WRITE-UP MARVEL – CTF UAM - HISPASEC

EPISODIO-2

Elaborado por: Arsenics

Misión:

Después de la explotación del programa de reclutamiento infiltramos a un informático como agente de Hydra. Tras unos días sin noticias, nos ha notificado que tiene en su poder el PC que utilizaban para las comunicaciones de los ataques, pero que este se ha visto afectado por un ransomware desconocido.

Tu misión es conseguir desencriptar el archivo principal, entender las comunicaciones que realizan y conseguir la fecha del próximo ataque.

Mucha suerte soldado.

Nick Furia.

Enlace de descarga de la VM: https://drive.google.com/open?id=1AvXC-ywgpmPFTaQKlk2Wklx5eD_xBNUj

Info: La flag tiene el formato UAM{md5 de la frase en mayúsculas y sin espacios}

Bibliografía:

-Radare2: https://github.com/radare/radare2

-Cyberchef – http://icyberchef.com/

-Faketime: https://github.com/wolfcw/libfaketime

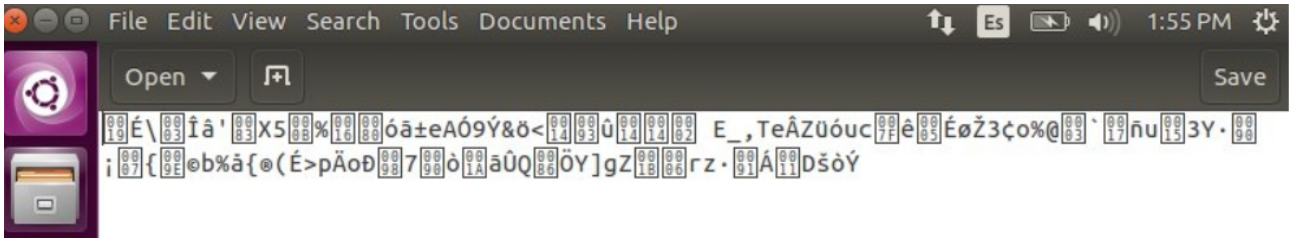
-Virtualbox: https://www.virtualbox.org/

Walktrough:

La descarga nos deja un OVF que montamos en virtualbox y un text con unas credenciales

Usuario: hydrauser Pass: hailhydra

Encontramos un programa UAMsom y un flag.txt.uam Retiramos la extensión y abrimos el arhivo de texto y nos aparece un cifrado un tanto extraño.



Para estudiar el programa miramos q tipo de archivo con el comando: file UAMsom:

```
UAMsom: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically li
nked, interpreter /lib/ld-linux.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=2e29b1a
7bae71f06dea2f99a69f706d48af0335c, not stripped
```

y ejecutamos el programa ./UAMsom viendo que nos devuelve un timestamp. Esto me lleva a probar de meter un delay para ver si en vez del timestamp devolviese otra respuesta. Pero no...por aquí no es el camino. Compruebo a ver si por casualidad devolviese un segmentation fault.
Python -c 'AAAAAAA' | ./UAMsom

```
BAA9AA0ABBABCABDABEABFABGABHABIABJABKABLABMABNABOABPABQABRABSABTABUABVABWABXABYABZABaABbABcABdABeABfABgABhABiABjABkABlABmABnABoABpABqABrABsABtABuABvABwABxAByABzAB1AB2AB3AB4AB5AB6AB7ABBAB9ABBACBACCACDACEACFACGACH
ACIACJACKACLACMACNACOACPACQACRACSACTACUACVACWACXACYACZACaACbACcACdACeACfACgACHACiACjACkAClACmACnACoACpACqACrACsACtACuACvACwACxACyACzAC1AC2AC3AC4AC5AC6AC7ACBAC9ACBADBADCADDADEADFADGADHADIADJADKADLADMADNADOADPADQA
D' | ./UAMsom
Welcome to UAMsomware

Time: 1547593664
Traceback (most recent call last):
  File "<string>", line 1, in <module>
NameError: name 'AAABAACAADAAEAAFAAGAAHAAIAAJAAKAALAAMAANAAOAAPAAQAARAASAATAAUAAVAAWAAXAAYAAZAAaAAbAAcAAdAAeAAfAAgAAhAAiAAjAAkAAlAAmAAnAAoAApAAqAArAAsAAtAAuAAvAAwAAxAAyAAzAA1AA2AA3AA4AA5AA6AA7AA8AA9AA0ABBABCABDA
BEABFA' is not defined
```

