

Advanced Techniques Ring0/Kernel Levels Rootkit Development on Linux 2.6

- WonoKaerun -

**Indonesian Security Conference 2011
Palcomtech – Palembang**

16-17 Juli 2011

\$whoami

- ▶ InfoSec Enthusiast
- ▶ Independent IT Security Researcher
- ▶ Slackware & FreeBSD Hobbier
- ▶ Still.. a Lazy Student #FYM ;)

T : @sukebett

M : dante_at_indiefinite.com

Agenda

- ▶ Introduction
 - ▶ Definition
 - ▶ Classification
 - ▶ Main Contents
 - ▶ Demo
 - ▶ Conclusion
- 

Definition (1/4)

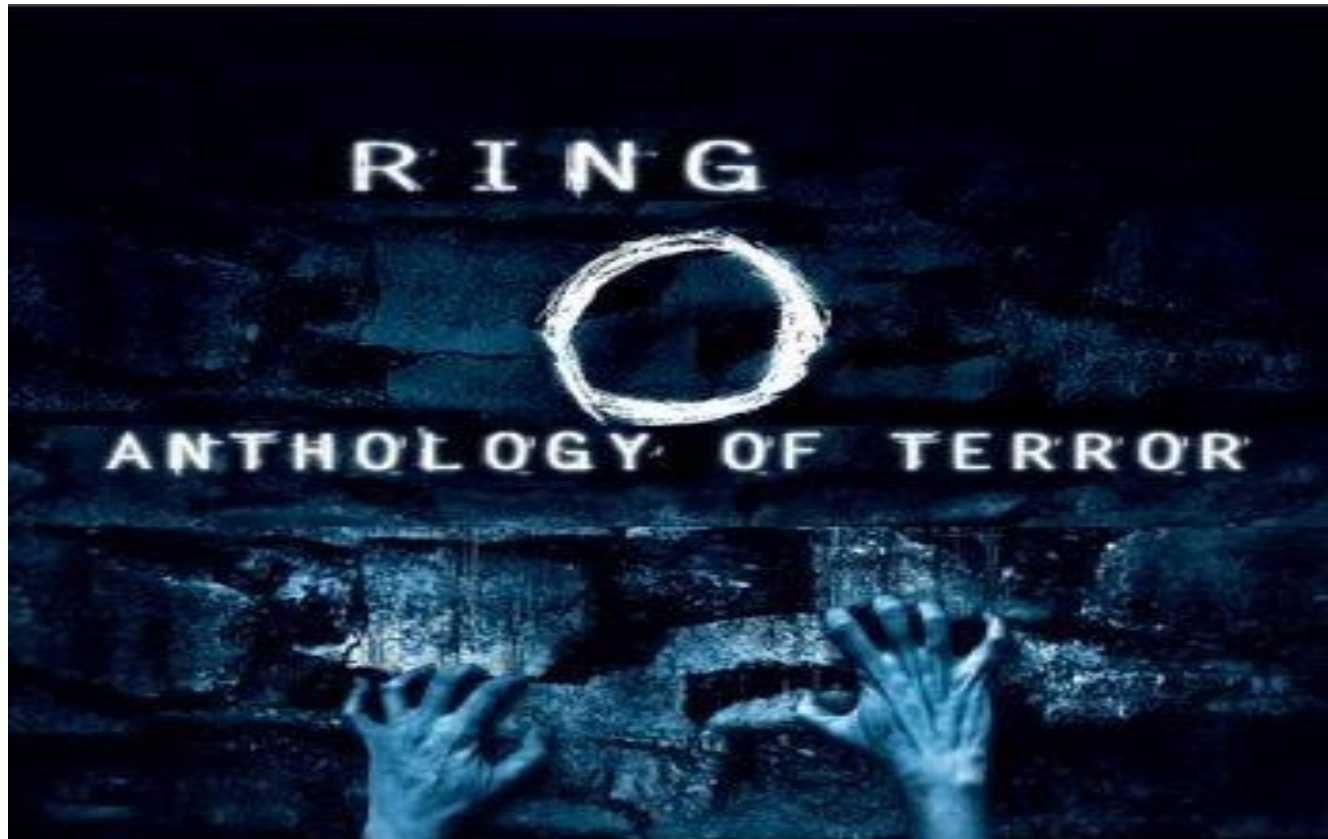
- ▶ Rootkit



Source: <http://www.flickr.com/photos/jraptor/4459405455/>

Definition (2/4)

