Práctica: **Programación de Código Seguro**

En esta sesión se van a poner en práctica los conocimientos adquiridos durante las clases de programación segura, practicando con algunas de las funciones que has visto que son inseguras y que pueden provocar vulnerabilidades. En concreto, las vulnerabilidades de *desbordamiento de búfer* (BOF) y *formateo de cadena* (format string).

1. Entorno de prácticas

1.1. Configuración inicial

Para la realización de esta práctica, tendrás que descargarte la máquina virtual que proporcionada por el profesor en formato OVA. Esta máquina virtual está ejecutando una máquina Debian stretch 9.5 de arquitectura 32 bits.

La máquina virtual tendrás que desplegarla en el ordenador de prácticas (o tu ordenador) con el software de virtualización de tu elección (VirtualBox, VMWare, u otros).

Las credenciales de acceso a la máquina virtual que tienes que usar son `student` / `student`. La contraseña del usuario `root` es `toor` (aunque no deberías de usar esta cuenta!).

1.2. Compilación

Para esta práctica vamos a usar el compilador GNU de C, `gcc`. Mediante el comando:

```
gcc mycode.c -o mycode.out
```

Conseguirás compilar un fichero de nombre "mycode.c" y crear un fichero ejecutable de Linux de nombre "mycode.out". Ahora, puedes ejecutar el binario obtenido mediante la siguiente orden en una terminal de la máquina virtual:

```
./mycode.out
```

1.3. Depurando errores

Cuando aparece un error durante la ejecución de un programa, es muy probable que su ejecución se interrumpa de forma inesperada (por ejemplo, cuando sucede una división entre cero). Sin embargo, las razones específicas del fallo que ha provocado ese cierre inesperado no se muestran de manera clara al usuario/desarrollador. En el caso de C, muchas veces se obtiene un mensaje algo críptico como "Segmentation fault", sin detalles adicionales.

Código 1: Ejemplo de código fuente con un error (file *error.c*).

```
1 #include <stdio.h>
2
3 int main()
4 {
5     for(int i = 10; i >= 0; i--)
6         printf("Do stuff %d\n", 10 /i);
7     return 0;
8 }
```

Observa el Código 1. Como puedes ver, hay un error en la línea 6, dado que la variable `i` toma el valor de cero, provocando por tanto un error en tiempo de ejecución derivado de la división entre cero. Si compilas este programa con `gcc` y lo ejecutas, verás algo como lo siguiente:

```
$ gcc error.c -o error
$ ./error
Do stuff 1
Do stuff 1
Do stuff 1
Do stuff 1
Do stuff 1
Do stuff 2
Do stuff 2
Do stuff 3
Do stuff 5
Do stuff 10
Floating point exception
```

Como esperábamos, la ejecución ha finalizado con error debido a esa división entre cero. Cuando el programa rompe durante su ejecución, se puede obtener información adicional acerca de cuál fue la causa concreta que generó el fallo. Esta información adicional facilita una posterior sesión de depuración del programa, realizada con objeto de solucionar el error.

Esta información adicional se almacena en un fichero, de nombre *core dump*, que se genera por el propio sistema operativo cuando el programa rompe su ejecución. En concreto, este fichero contiene una "foto" del estado de la memoria del programa que ha fallado en el momento en que se produjo el fallo o se terminó de forma abrupta.

Por defecto, estos ficheros core dump no se generan cuando se produce un fallo en ejecución. Para activar su generación, hay que ejecutar el siguiente comando en una terminal:

```
ulimit -c unlimited
```

Una cosa importante de este comando es que funciona por *sesión de terminal*: es decir, si cierras la termina y vuelves a abrir otra, tendrás que volver a ejecutar el comando para activar la generación de ficheros core dump.

Ahora, abre una termina, activa la generación de ficheros core dump y vuelve a ejecutar el programa anterior:

```
$ ulimit -c unlimited
$ ./error
Do stuff 1
Do stuff 1
Do stuff 1
Do stuff 1
```



```
Do stuff 1
Do stuff 2
Do stuff 2
Do stuff 3
Do stuff 5
Do stuff 10
Floating point exception (core dumped)
```

¿Observas la diferencia? Exactamente! Ahora, dice "core dumped" cuando la excepción de punto flotante ha sucedido. De hecho, si ahora listas el contenido del directorio actual (mediante el comando `ls`), deberías de encontrar que existe un nuevo fichero llamado `core`.

