RETO 8

Manipulación de memoria y registros - Programa bufOver4.c

ANÁLISIS DE LA VULNERABILIDAD

El programa utiliza la función insegura gets(buf), lo que permite introducir más de 80 bytes desde la entrada estándar sin control de tamaño, sobrescribiendo regiones adyacentes en la pila, como la variable cookie e incluso la dirección de retorno de la función main.

El objetivo es provocar que la variable cookie tome el valor exacto 0x000d0a00 o, alternativamente, redirigir el flujo de ejecución del programa manipulando la dirección de retorno para que se ejecute directamente el bloque de código que imprime el mensaje "you win!".

ESTRATEGIA SEGUIDA

1. **Limitaciones del enfoque directo**
   El objetivo inicial era modificar directamente el valor de la variable cookie a 0x000d0a00 utilizando la vulnerabilidad del buffer overflow. Sin embargo, este enfoque resulta inviable porque la función gets() detiene la lectura al encontrar el byte 0x0a (salto de línea), impidiendo introducir completamente ese valor.

2. **Ataque “Smash the Stack”**
   Ante esta limitación, se opta por una estrategia alternativa: sobrescribir la dirección de retorno de la función main. De este modo, se redirige el flujo de ejecución hacia una parte del código ya existente que imprime directamente el mensaje deseado: "you win!". Esta técnica, conocida como “Smash the Stack”, consiste en provocar un desbordamiento del búfer que permita alcanzar y modificar la dirección de retorno almacenada en la pila. En este caso, se identifica que la instrucción que imprime "you win!" se encuentra en la dirección 0x401183, y ese será el destino que se inyectará como nueva dirección de retorno.

3. **Preparación del entorno y compilación del binario**
   Para poder realizar el análisis y explotación de la vulnerabilidad, se compila el código fuente bufOver4.c Durante la compilación aparece la advertencia: warning: the ‘gets’ function is dangerous and should not be used. Esto es esperable, ya que gets() es una función insegura y obsoleta. Una vez compilado, se ejecuta el binario: ./bufOver4. Y la salida es: buf: 6f1b9ce0 cookie: 6f1b9d3c

```
[guille@Orion ~]$ cd /home/guille/Documentos/reto8
[guille@Orion reto8]$ gcc -no-pie -fno-stack-protector -z execstack -mpreferred-stack-boundary=4 -o bufOver4 bufOver4.c
/usr/bin/ld: /tmp/cc6RZTZf.o: in function `main':
bufOver4.c:(.text+0x2f): warning: the `gets' function is dangerous and should not be used.
[guille@Orion reto8]$ ./bufOver4
buf: 6f1b9ce0 cookie: 6f1b9d3c
^C
[guille@Orion reto8]$
```

Esto muestra la dirección de inicio del buffer y de la variable cookie en la pila. Estos valores son útiles para entender cómo están dispuestas en memoria y cuánto hay que sobrescribir para alcanzar la dirección de retorno

4. **Análisis con GBD**

   Una vez compilado el binario sin protecciones, se utiliza GDB (GNU Debugger) para analizar su estructura en memoria y su flujo de ejecución.