Vaya por aquí tampoco hay nada. Recogemos toda la información posible para enfrentarnos al caso con todas las armas. En primer lugar revisamos las protecciones que tiene el programa: **./checksec -f UAMsom**

```
RELRO           STACK CANARY      NX            PIE           RPATH      RUNPATH      Symbols       FORTIFY Fortified      Fortifiable  FILE
Partial RELRO   Canary found      NX disabled   No PIE        No RPATH   No RUNPATH   148 Symbols   Yes     0              2            /root/Downloads/UAMsom
```

Vemos que tiene Partial RELRO y Canary activado. Entonces me decido a abrir radare2 para realizar un análisis de las funciones. Con el comando aaaa y sacando el afll:

```
0x08049190    6     1     0     1     3 0x08049190     6 0x08049196     0     0     0     1     0 sub.__gmon_start_190
0x080491a0  328     7     8     5   170 0x080491a0   328 0x080492e8    16     5     1     1   136 main
0x080492b0   56     1     0     1    32 0x080492b0    56 0x080492e8     3     0     0     0    28 sym._GLOBAL__sub_I_NUM_OF_BLOCKS_PER_CHUNK
0x080492f0   50     1     0     1    30 0x080492f0    50 0x08049322     2     0     0     0    28 entry0
0x08049323    4     1     0     1     4 0x08049323     4 0x08049327     0     0     0     1     0 fcn.08049323
0x08049330    2     1     0     1     3 0x08049330     2 0x08049332     0     0     0     0     0 sym._dl_relocate_static_pie
0x08049340    4     1     0     1     4 0x08049340     4 0x08049344     0     0     0     7     0 sym.__x86.get_pc_thunk.bx
0x08049350   41     4     4     4    24 0x08049350    50 0x08049382     0     0     0     1    24 sym.deregister_tm_clones
0x08049390   54     4     4     4    31 0x08049390    58 0x080493ca     0     0     0     1    24 sym.register_tm_clones
0x080493d0   31     3     2     3    18 0x080493d0    34 0x080493f2     1     0     0     0     8 sym.__do_global_dtors_aux
0x08049400    6     1     1     0     6 0x08049400     6 0x08049406     0     0     0     0     4 entry1.init
0x08049410  267    15    20     9   137 0x08049410   274 0x08049522     9     5     0     2    72 sym.voidstd::__cxx11::basic_string_char_std::
r__std::forward_iterator_tag__clone.isra.30
0x08049530   41     4     5     3    23 0x08049530    41 0x08049559     0     0     3     0     8 sym._xor_unsignedchar__unsignedchar__int
0x08049560 5261    64    94    32  1419 0x08049560  5291 0x0804aa0b    54   129     4     1  1180 sym.encryptFile_std::__cxx11::basic_string_ch
0x0804aa80    6     1     0     1     4 0x0804aa80     6 0x0804aa86     0     0     1     0     0 sym.std::ctype_char_::do_widen_char_const
0x0804aa86    4     1     0     1     4 0x0804aa86     4 0x0804aa8a     0     0     0     1     0 sym.__x86.get_pc_thunk.ax
0x0804aa8a    4     1     0     1     4 0x0804aa8a     4 0x0804aa8e     0     0     0     1     0 sym.__x86.get_pc_thunk.si
0x0804aa90   93     4     5     3    50 0x0804aa90    93 0x0804aaed     3     0     2     0    28 sym.__libc_csu_init
0x0804aaf0    2     1     0     1     3 0x0804aaf0     2 0x0804aaf2     0     0     0     0     0 sym.__libc_csu_fini
0x0804ab00   20     1     0     1    10 0x0804ab00    20 0x0804ab14     2     0     0     3    12 sym.__stack_chk_fail_local
0x0804ab14   20     1     0     1    12 0x0804ab14    20 0x0804ab28     1     0     0     0    12 sym._fini
```

