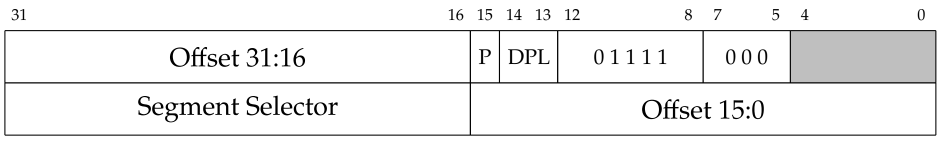


Figure 2.4: Trap Gate Descriptor

Fortunately, a good explanation of the inner details that justifies the search of such pattern can be found in the Intel manuals. In the Opcode Map in Appendix A, Volume 2[5], it is shown that the `FF` one-byte opcode is a `Group 5` instruction, which means that bits 5, 4 and 3 of the ModR/M byte are used as an opcode extension.

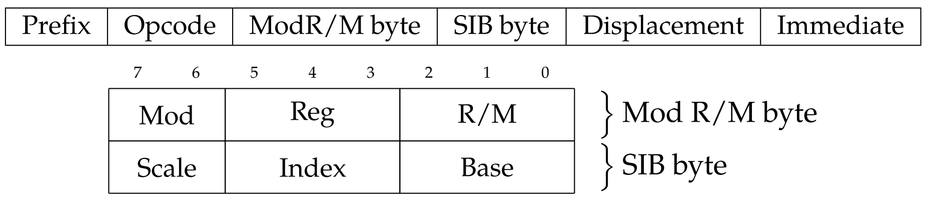


Figure 2.5: IA-32 Instruction Format, ModR/M and SIB bytes

The ModR/M byte, `0x14`, has a binary representation of `0b00010100`, so, according to the Opcode Extensions Table, `FF Grp 5` is a `near CALL Ev`. However, the 32-bit addressing format says that with our Mod bits, `0b00`, and R/M bits, `0b100`, a SIB byte follows the ModR/M byte. Its value, `0x85`, corresponds to the very familiar `[eax * 4]`, leading to an effective address of `[scaled index] + disp32`. In other words, the address of the `call` instruction is `[eax * 4] + sys_call_table`. Recalling what happens when software traps with the `int 0x80` instruction and by looking at the Linux kernel code above, this is our expected `call` to the SCT.

A very simple Linux kernel module that implements the aforementioned technique is:

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/slab.h>
#define SEARCH_LEN 128

static unsigned long * find_sct(const unsigned long addr,
        const unsigned int size)
{
       unsigned int i;
       unsigned char *ptr;

       ptr = (unsigned char *) addr;

       for (i = 0; i < size; i++) {
               ptr++;
               if (*ptr == 0xff && *(ptr + 1) == 0x14
                       && *(ptr + 2) == 0x85) {
                       return (unsigned long *)(ptr + 3);
               }
       }
       /* didn't find */
       return NULL;
}

int init_module(void)
{
       struct desc_ptr idtr;
       unsigned long handler, sct;
       unsigned long *pattern_addr;

       __asm__ __volatile__ ("sidt %0" : "=m" (idtr));
       printk(KERN_INFO "[%s] IDT is @ 0x%lx\n",
               __this_module.name, idtr.address);
       handler = gate_offset(*((gate_desc *) idtr.address + 0x80));
       printk(KERN_INFO "[%s] HDL is @ 0x%lx\n",
               __this_module.name, handler);
       pattern_addr = find_sct(handler, SEARCH_LEN);
       if (pattern_addr == NULL)
```

```
                printk(KERN_INFO "[%s] SCT not found\n", __this_module.name);
        else {
                sct = *pattern_addr;
                printk(KERN_INFO "[%s] SCT is @ 0x%lx\n",
                        __this_module.name, sct);
        }
        return 0;
}

void cleanup_module(void) { }

MODULE_LICENSE("GPL");
```

Which provides the following output in the `dmesg` log on a stock Debian 7, confirmed also by a `grep` on the `System.map` file:

```
[ 176.828627] [info] IDT is @ 0xc13de000
[ 176.828691] [info] HDL is @ 0xc12c3c68
[ 176.828770] [info] SCT is @ 0xc12cbe90
...
c12cbe90 R sys_call_table
```

And yet, what has been achieved so far is really much more than simply getting the address of the SCT. A malicious LKM now knows the address of the IDT, its size in bytes, the entry point of the HDL, the address of the operand of the aforementioned `call` instruction and its value, which is the address of the SCT. The system is now completely in the hands of the attacker, since it is not possible to prevent anymore a "classic" system call theft, a code injection in the system calls, a more subtle hot-patching of the HDL to have it point to a malicious copy of the SCT or even a deeper attack to the IDT or `IDTR` register.

The next chapter will dive into the coding part of my thesis, the kernel patch, discussing the architectural choices.

Chapter 3

Summary of the YASI protection mechanism

It will be probably much easier to start explaining my thesis from the very end, or, in other words, from what I actually achieved. I will try to be very concise, explaining here just the fundamental blocks of the work, leaving all the inner details, for an interested reader, to the patch code in Appendix A. The next code listing shows the output of the dmesg command after loading the same, unmodified kernel module shown at the end of the previous chapter, but this time in a YASI-patched kernel:

```
[ 62.932432] [info] IDT is @ 0xf5ccc800
[ 62.932508] [info] HDL is @ 0xf8016000
[ 62.932590] [info] SCT is @ 0xf5e4e000
[ 62.932650] [YASI] No malicious behavior detected in module info.
```

Ignoring for now the last line, we can safely say that the module has been tricked into seeing fake addresses, fending off a potential attack. It is important to remark now that any kind of malicious behavior will fail since the evil code will try to hot-patch the *wrong* memory addresses. The only notable exception, due to a design choice, is a code-injection in the system calls, but further considerations on this case will be given in the next chapter. Those addresses are just pointers to *fake* copies of the IDT, HDL and SCT, since it is obviously impossible to change the real addresses of these structures on demand. Just to make an example, even without considering concurrency issues when attempting that, the SCT address is the same between reboots since it is determined at kernel compile time.

Considering that the evil code is directly using assembly code without relying on any kernel API, different parts of the Linux kernel core had to be modified in order to trick the attacking module. The whole idea of my work can be broken down in the following simple steps:

- When a module is about to execute its `__init` function, create a copy of the SCT, HDL and of the IDT. Modify the latter, in particular its `0x80`-th entry, in order to point to the fake HDL, which is then further modified to obviously point to the fake SCT in its `call` instruction.
- Save this information in an appropriate data structure, along with a pointer to the `task_struct` of the task that has requested the module load, or, in other words, the task that has entered kernel mode in order to execute the `init_module` system call. Usually, such system call is invoked only by the `modprobe` or `insmod` programs.
- Change the `IDTR` register on the CPU the module is loading on, pointing to the fake IDT. From now on, when the module will enter its `__init` function, it will see the whole chain of fake addresses (IDT - HDL -

SCT).

- However, by having a way to keep track of which tasks have asked for a module load, the scheduler can potentially take some action depending on the tasks involved in a context switch. In theory, the task executing the `init_module` function can be preempted, since it is running in "process context". If that is the case, or, in other words, such task is being preempted, the scheduler can restore the original, safe `IDTR` register with a simple `lidt` assembly instruction. Obviously, when such task will be about to resume its execution in another context switch, the scheduler will change again the `IDTR` register with the fake one, in order to present a coherent environment.

- When the module has finished its initialization phase, check for potential malicious behavior and restore the original `IDTR` register. The comparison is done, on a byte-by-byte basis, between the copy of the SCT and the original one and between the MD5 hashes of the IDT and of the HDL. The need for these two hashes, while computationally expensive, is unavoidable. The fake IDT must point to the fake HDL, which has a different memory address from the true HDL and which in turn points to a different SCT address. Obviously, a byte-by-byte comparison between the two IDTs or the two HDLs will fail. Furthermore, if a malicious module modifies some bytes of the fake IDT or of the fake HDL, granted that the "original state" of these fake structures is not being stored, a decision about the behavior of the evil code is completely impossible. The solution is to store, during the creation of the fake IDT and HDL, the MD5 hashes of their current state. Now, checking for a potential malicious behavior basically amounts to re-computing again and then comparing the hashes on the same memory segments with the saved ones.

If needed, at the end of this chapter, Figure 3.1 presents a simple visual explanation of the execution flow of a system call invocation under a YASI-patched kernel.

The aforementioned main data structure, `yasi_task_info`, is declared in the `<include/linux/yasi.h>` file, along with some extern variables and functions declarations:

