David Criado Ramón EC Grupo A1

PRÁCTICA 4 - SESIÓN 2
OBTENER Y MODIFICAR CLAVES DE LAS BOMBAS

Bombas realizadas en clase: **bombaProf5_08** **bombaProf6_08**
Bomba empezada en clase y terminada en casa: **bombaProf7_08**
Herramientas utilizadas: ghex y gdb (*tengo gdb configurado a mi manera y todos los datos son por defecto en hexadecimal, no obstante intentará no olvidarme de añadir los 0x pertinentes para que las operaciones tengan sentido)*.

BOMBA DE DIFICULTAD 5

5.A OBTENER LAS CLAVES: DFWwKFcn 5678

Tras realizar *disas main* de la función y ver el código se ve claro que es un formato muy similar al realizado, aunque probablemente compilado con -O1 (por la aparición de *repnz scas*). Así pues analizo los puntos junto antes de realizar las comparaciones que me pueden llevar a una explosión y que no tienen que ver con el tiempo límite:

```
(gdb) break *main+0x84
(gdb) break *main+0xE7
(gdb) run
      Introduce la contraseña: pepote
```

Así pues nos paramos en el primer breakpoint y comprobamos que hace strncmp a strncmp se le pasan dos parámetros iniciales que sé según el manual del programador de Linux que son cadenas de caracteres a comprobar, como primer argumento se pasa lo que había en %ebx, y como segundo argumento lo que había en una dirección de memoria sospechosa (0x804a02c) analizemos lo que hay:

```
(gdb) x/s $ebp
       0xffffd028:     "pepote\n"
(gdb) x/s 0x804a02c
       0x804a02c <password>:       "DFWwKFcn\n"
```

Con esto ya tenemos la primera contraseña DFWwKFcn, pero nos queda desverlar el código

```
(gdb) run
      Introduce la contraseña: DFWwKFcn
      Introduce el código: 7777
```

Ahora nos paramos en el segundo breakpoint (si lo deseas puedes eliminar el primero) y antes de que se realize la comparación entre una dirección de memoria muy sospechosa y %eax comprobamos que hay en cada una de ellas:

```
(gdb) print /d $eax
       $1 = 7777
(gdb) x/w 0x804a038
       0x804a038 <passcode>:       5678
```

Así pues tenemos también la contraseña, por tanto los datos necesarios para desactivar eran

DFWwKFcn 5678

5.B MODIFICAR LAS CLAVES: **ubrucudu** **5555**

En este caso utilizamos ghex, como sabemos que ambas están en memoria aprovechamos la herramienta de búsqueda textual y buscamos la clave:

(ghex->buscar->ventana texto) DFWwKFcn

Y sobrescribimos cada una de las letras por las nuevas correspondientes en la ventana de texto.
Buscamos ahora la clave numérica (en Little Endian):
5678 → (hex) 16 2E → (little endian) 2E 16
5555 → (hex) 15 B3 → (little endian) B3 15

(ghex->buscar->ventana hexa) 2E 16

Y sobrescribimos de nuevo pero en la venta hexadecimal los números dígito a dígito.

BOMBA DE DIFICULTAD 6

6.A OBTENER LAS CLAVES: **JSNhyBVR** **8585**

En esta bomba se nota que ha aumentado un poco la dificultad, en primer lugar vemos que ya las comparaciones que nos llevaban a la explosión de la bomba no están en en el main sino en dos funciones fun1 y fun2. Así pues en fun1 se trabaja con la primera entrada (string) y en fun2 con la segunda (entero). Centrémonos en fun1:

Lo primero que podemos ver es una cantidad muy elevada de mov sospechosos de datos inmediatos. Además nos fijamos en que el primero que introduce en memoria consecutiva ordenadamente. Así que le preguntamos a gdb tras hacer los mov, ¿que has puesto ahí? Como no conozco otra manera mejor miro las letras una a una:

```
(gdb) x/s $ebp-38
        …
(gdb) x/s $ebp-1c
```

Concatenando todas estas letras veo que la cadena en cuestión es: JSNhyBVR

Esta ya puedo adelantar qué es la contraseña pero necesito saber qué hace el resto del código para asegurarme de que todo funciona correctamente. La respuesta es bastante sencilla mientras que utiliza %ebx como índice crecientes de posiciones va comparando si el valor introducido almacenado en %edx corresponde con el valor que debe entrar almacenando en %eax (donde tanto %eax como %edx corresponden a una única letra) provocando la explosión en la primera comprobación que falle hasta que %ebx valga 8 y salga del bucle.

Pasemos a era ver el desensamblado de fun2.
En este caso la resolución es algo más compleja, vemos que una serie posiciones consecutivas por debajo de %ebp (variables locales probablemente) se escriben los enteros 5, 8, 5 y 8.
Tras hacer bastantes operaciones matemáticas diversas nos fijamos en una comparación particular antes de un salto en <fun2+0x4E> y vemos que va comparando (aumentando el índice en %ebx hasta 4) lo que hemos introducido del revés con lo que había inicialmente en memoria, por tanto resulta bastante evidente que la contraseña era 8585.

