Módulo 3.2

Redes y Seguridad
Gr. Ing. Informática
J.L. Vázquez Poletti

Objetivo

En este módulo probaremos varias amenazas derivadas de una programación irresponsable a través de varios códigos de ejemplo.

CONSEJO: Debido a la longitud de algunos programas, puede ser recomendable traerlos en un llavero USB y conectarlo a la máquina virtual.

Punto de partida

La principal línea de productos de ENCOM es el software, por lo que es importante estudiar diversos casos de uso y escenarios a fin de evaluar las amenazas. Estas amenazas pueden estar originadas en un descuido del equipo de desarrollo o ser directamente intencionadas.

Se ha escogido código perteneciente a los nuevos productos (MCP, Crom,…). Éste está escrito en **C** y se compilará (si no se indicar lo contrario) ejecutando: **gcc -o *programa código***.

Ejecución recursiva infinita

Se generará un programa que llamará a una función que a su vez se irá llamando a sí misma de forma recursiva e infinita.

El comando **strace *programa*** permite obtener más información de la ejecución.

Código (**recursive.c**):

```
#include <stdio.h>

int foo() {

        printf ("Llamada a foo()\n");

        return foo(); }

void main(){

        foo(); }
```

¿A qué se corresponde el término SIGSEGV y quién lo ha invocado?

Bucle "Alderson"

Este tipo de bucle le debe su nombre a un desarrollador de Microsoft que cometió un grave error cuando programó las notificaciones de Microsoft Access.

El siguiente código de ejemplo debería ir sumando los números que el usuario va introduciendo pero contiene un error del tipo mencionado antes.

Máquinas Virtuales

- RyS-ENCOM (ubuntu:reverse)

Más información

Strace: http://sourceforge.net/projects/strace/

GDB: http://sources.redhat.com/gdb/

Radamsa: http://code.google.com/p/ouspg/wiki/Radamsa

NASM: http://sourceforge.net/projects/nasm/

MSF payload: http://www.offensive-security.com/metasploit-unleashed/Msfpayload

Código (**alderson.c**):

```
#include <stdio.h>

void main (){
        int i;
        while(1){
                printf ("Introducir numero: ");
                scanf ("%d",&i);
                if (i * 0) {
                        i += i;
                } else {
                        break;
                }
        }
}
```

¿Cuál es el fallo de este código?
¿Qué tipo de vulnerabilidad es (de las vistas en clase)?
(OPCIONAL) ¿En qué consistió el fallo cometido en Microsoft Access?

Variables demasiado grandes

El compilador **gcc** directamente se negará a compilar el siguiente código.

Código (**xxxl.c**):

```
void main(){
        double xxxl[1000000000]; }
```

¿Cuál es el tamaño en bytes que se reserva para la variable **xxxl**?
¿Cuál es el tamaño máximo que se permite para la variable **xxxl**?
¿Cuál es el tamaño máximo que se permite para la variable **xxxl** si se emplearan otros tipos de datos (short, int, long double)?

Desbordamiento de montículo

En el siguiente código se verá como el contenido de un buffer se desborda y afecta a otro.

Código (**headoverflow.c**):

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>


#define BUFSIZE 16

#define OVERSIZE 8 /* desbordar buf2 en OVERSIZE bytes */


int main()

{

        u_long diff;

        char *buf1 = (char *)malloc(BUFSIZE), *buf2 = (char
        *)malloc(BUFSIZE);



        diff = (u_long)buf2 - (u_long)buf1;

        printf("buf1 = %p, buf2 = %p, diferencia = 0x%lu bytes\n", buf1,
        buf2, diff);



        memset(buf2, 'A', BUFSIZE-1), buf2[BUFSIZE-1] = '\0';



        printf("Antes del desbordamiento: buf2 = %s\n", buf2);

        memset(buf1, 'B', (u_int)(diff + OVERSIZE));

        printf("Después del desbordamiento: buf2 = %s\n", buf2);



        return 0;

}
```

Probar con varios tamaños de desbordamiento
¿Por qué el programa no devuelve fallo?

Desbordamiento de pila al detalle

En este ejemplo se verá más en detalle cómo desbordando un buffer sobrescribe la dirección que contiene la dirección base de la pila (EBP, *Extended Base Pointer*).

Código (**buffer.c**):

```
#include <stdio.h>

#include <unistd.h>

void Test()

{

  char buff[4];

  printf("Some input: ");

  gets(buff);

  puts(buff);

}

int main(int argc, char *argv[ ])

{

  Test();

  return 0;

