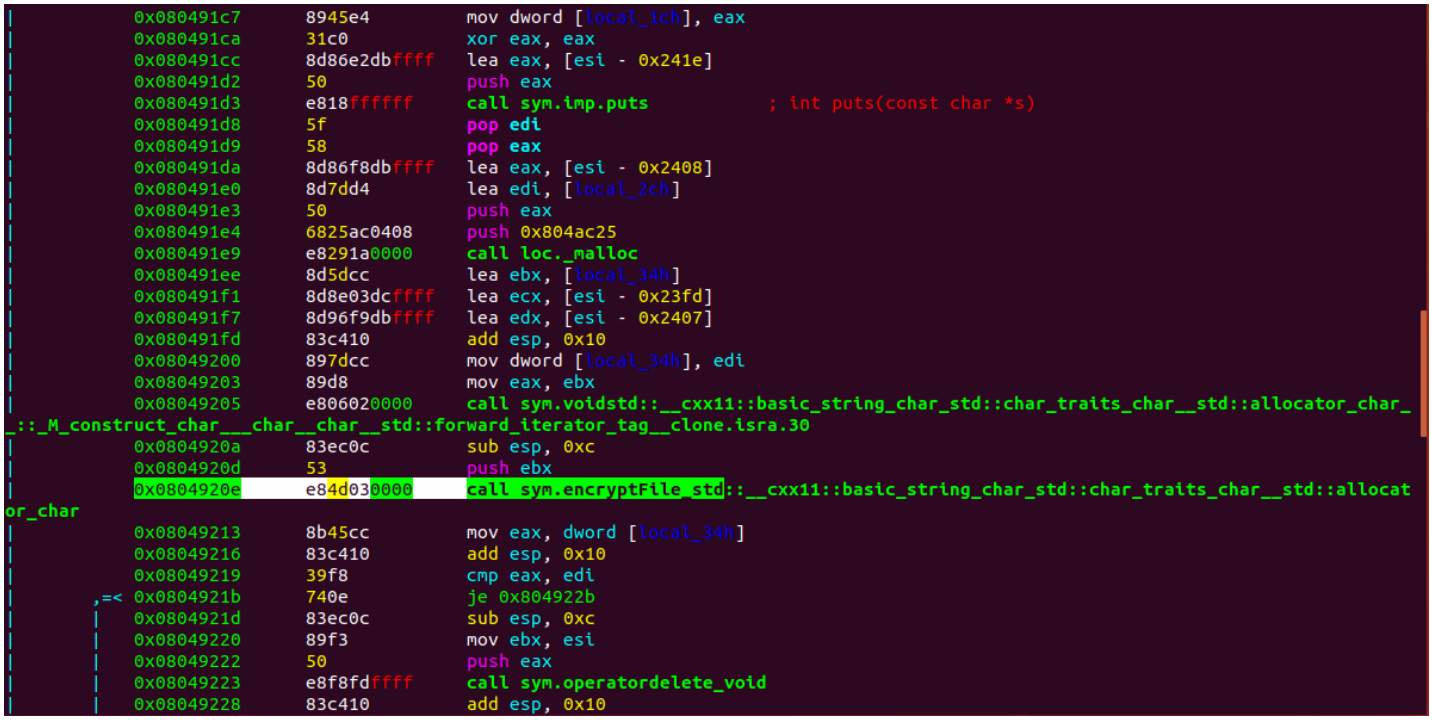
