UAM - Marvel Episode 2

Julian J. M. 25/01/2019

Nos encontramos con una máquina virtual que ha sido víctima de un ransomware muy particular. Encontramos en el escritorio el ejecutable y un fichero cifrado, flag.txt.uam. Nuestro objetivo es descifrar dicho fichero.

Un rápido análisis de ese ejecutable revela que lee un fichero flag.txt, lo cifra, y lo guarda en flag.txt.uam. Indica, además, una fecha en formato epoch (unix timestamp).

Al comparar el fichero original con el cifrado, nos damos cuenta de que tienen exactamente la misma longitud, sea la que sea. Sacamos algunas conclusiones:

- No hay padding, seguramente sea un cifrado de flujo (stream cipher), y no por bloques.
- No se guarda la clave de cifrado en el propio fichero .uam, por lo que tiene que poder generarse de nuevo a la hora de descifrar.
- La fecha de modificación del fichero .uam es la misma que la aplicación indica por pantalla.

Con esta información, podemos suponer que la clave de cifrado depende de la hora a la que se ejecuta el programa. Un stream cipher funciona haciendo xor del plaintext, es decir, los datos del fichero original, con un keystream, la secuencia de bytes que se generan a partir de la clave. Debería ser posible, por tanto, descifrar un fichero .uam volviéndolo a cifrar con exactamente la misma clave.

Comprobamos nuestra teoría:

```
$ echo “Hola mundo” > flag.txt
$ date +%s                          # obtenemos la fecha actual
1548433151
$ sudo date -s @1548433151 ./UAMson
Fri Jan 25 08:19:11 PST 2019
Welcome to UAMsomware

Time: 1548433151
$ hd flag.txt
00000000  48 6f 6c 61 20 6d 75 6e  64 6f 0a                 |Hola mundo.|
0000000b
$ hd flag.txt.uam
00000000  df 8d e5 ae eb 9a a1 ef  41 93 73                 |........A.s|
0000000b
$ mv flag.txt.uam flag.txt
```

```
$ sudo date -s @1548433151 ; ./UAMsom
Fri Jan 25 08:19:11 PST 2019
Welcome to UAMsomware

Time: 1548433151

$ hd flag.txt.uam
00000000  48 6f 6c 61 20 6d 75 6e  64 6f 0a                 |Hola mundo.|
0000000b
```

Bien!. Hemos recuperado nuestro plaintext original y confirmado nuestra teoría. Ahora necesitamos conocer el momento exacto en que se cifró el fichero del reto, y así poder volver al pasado y repetir la misma operación. Great Scott!

Una forma sencilla de obtener este dato, es con el comando stat, que nos da directamente la fecha de modificación del fichero en formato epoch:

```
$ stat -c %Y flag.txt.uam
1547554536

$ mv flag.txt.uam flag.txt

$ sudo date -s @1547554536 ; ./UAMsom
Tue Jan 15 04:15:36 PST 2019
Welcome to UAMsomware

Time: 1547554536

$ hd flag.txt.uam
00000000  2b 32 30 2b 32 33 34 2b  33 33 2b 32 30 2b 35 35  |+20+234+33+20+55|
00000010  2b 37 2b 32 30 2b 37 2b  39 36 38 2b 33 35 35 2b  |+7+20+7+968+355+|
00000020  38 38 36 2b 33 35 35 2b  35 36 2b 33 35 35 2b 37  |886+355+56+355+7|
00000030  2b 32 30 2b 33 35 36 2b  39 36 38 2b 33 34 2b 32  |+20+356+968+34+2|
00000040  31 38 2b 33 35 35 2b 35  35 2b 33 35 35 2b 33 34  |18+355+55+355+34|
00000050  2b 32 30 2b 34 35 2b 32  30 2b 35 30 34 2b 33 35  |+20+45+20+504+35|
00000060  35 2b 33 39 2b 38 38 36  2b 33 39 0a              |5+39+886+39.|
0000006c

$ cat  flag.txt.uam
+20+234+33+20+55+7+20+7+968+355+886+355+56+355+7+20+356+968+34+218+355+55+355
+34+20+45+20+504+355+39+886+39
```

Parece que hemos obtenido el plaintext original. Solo falta descifrar ese código, que tras un tuit que sugería estar relacionado con las iniciales de los países, nos lleva a pensar que se puede tratar de prefijos de teléfono internacionales. +20 Egypt, +34 Spain, etc...