}
```

1. Compilar de la siguiente manera: **gcc buffer.c -o buffer -ggdb -static**.
2. Ejecutar introduciendo varias veces "A".

¿Cuál es el tamaño máximo del buffer?
¿Cuántos veces se puede introducir "A" sin que falle el programa? ¿Es diferente al valor anterior? ¿Por qué?

A continuación se usará el **GNU Debugger (GDB)**, herramienta clave en cualquier máquina de desarrollo, para hacer un primer reconocimiento del código desensamblado.

1. Ejecutar **gdb buffer**.
2. Suspender la ejecución con **break main**.
3. Desensamblar la función **main** con **disass**.

4. Desensamblar la función **Test**.

¿Cómo se calcula la dirección efectiva para guardar los datos en el buffer (instrucción **lea**)?

NOTA: Para salir de **gdb** se debe introducir **quit**.

Ahora se hará una pasada más detallada de la ejecución del programa pero sin realizar desbordamiento.

1. Volver a arrancar **gdb** con **buffer** como argumento.
2. Suspender la ejecución.
3. Comenzar la ejecución con **r**.
4. Indicar que se quiere ejecutar paso a paso con **s**.
5. Pulsar **enter** hasta que se pida introducir texto.
6. Introducir "AAA".
7. Averiguar la dirección del ESP (*Extended Stack Pointer***,** la parte de arriba de la pila aunque en la arquitectura x86 esté ubicada abajo) con **x/x $esp**.
8. Hacer lo mismo con **$ebp**.
9. Usar **x/s** seguido de las direcciones de ESP y EBP para obtener su contenido.

Encontrar el comienzo de los datos introducidos con **x/s $ebp-X**, siendo **X** el valor decimal empleado para la dirección efectiva calculada en el apartado anterior). **PISTA:** Se debe visualizar claramente "AAA".

Ahora, se procederá a provocar el desbordamiento y verlo con más detalle.

1. Volver a arrancar **gdb** con **buffer** como argumento.
2. Suspender la ejecución.
3. Comenzar la ejecución.
4. Indicar que se quiere ejecutar paso a paso.
5. Pulsar **enter** hasta que se pida introducir texto.
6. Introducir 14 veces "A".
7. Averiguar la dirección de ESP, EBP y el buffer.
8. Visualizar el contenido del buffer y EBP.

Aparte de ser un desbordamiento de pila, este tipo de agujero responde a otro de los tipos visto en clase, ¿cuál es?

Fuzzing de aplicaciones

Un *fuzzer* es una aplicación que busca vulnerabilidades en aplicaciones enviando entradas arbitrarias. La variedad de las entradas y la cantidad de experimentos diferentes es clave para localizar fallos que serían fatales.

La herramienta empleada es **Radamsa**, un *fuzzer* de código abierto, sencillo pero con un gran potencial.

1. Ejecutar **echo "1 + (2 + (3 + 4))" | radamsa --seed 12 -n 4**.

Nótese que **-n** indica el número de experimentos y **--seed** la semilla con la que se iniciará el módulo de aleatoriedad.

Ejecutar el comando varias veces y probar con varias semillas. ¿Por qué es importante que los resultados sean los mismos con la misma semilla?

2. Someter a prueba a la calculadora de la terminal (**bc**). Se piden 1000 experimentos con variaciones de la entrada **"100 * (1 + (2 / 3))"**.

¿Se puede concluir que **bc** es robusto?

3. Crear un archivo comprimido de la siguiente manera**: gzip -c /bin/bash > archivo.gz**.
4. Someter a prueba **gzip**: **while true; do radamsa archivo.gz | gzip -d > /dev/null; done**

   Dejarlo ejecutándose durante unos minutos.

¿Los errores mostrados por **gzip** indican que el programa en sí tiene fallos?

Uso de shellcode

El *shellcode* es un conjunto de órdenes programadas a nivel de código máquina y que pueden ser inyectadas en la pila. Le debe su nombre a su propósito más común, que es obtener acceso a una terminal (shell) desde un punto en el que no estaba pensado.

En particular, se va a trabajar con el siguiente *shellcode* de 25 caracteres:

"\x31\xc0\x31\xd2\xb0\x0b\x52\x68\x6e\x2f\x73\x68\x68\x2f

\x2f\x62\x69\x89\xe3\x52\x53\x89\xe1\xcd\x80"

que ejecuta **/bin/sh**.

1. Comprobar la traducción a instrucciones en ensamblador con el siguiente comando perteneciente al **Netwide Assembler (NASM)**: **echo –ne "*shellcode*" | ndisasm -u -**.
2. Equipar el *shellcode* en un programa en C para poderlo utilizar.

Código (**shellcode.c**):

```
void main()

{

((void(*)(void))

{

"shellcode"

}
```

```
)();
}
```

3. Compilar (ignorar el warning) y ejecutar el programa.

¿Cómo se sabe que ha funcionado?
¿En qué situaciones/programas se podría inyectar este *shellcode*?

IMPORTANTE: existen sistemas como el **MetaSploit Framework (MSF)** que preparan *shellcode* que ejecuta los comandos deseados (ejemplo: "/bin/cat /etc/shadow") para incluirlo como carga útil (*payload*) en un código.