Con el comando iz vemos las strings del UAMsom:

```
[0xf7fa00b0]> iz
[Strings]
Num Paddr      Vaddr      Len Size Section  Type  String
000 0x00002b30 0x0804ab30  41  42 (.rodata) ascii basic_string::_M_construct null not valid
001 0x00002b5c 0x0804ab5c  32  33 (.rodata) ascii E: Could not create output file.
002 0x00002baa 0x0804abaa  20  21 (.rodata) ascii basic_string::append
003 0x00002bbf 0x0804abbf   4   5 (.rodata) ascii .uam
004 0x00002bc4 0x0804abc4  29  30 (.rodata) ascii E: Could not open input file.
005 0x00002be2 0x0804abe2  22  23 (.rodata) ascii Welcome to UAMsomware\n
006 0x00002bf9 0x0804abf9  10  11 (.rodata) ascii ./flag.txt
007 0x00002c04 0x0804ac04   6   7 (.rodata) ascii Time:
000 0x000040e0 0x0804d0e0  46  47 (.data) ascii Encrypting your files... :P\nYou are a looser!\n
001 0x00004110 0x0804d110  16  17 (.data) ascii expand 32-byte k
```

Con las strings entendemos que el programa lo que hace es leer el archivo flag.txt, aplicarle la extensión ".uam" lickear el "Welcome to UAMsomware", sacar como output el flag.txt.uam y devolver el timestamp.

Comparamos las funciones que son llamadas a través de main con el afl utilizando el comando: agc@main

Nos fijamos en 3 funciones descritas en el afl: sym,generateKey, sym._xor y sym.encriptedFile

```
0x080494d0     3 82           sym.generateKey
0x08049530     4 41           sym._xor_unsignedchar__unsignedchar__int
0x08049560    64 5291 -> 5261 sym.encryptFile_std::__cxx11::basic_string_char_std::char_traits_char_
0x0804aa80     1 6            sym.std::ctype_char_::do_widen_char_const
```

Por qué en estas funciones y no en otras? Pues capta mi atención debido a que tenemos un archivo cifrado (encripted) por algo extraño que tras verlo con r2 vemos que es un XOR y sería interesante ver la función generateKey decompilada para ver como podemos conseguir nuestra amada flag.