Mensaje: ENFEBREROATACAREMOSLABASEDEHAITI

Descifrando a mano

Esta forma de descifrar el fichero ha sido, creo, la que hemos usado todos los participantes de este reto. Es la más intuitiva y sencilla de llevar a cabo. Sin embargo, ésta no era forma de resolverlo esperada por el creador del reto, así que vamos a intentar hacerlo a su manera. Disclaimer: mucho de lo que viene a continuación no se me habría pasado por la cabeza sin las pistas recibidas X-D.

Sacando los strings, vemos una cadena “expand 32-byte k”, que según la wikipedia es una constante utilizada en el cifrado de flujo Salsa20. Utiliza una clave de 256bit (32 bytes) y un nonce (o IV, que debería ser único entre cada cifrado) de 64bits (8 bytes).

Decíamos antes que el cifrado depende del tiempo en que se ejecute el programa. Un vistazo rápido en radare a la función generateKey() nos indica que la llamada a srand se realiza con un valor estático (55, o 0x37). Algo se nos escapa.
Ejecutando el progama con ltrace, vemos dos llamadas a srand, una con 55 y otra con el resultado de la función time(), que devuelve el epoch actual. Hay gato encerrado.

La función _malloc en generateKey() no es la funcion de la libc que reserva memoria, es una función definida localmente y que recibe dos parámetros: Una dirección de memoria a la que saltar, y un buffer estático.

Esta dirección a la que salta contiene una especie de máquina virtual. Decodifica las instrucciones y datos que se le pasan en ese buffer, y cuando finaliza retorna a la dirección siguiente a la llamada a _malloc, continuando como si no hubiese pasado nada.

Esta máquina virtual tiene un contador, que indica la siguiente instrucción a leer. Las instrucciones implementadas son:

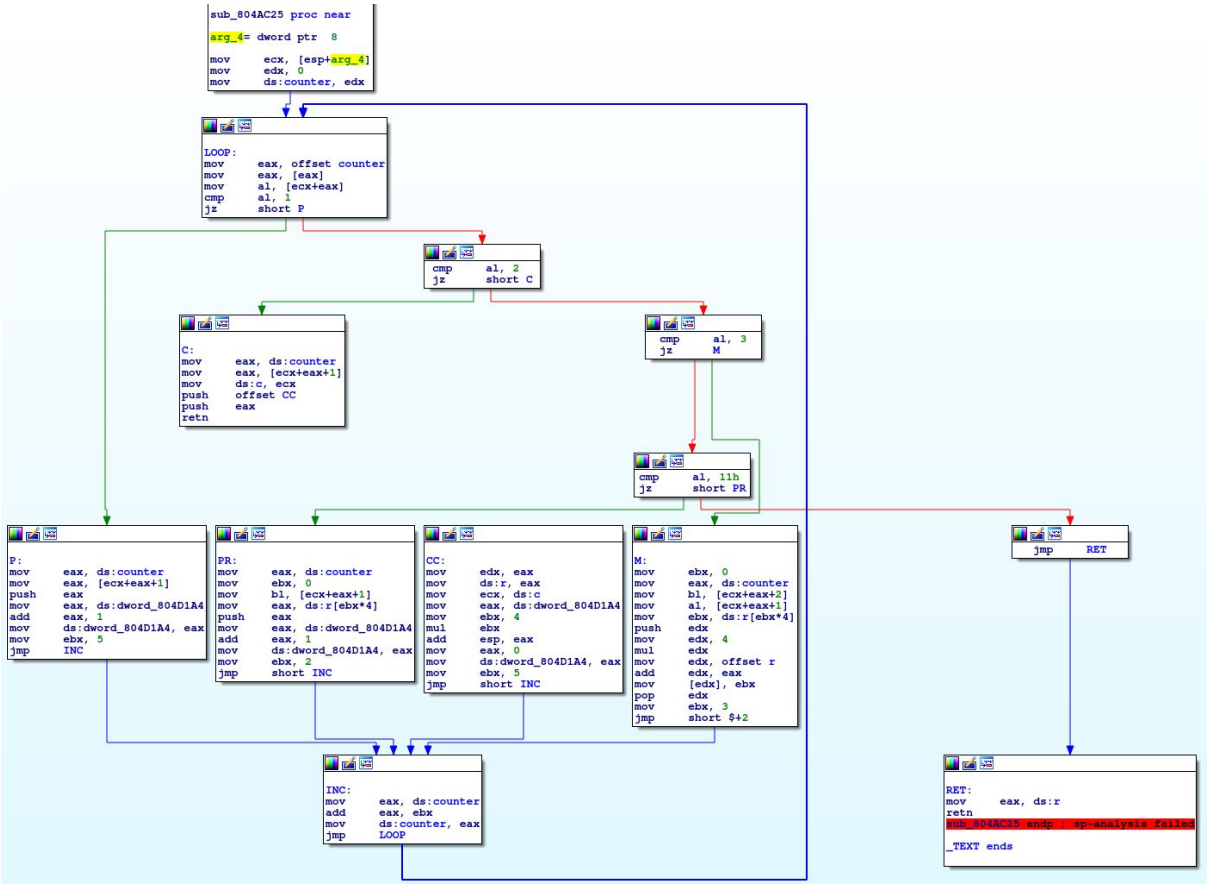
- 1 (Push):
    - Añade un valor (los siguientes 4 bytes) a la pila
    - Incrementa una variable con el número de parámetros
    - Incrementa contador en 5 (1 + 4)