JSNhyBVR 8585

6.B MODIFICAR LAS CLAVES: **abracada** **7777**

Como en este caso los números están "camuflados" en las instrucciones mov, necesito modificar directamente las instrucciones. Así que la idea básica es mirar el contador de programa para alguna de ellas y ver la instrucción completa en hexadecimal para buscarla en ghex. Así pues pongo un breakpoint en el primer mov de fun1 y obtengo el siguiente resultado:

```
(gdb) x/x $eip
0x80486ab <fun1+0xC>:    0x4ac845c7
```

Así pues pasándolo a little endian busco dicha instrucción:

(ghex->buscar->ventana hexa) C7 45 C8 4A

Rápidamente en la ventana de texto vemos las letras separadas más o menos con .E (supongo que tendrá que ver con el código de la instrucción en concreto) y modificamos cada una de las claves.

El procedimiento que utilizo para modificar las claves numéricas es exactamente el mismo. Miro el contador de programa:

```
(gdb) x/x $eip
0x8048638 <fun2+0xC>:    0x05d845c7
```

Así pues pasándolo a little endian busco dicha instrucción:

(ghex->buscar->ventana hexa) C7 45 D8 05

Ahora, como sé que el último número es el 5 que quiero modificar simplemente lo modifico por 7, y como todas las instrucciones son consecutivas y el código de operación es muy similar modifico cuidadosamente el resto de datos.

BOMBA DE DIFICULTAD 7

7.A OBTENER LAS CLAVES: **aQmbYGsh** **2823**

Primero vemos que esta vez aparece una nueva función, con un nombre un poco obvio que lo que hace es aumentar el primer carácter que aparece en el segundo argumento, se le suma la cantidad que aparece en el primer argumento.

Una vez hemos averiguado lo que hace esa nueva función empezamos enfrentándonos al primer obstáculo para la desactivación: fun1.

Vemos con facilidad que hay una llamada a césar con la que se pasa al siguiente carácter a la primera letra de nuestra cadena (que se ha pasado como argumento 8(%ebp)). Sin embargo a la cadena que ahora buscaremos en una posición de memoria misteriosa se le resta un carácter. Por lo demás vemos que se encuentra en un bucle con comparaciones letra a letra y que el índice va hacia atrás indicando que la contraseña está almacenada en memoria del revés.

Así pues por ejemplo para la primera letra puedo saber que será la última de una serie de letras pero el valor que he de introducir es el de dos letras antes que la que obtenga de memoria.

Haciendo breakpoint en <fun1+2E> para cada iteración (he de empezar desde el principio cada vez que encuentre una letra poniendo la correcta sé que:

```
(gdb) x/c 0x8048a00+$ebx*4
0x8048a1c <arr1.3380+0x1C>:      0x63 → (ascii) 'c'
```

Que yo he de introducir como 'a'.
Siguiendo recursivamente este procedimiento obtengo que la contraseña es aQmbYGsh y en memoria está almacenada como juI[doSc, que me será de utilidad a la hora de modificar la bomba.

El siguiente obstáculo es la contraseña numérica que viene siendo una mezcla bastante fácil del sistema con la aparición de alguna operación peculiar como nop o un lea que no hace nada y utiliza %eiz (un pseudo-registro que siempre vale cero) que me hace sospechar que posiblemente esto haya sido compilado con -O2. Vemos que el dato que introducimos que se encuentra en %edx antes de llamar a césar es aumentado en un número mientras que el dato que obtenemos de una posición de memoria sospechosa antes de hacer la comparación es reducido en 1. Así pues sabemos que tenemos que introducir un número dos veces por debajo de lo que había en memoria al principio. Por tanto en un procedimiento análogo al que hicimos para la contraseña alfanumérica ponemos un breakpoint en <fun2+4C> y vemos que dato hay en dicha posición de memoria

```
(gdb) x/x 0x80489e0+$ebx*4
0x80489ec <arr2.3407+12>: 0x05
```

Además me doy cuenta de que otra vez el bucle va hacia atrás y por tanto estoy comprobando primero la último número y que $ebx empieza en 3 y acaba en -1 por lo que es un pin de 4 números. Repitiendo recursivamente el procedimiento de comprobar el dato que había en memoria (también podía haberlo comprobado en la siguiente línea en $eax tras hacer el mov) detecto la secuencia 5 4 A 4, que dándole la vuelta y aplicándole la codificación césar implica que debo introducir 2823.

Por tanto los datos que buscábamos eran:
aQmbYGsh **2823**

7.B MODIFICAR LAS CLAVES: **abracada** **2016**

Para modificar la contraseña nos fijamos en que habíamos introducido en posiciones de memoria consecutivas la cadena en cuestión, así que si buscando la j inicial, sólo me salen dos resultados y en uno de ellos se ve la secuencia, separada por series de 00 00 00 (debido al espacio restante, se puede deducir de la instrucción lea con la que se accede a memoria), así pues para poner como contraseña abracada me encargo de ir sustituyendo cada una de las letras originales por las letras cdtcecfc (para contrarrestar el efecto de la codificación césar) y yendo desde el final de la cadena hacia el principio ya que estaba puesto en memoria del revés.

En el caso de la contraseña numérica el formato es muy similar de hecho puedo buscar la cadena completa, que ahora si está en orden y modificarla en hexadecimal

(ghex → buscar → hexa) 04 00 00 00 0A 00 00 00 04 00 00 00 05 00 00 00
Y los modifico por sus correpondientes números para 2016 → 4238 en memoria
04 00 00 00 **02** 00 00 00 **03** 00 00 00 **08** 00 00 00