Ejercicios Módulo 1. Sesión 3: Usuarios y premisos

Ampliación de Sistemas Operativos. Curso 2017-2018

1. Objetivo

El objetivo de esta hoja de ejercicios es familizarizarse con las variables de entorno y la estructura de la pila en el entorno Linux / x64. Específicamente, trabajaremos:

- Consulta y modificación de variables de entorno
- Estructura de la pila al iniciar la ejecución de un proceso
- Introducción al problema de *buffer overflow* y su especial peligro en variables de pila (locales)

2. Lecturas previas recomendadas

- *Advanced Linux Programming*, capítulo 10.
- *Linux System Programming*, capítulo 5.

3. Variables de entorno y pila

Cuestión 1 . ¿Qué comando usarías para mostrar el contenido de la variable de entorno `PATH` en un terminal? Modifica dicha variable de modo que se incluya siempre el directorio actual (`.`) en la búsqueda de ejecutables. Comprueba que surte efecto tratando de ejecutar cualquier ejecutable del directorio actual sin poner la ruta relativa (`./`). Ahora, abre un nuevo terminal e inténtalo nuevamente. ¿Qué suecede? ¿Por qué?

entorno.c . Completa el código del fichero `entorno.c` tal y como se indica en las propias fuentes. Comprueba el valor de la variable `EDITOR` antes de su ejecución y después de la ejecución. ¿Qué ha ocurrido? ¿Ha funcionado correctamente tu código?

pilaInicial.c Lee el código y trata de prever qué ocurrirá al ejectuarlo. Compila y ejecuta el código. ¿Qué ha ocurrido? ¿Qué se muestra por pantalla?.

Prueba a pasar varios argumentos al ejecutar tu código. ¿Cambia algo la salida? ¿Por qué?

pila-buffer.c Lee y compila el código. Pruébablo introduciendo cadenas de caracteres de diferentes longitudes (es aconsejable hacerlo de forma controlada: primero 14 caracteres, luego 15, 16, 17...). ¿Observas comportamientos extraños? ¿Cuántos caracteres debes introducir para conseguir una violación de segmento? ¿Y para cambiar el valor de alguna variable local sin que se produzca la violación de segmento? ¿Cómo solucionarías el problema?

pila-buffer2.c (OPCIONAL). Este código es similar al anterior, pero permite pasar la cadena de caracteres directamente como argumento de entrada, y podemos comprobar que no sólo la función `gets()` es insegura. Trata de ejecutar el código de modo que NO se ejecuta el primer `printf` tras la llamada a `foo`. Para ello, realiza las siguientes acciones:

- Compila el código las opciones `-g -m64 -fno-stack-protector -z execstack`
- Depura el código usando `gdb`. Ejecuta paso a paso el código hasta llegar a la función `foo`. Anota la dirección de la instrucción posterior a la llamada a `printf` que quieres saltarte. Comprueba en qué dirección se ubica `buf` y en qué dirección se ha escrito la dirección de retorno.
- Si la diferencia de las direcciones anteriores es de 28 bytes, debes invocar el código con una cadena de 28 bytes cuyos últimos 4 bytes se correspondan con la dirección anotada en el punto anterior.
- Para conseguirlo, puedes usar `python`, de modo que la llamada al ejecutable sea:
  `>./pila-buffer2 $(python -c 'print "A" * 24 + "\x00\x00\x00\x40\x05\xa3"[::-1]')`
  siendo 0x004005a3 la dirección de la instrucción posterior al `printf` que deseamos saltar.

Cuestión 2 (OPCIONAL). Consulta la página de manual de `execstack` (`https://linux.die.net/man/8/execstack`). ¿Para qué servirá el flag `-z execstack` de la compilación anterior? ¿Cómo se puede explotar la vulnerabilidad anterior (*buffer overflow*) haciendo uso de este *flag*? ¿Qué ejecutables del sistema fijarías como objetivo prioritario para un ataque de este estilo?.

Compilación y Entrega

Crea un *makefile* que compile compile todos los ejemplos del directorio de modo que se generen ejecutables con el mismo nombre que el fuente, pero con extensión `.elf`. El *makefile* deberá incluir una regla `clean` que borre todos los ficheros `.o` y `.elf` generados. Asimismo, entrega un fichero PDF con las respuestas a las cuestiones y el código fuente de cada ejercicio (con los comentarios que estimes oportunos)