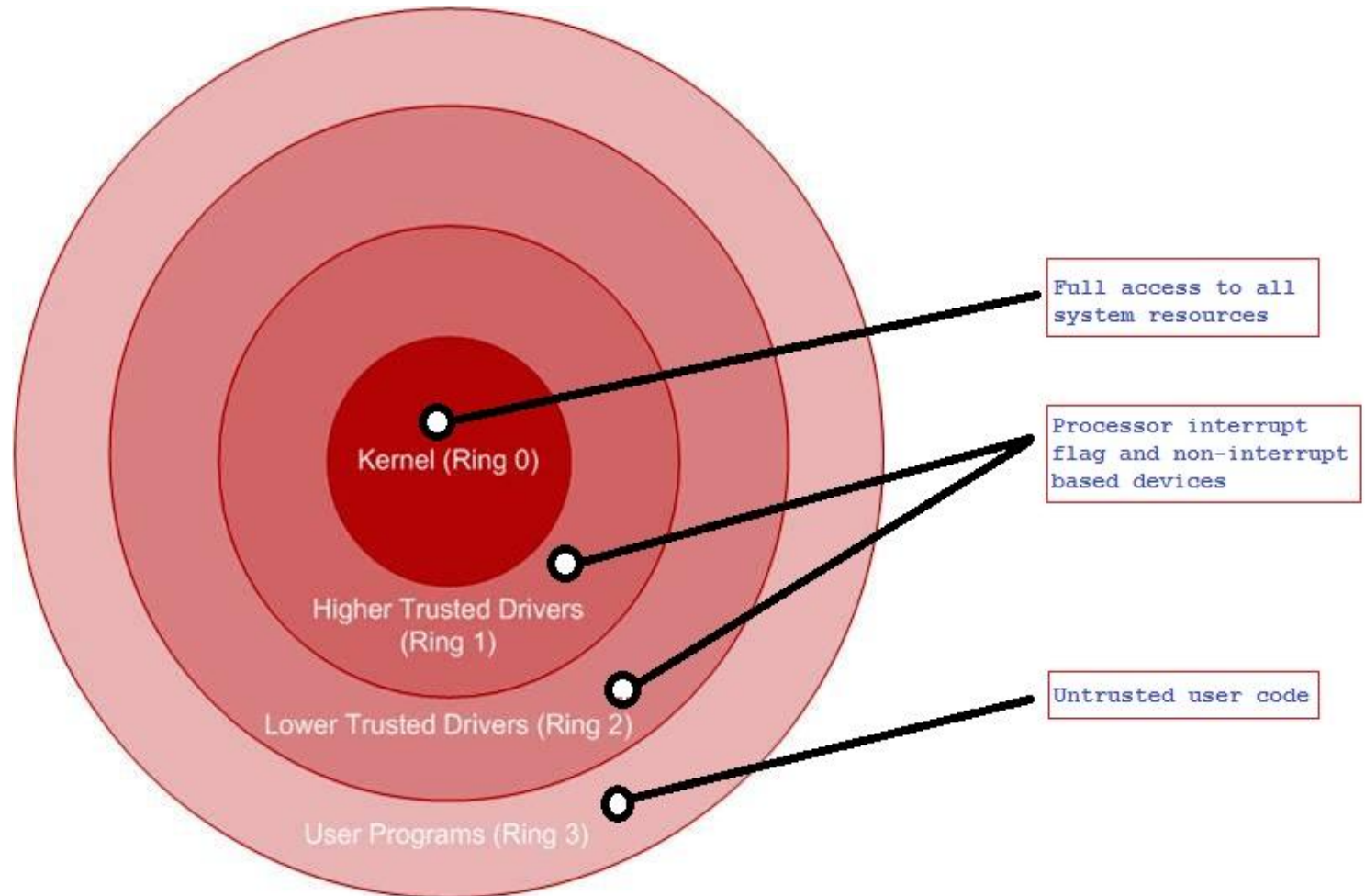
- ▶ Ring0



Source: <http://www.imdb.com/title/tt0235712/>

Definition (3/4)

