Código seguro

Josep Vañó Chic

PID_00217348

Universitat Oberta de Catalunya

Índice

Introducción

Un software que funciona correctamente es aquel que hace exactamente todo aquello para lo que fue creado y diseñado. Sin embargo, el programa puede ser correcto desde el punto de vista funcional pero a la vez puede ser inseguro.

Los errores en el software pueden ser utilizados para atacar el sistema y poner en peligro su buen funcionamiento, así como la confidencialidad y el uso de los datos que hay almacenados, además, los errores pueden ser utilizados como puerta de entrada para ejecutar código malicioso.

El software realizado con código de programación no seguro es fácilmente vulnerable. Los *hackers* aprovechan estas vulnerabilidades para provocar el error y entrar en el sistema del ordenador atacado.

En este módulo se podrá constatar el cómo, el por qué y las implicaciones que comporta el código no seguro. Al mismo tiempo, se muestra el uso de herramientas para inspeccionar el funcionamiento del software durante su ejecución y así observar en qué momento es vulnerable y cuáles son los motivos de esta vulnerabilidad.

Así pues, es importante ser conscientes de los peligros que pueden comportar estos errores para poner los medios adecuados para que no se produzcan.

Objetivos

Al finalizar la lectura de este material, los estudiantes habrán conseguido las competencias siguientes:

1. Conocer el riesgo de la programación de código no seguro.
2. Conocer los principales tipos de código no seguro.
3. Conocer cómo y por qué se producen los errores de ***Integer Overflow***.
4. Conocer cómo y por qué se producen los errores de ***Buffer Overflow***.
5. Saber identificar la diferencia entre los diversos tipos de ***Buffer Overflow***.
6. Conocer las funciones no seguras.
7. Conocer las implicaciones que comporta la programación no segura.
8. Utilizar las herramientas de depuración y desensamblado para inspeccionar el funcionamiento de software.

1. *Integer Overflow*

El ***Integer Overflow*** sucede cuando una operación aritmética intenta crear un valor numérico que es demasiado grande para ser representado en el espacio de almacenamiento que tiene asignado.

En programación, una variable es un espacio de memoria reservado para almacenar un valor que corresponde a un tipo de datos soportado por el lenguaje de programación.

Los lenguajes de programación disponen de varios tipos de variables, y la medida del espacio de memoria reservado para la variable irá en función del tipo de variable que se defina.

Por ejemplo, en ANSI C, las variables y sus medidas son:

Tipo	Rango de valores	Medida
char	De -128 a 127	8 bits
unsigned char	De 0 a 255	8 bits
short	De -32.768 a 32.767	16 bits
unsigned short	De 0 a 65.535	16 bits
int	De -2.147.483.648 a 2.147.483.647	32 bits

A continuación se muestra un programa en C donde se reflejan las medidas de estos tipos de datos. Hay que tener en cuenta que el resultado puede variar de un ordenador a otro dependiendo de la versión del compilador que se utilice y la arquitectura del equipo. Este ejemplo ha sido realizado en un entorno virtualizado en ***Oracle VM VirtualBox***, utilizando el compilador **Cygwin GCC** a través del IDE de ***Code Blocks*** sobre un sistema operativo Windows 7 Profesional de 32 bits.

Este programa muestra el rango de valores mínimos y máximos de varios tipos de variables que se definen; al mismo tiempo, el programa muestra su representación en hexadecimal y su medida en bits.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
     char c =  127;              // 2^7 -1
     unsigned char uc = 255;     // 2^8 -1
```

```
    short s =  32767;           // 2^15 -1
    unsigned short us = 65535;  // 2^16 -1
    int i = 2147483647;         // 2^31 -1
    printf("Maximum values\n");
    printf("char           = %d (0x%x) [%d bits]\n", c, c, sizeof(c) * 8);
    printf("unsigned char  = %d (0x%x) [%d bits]\n", uc, uc, sizeof(uc) * 8);
    printf("short          = %d (0x%x) [%d bits]\n", s, s, sizeof(s) * 8);
    printf("unsigned short = %d (0x%x) [%d bits]\n", us, us, sizeof(us) * 8);
    printf("int            = %d (0x%x) [%d bits]\n", i, i, sizeof(i) * 8);

    c =  -128;        // -2^7
    s =  -32768;      // -2^15
    i = -2147483648;  // -2^31
    printf("\nMinimum values\n");
    printf("char           = %d (0x%x) [%d bits]\n", c, c, sizeof(c) * 8);
    printf("short          = %d (0x%x) [%d bits]\n", s, s, sizeof(s) * 8);
    printf("int            = %d (0x%x) [%d bits]\n", i, i, sizeof(i) * 8);

    return 0;
}
```

sizes.c

El resultado de la ejecución del programa es el siguiente:

```
Maximum values
char           = 127 (0x7f) [8 bits]
unsigned char  = 255 (0xff) [8 bits]
short          = 32767 (0x7fff) [16 bits]
unsigned short = 65535 (0xffff) [16 bits]
int            = 2147483647 (0x7fffffff) [32 bits]

Minimum values
char           = -128 (0xffffff80) [8 bits]
unsigned char  = 0 (0x0) [8 bits]
short          = -32768 (0xffff8000) [16 bits]
unsigned short = 0 (0x0) [16 bits]
int            = -2147483648 (0x80000000) [32 bits]
```

1.1. Representación de los números

Como se puede observar en el ejemplo anterior, en la salida del programa la diferencia entre ***signed*** y ***unsigned*** es que el bit más significativo –conocido como MSB (***most significant bit***)– en las variables ***signed*** es 0 en los valores positivos (7 en hexadecimal es 0111 en binario), en cambio tiene el valor 1 en los valores negativos (f en hexadecimal es 1111 en binario).

Para poner un ejemplo, el tipo `signed char`, a pesar de ser de 8 bits, como el ***MSB*** identifica el signo, solo se pueden utilizar 7 bits para representar el valor. Teniendo en cuenta la representación en binario de complemento ***a 2***, el rango de enteros representables en ***n*** bits es [-2^{n-1} , $2^{n-1} - 1$], por lo tanto, en el caso del tipo `char` el rango de valores es de -2^7 a $2^7 - 1$, es decir, de -128 a 127. Dicho de otra forma, hay $2^7 = 128$ posibles valores para cada signo, teniendo en cuenta que en C2 (***complemento a 2***) el cero se considera un valor positivo, el rango de valores positivos va del 0 hasta el 127 y el de los negativos del -1 al -128.

Complemento a 2

El formato de complemento a 2^1 (Can2 o C2), es un sistema de representación de números con signo en base 2.

Los números positivos en Ca2 se codifican del mismo modo que en signo y magnitud. El bit MSB es 0, para indicar signo positivo, y el resto contiene la magnitud.

La codificación de los números negativos se obtiene a partir de la operación en binario de 2^n - |X| en base 2, donde |X| es el valor absoluto de ***X***.

(1)**Two's Complement**: http://en.wikipedia.org/wiki/two%27s_complemento.

Teniendo en cuenta estos conceptos, en positivo, el valor 127 tiene una representación en binario de 0111 1111.

Para representar un valor en negativo, por ejemplo el valor -26, se realiza la operación siguiente:

$2^8 - |-26| = 100000000_{(2} - 11010_{(2} = 11100110$

1.2. Desbordamiento de tipo de dato

A pesar de que los tipos `char, int y short` tienen unos rangos de valores determinados, si se asignan unos valores superiores a los valores admitidos, la ejecución del programa no dará ningún error, sino que truncará los valores.

En el ejemplo siguiente, en el programa se asignan valores fuera de los rangos de válidos a las variables.

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

int main()
{
     char c =  129;
     unsigned char uc = 257;
     short s =  32769;
     unsigned short us = 65537;
     int i = 2147483649;
     printf("Maximum values\n");
```

