Código seguro

José María Alonso Cebrián
Jordi Gay Sensat
Antonio Guzmán Sacristán
Pedro Laguna Durán
Alejandro Martín Bailón
Jordi Serra Ruiz

PID_00208418

UOC
Universitat Oberta de Catalunya

Índice

1. *Integer Overflow*

A partir de este módulo se van a estudiar una serie de módulos centrados en la explotación de fallos de seguridad en el código mediante la ausencia de control de los datos de entrada. En el presente módulo, los desbordamientos se van a producir en los tipos de datos numéricos y se conoce la técnica como *Integer Overflow*.

Este fallo no es un error que nos vaya a permitir ejecutar código como en los ejemplos que veremos posteriormente o como en los que hemos mencionado en puntos anteriores, sino que nos va a permitir provocar fallos en la aplicación para comprobar su estabilidad o para saltarnos filtros de comprobación.

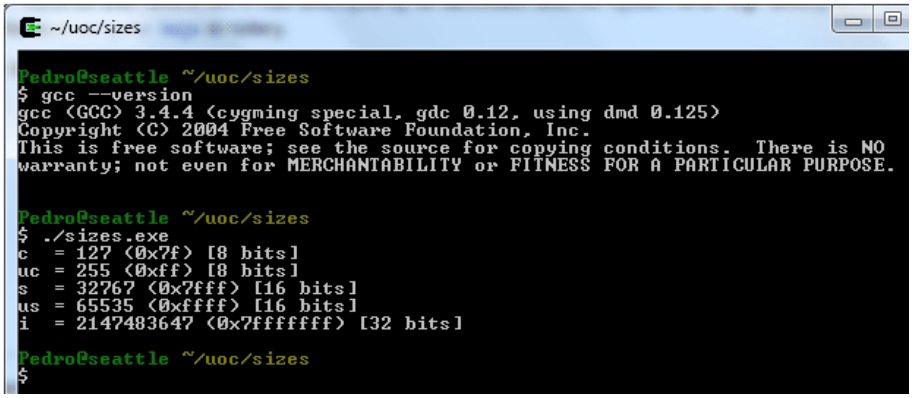
1.1. Tipos de datos númericos

Los ordenadores (la mayoría de ellos) guardan los datos numéricos en registros que son del tamaño de los punteros que utilizan. Esto, en el uso diario, es válido, pues corresponde a 32 bits (desde -2.147.483.648 a 2.147.483.647). Además, los ordenadores pueden trabajar con números más grandes y también más pequeños, pensados para adaptarse a las necesidades del programador y del programa. En ANSI C, las variables y sus tamaños máximos son:

Nombre	Tamaño
Char	-128 a 127 (8 bits)
unsigned char	0 a 255 (8 bits)
Short	-32768 a 32767 (16 bits)
unsigned short	0 a 65535 (16 bits)
Int	-2147483648 a 2147483647 (32 bits)

Tipos de datos numéricos en ANSI C

Para probar esto vamos a realizar un pequeño programa en C que compruebe los tamaños de cada tipo de dato. Debe notarse que el resultado puede variar de un ordenador a otro dependiendo de la versión del compilador que se esté usando y la arquitectura del equipo. En este ejemplo se está usando el compilador `gcc` en un entorno con CYGWIN sobre un sistema operativo Windows 7 Beta 1 de 32 bits:



Versión de GCC y ejecución del programa `sizes.c`

Como se puede observar en la figura anterior, el programa muestra el tamaño máximo de cada una de las variables que podemos definir. Al mismo tiempo el programa muestra su representación en hexadecimal y su tamaño en bits.

El código utilizado para esta prueba se puede ver a continuación para que puedas probarlo en tu máquina, con tu entorno software y con el compilador que estés utilizando.

```
#include <stdio.h>
int main(void)
{
   char c = 127;
   unsigned char uc = 255;
   short s = 32767;
   unsigned short us = 65535;
   int i = 2147483647;

   printf("c = %d (0x%x) [%d bits]\n", c, c, sizeof(c) * 8);
   printf("uc = %d (0x%x) [%d bits]\n", uc, uc, sizeof(uc) * 8);
   printf("s = %d (0x%x) [%d bits]\n", s, s, sizeof(s) * 8);
   printf("us = %d (0x%x) [%d bits]\n", us, us, sizeof(us) * 8);
   printf("i = %d (0x%x) [%d bits]\n", i, i, sizeof(i) * 8);

   return 0;
}
```

`sizes.c`

Como se puede observar, en la salida del programa la diferencia entre *signed* y *unsigned* es que el bit más significativo, conocido como MSB, en las variables *signed* es 0 en los valores positivos (7 en hexadecimal es 0111).

1.2. Representación de números negativos

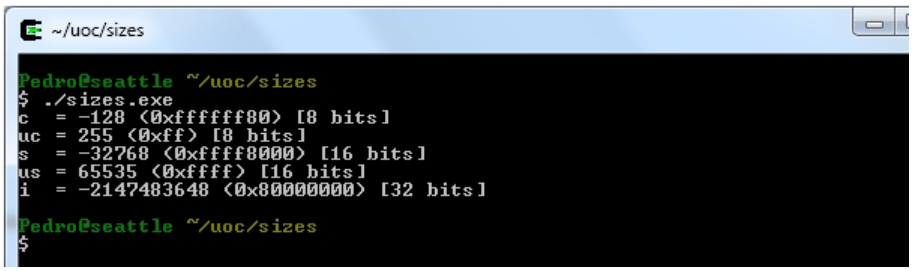
Como se puede ver en la imagen 3.2, en el tipo `char`, con sólo 8 bits, se puede representar el 127 como los bits 0111 1111 [0x7f]. Es decir, el MSB a 0 indicando un valor positivo y luego el valor 127. Por el contrario, en los valores negativos esto funcionará de forma diferente. En este caso el primer bit será el

1, indicando el signo negativo del número y después irá el valor a restar del mayor valor. Así, el valor 1000 0000 [0x80] será el -128. El -127 será 1000 0001 [0x81] y así sucesivamente hasta llegar al valor 1111 1111 [0xff] que será el -1.

Si establecemos los valores negativos de los tipos de datos que soportan el signo podremos ver cómo se almacenan los datos. Vamos a cambiar la inicialización de las variables en el programa anterior por los siguientes valores y vamos a ver cómo se representan los tipos de datos *signed*:

```
char c = -128;
unsigned char uc = 255;
short s = -32768;
unsigned short us = 65535;
int i = -2147483648;
```

En la siguiente figura se puede ver cómo los valores con signo son representados tal como se acaba de exponer.



Salida del programa `sizes.c` con valores negativos

Esto se implementa así para poder representar números negativos y, dependiendo de cómo se procesen los números entre tipos de datos de distintos tamaños, puede dar lugar a un fallo de *Integer Overflow*, como vamos a ver más adelante.

1.3. Desbordamiento de tipo de dato

Dadas las limitaciones y las formas de representar los datos en los tipos vistos, la pregunta que cabe hacerse es: ¿qué sucederá si intentamos instanciar una variable de mayor tamaño, un `int` o un `short`, en una más pequeña, `short` o `char`?

La repuesta es que el computador truncará los valores de datos introduciendo en la variable destino los bits que entren, empezando por los bits menos significativos y perdiendo, por tanto, los bits más significativos del valor.

Vamos a modificar el código `sizes.c` para asignar valores de mayor tamaño a variables de menor tamaño. Mostraremos por pantalla los resultados obtenidos y podremos ver cómo se ha comportado el sistema.