```
struct yasi_chain_info {
        unsigned char idt_hash[16];
        unsigned char hdl_hash[16];
        unsigned long sct;
        unsigned long hdl;
        struct desc_ptr idtr;
};

/* Our main data-structure */
struct yasi_task_info {
        struct task_struct *task;
        struct yasi_chain_info info;
        struct list_head list;
};
```

In the `yasi_chain_info` data structure, the two unsigned char arrays are used to store the MD5 digests of the fake IDT and of the fake HDL. The `unsigned long sct` and `unsigned long hdl` variables are used, respectively, to store the address of the fake SCT and of the fake HDL. The address of the fake IDT is stored, as seen in the previous chapter, in a `struct desc_-`

`ptr idtr` variable.

In the `yasi_task_info` data structure instead, the `struct task_struct *task` pointer is used to save a reference to the task that has entered kernel mode to execute the `init_module` system call, as said before. By looking at the declarations above, it should be clear that the fake addresses are not unique or global, but they are created *for each task* that has asked for a module load. This means that there must be somewhere a list of `yasi_-task_info` structures, protected by some sort of locking. In the next lines of the `<include/linux/yasi.h>` file we see such things, a `struct list_-head yasi_list` and a read-write lock `rwlock_t yasi_lock`. A read-write lock is used here, instead of a simpler spinlock, because code accessing the `yasi_list` list is perfectly separated in readers and writers.

```
extern struct desc_ptr yasi_idtr;
extern struct list_head yasi_list;
extern rwlock_t yasi_lock;
```

The `yasi_idtr` variable, which contains the real IDTR of the system, is initialized at early boot in the `<init/main.c>` file:

```
__asm__ __volatile__ ("sidt %0" : "=m" (yasi_idtr));
printk(KERN_INFO "YASI KPP is up.\n");
```

As stated before, a `yasi_task_info` structure, `ptr_ypt` in the code listing below, has to be initialized when a module loads. In the `init_module` system call, in file `<kernel/module.c>`, such action takes place:

```
ptr_ypt = kmalloc(sizeof *ptr_ypt, GFP_KERNEL);
if (((void *) ptr_ypt) == NULL) {
        ret = -1;
        goto yasi_handle_error; /* jump to common exit */
3008  }
```

The function `yasi_create_chain`, at line 3024, is responsible for creating a fake IDT, a fake HDL and a fake SCT:

```
3012  ptr_ypt->task = current;
3013  INIT_LIST_HEAD(&(ptr_ypt->list));

3024  ret = yasi_create_chain(&(ptr_ypt->info));
3025  if (unlikely(ret != 0)) {

3030  if (ret == -ENOMEM)
3031          printk(KERN_WARNING "[YASI] Unable to allocate enough"
3032                  "memory! Denying execution of module %s\n", mod->name)
                          ;
3033          else
3034          printk(KERN_WARNING "[YASI] Unable to create fake "
3035                  "chain! Denying execution of module %s\n", mod->name);
3036          goto yasi_handle_error;
3037  }
```

After that, the `ptr_ypt` structure is added atomically to the `yasi_list` list:

```
3043  write_lock_irqsave(&yasi_lock, yasi_irq_flags);
3044  list_add_tail(&(ptr_ypt->list), &yasi_list);
3045  write_unlock_irqrestore(&yasi_lock, yasi_irq_flags);
```

Now that the `yasi_list` is not empty, the scheduler will change, if needed, the `IDTR` register, depending on the tasks involved in the context switch. In fact, during a context switch, the Linux kernel scheduler keeps track of the `next` and `prev` task pointers:

- If the next task is in the yasi_list, get the associated yasi_task_info structure and change the IDTR register accordingly.
- Otherwise, if the prev task is in the yasi_list, restore the good IDTR register saved in early boot.

Now that the scheduler knows always what to do in case of a context switch, change the IDTR register with the fake one and launch the __init function of the module:

```
3055 __asm__ __volatile__ ("lidt %0" : : "m" (ptr_ypt->info.idtr));

3059 if (mod->init != NULL)
3060         ret = do_one_initcall(mod->init);
```

After the module has finished its initialization, a check on its behavior is performed with the yasi_system_safe function and the yasi_task_info structure is removed from the list atomically:

```
3069 yasi_system_safe(&(ptr_ypt->info), mod->name);

3072 write_lock_irqsave(&yasi_lock, yasi_irq_flags);

3078 list_del(&(ptr_ypt->list));
3079 deferred_vfree = (void *) ptr_ypt->info.hdl;
3080 yasi_delete_chain(&(ptr_ypt->info));
3081 kfree(ptr_ypt);
```

The yasi_delete_chain function, at line 3080, frees every kmalloc allocated memory, without freeing any memory allocated with the vmalloc function. The main reason behind the need for a deferred vfree is that executing it while still holding a spinlock might create HARDIRQ unsafe scenarios, as stated by the "lockdep" validator[16]. In addition to that, the real IDTR register must be restored before exiting the atomic section, because the fake one is now invalid and will lead to kernel faults when serving interrupts.

