

# SEGURIDAD EN SISTEMAS OPERATIVOS

## 4º Grado en Informática - Complementos de Ing. del Software

### Curso 2018-19

---

#### Práctica 2. Ingeniería inversa en Linux

#### Sesión 2. Explotaciones y protecciones del formato ELF

**Autor<sup>1</sup>:** Víctor García Carrera

---

#### Ejercicio 1.

---

Comenzamos el desarrollo de esta práctica conociendo algunas de las explotaciones y mecanismos de protección de un binario ELF. Vamos a recopilar diversa información acerca de nuestro sistema como su distribución, arquitectura y el compilador utilizado.

Distribución: Linux Mint 18.1 Serena (comando `cat /etc/issue`)

Arquitectura: 64-bit, x86\_64 (podemos verlo en la información de la CPU que podemos visualizar mediante el comando `lscpu`)

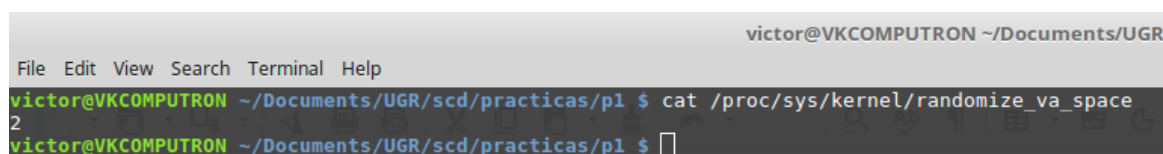
Kernel Release: 4.4.0-53-generic

Versión del compilador gcc (comando `cat /proc/version`): 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.4)

Una vez conocemos las versiones de nuestro sistema, procedemos a analizar los diversos mecanismos existentes para protegerse frente a ataques maliciosos de *buffer overflow* o desbordamiento de buffer.

El primer mecanismo observado es el *ALSR* (*Aleatorización de la disposición del espacio de direcciones*). Este mecanismo pertenece al kernel, estando presente en el kernel de Linux a partir de la versión 3.14, por lo que nuestro sistema dispone de ella. El kernel junto con el cargador de programas aleatoriza la localización dentro del espacio de direcciones de las áreas de datos tales como la base del ejecutable, las posiciones de la pila, el heap y las librerías con el objetivo de dificultar la predicción de un atacante acerca de las direcciones de memoria utilizadas por el programa para evitar que pueda alterarlas.

Este mecanismo necesita de la opción de compilación *-fPIE* para activar el mecanismo de *ejecutables independientes de la posición (PIE)*, el cual también presente en nuestro compilador. Podemos controlar ASLR mediante el valor de `/proc/sys/kernel/randomize_va_space`. De forma predeterminada, su valor es 2, indicando que realiza lo descrito previamente junto con la aleatorización de los segmentos de datos.



```
victor@VKCOMPUTRON ~/Documents/UGR
File Edit View Search Terminal Help
victor@VKCOMPUTRON ~/Documents/UGR/scd/practicass/pl $ cat /proc/sys/kernel/randomize_va_space
2
victor@VKCOMPUTRON ~/Documents/UGR/scd/practicass/pl $
```

---

<sup>1</sup> Como autor declaro que los contenidos del presente documento son originales y elaborados por mi. De no cumplir con este compromiso, soy consciente de que, de acuerdo con la “[Normativa de evaluación y de calificaciones de los estudiantes de la Universidad de Granada](#)” esto “conllevará la calificación numérica de cero ... independientemente del resto de calificaciones que el estudiante hubiera obtenido ...”

