

Práctica 4: Vulnerabilidades de desbordamiento

Seguridad Informática

Pedro Allué Tamargo (758267)

Juan José Tambo Tambo (755742)

5 de diciembre de 2020

Índice

1. Identificación de vulnerabilidades	1
1.1. Hay una vulnerabilidad asociada a una variable que puede ser indexada fuera de su límite	1
1.2. Hay vulnerabilidades de desbordamiento de búfer en el programa	1
1.3. ¿Hay otros tipos de vulnerabilidades en el código? ¿Cuáles?	1
2. Redirección de la ejecución	2
2.1. ¿Cuál es la dirección de las variables <code>func</code> y <code>funcsec</code> ? ¿En qué parte de la memoria se encuentran?	2
2.2. ¿Cuál es la dirección del método <code>showSecret1</code> ?	2
2.3. ¿Qué datos de entrada proporcionas al programa para que <code>func[s]</code> lea el puntero a la función guardado en <code>funcsec</code> , en lugar de un puntero a una función guardado en <code>func</code> ?	2
3. Ejecución del método <code>mostrarSecreto2</code>	3
3.1. ¿Cuál es la dirección del búfer asociado a la variable <code>resp</code> ?	3
3.2. ¿Qué datos de entrada proporcionas al programa para que <code>func[s]</code> lea a partir del 126º byte en <code>resp</code> , es decir, a partir de <code>resp[125]</code> ?	3
3.3. ¿Hay otra forma de conseguir la escritura del segundo mensaje secreto por pantalla?	4
4. Comentar las contramedidas	5
4.1. ASLR	5
4.2. Canarios de pila	5
4.3. Pila no ejecutable (NX)	5

1. Identificación de vulnerabilidades

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

- ¿Cuál es la variable?
 - La variable es `func`. Esta variable almacena un *array* de punteros a funciones que devuelven `void` y no aceptan parámetros.
 - Ejecutando el programa sin las contramedidas, cuando pide la introducción de una opción del menú, se introduce la opción 6 y se indexa el *array* `funcsec`, declarado en direcciones contiguas.
- Indicar la línea de código que puede indexar la variable fuera de su límite.
 - La variable se puede indexar fuera de su límite en la línea 131.

1.2. Hay vulnerabilidades de desbordamiento de búfer en el programa

- ¿Cuáles son las variables?
 - La variable `comida` en la función `llenarCarrito`. Acepta 512 *bytes* de longitud pero la función `scanf` no establece un límite para controlar la longitud de la cadena a copiar.
 - La variable `malo` en la función `mostrarCalorias` utiliza una versión no segura de la función `strlen` que devuelve el número de bytes (caracteres) entre una dirección de inicio y el carácter terminador 0. Si esta longitud es mayor que 512 (*MAX_SIZE*) se copiarán tantos caracteres como diga `len` o hasta llegar al carácter terminador.
- ¿Qué parte de la memoria asociada al proceso se puede desbordar?
 - Se podría desbordar la pila. Al ser variables que se declaran en funciones y no son globales se almacenan en la pila.
- Indicar las líneas de código que pueden desbordar los búferes.
 - `comida`: la función `scanf` (línea 58)
 - `malo`: la función `strlen` (línea 86) junto con la función `strncat` (línea 87).

1.3. ¿Hay otros tipos de vulnerabilidades en el código? ¿Cuáles?

- Hay una vulnerabilidad de desbordamiento de enteros en la función `mostrarCalorias`, en la línea 90 (variable `total`). Dada una lista de comidas lo suficientemente grande, y debido al bucle `for` de la línea 81 se podría desbordar el valor de esta variable.
- Existe otra vulnerabilidad de desbordamiento de enteros relacionada con la elección de la opción del menú del usuario (línea 127) puede desbordar el valor del entero `s` (función `atoi`, línea 130) si `resp` no se puede representar en el rango de los enteros (`int`). Una solución sería utilizar la función `strtol` que convierte una cadena de texto a un entero tipo `long` y ante desbordamientos, devuelve los límites máximos o mínimos del tipo `long`, dependiendo de por donde ha desbordado.