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

    c =  -130;
    uc = -2;
    s =  -32770;
    us = -2;
    i = -2147483650;
    printf("\nMinimum values\n");
    printf("char           = %d (0x%x) [%d bits]\n", c, c, sizeof(c) * 8);
    printf("unsigned char  = %d (0x%x) [%d bits]\n", uc, uc, sizeof(uc) * 8);
    printf("short          = %d (0x%x) [%d bits]\n", s, s, sizeof(s) * 8);
    printf("unsigned short = %d (0x%x) [%d bits]\n", us, us, sizeof(us) * 8);
    printf("int            = %d (0x%x) [%d bits]\n", i, i, sizeof(i) * 8);

    return 0;
}
```

sizes2.c

El resultado de la ejecución del programa es el siguiente:

```
Maximum values
char           = -127 (0xffffff81) [8 bits]
unsigned char  = 1 (0x1) [8 bits]
short          = -32767 (0xffff8001) [16 bits]
unsigned short = 1 (0x1) [16 bits]
int            = -2147483647 (0x80000001) [32 bits]

Minimum values
char           = 126 (0x7e) [8 bits]
unsigned char  = 254 (0xfe) [8 bits]
short          = 32766 (0x7ffe) [16 bits]
unsigned short = 65534 (0xfffe) [16 bits]
int            = 2147483646 (0x7ffffffe) [32 bits]
```

Hay que observar que los valores que se muestran en la ejecución del programa no son los valores que se han asignado a las variables. El motivo de este hecho es que los valores que se han asignado están fuera del rango admitido según la definición de cada tipo de variable.

Los próximos ejemplos se basan en los resultados del tipo de variable `char`, pero son extrapolables al resto de tipo de variables de enteros.

¿Por qué el valor 129 asignado a una variable de tipo `char` no ha dado ningún error de fuera de rango u ***overflow*** pero en cambio ha mostrado un valor de -127?

El valor 129 tiene una equivalencia en binario: 10000001. Pero siguiendo la representación en C2, el hecho de que el bit más significativo (MSB), es decir, el bit de la izquierda, sea un 1 indica que se trata de un número negativo. Ahora hay que tener en cuenta que la representación de un número negativo en binario y en C2 varía respecto de si se trata de un número positivo o de si se trata de un número negativo.

Anteriormente se ha mostrado cómo obtener, a partir de un número negativo en decimal, su representación en binario en el formato de complemento ***a 2*** (C2). A continuación se muestra el paso a la inversa, es decir, a partir de un número negativo en binario en formato C2, su equivalente en el sistema decimal.

Siguiendo el TFN (**Teorema fundamental de numeración**), para obtener el valor en decimal del valor en binario de 10000001 teniendo en cuenta el formato C2, hay que aplicar el TFN como en el caso positivo, pero considerando que el bit más significativo es negativo:

Enlace recomendado

Sobre el teorema fundamental de numeración:

http://es.wikipedia.org/wiki/Sistema_de_ numeraci%C3%B3n

http://electronicamarti.files.wordpress.com/2010/01/sistemas-de-numeracion.pdf

$$-1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = -127$$

¿Cuál hubiera sido el valor presentado si se hubiera asignado el valor 1524 a la variable de tipo `char`?

La representación de $1524_{(10}$ en binario es 10111110100.

Como se trata de una variable de tipo `char` y por lo tanto de 8 bits, el computador trunca este valor y solo interpreta los 8 primeros bits menos significativos, es decir, 11110100. Se trunca el valor a la altura del máximo número de bits que pueda contener el tipo de dato.

A partir de este valor, hay que realizar la misma operación para obtener el número en decimal a partir de una representación binaria en C2.

$$-1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = -12$$

Así pues, en este caso, si se asigna el valor 1524 a una variable de tipo `char`, en realidad el valor asignado es: -12.

En el próximo caso, se observa cuál será el resultado de asignar el valor 1031 a una variable de tipo `char`.

- La representación en binario del valor $1031_{(10}$ es 10000000111.

- Se cogen los 8 bits menos significativos, es decir, los 8 primeros empezando por la derecha: 00000111.

- El bit más significativo (el primero por la izquierda) es 0; por lo tanto, se trata de un número positivo.

- Como se trata de un número binario en positivo, su conversión en decimal es : $0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 7$.

En este caso, si se asigna el valor 1031 a una variable de tipo `char`, en realidad el valor asignado es: 7.

1.3. Ataques *integer overflow*

La norma ISO C99 dice que un ***integer overflow*** causa "***undefined behaviour***", lo que significa que los compiladores compatibles con el estándar pueden hacer lo que quieran, desde ignorar completamente el desbordamiento a abortar el programa. Lo que hacen la mayoría de los compiladores es ignorar el ***integer overflow***.

Los ***integer overflow*** no pueden ser detectados hasta que hayan ocurrido. Esto puede ser peligroso si el cálculo tiene que ver con la medida de un ***buffer*** o el índice de un ***array***. La mayoría de los ***integer overflow*** no son explotables porque la memoria no está siendo directamente sobrescrita, pero a veces pueden conducir a otras clases de ***bugs***, frecuentemente de ***buffer overflow***.

Los ataques de ***integer overflow*** no permitirán sobrescribir zonas de memoria, variables o código, pero sí cambiar la lógica de la aplicación e incluso desbordar estructuras de memoria creadas por medio de variables inseguras.

Así pues, se puede observar que si se asignan valores fuera de rango a variables de tipo entero (`char, short, int`), el resultado puede ser inesperado dado que el valor resultante no será el previsto según la lógica definida en el programa, como sucede con el ejemplo siguiente:

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

int main(int argc, char *argv[]){
   int i, j;
   char c;
   int result;
   if (argc == 4) {
      i = atoi(argv[1]);
      j = atoi(argv[2]);
```

```
        c = atoi(argv[3]);
        if (i<=0 || j<=0 || c<=0) {
           printf("Invalid values\n");
        }
        else {
           result = i + j + c;
           printf("i: %d | j: %d | c: %d | result: %d\n", i, j, c, result);
           if (result == 0) {
              printf("Protected area\n");
              }
        }
     }
     else {
           printf("Three parametres are needed\n");
           printf("Sample1 param1 param2 param3\n");
     }
     return(0);
     }
```

protectedArea.c

En este ejemplo se puede observar que no se permite introducir valores negativos en los parámetros y después se realiza la suma de los tres parámetros. Esto hace pensar que la suma de los tres parámetros no puede dar como resultado cero, y por lo tanto, el programa no mostrará por pantalla la frase "Protected area". Pero ¿y si se introducen los valores siguientes?:

Variable	Valor en decimal	Valor en binario
i	2147483647	01111111 11111111 11111111 11111111
j	2147483647	01111111 11111111 11111111 11111111
c	2	00000010

En este caso la suma de y + j = 4294967294. Pero este valor produce un ***integer overflow*** dado que supera el valor máximo admitido en una variable de tipo `int`.

La representación de 4294967294 en binario es: 11111111 11111111 11111111 11111110

Pero al ser tratado en formato de complemento ***a 2***, se trata del número negativo –2 y por lo tanto, el resultado de y + j + c = 0.

Así pues, el resultado de la ejecución del programa es el siguiente:

```
C:\pcs>ProtectedArea 2147483647 2147483647 2


i: 2147483647 | j: 2147483647 | c: 2 | result: 0
Protected area
```

Otros valores posibles para conseguir lo mismo son los siguientes:

Se trata de que la suma de dos números positivos desborde la capacidad de la variable de resultado (***integer overflow***), obteniendo un número negativo.

Valor de la variable c:	+127	01111111
Valor de la suma parcial i + j	-127	11111111 11111111 11111111 10000001

Si se considera 11111111 11111111 11111111 10000001 como un número positivo, en lugar de -127 se obtiene 4294967169.

Ahora se trata de hacer que c = 127 y que la suma de i + j = 4294967169, por ejemplo i = 2147483647 j = 2147483522.

```
C:\pcs>ProtectedArea 2147483647 2147483522 127
i: 2147483647 | j: 2147483522 | c: 127 | result: 0
Protected area
```

Para evitar que se produzca un ***integer overflow***, la comprobación de los valores numéricos tiene que ser exhaustiva para que no se produzcan errores inesperados. Por ejemplo, en el código anterior, la solución habría implicado incluir una comprobación para detectar si los valores introducidos están entre un rango de valores determinado y, por supuesto, comprobar la medida del tipo de datos antes de empezar a utilizarla.