- 2(Call):
    - Llama al offset definido en los siguientes 4 bytes
    - Almacena el resultado, eax, al inicio del vector r
    - Elimina de la pila los parámetros utilizados. (esp=esp+4*nParams)
    - Incrementa contador en 5

- 3 (Mem?): Manipula el vector de resultados
    - El índice de destino es el siguiente byte (contador+1)
    - El índice de origen en contador+2
    - r[destino] = r[origen]
    - Incrementa contador en 3

- 0x11 (Push Result):
    - El siguiente byte indica el índice en el vector de resultados r
    - Añade r[indice] a la pila
    - Incrementa contador en 2

- Cualquier otro valor, retorna, devolviendo en eax el primer valor del vector de resultados, r[0]

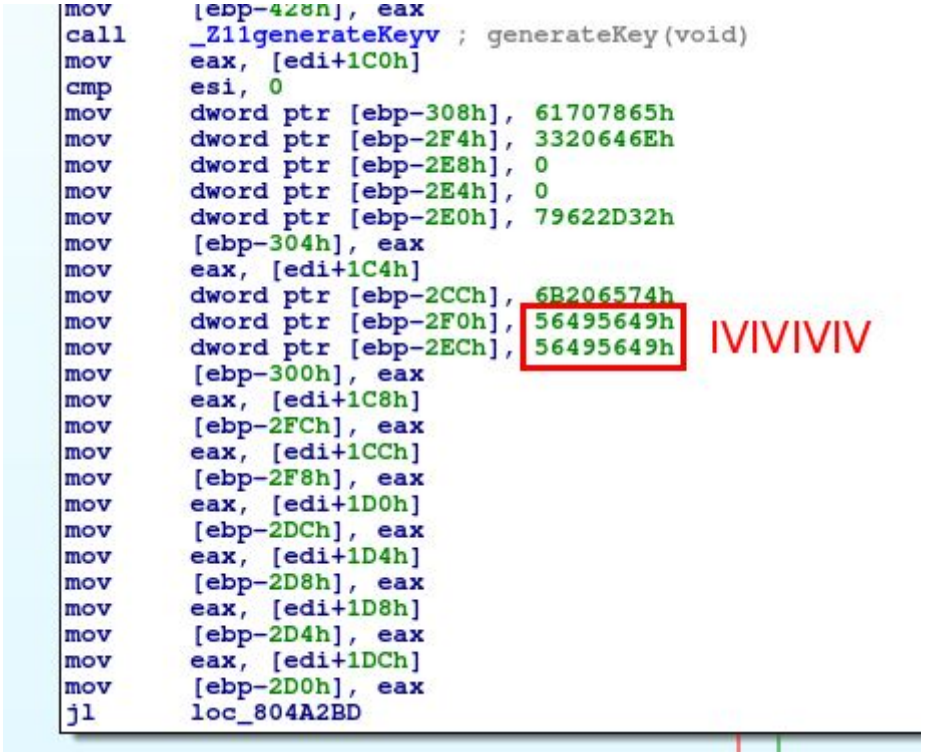


El código que se le pasa en la función generateKey es el siguiente:

```
1     0                     ; push 0
2     offset _time          ; call time(0). Deja en r[0] el resultado
3     1     0               ; r[1] = r[0]
11    0                     ; push r[0]
2     offset _srand         ; call srand(r[0])
1     0x20                  ; push 0x20
1     offset _malloc        ;
1     offset fileKey        ;
2     offset _xor           ; _xor( filekey, _malloc, 32)
0                           ; ret
```

Cabe destacar que la llamada a la función _xor inicializa el vector fileKey, al que posteriormente, en generateKey, se le hará un xor con los valores obtenidos de rand(). Lo que hace el código es usar los primeros 32 bytes de la propia función _malloc como base para generar la key de cifrado. Poca broma.

A partir de aquí, con la clave generada, solo nos falta el nonce o IV. Sinceramente no he conseguido sacar dónde se inicializa, y tengo que fiarme de las pistas recibidas. De haber escogido otro IV menos obvio, dudo que hubiese dado con él :)



Ya tenemos todos los datos, así que solo falta codificar un programa que descifre el fichero de marras. He tenido que hacerlo en C para utilizar el mismo generador de números pseudoaleatorios, ya que el que usa Python es distinto. La librería de cifrado Salsa20 utilizada fue https://github.com/alexwebr/salsa20

```
$ ./descifraUAM flag.txt.uam
File mtime: 1547554536