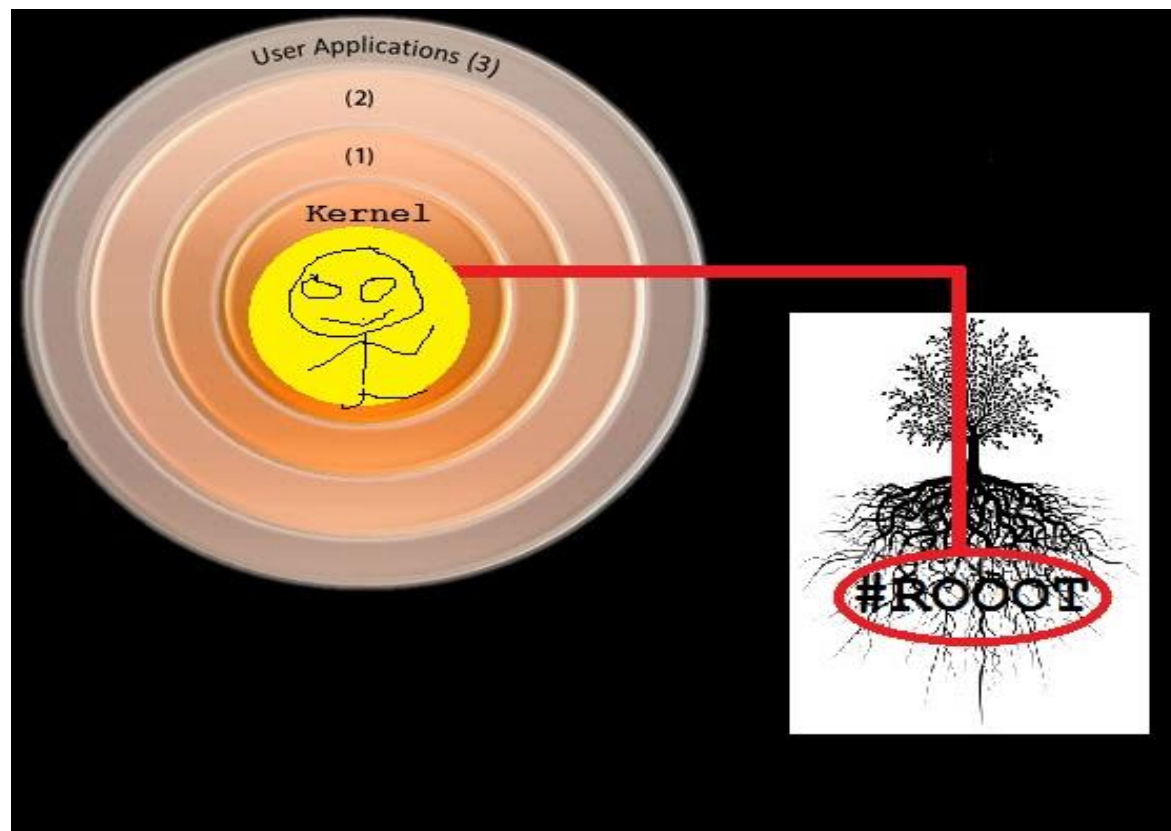
► Ring0



Definition (4/4)

- ▶ Ring0 Levels Rootkit

“Rootkits that are running at Kernel Mode!”