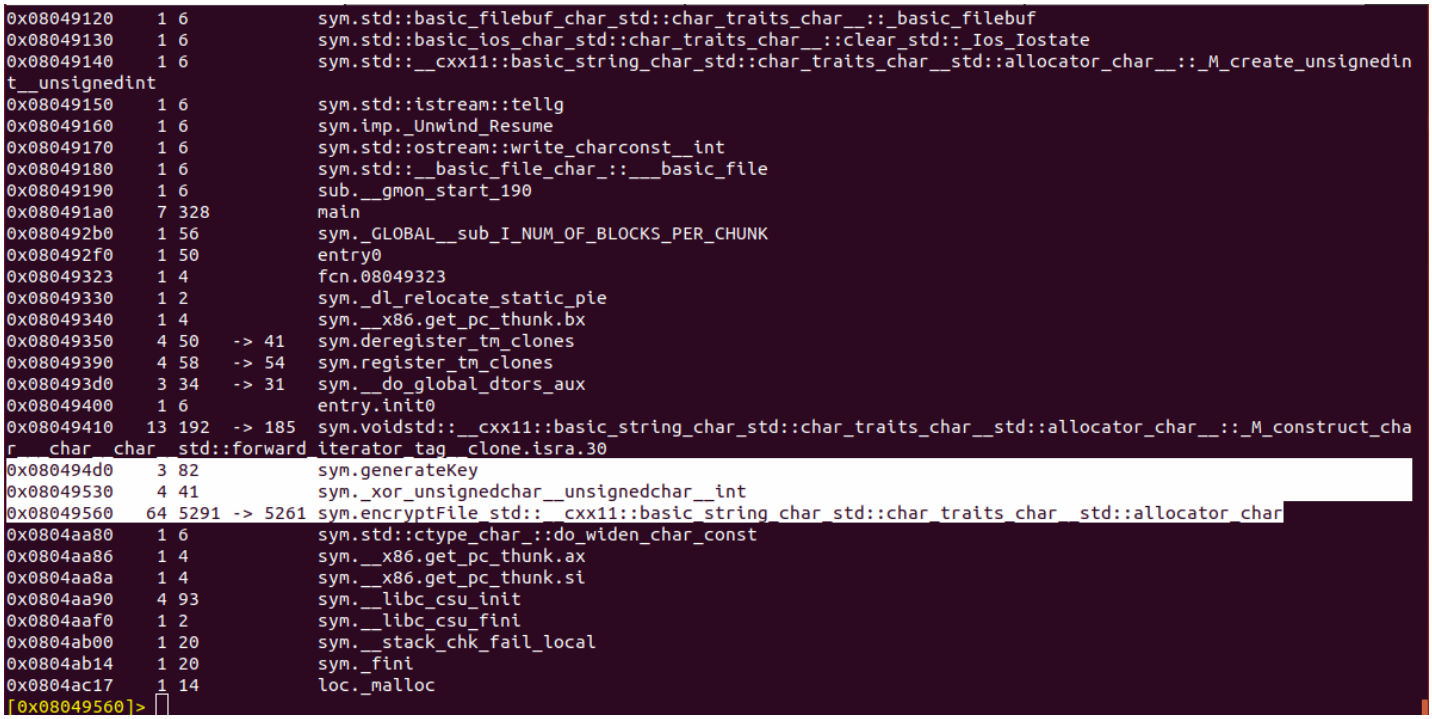
Write Up

