

Curso: (62612) Diseño de aplicaciones seguras

Fernando Tricas García

Departamento de Informática e Ingeniería de Sistemas
Universidad de Zaragoza

<http://webdiis.unizar.es/~ftricas/>

<http://moodle.unizar.es/>

ftricas@unizar.es

Tema X: Desbordamientos de memoria

Fernando Tricas García

Departamento de Informática e Ingeniería de Sistemas
Universidad de Zaragoza

<http://webdiis.unizar.es/~ftricas/>

<http://moodle.unizar.es/>

ftricas@unizar.es



Desbordamientos de memoria

'Buffer Overflows' 'Buffer Overruns'

Los desbordamientos de memoria llevan décadas causando problemas.

- ▶ Internet worm y fingerd (1988)
- ▶ Más del 50% de los problemas de seguridad en 1999 (CERT/CC)
- ▶ Enero 2000-2004, 19% de vulnerabilidades
- ▶ 48% de los problemas en de seguridad (CERT/CC, 12 octubre 2004)



Una demostración sencilla

Desbordamiento de la pila

```
#include <stdio.h>

int main(int argc, char **argv)
{
    int a;
    char b[2];
    char stop[10];

    stop[0]='\n';

    a=0;
    b[0]='1';
    b[1]='\0';

    printf("%s\n",b);

    strcpy(b, argv[1]);

    if (a!=0) {
        printf("Vale... te pillé!\n", a);
    }

    printf("a: %d\n", a);
    printf("b: %s\n", b);
}
```

Si ejecutamos ...

```
./overflow A
1
a: 0
b: A
```

```
./overflow AA
1
a: 0
b: AA
```

```
./overflow AAA
1
Vale ... te pillé!
a: 65
b: AAA
```



Desbordamientos: ¿Todavía?

- ▶ Es extremadamente sencillo equivocarse
 - ▶ Mal diseño del lenguaje
 - ▶ Malas prácticas de programación
- ▶ Hay lenguajes inmunes, pero no siempre podremos usarlos
 - ▶ Aún así, estos lenguajes inmunes utilizan bibliotecas escritas en lenguajes 'peligrosos'.



Desbordamientos: ¿Qué son?

- ▶ Los programas necesitan almacenar datos en la memoria
- ▶ C (y otros) memoria dinámica
 - ▶ Se puede alojar en la
 - ▶ pila ('stack')
 - ▶ zona de memoria dinámica ('heap')

'buffer'

- ▶ Se trata de almacenar más datos de los que 'caben' en la memoria reservada y sacar partido de ello.



Desbordamientos de memoria

No sólo variables de texto...

Además

- ▶ Desbordamiento de enteros
- ▶ Ataques de cadenas de formato



Desbordamientos

Y la pila

```
void funcion_chunga(char *cadenaEntrada)
{
    char memoriaAuxiliar[10];

    strcpy(memoriaAuxiliar, cadenaEntrada);
}
```

La pila crece por aquí ... ↑

memoriaAuxiliar
(zona de variables locales)

'Crecen hacia abajo' ... ↓

Dirección de retorno (RET)

*cadenaEntrada
(parámetros de la función)

la pila ...



Departamento de
Informática e Ingeniería
de Sistemas
Universidad Zaragoza

Desbordamientos: ¿Qué sucede?

- ▶ Si alguien intenta (y consigue) almacenar más de lo que cabe ... normalmente irá a parar encima del siguiente 'trozo' de memoria
- ▶ Los programas pueden
 - ▶ Actuar de forma extraña
 - ▶ Fallar
 - ▶ Seguir ejecutándose



Desbordamientos: ¿De qué depende?

Depende de:

- ▶ Cuántos datos
- ▶ Qué datos se sobrescribieron
- ▶ Cuales se escribieron en su lugar
- ▶ ¿El programa accede a esos datos?



Desbordamientos: ¿problema de seguridad?