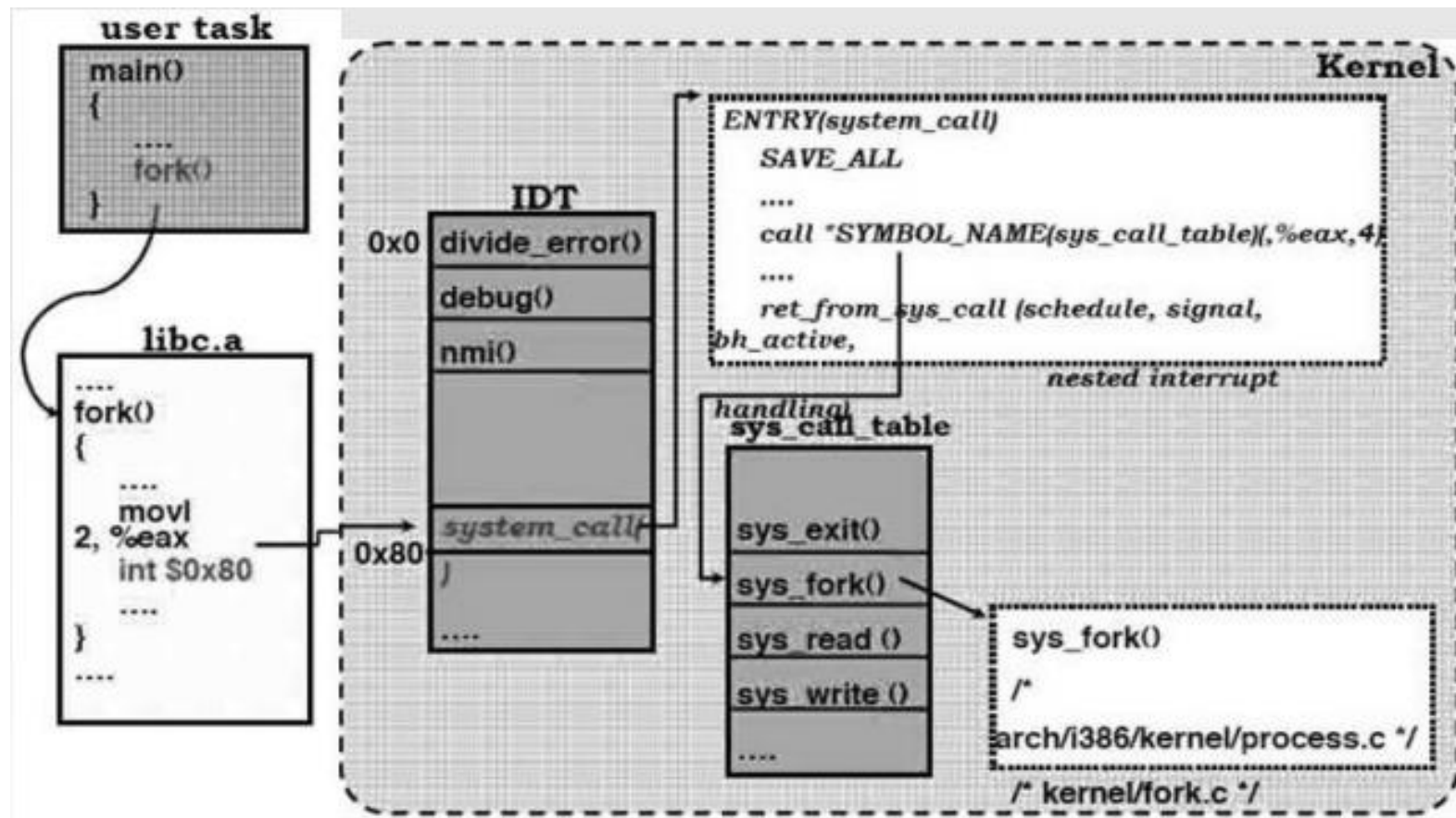
Classification

1. LKM Based Rootkit
 2. Non-LKM Based Rootkit
- So, What is LKM (Loadable Kernel Module)?



Main Contents

1.a. Hooking System Call Table Address



Main Contents

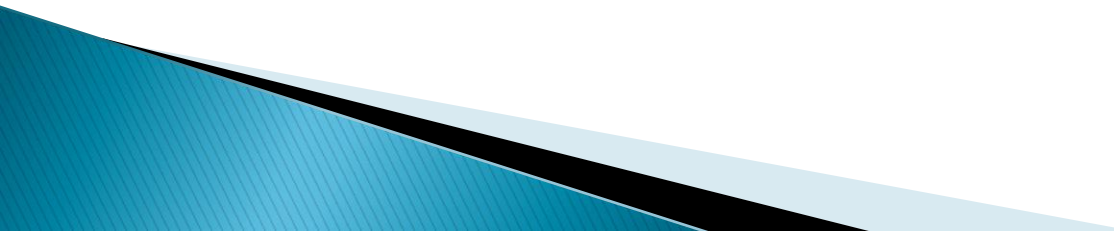
- ▶ Historically, LKM-based rootkits used the 'sys_call_table[]' symbol to perform hooks on the system calls

```
sys_call_table[__NR_sc] = (void *) hacked_sc_ptr;
```

