rfdslabs

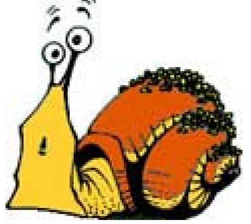
Kernel-Land Rootkits

For Linux 2.6 over x86

“Strauss” <strauss AT rfdslabs DOT com DOT br>

Agenda

- Rootkits In Brief
- Why This Presentation? (or “How It All Started...”)
- Hooking System Calls In Linux 2.6
- Meeting Our Enemy/Friend (whose side are you on, anyway?)
- Meeting Our Friend/Enemy
- <Insert Surprise Here>
- References
- Kudos



Rootkits In Brief - Foundations

- Taken from Wikipedia’s wise words:
  - “A **rootkit** is a set of software tools intended to conceal running **processes**, **files** or system data from the **operating system**... Rootkits often modify parts of the operating system or install themselves as **drivers** or **kernel modules**. ”
- In other (less wise) words:
  - “Rootkits are things that malicious hackers use when they’re root to help’em to stay this way”
  - ...and sometimes to enforce DRM [1]

Rootkits In Brief – (User/Kernel)-Land

- User-Land Rookits...
  - ...run from user-space applications and
  - rely on process infection, binary patching, library-level syscall hooking, etc
- **Kernel-Land** Rootkits...
  - ...run from inside the kernel,
  - modify kernel structures, hook system calls at the lowest level, and have little interaction with user-space programs

Rootkits In Brief – Newest Directions

- Virtualised rootkits
  - Blue Pill [2]
  - SubVirt [3]
- PCI/BIOS rootkits [4]
- God knows what’s next…

Rootkits In Brief – Win vs. Linux

- Windows
  - Rootkits: HackerDefender, HE4Hook, FU(To)...
  - Anti-Rootkits: RootkitRevealer, klister, GMER...
- **Linux**:
  - Rootkits: SucKIT, Adore, SSHEater...
  - Anti-Rootkits: St. Jude/St. Michael, rkhunter, chkrootkit...

Rootkits In Brief – Linux Kernel-Land RKs

- Popularized since THC's paper[5]
- Written as Linux Kernel Modules (LKMs)
  - Ring0
  - Extensive kernel API
  - And, obviously, kernel source-code available
- Tipically, heavily relied on system call hooks

Why This Presentation?

- Not many kernel-level rootkits for Linux 2.6
  - As the time of writing, 5 publicly available [6]
  - After 3 years since its release
- Why?
  - Amongst many possible reasons, the inability of hooking system calls
  - Previous kernels used to export the variable ‘sys_call_table[]’, 2.6 doesn’t
  - No hook, no fun (usually)

Why This Presentation? (2)

- However, there’s a trick that can be used get around this [7]
- This presentation discusses
  - This trick
  - Possible counter-measures
  - Easy writing a rootkit just by assembling publicly available pieces of code

Hooking System Calls In Linux 2.6

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

```
sys_call_table[__NR_open] = (void *) my_func_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[]’

Hooking System Calls In Linux 2.6 (2)

- The function ‘system_call’ makes a direct access to ‘sys_call_table[]’ (arch/i386/kernel/entry.S:240)

```
call *sys_call_table(,%eax,4)
```

- In x86 machine code, this translates to:

```
0xff 0x14 0x85 <addr4> <addr3> <addr2> <addr1>
```

  - Where the 4 ‘addr’ bytes form the address of ‘sys_call_table[]’

Hooking System Calls In Linux 2.6 (3)

- So, what we must do is search the code in ‘system_call’ for this fingerprint
  - Author’s note: Notice this is a much more relevant concept than a simple syscall hooking technique. It means that, given a fine set of code ‘fingerprints’, any private symbol can be unhidden in runtime starting from the main entry point or another public (or otherwise known) symbol deeper in the control-flow tree. It’s an invalidation to Information Hiding.

Hooking System Calls In Linux 2.6 (4)

- Problem: ‘system_call’ is not exported too
  - It’s not, but we can discover where it is!
- ‘system_call’ is set as a trap gate of the system (arch/i386/kernel/traps.c:1195):