- ▶ Según dónde se escriba, se pueden modificar valores críticos para el programa.
 - ▶ Pero todavía puede ser más sofisticado: ataques abusando de la pila ('stack-smashing attacks' y similares).
 - ▶ Return-into-libc (Con cualquier desbordamiento...)
- ▶ Pueden ser mucho más serios que un simple cambio de valor en una variable.

Con un poco de creatividad, el atacante puede llegar a ejecutar cualquier cosa



Pero ... ¿cómo funciona?

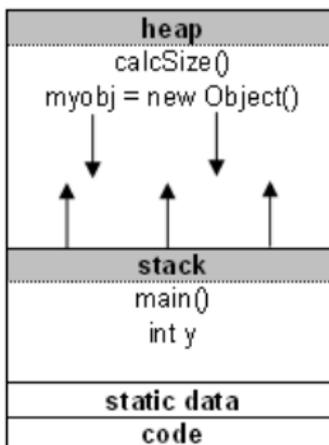
Zonas de memoria

- ▶ La pila (**stack**) ...
 - ▶ Parámetros y entorno del programa
 - ▶ La pila (crece cuando el programa avanza, crece hacia la zona de memoria dinámica)
- ▶ La zona de memoria dinámica (**heap**, también crece, hacia la pila)
 - ▶ Segmento de almacenamiento de bloques ('block storage segment'): datos de acceso global
 - ▶ Segmento de datos. Datos de acceso global, inicializados.
 - ▶ Segmento de texto. Código del programa, sólo lectura.



Los candidatos

- ▶ La pila y la zona de memoria dinámica crecen cuando el programa se ejecuta
- ▶ La pila se gestiona automáticamente (llamadas a funciones: contexto, variables no estáticas, parámetros, ...)
- ▶ Se pueden desbordar las dos



<http://www.maxi-pedia.com/what+is+heap+and+stack>

Desbordamientos

Zona de memoria dinámica ('heap')

Sencillos en teoría, pero complicados de llevar a cabo.

1. Hay que saber qué variables son críticas
2. Encontrar alguna zona de memoria, que adecuadamente modificada altere esas variables
3. Puede ocurrir que la modificación 'rompa' la ejecución del programa



Una demostración sencilla

Desbordamiento del heap

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

int main(int argc, char **argv)
{
    char * buf1 = (char *) malloc(5);
    char * buf2 = (char *) malloc(10);

    strcpy(buf1, argv[1]);
    strcpy(buf2, argv[2]);

    printf("buf1: %s\n", buf1);
    printf("buf2: %s\n", buf2);
}
```

Si ejecutamos ...

```
./overflowHeap A B
buf1: A
buf2: B
```

```
./overflowHeap AA B
buf1: AA
buf2: B
```

```
...
./overflowHeap AAAAAAAAAAAAAAAAAA B
buf1: AAAAAAAAAAAAAAAAAA
buf2: B
```

```
./overflowHeap AAAAAAAAAAAAAAAAAA B
buf1: AAAAAAAAAAAAAAAAAAB
buf2: B
```



Desbordamientos

Desbordando la pila

Una diferencia fundamental: siempre hay algo interesante para escribir, la dirección de retorno

1. Encontrar una zona de memoria candidata en la pila
2. Colocar código hostil en algún sitio
3. Escribir sobre la dirección de retorno, la dirección del código

Hacer 'mapa' para encontrar variables interesantes según la dirección que se va a sobrescribir

'Smashing The Stack For Fun And Profit'
Aleph One

<http://insecure.org/stf/smashstack.html>



Desbordamiento de la pila

Código de ataque

- ▶ No es sencillo
- ▶ Ensamblador para la máquina atacada
- ▶ Se puede encontrar en la red (y en los libros, artículos, ...)

Entonces ...

- ▶ Se coloca el código adecuado en cualquier parte
- ▶ Se sobrescribe la pila para que el control pase a ese código
- ▶ Si el programa se ejecutaba con privilegios altos, el atacante los consigue
- ▶ La zona de memoria es más difícil de atacar, pero no es la solución del problema.



Desbordamientos: defensa

- ▶ Programación defensiva. Cuidado con:

```
strcpy() strcat() sprintf() scanf() sscanf() fscanf()  
vfscanf() vsprintf vscanf() vsscanf() streadd()  
strcpy() strtrns()
```

- ▶ Evitarlas siempre que sea posible, casi todas tienen alternativas razonables.

strncpy, strncat, ... pero cuidado!



Desbordamientos internos

Algunas funciones 'expanden' sus parámetros internamente, y que no siempre controlan bien los límites

- ▶ `realpath()`
- ▶ `syslog()` En principio, los problemas resueltos, cuidado con sistemas viejos
- ▶ `getopt()` `getpass()` Consejo: cuidado con lo que pasamos.



Desbordamientos: más con las entradas

```
gets() getchar(), fgetc(),getc(), read()
```

Consejo, comprobar:

- ▶ Siempre los límites!
- ▶ Longitud de los datos antes de almacenarlos
- ▶ No pasar datos excesivamente grandes a otras funciones



Más riesgos

- ▶ Las funciones 'seguras' no lo son tanto

`strncpy()` `strncat()`

...

- ▶ Pueden dejar cadenas sin cerrar
- ▶ Pueden inducir problemas (un carácter)
- ▶ Fácil equivocarse



Más funciones con las que tener cuidado

strcadd memcopy
strcpy strncpy
vsnprintf fgets
bcopy snprintf

bcopy(), fgets(), memcpy(), snprintf(), strcpy(),
strcadd(), strncpy(), vsnprintf()

getenv(): nunca suponer pequeño el tamaño de las variables



Mas riesgos: aún más

- ▶ Hay más (incluso desconocidas)
- ▶ No fiarse de las bibliotecas de otros (ni de las comerciales)
- ▶ No suponer nada acerca de los programas de otros

Ya tenemos por donde empezar ... el resto es cosa nuestra



Algunas herramientas

Ya nombramos algunas herramientas de análisis. Otras ideas:

- ▶ Pila no ejecutable
 - ▶ Comprobación de límites en el compilador
 - ▶ /GS (en compiladores de Microsoft)
 - ▶ -fstack-protector (en el compilador gcc)
- ¡Mirar la documentación!
- ▶ Utilizar un lenguaje seguro (al menos en esto)
 - ▶ Stackguard (canario)
 - ▶ Address Space Layout Randomization



Bibliotecas seguras

- ▶ Strsafe:

<http://msdn.microsoft.com/en-us/library/ms647466.aspx>

- ▶ strlcpy, strlcat:

<http://www.courtesan.com/todd/papers/strlcpy.html>

- ▶ libsafe: reemplazar las funciones inseguras por versiones seguras

- ▶ glib:

<http://developer.gnome.org/glib/2.34/>

[glib-String-Utility-Functions.html](http://developer.gnome.org/glib/2.34/glib-String-Utility-Functions.html)

[http://www.research.avayalabs.com/gcm/usa/en-us/initiatives/all/nsr.htm&Filter=](http://www.research.avayalabs.com/gcm/usa/en-us/initiatives/all/nsr.htm&Filter=ProjectTitle:Libsafe&Wrapper=LabsProjectDetails&View=LabsProjectDetails)

[ProjectTitle:Libsafe&Wrapper=LabsProjectDetails&View=LabsProjectDetails](http://www.research.avayalabs.com/gcm/usa/en-us/initiatives/all/nsr.htm&Filter=ProjectTitle:Libsafe&Wrapper=LabsProjectDetails&View=LabsProjectDetails)

¿Abandonado?



¿Y en Windows?

- ▶ Es más difícil
- ▶ Muchas funciones 'interesantes' se cargan dinámicamente
- ▶ Esto dificulta 'encontrarlas' en la memoria

'The Tao of Windows Buffer Overflow'

DilDog

http://www.cultdeadcow.com/cDc_files/cDc-351/



Desbordamiento de enteros

- ▶ El problema aparece cuando el tipo de datos seleccionado para almacenar ciertos datos no es capaz de albergar los datos de nuestro programa.
 - ▶ Es frecuente olvidar que los tipos de datos son capaces de almacenar una cantidad limitada de ellos
- Ejemplo: números enteros con signo (16 bits)

Binario	Decimal
0111 1111 1111 1111	32767
1000 0000 0000 0000	-32767
1111 1111 1111 1111	-1



Entonces ...

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

main(int argc, char *argv[])
{
    short int a = 25;
    short int b = 25;
    short int c = 2*a;

    printf("a:␣%d\n", a);
    printf("b:␣%d\n", b);
    printf("c:␣%d\n", c);

    a = atoi(argv[1]);
    b = atoi(argv[2]);
    c = a + b;

    printf("-----\n\n", a);
    printf("a:␣%d\n", a);
    printf("b:␣%d\n", b);
    printf("a+b:␣%d\n", a+b);
    printf("c=a+b:␣%d\n\n", c);
}
```