- ▶ However, since sys_call_table[] is not an exported symbol anymore, this code isn't valid
- ▶ We need another way to find 'sys_call_table' []

Main Contents

1.b. Finding SysCallTable Address

1. Get the IDTR using SIDT
 2. Extract the IDT address from the IDTR
 3. Get the address of 'system_call' from the 0x80th entry of the IDT
 4. Search 'system_call' for our code fingerprint
 5. Finally, we should have the address of 'sys_call_table[]' !
- 

Main Contents

1.c. Bypass WP (Write Protection)

- *Problem* : `sys_call_table[]` is read-only!
- *Solution* : We must clear 16th bit of `cr0`!

```
static void disable_wp_cr0 (void) {  
    unsigned long value;  
    asm volatile("mov %%cr0,%0" : "=r" (value));  
    if (value & 0x00010000) {  
        value &= ~0x00010000;  
        asm volatile("mov %0,%%cr0": : "r" (value));  
    }  
}
```

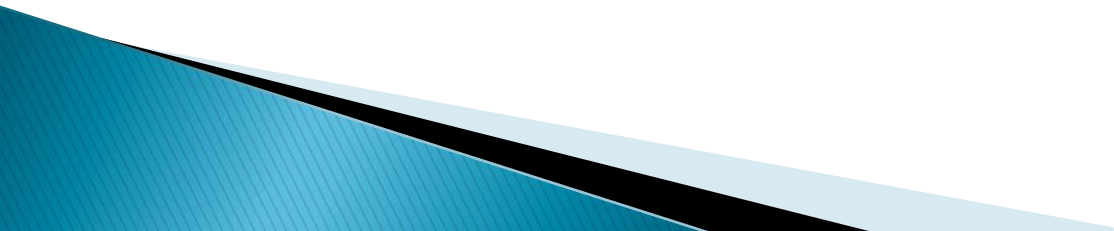
Main Contents

1.d. On x86_64

- Actually this is NOT new architecture, it's just specifically different in memory addressing plus with additional of new CPU instructions.
- We can find `sys_call_table[]` by bruteforcing in range memory address between:
[0xffffffff00000000 – 0xffffffffffffffff]

Main Contents

1.e. Capabilities

- Hiding File/Directory
 - Hiding Process
 - Hiding Network Traffic
 - Sniffing
 - Keylogging
 - Etc..
- 

Main Contents

1.f. References

- http://thc.org/papers/LKM_HACKING.html
- <http://www.phrack.org/issues.html?issue=52&id=18>
- <http://www.slideshare.net/fisher.w.y/rootkit-on-linux-x86-v26>
- <http://www.exploit-db.com/papers/13146/>

Main Contents

2.a. IDT(Interrupt Descriptor Table) Handling

- Interrupt: "An event that alters the sequence of instructions executed by a processor. Such events correspond to electrical signals generated by hardware circuits both inside and outside of the CPU chip." (Understanding the Linux kernel ,O'reilly)
- The IDT is a linear table of 256 entries which associates an interrupt handler with each interrupt vector, and each entry of the IDT is a descriptor of 8 bytes which blows the entire IDT up to a size of $256 * 8 = 2048$ bytes.

Main Contents

2.b. Hijacking Methods

1. Create a fake IDT handler
2. Copy our handler's address into new_addr
3. Make the idt variable point on the first IDT descriptor, via idt, idtr dan sidt.