Cargamos el binario en radare2 y analizamos (aaaa).

La función main es sencilla, y se llama a una función encryptFile que recive como parámetro un std::string



Podemos también listar todas las funciones del binario con "afl"



Y podemos ver dos funciones importantes: encryptFile y generateKey.

Si vemos la función generateKey, podemos ver que se llama desde encryptFile:

```
[0x080494d0]> pdf
/ (fcn) sym.generateKey 82
|   sym.generateKey ();
|           ; CALL XREF from sym.encryptFile_std::__cxx11::basic_string_char_std::char_traits_char__std::allocator_char (0x8049be1)
|           0x080494d0      57              push edi
|           0x080494d1      56              push esi
|           0x080494d2      53              push ebx
|           0x080494d3      e868feffff      call sym.__x86.get_pc_thunk.bx
|           0x080494d8      81c3283b0000    add ebx, 0x3b28             ; '(;'
|           0x080494de      83ec0c          sub esp, 0xc
|           0x080494e1      6a37            push 0x37                   ; '7' ; 55
|           0x080494e3      8db3c0010000    lea esi, [ebx + 0x1c0]      ; 448
|           0x080494e9      8dbbe0010000    lea edi, [ebx + 0x1e0]      ; 480
|           0x080494ef      e81cfbffff      call sym.imp.srand          ; void srand(int seed)
|           0x080494f4      58              pop eax
|           0x080494f5      8d8380dbffff    lea eax, [ebx - 0x2480]
|           0x080494fb      5a              pop edx
|           0x080494fc      50              push eax
|           0x080494fd      6825ac0408      push 0x804ac25
|           0x08049502      e810170000      call loc._malloc
|           0x08049507      83c410          add esp, 0x10
|           0x0804950a      8db600000000    lea esi, [esi]
|           ; CODE XREF from sym.generateKey (0x804951c)
|       .-> 0x08049510      e85bfbffff      call sym.imp.rand           ; int rand(void)
|       :   0x08049515      3006            xor byte [esi], al
|       :   0x08049517      83c601          add esi, 1
|       :   0x0804951a      39fe            cmp esi, edi
|       `=< 0x0804951c      75f2            jne 0x8049510
|           0x0804951e      5b              pop ebx
|           0x0804951f      5e              pop esi
|           0x08049520      5f              pop edi
\           0x08049521      c3              ret
[0x080494d0]>
```

```
[0x08049be1]> pd 20
|           0x08049be1      e8eaf8ffff      call sym.generateKey
|           0x08049be6      8b87c0010000    mov eax, dword [edi + 0x1c0] ; [0x1c0:4]=-1 ; 448
|           0x08049bec      83fe00          cmp esi, 0
|           0x08049bef      c785f8fcffff.   mov dword [local_308h], 0x61707865 ; 'expa'
|           0x08049bf9      c7850cfdffff.   mov dword [local_2f4h], 0x3320646e ; 'nd 3'
|           0x08049c03      c78518fdffff.   mov dword [local_2e8h], 0
|           0x08049c0d      c7851cfdffff.   mov dword [local_2e4h], 0
|           0x08049c17      c78520fdffff.   mov dword [local_2e0h], 0x79622d32 ; '2-by'
|           0x08049c21      8985fcfcffff    mov dword [local_304h], eax
|           0x08049c27      8b87c4010000    mov eax, dword [edi + 0x1c4] ; [0x1c4:4]=-1 ; 452
|           0x08049c2d      c78534fdffff.   mov dword [local_2cch], 0x6b206574 ; 'te k'
|           0x08049c37      c78510fdffff.   mov dword [local_2f0h], 0x56495649 ; 'IVIV'
|           0x08049c41      c78514fdffff.   mov dword [local_2ech], 0x56495649 ; 'IVIV'
|           0x08049c4b      898500fdffff    mov dword [local_300h], eax
|           0x08049c51      8b87c8010000    mov eax, dword [edi + 0x1c8] ; [0x1c8:4]=-1 ; 456
|           0x08049c57      898504fdffff    mov dword [local_2fch], eax
|           0x08049c5d      8b87cc010000    mov eax, dword [edi + 0x1cc] ; [0x1cc:4]=-1 ; 460
|           0x08049c63      898508fdffff    mov dword [local_2f8h], eax
|           0x08049c69      8b87d0010000    mov eax, dword [edi + 0x1d0] ; [0x1d0:4]=-1 ; 464
|           0x08049c6f      898524fdffff    mov dword [local_2dch], eax
[0x08049be1]>
```

Además, si nos fijamos en la segunda imagen (call a generateKey desde encryptFile), podemos ver la variable mágica de Salsa20 que se está usando para inicializar la clave de Salsa20. Aquí podemos darnos cuenta de que el algoritmo es Salsa20 (buscando en Google "expand 3 2-byte k").

Incluso podemos ver como se inicializa también la clave usando el IV "IVIVIVIV" de 8 bytes. Solo falta ver como se genera la clave de cifrado.

En la imagen de generateKey, podemos ver que se llama a srand pasando 55 como parámetro, y después se usa rand para generar un número aleatorio y hascer un XOR con un elemento de una array (sabemos que es un array porque es un bucle en el que esi tiene la dirección del array y la va aumentando 1 byte cada iteración; para acceder a cada elemento al hacer el xor byte [esi], al). Al principio se inicializan: l

0x080494e3 8db3c0010000 lea esi, [ebx + 0x1c0] ; 448 | 0x080494e9 8dbbe0010000 lea edi, [ebx + 0x1e0] ; 480

edi contiene la dirección del último elemento, de forma que se compara esi y edi en cada iteración para salir del bucle. Que en lugar de un MOV se utilice un XOR huele raro.. Da lugar a pensar que en el array en el que se guarda la clave generada podría haber algo.

