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.

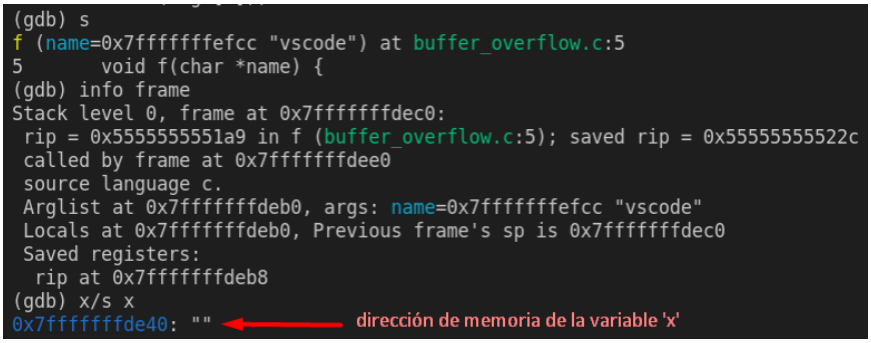

Figura 1: Ejecución del programa `buffer_overflow` usando gdb

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:

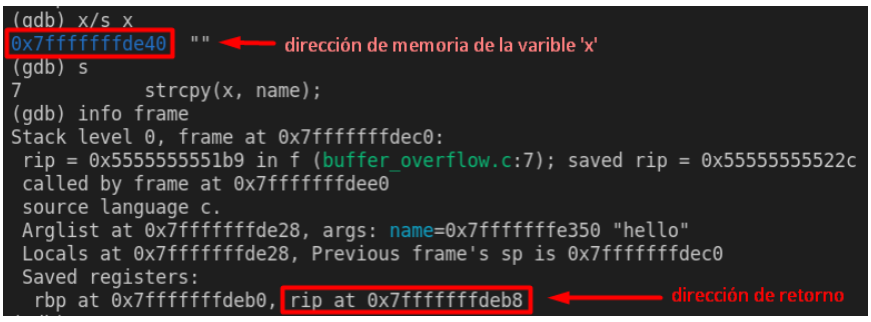

Figura 2: Ejecución del programa `buffer_overflow` usando gdb

En este caso, tenemos que la dirección de memoria del código retorno corresponde al hexadecimal `0x7fffffffdeb8`, mientras que el código de retorno corresponde a `0x5555555551b9`.

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 `120btyes`:

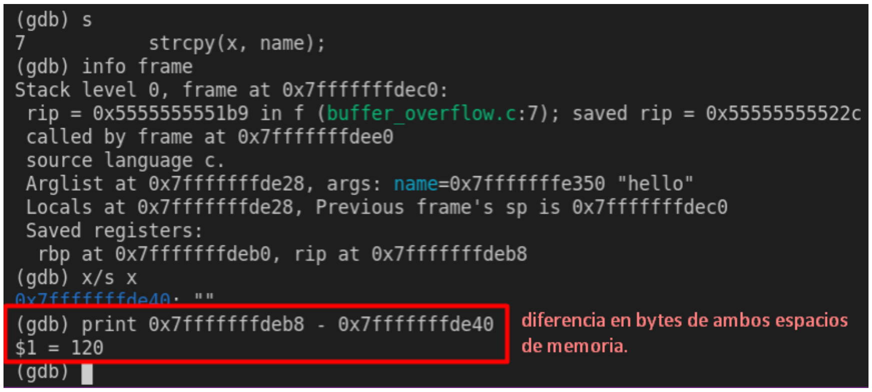

Figura 3: Calculo en bytes de la diferencia entre ambas direcciones calculadas

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 vilación de segmento. Debido a que la distancia entre esta dirección y el buffer es de `120btyes`, un argumento de `121btyes` alterará la dirección de retorno con el último byte del argumento:

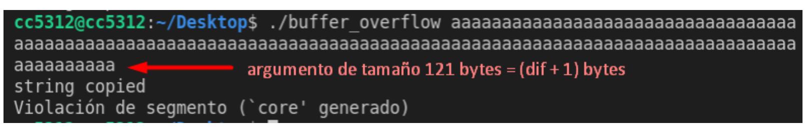

Figura 4: Ejecución del programa con un string de 121bytes

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 `121btyes`, el buffer solo registra `100btyes` (`á'` repetido 100 veces).