(Ref. Phrack 58 article 7)

4. Save the old handler's address
(with get_stub_from_idt() function)
5. new_addr contain our handler's address!


Main Contents

References

- <http://www.phrack.org/issues.html?issue=59&id=4>
- http://codenull.net/articles/kmh_en.html
- <http://burrowscode.wordpress.com/2010/06/23/idt-hookingunhooking-module/>

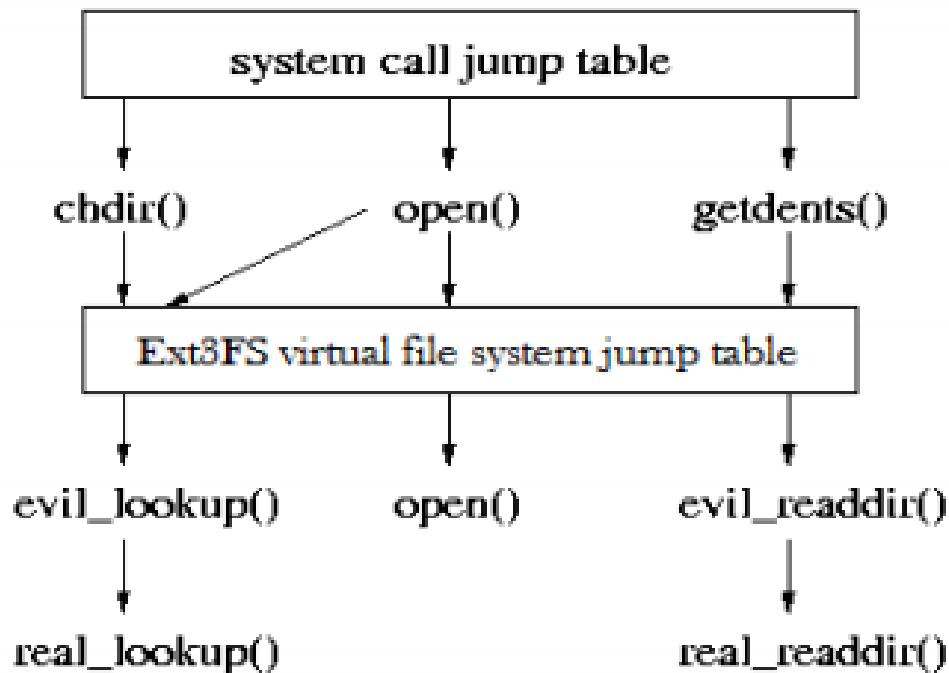
Main Contents

3.a. VFS(Virtual File System) Hacking

- VFS and /proc
 1. It is a filesystem
 2. It lives completely in kernel memory
 - All access from the userland is limited to the functionality of VFS layer provided by the kernel, namely read, write, open and alike system calls .
 - So, how the kernel can be backdoored without changing system calls?
- 

Main Contents

3.b. System Call Flow in VFS Hijacking



Main Contents

References

- <http://www.phrack.org/issues.html?issue=58&id=6>
- <http://www.phrack.org/issues.html?issue=61&id=14>
- <http://www.trapkit.de/research/rkprofiler/rkplx/rkplx.html>