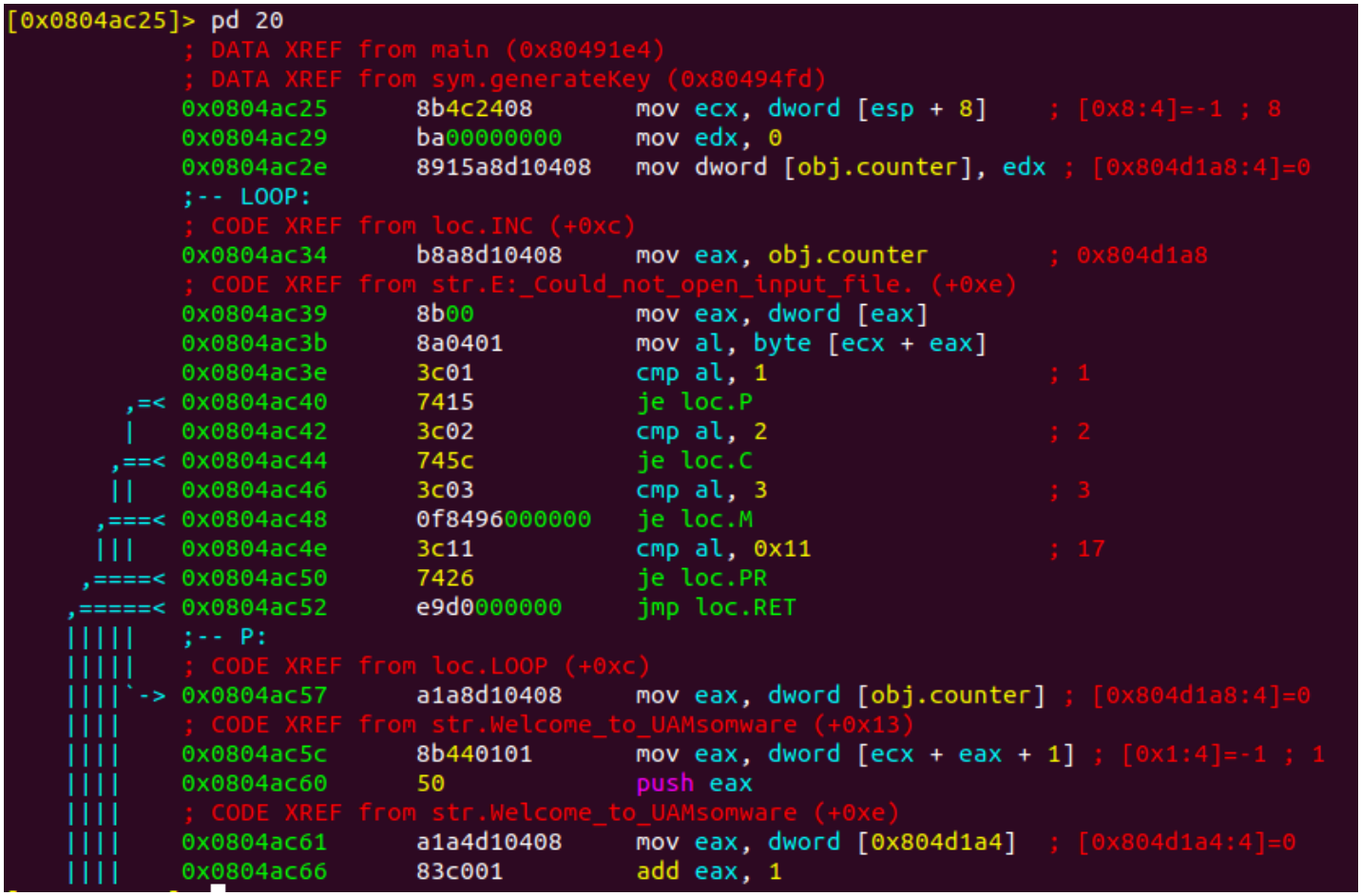
Aparentemente la semilla es 55. Pero también se llama a la función _malloc, que recibe dos parámetros en lugar de 1 que es lo normal, y además son parámetros extraños (son direcciones del binario, no una cantidad para reservar memoria..)