Antes de ello vemos la función generateKey (offset 0x80494d0 que hemos extraído del afl) y usamos el comando pd@0x080494d0 viendo que esta llama al XOR (offset 0x08049530) y tras aplicarlo se cifra el archivo con la función sym.encryptedFile (offset 0x08049560)

```
[0x080492f0]> pd@0x080494d0
/ (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, dword [ebx + 0x1c0] ; 448
|           0x080494e9      8dbbe0010000   lea edi, dword [ebx + 0x1e0] ; 480
|           0x080494ef      e81cfbffff     call sym.imp.srand          ; void srand(int seed)
|           0x080494f4      58             pop eax
|           0x080494f5      8d8380dbffff   lea eax, dword [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, dword [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
            0x08049522      8db426000000.  lea esi, dword [esi]
            0x08049529      8dbc27000000.  lea edi, dword [edi]
/ (fcn) sym._xor_unsignedchar__unsignedchar__int 41
|   sym._xor_unsignedchar__unsignedchar__int (int arg_ch, int arg_10h, int arg_14h);
|           ; arg int arg_ch @ esp+0xc
|           ; arg int arg_10h @ esp+0x10
|           ; arg int arg_14h @ esp+0x14
```

```
[0x080492f0]> pd@0x08049530
/ (fcn) sym._xor_unsignedchar__unsignedchar__int 41
|   sym._xor_unsignedchar__unsignedchar__int (int arg_ch, int arg_10h, int arg_14h);
|           ; arg int arg_ch @ esp+0xc
|           ; arg int arg_10h @ esp+0x10
|           ; arg int arg_14h @ esp+0x14
|           0x08049530      56              push esi
|           0x08049531      53              push ebx
|           0x08049532      8b5c2414        mov ebx, dword [arg_14h]    ; [0x14:4]=-1 ; 20
|           0x08049536      8b54240c        mov edx, dword [arg_ch]     ; [0xc:4]=-1 ; 12
|           0x0804953a      8b742410        mov esi, dword [arg_10h]    ; [0x10:4]=-1 ; 16
|           0x0804953e      85db            test ebx, ebx
|       ,=< 0x08049540      7e14            jle 0x8049556
|       |   0x08049542      31c0            xor eax, eax
|       |   0x08049544      8d742600        lea esi, dword [esi]
|       |   ; CODE XREF from sym._xor_unsignedchar__unsignedchar__int (0x8049554)
|       .--> 0x08049548      0fb60c06        movzx ecx, byte [esi + eax]
|       :|   0x0804954c      300c02          xor byte [edx + eax], cl
|       :|   0x0804954f      83c001          add eax, 1
|       :|   0x08049552      39c3            cmp ebx, eax
|       `==< 0x08049554      75f2            jne 0x8049548
|       |   ; CODE XREF from sym._xor_unsignedchar__unsignedchar__int (0x8049540)
|       `-> 0x08049556      5b              pop ebx
|           0x08049557      5e              pop esi
|           0x08049558      c3              ret
\           0x08049559      8db426000000.   lea esi, dword [esi]
/ (fcn) sym.encryptFile_std::__cxx11::basic_string_char_std::char_traits_char__std::allocator_char 5261
|   sym.encryptFile_std::__cxx11::basic_string_char_std::char_traits_char__std::allocator_char (int arg_308h, in
|           ; var int local_3f4h @ ebp-0x3f4
|           ; var int local_3f8h @ ebp-0x3f8
|           ; arg int arg_308h @ ebp+0x308
|           ; var int local_2ach @ ebp-0x2ac
```

Decidimos poner unos breakpoints en radare para confirmar que efectivamente se ponen ejecutan estas funciones en este orden. Con el comando ood entramos en modo debug y para poner el breakpoint hacemos **db + el offset de la función** y dc para continuar con la ejecución cada vez que se pare en uno de los breakpoints indicados. Además de ello hacemos un dr para ver los registros.

```
[0x080492f0]> ood
Process with PID 2427 started...
File dbg:///root/Downloads/uam/UAMsom  reopened in read-write mode
= attach 2427 2427
2427
[0xf7f9c0b0]> db 0x080494d0
[0xf7f9c0b0]> db 0x08049530
[0xf7f9c0b0]> db 0x08049560
[0xf7f9c0b0]> dc
Welcome to UAMsomware

hit breakpoint at: 8049560
[0x08049560]> dc
hit breakpoint at: 80494d0
[0x080494d0]> dc
hit breakpoint at: 8049530
[0x08049530]> dc
Time: 1548277400
[0xf7f9a079]> dr
eax = 0xffffffda
ebx = 0x00000000
ecx = 0x00000000
edx = 0x00000000
esi = 0xf7dca200
edi = 0x00000000
esp = 0xffafe97c
ebp = 0xf7dcc000
eip = 0xf7f9a079
eflags = 0x00000282
oeax = 0x000000fc
```

Decompilamos la función generateKey y la del XOR para ver si encontramos más información de la clave que genera el cifrado.

```
int generateKey(void)
{
  _BYTE *v0; // esi@1
  int result; // eax@2

  v0 = &fileKey;
  srand(0x37u);
  malloc((size_t)"ïL$\b¦");
  do
  {
    result = rand();
    *v0++ ^= result;
  }
  while ( (_UNKNOWN *)v0 != &std::__ioinit );
  return result;
}
```