El siguiente mecanismo a escrutar es el de *protección de rotura de pila*, que va ligado al de *fuerza fortificada* o *fortify source*. Se trata de una protección en tiempo de compilación que activa una serie de protecciones en la *glibc* para detectar *buffer overflow* en diversas funciones que realizan operaciones con memoria y strings. No detecta todos los tipos de *buffer overflow* pero si que proporciona un nivel mayor de seguridad en aquellas funciones que son más propensas a causar fugas de memoria. Se activa con la directiva de compilación `-D_FORTIFY_SOURCE=x`, donde *x* puede valer 1, en cuyo caso es necesario también utilizar la opción de optimización del compilador de nivel 1 (`gcc -O1`) u otros niveles en adelante, y no afecta al comportamiento del programa, o *x* puede valer 2, donde se realizan más comprobaciones pero puede modificar el comportamiento esperado del programa. La utilización de este mecanismo está disponible en versiones de GCC posteriores a la 4.0, por lo que disponemos del mismo.

El siguiente mecanismo es bastante conocido, el de *protección de pila* (*Stack Smashing Protection*). Está disponible en versiones de GCC posteriores a la 3.2-7 (disponemos de ella). Las posibles opciones de compilación para implementarla son las siguientes: `-fno-stack-protector` deshabilita este mecanismo, `-fstack-protector` es la opción más utilizada y activa la técnica de canario de pila o *Stack Canary* sobre un conjunto de funciones potenciales de generar un *buffer overflow*, `-fstack-protector-all` cumple la misma función que la anterior pero sobre todas las funciones, y finalmente `-fstack-protector-strong`, que incluye funciones adicionales a proteger.

El último mecanismo es el de *protección de pila no ejecutable*. Creamos un ejecutable ELF llamado *prodcons\_exe* y con la instrucción `readelf -l prodcons_exe` podemos observar si la pila es o no ejecutable. La siguiente salida muestra como, en la línea referente a GNU\_STACK, los permisos son solo de lectura y escritura, por lo que la pila no es ejecutable. Podemos cambiar los permisos de la pila con el programa `exestack -s [binario]` o con la opción de compilación `-z execstack`.

```
victor@VKCOMPUTRON ~/Documents/UGR/scd/practicas/p1 $ readelf -l prodcons_exe
Elf file type is EXEC (Executable file)
Entry point 0x4018e0
There are 9 program headers, starting at offset 64

Program Headers:
Type           Offset             VirtAddr           PhysAddr
FileSiz        MemSiz             Flags             Align
PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
0x00000000000001f8 0x00000000000001f8 R E 8
INTERP         0x0000000000000238 0x0000000000000238 0x0000000000000238
0x000000000000001c 0x000000000000001c R 1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
LOAD           0x000000000000b7b8 0x000000000000b7b8 R E 200000
0x000000000000b7b8 0x000000000000b7b8 R E 200000
LOAD           0x000000000000b7b8 0x000000000000b7b8 R E 200000
0x000000000000b7b8 0x000000000000b7b8 R E 200000
DYNAMIC        0x000000000000b7f8 0x000000000000b7f8 0x000000000000b7f8
0x000000000000b7f8 0x000000000000b7f8 RW 8
NOTE           0x000000000000b7f8 0x000000000000b7f8 0x000000000000b7f8
0x000000000000b7f8 0x000000000000b7f8 RW 8
GNU_EH_FRAME   0x000000000000b7f8 0x000000000000b7f8 0x000000000000b7f8
0x000000000000b7f8 0x000000000000b7f8 R 4
GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
0x0000000000000000 0x0000000000000000 RW 10
GNU_RELRO      0x000000000000b7f8 0x000000000000b7f8 0x000000000000b7f8
0x000000000000b7f8 0x000000000000b7f8 R 1

Section to Segment mapping:
Segment Sections...
```

## Ejercicio 2.

En este último ejercicio utilizamos el código provisto del virus *lx3k2* que infecta un binario ELF, modificando su comportamiento para no generar ninguna carga maligna. Modificamos la ruta donde realiza el escaneo de ejecutables ELF a `/home/victor/VIRUS_PRUEBA` donde tenemos el binario *prodcons\_exe* a infectar. Utilizamos la directiva `#ifdef` para comprobar en tiempo de compilación la arquitectura que utilizamos y modificar el valor del campo `e_machine` de la struct `Elf32_Ehdr` en lugar del `EM_386` que el virus utiliza de forma predeterminada (significa que la CPU tiene una arquitectura de Intel 80386).