2. Desbordamiento de pila (*stack overvlow*)

Enlace recomendado

CWECommon Weakness Enumeration

CWE-121: *Stack-based buffer overflow*: http://cwe.mitre.org/data/definitions/121.html

La memoria tiene una zona dedicada a las variables del programa que se divide en dos: la pila o ***stack*** y el ***heap*** o zona de memoria dinámica. La zona de pila tiene un crecimiento de arriba abajo en cuanto a posiciones de memoria en forma de LIFO (***Last in, First out***). Cuando un programa hace una llamada a una función se crea un nuevo ***stack frame***, que se utiliza para pasar argumentos a los procedimientos y funciones y para almacenar las variables locales, y se va reservando memoria a medida que el programa va definiendo variables, almacenándolas en formato ***little endian***, es decir, con el bit menos significativo a la izquierda.

Además, las direcciones de retorno de las llamadas a las funciones también se almacenan en la pila y por eso, en ocasiones, esta se desborda, dado que si la medida del valor de una variable en una función es superior a la medida del espacio que se le ha reservado en la pila, puede sobrescribir la dirección de retorno de la función, lo que podría permitir a un usuario malicioso ejecutar cualquier código que quisiese. Este desbordamiento puede ser provocado y hacer de forma intencionada que el valor que se sobrescribe en la dirección de retorno sea una dirección escogida por el atacante; es decir, el retorno de la llamada no se realizaría en la dirección prevista por la ejecución del programa, sino en una dirección escogida por el atacante, y así podría ejecutar código con finalidades maliciosas.

Los ataques por ***buffer overflow***, tanto el ***stack overflow*** como el ***heap overflow***, son muy utilizados por los *hackers*, porque permiten aprovechar la vulnerabilidad para ejecutar su propio código y a la vez realizar ataques al resto del sistema; así pues, hay que tener mucho cuidado a la hora de escribir el código de programación para que no se puedan producir estos desbordamientos y así evitar ser vulnerables a los ataques.

2.1. Los registros

Dentro de la arquitectura x86, en un mismo lenguaje de programación, por ejemplo en lenguaje ANSI C, se pueden generar diferentes códigos en ensamblador dependiente del compilador, que a la vez pueden generar varias maneras de gestionar los registros del sistema. Aun así, hay una serie de registros e instrucciones bastante comunes en la mayoría de los compiladores que se utilizan. Entre todos los registros, destacaremos los tres siguientes:

- **EIP (*extended instruction pointer*)**. Contiene la dirección de la próxima instrucción que hay que ejecutar. Cuando la función A llama a la función B, la siguiente dirección que tenemos que ejecutar una vez se retorna de la función B se almacena en la pila. Cuando retorna la función B, la CPU

recoge la dirección de la pila y la almacena en el registro EIP. La dirección que hay en el registro EIP determina en qué dirección está el código en el que tiene que continuar la ejecución del programa.

- **ESP (*extended stack pointer*)**. Contiene la dirección que apunta al valor superior de la pila, es decir, a la cabeza de la pila. Cabe tener en cuenta que la pila crece de manera invertida, es decir, cada vez que la pila crece, la dirección de memoria decrece.

- **EBP (*extended base pointer*)**. Contiene la dirección de memoria donde empieza la pila. Como crece de forma invertida, es la dirección más grande 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 apunta a la dirección base de la pila y el registro ESP apunta al principio de la pila.

2.2. Gestión de la pila

El ***stack frame*** es la zona de memoria situada entre el `EBP` y el `ESP` y marca la zona de memoria de la función asignada a la pila.

Cuando se hace 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 sacar este valor de la pila. En este caso, la pila se desplaza a la medida del registro extraído, de manera que `ESP` quedará asignado a ESP +4 [en arquitecturas de 32 bits].

Cuando se hace una llamada a la función `PUSH` con un registro o valor como parámetro, este se pondrá 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.

2.3. Llamada y retorno de funciones

Cuando se produce la llamada a una función, hay tres funciones importantes que también afectan a los registros y a la pila:

- **`CALL`**. Cuando se produce una llamada a otra función, es necesario hacer 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 instrucción siguiente para ejecutar después de que el control de programa retorne 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

en 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 abandona 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` en el registro `EBP`, es decir, restaura el valor original de EBP. Este valor de `EBP` se guarda en el comienzo de la función. Esta situación deja el registro `ESP` apuntando a la dirección de retorno `CALL`, es decir, a la instrucción siguiente para ejecutar después de 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`.

2.4. Ejemplo de desbordamiento de pila

En este ejemplo se presenta un programa en lenguaje C vulnerable a un desbordamiento de pila (***stack overflow)***. En este caso, si se introduce como parámetro una cadena de una determinada longitud, se podrá sobrescribir la dirección de retorno de su ***stack frame***.

Entorno de desarrollo

- Sistema operativo: Windows 7 Professional SP1 32 bits funcionando en máquina virtual.

- Virtualización: Oracle VM VirtualBox 4.3.12.

- Compilador C: CodeBlocks 13.12 with GNU GCC Compiler

- Desensamblador y depurador: Ollydbg 2.01 Beta 2.

Este programa recibe como parámetro una dirección de email e imprime el dominio al que pertenece la dirección del email. El formato es el siguiente: acount@domain.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char** argv) {
    char *domain;
    char email[40];

    if (argc != 2){
        printf("Usage: obtain email address\n");
```

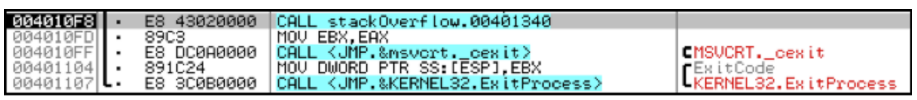
```
        return(-1);
    }
    strcpy(email, argv[1]);      //La función strcpy copia la dirección del email
                                 //introducida como parámetro (es decir,
                                 //argv[1]) en la variable local email.
    domain = strchr(email,'@');  //La función strchr busca el carácter '@'
                                 //en la cadena email, si lo encuentra, retorna un
                                 // puntero a esta posición, si no lo encuentra,
                                 //retorna NULL. El valor retornado se almacena
                                 //en la variable local domain.
    if (domain!= NULL){          // Si ha encontrado el carácter '@',
                                 // incrementa en una posición el puntero domain
        domain++;                //para que se salte el carácter @ y solo apunte al dominio.
        printf("Domain: %s", domain);
    }
    else{
        printf("Incorrect email");
    }
    return (0);
}
```

stackOverflow.c

El resultado de la ejecución del programa puede ser por ejemplo el siguiente:

```
C:\pcs>stackOverflow AAAABBBBCCCCDDDDEEEEFFFF@domainname.com
Domain: domainname.com
```

Durante el proceso de la ejecución, se puede observar, por ejemplo con la herramienta Ollydbg, que se realiza la llamada `CALL stackOver-vlow.00401340`, y en ese momento la dirección en ejecución es 004010F8. En este punto se hace una llamada (CALL) y, por lo tanto, el control del programa pasará a la dirección 00401340 para ejecutar un bloque de instrucciones hasta que se realice un retorno con la instrucción RETN. Entonces el control del programa volverá hacia la instrucción siguiente a la de la llamada, es decir, volverá el control del programa hacia la dirección 004010FD. Dicho de otra manera, se realiza una llamada a una función, y cuando finaliza la ejecución de esta función, el programa continúa la ejecución a partir de la siguiente instrucción que ha realizado la llamada.



En la dirección 0022FF2C de la pila se almacena la dirección de retorno que tendrá que utilizar para volver el control del programa a la siguiente instrucción posterior a la llamada CALL. Como se puede observar, esta dirección de retorno es 004010FD.

```
0022FF2C  004010FD  ²►@  RETURN from stackOverflow.00401340 to stackOverflow.004010FD
```

Una vez se realiza la llamada, el control del programa pasa a la función que se encuentra en el espacio comprendido desde la dirección 00401340 hasta la 004013CB, que es cuando se realizará el retorno a la siguiente instrucción de la llamada.