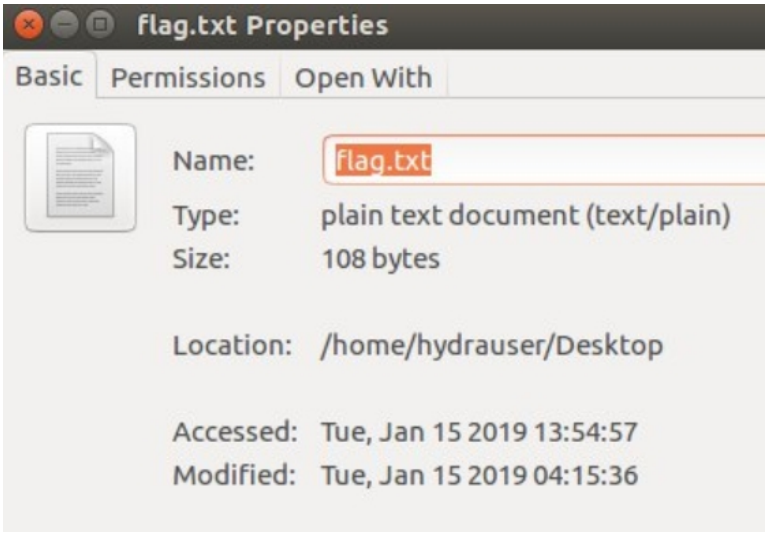
```
void __cdecl _xor(unsigned __int8 *a1, unsigned __int8 *a2, int a3)
{
  int v3; // eax@2

  if ( a3 > 0 )
  {
    v3 = 0;
    do
    {
      a1[v3] ^= a2[v3];
      ++v3;
    }
    while ( a3 != v3 );
  }
}
```

Decidimos probar con clave del tamaño del malloc por si un caso en cyberchef. De perdidos al río!!

Que disparate… pues va a ser que no!!! Hago fuerza bruta al XOR y tampoco nos da un resultado coherente con lo que llego a un punto de estanque. Con todas las vueltas que he dado. Por dónde debería seguir? Recapitulando todo lo aprendido. Tenemos un txt cifrado con XOR no sabemos la clave y el programa cada vez que lo ejecuto devuelve un timestamp distinto…
Pero que ocurre si ejecutamos un XOR 2 veces??? que nos devuelve la información original. Le vuelve a dar la vuelta. La idea que decido seguir es volver a ejecutar el XOR en la misma hora y fecha que se ejecutó ya que no he logrado entender como descifrar la clave.

Tras unas pocas vueltas recuerdo que en el OVF original estaba el programa con el output cifrado y voy a buscar en las propiedades la fecha de creación.



En este punto con la necesidad de utilizar esta fecha concreta sin cambiar la de la Vm descubro la tool faketime. Me debato con ella y tras algunos problemillas con la variable de entorno LD_PRELOAD pienso en otra forma más sencilla aún. Tenemos el UAMsom y el flag.txt en la OVF. Solo necesitamos cambiar la fecha del ubuntu de esta VM que nos dan y con ejecutar el programa una simple vez ya tenemOs el flag.txt descifrado… Pero dios cómo le he podido dar tantas vueltas para llegar a una solución tan sencilla q hubiese podido ver desde el minuto 1 sino hubiese estado con tanta obsesión de reversing al encontrar el programa!

Me percato de que han modificado el Accessed time pero el Modified time no se puede cambiar cosa que hace pensar que esta sea la fecha buena. Voilà aquí tenemos el descifrado.

```
+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
```

Vaya.. pues esto si que no me lo esperaba. No es un base64, ni un rot, ni nada que a simple vista reconozca. Un porrón de numeros y ninguna pisa por dónde continuar… Tras dormir unas horitas me da por volver a echarle un ojo a los números… hay un + delante de ellos… entre ellos está el +34 que es el prefijo de España… no serán los prefijos de varios países mmm Y si con la primera letra de cada país formase una frase? Puede que no sea la flag pero igual me indique el camino. Googleamos para buscar una lista de los prefijos telefónicos y al escribir la primera letra y juntarlos damos con la frase:
EN FEBRERO ATACAREMOS LA BASE DE HAITI

Bingo!! tenemos las comunicaciones que buscábamos. La flag en mayúsculas y sin espacios en formato UAM{md5} queda:

UAM{0f34e05951b864bd0621680af1f94acc}

Autoría: Arsenics