Este fichero es el denominado fichero core dump, que resulta muy útil cuando se quiere realizar una sesión de depuración con el depurador de GNU `gdb`. Un depurador es un programa especial que nos permite ir viendo la evolución de otro programa durante su ejecución, y resulta útil para detectar posibles fallos o problemas en el código fuente. Para abrir una sesión de depuración en `gdb` usando el fichero de core dump generado no tienes más que indicar el nombre de este fichero justo después del nombre de la aplicación que quieres analizar. En este ejemplo:

```
$ gdb error core
GNU gdb (Debian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from error...(no debugging symbols found)...done.
[New LWP 1857]
Core was generated by `./error'.
Program terminated with signal SIGFPE, Arithmetic exception.
#0  0x004055cc in main ()
```

Como ves, nos informa de que el programa ha terminado debido a la señal `SIGFPE`, que se asocia a excepciones aritméticas (errores ocurridos durante operaciones de cálculo). Además, te indica en qué función se ha producido el error (`main`, en este caso) y en qué dirección de memoria se generó esta excepción (`0x004055cc`, en este caso).

Para ayudarte durante la sesión de depuración de un programa es recomendable que compiles el ejecutable con soporte de depuración (opción `-g`) como se muestra a continuación:

```
gcc error.c -o error -g
```

Además, `gdb` incorpora una interfaz de usuario desde terminal que se puede activar añadiendo la opción `-tui` cuando se ejecuta el depurador. Con esta opción se abrirá el programa `gdb`,

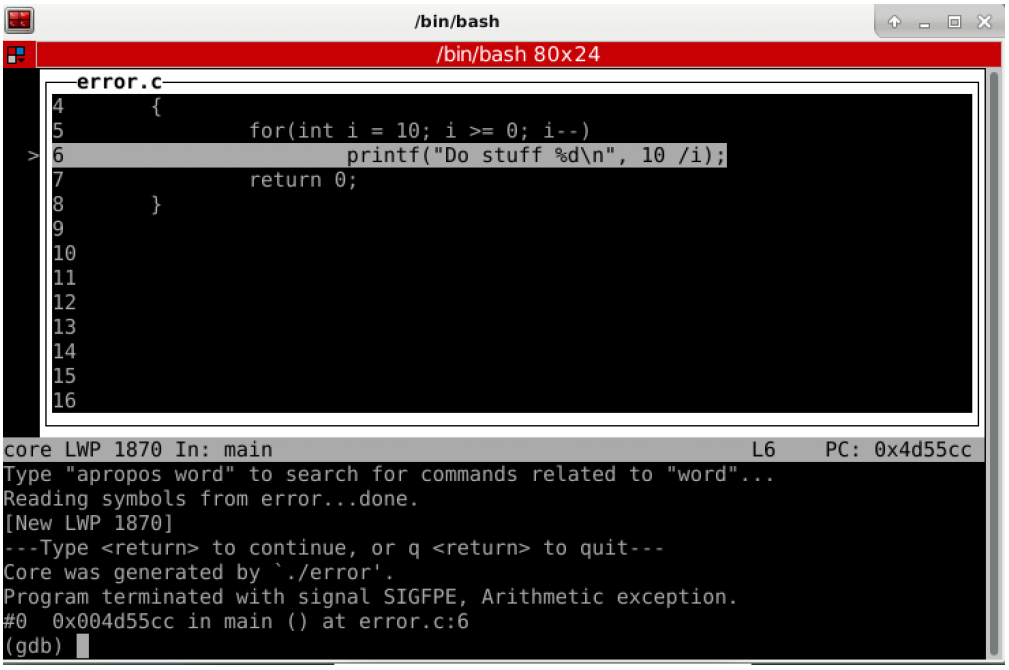

Figura 1: Textual user interface of `gdb` with the core dump file.

mostrando el código fuente en la ventana superior. Si vuelves a ejecutar el programa anterior para generar el fichero core dump otra vez y ejecutas el comando `gdb` con la opción `-tui` y con el fichero core dump:

```
gdb error core -tui
```

Verás algo similar a lo que se muestra en la Figura 1. Observa que la línea de código en donde se generó el error aparece destacada en esta ventana superior.

2. Vulnerabilidad de buffer overflow

En esta primera parte de la práctica vas a familiarizarte con la vulnerabilidad de desbordamiento de búfer.

