Buffer Overflow 3 Writeup

Código

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

__attribute__ ((force_align_arg_pointer)) // ignore this
void win() {
        system("cat flag.txt");
}

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

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

        printf("Esta vez es imposible...\n");
        fgets(answer, 0x20, stdin);
        printf("y ahora qué?\n");
}
```

En la tercera versión de este reto nos encontramos una vez más con un buffer overflow, esta vez de más tamaño, pero no se utiiza en ningún momento la variable `value`. De hecho, ni siquiera se llama a la función `win`, por lo que tendremos que buscar otra forma de explotar la vulnerabilidad.

Explotación

Aquí es cuando debemos recordar que en la pila no sólo se guardan variables locales: también las direcciones de retorno de las funciones. Recordemos que a la hora de llamar a una función en ensamblador se hace con la instrucción `call`, que introduce la dirección de retorno en la pila, de forma que más tarde la instrucción `ret` de la función a la que hemos llamado pueda retornar y continuar la ejecución. Por tanto, si con nuestro overflow conseguimos sobreescribir la dirección de retorno del `main` con la dirección de la función `win`, esta se ejecutará al terminar `main`.

Podemos probar a introducir una cadena suficientemente larga, y veremos que el programa termina con una violación de segmento. Si lo ejecutamos con gdb podemos apreciar lo siguiente:

La excepción ha ocurrido al ejecutarse la instrucción `ret`, que carga el valor de lo alto de la pila en el registro puntero de ejecución `rip`. Leyendo el valor del tope de la pila podemos ver que está nuestra entrada: un montón de 'A's.

Recordatorio: hemos usado `x/i $rip` para examinar como instrucción la dirección a la apunta el registro `rip` (instruction pointer), y `x/gx $rsp` para examinar como un valor de 8 bytes hexadecimal la dirección a la que apnta el registro `rsp` (stack pointer). Análogamente podríamos haber hecho `x/bx`, `x/hx` o `x/wx` para examinar como byte, half (2 bytes) o word (4 bytes).

Introduciendo una cadena más hetereogénea podemos ver lo siguiente:

Como 0x43 se corresponde con 'C' y 0x44 con 'D', de aquí deducimos que estamos sobreescribiendo la dirección de retorno con los bytes que hay a partir de la posición 22 de nuestra entrada.

Para obtener la dirección de la función `win` podemos hacer uso del comando:

```
objdump -d ./buffer_overflow3 | grep win
```

que nos imprimirá 0x401160. Es decir, queremos introducir 22 bytes arbitrarios, y después el número 0x401160. Ya que la codificación es little endian, y el puntero de instrucción ocupa 8 bytes, 0x401160 se codificará como: `60 11 40 00 00 00 00 00` (en hexadecimal). Finalmente podemos hacer uso de `echo` para producir nuestra entrada:

```
$ echo -e "12345678901234567890123456789012\x60\x11\x40\x00\x00\x00\x00\x00\x00" | ./buff
er_overflow3
Esta vez es imposible...
y ahora qué?
ETSIIT_CTF{m4st3r_0f_pwn31337}
Violación de segmento (`core' generado)
```

El programa termina con una violación de segmento, ya que para alterar el flujo de ejecución hemos corrompido la stack, y su contenido ya no es válido.

Conclusión

Cabe destacar la importancia de este tipo de vulnerabilidades: permiten redirigir el flujo de ejecución a una dirección controlada por el atacante. En este caso estamos ejecutando código que ya se encontraba en el binario, pero técnicas un poco más avanzadas permiten la ejecución de código arbitrario (por ejemplo, introduciendo el código por la entrada estándar, y reemplazando la dirección de retorno con la dirección de nuestra entrada, de forma que se ejecute el código que hemos introducido).

Autor del reto

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