```
__asm__ __volatile__ ("lidt %0" : : "m" (yasi_idtr));
write_unlock_irqrestore(&yasi_lock, yasi_irq_flags);

vfree(deferred_vfree);
```

What is still to be discussed, is what the scheduler does when context switches are taking place. Basically, the scheduler, in the `context_switch` function, file `<kernel/sched.c>`, checks if `yasi_list` is empty or not:

```
read_lock(&yasi_lock);
if (unlikely(!list_empty(&yasi_list)))
        yasi_switch(prev, next);
read_unlock(&yasi_lock);
```

If that is the case, the `yasi_switch` function is invoked, which traverses the `yasi_list` list and checks if any of the tasks involved in the context switch are present in it, modifying the `IDTR` register as needed. The “unlikely” macros, in the functions below and above, were added not to incur in an almost always avoidable performance penalty. As a design choice, it was supposed that the wrong branch prediction penalties, due to module insertions at boot time, would greatly offset the penalties incurred without the “unlikely’ macros in a prolonged use scenario.

```
static void __always_inline yasi_switch(const struct task_struct *
    const prev,
         const struct task_struct *const next)
{
        struct yasi_task_info *ptr_ypt;
```

```
list_for_each_entry(ptr_ypt, &yasi_list, list) {
        if (unlikely(ptr_ypt->task == next)) {
                /* Set the fake IDTR */
                __asm__ __volatile__ ("lidt %0" : : "m"
                        (ptr_ypt->info.idtr));

        if (unlikely(ptr_ypt->task == prev))
                /* Set real IDTR */
                __asm__ __volatile__ ("lidt %0" : : "m" (yasi_idtr));
```

That was the last fundamental building block of my thesis work. In order not to make this chapter any heavier, the very hackish and engaging details of the `yasi_create_chain` and `yasi_system_safe` functions are available in Appendix A, as stated at the beginning of this chapter. As a last thing, I would like to point out the effects of loading a malicious module that makes a copy of the SCT and attempts to change the operand of the `call` instruction in the HDL to such copied and potentially evil SCT:

```
[19004.776065] [evil] IDT is @ 0xf5810800
[19004.776148] [evil] HDL is @ 0xf8014000
[19004.776238] [evil] SCT is @ 0xf5e6a800
[19004.776345] [evil] New SCT is @ 0xf5c37200
[19004.776818] [YASI] module evil has tried to modify at least the
    system call handler.
[19004.776830] The system is SAFE but it is suggested to reboot to
    revert to a clean state.
[19004.776836] Do not rmmod the module, a kernel OOPS may appear.
```

The evil kernel module source code that attempts such attack, a little too long to be listed here, is available in Appendix B. However, in addition to this HDL patching attack scenario, the YASI patch is also able to discover and prevent attacks on the IDTR register, on the IDT and on the SCT.

The next chapter will discuss the weaknesses of the protection mechanisms and the possible future improvements.

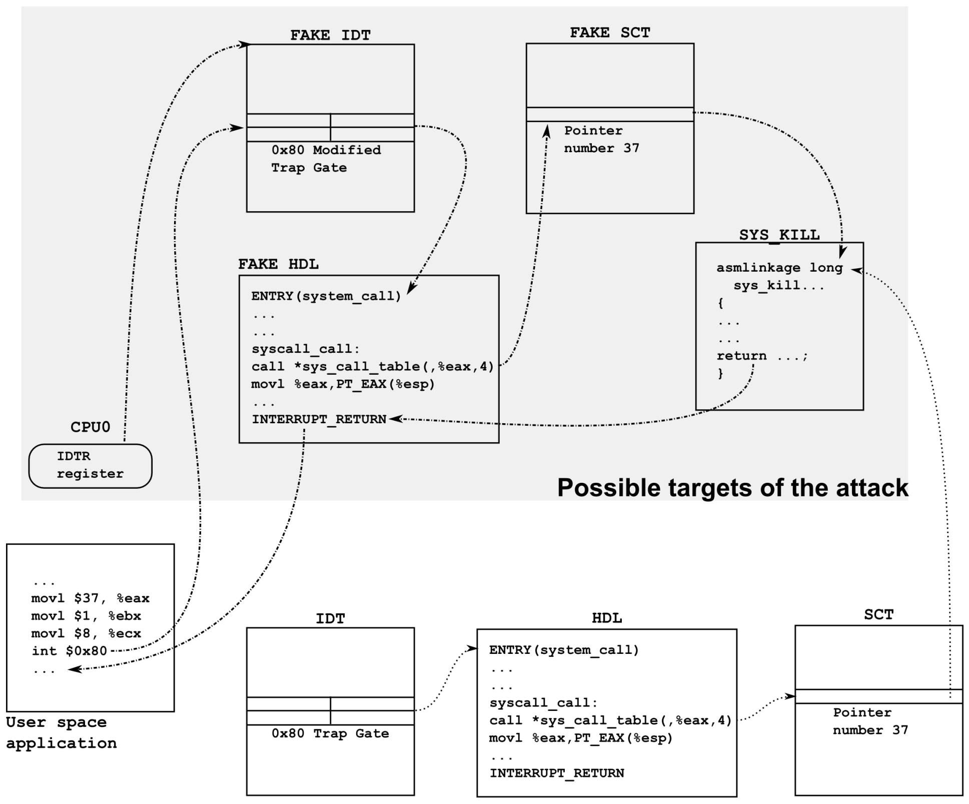


Figure 3.1: Execution flow of a system call invocation under a YASI-patched kernel (x86 architecture)

Chapter 4

Possible enhancements

One of the first objections to this work might be that the `int 0x80` interrupt mechanism is obsolete and almost completely phased out in favor of the newer, faster and vendor specific `SYSENTER` (Intel) and `SYSCALL` (AMD) mechanisms[19]. Usually, modern software relies on the `glibc` library in order to execute system calls, which in turn calls the Virtual Dynamic Shared Object (VDSO). The VDSO then selects an appropriate mechanism to execute system calls, falling back to the old `int 0x80` scheme if nothing newer is supported by the processor. Figure 4.1 shows a semantically equivalent assembly code for sending a `SIGFPE` signal (8) to the `init` process (pid 1).

```
movl $37, %eax      movl $37, %eax
movl $1, %ebx       movl $1, %ebx
movl $8, %ecx       movl $8, %ecx
int $0x80           call *%gs:0x10
```

Figure 4.1: Invoking a system call in x86 assembly

Although a deep explanation about the inner workings of the `linux-gate` shared object file should be provided here, the discussion will be greatly simplified, for the sake of brevity, by saying that the `*gs:0x10` address is always pointing to the `AT_SYSINFO` ELF auxiliary vector[2], which provides the entry point for the `__kernel_vsyscall` function. The setting of such value can be seen in the `<arch/x86/vdso/vdso32/vdso32.lds.S>` file. The `__kernel_vsyscall` function will contain the assembly instruction of the selected interrupt mechanism, which might be the `sysenter` instruction on Intel hardware, `syscall` on AMD (but Intel-supported too[5]) or, as we saw before, the old `int 0x80` instruction. Without loss of generality, let's concentrate on the Intel specific `SYSENTER` mechanism, which, as stated by the Instruction Reference on the Intel manual[5], "executes a fast call to a level 0 system procedure or routine". Such mechanism uses different machine specific registers (MSRs), but the only register an attacker might be interested in is the `IA32_SYSENTER_EIP`[5] MSR, which provides the entry point to the `SYSENTER` handler code. Such handler, called `ia32_sysenter_-target`, is defined in the `<arch/x86/kernel/entry_32.S>` file, and is very similar to the already seen `system_call` handler:

```
ENTRY(ia32_sysenter_target)
        CFI_STARTPROC simple
        CFI_SIGNAL_FRAME
        CFI_DEF_CFA esp, 0
        CFI_REGISTER esp, ebp
        movl TSS_sysenter_sp0(%esp),%esp
...

sysenter_do_call:
        cmpl $(nr_syscalls), %eax
        jae syscall_badsys
        call *sys_call_table(,%eax,4)
        movl %eax,PT_EAX(%esp)
        LOCKDEP_SYS_EXIT
        DISABLE_INTERRUPTS(CLBR_ANY)
        TRACE_IRQS_OFF
        movl TI_flags(%ebp), %ecx
        testl $_TIF_ALLWORK_MASK, %ecx
        jne sysexit_audit
sysenter_exit:
...
```

Since MSRs can also be read or written in C with extended assembly using the `rdmsr` and `wrmsr` assembly instructions[8], there are striking similarities between this `IA32_SYSENTER_EIP` register and the already seen `IDTR` register:

```
#define IA32_SYSENTER_EIP 0x176
u64 value = 0;
...
/* read */
__asm__ __volatile__ ("rdmsr" : "=A" (value) : "c" (
    IA32_SYSENTER_EIP));
...
/* write */
__asm__ __volatile__ ("wrmsr" : : "c" (IA32_SYSENTER_EIP), "a" ((u32
    ) value), "d" ((u32) (value >> 32)) : "memory");
```

While the YASI patch does not provide any kind of protection against attacks on the `SYSENTER` mechanism, securing the `IA32_SYSENTER_EIP` register should not require any more work than securing the `IDTR` register, which is an already achieved goal.

As stated in the previous chapter, the other major objection to this work is that it does not prevent code-injection in the system calls, since, in the `yasi_create_chain` function, the fake SCT is created by simply copying the contents of the true SCT:

```
memcpy((void *) info->sct, sys_call_table, NR_syscalls * 4);
```

While a code-injection attack is in theory possible, it is not relatively common, requiring a deeper knowledge of the target in comparison to a simple system call theft. Typically, a certain number of bytes from the beginning of the attacked function are saved, depending on the length of a `call` or `jump` instruction on the target's architecture. These bytes are then overwritten with a `call` or `jump` instruction pointing to the malicious code that, on return, takes appropriate actions to preserve the original semantics of the function. A very quick and dirty workaround to this problem would be saving a certain amount of bytes (or a hash) from the beginning of each system call and checking if the suspected kernel module, after its initialization or on any context switch, has modified such bytes. However, this is a larger topic and problem for which it is due to refer to the StMichael/StJude intrusion detection systems[17].

The third and last objection, apart from the clear weakness of not protecting against asynchronous attacks, is that the YASI patch cannot defend against memory scanning attacks[18], recalling that they are used to get the SCT address. However, memory scanning techniques require at least a partial knowledge on where to start and where to finish the attempt, a knowledge based on the fact that the SCT is between two known exported symbols. If the SCT is "moved" before compiling the kernel, making the premise false, the attack suddenly becomes ineffective, requiring the unmodified version of the kernel. The last resort for an attacker is a complete scan of the memory, which looks somewhat implausible, lengthy and probably vulnerable to a hardware or software timer if executed synchronously.

Appendix A

YASI patch code

```
diff -uprN -X linux-vanilla/Documentation/dontdiff linux-vanilla/
    init/main.c linux/init/main.c
--- linux-vanilla/init/main.c 2013-09-10 02:57:42.000000000 +0200
+++ linux/init/main.c 2013-12-11 16:47:21.352839765 +0100
@@ -69,6 +69,10 @@
 #include <linux/slab.h>
 #include <linux/perf_event.h>

+#ifdef CONFIG_YASI
+#include <linux/yasi.h>
+#endif
+
 #include <asm/io.h>
 #include <asm/bugs.h>
 #include <asm/setup.h>
@@ -636,6 +640,15 @@ asmlinkage void __init start_kernel(void

        ftrace_init();
+ #ifdef CONFIG_YASI
+ /*
+ * This is a good position to read and store the real IDTR of
+ * the system. Later on, rest_init() will call into cpu_idle().
+ */
+ __asm__ __volatile__ ("sidt %0" : "=m" (yasi_idtr));
+ printk(KERN_INFO "YASI KPP is up.\n");
+ #endif /* CONFIG_YASI */
+
       /* Do the rest non-__init'ed, we're now alive */
       rest_init();
 }
diff -uprN -X linux-vanilla/Documentation/dontdiff linux-vanilla/
    kernel/Makefile linux/kernel/Makefile
--- linux-vanilla/kernel/Makefile 2013-09-10 02:57:42.000000000
    +0200
+++ linux/kernel/Makefile 2013-11-28 22:33:11.455230277 +0100
@@ -46,6 +46,9 @@ obj-$(CONFIG_SMP) += smp.o
 ifneq ($(CONFIG_SMP),y)
 obj-y += up.o
 endif
+ifeq ($(CONFIG_YASI),y)
+obj-y += yasi.o
+endif
 obj-$(CONFIG_SMP) += spinlock.o
 obj-$(CONFIG_DEBUG_SPINLOCK) += spinlock.o
 obj-$(CONFIG_PROVE_LOCKING) += spinlock.o
diff -uprN -X linux-vanilla/Documentation/dontdiff linux-vanilla/
    kernel/module.c linux/kernel/module.c
```

```
--- linux-vanilla/kernel/module.c 2013-09-10 02:57:42.000000000
    +0200
+++ linux/kernel/module.c 2013-12-11 16:47:01.824854170 +0100
@@ -59,6 +59,11 @@
 #include <linux/pfn.h>
 #include <linux/bsearch.h>

+#ifdef CONFIG_YASI
+#include <linux/yasi.h>
+#include <asm/current.h> /* current macro */
+#endif
+
 #define CREATE_TRACE_POINTS
 #include <trace/events/module.h>

@@ -2958,6 +2963,12 @@ SYSCALL_DEFINE3(init_module, void __user
        struct module *mod;
        int ret = 0;

+ #ifdef CONFIG_YASI
+ struct yasi_task_info *ptr_ypt;
+ unsigned long yasi_irq_flags;
+ void *deferred_vfree;
+ #endif /* CONFIG_YASI */
+
        /* Must have permission */
        if (!capable(CAP_SYS_MODULE) || modules_disabled)
               return -EPERM;
@@ -2983,12 +2994,112 @@ SYSCALL_DEFINE3(init_module, void __user
                            mod->init_size);
```

