Práctica 4: Vulnerabilidades de desbordamiento

En esta práctica se van a aplicar los conocimientos adquiridos en clase en relación a la explotación de vulnerabilidades de *desbordamiento*. Se proporciona un programa en lenguaje C++, el objetivo de la práctica es identificar las vulnerabilidades del programa y explotarlas para redireccionar la ejecución. Se desactivarán algunas de las contramedidas de protección explicadas en clase para simplificar la explotación. Finalmente, se analiza la utilidad de las contramedidas de protección para hacer más difícil la explotación.

1. Entorno de práctica

La práctica se realiza con la máquina virtual que se proporciona en formato OVA (enlace disponible en Moodle). La máquina ejecuta una versión de Linux Ubuntu (32-bit) y hay que desplegarla en el ordenador personal con el software de virtualización de elección (VirtualBox, VMWare,etc.). Las credenciales de acceso a la máquina virtual son:

usuario: seed, **contraseña**: dees

2. El programa vulnerable

El programa vulnerable se encuentra en el directorio: `practica4/` de la máquina virtual. Para simplificar la explotación, desactivamos las contramedidas establecidas por defecto es decir:

1. La asignación aleatoria de las direcciones en el espacio de memoria (Address Space Layout Randomization – ASLR).
2. La protección de los canarios de pila.
3. La protección de pila no ejecutable (NX).

Para desactivar la primera contramedida usamos el siguiente comando:

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

Para desactivar la segunda y la tercera contramedidas, compilamos el programa utilizando las opciones `-fno-stack-protector` y `-z execstack`.

En el directorio `practica4/` se encuentra ya un ejecutable obtenido utilizando las siguientes opciones de compilación:

```
$ g++ -fno-stack-protector -z execstack -ggdb carrito.cpp -o carrito
```

2.1. Ejecución del programa

El programa lee datos introducidos por el stdin y escribe datos en el stdout. Ofrece cuatro opciones en el menú como se muestra a continuación.

```
------------------------------------------------------
Hola! ¿Qué quieres hacer?

1. Ver la comida en el carrito
2. Llenar el carrito
3. Mostrar calorías
4. Terminar

Elige >
```

La primera tarea es explorar como funciona el programa, eligiendo las diferentes opciones del menú.

Fallo del programa El programa tiene varias vulnerabilidades. Por ejemplo, si se introduce el valor `666` cuando el programa pide elegir una opción del menú (`1-4`) se verifica un fallo de segmento. Este fallo identifica una vulnerabilidad, pero hay otras. El objetivo de esta práctica es encontrar y explotar dos tipos de vulnerabilidades diferentes.

3. Explotación del programa

Para poder explotar una de las dos vulnerabilidades del programa, se necesitará introducir datos en binario. En el mismo directorio donde se encuentra el programa vulnerable hay un *script* (`runbin.sh`) que puede ser útil: convierte los dígitos en binario (por ejemplo, utilizando el formato en hexadecimal `\xHH`) y los pasa al programa vulnerable. Por ejemplo, ejecutando directamente el *script*:

```
------------------------------------------------------
Hola! ¿Qué quieres hacer?

1. Ver la comida en el carrito
2. Llenar el carrito
3. Mostrar calorías
4. Terminar

Elige > 2
Escribe la comida introducida en el carrito:
------------------------------------------------------

\x68\x75\x65\x76\x6f\x73
```

```
------------------------------------------------------
Hola! ¿Qué quieres hacer?

1. Ver la comida en el carrito
2. Llenar el carrito
3. Mostrar calorías
4. Terminar

Elige > 1
Comida: huevos
```

Herramientas básicas

- El depurador GNU (comando `gdb`) para obtener información sobre las direcciones y estado de la memoria asociada al programa en ejecución. A continuación dos enlaces de interés:
  - Comando básicos del depurador GDB: *GDB Cheat Sheet - DarkDust*, `http://darkdust.net/files/GDB Cheat Sheet.pdf`
  - Manual GDB: *GDB: The GNU Project Debugger*, `https://www.gnu.org/software/gdb/documentation/`
- Python o Perl, para la automatizar la generación de entradas de datos y/o ejecución.

3.1. Identificación vulnerabilidades

El primer paso es localizar las vulnerabilidades. Con este propósito, podemos mirar el código `carrito.cpp` y contestar a las siguientes preguntas:

Pregunta 1. Hay una vulnerabilidad asociada a una variable que puede ser indexada fuera de su límite.

1. ¿Cuál es la variable?
2. Indicar la línea de código que puede indexar la variable fuera de su límite.

Pregunta 2. Hay vulnerabilidades de desbordamiento de búfer en el programa.

1. ¿Cuáles son las variables?
2. ¿Qué parte de la memoria (e.g., código, data segment, bss, pila, heap, ...) asociada al proceso se puede desbordar?
3. Indicar las líneas de código que pueden desbordar los búferes.