Main Contents

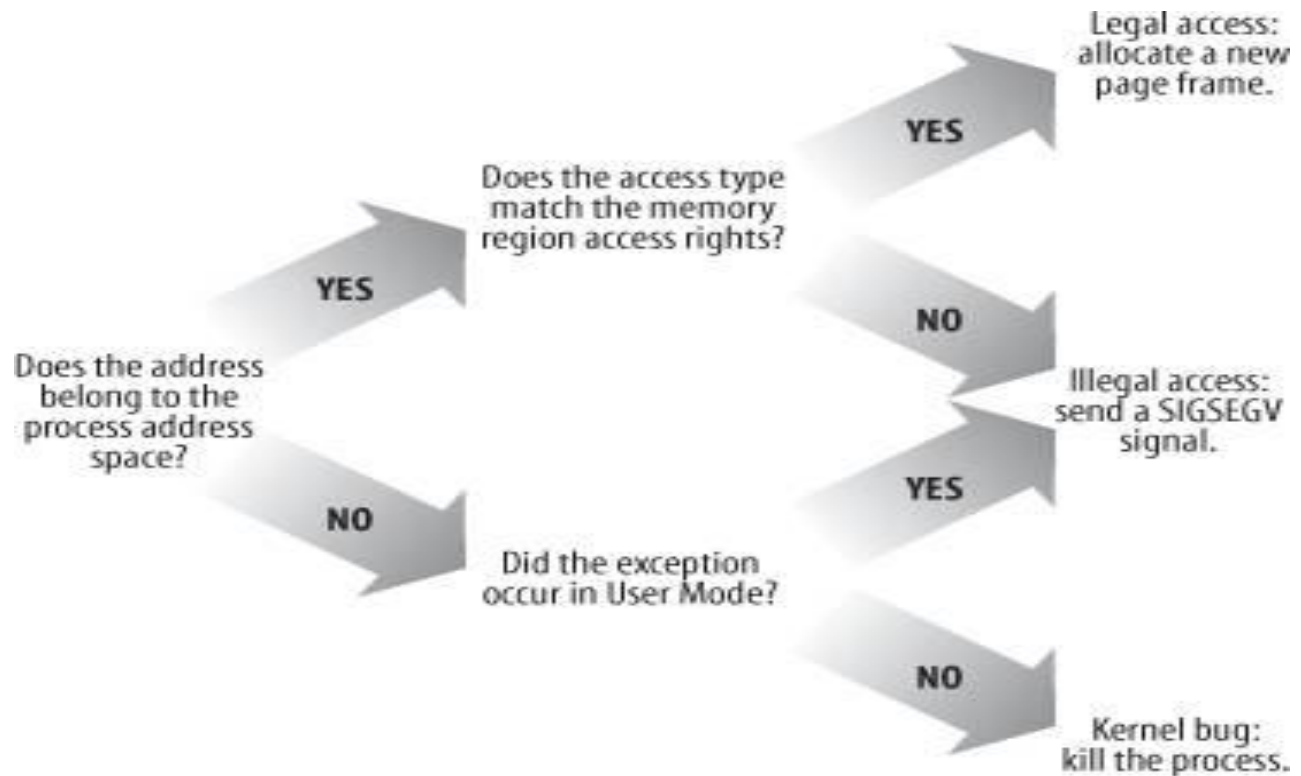
4.a Page Fault Handler Hijacking

- "A page fault exception is raised when the addressed page is not present in memory, the corresponding page table entry is null or a violation of the paging protection mechanism has occurred." (Understanding The Linux Kernel, O'reilly)

- *When?* -> The kernel attempts to address a page belonging to the process address space, but either the corresponding page frame does not exist (Demand Paging) or the kernel is trying to write a read-only page.

Main Contents

4.b. Schema on Page Fault Hijacking Process



Main Contents

4.c. References

- <http://www.phrack.org/issues.html?issue=61&id=7>
- <http://www.s0ftpj.org/bfi/dev/en/BFi12-dev-08-en>
- http://whatisthekernel.blogspot.com/2005/09/back-door-entry-getting-hold-of-kernel_01.html

Main Contents

5.a. Abusing Debug Register

"The IA-32 architecture provides extensive debugging facilities for use in debugging code and monitoring code execution and processor performance. These facilities are valuable for debugging applications software, system software, and multitasking operating systems."

- A debug exception (#DB) is generated when a memory or I/O access is made to one of these breakpoint addresses.
- There are 8 debug registers supported by the Intel processors, which control the debug operation of the processor (dr0-dr7).

Main Contents

5.b. Debug Register Address

debug registers DR0..7																																
reg.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DR0	breakpoint #0 virtual address																															
DR1	breakpoint #1 virtual address																															
DR2	breakpoint #2 virtual address																															
DR3	breakpoint #3 virtual address																															
DR4	reserved																															
DR5	reserved																															
DR6	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
DR7	LEN 3	R/W 3	LEN 2	R/W 2	LEN 1	R/W 1	LEN 0	R/W 0	T T	T B	G D	S M M I C E	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	


Main Contents

5.c. References

- <http://www.phrack.org/issues.html?issue=65&id=8>
- <http://seclists.org/dailydave/2008/q3/224>
- <http://l33ckma.tuxfamily.org/?p=174>
- [http://darkangel.antifork.org/publications/Abuso dell'Hardware nell'Attacco al Kernel di Linux.pdf](http://darkangel.antifork.org/publications/Abuso%20dell'Hardware%20nell'Attacco%20al%20Kernel%20di%20Linux.pdf)
- http://packetstormsecurity.org/files/view/57016/mood-nt_2.3.tgz