   Primero se establece un punto de ruptura (breakpoint) al inicio de la función main, y luego se ejecuta el programa para examinar el código ensamblador generado:

```
[guille@Orion reto8]$ gdb ./bufOver4
GNU gdb (GDB) 16.3
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./bufOver4...

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.archlinux.org>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
(No debugging symbols found in ./bufOver4)
(gdb) break main
Breakpoint 1 at 0x40114a
(gdb) run
Starting program: /home/guille/Documentos/reto8/bufOver4
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Breakpoint 1, 0x000000000040114a in main ()
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000401146 <+0>:     push   %rbp
   0x0000000000401147 <+1>:     mov    %rsp,%rbp
=> 0x000000000040114a <+4>:     sub    $0x60,%rsp
   0x000000000040114e <+8>:     lea    -0x4(%rbp),%rdx
   0x0000000000401152 <+12>:    lea    -0x60(%rbp),%rax
   0x0000000000401156 <+16>:    lea    0xea7(%rip),%rcx        # 0x402004
   0x000000000040115d <+23>:    mov    %rax,%rsi
   0x0000000000401160 <+26>:    mov    %rcx,%rdi
   0x0000000000401163 <+29>:    mov    $0x0,%eax
   0x0000000000401168 <+34>:    call   0x401040 <printf@plt>
   0x000000000040116d <+39>:    lea    -0x60(%rbp),%rax
   0x0000000000401171 <+43>:    mov    %rax,%rdi
   0x0000000000401174 <+46>:    call   0x401050 <gets@plt>
   0x0000000000401179 <+51>:    mov    -0x4(%rbp),%eax
   0x000000000040117c <+54>:    cmp    $0xd0a00,%eax
   0x0000000000401181 <+59>:    jne    0x401192 <main+76>
   0x0000000000401183 <+61>:    lea    0xe92(%rip),%rax        # 0x40201c
   0x000000000040118a <+68>:    mov    %rax,%rdi
   0x000000000040118d <+71>:    call   0x401030 <puts@plt>
   0x0000000000401192 <+76>:    mov    $0x0,%eax
   0x0000000000401197 <+81>:    leave
   0x0000000000401198 <+82>:    ret
End of assembler dump.
(gdb)
```

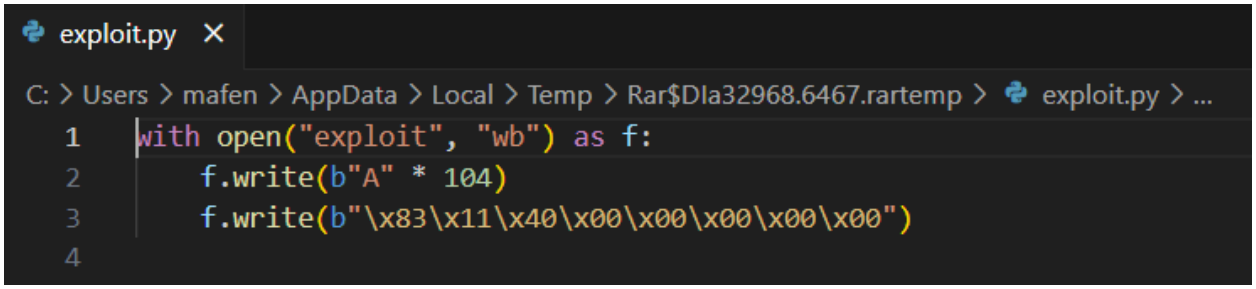
- En 0x401183 se encuentra la instrucción lea que carga en %rax la dirección de la cadena "you win!\n".
- Luego, mov %rax, %rdi coloca esta dirección como argumento para puts().
- Finalmente, call 0x401030 puts@plt imprime el mensaje.

Este bloque es exactamente lo que queremos ejecutar al hacer el exploit. Por tanto, nuestro objetivo será sobrescribir la dirección de retorno del main con la dirección 0x401183.

5. **Construcción del payload**

   Se determina que la dirección de retorno se encuentra a 104 bytes del inicio del buffer (96 bytes del buffer más 8 bytes para llegar al return pointer en sistemas de 64 bits). Por tanto, el exploit consiste en:

   - 104 bytes de relleno (carácter “A”)
   - La dirección 0x401183 codificada en formato little-endian



EJECUCIÓN DEL EXPLOIT

Una vez generado el archivo exploit usando el script Python, se ejecuta el binario redirigiendo su entrada estándar desde ese archivo:

Tras el mensaje aparece: Bus error (core dumped)

Este error es esperado, ya que el flujo de ejecución ha sido alterado intencionalmente y, tras ejecutar puts(), el programa intenta continuar en una dirección inválida. A pesar de ello, se demuestra que el exploit es exitoso al imprimir el mensaje objetivo.