+20+234+33+20+55+7+20+7+968+355+886+355+56+355+7+20+356+968+34+218+355+55+355
+34+20+45+20+504+355+39+886+39
```

Y eso es todo. El código de descifraUAM a continuación.

Julian J. M.
Telegram: @julianjm
Email: julianjm@gmail.com

```c
/*
   Descifrador para el reto Marvel - Episodio 2 - UAM
   Julian J. M.
   Telegram @julianjm
   Email: julianjm@gmail.com

   Librería de cifrado utilizada: https://github.com/alexwebr/salsa20

   Compilar con:  gcc descifraUAM.c salsa20/salsa20.c -o descifraUAM
*/

#include<stdio.h>
#include<sys/stat.h>
#include<stdlib.h>
#include<unistd.h>

#include"salsa20/salsa20.h"

// Bytes que se usan como base para generar la clave de cifrado. A estos 32 bytes se les hace un xor con 32 rand().
// Son los 32 primeros bytes del código de la función _malloc
unsigned char key[32]={106,32,232,130,227,255,255,89,139,76,36,4,81,195,139,76,36,8,186,0,0,0,0,137,21,168,209,4,8,184,168,209};

int main(int argc, char**argv) {
   if (argc<2) {
       printf("Usage: %s <uamfile> [epoch]\n", argv[0]);
       return 1;
   }
   char * filename = argv[1];

   // Obtenemos información del fichero como su fecha de modificacioń, tamaño, etc
   // (y de paso si existe)
   struct stat statbuf;
   if (stat(filename, &statbuf)<0) {
       printf("Error reading file %s\n", filename);
   }
   fprintf(stderr, "File mtime: %lu\n",statbuf.st_mtime);

   if (argc<3) {
       srand(statbuf.st_mtime);
   } else {
       unsigned int seed = atoi(argv[2]);
       srand(seed);
       fprintf(stderr, "Using custom random seed: %u\n", seed);
   }

   //Generamos la clave de cifrado, haciendo xor con los valores obtenidos de rand().
   for(int i=0; i<32; i++) {
       key[i] ^= rand();
   }
```

```
    //Leemos el fichero a memoria
    unsigned int length = statbuf.st_size;
    char *data = malloc(length);
    FILE *fh = fopen(filename, "rb");
    fread(data, length, 1, fh);
    fclose(fh);

    //Ciframos usando la clave generada y el vector de inicialización
    s20_crypt(key, S20_KEYLEN_256, "IVIVIVIV", 0, data, length);

    //Sacamos por pantalla (stdout) los datos
    fprintf(stderr,"\n");
    write(1, data, length);
    fprintf(stderr, "\n");
}
```