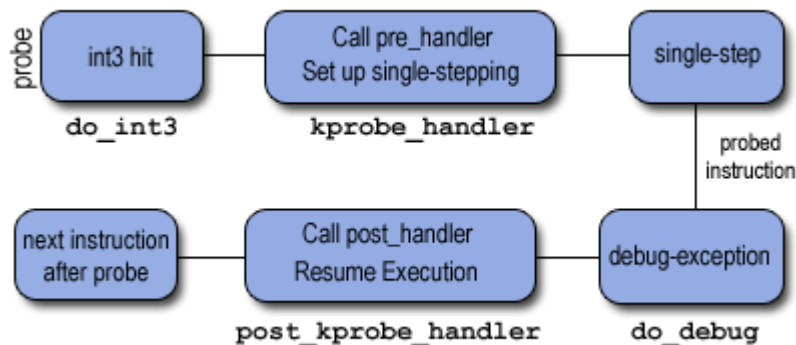
Main Contents

6.a. Kernel Instrumentation Patching

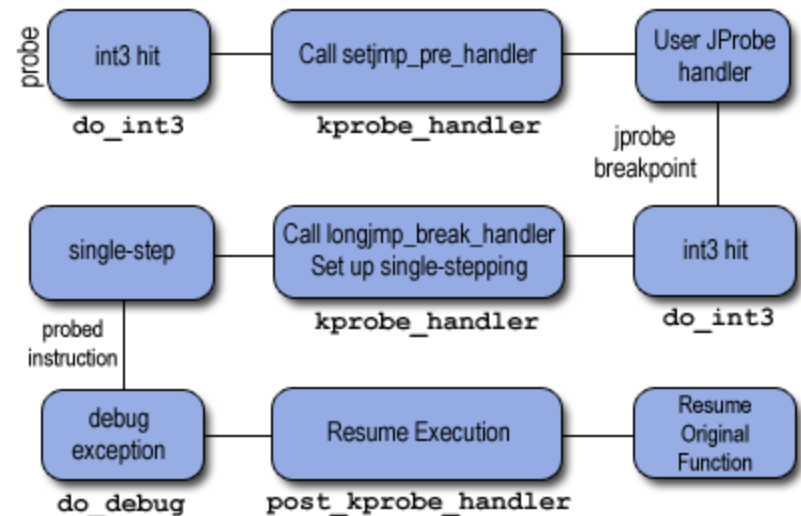
- **Kprobe** "Simple method to probe the running kernel. At a fundamental level, it requires the address of a kernel function that needs to be debugged".
 - **Jprobe** "Jprobe is another kind of probing technique, which can be used to access the target function's arguments, and thus display what was passed to the function".
 - **Kretprobes** "A return probe fires when a specified function returns ".
- 

Main Contents

6.b. Schema of Kprobe and Jprobe Execution



Kprobes Flow Execution



Jprobes Flow Execution

Main Contents

6.c. References

- <http://www.phrack.org/issues.html?issue=67&id=6>
- <http://www.chunghwan.com/systems/gaining-insight-into-the-linux-kernel-with-kprobes/>
- <http://lxr.osuosl.org/source/Documentation/kprobes.txt>

Main Contents

▶ Hiding Modules

```
- if(m->init == init_module)
    list_del(&m->list);
- kobject_unregister(&m->mkobj.kobj);
  //kobject_del for < Kernel 2.6.7
```

Main Contents

▶ **Non-LKM Rootkits**

- Via /dev/kmem
- Via /dev/mem
- How about /dev/port?

Demo

IT'S SHOW TIME!

No POC = HOAX!

Conclusion

"Any rootkit created with existing detection capabilities in mind will evade the protective measures provided by such systems. Warfare at kernel level comes down to a question of who takes over first – the rootkit or the anti-rootkit solution."

(<http://www.securelist.com/en/analysis?pubid=204792011>)



Hahaha Yeaahh...

“Subtle and insubstantial, the expert leaves no trace; divinely mysterious, he is inaudible. Thus, he is the master of his enemy's fate.”

- The Art of War, Sun Tzu