```
set_system_gate(SYSCALL_VECTOR,&system_call);
```

- In x86, this means that its address is stored inside the Interrupt Descriptor Table (IDT)
- The IDT location can be known via the IDT register (IDTR)
- And the IDTR, finally, can be retrieved by the SIDT (Store IDT) instruction

rfdslabs

Hooking System Calls In Linux 2.6 (5)

- Putting it all together:
  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) We should have the address of ‘sys_call_table[]’ by now, have fun!

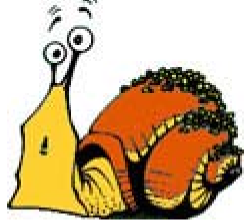
Meeting Our Enemy/Friend

Captain Hook

Meeting Our Enemy/Friend (2)

- Captain Hook exemplifies:
  - The use of the SIDT technique
  - How rootkits can be written with little effort
- Captain Hook is an **EXAMPLE** rootkit
- Contains only the minimum to **DEMONSTRATE** some concepts
- **NOT** supposed to be usable in the wild

rfdslabs



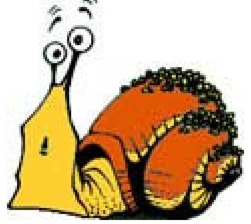
Meeting Our Enemy/Friend (3)

- Features:
  - Hides the file “capnhook.ko” (itself)
  - Runs a UDP server inside the kernel space that can receive and execute programs
- The techique to hide files can be used for other puposes, eg. hiding processes
- The remote execution feature is a way to make the design simple without losing power
  - You can make the computer do whatever you want if you can write the proper program
  - If you’d want a shell, for example, you can write 5KB bindshell and make your rootkit execute it

Meeting Our Enemy/Friend (4)

- Getting the System Call Table (SCT) [7]:

```
IDTR idtr; interrupt_descriptor *IDT, *sytem_gate;
asm("sidt %0" : "=m" (idtr));
IDT = (interrupt_descriptor *) idtr.base_addr;
system_gate = &IDT[0x80];
sys_call_asm = (char *) ((system_gate->off2 << 16) | system_gate->off1);
for (i = 0; i < 100; i++) {
    if (sys_call_asm[i] == (unsigned char) 0xff &&
    sys_call_asm[i+1] == (unsigned char) 0x14 &&
    sys_call_asm[i+2] == (unsigned char) 0x85)
        *guessed_sct = (unsigned int *) *(unsigned int *) &sys_call_asm[i+3];
}
```

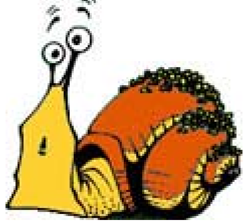


Meeting Our Enemy/Friend (5)

- Hooking a system call:

```
capnhook_get_sct(&capnhook_sct);
old_sys_getdents64 = (void *) capnhook_sct[__NR_getdents64];
capnhook_sct[__NR_getdents64] = (unsigned int) capnhook_sys_getdents64;
```

- The ‘getdents’ and ‘getdents64’ system call get directory entries
- They’re used by the program ls to list files
- We hook it to hide the file ‘capnhook.ko’
- The implementation for our hook is too big to fit here and its beyond the scope of this presentation
  - It was taken from [8]



Meeting Our Enemy/Friend (6)

- The UDP server code is also too big to list here
  - It was taken from [9]
- The UDP server listens on port 2323
- Slightly modified so that the initialization function receives a callback function
- On the receipt of a packet, the callback function is called with the data and the length of the data passed as arguments