Si ejecutamos ...

```
./integerOverflow 10 10
```

```
a: 25
b: 25
c: 50
```

```
a: 10
b: 10
a+b: 20
c=a+b: 20
```

```
./integerOverflow 100000 10
```

```
a: 25
b: 25
c: 50
```

```
a: -31072
b: 10
a+b: -31062
c=a+b: -31062
```



Problemas

- ▶ Los obvios con cambios de valores no esperados
- ▶ rangos de índices de vectores y matrices
- ▶ reserva de memoria

'Basic Integer Overflows'

blexim

<http://www.phrack.org/issues.html?issue=60&id=10>



Ataques de cadenas de formato

Reconocido en el año 2000 *format string attack*

- ▶ `fprintf(sock, username)`
- ▶ en lugar de
- ▶ `fprintf(sock, "%s", username)`



Cadenas de formato

El problema: %n %x %s

```
int
main(int argc, char **argv)
{
    int          num;
    printf("%s\n\n", "foobar", &num);
    printf("%d\n", num);
}
```

Imprime:

foobar

6

(el %n está antes del \n)

Por lo tanto ...

Podemos escribir enteros en la pila, con una cadena de formato adecuada



Una demostración sencilla

Cadenas de formato

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

int main(int argc, char *argv[])
{
    char args[512];

    strcpy(args, argv[1]);
    printf(args);
    printf("\n");
}
```

Si ejecutamos ...

```
./formatString2 AAAA
AAAA
```

```
./formatString2 %x%x
bf9dc575b78535c9
```

```
./formatString2 AAAA%x%x%x%x%x%x%x%x%x%x
AAAAbff4d563b78015c9bff4c768b7810cc2710bff4c9b40b781dff4
```

```
./formatString2 AAAA%x%x%x%x%x%x%x%x%x%x%x%x%x%x
```

```
AAAAbfe6855fb77e65c9bfe66db8b77f5cc2710bfe670040b7802ff4b780353c41414141
```



Consiguiendo información

Cadenas de formato

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

int main(int argc, char *argv[])
{
    char args[512];
    char * password= "TESLA";

    strcpy(args, argv[1]);
    printf("Dirección del secreto:");
    printf("secreto: %08x\n",
           password);
    printf(args);
    printf("\n");
}
```

Si ejecutamos ...

```
./formatString3 AAAA
Dirección del secreto: 00400764
AAAA
```

```
./formatString3 AAAA%x%x%x%x%x%x%x%x%x%x%x%x
Dirección del secreto: 00400764
AAAA12ab00001288ca ... a84f6faedb040076441414141
```

```
./formatString3 `printf "\x64\x07\x40\x00" `%x%x%x%x%x%x%x%x
Dirección del secreto: 00400764
d@61a4f0006182bad0161a4f01f136a0a15861839a8436a0a010TESLA
```

Sacha Fuentes. 'Exploiting Software'. In hakin 9 starter kit 3/2007