```
00401340  $  55              PUSH EBP
00401341  .  89E5            MOV EBP,ESP
00401343  .  83E4 F0         AND ESP,FFFFFFF0                    DQWORD (16.-byte) stack alignment
00401346  .  83EC 40         SUB ESP,40
00401349  .  E8 32060000     CALL stackOverflow.00401980
0040134E  .  837D 08 02      CMP DWORD PTR SS:[ARG.1],2
00401352  .v 74 13           JE SHORT stackOverflow.00401367
00401354  .  C70424 24304(   MOV DWORD PTR SS:[LOCAL.16],OFFSET stac  string => "Usage: obtain email address"
0040135B  .  E8 90080000     CALL <JMP.&msvcrt.puts>                 MSVCRT.puts
00401360  .  B8 FFFFFFFF     MOV EAX,-1
00401365  .v EB 63           JMP SHORT stackOverflow.004013CA
00401367  >  8B45 0C         MOV EAX,DWORD PTR SS:[ARG.2]
0040136A  .  83C0 04         ADD EAX,4
0040136D  .  8B00            MOV EAX,DWORD PTR DS:[EAX]
0040136F  .  894424 04       MOV DWORD PTR SS:[LOCAL.15],EAX         src
00401373  .  8D4424 14       LEA EAX,[LOCAL.11]
00401377  .  890424          MOV DWORD PTR SS:[LOCAL.16],EAX         dest => OFFSET LOCAL.11
0040137A  .  E8 79080000     CALL <JMP.&msvcrt.strcpy>               MSVCRT._mbscpy
0040137F  .  C74424 04 40(   MOV DWORD PTR SS:[LOCAL.15],40          c => '@'
00401387  .  8D4424 14       LEA EAX,[LOCAL.11]
0040138B  .  890424          MOV DWORD PTR SS:[LOCAL.16],EAX         string => OFFSET LOCAL.11
0040138E  .  E8 6D080000     CALL <JMP.&msvcrt.strchr>               MSVCRT.strchr
00401393  .  894424 3C       MOV DWORD PTR SS:[LOCAL.1],EAX
00401397  .  837C24 3C 00    CMP DWORD PTR SS:[LOCAL.1],0
0040139C  .v 74 1B           JE SHORT stackOverflow.004013B9
0040139E  .  834424 3C 01    ADD DWORD PTR SS:[LOCAL.1],1
004013A3  .  8B4424 3C       MOV EAX,DWORD PTR SS:[LOCAL.1]
004013A7  .  894424 04       MOV DWORD PTR SS:[LOCAL.15],EAX         (%s) => [LOCAL.1]
004013AB  .  C70424 40304(   MOV DWORD PTR SS:[LOCAL.16],OFFSET stac  format => "Domain: %s"
004013B2  .  E8 51080000     CALL <JMP.&msvcrt.printf>               MSVCRT.printf
004013B7  .v EB 0C           JMP SHORT stackOverflow.004013C5
004013B9  >  C70424 4B304(   MOV DWORD PTR SS:[LOCAL.16],OFFSET stac  format => "Incorrect email"
004013C0  .  E8 43080000     CALL <JMP.&msvcrt.printf>               MSVCRT.printf
004013C5  >  B8 00000000     MOV EAX,0
004013CA  >  C9              LEAVE
004013CB  L. C3              RETN
```

En la pila se reserva el espacio para almacenar el valor de la variable ***email***. Este espacio se encuentra a partir de la dirección de la pila 0022FEF4, y tiene reservado un espacio para almacenar hasta 40 caracteres. En este ejemplo se ha introducido como parámetro el valor Aaaabbbbccccddddeeeeffff@domainname.com, en el cual se pueden observar las posiciones donde está almacenado en la pila durante la ejecución del programa.

```
0022FEE0  0022FEF4  ¶■"  ASCII "AAAABBBBCCCCDDDDEEEEFFFF@domainname.com"
0022FEE4  00000040  @
0022FEE8  0022FF08  ◘ "  ASCII "FFFF@domainname.com"
0022FEEC  0022FFC4  ─ "
0022FEF0  75CB8CD5  'ïru
0022FEF4  41414141  AAAA
0022FEF8  42424242  BBBB
0022FEFC  43434343  CCCC
0022FF00  44444444  DDDD
0022FF04  45454545  EEEE
0022FF08  46464646  FFFF
0022FF0C  6D6F6440  @dom
0022FF10  6E6E6961  ainn
0022FF14  2E656D61  ame.
0022FF18  006D6F63  com
0022FF1C  0022FF0D  ♪ "  ASCII "domainname.com"
0022FF20  0022FF28  ( "
0022FF24  75CA9E34  4x╩u  RETURN from msvcrt.75CA9E3E to msvcrt.75CA9E34
0022FF28  0022FF94  ö "
0022FF2C  004010FD  ²►@  RETURN from stackOverflow.00401340 to stackOverflow.004010FD
```

En este ejemplo el programa funciona sin ninguna incidencia anormal y finaliza mostrando el siguiente resultado:

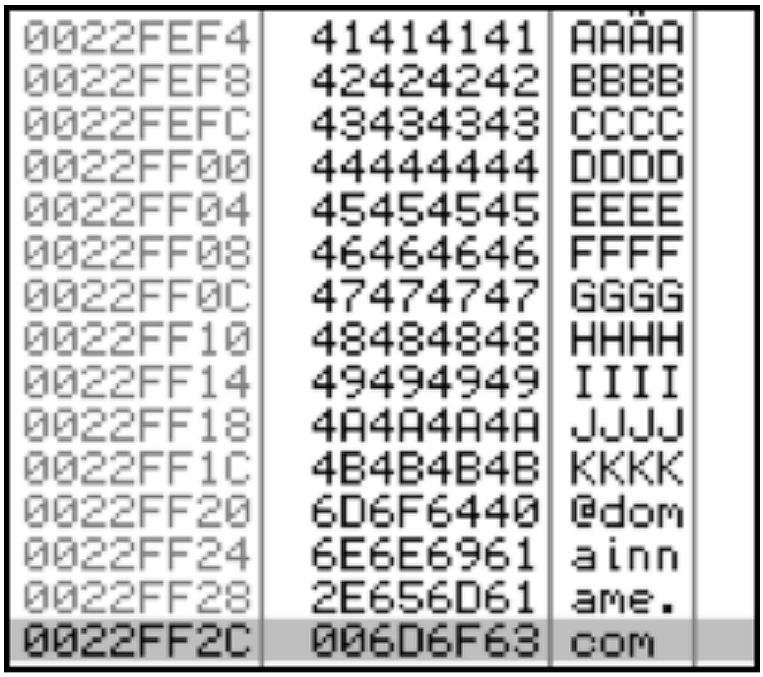
```
C:\pcs>stackOverflow AAAABBBBCCCCDDDDEEEEFFFF@domainname.com
```

