Reto de reversing : Flexunil

Writeup realizado por: m0riart3

Nota: el binario compilado en la web UAM se compiló con una versión antigua de librerías, por lo que podrán encontrarse diferencias al compilarlo de manera local.

Flag 1

Descripción

Hemos recibido un fichero de naturaleza desconocida de una extraña entidad denominada Flexunil. Aseguran que han ocultado en él un mensaje importante para nosotros.

Solución

Para esta flag solo tendremos que usar el comando strings para ver la string “Flag 1: VUFNezU4NGUwNjZjY2I5Y2UwNTk4N2QwYmNlNmY3ZDNlMzIzfQ==”. Si decodeamos el mensaje en base64 obtendremos la flag

```
└─$ echo 'VUFNezU4NGUwNjZjY2I5Y2UwNTk4N2QwYmNlNmY3ZDNlMzIzfQ==' | base64 --decode
UAM{584e066ccb9ce05987d0bce6f7d3e323}
```

Flag 2

Descripción

Hemos explorado el archivo y parece ser que hay otro mensaje escondido. Sin embargo, nuestros analistas se encuentran demasiado ocupados redactando informes como para poder dedicarle siquiera unos minutos.

Nota: usar el mismo fichero que en Flexunil (Flag 1).

Solución

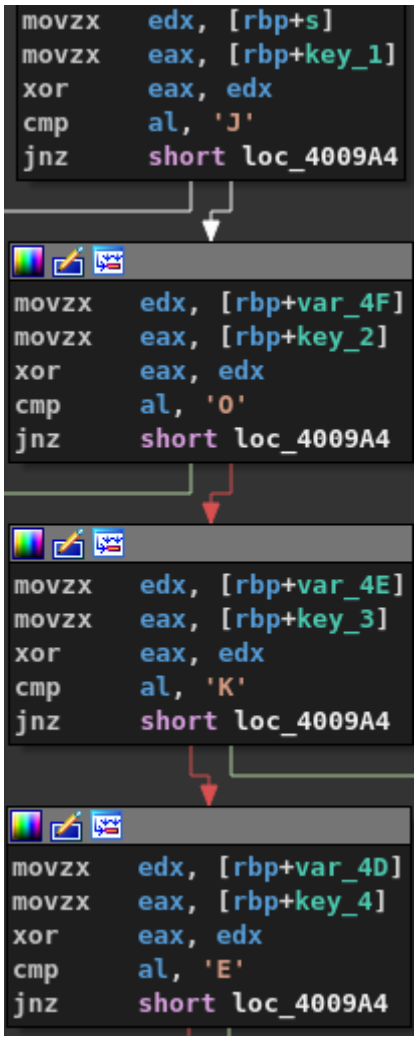
Lo primero que haremos, será abrir nuestro debugger/disassembler, en mi caso usaré IDA. Si abrimos la función main podremos ver como se inicializan los valores que posteriormente se usarán como claves para el xor.

```
mov     [rbp+key_1], 22h ; '"'
mov     [rbp+key_2], 26h ; '&'
mov     [rbp+key_3], 25h ; '%'
mov     [rbp+key_4], 31h ; '1'
```

Lo siguiente que vemos, son las llamadas a printf para mostrarnos la string “Second flag is hidden somewhere. Please check your guess:” y la llamada de fgets para almacenar nuestro input.

```
mov     edi, offset format ; "Second flag is hidden somewhere. Please"...
mov     eax, 0
call    _printf
mov     rdx, cs:stdin   ; stream
lea     rax, [rbp+s]
mov     esi, 33h ; '3'  ; n
mov     rdi, rax        ; s
call     fgets
```