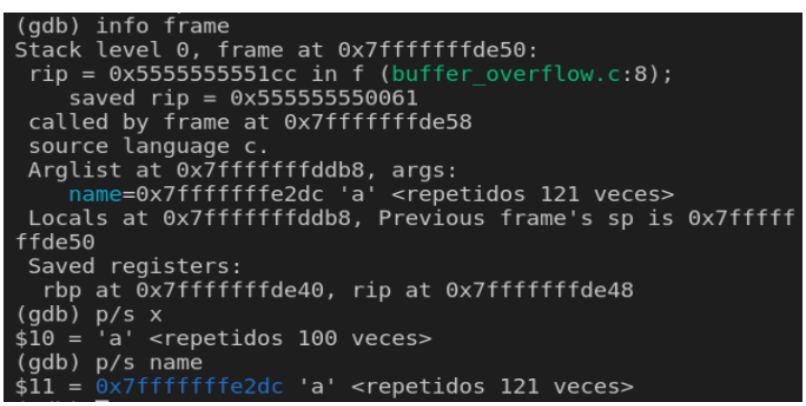

Figura 5: Estado de la pila y de la variable `name` después de copiar

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 sobre-escritura:

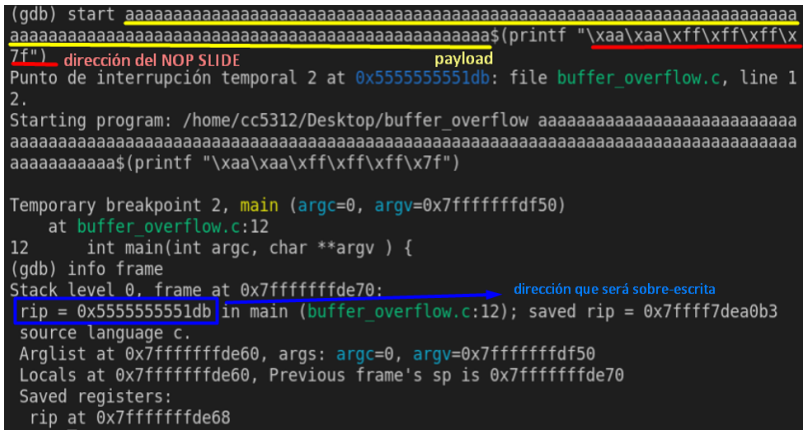

Figura 6: Ejecución del programa con NOP SLIDE: `0x7fffffffaaaa`

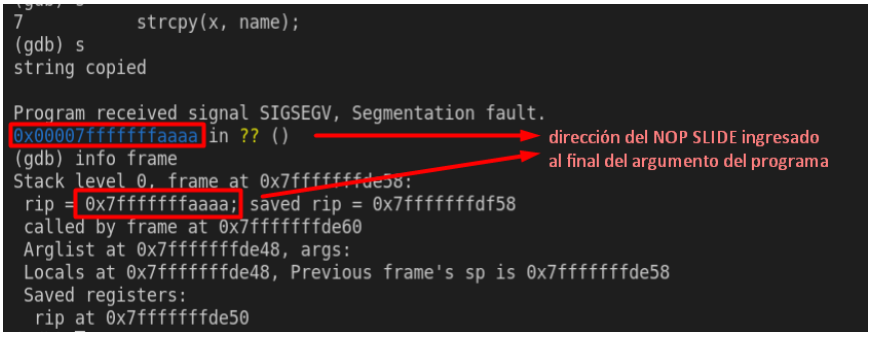

Figura 7: Visualización de la sobre-escritura de la dirección de retorno del programa

Inyectar 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 sobreescribir 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`:

```
start $(printf "\x48\x31\xf6\x56\x48\xbf\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x57
\x54\x5f\x6a\x3b\x58\x99\x0f\x05")aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa$(printf "\xd0\xdd\xff
\xff\xff\x7f")
```

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.

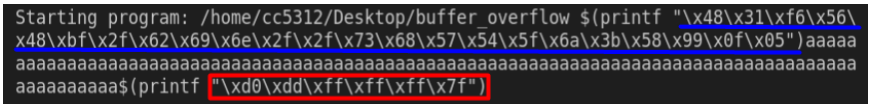

Figura 8: Inyección del shellcode

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:

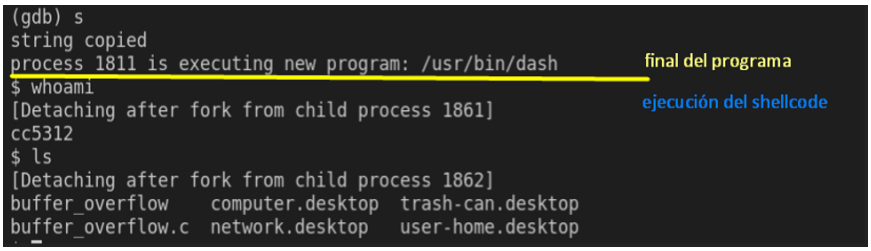

Figura 9: Ejecución del shellcode inyectado

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`).