Según radare2 e ida, la función _malloc es esta:

[0x0804ac17]> pdf ;-- section..TEXT: / (fcn) loc.*malloc 14* | *loc.*malloc (int arg*4h); | ; arg int arg*4h @ esp+0x4 | ; CALL XREF from main (0x80491e9) | ; CALL XREF from sym.generateKey (0x8049502) | 0x0804ac17 6a20 push 0x20 ; 32 ; [18] -r-- section size 278 named .TEXT | 0x0804ac19 e882e3ffff call sym.imp.malloc ; void *malloc(size*t size) | 0x0804ac1e 59 pop ecx | 0x0804ac1f 8b4c2404 mov ecx, dword [arg*4h] ; [0x4:4]=-1 ; 4 | ; CODE XREF from str..uam (+0x1) | 0x0804ac23 51 push ecx \ 0x0804ac24 c3 ret

Se llama a la función malloc real para reservar 32 bytes (0x20). Sin embargo, las últimas dos instrucciones son muy extrañas, porque primero se hace push ecx, y después un ret. ret hace un pop de la pila y salta a ese valor, por lo que estas instrucciones se traducen en jmp ecx.

Como se puede ver, justo antes de hacer el push, se hace mov ecx, dword [arg_4h] esto significa que se mueve a ecx el primer parámetro de la función, es decir, la dirección que veiamos en la llamada (0x804ac25)



Como se puede ver, en esa dirección hay lógica para procesar algo y tomar diferentes decisiones según el procesamiento.

Por el momento se puede seguir sin entrar en detalle a analizar lo que ocurre en ese código. Aunque se lo ojeamos por encima, podemos ver que se dan nombres a las diferentes partes del código (LOOP, P, C, CC, PR, RET, INC). Y curiosamente, esas partes hacen: P: hace un push eax C: hace un jmp eax usando el truco de push ret; pero añadiendo un push loc.CC antes, lo que significa que se usa para llamar funciones y la dirección de retorno es loc.CC. RET retorna de la función _malloc INC: incrementa la variable counter tanto como indique ebx; y todos los labels acaban poniendo un valor en ebx y llamando a INC..

Con este simple vistazo, podemos asumir que es una especio de VM que procesa un buffer y realiza acciones en base a ese buffer. Algunos opcodes de la VM serán "call" (loc.C) para llamar funciones, "push" (loc.P) para pushear elementos en el stack..

Esta función debe estar siendo utilizada para llamar de forma oculta a alguna otra función que inicializa de alguna forma el array de la clave, y por eso se hace un XOR en lugar de un simple MOV.

A partir de aquí la idea es hacer debugging con breakpoints en _malloc, para ver si lo que pensamos es correcto y ver que funciones está llamando de forma "oculta". También breakpoints en el bucle del XOR a la clave generada, para ver que contiene.

Con eso veremos que _malloc efectivamente se usa para llamar de forma oculta a time, srand y la función

_xor, de forma que nos damos cuenta de que srand se usa para inicializar el generador con el timestamp devuelto por time, y _xor para inicializar la clave con los primeros 32bytes de la función _malloc.

Después, en el bucle XOR de generateKey, lo que esta haciendo es: key[i] = key[i] ^ rand()

Y como ya hemos visto, key esta inicializado con los primeros 32 bytes de la función _malloc.

En este punto sabemos el algoritmo de cifrado, el IV y como se genera la clave para el fichero. Para resolver el reto y descifrar el fichero hay dos opciones principales:

- Hacerse su propio programa C para generar y descifrar el fichero; o al menos para generar la clave con srand y rand, y después usar un script python o similar.
- Reutilizar el propio programa del reto poniendo breakpoint en srand y modificando el argumento con el timestamp de la última modificación del fichero (o bien usar frida para hookear srand).

Para obtener la última modificación del fichero se puede usar: stat -c "%Y" flag.txt.uam

Eso nos dará el timestamp a usar para generar la clave de descifrado.