El Código 2 contiene una vulnerabilidad de desbordamiento de búfer. Localiza la vulnerabilidad en el código e intenta explotarla. El objetivo es conseguir ejecutar la función `secret` para escribir el mensaje “YOU WIN!” por pantalla.

Recuerda que puedes usar en lenguaje interpretado de tu elección para generar entradas:

- python -c ’print “A”*20’
- perl -e ’print “A”*20’

Los dos ejemplos anteriores generan una cadena de 20 “A”’es.

Código 2: Código vulnerable (fichero *bof1.c*).

```c
#include <stdio.h>

#define BUFLEN 256

void secret()
{
    printf("YOU WIN!\n");
}

void echo()
{
    char buffer[BUFLEN];

    printf("Type your input:\n");
    scanf("%s", buffer);
    printf("Input given: %s\n", buffer);
}

int main()
{
    echo();

    return 0;
}
```

El Código 3 contiene una vulnerabilidad de desbordamiento de búfer. Localiza la vulnerabilidad en el código e intenta explotarla del mismo modo que en el ejemplo anterior.

Código 3: Código vulnerable (fichero *bof2.c*).

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

void secret()
{
    printf("YOU WIN!\n");
}

void check_argument(char *arg)
{
   char buffer[64];

   strcpy(buffer, arg);
   printf("Your argument is: %s\n", buffer);
}


int main(int argc, char *argv[])
{
   // check args
   if (argc != 2) {
      printf("usage: %s argument\n", argv[0]);
      return 1;
   }

   check_argument(argv[1]);
   return 0;
}
```

3. Vulnerabilidad de format string

En esta otra parte de la práctica vas a familiarizarte con la vulnerabilidad de formateo de cadenas.

El Código 4 contiene una vulnerabilidad de formateo de cadena. Localiza la vulnerabilidad e intenta explotarla. Como antes, el objetivo es conseguir ejecutar la función `secret` para escribir el mensaje "WE HAVE A WINNER!" por pantalla. Reflexiona sobre de qué tipo de vulnerabilidad de formateo de cadena se trata.

Código 4: Código vulnerable (fichero *fmt1.c*).

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>

#define BUFLEN 100

int main(int argc, char *argv[])
{
    char s[BUFLEN], name[BUFLEN];
```



```
    struct timeval tv;
    gettimeofday(&tv, NULL); // get current time for seed
    srand(tv.tv_usec);  // initialize random seed

    int canary1 = 0x0C0FFEE0,
            random_key = rand(),
            canary2 = 0xF00DF00D;

    // get username, and show it
    printf("Insert your name to continue: ");
    fgets(name, sizeof(name), stdin);
    name[strlen(name) - 1] = 0x00;
    sprintf(s, "Hello, %s! Take your best guess: ", name);
    printf(s);

    // get user number
    fgets(s, sizeof(s), stdin);
    int user_guess = atoi(s);

    if(random_key == user_guess)
        printf("#### WE HAVE A WINNER! ####\n");
    else
        printf("No luck this time :(. Try again!\n");

    return 0;
}
```

Después, fíjate en el Código 5 que también contiene una vulnerabilidad de formateo de cadena. Localiza la vulnerabilidad e intenta explotarla. Ahora, el objetivo es conseguir ejecutar la función `secret` para escribir el mensaje "YOU WIN!" por pantalla. Observa el código de la aplicación y piensa cómo conseguir explotar la vulnerabilidad. Reflexiona sobre de qué tipo de vulnerabilidad de formateo de cadena se trata.

Código 5: Código vulnerable (fichero *fmt2.c*).

```
#include <stdio.h>

#define BUFLEN 256

void secret()
{
    printf("YOU WIN!\n");
}

void echo()
{
    char buffer[BUFLEN];
    char *buf = 0;

    printf("Type your input: ");

    fgets(buffer, sizeof(buffer), stdin);
    printf("Input given: ");
    printf(buffer);
```



```
    printf("\n");
}

int main()
{
    int x, *p = &x; // variable p is a pointer to x

    x = 0xBAADF00D;
    printf("Address of x: 0x%08x\n", &x); // for helping you

    echo();
    if(x == 0x10)
        secret();

    return 0;
}
```

Enlaces de interés

Manual de uso del depurador GDB `http://ldc.usb.ve/~figueira/cursos/ci3825/taller/material/gdb.html`

Comandos básicos del depurador GDB `http://www.eis.uva.es/~fergay/III/enlaces/gdb.html`

Cheatsheet de radare2 `https://github.com/radare/radare2/blob/master/doc/intro.md`