```
Domain: domainname.com
```

El espacio reservado para la variable ***email*** es de 40 posiciones. ¿Qué pasaría si se introduce como parámetro un email con una longitud de más de 40 caracteres, como por ejemplo Aaaabbbbccccddddeeeeffffgggghhhhiiiijjjjkkkk@domainname.com?

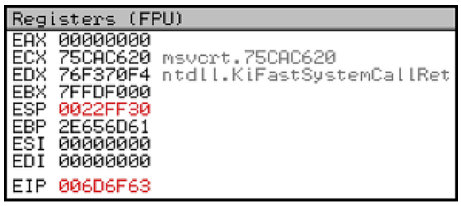
En el momento de hacer la llamada a la función, se guarda el valor de la dirección de retorno en la dirección 0022FF2C y, como se observa, la dirección de retorno es 004010FD.

Posteriormente, se almacena el valor del parámetro introducido en la pila, pero la función `strcpy` no controla si la medida de la información que tiene que almacenar en la pila es superior al espacio que hay reservado para la variable, en este caso, la variable ***email***. Este hace que se sobreescriba la posición 0022FF2C, que es donde estaba la dirección de retorno de la llamada `CALL stackOvervlow.00401340` que realiza el programa.

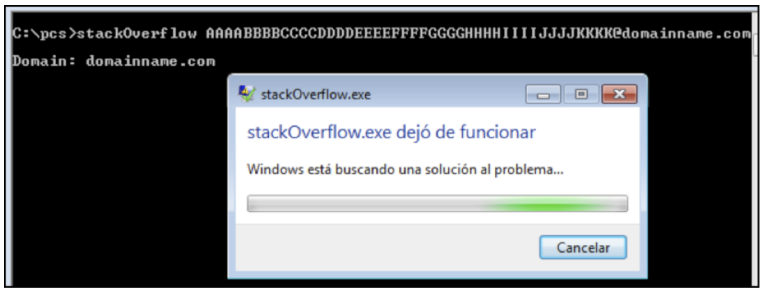


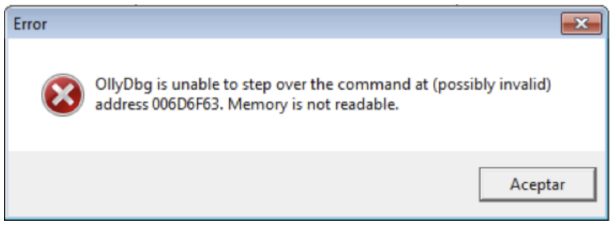
Cuando el programa ejecuta la instrucción RETN para retornar a la instrucción siguiente de la llamada, accede a la dirección 0022FF2C para obtener la dirección de retorno. Esta dirección de retorno tendría que ser 004010FD, pero en su lugar encuentra el valor 006D6F63.

En este punto, el EIP apunta a la dirección 006D6F63, que es donde tendría que estar la instrucción que se tiene que ejecutar al retornar después de la llamada a la función. Pero como esta dirección no era la prevista en el flujo del programa y además es ***not readable***, el sistema provocará la interrupción del programa y aparecerá un mensaje de error.



A continuación se muestra el mensaje motivado como consecuencia del desbordamiento de pila:





En este ejemplo el sistema ha mostrado un mensaje de error y el programa ha dejado de funcionar. Aun así, este desbordamiento de pila podía haber sido aprovechado por un ***hacker***. En caso de que se hubiera tratado de un ataque intencionado, se hubiera podido enviar un parámetro a través de un ***shellcode*** o un ***payload***, de forma que en la dirección de la pila 0022FF2C se hubiera sobrescrito un valor intencionado que correspondiera a una dirección de memoria, en la que el ***hacker*** hubiera inyectado algún código con las variables y así poderlo ejecutar.

En este ejemplo se ha mostrado que se puede sobrescribir la dirección de memoria por medio de un desbordamiento de las variables de la pila. Así pues, 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.

3. Desbordamiento de *heap*

El ***heap*** es un segmento de memoria que se utiliza para almacenar datos asignados dinámicamente en tiempos de ejecución.

Otra zona de memoria o segmento de datos muy similar es el ***BSS***, una zona destinada a almacenar variables globales sin inicializar, que también se asignan en tiempos de ejecución, y en este caso son rellenadas con ceros hasta que se les asigna un nuevo valor. Hay que tener en cuenta que los errores relacionados con el desbordamiento de *heap* son idénticos a los ocurridos en el ***BSS***.

Enlace recomendado

CWECommon Weakness Enumeration

CWE-122: *heap-based buffer overflow*: http://cwe.mitre.org/data/definitions/122.html

El desbordamiento de *heap* es un tipo de *buffer**overflow***, donde la memoria intermedia (***buffer***) que puede ser sobrescrita se asigna a la parte del ***heap*** de la memoria. Esto significa que, en general, este *buffer* ha sido asignado utilizando una instrucción de tipo `malloc ()`.

La función `malloc()` se utiliza para asignar un bloque de memoria en el ***heap***. El programa accede a este bloque de memoria vía un puntero que retorna la función `malloc()`.

Los desbordamientos de la memoria intermedia a menudo se pueden utilizar para ejecutar código arbitrario, que sucede en general fuera del ámbito de la política de seguridad implícita de un programa.

Además de los datos importantes del usuario, los ataques ***heap overflows*** se pueden utilizar para sobrescribir punteros de función, que pueden estar actualmente en la memoria, y apuntarlos hacia el código del atacante. Incluso en las aplicaciones que no utilizan explícitamente los punteros de función, el ***runtime*** suele dejar alguno en la memoria. Por ejemplo, los métodos de objetos en C++ se implementan generalmente usando los punteros en funciones, incluso en los programas en C a menudo hay una tabla de desplazamiento global que se utiliza en tiempo de ejecución subyacente.

El siguiente ejemplo es un programa sencillo que utiliza memoria ***heap***. El programa contiene un ***bug*** explotable de ***buffer overflow***.

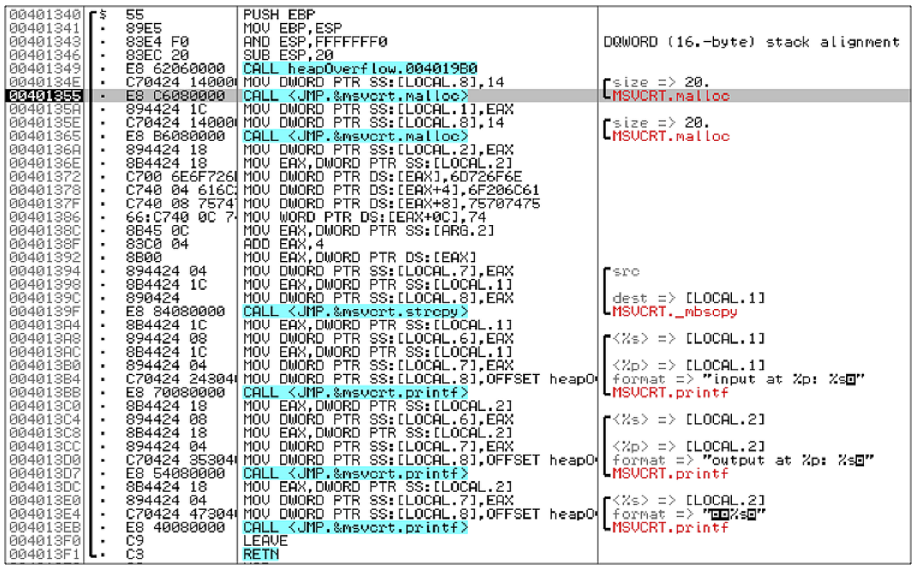
En el primer caso, el proceso del programa es normal y no se produce ningún ***heap overflow***, puesto que la longitud del parámetro de entrada no provoca ningún desbordamiento de memoria; en cambio, en el segundo caso se puede observar que al introducir un parámetro sobredimensionado, ha provocado un desbordamiento de ***heap***.

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

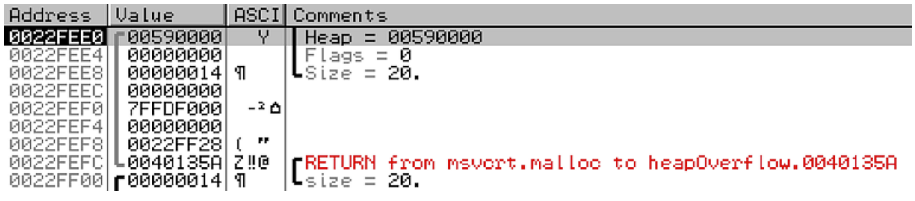
```
int main(int argc, char *argv[])
{
       char *input = malloc (20);
       char *output = malloc (20);

       strcpy (output, "normal output");
       strcpy (input, argv[1]);

       printf ("input at %p: %s\n", input, input);
       printf ("output at %p: %s\n", output, output);