```
       do_mod_ctors(mod);
+
+ #ifdef CONFIG_YASI
+ /*
+ * We need to keep track of which process (task struct)
+ * has entered this section (init_module). We do this
+ * with a list of yasi_task_info structs (see <linux/yasi.h>).
+ */
+ ptr_ypt = kmalloc(sizeof *ptr_ypt, GFP_KERNEL);
+ if (((void *) ptr_ypt) == NULL) {
+  ret = -1;
+  goto yasi_handle_error; /* jump to common exit */
+ }
+
+ /* Set the fields of the yasi_task_info struct */
+ ptr_ypt->task = current;
+ INIT_LIST_HEAD(&(ptr_ypt->list));
+
+ /*
+ * What we want to do is to make each task see a different chain:
+ * IDTR -> IDT -> SYSCALL_HANDLER -> SYSCALL_TABLE
+ * Let's generate a copy (per-task) of all of these structures
+ * somewhere in the memory, with yasi_create_chain function.
+ *
+ * We don't really need ret now, it will be overwritten later by
+ * the real return value of the module init().
+ * If yasi_create_chain fails, deny module load.
+ */
+ ret = yasi_create_chain(&(ptr_ypt->info));
+ if (unlikely(ret != 0)) {
+  /*
+  * Actually, we have only -ENOMEM and -EPERM.
+  * -EPERM means that we couldn't hot-patch memory as needed.
+  */
+  if (ret == -ENOMEM)
+   printk(KERN_WARNING "[YASI] Unable to allocate enough"
+   "memory! Denying execution of module %s\n", mod->name);
+  else
+   printk(KERN_WARNING "[YASI] Unable to create fake "
+   "chain! Denying execution of module %s\n", mod->name);
+  goto yasi_handle_error;
+ }
+
+ /*
+ * Everything we need is in place, modify the list atomically,
+ * and then hijack the IDTR.
+ */
+ write_lock_irqsave(&yasi_lock, yasi_irq_flags);
+ list_add_tail(&(ptr_ypt->list), &yasi_list);
+ write_unlock_irqrestore(&yasi_lock, yasi_irq_flags);
+
+ /* The list now is not empty, the scheduler will check this
+ * condition when doing context switches.
+ *
+ * At this point we're done, if code is preempted, scheduler
+ * will set back a safe
+ * value for the IDTR. Otherwise, carry on with a fake environment.
+ *
+ */
+ __asm__ __volatile__ ("lidt %0" : : "m" (ptr_ypt->info.idtr));
+ #endif /* CONFIG_YASI */
+
        /* Start the module */
        if (mod->init != NULL)
                ret = do_one_initcall(mod->init);
+
+ #ifdef CONFIG_YASI
+
+ /*
+ * Was the module malicious? We don't want to compute this check in
+ * atomic context, so we do this here, before we acquire a lock.
+ * No problem if someone preempts, scheduler will take care.
+ */
+ yasi_system_safe(&(ptr_ypt->info), mod->name);
+
+ /* Now go in atomic context... */
+ write_lock_irqsave(&yasi_lock, yasi_irq_flags);
+
+ /*
+ * ... and remove the current task from the list (and don't forget
+ * to save a pointer to the vmalloc'ed memory).
+ */
+ list_del(&(ptr_ypt->list));
+ deferred_vfree = (void *) ptr_ypt->info.hdl;
+ yasi_delete_chain(&(ptr_ypt->info));
+ kfree(ptr_ypt);
+
+ /* The fake IDTR is now invalid. We have to restore the safe IDTR
+ * being atomic, otherwise we will get a nice page fault OOPS when
+ * serving interrupts.
+ */
+ __asm__ __volatile__ ("lidt %0" : : "m" (yasi_idtr));
+ write_unlock_irqrestore(&yasi_lock, yasi_irq_flags);
+
+ /* Deferred vfree */
+ vfree(deferred_vfree);
+ #endif /* CONFIG_YASI */
+
        if (ret < 0) {
                /* Init routine failed: abort. Try to protect us from
                   buggy refcounters. */
+
+ #ifdef CONFIG_YASI
+ /* Common exit code in case of error */
+ yasi_handle_error:
+ #endif /* CONFIG_YASI */
+
                mod->state = MODULE_STATE_GOING;
                synchronize_sched();
                module_put(mod);
diff -uprN -X linux-vanilla/Documentation/dontdiff linux-vanilla/
    kernel/sched.c linux/kernel/sched.c
--- linux-vanilla/kernel/sched.c 2013-09-18 15:25:47.000000000 +0200
+++ linux/kernel/sched.c 2013-12-11 16:54:55.036855610 +0100
@@ -80,6 +80,10 @@
 #include <asm/paravirt.h>
 #endif

+#ifdef CONFIG_YASI
```

```
+#include <linux/yasi.h>
+#endif
+
 #include "sched_cpupri.h"
 #include "workqueue_sched.h"
 #include "sched_autogroup.h"
@@ -3278,6 +3282,39 @@ asmlinkage void schedule_tail(struct tas
              put_user(task_pid_vnr(current), current->set_child_tid
                 );
 }

+#ifdef CONFIG_YASI
+/*
+ * We have prev and next as parameters (task_struct *):
+ * If prev is in our list, we restore the safe IDTR.
+ * If next is in our list, we set the fake IDTR.
+ *
+ * Do not make a call to yasi_switch. It is in an external function
     for
+ * styling reasons (__always_inline).
+ */
+static void __always_inline yasi_switch(const struct task_struct *
    const prev,
+ const struct task_struct *const next)
+{
+ struct yasi_task_info *ptr_ypt;
+
+ /* We are under a read_lock here... */
+ list_for_each_entry(ptr_ypt, &yasi_list, list) {
+  if (unlikely(ptr_ypt->task == next)) {
```

```
+   /* Set the fake IDTR */
+   __asm__ __volatile__ ("lidt %0" : : "m"
+    (ptr_ypt->info.idtr));
+   /*
+   * Now break the list_for_each_entry loop!
+   * Anything else won't matter now.
+   */
+   break;
+  }
+  if (unlikely(ptr_ypt->task == prev))
+   /* Set real IDTR */
+   __asm__ __volatile__ ("lidt %0" : : "m" (yasi_idtr));
+ }
+}
+#endif /* CONFIG_YASI */
+
 /*
  * context_switch - switch to the new MM and the new
  * thread's register state.
@@ -3288,6 +3325,20 @@ context_switch(struct rq *rq, struct tas
 {
       struct mm_struct *mm, *oldmm;

+ #ifdef CONFIG_YASI
+
+ /*
+ * Switch the IDTR (if needed) as early as possibile.
+ *
+ * The speedup of the unlikely macro should offset the
+ * pipeline penalty incurred when the branching is wrong.
+ */
+ read_lock(&yasi_lock); /* irqs should be already disabled here */
+ if (unlikely(!list_empty(&yasi_list)))
+  yasi_switch(prev, next);
+ read_unlock(&yasi_lock);
+ #endif /* CONFIG_YASI */
+
       prepare_task_switch(rq, prev, next);

       mm = next->mm;
diff -uprN -X linux-vanilla/Documentation/dontdiff linux-vanilla/
    kernel/yasi.c linux/kernel/yasi.c
--- linux-vanilla/kernel/yasi.c 1970-01-01 01:00:00.000000000 +0100
+++ linux/kernel/yasi.c 2013-12-11 22:58:20.510205735 +0100
@@ -0,0 +1,384 @@
+#include <linux/yasi.h>
+#include <linux/slab.h> /* kmalloc stuff */
+#include <linux/vmalloc.h> /* vmalloc stuff */
+#include <linux/unistd.h> /* NR_syscalls */
+#include <asm/syscall.h> /* sys_call_table */
+#include <asm/desc.h> /* for pack_gate in yasi_overwrite_gate */
+#include <asm/pgtable_types.h> /* __pgprot and __PAGE_KERNEL_EXEC
    */
+#include <crypto/hash.h> /* we want to compute MD5 hashes */
+
+/* This is our safe IDTR. It is properly set in init/main.c (early
    boot) */
+struct desc_ptr yasi_idtr = {0, 0};
+
+/* Here are defined a read-write lock and an empty list */
```