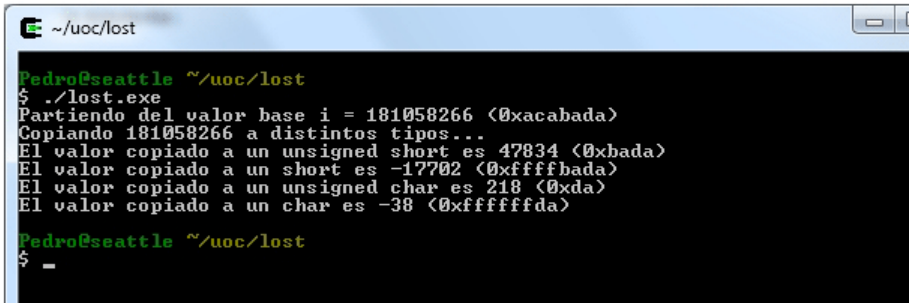
```
#include <stdio.h>
```

```
int main(void)
{
   char c = 0;
   unsigned char uc= 0;
   short s = 0;
   unsigned short us = 0;
   int i = 181058266;

   printf("Partiendo del valor base i = %d (0x%x)\n", i, i);
   printf("Copiando %d a distintos tipos...\n", i);

   us = i;
   printf("El valor copiado a un unsigned short es %d (0x%x)\n",us,us);

   s = i;
   printf("El valor copiado a un short es %d (0x%x)\n",s,s);

   uc = i;
   printf("El valor copiado a un unsigned char es %d (0x%x)\n",uc,uc);

   c = i;
   printf("El valor copiado a un char es %d (0x%x)\n",c,c);

   return 0;
}
```

Copia de valores mayores que el tipo de datos

En este código se ha inicializado la variable de tipo `intenger i` con un valor grande, en este caso concreto `181058266`. Este número es suficientemente grande como para desbordar el resto de los tipos de datos usados en el programa. Como se puede ver, este número va siendo copiado dentro de variables de menor tamaño y mostrado por pantalla.

La ejecución de este programa compilado dará una salida como la que se puede apreciar en la imagen siguiente:



Desbordamiento de tipos

Como se puede observar, los valores se truncan a la altura del máximo número de bits que pueda contener el tipo de datos sobre el que se ha copiado el valor de la variable original. Esto ocurre de manera drástica, sin acercarse siquiera al valor máximo permitido de cada uno de los tipos.

Esto puede implicar, como se puede ver en el caso del ejemplo, que para algunos tamaños de variables los valores contenidos sean positivos y otros negativos porque haya quedado un bit a uno en el bit de signo.

1.4. Ataques *Integer Overflow*

Después de ver cómo funcionan las variables en memoria en el módulo de introducción y cómo se tratan e interpretan, podemos empezar a analizar los procesos que ocurren internamente para que se produzca un *integer overflow*.

Los ataques de *Integer overflow* no nos permitirán sobrescribir zonas de memoria, variables o código, pero sí cambiar la lógica de la aplicación e incluso desbordar estructuras de memoria creadas a través de variables inseguras. Esto será debido a que se introducen en variables valores nunca esperados mediante el truncado indiscriminado de valores.

Vamos a hacer un ejemplo que muestre los dos fallos en un único código.

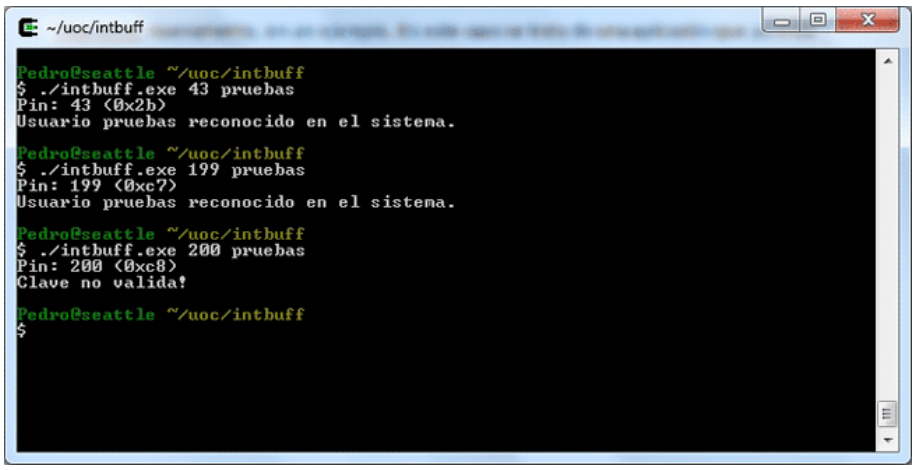
1.4.1. De mayor positivo a menor negativo

Supongamos un programa que reciba un número de *pin* y un usuario para acceder a una aplicación. Supongamos que sólo los usuarios con valor de *pin* inferior a 200 pueden entrar en el sistema. Una mala comprobación podría hacer que se utilice una variable short que permita valores negativos, a la que le asignaríamos un valor de tipo `unsigned short`.

El tipo `unsigned short` permite valores positivos que si se asignan a valores short, se convertirán en valores negativos.

Así, si asumimos que todos los usuarios tienen un *pin* positivo pero asignamos de alguna forma un valor de entrada `integer` o `unsigned short` a una variable `signed short`, conseguiremos que un valor mayor se convierta en un valor menor.

Si implementamos esta comprobación, es decir, que sólo puedan entrar usuarios con valores de *pin*, con una simple comparación del tipo *pin*<200, tendremos un problema a la hora de asignar valores `integer` o `unsigned` de gran tamaño a la variable `short pin`.



Ejecución de control por pin

Como se puede observar en la figura anterior, el programa ha recibido tres valores y sólo el último de ellos ha resultado incorrecto, pues llegaba al límite impuesto por la aplicación. Si echamos un vistazo al código, sólo vamos a obtener esos dos valores, es decir, usuario no reconocido o clave no válida, pues es sólo esto, una aplicación sintética para probar el desbordamiento.

Sin embargo, vemos que estamos asignando el valor que venga por línea de entrada `argc[1]` convertido a `integer` con la función `atoi` a una variable de tipo `short`. Además, la comprobación del *pin* se realiza como una comparación, como ya se ha explicado.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   short pin;
   char user[200];

   if(argc < 3)
   {
      printf("Uso: intbuff <pin> <usuario>\n");
      return -1;
   }

   pin = atoi(argv[1]);

   printf("Pin: %d (0x%x)\n", pin, pin);

   if(pin >= 200)
   {
      printf("Clave no valida!\n");
      return -1;
   }

   memcpy(user, argv[2], pin);

   printf("Usuario %s reconocido en el sistema.\n", user);