       printf("\n\n%s\n", output);
}
```

heapOverflow.c

Mediante la herramienta Ollydbg se puede observar el proceso de la ejecución del programa.



Al hacer la llamada CALL la función `malloc()`, se ha creado un ***stack frame*** donde se puede observar el puntero hacia la dirección de la memoria ***heap***.



En esta imagen se puede observar el mapa de memoria y la ubicación del espacio reservado para el ***heap***.

Memory map

Address	Size	Owner	Section	Contains	Type	Access	Initial
00010000	00010000			Heap	Map	RW	RW
00020000	00010000			Heap	Map	RW	RW
0022D000	00001000				Priv	RW Gua	RW Gua
0022E000	00002000			Stack of main thread	Priv	RW	RW
00230000	00004000				Map	R	R
00240000	00001000				Priv	RW	RW
00290000	00004000			Default heap	Priv	RW	RW
00390000	00067000				Map	R	R
00400000	00001000	heapOverflow		PE header	Img	R	RWE Cop
00401000	00001000	heapOverflow	.text	Code	Img	R E	RWE Cop
00402000	00001000	heapOverflow	.data	Data	Img	RW Cop	RWE Cop
00403000	00001000	heapOverflow	.rdata		Img	R	RWE Cop
00404000	00001000	heapOverflow	/4		Img	R	RWE Cop
00405000	00001000	heapOverflow	.bss		Img	RW Cop	RWE Cop
00406000	00001000	heapOverflow	.idata	Imports	Img	RW Cop	RWE Cop
00407000	00001000	heapOverflow	.CRT		Img	RW Cop	RWE Cop
00408000	00001000	heapOverflow	.tls		Img	RW Cop	RWE Cop
00409000	00001000	heapOverflow	/14		Img	R	RWE Cop
0040A000	00001000	heapOverflow	/29		Img	R	RWE Cop
0040B000	00001000	heapOverflow	/41		Img	R	RWE Cop
0040C000	00001000	heapOverflow	/55		Img	R	RWE Cop
00590000	00003000			Heap	Priv	RW	RW

En este caso, el programa finaliza sin ninguna incidencia.

```
input at 00591000: heap_program
output at 00591068: normal output


normal output
```

En el próximo caso, la entrada del parámetro sobrepasa el espacio de memoria reservado. Además, la función `strcpy()` no tiene ningún control en lo referente a su medida.

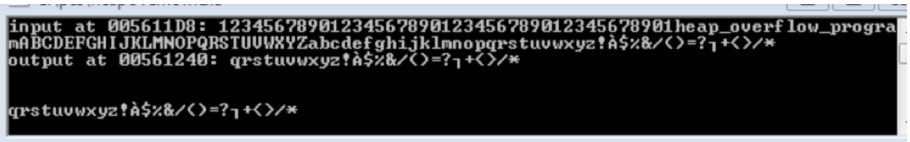
En este punto se puede observar el contenido de la pila durante la ejecución de la función `malloc()`.

Address	Value	ASCI	Comments
0022FEE0	00560000	V	Heap = 00560000
0022FEE4	00000000		Flags = 0
0022FEE8	00000014	¶	Size = 20.
0022FEEC	00000000		
0022FEF0	7FFD9000		
0022FEF4	00000000		
0022FEF8	0022FF28	("	
0022FEFC	0040135A	Z‼@	RETURN from msvcrt.malloc to heapOverflow.0040135A
0022FF00	00000014	¶	size = 20.

Memory map

Address	Size	Owner	Section	Contains	Type	Access	Initial
00010000	00010000			Heap	Map	RW	RW
00020000	00010000			Heap	Map	RW	RW
0022D000	00001000				Priv	RW Gua	RW Gua
0022E000	00002000			Stack of main thread	Priv	RW	RW
00230000	00004000				Map	R	R
00240000	00001000				Priv	RW	RW
00280000	00004000			Default heap	Priv	RW	RW
00380000	00067000				Map	R	R
00400000	00001000	heapOverflow		PE header	Img	R	RWE Cop
00401000	00001000	heapOverflow	.text	Code	Img	R E	RWE Cop
00402000	00001000	heapOverflow	.data	Data	Img	RW Cop	RWE Cop
00403000	00001000	heapOverflow	.rdata		Img	R	RWE Cop
00404000	00001000	heapOverflow	/4		Img	R	RWE Cop
00405000	00001000	heapOverflow	.bss		Img	RW Cop	RWE Cop
00406000	00001000	heapOverflow	.idata	Imports	Img	RW Cop	RWE Cop
00407000	00001000	heapOverflow	.CRT		Img	RW Cop	RWE Cop
00408000	00001000	heapOverflow	.tls		Img	RW Cop	RWE Cop
00409000	00001000	heapOverflow	/14		Img	R	RWE Cop
0040A000	00001000	heapOverflow	/29		Img	R	RWE Cop
0040B000	00001000	heapOverflow	/41		Img	R	RWE Cop
0040C000	00001000	heapOverflow	/55		Img	R	RWE Cop
00560000	00003000			Heap	Priv	RW	RW

Como se puede observar, en este caso se ha producido un desbordamiento de ***heap***.



Posteriormente a la presentación de la información por pantalla con la función `printf()`, también da el siguiente error antes de finalizar el programa.

4. Funciones vulnerables

La elección de un lenguaje de programación comporta conocer el nivel de seguridad de sus librerías y sus funciones. Conviene tener en cuenta que no todas las librerías y funciones de un lenguaje de programación son seguras en todas las circunstancias; un dato sobredimensionado, un valor fuera del rango previsto introducido como parámetro en una función no segura, no dará error en el momento de la compilación del programa pero sí puede provocar, por ejemplo, un desbordamiento de memoria. Cualquier ***hacker*** podría introducir valores sobredimensionados para provocar un desbordamiento de pila y aprovechar la situación para ejecutar el código que haya inyectado para atacar el sistema.

No vamos a analizar aquí cada uno de los lenguajes de programación ni todas sus funciones no seguras o vulnerables. Solo por poner un ejemplo, el lenguaje de programación C no proporciona una función de protección contra el acceso o sobrescritura de datos en la memoria; no comprueba que la escritura de datos en una memoria intermedia se encuentre dentro de los límites de dicha memoria; tampoco realiza la comprobación automática de los límites de las matrices o indicadores como muchos otros lenguajes. A esto hay que añadir que en la biblioteca estándar de C hay una gran diversidad de funciones vulnerables, como por ejemplo ***gets, getwd, strcpy, strcat, sprintf, scanf, sscanf, fscanf, vfscanf, vsprintf, vscanf, vsscanf, streadd, strecpy, realpath, syslog, getopt, getopt_long, getpass***, etc. La mayoría de estas funciones pueden provocar un ***buffer overflow*** si los valores de los parámetros que se asignan no son correctos o están sobredimensionados.

Enlaces recomendados

Unsafe (C# Reference): http://msdn.microsoft.com/es-es/library/chfa2zb8.aspx

Codi no segur i punters (Guia de programació de C#): http://msdn.microsoft.com/es-es/library/t2yzs44b.aspx

Pointer types (Guia de programació de C#: http://msdn.microsoft.com/es-es/library/y31yhkeb.aspx

En el lenguaje ***C#***, por ejemplo, para mantener la seguridad de tipo, no soporta punteros aritméticos (+, ++, -, --, *, &, ==, !=, >, etc.) por defecto o de manera predeterminada. Aun así, si se utiliza la palabra clave ***unsafe***, se puede definir un contexto no seguro en el cual se pueden utilizar estos punteros.

Así pues, el conocimiento de las particularidades de las librerías y funciones del lenguaje de programación escogido, para escoger las más seguras, es de vital importancia para realizar una programación segura.

A continuación se muestra un ejemplo de la función vulnerable ***strcat***.

```
char strcat(char *dest, const char *src)
```

La función ***strcat*** no valida la medida de las cadenas que va a concatenar. Esta función concatena una cadena ***src*** (cadena origen) a la cadena ***dest*** (cadena destino). Esta operación puede provocar un desbordamiento de memoria si la longitud de la concatenación de las dos cadenas es superior al espacio reservado.

En realidad, la función ***strcat*** añade una copia de la cadena apuntada por ***src*** al final de la cadena apuntada por ***dest*** y retorna un puntero a ***dest*** en el cual reside la cadena concatenada resultante.

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

void countStr(char str[])
{
      char frase[100] = "The string: \"";
      char total[6];
      snprintf(total, 6 , "%d", strlen(str));

      strcat(frase, str);
      strcat(frase, "\" has ");
      strcat(frase, total);
      printf("%s\n", frase);
      return;
}

int main(int argc, char *argv[])
{
      if(argc < 2){
            printf("Usage>text:");
            return -1;
      }
      countStr(argv[1]);
      printf("The program completes successfully \n");
}
```

strcatExample.c

El programa recibe un *text* como parámetro de entrada, y sin validar su medida, lo concatena con la variable ***frase***, que tiene reservado un espacio de 100 bytes, de los cuales ya tiene ocupados 13. Si la medida del parámetro de entrada es superior a 87 caracteres, al concatenarlo con la variable ***frase*** se sobrescribirá un espacio de la pila no reservado para esta variable. Según la longitud del valor del parámetro de entrada, puede ser que esta sobrescritura no tenga ningún efecto o altere la ejecución del programa.

