Defcon CTF quals 2008: rev400
Write-up by Gaël from Routards team

We are provided with an unknown binary. We are informed that it is running on Kenshoto's server. The goal is to understand what this binary does, and use this knowledge on the Kenshoto FreeBSD 6.x server to get a secret key.

Trying to run the binary on the command line of a FreeBSD machine doesn't work as the file is not recognized as an executable. By looking at the strings and functions of the binary, we conclude that the binary is a kernel module:

- The "module_register_init" kernel function is called ;
- We have strings with "mod" in it, like "set_modmetadata_set" ;
- The uprintf function is used, which is a kernel function ;
- There are data structures associated to the strings "captain" and "captain_kmod", which look like the meta data used to tell the kernel what are the module name, version, and dependencies on other modules when registering it (as declared in "sys/module.h" in the kernel sources).

Trying to load the module into memory with "kldload –v captain.ko" fails because of the "captain_kmod" module not being available. Thus, we have only one part of the program, the other part being in the "captain_kmod" module. It is probably this module that is holding the key… we assume it is running on the Kenshoto server.

Now let's analyze the beast!

Loading the binary under IDA Pro shows recurrent use of a basic code obfuscation technique: the x86 assembly instruction "\xEB\xFF" aka "JMP +1" is inserted at multiple places in the code, which confuses IDA and make some functions look like they are returning early with a "ret" instruction while it is not actually the case. We get rid of them by manually by un-defining the JMP instruction (shortcut "U") and converting to code (shortcut "C") the code starting from the "\xFF" byte. We can also safely convert the "\xEB" byte to a NOP instruction.

Okay, now let's find out about what this module really does, by disassembling all functions and trying to understand the logic of the module.

At first look: syscall hooks.

Most of the code is used to hook several system calls. The part that is installing the hook modifies the system calls table "sysent" to hijack some system calls pointers with the address of the related hooking function in the module:

```
.text:00001055                  mov     dword ptr ds:sysent+10h, offset do_sysexit
.text:0000105F                  mov     dword ptr ds:sysent+28h, offset do_read
.text:00001069                  mov     dword ptr ds:sysent+34h, offset do_write
.text:00001073                  mov     dword ptr ds:sysent+40h, offset do_open
.text:0000107D                  mov     dword ptr sysent+4Ch, offset do_close
.text:00001087                  mov     dword ptr sysent+0A0h, offset do_fchdir
.text:00001091                  mov     dword ptr sysent+0ACh, offset do_mknod
.text:0000109B                  mov     dword ptr sysent+0B8h, offset do_chmod
.text:000010A5                  mov     dword ptr sysent+0C4h, offset do_chown
.text:000010AF                  mov     dword ptr sysent+148h, offset do_recvmsg
.text:000010B9                  mov     dword ptr sysent+154h, offset do_sendmsg
.text:000010C3                  mov     dword ptr sysent+160h, offset do_recvfrom
.text:000010CD                  mov     dword ptr sysent+16Ch, offset do_accept
.text:000010D7                  mov     dword ptr sysent+178h, offset do_getpeername
```

```
.text:000010E1                  mov     dword ptr sysent+184h, offset do_getsockname
```

Note that I changed the functions name in order to be more readable. The name of the system calls that are hooked can be found from the offset in the "sysent" table, but also more easily by looking at the code that reverses the hooks when the module is unloaded, which is located at .text:00001185.

Let's take a look at what does a hooked function. For instance, we can look at the do_read function located at .text:00000B98. Basically, what such function does is the following:

1) Check some numerical conditions on the syscall parameters;
2) If and only if these conditions are met, modify the global variable located at .bss:00002370 with some algorithm that changes depending in which the syscall function we are;
3) Call the "real" syscall function in the kernel.

Actually, all this stuff seems to be a false lead! We will not care about it anymore, since we do not see from these hooked functions how all this global variable tweaking could ever give us a key. And we need a key.

At second look: IDT "int 0x80" hijack.

What we do see as an interesting lead is this "uprintf" kernel function, which is used at .text:00000ADA to print a string pointed to by an external variable named "currentpid". The "uprintf" function prints a string to the controlling tty of the current process. Since the "currentpid" variable doesn't seem to be a standard kernel variable, we can imagine that it could very well be contained in the missing "captain_kmod" module, and would probably hold the key we are looking for.

Going backwards, we find that the function doing the "uprintf" is only called by one function, located at .text:00000AE4, depending on some conditions. This function is itself called only by a short wrapper function located at .text:00000A15. This function is itself never called… but its address is used at offset .text:00001138 in the following piece of code.