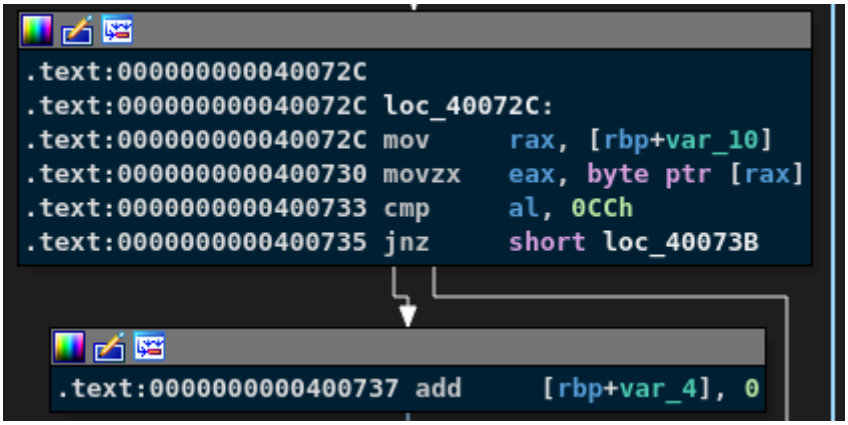
Posteriormente, se usan los valores que vimos anteriormente como claves de xor con los 4 primeros valores de nuestro input, se va comprobando uno a uno si el resultado de la operación coincide con el valor “joke”. Teniendo las claves, y el resultado, podemos calcular el valor restante de la operación, es decir, el valor que tenemos que introducir para que se cumpla la condición. En este caso, el valor será “hint”.



En caso de que el valor introducido sea “hint”, se llama a una función en la que se realiza un xor con la string “Gexpevz7xbcgbc7zbdc7ur75Vzvm~yp67Q{vp7~d7BVZlnxbe7~ygbcj57cx7dbtt” y el valor hexadecimal 0x17, posteriormente, se llama a la función “putchar” para imprimir el valor de la flag que debemos conseguir.

```
Second flag is hidden somewhere. Please check your guess: hint
Program output must be "Amazing! Flag is UAM{your input}" to succeed
```

Ahora que sabemos la string que debemos conseguir que el programa nos devuelva, nos toca reversear las distintas operaciones que realiza el programa antes de darnos un output. El programa llama a una función en la que se emplea una técnica de anti-debug. Este mecanismo consiste en buscar en el hexadecimal el valor “0xCC”, este valor se corresponde con una interrupción por “software breakpoint”. Existen diversas formas de saltarse este mecanismo, en mi caso he optado por parchear el valor que es sumado al contador, de esta forma nunca se aumentará el valor del contador de breakpoints encontrados.



Una vez saltado ese mecanismo, el siguiente paso será reversear el algoritmo para conseguir el output deseado. Podemos ver como por cada iteración del bucle selecciona un valor en forma de key para hacerle xor, tal y como ocurría anteriormente, utiliza la instrucción shl para desplazar 2 bits a la izquierda a cada letra de la clave que hemos introducido, luego, le añade el valor del carácter usando add. Una vez realizado todo esto, multiplica ese valor por 4 y lo suma al valor original, posteriormente, realiza el xor que hemos comentado anteriormente con la clave que carga en memoria, y selecciona los dos últimos bits del valor resultante.

```
mov     eax, [rbp+var_14]
movsxd  rdx, eax
mov     rax, [rbp+var_38]
lea     rcx, [rdx+rax]
mov     eax, [rbp+var_14]
cdqe
movzx   esi, clave[rax]
mov     eax, [rbp+var_14]
movsxd  rdx, eax
mov     rax, [rbp+var_28]
add     rax, rdx
movzx   eax, byte ptr [rax]
mov     edx, eax
mov     eax, edx
shl     eax, 2
add     eax, edx
lea     edx, ds:0[rax*4]
add     eax, edx
xor     eax, esi
```

Ahora que sabemos las operaciones que realiza, haremos un script en python que nos dé los caracteres que debemos introducir para que nos dé el output necesario. En mi caso, lo he hecho comprobando cada posibilidad hasta que el output coincida con la letra deseada.

```
key = [0xEA,0xA4,0xD1,0xBE,0x9F,0xFC,0x4a,0x91,0xB1,0x3e,0xA5,0x4c,0xAE,0x58,0x11,
0xD8,0x58,0xC4,0x7,0xDF,0x2,0x1,0x94,0xC,0x90,0xC2,0xF8,0x8C,0xB4,0x97,0xC,0x5]
message = "Amazing! Flag is UAM{your input}"
flag = ""

for key , l in zip(key,message):
    bytes_message = hex(ord(l))[-2:]
    for i in range(0,254):
        c = i << 2
        c += i
        c2 = c*4
        c = int(c) + c2
        c = hex(c ^ key)
        bytes = c[-2:]
print(flag)
```

Una vez tenemos el script, lo ejecutamos y obtenemos la flag que debemos introducir en el binario.

```
└─$ python3 solver.py
c10dfb509815188c896ba83a2292d288
```