Laboratorio 3
Bypass completo: NX + ASLR + SSP

Pentesting y **Hacking Ético**

Explotación de Binarios

Héctor Marco
Universitat Politècnica de València

Curso 2024-25

En este laboratorio se culmina la cadena de defensas vistas hasta ahora.

Introducción y objetivos

En los temas anteriores aprendiste a:

- Saltarte **NX** con *return-to-libc*.
- Superar **ASLR** mediante *trial-and-test* en 32 bits.

Ahora añadimos la protección **Stack Canaries (SSP)** y verás:

1. La diferencia práctica entre un *segmentation fault* y el mensaje `*** stack smashing detected ***`.
2. Cómo escribir un **servidor echo** (`fork()` por conexión) vulnerable a *stack buffer overflow* con SSP activo.
3. Cómo obtener el canario mediante **byte-for-byte** en un *forking* server.
4. Con el canario conocido, cómo **brute-forzar** la `libc` (máx. 2^8 intentos) para ejecutar comandos usando `system()`.

Al finalizar habrás derrotado las tres protecciones clásicas, **NX + ASLR + SSP**, en sistemas de 32-bits contra un *forking server*.

1. Segmentation fault vs `stack smashing detected`

1.1 Programa mínimo

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

void vuln(void) {
    char buf[16];
    gets(buf);
    puts("Fin.");
}

int main(void) {
    vuln();
    return 0;
}
```

Listing 1: `crash.c` – stack overflow sencillo

1. **Compila sin canarios** para observar el *segfault*:

```
$ gcc -m32 -fno-stack-protector crash.c -o nocanary
$ python3 -c 'print("a" * 40)' | ./nocanary
Fin.
Segmentation fault
```

2. **Compila con canarios** (`-fstack-protector`) y provoca el desbordamiento :

```
$ gcc -m32 -fstack-protector crash.c -o canary
$ python3 -c 'print("a" * 40)' | ./canary
Fin.
*** stack smashing detected ***: terminated
Aborted
```

   a) Indica el mensaje exacto que obtienes en ambos casos.

   b) Indica en que función se produce el stack buffer overflow.

   c) Explica por qué se obtiene el mensaje “Fin.” incluso usando `-fstack-protector`.

2. Mostrando el Canario

Compila y ejecuta el siguiente mini-programa que lee directamente el *stack cookie* almacenado por el kernel en el segmento `gs`:

```c
#include <stdio.h>
#include <stdint.h>

static inline uintptr_t stack_cookie(void)
{
    uintptr_t v;
    __asm__("mov %%gs:0x14,%0" : "=r"(v));
    return v;
}

int main(void)
{
    printf("%#lx\n", (unsigned long) stack_cookie());
    return 0;
}
```

Listing 2: show_canary_32.c – imprime el canario global en x86

1. Compila en 32 bits:

   ```
   $ gcc -m32 show_canary_32.c -o show_canary_32
   ```

2. Ejecuta **cinco veces** y anote los valores.

3. Indica:

   - Los cinco canarios obtenidos.
   - Distribución/GNU Linux exacta (comando `uname -a`).
   - ¿Alguno de los bytes se mantiene siempre en `0x00`? Comenta si tu distribución de Linux fija el último byte del canario a cero.

4. Compila ahora sin canario (`-fno-stack-protector`) y vuelve a ejecutarlo 5 veces.

   - Anota los 5 valores obtenidos.
   - Comenta si los valores tienen pinta de ser el canario y en que te apoyas para decirlo.

5. Finalmente, crea el programa `show_canar_64.c` que muestre el canario de un programa compilado en 64 bits. Pista, no hay que usar el registro `gs` sino el `fs`:

   - Proporciona el programa completo.
   - Ejecútalo 5 veces y anota los valores.
   - Observa y comenta si hay algún byte que se mantenga siempre a `0x00`.

3. Servidor *echo* vulnerable

3.1 Código base (`echosrv.c`)

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define PORT 9999

unsigned char global[1024];

void vulnerable(unsigned char *msg, int len){
    char buf[64];
    memcpy(buf, msg, len);
    printf("Recibido %d bytes: ", len);
    for (size_t i = 0; i < len; i++) {
            printf("%02x ", msg[i]);
    }
    printf("\n");
}

void handle(int client){
    ssize_t r = read(client, global, sizeof(global));
    vulnerable(global, r);
    write(client, global, r);
    close(client);
    puts("Conexión cerrada.");
}

int main(void){
    int s = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr = {.sin_family=AF_INET,
                               .sin_port=htons(PORT),
                               .sin_addr.s_addr=INADDR_ANY};
    bind(s, (struct sockaddr*)&addr, sizeof(addr));
    listen(s, 1);

    printf("==== Echo Server in port %d ====\n", PORT);
    while (1) {
        int c = accept(s, NULL, NULL);
        if (!fork()) {      // proceso hijo
            close(s);
            handle(c);
            _exit(0);
        }
        close(c);           // padre
    }
}
```