```
static void req_handler(unsigned char *, unsigned int);
...
    capnhook_udp_init(req_handler);
```

Meeting Our Enemy/Friend (7)

- The protocol is minimal
  - The UDP server waits for a message starting with "EXECUTE"
  - Following the command must be the size in bytes of the executable to be received
  - Captain Hook allocates the necessary space in memory and assembles the upcoming packets
  - Finally, a file named 'capnhook_xctbl' is created in the root directory, written with the executable code, executed, and ultimately removed from the filesystem

rfdslabs

Meeting Our Enemy/Friend (8)

- The code to write to the file was taken from [10]
- Creating, writing and removing: the plan is to use sytem calls (creat, write, and unlink)
  - Problem 1: system calls expect pointers from the user space, but we're running from kernel space ☹
    - Some food for thought: suppose we write our values to some random address below 0xC0000000 to trick the kernel. Which process would own the address space we'd be writing to?

Meeting Our Enemy/Friend (9)

- The solution is to “fix” the address space ☺
- We use the function ‘set_fs’ to select which data segment we want to use, KERNEL_DS or USER_DS

```
mm_segment_t old_fs = get_fs();
set_fs(KERNEL_DS);
...
/* Making system calls here */
...
set_fs(old_fs);
```

Meeting Our Enemy/Friend (10)

- Problem 2: ‘sys_write’ is an exported system call, but ‘sys_creat’ and ‘sys_unlink’ are not! ☹
- Solution: Hey! We have the SCT in our hands, haven’t we? ☺

```
capnhook_sys_creat = (void *) capnhook_sct[__NR_creat];
...
fd = capnhook_sys_creat("/capnhook_xctbl", 0777);
```

rfdslabs

Meeting Our Enemy/Friend (11)

- Results
  - ‘capnhook.ko’ has around 6KB and lets you do anything with the victim’s computer
  - The Captain Hook package comes with a client program