   return 0;
}
```

Código de comprobación de pin

A primera vista el código parece bueno, pero a la hora de ejecutarlo puede llegar a fallar.

Si en la aplicación introducimos un valor mayor del valor máximo de un `short` (32767) nos encontraremos con que la aplicación falla. ¿Por qué? Pues por la falta de comprobación de las variables.

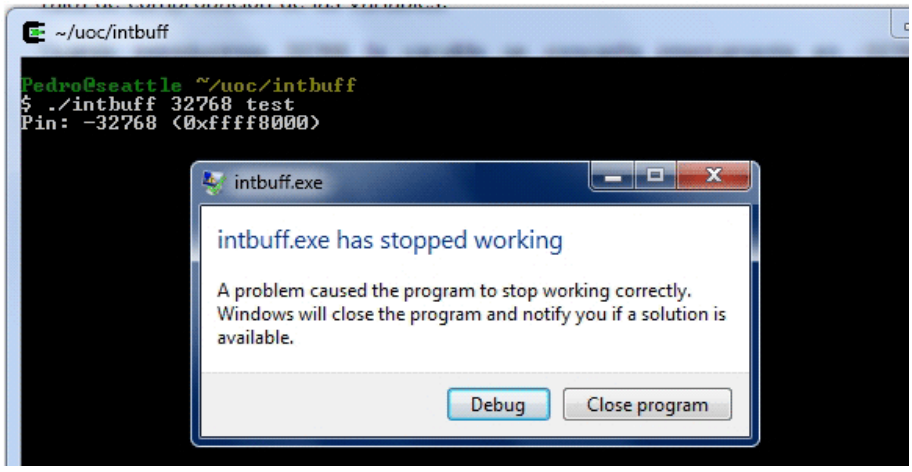
Cuando introducimos 32768, un valor mayor que el soportado por tipo short, la variable se convierte internamente en -32768 debido al uso que hacen los ordenadores del bit más significativo (MSB). Para el ordenador, el número `0xffff8000`, al valer 1 el MSB, es negativo. Por tanto, la comprobación de si el número es mayor o igual a 200 será, a todas luces, falsa.

Como se puede ver, gracias a una mala comprobación en el tipo de datos se ha producido un comportamiento erróneo en la lógica de la aplicación.

1.4.2. Desbordamiento de la aplicación

En la segunda parte del código de esta aplicación se hace uso de la variable para reservar memoria con la función `memcpy`. Lo habitual en esta función es realizar una reserva de memoria del tamaño de la variable que se quiere copiar, es decir, algo como `sizeof(user)`, pero en este caso se está realizando una asignación de memoria de la variable *pin*, que es de tipo `short` tras haber sido truncada de una variable `integer`.

El resultado que obtenemos si ponemos un tamaño muy grande es que vamos a desbordar la memoria de la aplicación y vamos a obtener un error como el que se puede ver en la siguiente figura.

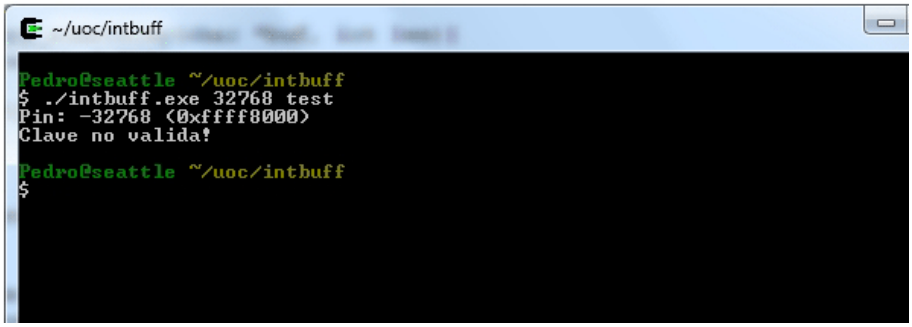


Desbordamiento de la memoria

No es el ejemplo más común del mundo, pero queríamos dejar clara la importancia de controlar el uso correcto de los tipos de datos numéricos.en todos las asignaciones de variables De cualquier manera, `memcpy`, como ya veremos más adelante, es una de las funciones más utilizadas en la creación de *exploits* de desbordamiento de memoria.

Como se puede observar, la comprobación de los valores numéricos ha de ser exhaustiva para que no se produzcan errores inesperados. Por ejemplo, en el código anterior, la solución habría pasado por incluir una comprobación para detectar si el *pin* introducido, además de ser menor de 200, era mayor que 0 y, por supuesto, comprobar el tamaño del tipo de datos antes de empezar a trabajar con él.

```
if(pin < 0 || pin >= 200)
{
   ...
}
```



Ejecución con el código parcheado

Como se ha visto, las conversiones de tipos pueden ser muy peligrosas; por eso, al trabajar con este tipo de variables debemos tener en cuenta todos los posibles valores que puedan tomar según el tipo de datos, por ejemplo:

- Menor que el valor mínimo.
- Menor que cero.
- Cero.
- Mayor que cero.
- Mayor que el valor máximo.

Estas son las pruebas más utilizadas por las herramientas de *fuzzing* a la hora de detectar valores inseguros y conversiones erróneas de tipos de datos numéricos en aplicaciones.

2. *Stack Overflow*

Uno de los fallos más importantes en el mundo de la seguridad informática ha sido el desbordamiento de variables en memoria. Este fallo es uno de los que más veces han posibilitado la creación de *exploits* para la ejecución de código en la máquina con el software vulnerable. Con este tipo de ataques en muchas aplicaciones vulnerables va a ser posible tomar el control sobre el flujo de ejecución de la aplicación.

Antes de comenzar es conveniente recordar los conceptos básicos de la memoria vistos anteriormente. La memoria tiene, según se vio, una zona dedicada a las variables del programa que se divide en dos: eel *stack* o pila y el *heap* o zona de memoria dinámica. La zona de *stack* tiene un crecimiento de arriba hacia abajo en cuanto a posiciones de memoria se refiere. En ella se va reservando memoria a medida que el programa va definiendo variables y almacenando estas últimas en formato *Little Endian*, es decir, con el bit menos significativo a la izquierda. Es importante tener presente estas peculiaridades, pues serán de gran importancia en todos los *exploits* que iremos viendo a la hora de modificar y sobrescribir la memoria.

2.1. Un ejemplo de desbordamiento de memoria

Empecemos con un pequeño programa que nos va a permitir entender y comprobar cómo funciona nuestro ordenador en lo que a registros y memoria se refiere. Los ejemplos que siguen a continuación se han realizado sobre una máquina de 32 bits, usando el compilador GCC en su versión 3.4.4 a través de la aplicación CYGWIN. Puede ocurrir que los datos aquí usados varíen un poco respecto a los que se deban introducir en otros entornos, pero se propondrá un método para aprender a ajustar los valores a cualquier plataforma.

Para poder desarrollar los ejercicios, vamos a trabajar con el programa que se puede ver en el código de ejemplo. En él se puede apreciar la existencia de la función `function` que recibe tres valores numéricos que son recogidos en las variables `a`, `b` y `c`.

La función inicializa dos *arrays* de 5 y 10 posiciones `buffer1` y `buffer2` que inicializa con dos *strings* de letras `A` y letras `B`.

Después, la función crea un puntero a un valor numérico. Este puntero se hace apuntar a la dirección de memoria situada 28 bytes más allá del comienzo del `buffer1`. Recordemos que `buffer1` en realidad es un puntero a la primera posición del *array*.

Por último, la función incrementa en 7 el valor existente en esa posición de memoria.

El código principal del programa crea una variable `integer x` que es inicializada a 0. Después se llama a la función creada con los valores `1,2,3`, se asigna el valor `1` a la variable `x` y se imprime el valor de `x`.

```
#include <stdio.h>
void function(int a, int b, int c)
{
   char buffer1[5] = "AAAAA";
   char buffer2[10] = "BBBBBBBBBB";
   int *ret;

   ret = buffer1 + 28;
   (*ret) += 7;
}

int main(void)
{
   int x;

   x = 0;
   function(1,2,3);
   x = 1;
   printf("%d\n",x);

   return 0;
}
```

Código de ejemplo

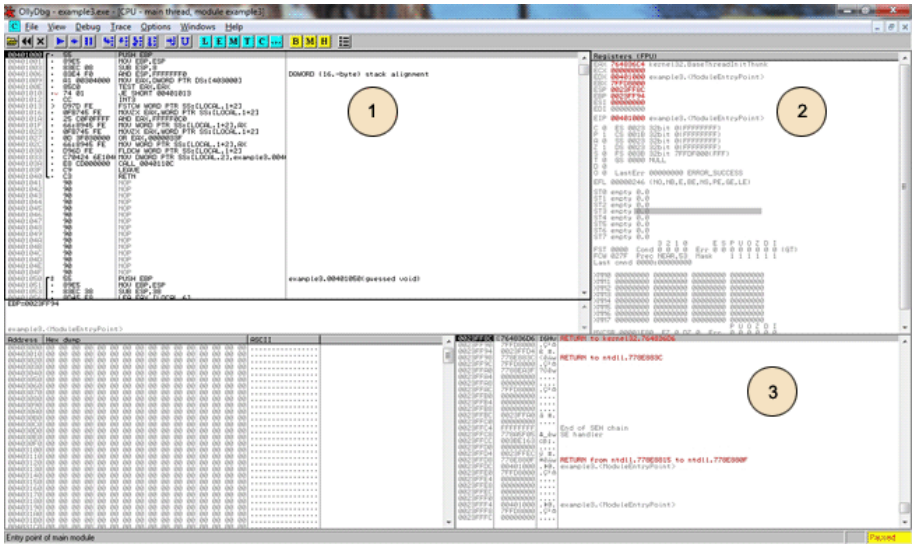
Este programa de propósito simple debería crear una variable que se inicia a `0`, llamar a una función que aparentemente no hace nada, poner la variable a `1` y terminar imprimiendo este valor. Sin embargo, y gracias al conocimiento que vamos a ir adquiriendo sobre la memoria, vamos a lograr que nuestro programa se "salte" la asignación de un nuevo valor a la variable `x` e imprima por consola el valor de `0`.

2.1.1. Análisis inicial

Para empezar con el análisis debemos compilar el programa y usar algún tipo de *debugger* para observar el código ensamblador generado. De esta manera se podrá depurar poco a poco.

En esta ocasión vamos a usar el programa gratuito llamado Ollydbg en su versión 2 (Beta 1). Este *debugger*, como ya hemos visto, nos va a permitir abarcar de un vistazo distintas zonas importantes de la ejecución de nuestro código, tal como se vio en los ejercicios del primer módulo:

- Código ensamblador
- Registros de memoria
- Memoria del programa



Zonas de pantalla en Ollydbg

Tras compilar el programa anterior y abrirlo con Ollydbg, podremos observar que se ha obtenido el código ensamblador mostrado a continuación. Como se puede ver, el código generado no es muy grande, lo que nos permitirá analizarlo más fácilmente.