```
+DEFINE_RWLOCK(yasi_lock);
+LIST_HEAD(yasi_list);
+
+/* Search length - see yasi_patch_handler function */
+#define S_LEN 128
+/* Copy length - see yasi_create_chain function */
+#define C_LEN 512
+/* Lenght of a MD5 digest - see yasi_system_safe and nested
    functions */
+#define MD5_LEN 16
+
+/* Static function definitions */
+/*************************************************************/
+
+/*
+ * yasi_patch_handler: search for the 0xff 0x14 0x85 pattern and
    replace it.
+ *
+ * Parameter size is S_LEN, which is 128 bytes. As stated below in
+ * yasi_create_chain, syscall_call is 69 bytes away, 128 should
    suffice.
+ */
+static int yasi_patch_handler(const unsigned long addr, const
    unsigned int size,
+ const unsigned long sct_addr)
+{
+ unsigned int i;
+ unsigned char is_found = 0;
+ unsigned char *ptr = (unsigned char *) addr;
+
+ for (i = 0; i < size; i++) {
+  ptr++;
+  if (*(ptr) == 0xff && *(ptr + 1) == 0x14
+   && *(ptr + 2) == 0x85) {
+   /* We have found the pattern, patch! */
+   *(unsigned long *)(ptr + 3) = sct_addr;
+   is_found = 1;
+   break;
+  }
+ }
+
+ /* Return value accordingly */
+ if (is_found == 1)
+  return 0;
+ else return -EPERM; /* return -1 */
+}
+
+/*
+ * yasi_overwrite_gate is heavily inspired from the real kernel
    code in
+ * <arch/x86/include/asm/desc.h>.
+ * The idea is to copy what the set_system_trap_gate function does.
+ */
+static inline void yasi_overwrite_gate(gate_desc *const idt,
+ const unsigned long addr)
+{
+ gate_desc s;
+
+ pack_gate(&s, GATE_TRAP, addr, 0x3, 0, __KERNEL_CS);
+ memcpy(&(idt[0x80]), &s, sizeof s);
```

```
+}
+
+/*
+ * Do the real MD5 hashing, using crypto_shash API.
+ * This function is used solely by yasi_hash_md5 (see below).
+ */
+static int yasi_do_md5(struct crypto_shash *const shash_tfm,
+ const unsigned char *const input, const unsigned int len,
+  unsigned char *const output)
+{
+ struct {
+  struct shash_desc shash;
+  char ctx[crypto_shash_descsize(shash_tfm)];
+ } desc;
+
+ desc.shash.tfm = shash_tfm;
+ desc.shash.flags = CRYPTO_TFM_REQ_MAY_SLEEP;
+
+ return crypto_shash_digest(&desc.shash, input, len, output);
+}
+
+/*
+ * yasi_hash_md5 computes a MD5 hash of the input and stores it in
    the
+ * specified output.
+ *
+ * Code is greatly inspired from <security/integrity/ima/ima_crypto.
    c>
+ */
+static int yasi_hash_md5(const unsigned char *const input,
+ const unsigned int len, unsigned char *const output)
+{
+ struct crypto_shash *shash_tfm;
+ int ret;
+
+ shash_tfm = crypto_alloc_shash("md5", 0, 0);
+ if (IS_ERR(shash_tfm))
+  return -EPERM;
+
+ ret = yasi_do_md5(shash_tfm, input, len, output);
+ crypto_free_shash(shash_tfm);
+ return ret;
+}
+
+/*
+ * Compare byte a byte between the original (addr_o) and copy (
    addr_c)
+ * Code is a slight variation of memcmp in <lib/string.c>.
+ */
+static int yasi_chk_mem(const unsigned char *const addr_o,
+ const unsigned char *const addr_c, unsigned int size)
+{
+ const unsigned char *ptr_o, *ptr_c;
+
+ for (ptr_o = addr_o, ptr_c = addr_c; size > 0; ptr_o++, ptr_c++,
+  --size) {
+  if (*(ptr_c) != *(ptr_o))
+   return 0;
+ }
+ return 1;
```

```
+}
+
+/*
+ * Check if the IDT has been modified after module load.
+ */
+static int yasi_IDT_safe(const unsigned long idt_addr,
+ const unsigned char *const true_hash)
+{
+ unsigned char tmp_hash[MD5_LEN];
+
+ if (yasi_hash_md5((unsigned char *) idt_addr, yasi_idtr.size,
+  tmp_hash) != 0) {
+  /*
+  * We cannot perform checking! Warn the user, but return
+  * conservatively 1, we cannot return 0 here, we are unsure.
+  */
+  printk(KERN_WARNING "[YASI] Error! Unable to check memory!\n");
+  return 1;
+ }
+
+ /* We have the MD5 hash here in tmp_hash, compare it byte a byte
    */
+ return yasi_chk_mem(true_hash, tmp_hash, MD5_LEN);
+}
+
+/*
+ * Check if the HDL has been modified after module load.
+ */
+static int yasi_HDL_safe(const unsigned long hdl_addr,
+ const unsigned char *const true_hash)
```

```
+{
+ unsigned char tmp_hash[MD5_LEN];
+
+ if (yasi_hash_md5((unsigned char *) hdl_addr, C_LEN,
+  tmp_hash) != 0) {
+  /*
+  * We cannot perform checking! Warn the user, but return
+  * conservatively 1, we cannot return 0 here, we are unsure.
+  */
+  printk(KERN_WARNING "[YASI] Error! Unable to check memory!\n");
+  return 1;
+ }
+
+ /* We have the MD5 hash here in tmp_hash, compare it byte a byte
    */
+ return yasi_chk_mem(true_hash, tmp_hash, MD5_LEN);
+}
+
+/*
+ * Check if the SCT has been modified after module load.
+ */
+static inline int yasi_SCT_safe(const unsigned long sct_addr)
+{
+ /*
+ * Computing hashes is quite expensive. We could not do otherwise
+ * with the HDL and the IDT, but here we can avoid this cost since
+ * we have a good and safe copy in memory.
+ */
+ return yasi_chk_mem((unsigned char *) sys_call_table,
+  (unsigned char *) sct_addr, NR_syscalls * 4);
+}
+
+/* Extern function definitions */
+/**********************************************************/
+
+/*
+ * yasi_delete_chain: simply free every kmalloc'ed memory.
+ * We cannot free vmalloc'ed memory, because we are under spinlocks,