Listing 3: Servidor *echo* con un stack buffer overflow

- Compílalo para 32 bits con canario. Al ejecutarlo deberías ver el mensaje del `printf()`.

  ```
  $ gcc -m32 -fstack-protector echosrv.c -o echosrv
  ```

- Ejecuta el servidor, deberías ver el mensaje:

  ```
  ==== Echo Server in port 9999 ====
  ```

- Estudia el código para entender donde y cómo se produce el stack buffer overflow ya que en la siguiente parte vamos a exfiltrar el canario usando la técnica `byte-for-byte`.

- Envía un payload lo suficientemente largo para que veas en el servidor el mensaje:

  ```
  *** stack smashing detected ***
  ```

4. Filtrado byte-a-byte del canario

Como se ha visto en las sesiones de teoría, el modelo ***forking*** mantiene *el mismo* canario de referencia en todos los hijos de un proceso padre. Esto permite realizar ataques muy eficientes si el atacante tiene la posibilidad de realizar overflows con granularidad de byte. En este caso, el atacante podría:

1. Detectar donde empieza el canario (el offset no es necesariamente el tamaño del buffer).
2. Sobrescribir *sólo* el primer byte después del **NULL final** del canario (podemos evitarnos averiguar el primer byte en algunas distros).
3. Repetirlo hasta detectar cuando el proceso **no** ha muerto $\Rightarrow$ el byte es correcto.
4. Aplicando esto a los tres bytes del canario serían $\Rightarrow 256 \times 3 = 768$ intentos máx.
5. Siguiendo la estrategia anterior, crea un script que obtenga el valor del canario del `echosrv` utilizando la estrategia `byte-for-byte`.
6. Guarda el valor obtenido del canario para la siguiente fase.

5. Brute force de `system`

Hasta este punto hemos visto como bypassear el `NX` y `SSP` en un forking server compilado para `i386 (32-bits)`, ahora nos faltaría el `ASLR`.

Dado que ya conocemos el canario, podemos realizar un ataque de brute-force para adivinar donde está la llamada a `system` de la libc. Una aproximación inicial sería construir un payload que ejecute un comando en el servidor. Puedes reutilizar la parte 3 del lab2.

6. Exploit para 64-bits (Avanzado)

Compila el servidor de *echo* para 64-bits y adapta el exploit generado en el punto 4 para que bypassee el `NX` y `SSP` pero NO el `ASLR`.

1. Crea un script que permita ejecutar un comando (el que quieras) llamando a la `system()` de la `libc`. Puedes partir del script que tenias del punto 4. Recuerda que el `ABI` es diferente en 64-bits y que los parámetros se pasan por registro

Si se te queda corto, puedes intentar crear el comando utilizando `ROP`

1. Crea un script que permita ejecutar un comando usando ROP para construir el string del comando en una zona donde puedas escribir y luego usa la dirección para saltar a `system()`. Primero busca Gadgets en el propio ejecutable (recuerda que en i386 y x86_64 se permite saltos a instrucciones no alineadas).
2. Crea un script que use `ROP` para hacer una syscall en vez saltar a `system()` o similares. Primero busca Gadgets en el propio ejecutable (recuerda que en i386 y x86_64 se permite saltos a instrucciones no alineadas).
3. Si aún te quedan fuerzas, averigua como podrías bypassear el ASLR en 64-bits y proporciona el exploit completo bypasseando todo, el `NX`, `SSP` y `ASLR`.

8. Para entregar (PoliformaT)

1. PDF con:
   - Resultados y pasos de todas las partes.
   - Código del servidor y de los scripts, así como los comandos de compilación y cualquier otro que sea necesario para reproducir los resultados.

¡Fin del Laboratorio 3!

Pruébalo, rómpelo y *happy hacking.*