```
00401050 /$ 55              PUSH EBP
00401051 |. 89E5            MOV EBP,ESP
00401053 |. 83EC 38          SUB ESP,38
00401056 |. 8D45 E8          LEA EAX,[LOCAL.6]
00401059 |. 83C0 1C          ADD EAX,1C
0040105C |. 8945 D4          MOV DWORD PTR SS:[LOCAL.11],EAX
0040105F |. 8B55 D4          MOV EDX,DWORD PTR SS:[LOCAL.11]
00401062 |. 8B45 D4          MOV EAX,DWORD PTR SS:[LOCAL.11]
00401065 |. 8B00             MOV EAX,DWORD PTR DS:[EAX]
00401067 |. 83C0 07          ADD EAX,7
0040106A |. 8902             MOV DWORD PTR DS:[EDX],EAX
0040106C |. C9               LEAVE
0040106D \. C3               RETN

0040106E /. 55              PUSH EBP
0040106F |. 89E5            MOV EBP,ESP
00401071 |. 83EC 18          SUB ESP,18
00401074 |. 83E4 F0          AND ESP,FFFFFFF0
00401077 |. B8 00000000      MOV EAX,0
0040107C |. 83C0 0F          ADD EAX,0F
0040107F |. 83C0 0F          ADD EAX,0F
00401082 |. C1E8 04          SHR EAX,4
00401085 |. C1E0 04          SHL EAX,4
00401088 |. 8945 F8          MOV DWORD PTR SS:[LOCAL.2],EAX
0040108B |. 8B45 F8          MOV EAX,DWORD PTR SS:[LOCAL.2]
0040108E |. E8 49000000      CALL 004010DC
00401093 |. E8 D4000000      CALL <JMP.&cygwin1.__main>
00401098 |. C745 FC 00000    MOV DWORD PTR SS:[LOCAL.1],0
0040109F |. C74424 08 030    MOV DWORD PTR SS:[ESP+8],3
004010A7 |. C74424 04 020    MOV DWORD PTR SS:[ESP+4],2
004010AF |. C70424 010000    MOV DWORD PTR SS:[ESP],1
004010B6 |. E8 95FFFFFF      CALL 00401050
004010BB |. C745 FC 01000    MOV DWORD PTR SS:[LOCAL.1],1
004010C2 |. 8B45 FC          MOV EAX,DWORD PTR SS:[LOCAL.1]
004010C5 |. 894424 04        MOV DWORD PTR SS:[ESP+4],EAX
004010C9 |. C70424 002040    MOV DWORD PTR SS:[ESP],00402000
004010D0 |. E8 A7000000      CALL <JMP.&cygwin1.printf>
004010D5 |. B8 00000000      MOV EAX,0
004010DA |. C9               LEAVE
004010DB \. C3               RETN
```

Código de ensamblador resultante

Analicemos el código ensamblador para entender cómo funciona el ordenador internamente en esta arquitectura concreta, pero para entenderlo, hagamos un breve repaso a la arquitectura interna de la máquina que vamos a necesitar ahora.

2.1.2. Los registros

Dentro de la arquitectura x86 hay diversas formas de programar en ensamblador a la hora de gestionar los registros del sistema, pero hay ciertos registros e instrucciones bastante comunes en la mayoría de los compiladores que se utilicen. De todos modos, como ya se anticipó en el segundo módulo, debemos tener presente que un mismo código en lenguaje ANSI C puede generar distintos códigos en ensamblador dependiendo del compilador utilizado. De entre todos los registros, los más importantes que vamos a utilizar aquí son:

- `EIP`. Es el *Instruction Pointer* y guarda la dirección actualmente en ejecución, es decir, el registro donde se almacena el control de programa.

- `ESP`. Es el *Stack Pointer* y guarda la cabeza de la pila. Hay que tener en cuenta que la pila crece de forma invertida, es decir, cada vez que la pila crece la dirección de memoria decrece.

- `EBP`. Es el *Base Pointer* y guarda la dirección de memoria donde comienza la pila. Al crecer de forma invertida, es la dirección mayor dentro de la pila.

Aparte de estos tres registros especiales del código, se puede ver que se hace uso del registro EAX como variable auxiliar para mover valores entre variables.

Cuando una función está en ejecución, el registro EIP apunta a la instrucción en ejecución, el registro EBP a la dirección base de la pila y el registro ESP a la cabeza de la pila.

2.1.3. Gestión de la pila

La zona de memoria situada entre el EBP y el ESP se conoce como el *Stack Frame*, y marca la zona de memoria de la función asignada a la pila.

Cuando se realiza una llamada a la función `POP` con un registro como parámetro, el valor situado en la posición de memoria apuntado por ESP será asignado al registro y el valor de ESP se desplazará para quitar este valor de la pila. En este caso, la pila se desplaza al tamaño del registro extraído, así que ESP quedará asignado a ESP +4 [en arquitecturas de 32 bits].

Cuando se realiza una llamada a la función `PUSH` con un registro o valor como parámetro, este será puesto en la cima de la pila y el registro ESP se desplazará para indicar que la pila está en la nueva posición. En este caso ESP será asignado a ESP – 4.

2.1.4. Llamada y retorno de funciones

Cuando se produce la invocación de una función, existen otras tres funciones importantes que también afectan a los registros y la pila. Es obligatorio conocerlas para comprender cómo funciona el código del ejemplo:

- `CALL`. Cuando se produce una llamada a otra función, es necesario realizar una serie de acciones. `CALL` automatiza estas acciones. En primer lugar hace un `PUSH` del valor siguiente de EIP, es decir, de la dirección donde se encuentra la siguiente instrucción a ejecutar después de que el control de programa regrese de la llamada a la función. Este valor será el valor de retorno de la función llamada. En segundo lugar actualizará el valor de EIP a la dirección de la función llamada. Es decir, `CALL Address` genera un `PUSH EIP` siguiente y una llamada a la función `MOV` (mover) `EIP, Address`.

- `LEAVE`. Cuando se va a abandonar la ejecución de una rutina, se puede usar la llamada a `LEAVE` para preparar la salida. Para ello, esta función sitúa la cabeza de la pila en la dirección de la base, es decir, `MOV ESP, EBP`. Después hace un `POP` al registro EBP, es decir, restaura el valor original de EBP. Este valor de EBP es guardado en el comienzo de la función, como se verá en el ejemplo de este tema. Esta situación deja al registro ESP apuntando a la dirección de retorno `CALL`, es decir, la siguiente instrucción a ejecutar tras la finalización de la función.

- `RTN`. La llamada a `RTN` genera el fin de la ejecución de una función y lo que se hace es actualizar el valor de EIP al valor de ESP, es decir, es un `POP EIP`.

2.2. Analizando el código del ejemplo

Como se puede observar a simple vista, tenemos dos zonas de código que corresponden a las dos funciones de nuestro programa: la función `function` y el programa principal `main`. La primera de ellas termina en `0040106D` y la segunda empieza en la línea siguiente y termina en `004010DB`, es decir, en la última línea del código ensamblador mostrado. Al trabajar con un entorno cygwin se puede ver que al principio el programa realizar una serie de funciones de inicialización y pre- proceso.

Después podemos ver cómo se asigna el valor 0 a la variable `X` en la posición de memoria `00401098`:

```
00401098 |.     C745 FC 00000      MOV DWORD PTR SS:[LOCAL.1],0
```

Después se introducen en la pila los valores 1, 2 y 3 que se pasan a `function` en las líneas `0040109F`, `004010A7` y `004010AF`. Podrían haberse utilizado llamadas a `PUSH`, pero es decisión del compilador copiar los valores que se pasan por parámetro con la función `MOV` y, en el comienzo de la función llamada actualizar el *stack frame*.