```
./captain-client <host> <port> <xctbl>
```

  - It also comes with a test executable that just creates an empty file named ‘huhuhu’ in the root directory

Meeting Our Friend / Enemy

Tick-Tock, The Croc

Meeting Our Friend/Enemy (2)

- Tick-Tock was written as an attempt to offer some resistance to the SIDT trick
- It’s not a stand-alone anti-rootkit solution, but rather another protection to be added to existing solutions in the (lost?) cause of defeating rootkits
- The idea: every module should be checked before its insertion to see if it contains the SIDT instruction
- If it does, than block it from being inserted into the kernel
  - Frankly speaking, it’s hard to imagine a legitimate use of the SIDT instruction, except for operating system core code

Meeting Our Friend/Enemy (3)

- I can easily spot two flaws in this approach
  - SIDT can be used in ring3
  - Can you spot the second? ;)
- But still Tick-Tock can stop one of the 5 publicly available rootkits for Linux 2.6
  - And Captain Hook too!

Meeting Our Friend/Enemy (4)

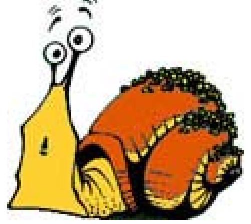
- Tick-Tock uses a minimalized version of bastard’s libdisasm [11]
  - All I wanted was something to return the length of a given instruction, yet it’s very complex
- It’s loaded as an LKM and hooks the ‘init_module’ system call
- Upon attempt of a module insertion, it search every executable section of the type SHT_PROGBITS for an occurrence of the SIDT instruction
  - Author’s Note: another wide concept. Tick-Tock could be used to block other kinds of instructions, eg. x87 FPU instructions

rfdslabs

Surprise!

3j33t t3qn33kq5 – Raising The Bar For Rootkit Detection

rfdslabs



Suprise! (2)

- No source code will be available
  - Exposing the concepts is disclosure enough
- We have 1 case study
- Counter-measures will be proposed for discussion when possible
- On to the show...

Kansas City Shuffle - Bypassing St. Michael

"It's a blindfold kickback type of a game called the Kansas City Shuffle. **When the suits look left they fall right into the Kansas City Shuffle**..."

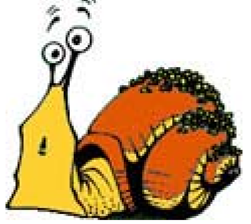
(J Ralph – Kansas City Shuffle)

Kansas City Shuffle - Bypassing St. Michael (2)

- St. Michael does a series of integrity checks to avoid hooks of system calls
  - Saves the addresses of every syscall
  - Saves the checksums of the first 31 bytes of every syscall’s code
  - Saves the checksums of these data themselves
- Now you can’t change the addresses in the system call table
- Also can’t patch the system calls with jmp’s to your hooks

Kansas City Shuffle - Bypassing St. Michael (3)

- But there’s a few things you can do (and another few that St. Michael can do too)
- Essentially, Kansas City Shufflin’ consists on making the kernel use another (modified) copy of some interesting data while the defender performs checks on the old one



Kansas City Shuffle - Bypassing St. Michael (4)

- Trick 1: Copy the system call table and patch the proper bytes in ‘system_call’ with the new address
  - This can be avoided by having St. Michael making checksums of ‘system_call’ code too
- Trick 2: Copy ‘system_call’ code, apply Trick 1 on it, and modified the 0x80th ID in the IDT with the new address
  - This can be avoided by having St. Michael storing the address of ‘system_call’ too

Kansas City Shuffle - Bypassing St. Michael (5)

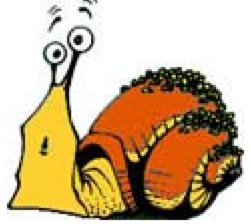
- Trick 3 (the original Kansas City Shuffle): make a copy of the IDT, apply the Trick 2 on it, and load it on the CPU with the LIDT instruction
  - This can be avoided by having St. Michael storing the address of the IDT and always SIDT'ing to check it before applying the other checks
  - Or by using Tick-Tock ;)
    - SIDT can be run on ring3, but LIDT is ring0 only

References

- [1] Sony Rootkit: http://blogs.technet.com/markrussinovich/archive/2005/10/31/sony-rootkits-and-digital-rights-management-gone-too-far.aspx
- [2] Blue Pill: www.blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf
- [3] SubVirt: www.eecs.umich.edu/virtual/papers/king06.pdf
- [4] PCI Rootkit: http://www.ngssoftware.com/research/papers/Implementing_And_Detecting_A_PCI_Rootkit.pdf
- [5] THC's Paper: http://packetstormsecurity.org/docs/hack/LKM_HACKING.html
- [6] Rootkits for download: http://packetstormsecurity.org/UNIX/penetration/rootkits/

References (2)

- [7] SIDT trick (SuckIT article): http://www.phrack.org/archives/58/p58-0x07
- [8] Hook for getdents64 (and more): http://www.s0ftpj.org/bfi/dev/BFi13-dev-22
- [9] UDP Server: http://kernelnewbies.org/Simple_UDP_Server
- [10] File writing code (and more): http://www.linuxjournal.com/article/8110
- [11] libdisasm: http://bastard.sourceforge.net/libdisasm.html



Kudos

- To H2HC III organization staff (special thx for flying me here ☺)
- To my friends @ rfdslabs and gotfault
- To BSDaemon, for some ideas, insights and his Defcon presentation
- To the co-founder of the Golden Monkey Gods Appreciation And Praise Church, for co-founding the Golden Monkey Gods Appreciation And Praise Church
  - fr33(vUg0);
  - 53gm3nt4t1on f4ul7
- To sandimas, for discussing hacking with me uncountable times

rfdslabs

Questions?

rfdslabs

Kernel-Land Rootkits

For Linux 2.6 over x86

“Strauss” <strauss AT rfdslabs DOT com DOT br>

http://strauss.rfdslabs.com.br