We see that the Interrupt Descriptor Table (IDT) of the system is modified in order to replace the "int 0x80" handler with the address of the function at .text:00000A15. To understand this code, one should known that the size of an entry in the IDT is 8 bytes and that the address of the interruption handler in an IDT entry is split in two, the low word being stored at offset 0 and the high word at offset 6 of the entry. (Note: for more information about the IDT one could read http://www.phrack.org/issues.html?issue=59&id=4).

```
.text:00001104 sidt    fword ptr [ebp-14h] ; gets IDT information structure
.text:0000110E mov     ecx, [ebp-12h]      ; gets IDT base address
.text:00001111 lea     edx, [ecx+400h]     ; gets address of "int 0x80" entry in the
IDT (0x400/8 = 0x80)
.text:00001117 mov     ds:idt_hijackaddr, edx   ; saves this address for future use
.text:00001123 movzx   eax, word ptr [edx+6]
.text:00001127 shl     eax, 10h
.text:0000112A movzx   edx, word ptr [ecx+400h]
.text:00001131 or      eax, edx             ; eax = address of "int 0x80" handler
.text:00001133 mov     ds:dword_2364, eax
.text:00001138 mov     edx, offset int80_hijack_function
.text:0000113D lea     eax, [ebp-1Ch]
.text:00001140 call  loc_1008   ; puts LOW(edx) in [eax], and HIGH(edx) in [eax+6],
which will hijack the "int 0x80" IDT entry by putting the address contained in edx
instead of the original handler. eax points to an IDT entry structure located on
the stack.
.text:00001145 mov     edx, ds:idt_hijackaddr
.text:0000114B movzx   ax, byte ptr [edx+2]
```

```
.text:00001150 mov     [ebp-1Ah], ax    ; complete the new IDT structure on the
stack with the original data located at offsets 2 and 5 (flags, segment
selector...)
.text:00001154 mov     al, [edx+5]
.text:00001157 mov     [ebp-17h], al
.text:0000115A lea     eax, [ebp-1Ch]
.text:0000115D push    eax             ; pointer to local IDT entry structure
.text:0000115E push    edx             ; pointer to "int 0x80" IDT entry
.text:0000115F call      near ptr write_64_bits ; actually performs the hijack by
writing the 8 bytes of the local structure in the IDT
```

Well, that's very good! So far we are pleased of our work, since we now understand the logic of the module, and thus the way to the key.
Or do we? We know we will have to issue an "int 0x80" instruction to get to the interesting part of the code, but there is actually one last problem to overcome: the uprintf function which prints to key to our controlling tty is called only if certain conditions are met. We need to understand these conditions, by analyzing the code.

```
      [...]        ; previously the first argument of the stack is put into ebx
.text:00000AF2 mov     eax, large fs:0
.text:00000AF8 mov     esi, [eax]
.text:00000B00 mov     ecx, [esi+3Ch]
.text:00000B03 mov     eax, ecx
.text:00000B05 and     eax, 0FFFF0000h
.text:00000B0A cmp     eax, 5BE90000h ; <-- high word of [fs:3c] must be 5BE9
.text:00000B0F jz      short loc_B38
[…]
.text:00000B38 loc_B38:
.text:00000B38 cmp     ebx, 1  ;      <-- arg0 must be 1
.text:00000B3B jnz     short near ptr unk_B11
.text:00000B43 call    print_fmt_string_to_proc_tty  ; prints the key to tty :-)
```

The key is printed only if a flag located at address fs:3c has value 5BE9 in its high word, and if the first argument to the function is 0x01.

Since the "int 0x80" instruction is used by the FreeBSD system to allow userland (ring 3) code to issue system calls, the first argument to the function would be the syscall number. Syscall 0x01 is the "exit" syscall, which exits the current process. Getting the key is thus only a matter of quitting… after having correctly set up the fs:3c flag. To perform this, we see that this flag is updated at each system call:

```
.text:00000B17 movzx   edx, cx          ; dx = low word of [fs:3c]
.text:00000B1A and      ebx, 7         ; ebx = the 3 low bits of the syscall number
pushed on the stack
.text:00000B1D and     ecx, 0FFFF0000h ; ecx = high word of [fs:3c]
.text:00000B23 shl      ebx, 10h         ; the 3 low bits of the syscall number of
put into the high word
.text:00000B26 lea     eax, ds:0[ecx*4]   ; shifts left ecx by 2 bits (eax = ecx <<
2)
.text:00000B2D xor     eax, ebx           ; no comment
.text:00000B2F or       edx, eax         ; edx = low word | ( high word<<2 ^ the 3
low bits )
.text:00000B31 mov     [esi+3Ch], edx   ; updates flag at [fs:3c]
```

At each system call performed, int 0x80 is called and the high word of the flag at [fs:3c] is updated by shifting it by two bits on the left and XORing it with the last 3 bits of the syscall number.

Thus, to setup the fs:3c flag to the correct value (5BE9) we can issue several system calls sequentially, with different syscall numbers. As 5BE9 is 0101101111101001 in binary, to construct it in two bits chunks, we could send the following syscalls:

```
Bits: 01 01 10 11 11 10 10 01 = 5BE9
Syscall numbers sequence: 1 1 2 3 3 2 2 1
```

However, some syscall numbers cannot be used, for instance 1 will terminate the program, and 2 will fork. We can look in the “syscall.h” include file to get the list of syscall numbers. We must use only innocuous syscall numbers, and that’s why the good folks at Kenshoto allowed us to use one additional bit and XOR it with the previous one (even though it’s not the only way).

Then, another possibility to generate the value 5BE9 without using the “fork” and “exit” syscalls would be:

```
Sycall bits: 100 101 11 111 11 11 111 101
Syscall numbers: 4 5 3 7 3 3 7 5
```

Now all that is left is to write a small assembly program which issues these system calls sequentially by pushing the value of the syscall on the stack and calling the 0x80 interruption (“move $0xNN, %eax, push %eax, int $0x80”). At this time the value of the flag at [fs:3c] is now 5BE9. We can issue a last call to syscall number 1 (exit) and the module will print the key on the current tty.

So we connect on a shell on the Kenshoto server, get a tty for the shell (mandatory step for uprintf to work), run the code… And, almost magically, the secret key appears.