```
0040109F |.      C74424 08 030      MOV DWORD PTR SS:[ESP+8],3
004010A7 |.      C74424 04 020      MOV DWORD PTR SS:[ESP+4],2
004010AF |.      C70424 010000      MOV DWORD PTR SS:[ESP],1
```

Como se puede ver, la memoria funciona como una pila *LIFO*, por lo que se entiende de esta manera que las variables sean pasadas en orden inverso.

En la línea siguiente, la `004010B6`, se hace una llamada a `00401050` con la instrucción `CALL`, o lo que es lo mismo, una llamada a la función `function`.

```
004010B6 |      E8 95FFFFFF      CALL 00401050
```

Esta instrucción apila el valor del siguiente EIP a ejecutar, es decir, la dirección de retorno de la función llamada y actualiza el valor de EIP a la dirección pasada por parámetro en la llamada a `CALL`.

Si vamos al comienzo de la función, hemos de pararnos para analizar las tres primeras instrucciones que acompañan a toda función y que son de gran importancia.

```
00401050 /$      55        PUSH EBP
00401051 |.      89E5      MOV EBP,ESP
00401053 |.      83EC      38 SUB ESP,38
```

La primera instrucción guarda en la pila el valor actual del registro EBP, es decir, la dirección base de la pila de la función que le llama y actualiza el valor de EBP con el valor actual de la cima de la pila. Después sitúa el registro ESP en el lugar donde se encuentra la cima de la pila actualmente. La pila reserva para las variables que usa la función, en este caso 38 en hexadecimal, por lo que reserva 56 bytes, es decir, 14 posiciones de 4 bytes:

- 16 bytes de `char buffer1[5]`
- 16 bytes de `char buffer2[10]`
- 4 bytes del `int *ret`
- 20 bytes faltan

2.3. La función `function`

Una vez entendidas estas ideas, vamos a centrarnos en la función `function` y analizar cómo modifica la dirección de retorno de la función `function` para que se salte la instrucción de asignación del valor `1` a la variable `x`. Esta instrucción se encuentra en la siguiente dirección de memoria:

```
004010BB |. C745 FC 01000 MOV DWORD PTR SS:[LOCAL.1],1
```

Si se quiere saltar esa instrucción, deberemos modificar el valor que obtendrá EIP cuando termine la ejecución de la rutina llamada. EIP será actualizado con el valor almacenado por la instrucción `CALL` que, en este caso, llevaría a la ejecución de la instrucción anterior. Si queremos saltarnos la ejecución de la instrucción `004010BB` deberemos desplazar el valor del registro de dirección de retorno tantas posiciones como distancia haya entre `004010C2` (la siguiente instrucción) y `004010BB` (la instrucción que deseamos saltarnos). Una sencilla resta en hexadecimal nos revela que la distancia es 7 bytes. Necesitamos, entonces, que la dirección de retorno de la función se desplace 7 bytes.

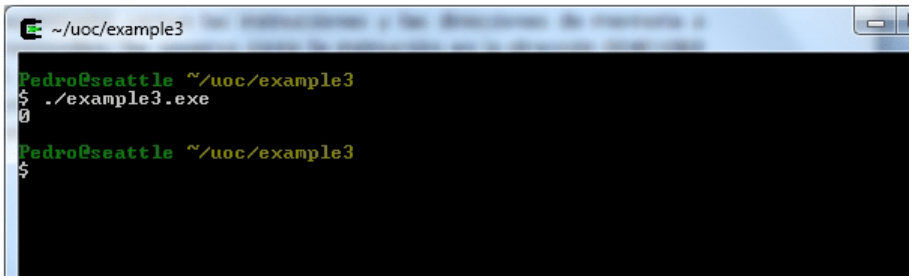
La parte del código en C que realizará el salto de esta dirección de memoria es la siguiente:

```
ret = buffer1 + 28;
(*ret) += 7;
```

La primera instrucción en el código C localiza la posición en memoria de la dirección de retorno. Para descubrir el valor que hay que desplazarse desde el inicio de la variable `buffer1`, hay que hacer un análisis del estado de la pila con Ollydbg o gdb.

Tras hacer el volcado se puede ver que la variable `Buffer1` está a 28 bytes de la dirección de retorno. Basta con desplazar la variable `ret` a esta posición e incrementar el valor almacenado en esa dirección en 7 bytes.

Con esto se consigue que cuando la rutina termine su ejecución EIP sea actualizado a la instrucción siguiente a la asignación del valor `1` a `x`, obteniéndose como resultado la siguiente salida por pantalla.



Salida del código sin ejecución de la instrucción x=1

2.4. Ataque *Stack Overflow*

El ejemplo anterior nos ha servido para analizar cómo funcionan los registros y cómo es posible alterar el flujo del programa alterando los valores de los registros en tiempo real. Lógicamente, a la hora de crear un *exploit* no va a ser posible compilar un código en C, pero en la práctica sí podremos modificar valores en los registros mediante vulnerabilidades de *stack overflow*.

¿En qué consiste el *stack overflow*? Como se ha visto anteriormente, en el programa hemos modificado el flujo de nuestra aplicación haciendo que se salte una instrucción. Esto se puede lograr haciendo uso de una variable no controlada que nos permita colocar como dirección de retorno la dirección de memoria que nosotros deseemos.

De nuevo para lograr un aprendizaje basado en la experiencia vamos a compilar un pequeño programa vulnerable y vamos a explotarlo para lograr el efecto que nosotros deseemos.

```
#include <stdio.h>

int main(void)
{
   char text[16];
   int i = 0;

   printf("texto:");
   scanf("%s", text);

   i++;
   printf("El valor de i es %d\n", i);

   return 0;
}
```

Código en ANSI C vulnerable a *Stack Overflow*

Este código está asignado el valor de la entrada por pantalla recogida por la función *scanf* a una *array* de caracteres de 16 bytes. Sin embargo, no se está realizando ninguna comprobación de tamaño, por lo que el código es vulnerable al desbordamiento. Al ejecutar este código podemos empezar a probar sobre él distintos tipos de entrada a la variable text. Los valores que vamos a probar son:

- X < 16 "A". Con esta entrada nuestro programa funcionara normalmente.

```
Pedro@seattle ~/uoc/stack
$ ./stack.exe
texto:AAAAAAAAAA
El valor de i es 1
```

Ejecución correcta con "A"<16

- X = 16 "A". El programa seguirá funcionando perfectamente, pues no sobrescribimos ninguna zona de memoria que sea crítica para la ejecución.

Ejecución correcta con "A"=16

- X > 16 "A". Aquí debemos observar dos posibles casos:

- X < 28 "A"

Este caso se parece al anterior, el programa se ejecuta normalmente y no parece verse afectado por la sobrescritura de la memoria. Para entender esto es necesario ver como se está asignando la memoria a las diferentes variables. Si realizamos un análisis con Ollydbg, obtendremos los siguientes:

Colocación de las variables en memoria

Como se puede observar en la imagen, la variable `i` ocupa la posición `0023CC5C` y la variable `text` debería ocupar desde `0023CC60` hasta `0023CC6C`, pero al haber sobrescrito el buffer, casi llegamos hasta la dirección `0023CC7C`, donde se guarda la dirección de retorno.

- X >= 28 "A". Recordemos que en C, al pulsar la tecla `Enter`, cuando se introduce una cadena se añade también el carácter `\0` de final de línea, así que serán 29 los caracteres almacenados en memoria. Esto quiere decir que si introducimos 28 o más "A" se estará empezando a escribir en la posición `0023CC7C` y salvo que se encuentre una dirección e memoria válida, se obtendrá un error en la ejecución.

Sobrecritura de la dirección de retorno con "A"

El programa llega a un punto donde se le dice que la siguiente instrucción a ejecutar se encuentra en la posición `41414141 (AAAA)` y al no encontrar instrucciones válidas falla y la ejecución se corta.

2.5. Ejecución de código por *Stack Overflow*

En este ejemplo queda claro que si podemos sobrescribir la dirección de memoria por medio de un desbordamiento de las variables de la pila, sería posible hacer que el control de programa fuera a cualquier parte de la memoria, es decir, se podría ejecutar cualquier programa cargado en el sistema o incluso inyectado en las variables.

Como todavía no hemos introducido código ejecutable en memoria y no sabemos aún como hacerlo, vamos a aprovecharnos del código propio de la aplicación para crear un bucle en el que se incremente por segunda vez la variable `i`.

Para eso, lo que vamos a hacer es determinar cuál es la instrucción que aumenta en uno la variable `i`. Si miramos el código ensamblado que se ha generado con esto programa obtenemos lo siguiente:

```
00401050 /. 55                PUSH EBP
00401051 |. 89E5              MOV EBP,ESP
00401053 |. 83EC 48           SUB ESP,48
00401056 |. 83E4 F0           AND ESP,FFFFFFF0
00401059 |. B8 00000000       MOV EAX,0
0040105E |. 83C0 0F           ADD EAX,0F
00401061 |. 83C0 0F           ADD EAX,0F
00401064 |. C1E8 04           SHR EAX,4
00401067 |. C1E0 04           SHL EAX,4
0040106A |. 8945 D4           MOV DWORD PTR SS:[LOCAL.11],EAX
0040106D |. 8B45 D4           MOV EAX,DWORD PTR SS:[LOCAL.11]
00401070 |. E8 4B000000       CALL 004010C0
00401075 |. E8 D6000000       CALL <JMP.&ygwin1.__main>
0040107A |. C745 E4 00000     MOV DWORD PTR SS:[LOCAL.7],0
00401081 |. C70424 002040     MOV DWORD PTR SS:[ESP],stack.00402000
00401088 |. E8 E3000000       CALL &lt;JMP.&cygwin1.printf&gt;
0040108D |. 8D45 E8           LEA EAX,[LOCAL.6]
00401090 |. 894424 04         MOV DWORD PTR SS:[ESP+4],EAX
00401094 |. C70424 072040     MOV DWORD PTR SS:[ESP],stack.00402007
0040109B |. E8 C0000000       CALL <JMP.&cygwin1.scanf>
004010A0 |. 8D45 E4           LEA EAX,[LOCAL.7]
004010A3 |. FF00              INC DWORD PTR DS:[EAX]
004010A5 |. 8B45 E4           MOV EAX,DWORD PTR SS:[LOCAL.7]
004010A8 |. 894424 04         MOV DWORD PTR SS:[ESP+4],EAX
004010AC |. C70424 0A2040     MOV DWORD PTR SS:[ESP],stack.0040200A
004010B3 |. E8 B8000000       CALL <JMP.&cygwin1.printf>
004010B8 |. B8 00000000       MOV EAX,0
004010BD |. C9                LEAVE
004010BE \. C3                RETN
```

Código ensamblador vulnerable

Como se puede ver en el código anterior, la instrucción que aumenta en uno la variable `i` está en `004010A3`, por lo que tendríamos que establecer la dirección de retorno a `004010A0`.

2.6. Introducción de direcciones de memoria por teclado

Al final, a la hora de escribir la dirección de memoria en la función de retorno, necesitaremos escribir mediante el teclado los valores que queremos introducir. Por desgracia en un código en ANSI C no se puede introducir por teclado ningún carácter que no esté en el estándar ASCII mostrado en la siguiente figura.

Cómo se puede ver en la tabla, el máximo valor representado en hexadecimal con el estándar ASCII es el `7F`, por lo que tendremos que limitarnos, por ahora, a zonas de memoria que contengan valores inferiores a `7F` (128).

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
00	00	NUL	43	2B	+	86	56	V
01	01	SOH	44	2C	,	87	57	W
02	02	STX	45	2D	-	88	58	X
03	03	ETX	46	2E	.	89	59	Y
04	04	EOT	47	2F	/	90	5ª	Z
05	05	ENQ	48	30	0	91	5B	[
06	06	ACK	49	31	1	92	5C	\
07	07	BEL	50	32	2	93	5D	]
08	08	BS	51	33	3	94	5E	^
09	09	HT	52	34	4	95	5F	_
10	0A	LF	53	35	5	96	60	`
11	0B	VT	54	36	6	97	61	a
12	0C	FF	55	37	7	98	62	b
13	0D	CR	56	38	8	99	63	c
14	0E	SO	57	39	9	100	64	d
15	0F	SI	58	3A	:	101	65	e
16	10	DEL	59	3B	;	102	66	f
17	11	DC1	60	3C	<	103	67	g
18	12	DC2	61	3D	=	104	68	h

Tabla estándar ASCII

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
19	13	DC3	62	3E	>	105	69	i
20	14	DC4	63	3F	?	106	6ª	j
21	15	NAK	64	40	@	107	6B	k
22	16	SYN	65	41	A	108	6C	l
23	17	ETB	66	42	B	109	6D	m
24	18	CAN	67	43	C	110	6E	n
25	19	EM	68	44	D	111	6F	o
26	1A	SUB	69	45	E	112	70	p
27	1B	ESC	70	46	F	113	71	q
28	1C	FS	71	47	G	114	72	r
29	1D	GS	72	48	H	115	73	s
30	1E	RS	73	49	I	116	74	t
31	1F	US	74	4A	J	117	75	u
32	20	(espa-cio)	75	4B	K	118	76	v
33	21	¡	76	4C	L	119	77	w
34	22	"	77	4D	M	120	78	x
35	23	#	78	4E	N	121	79	y
36	24	$	79	4F	O	122	7ª	z
37	25	%	80	50	P	123	7B	{
38	26	&	81	51	Q	124	7C	|
39	27	'	82	52	R	125	7D	}
40	28	(	83	53	S	126	7E	~
41	29	)	84	54	T	127	7F	DEL
42	2A	*	85	55	U			

Tabla estándar ASCII

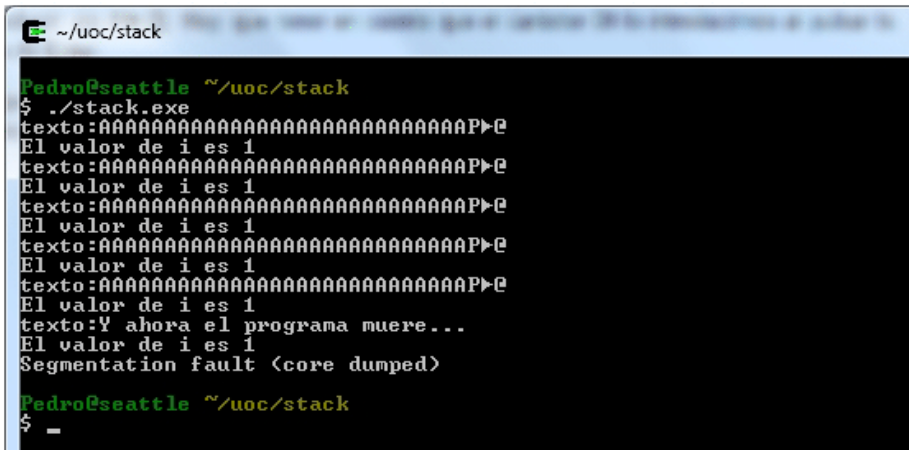
Con esta limitación en el direccionamiento podremos hacer, en este ejemplo y sin ver ninguna técnica más avanzada, que el programa se ejecute dos veces si hacemos que la dirección de retorno apunte a la dirección de memoria `00401050`, que es una dirección totalmente escribible con ASCII. Esto provocará que el programa se ejecute mientras introduzcamos direcciones válidas, hasta que no lo sobrescribamos o hasta que pongamos una dirección inválida y podamos comprobar cómo se cambia el flujo del programa con un *stack overflow*.

Para realizar esto vamos a introducir 28 caracteres, tantos como nos hacen falta en este ejemplo para llegar a colocarnos justo delante de la zona de memoria donde se guarda la dirección de retorno. Posteriormente vamos a introducir la dirección de memoria `00401050`.

Hay que recordar que en la memoria los datos se guardan al revés de cómo los hemos introducido nosotros, por lo que se deberá introducir la cadena `50104000` para que al final quede correctamente situada la dirección de memoria.

Consultando la tabla de valores en la figura 67, la dirección de memoria `00401050` se convierte en la cadena P►@. Hay que tener en cuenta que el carácter `00` lo introducimos al pulsar la tecla `Enter` mientras que los otros caracteres se introducirán pulsando Alt + el número correspondiente en decimal, usando para introducir el numero en cuestión el teclado numérico.

Si ejecutamos el programa desde la línea de comandos e introducimos 28 "`A`" y la cadena anteriormente especificada, obtendremos el siguiente resultado:



Sobrescritura recurrente de dirección de retorno

En la captura se puede ver cómo se ha hecho ejecutar el mismo programa seis veces hasta que le hemos introducido una dirección no válida en la dirección de retorno, lo que ha hecho que el programa produzca un fallo y se detenga.

Con esto queda demostrado que modificar la ejecución de un programa es posible mediante la introducción de una variable lo suficientemente grande y correctamente malformada si el código es vulnerable. Esto se tornara útil más adelante cuando se vea cómo lanzar un *shellcode* gracias a este tipo de vulnerabilidades.

3. *Heap overflow*

Las técnicas de *heap overflow* son quizá las más complicadas de explotar con éxito (aunque una denegación de servicios será siempre más fácil) debido a que depende más que ninguna de las técnicas anteriores acerca de las versiones concretas del software.

No nos referimos ahora sólo al software vulnerable que, evidentemente, sigue siendo el elemento más importante de esta ecuación, sino que aquí entran en juego, y mucho, el compilador usado y las librerías utilizadas en el proceso de compilación.

¿Y a qué es debido esto? Cuando un programa almacena datos en la pila, como ya vimos, es el compilador el que se encarga de reservar la memoria y organizarla. Cuando hablamos de *heap overflow*, es el propio compilador el que organiza las variables del programa que podremos llegar a sobrescribir de la forma que le parece más conveniente.

En el presente módulo veremos un ejemplo sencillo de su funcionamiento.

3.1. El *heap*

Antes de seguir discutiendo aspectos de la explotación de un *bug* de este tipo, vamos a recordar cómo funciona la memoria y en concreto el *heap*. El *heap* es una zona de memoria que usa el programa para almacenar variables inicializadas de tipo estático y para reservar bloques de memoria en tiempo de ejecución.

> Las cadenas de texto de un programa para los menús y diálogos localizados en distintos idiomas son un buen ejemplo de variable que se almacena en el *heap* de un programa.

Otra zona de memoria o segmento de datos muy similar es el *BSS*, que es una zona destinada a almacenar variables estáticas no inicializadas. En tiempo de ejecución estas variables son rellenadas con ceros hasta que se les asigna un nuevo valor. En el resto del módulo se explicarán los fallos relacionados con el *heap overflow*, siendo idénticos los ocurridos en el *BSS*.

El *heap* se reserva, como hemos dicho, al compilar un programa y es por eso que dependemos, más que nunca, del compilador usado y de las librerías instaladas. Vamos a generar y compilar un ejemplo para comprobar cómo es el funcionamiento.