Pregunta 3. ¿Hay otros tipos de vulnerabilidades en el código? ¿Cuáles (indicar las líneas de código correspondientes)?

3.2. Redirección de ejecución

En este paso consideramos la primera vulnerabilidad (ver la Pregunta 1) y la vamos a explotar para conseguir una redirección de la ejecución del programa vulnerable. En particular, queremos conseguir:

- Ejecutar el método `mostrarSecreto1` que escribe un mensaje por pantalla.
- Ejecutar el método `mostrarSecreto2` que escribe otro mensaje por pantalla.

Con este propósito vamos a lanzar el *script* `runbin.sh` y a utilizar el depurador `gdb` para examinar el estado de la memoria asociado al programa `carrito` en ejecución. Se puede lanzar el depurador con el siguiente comando:

```
$ gdb -p #PID
```

donde `#PID` es el identificador del proceso.

Las siguientes preguntas sirven para guiar, paso a paso, la explotación de esta vulnerabilidad. En el informe a entregar (ver Sección 6), se pide utilizar el formato hexadecimal en las respuestas (p.ej., 0xbffff530).

3.2.1. Ejecución del método mostrarSecreto1

Pregunta 4. ¿Cuál es las dirección de las variables `func` y `funcsec`? ¿En qué parte de la memoria se encuentran?

Pregunta 5. ¿Cuál es la dirección del método `showSecret1`?

Pregunta 6. ¿Qué datos de entrada proporcionas al programa para que `func[s]` lea (y luego intente ejecutar) el puntero a la función guardado en `funcsec`, en lugar de un puntero a una función guardado en `func`? Para contestar a esta pregunta tienes que hacer un cálculo aritmético considerando las direcciones que has encontrado anteriormente (teniendo en cuenta del tamaño de los punteros).

3.2.2. Ejecución del método mostrarSecreto2

Pregunta 7. ¿Cuál es la dirección del búfer asociado a la variable `resp`?

Pregunta 8. ¿Qué datos de entrada proporcionas al programa para que `func[s]` lea (y luego intente ejecutar) a partir del 126^o byte en `resp`, es decir a partir de `resp[125]`?

Se puede explotar el programa, proporcionando en entrada una secuencia de caracteres del siguiente tipo:

```
YYYYYYYYY\x00A...A\xEE\xEE\xEE\xEE
```

que causa una redirección de ejecución y la escritura del segundo mensaje secreto por pantalla. En particular, los puntos `...` indican una repetición de la 'A' tantas veces cuanto se necesite (*padding*). Los datos `YYYYYYYYY` y `\xEE\xEE\xEE\xEE` se obtienen depurando el programa.

Recuerda que estás trabajando con una arquitectura *little endian* así que, por ejemplo, la dirección `0xAABBCCDD` hay que introducirla en orden inverso, es decir `\xDD\xCC\xBB\xAA`.

Pregunta 9. ¿Hay otra forma de conseguir la escritura del segundo mensaje secreto por pantalla? En caso positivo, explicar cómo conseguirlo y motivar los pasos y datos introducidos.

3.3. Contramedidas

La explotación del programa se ha realizado con la desactivación de varias contramedidas (es decir: ASLR, canarios de pila y pila no ejecutable). Analiza cómo cada contramedida contribuye a proteger el sistema, activando una contramedida a la vez y repitiendo los pasos anteriores para redireccionar la ejecución del programa.

4. Bola extra: inyección de código

Se considera el segundo tipo de vulnerabilidad encontrada (ver la Pregunta 2). El objetivo es realizar un desbordamiento del búfer asociado a una variable para inyectar código y lanzar una *shell*.

Una posibilidad es elegir la opción 2 del menú presentado por el programa (opción *Llenar el carrito*) para realizar una llamada a la función correspondiente.

Se pide también analizar si las contramedidas ASLR, canarios de pila y pila no ejecutable, son eficaces para mitigar la inyección de código. **Este apartado no es guiado y es opcional.**

5. Puntuación

La valoración máxima de esta práctica es **10 puntos** así repartidos:

- Parte obligatoria (Secciones 1,2,3): **8 puntos**
- Bola extra (Sección 4): **2 puntos**

6. Entrega

Se pide entregar un informe *informePractica4.pdf* en Moodle que incluya la siguiente información:

- Nombre, apellidos y NIA de cada miembro del grupo de práctica, así como el correspondiente grupo de prácticas (*inf1* ... *inf6*).
- La respuesta a cada pregunta del enunciado y comentarios sobre la posible explotación del programa con las contramedidas activadas (sub-Sección 3.3). En las respuestas **hay que explicar el procedimiento seguido para obtener el resultado**, no es suficiente escribir sólo el resultado final.
- (**opcional**) Explicación de los pasos a seguir para realizar la inyección de código y análisis de la eficacia de las contramedidas.

La fecha límite de entrega de esta práctica es **tres semanas desde la realización de la práctica**.