+ * lockdep will start whining about HARDIRQ unsafe scenarios.
+ *
+ * Just remember to save a pointer to the vmalloc'ed memory,
+ * call this function and, when it's safe to do so, vfree that
    pointer.
+ */
+void yasi_delete_chain(struct yasi_chain_info *const info)
+{
+ kfree((void *) info->sct);
+ kfree((void *) info->idtr.address);
+}
+
+/*
+ * yasi_create_chain does the following stuff:
+ *
+ * Create a copy of the SCT
+ * Create a copy of the syscall_handler
+ * Patch the syscall_handler copy
+ * Create a copy of the IDT and a new IDTR
+ * Patch the copy of the IDT
+ * Create some MD5 hashes...
+ *
+ * ... and put all the needed stuff in *info.
+ */
+
+int yasi_create_chain(struct yasi_chain_info *const info)
+{
+ /* Default return code */
+ int ret = 0;
+
+ /*
+ * From my own System.map-3.2.51-zuarte:
+ * xxxx06c4 t work_pending
+ * xxxx060d t syscall_call
+ * xxxx05c8 T system_call
+ *
+ * The difference between work_pending and system_call
+ * is 0xfc = 252 (decimal). C_LEN = 512 looks very safe.
+ *
+ * However, the pattern the attacker is interested in is
+ * at syscall_call and tt's just 0x45 = 69 bytes off.
+ *
+ * Ideally, we could do some ASM magic, declaring a label as
+ * a function in <arch/x86/kernel/entry_32.S>, in order to get
+ * quickly its address. It is not much cleaner either though.
+ *
+ * I'm _VERY_ sorry for the _UGLY_ code here.
+ */
+
+ /*
+ * Create a copy of the System Call Table.
+ * Don't forget that each entry is 4 bytes long.
+ */
+ info->sct = (unsigned long) kmalloc(NR_syscalls * 4, GFP_KERNEL);
+ if (((void *) info->sct) == NULL) {
+  ret = -ENOMEM; /* -12 */
+  goto yasi_exit;
+ }
+ memcpy((void *) info->sct, sys_call_table, NR_syscalls * 4);
+
+ /*
+ * Create a copy of the syscall_handler.
+ * In order to do that grab the address of the system_call
+ * entry point from the real IDTR.
+ *
+ * Use __vmalloc to get executable pages.
+ */
+ info->hdl = (unsigned long) __vmalloc(C_LEN, GFP_KERNEL,
+  __pgprot(__PAGE_KERNEL_EXEC));
+ if (((void *) info->hdl) == NULL) {
+  /* Free the sct memory */
+  kfree((void *) info->sct);
+  ret = -ENOMEM; /* -12 */
+  goto yasi_exit;
+ }
+
+ /* See gate_offset in <arch/x86/include/asm/desc_defs.h> */
+ memcpy((void *) info->hdl,
+  (void *) gate_offset(*((gate_desc *) yasi_idtr.address + 0x80)),
+   C_LEN);
+
+ /*
+ * Now patch the handler. What we need to do is to find a
+ * 0xff 0x14 0x85 pattern, the next 4 bytes are the
+ * system call table address.
+ */
+ if ((ret = yasi_patch_handler(info->hdl, S_LEN, info->sct) != 0))
   {
+  /* Free the sct and handler memory */
+  vfree((void *) info->hdl);
+  kfree((void *) info->sct);
+  goto yasi_exit; /* return -1 (-EPERM) */
+ }
+
+ /* Create a copy of the IDT and a new IDTR */
+ info->idtr.size = yasi_idtr.size;
+ info->idtr.address = (unsigned long) kmalloc(yasi_idtr.size,
   GFP_KERNEL);
+ if (((void *) info->idtr.address) == NULL) {
+  /* Free the sct and handler memory */
+  vfree((void *) info->hdl);
+  kfree((void *) info->sct);
+  ret = -ENOMEM; /* -12 */
+  goto yasi_exit;
+ }
+ memcpy((void *) info->idtr.address, (void *) yasi_idtr.address,
+  yasi_idtr.size);
+
+ /*
+ * Patch the IDT! Go to the 0x80th entry of the copied IDT
+ * and you'll find there a gate descriptor, patch the address
+ * to our entry point.
+ */
+ yasi_overwrite_gate((gate_desc *) info->idtr.address, info->hdl);
+
+ /*
+ * Now we have:
+ * A fake IDTR -> a fake IDT -> 0x80 (fake trap gate) ->
+ * fake syscall_handler -> (fake call) [ff 14 85 xx xx xx xx] :)
+ *
+ * Now create the hashes of the fake IDT and HDL. We want to know
    if
+ * the module will modify these structures. We don't need to
+ * store the hash of the SCT because we already have a good copy
+ * in memory (sys_call_table symbol).
+ */
+ ret = yasi_hash_md5((unsigned char *) info->idtr.address,
+  yasi_idtr.size, info->idt_hash);
+ if (ret != 0) {
+  /* Free all the requested memory so far */
+  kfree((void *) info->idtr.address);
+  vfree((void *) info->hdl);
+  kfree((void *) info->sct);
+  goto yasi_exit; /* return -1 (-EPERM) */
+ }
+
+ ret = yasi_hash_md5((unsigned char *) info->hdl, C_LEN,
+  info->hdl_hash);
+ if (ret != 0) {
+  /* Free all the requested memory so far */
+  kfree((void *) info->idtr.address);
+  vfree((void *) info->hdl);
+  kfree((void *) info->sct);
+  goto yasi_exit; /* return -1 (-EPERM) */
+ }
+
+ /* Perfect, now remember that, on success, we return 0. */
+yasi_exit:
+ return ret;
+}
+
+/*
+ * yasi_system_safe: check behaviour of the module after loading it.

+ * return value is 1 if the system is safe, 0 otherwise.
+ */
+void yasi_system_safe(const struct yasi_chain_info *const info,
+  const char *const m_name)
+{
+
+ struct desc_ptr current_idtr;
+
+ __asm__ __volatile__ ("sidt %0" : "=m" (current_idtr));
+
+ if (unlikely(current_idtr.address != info->idtr.address)) {
+  printk(KERN_ALERT "[YASI] module %s has tried to modify "
+  "at least the IDTR.\n The system is SAFE but it is suggested "
+  "to reboot to revert to a clean state.\n"
+  "Do not rmmod the module, a kernel OOPS may appear.\n", m_name);
+ }
+
+ else if (unlikely(yasi_IDT_safe(info->idtr.address, info->idt_hash
    ) == 0)) {
+  printk(KERN_ALERT "[YASI] module %s has tried to modify "
+  "at least the IDT.\nThe system is SAFE but it is suggested "
+  "to reboot to revert to a clean state.\n"
+  "Do not rmmod the module, a kernel OOPS may appear.\n", m_name);
+
+ }
+
+ else if (unlikely(yasi_HDL_safe(info->hdl, info->hdl_hash) == 0))
    {
+  printk(KERN_ALERT "[YASI] module %s has tried to modify "
+  "at least the system call handler.\nThe system is SAFE but "
+  "it is suggested to reboot to revert to a clean "
+  "state.\n"
+  "Do not rmmod the module, a kernel OOPS may appear.\n", m_name);
+
+ }
+
+ else if (unlikely(yasi_SCT_safe(info->sct) == 0)) {
+  printk(KERN_ALERT "[YASI] module %s has tried to modify "
+  "the system call table.\nThe system is SAFE but it is "
+  "suggested to reboot to revert to a clean state.\n"
+  "Do not rmmod the module, a kernel OOPS may appear.\n", m_name);
+ }
+
+ else printk(KERN_INFO "[YASI] No malicious behaviour detected in "
+  "module %s.\n", m_name);
+}
diff -uprN -X linux-vanilla/Documentation/dontdiff linux-vanilla/
```