```
#include <stdio.h>
#include <string.h>

int main()
{
   static char buffer[16] = "";
   static char nombre[] = "Pedro Laguna";

   gets(buffer);

   printf("Hola %s!", nombre);

   return 0;
}
```

Código vulnerable a *heap overflow*

Como se puede observar, el ejemplo anterior no difiere mucho de los anteriormente realizados, pero sí que hay una sutil diferencia. En este código, las variables se han declarado como *static*, por lo tanto, se usa la zona de *heap* para almacenar las variables porque se conoce su tamaño exacto.

Esta última afirmación no es del todo cierta, pues hemos generado una variable buffer de 16 bytes a la cual no hemos asignado ningún valor, algo que haremos en tiempo de ejecución mediante la función *gets*.

3.1.1. Ejemplo en Windows

La compilación y ejecución de este programa nos determinará algunos aspectos acerca de cómo se está reservando la memoria nuestro compilador. Vamos a empezar por ejecutar el código anterior en nuestro entorno habitual de pruebas, Cygwin sobre entorno Windows con un compilador gcc 3.4.4. Posteriormente haremos lo mismo pero sobre una máquina con Ubuntu 8.10 y compilador gcc 4.3.2.

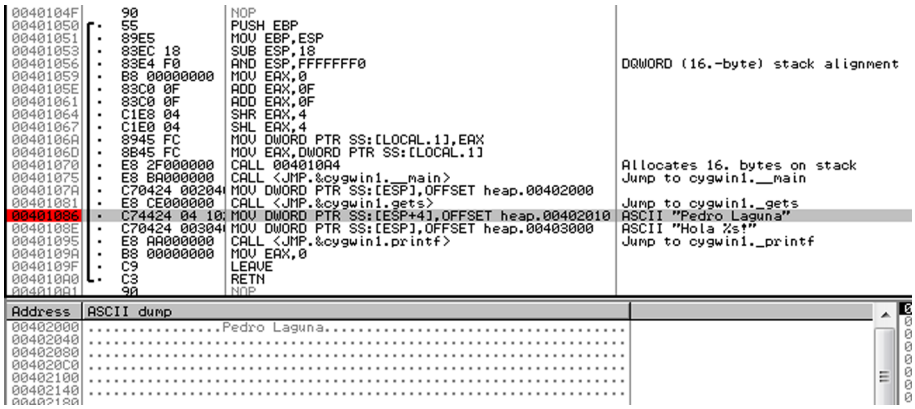
```
Pedro@seattle ~/uoc/heap
$ cat heap.c | grep -n "static"
6:      static char buffer[16] = "";
7:      static char nombre[] = "Pedro Laguna";