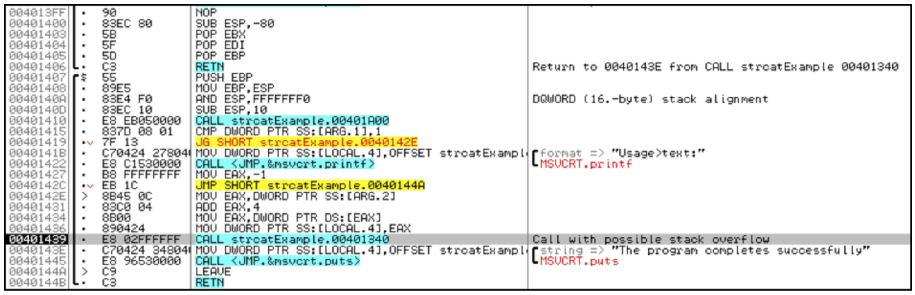
Además, a la variable ***frase*** se le concatena texto adicional; la adición de este número de caracteres podría superar el espacio reservado si todavía no lo había superado.

El comportamiento del programa variará en función de la longitud del parámetro de entrada. Si la cadena tiene hasta 78 caracteres, no se supera el espacio reservado para la variable ***frase***: 78 + 13 (The string : ") + 2 (número de caracteres) + 6 (" has) + carácter final de cadena = 100.

Una ejecución correcta puede ser como la siguiente:

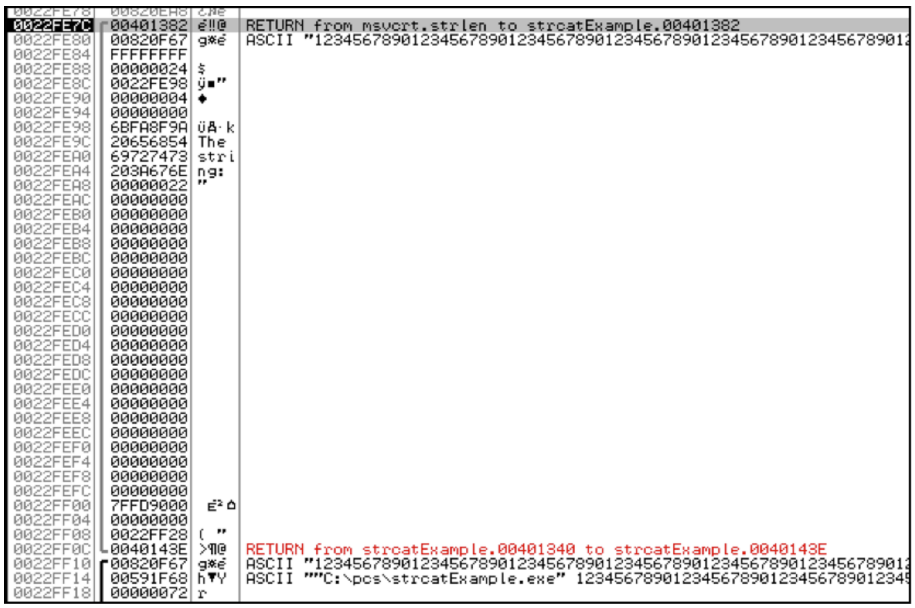
```
C:\pcs>strcatExample 12345678901234567890123456789O
The string: "12345678901234567890123456789O" has 30
The program completes successfully
```

Hay que observar que en la dirección 00401439 está la instrucción correspondiente a la llamada a la función `countStr(argv[1]);` es decir, `CALL strExemple.00401340` de la función ***main***, que llama a la función ***countStr***,, cuya ejecución retorna de la función ***countSt***. La ejecución del programa continúa en la siguiente instrucción de la llamada CALL, es decir, continúa la ejecución en la dirección 0040143E.

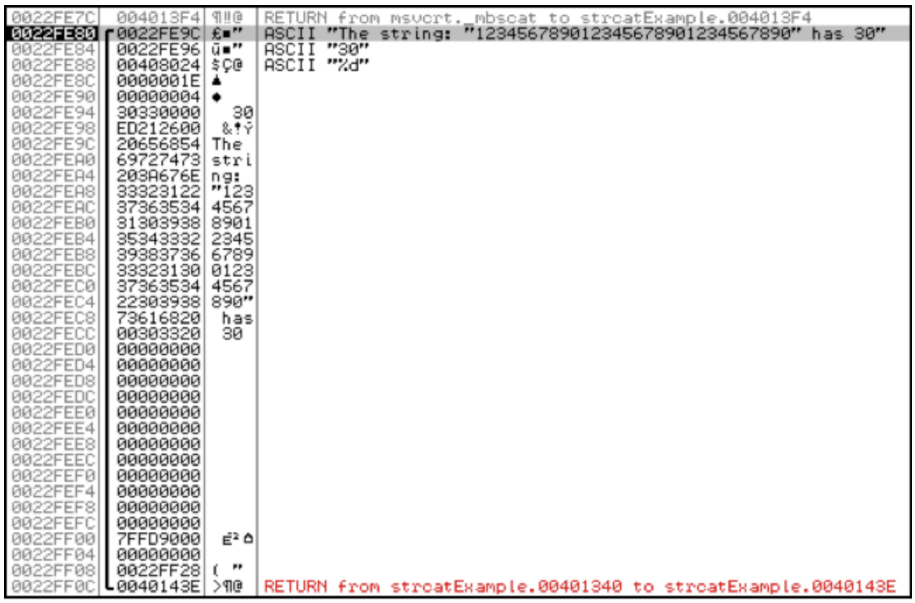


En la siguiente imagen se puede observar la información de la pila (***stack***) en el momento de hacer la llamada a la función ***countStr*** desde la función ***main***. En la dirección 0022FF0C de la pila se guarda la siguiente dirección de la llamada a la función, es decir, la dirección de retorno a la función ***main***, que en este caso es 0040143E.

Posteriormente, se reserva un espacio en la pila para la definición correspondiente a `char frase[100];` se puede observar que este espacio corresponde a la dirección 0022FE9C hasta la dirección 0022FF0F (0022FF0C + 3, 0022FF0C + 4 = 0022FF10)).



En este ejemplo vemos que el valor resultante de la concatenación no supera el espacio reservado en la memoria de la pila para la variable de destino y, por tanto, el programa finaliza sin incidencias.

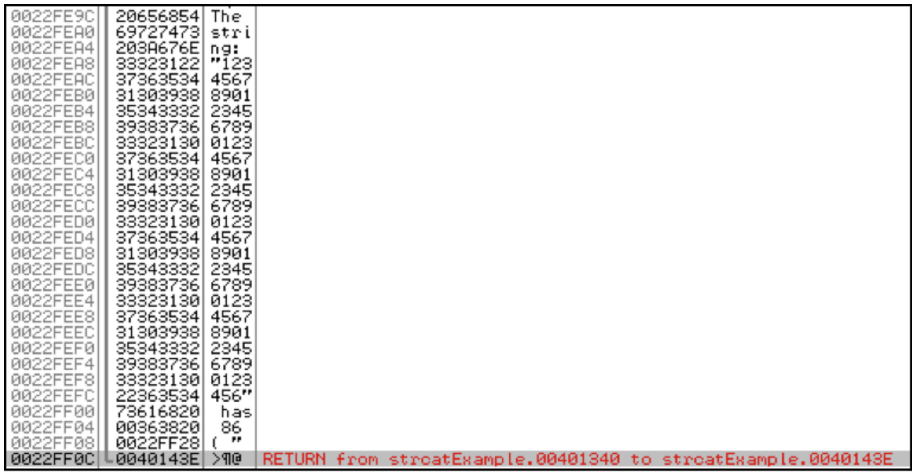


En el siguiente ejemplo la longitud de la cadena es de 86 caracteres:

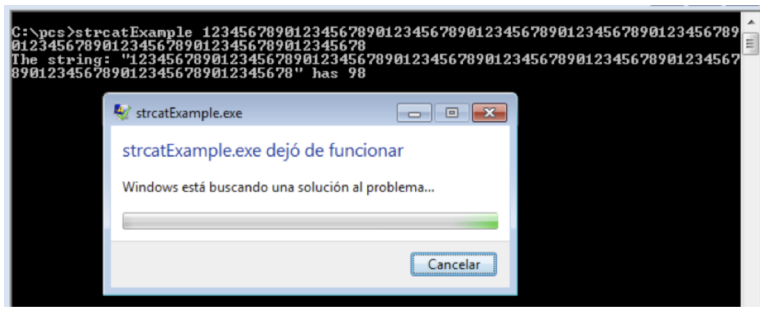
```
C:\pcs>strcatExample 12345678901234567890123456789012345678901234567
890123456789012345678901234567890123456
The string: "12345678901234567890123456789012345678901234567890123456
67890123456890123456789012345678" has 86
The program completes successfully
```

Se puede observar en la información de la pila que el resultado de la concatenación de la cadena ha sobrepasado el espacio reservado para almacenar la cadena y ha sobrescrito en las direcciones siguientes de la pila que correspondía

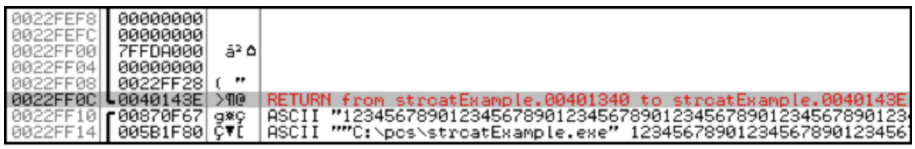
a la variable ***frase***. En este caso la sobrescritura no ha afectado a la ejecución del programa, dado que este espacio no se ha vuelto a utilizar posteriormente y no ha llegado a afectar a la dirección de retorno ni al proceso de ejecución, y el programa ha finalizado sin incidencias.



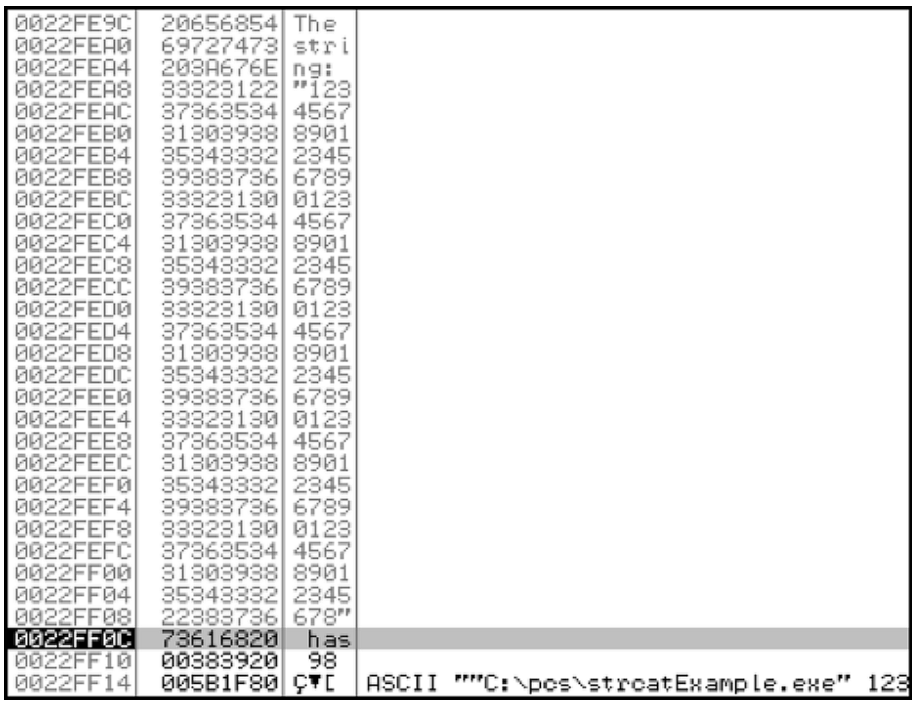
En el siguiente ejemplo se introducen 98 caracteres. Ahora sí se sobrescribirá la dirección de retorno y afectará a la ejecución del programa.



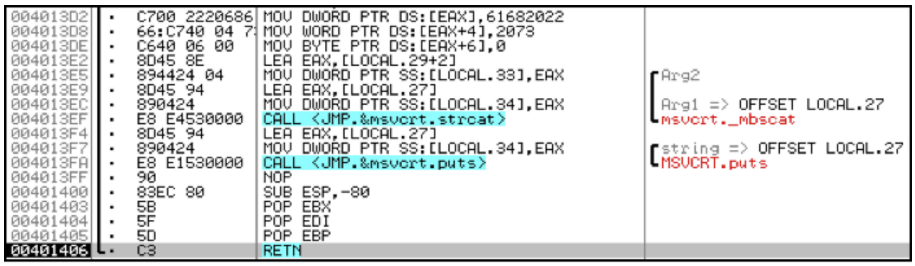
Analizando la evolución de la pila, se puede observar que la dirección de retorno 0040143E se almacena en la posición 0022FF0C.



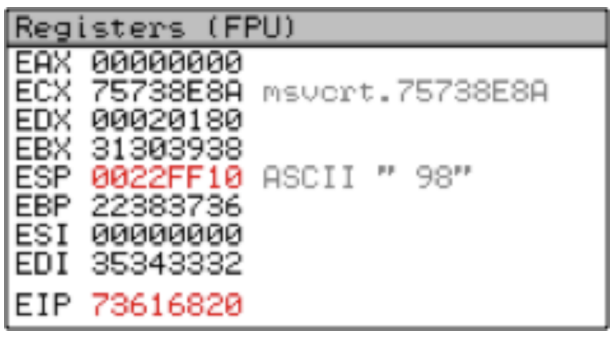
El resultado de la concatenación sobrepasa la medida reservada para la variable ***frase*** y sobrescribe la posición 0022FF0C, de modo que cambia el valor de la dirección de retorno 0040143E por el valor 73616820.



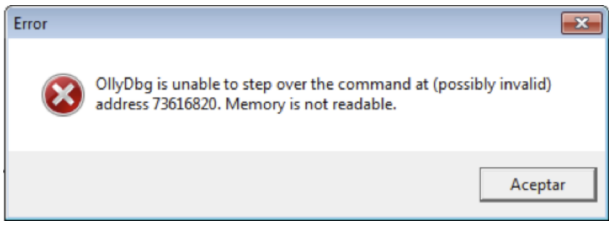
Cuando el programa ejecuta la instrucción RETN para retornar a la función ***main***, accede a la posición 0022FF0C de la pila para obtener la dirección de retorno. Esta dirección de retorno tendría que ser 0040143E, pero en su lugar encuentra el valor 73616820.



En este punto, el EIP apunta a la dirección 73616820, que es donde tendría que estar la instrucción que se tiene que ejecutar al retornar a la función ***main***, pero esta dirección no es la dirección correcta y, por tanto, da un error de memoria, `Memory is not readable`.



A continuación se muestra el mensaje motivado como consecuencia del desbordamiento de pila que ha provocado la concatenación.



En este caso se produce una situación similar a la mostrada en otros ejemplos en los que el sistema muestra un mensaje de error y el programa deja de funcionar. De nuevo, este desbordamiento podía haber sido aprovechado por un ***hacker***. En el caso de que se hubiera tratado de un ataque intencionado, se hubiera podido enviar un parámetro a través de un ***shellcode*** o un ***payload***, de forma que en la dirección de la pila 0022FF0C se hubiera sobrescrito un valor intencionado que correspondiera a una dirección de memoria, en la que el ***hacker*** hubiera inyectado algún código con las variables y así poderlo ejecutar, o ejecutar cualquier programa cargado en el sistema.

Bibliografía

Howard, M.; LeBlanc, D. (2002). *Writing Secure Code, Second* Edition . Redmond Washington: Microsoft Press.

Foster, J. C.; Osipov, V. (2005). *Buffer Overflow Atacks*. Syngress Publishing Inc.