```
    security/Kconfig linux/security/Kconfig
--- linux-vanilla/security/Kconfig 2013-09-10 02:57:42.000000000
    +0200
+++ linux/security/Kconfig 2013-12-11 23:00:51.839192369 +0100
@@ -183,6 +183,19 @@ config LSM_MMAP_MIN_ADDR
         this low address space will need the permission specific to
              the
         systems running LSM.

+config YASI
+ bool "YASI Kernel Patch Protection (DANGEROUS)"
+ depends on EXPERIMENTAL && MODULES && X86_32 && CRYPTO_MD5
+ default n
+ help
+ This activates the YASI KPP, which tries to protect the kernel
    against
+ malicious synchronous LKM.
+
+ The code is currently under development and, as such, it is very
+ experimental.
+
+ If you are unsure how to answer this question, answer N.
+
 source security/selinux/Kconfig
 source security/smack/Kconfig
 source security/tomoyo/Kconfig
```

Appendix B

Evil kernel code - HDL patching

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/unistd.h>

static unsigned long *pattern_addr;
static void ** sct;
static void ** new_sct;

static unsigned long * find_sct(const unsigned long addr,
        const unsigned int size)
{
        unsigned int i;
        unsigned char *ptr;

        ptr = (unsigned char *) addr;

        for (i = 0; i < size; i++) {
                ptr++;
                if (*ptr == 0xff && *(ptr + 1) == 0x14
                        && *(ptr + 2) == 0x85) {
                        return (unsigned long *)(ptr + 3);
                }
        }
        /* didn't find */
        return NULL;
}

int init_module(void)
{
        struct desc_ptr idtr;
        unsigned long handler;

        __asm__ __volatile__ ("sidt %0" : "=m" (idtr));
        printk(KERN_INFO "[%s] IDT is @ 0x%lx\n",
                __this_module.name, idtr.address);

        handler = gate_offset(*((gate_desc *) idtr.address + 0x80));
        printk(KERN_INFO "[%s] HDL is @ 0x%lx\n",
                __this_module.name, handler);

        pattern_addr = find_sct(handler, 128);

        if (pattern_addr == NULL) {
                printk(KERN_INFO "[%s] SCT not found\n", __this_module
                    .name);
                return -1;
        } else {
                sct = (void *) *pattern_addr;
                printk(KERN_INFO "[%s] SCT is @ 0x%lx\n",
                        __this_module.name, (unsigned long) sct);
        }

        new_sct = (void **) kmalloc(NR_syscalls, GFP_KERNEL);
        if ((void *) new_sct == NULL)
                return -1;
        memcpy((void *) new_sct, (void *) sct, NR_syscalls);
        printk(KERN_INFO "[%s] New SCT is @ 0x%lx\n", __this_module.
            name,
                (unsigned long) new_sct);
        get_cpu();
        write_cr0(read_cr0() & (~0x10000));
        /* Patch handler! */
        *pattern_addr = (unsigned long) new_sct;
        write_cr0(read_cr0() | (0x10000));
        put_cpu();
        return 0;
}

void cleanup_module(void)
{
        get_cpu();
        write_cr0(read_cr0() & (~0x10000));
        /* clean */
        *pattern_addr = (unsigned long) sct;
        write_cr0(read_cr0() | (0x10000));
        put_cpu();
        kfree((void *) new_sct);
}

MODULE_LICENSE("GPL");
```

References

[1] Defeating PatchGuard - Bypassing Kernel Security Patch Protection in Microsoft Windows. `http://www.mcafee.com/in/resources/reports/rp-defeating-patchguard.pdf`.

[2] ELF auxiliary vectors. `http://www.win.tue.nl/~aeb/linux/hh/hh-14.html#ss14.3`.

[3] GCC-Inline-Assembly-HOWTO. `http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html#ss5.4`.

[4] Gentoo Wiki: Signed kernel module support. `http://wiki.gentoo.org/wiki/Signed_kernel_module_support`.

[5] Intel 64 and IA-32 Architectures Software Developer's Manual. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf`.

[6] Kernelnewbies Mailing List Archive. `http://marc.info/?l=kernelnewbies&m=104123404531713`.

[7] Linus Responds To RdRand Petition With Scorn. http://linux.slashdot.org/story/13/09/10/1311247/linus-responds-to-rdrand-petition-with-scorn.

[8] Linux Cross Reference - msr.h. http://lxr.free-electrons.com/source/arch/x86/include/asm/msr.h?v=3.2#L68.

[9] Linux Cross Reference - syscall_table_32.S. http://lxr.free-electrons.com/source/arch/x86/kernel/syscall_table_32.S?v=3.2.

[10] Linux Kernel Mailing List Archive. https://lkml.org/lkml/2005/6/29/180.

[11] Linux on-the-fly kernel patching without LKM. http://www.phrack.org/issues.html?issue=58&id=7#article.

[12] Loading signed kernel modules. https://lwn.net/Articles/470906/.

[13] OSDev Wiki: Paging. http://wiki.osdev.org/Paging.

[14] Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture. http://www.freescale.com/files/product/doc/MPCFPE32B.pdf.

[15] Protection and Isolation. http://www.read.seas.harvard.edu/~kohler/class/05s-osp/notes/notes9.html.

[16] Runtime locking correctness validator. https://www.kernel.org/doc/Documentation/lockdep-design.txt.

[17] StJude/StMichael. http://www.kernelhacking.com/rodrigo/index.php?name=StMichael.

[18] StJude/StMichael FAQ. http://www.kernelhacking.com/rodrigo/docs/FAQ_saints.txt.

[19] SYSENTER. http://wiki.osdev.org/SYSENTER.

[20] We cannot trust them anymore: Engineers abandon encryption chips after Snowden leaks. http://rt.com/usa/snowden-leak-rng-randomness-019/.

[21] Wikipedia: Evaluation Assurance Level. https://en.wikipedia.org/wiki/Evaluation_Assurance_Level.

[22] Wikipedia: History of Linux. https://en.wikipedia.org/wiki/History_of_Linux.

[23] Wikipedia: Kernel Patch Protection. https://en.wikipedia.org/wiki/Kernel_Patch_Protection.

[24] Wikipedia: Virtual memory. https://en.wikipedia.org/wiki/Virtual_memory.