2. Redirección de la ejecución

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

Para obtener la localización de las variables en memoria mediante `gdb` se utilizará la orden: `print &variable`. Por lo tanto, las direcciones de las variables serán las siguientes:

- La variable `func` se encuentra en la dirección `0x804b064`
- La variable `funcsec` se encuentra en la dirección `0x804b078`

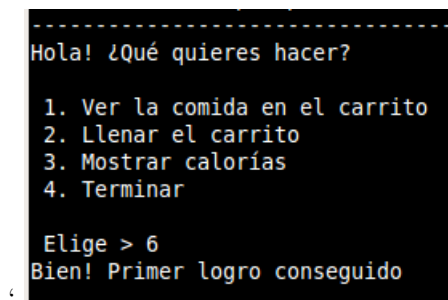
Las variables se encuentran en la zona de datos inicializados (*Initialized Data Segment*) ya que son variables globales cuyo valor ha sido otorgado por el programador.

2.2. ¿Cuál es la dirección del método `showSecret1`?

Para obtener la dirección de la función `showSecret1` mediante `gdb` se ha utilizado la siguiente orden: `print &Carrito::mostrarSecreto1`. Ya que `mostrarSecreto1` es un método estático de la clase `Carrito`. Su dirección de memoria es: `0x8048bce`.

2.3. ¿Qué datos de entrada proporcionas al programa para que `func[s]` lea el puntero a la función guardado en `funcsec`, en lugar de un puntero a una función guardado en `func`?

La entrada proporcionada al programa para leer un puntero guardado en `funcsec` sería de al menos 5. Esto es así ya que la dirección inicial de `func` es `0x804b064` y almacena punteros, cuyo tamaño son 4 *bytes*. Para leer un puntero de `funcsec` habría que indexar la quinta posición (empezando por 0) de `func` ($0x804b064 + (5 * \text{size_puntero}) = 0x804b078$).



```
-----
Hola! ¿Qué quieres hacer?

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

Elige > 6
Bien! Primer logro conseguido
```

Figura 1: Captura de pantalla del resultado de la ejecución de `mostrarSecreto1`

3.3. ¿Hay otra forma de conseguir la escritura del segundo mensaje secreto por pantalla?

Otra manera de explotar esta vulnerabilidad se corresponde con la modificación del registro que almacena la dirección de retorno de la función. Modificando este registro para que apunte a la función `mostrarSecreto2` conseguiremos ejecutarla.

La clave para esto es la utilización de la función `scanf()` de la función `llenarCarrito()`. Se va a explotar la función `scanf()` ya que es una función no segura que permite la escritura más allá del tamaño máximo del *buffer*, en este caso la variable *comida*.

Las direcciones de memoria importantes a tener en cuenta para explotar esta vulnerabilidad son las siguientes:

- Dirección de `mostrarSecreto2` = `0x08048a54`
- Dirección de `eip1` = `0xbffff32c`
- Dirección de `comida` = `0xbffff0f2`

Por lo tanto para sobrescribir el registro *eip* se deberá hacer que la variable comida desborde. La diferencia entre las direcciones de inicio de las variables es de *570B* y al ser variables de tipo **char** (1B) implica que son 570 posiciones de memoria. Para explotar esta vulnerabilidad se utilizará una cadena de caracteres de la forma: *AAAAA..AAAXXX* (Figura 4). En esta cadena existirán 570 caracteres (A) (codificados como espacios en hexadecimal (90)). Las X se corresponden con la dirección de la función `mostrarSecreto2()` codificado en *little endian* hexadecimal.

[illegible]

Figura 4: Captura de pantalla de la entrada necesaria para obtener el segundo logro explotando la vulnerabilidad encontrada en la función `scanf()`

¹<https://stackoverflow.com/questions/5144727/how-to-interpret-gdb-info-frame-output>

4. Comentar las contramedidas

4.1. ASLR

Para poder reactivar esta contramedida se debe ejecutar el siguiente comando:

```
$ sudo sysctl -w kernel.randomize_va_space=2
```

De esta manera se consigue la asignación aleatoria de las direcciones en el espacio de memoria. Si se procede a redireccionar la ejecución del programa y mostrar la función *mostrarSecreto2()* utilizando el primer método descrito, se muestra lo siguiente:

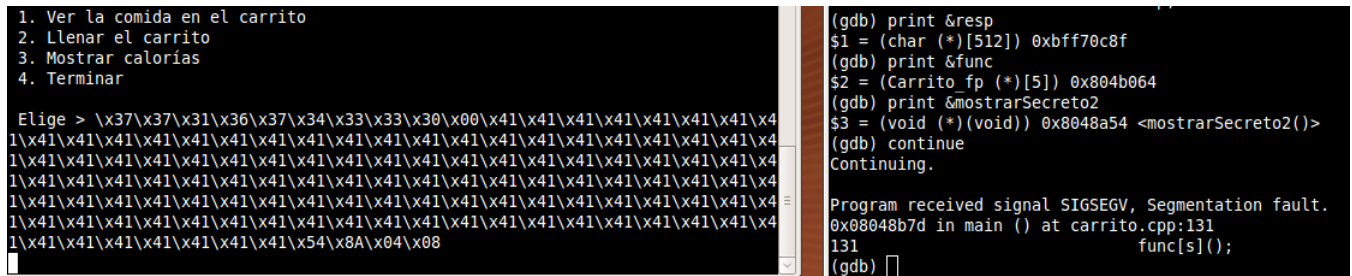


Figura 5: Captura de pantalla del intento de redireccionar código tras activar la primera contramedida (ASLR)

Como se puede observar, la dirección de la variable local *resp* (*0xbff70c8f*) ha variado con respecto a la que se había obtenido tras desactivar la contramedida (*0xbffff34f*). De hecho, la dirección varía con cada ejecución. Sin embargo, las direcciones de *func* y de la función *mostrarSecreto2()* no se han visto modificadas. La activación de esta contramedida hace más difícil los ataques de redirección, ya que al variar algunas direcciones de memoria, se debe volver a realizar los cálculos necesarios para calcular la entrada de tipo *YYYYYYYYx00A...AxEEExEEExEEExEE*.

4.2. Canarios de pila

Para poder reactivar la protección contra los canarios de pila, se debe compilar el programa de la siguiente manera:

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

La compilación activa esta contramedida por defecto. En el caso que se quiera indicar el flag de forma explícita, se debe compilar de la siguiente forma:

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

Esta contramedida lo que hace es proteger a funciones vulnerables que contienen:

- Un array de caracteres de más de 8 Bytes.
- Un entero (de 8 bits) de más de 8 Bytes.
- Una llamada a la función *alloca()* con un tamaño mayor a 8 Bytes.

Si se intenta realizar la redirección del programa siguiendo los pasos que se han mencionado en puntos anteriores, el propio programa aborta la ejecución al detectarse uno de los casos mencionados anteriormente, tal y como se puede observar en la siguiente figura:

Se observa que si un programa es compilado aplicando esta contramedida, resulta casi imposible realizar este tipo de ataques, ya que el programa abortará la ejecución sin poder acceder a la dirección de memoria deseada.

4.3. Pila no ejecutable (NX)

Para activar la contramedida que evita que la pila sea ejecutable, se debe compilar el programa de la siguiente manera:

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



Figura 6: Captura de pantalla del intento de redireccionar código tras activar la segunda contramedida (protección ante canarios de pila)

Cabe destacar que se ha vuelto a desactivar la contramedida de canario de pila, ya que se quiere observar el efecto que causa cada una de las contramedidas por separado. La contramedida *ASLR* también permanece desactivada.

Si se activa esta contramedida, se deshabilita la ejecución de la pila. Como se muestra a continuación, ejecutando `readelf -l carrito | grep -A2 -i stack` se puede observar el comportamiento de la pila del programa indicado.

```
seed@seed-desktop:~/practica4$ g++ -fno-stack-protector -z noexecstack -ggdb carrito.cpp -o carrito
seed@seed-desktop:~/practica4$ readelf -l carrito | grep -A2 -i stack
GNU_STACK 0x000000 0x00000000 0x00000000 0x000000 0x000000 RW 0x4
GNU_RELRO 0x001ef0 0x0804aef0 0x0804aef0 0x00110 0x00110 R 0x1

seed@seed-desktop:~/practica4$ g++ -fno-stack-protector -z execstack -ggdb carrito.cpp -o carrito
seed@seed-desktop:~/practica4$ readelf -l carrito | grep -A2 -i stack
GNU_STACK 0x000000 0x00000000 0x00000000 0x000000 0x000000 RWE 0x4
GNU_RELRO 0x001ef0 0x0804aef0 0x0804aef0 0x00110 0x00110 R 0x1
```

Figura 7: Captura de pantalla de aplicación de la contramedida contra la pila ejecutable (NX)

Se observa que, una vez activada la contramedida, la pila no puede ser ejecutada (*GNU_STACK* = RW). En caso contrario, sí que puede ser ejecutada (RWE). Sin embargo, se puede realizar el ataque de redirección de ejecución mediante los dos métodos mencionados en puntos anteriores (también se puede realizar el ataque de `mostrarSecreto1()`). Por ello, la activación de la contramedida de la pila ejecutable no evita que el programa sea vulnerable.