Pedro@seattle ~/uoc/heap
$ ./heap.exe
Texto pequeño
Hola Pedro Laguna!
Pedro@seattle ~/uoc/heap
$ ./heap.exe
Texto bastante mas grande
Hola as grande!
Pedro@seattle ~/uoc/heap
$ ./heap.exe
0123456789012345Chema Alonso
Hola Chema Alonso!
Pedro@seattle ~/uoc/heap
$ _
```

Ejecución del programa heap.c compilado por gcc 3.4.4

Como se ve en la figura anterior, tenemos un programa compilado donde la variable buffer está declarada antes que la variable nombre (esto se ve gracias al parámetro `-n` de `grep`, que nos muestra el número de línea donde ha encontrado la palabra `static`). En este caso nuestro compilador está usando

una manera secuencial de almacenar las variables que se declaran en nuestro código; esto quiere decir que primero reservará 16 bytes para la variable buffer y posteriormente almacenará el contenido de la variable `nombre`.

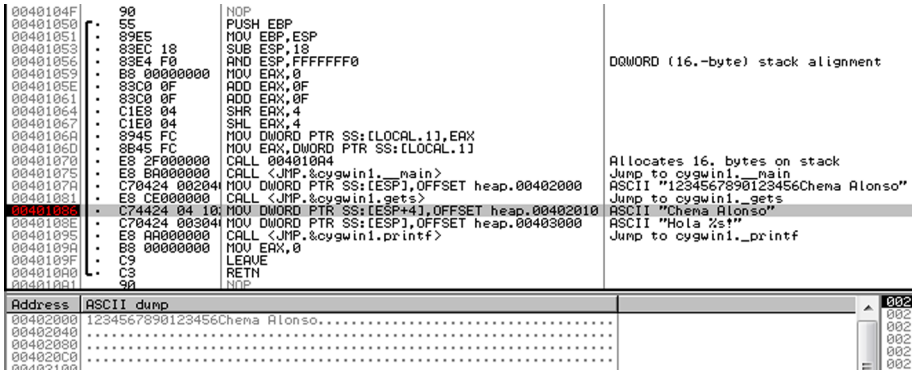
¿Y cómo ocurre esto visto a ojos de un *debugger* como Ollydbg? Si arrancamos el programa desde este *debugger* obtendremos un volcado similar a este:



Programa `heap.c` sin el `heap` modificado

Como se puede observar en la línea resaltada, debido a que tiene un *breakpoint*, el programa accede una posición de memoria fija, la `00402010`, donde espera encontrar el valor de la variable. Si observamos detenidamente veremos que la memoria empieza en la dirección `00402000`. Si sumamos 16 (`0x10`) al valor de la variable `buffer`, se podrá llegar hasta la posición de la variable `nombre`.

Si al ejecutar este programa establecemos como buffer la cadena `1234567890123456Chema Alonso` veremos cómo se sobrescribe la memoria convenientemente para situar en la posición deseada el valor introducido.

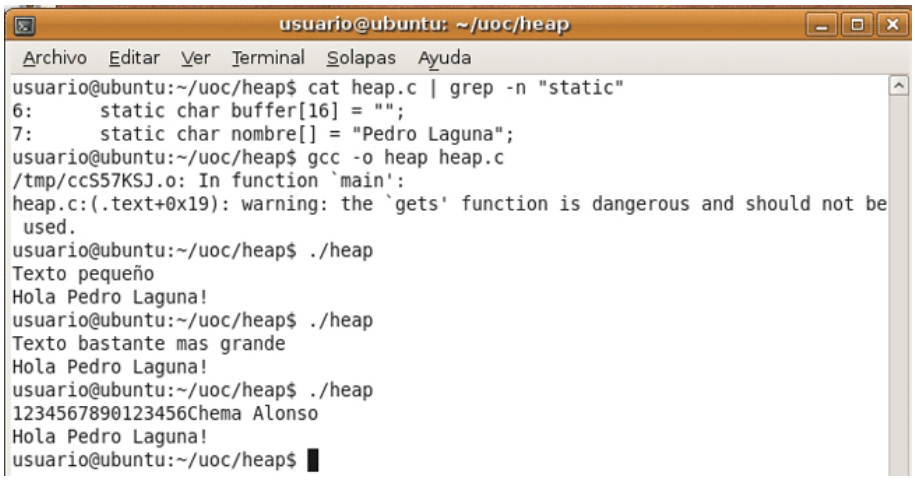


Programa `heap.c` después de la modificación del `heap`

Como se puede observar en la línea 00401086, el valor al que se referencia ahora se ha modificado y contiene actualmente el nombre `Chema Alonso`.

3.1.2. Ejemplo en Linux

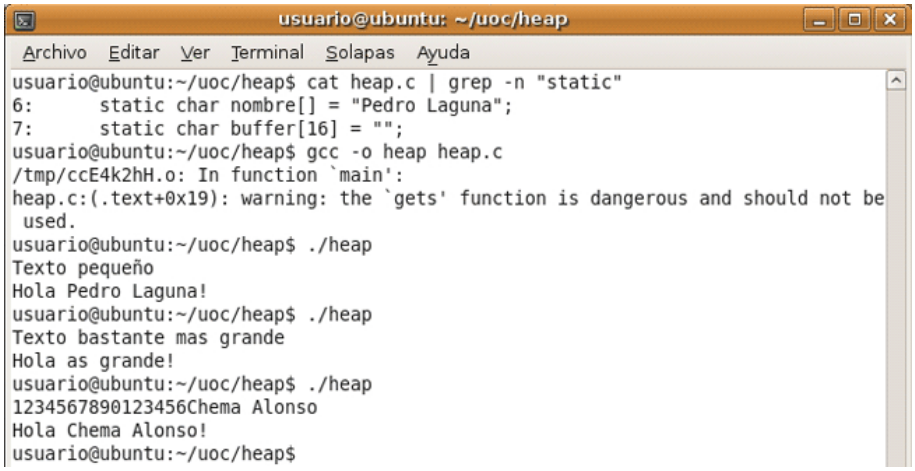
Vamos a cambiar ahora de escenario. Ejecutaremos Ubuntu 8.10 con un compilador gcc 4.3.2 y veremos las sutiles diferencias que acompañan al cambio de versión del compilador. Para comenzar, se va a proceder a ejecutar exactamente el mismo programa que en el ejemplo anterior.



Ejecución de `heap.c` en un sistema Linux

En este ejemplo que se observa en la figura anterior hemos querido incluir la línea de la compilación del programa para que se vea cómo el propio gcc nos advierte que el uso de la función `gets()` puede dar lugar a algún tipo de error y que deberíamos de evitarla. De todas maneras, y a pesar de usar una función no recomendada, nuestro programa parece seguro, no existe desbordamiento de la variable `buffer` que nos permita modificar el valor de la variable `nombre`.

¿Es esto cierto? Pues, lamentablemente, no. En gcc 4.3.2 las variables en el heap se organizan de manera inversa, esto es, primero se almacenaría el valor del nombre y posteriormente el del `buffer`, por lo que nunca podríamos llegar a sobrescribir su valor. Esto es, si cambiamos el orden en el que se declaran las variables, gcc, al compilar el código, les reservará la memoria de tal manera que la variable buffer quede delante de la variable nombre, por lo que podremos volver a sobrescribir su valor al igual que hicimos en el primer ejemplo.

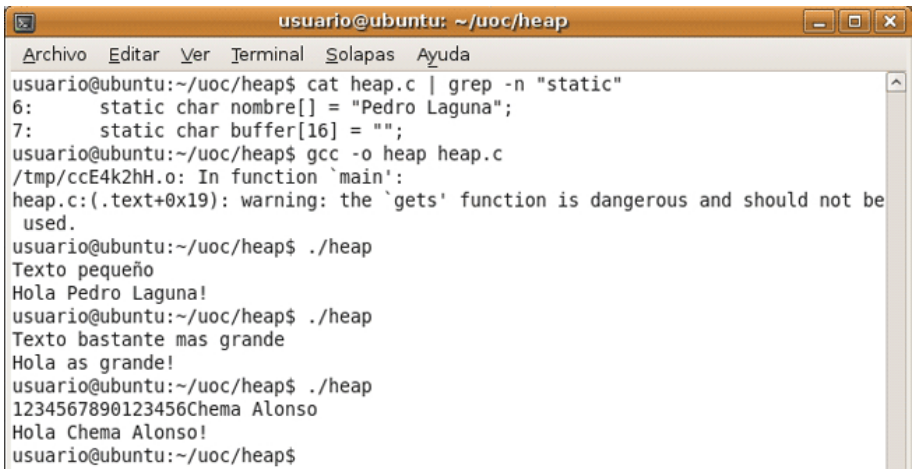


Ejecución del programa heap.c modificando el orden de las variables

De nuevo nuestro código vuelve a ser vulnerable, permitiendo la sobrescritura de la variable nombre como se ve en la figura anterior.

Para analizar estos cambios que se producen en Linux vamos a volver a usar el programa gdb y a aprender algunos trucos nuevos. Para ello es necesario que compilemos el programa `heap.c` con la opción `-g`, lo que generará los símbolos de *debug* que gdb entenderá, interpretará y usará. La manera de hacer esto ya se ha visto en módulos anteriores.

Mediante gdb disponemos de comandos para localizar las variables en memoria, por ejemplo mediante *info scope main*, que nos devolverá las variables declaradas dentro del método `main`. Con este comando podemos saber exactamente dónde localizar los valores que buscamos sin tener que recorrer la memoria en busca de una cadena conocida.



Realizando un análisis de las variables en memoria mediante gdb

En la figura anterior hemos realizado un análisis mediante gdb donde hemos localizado las posiciones de memoria de las variables `nombre` y `buffer` y, como se puede observar, la posición de buffer es menor a la de nombre. El volcado siguiente de la zona de memoria donde se encuentran lsd podivionrd de-

muestra inequívocamente cómo el crecimiento (recordemos, al contrario que en el *stack*) excesivo de la variable buffer puede dar lugar a una sobrescritura de la variable nombre.