Python Penetration Testing Essentials

This book is a practical guide that shows you the advantages of using Python for pentesting with the help of detailed code examples.

We start by exploring the basics of networking with Python and then proceed to network hacking. Next, we delve into hacking the application layer where we start with gathering information from a website. We then move on to concepts related to website hacking such as parameter tampering, DDoS, XSS, and SQL injection.

By reading this book, you will learn different techniques and methodologies that will familiarize you with the art of pentesting as well as creating automated programs to find the admin console, SQL injection, and XSS attack.

Who this book is written for

If you are a Python programmer or a security researcher who has basic knowledge of Python programming and want to learn about penetration testing with the help of Python, this book is ideal for you. Even if you are new to the field of ethical hacking, this book can help you find the vulnerabilities in your system so that you are ready to tackle any kind of attack or intrusion.

What you will learn from this book

- Monitor the Ethernet IP and TCP traffic over the network
- Explore wireless traffic with the help of various programs
- Perform wireless attacks with Python programs
- Check live systems and distinguish between the operating system and services of a remote machine
- Broaden your concepts in pentesting right from the basics of the client/server architecture in Python
- Gather passive information from a website using automated scripts
- Perform XSS, SQL injection, and parameter tampering attacks

Mohit

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 "Python with Penetration Testing and Networking"
- A synopsis of the book’s content
- More information on Python Penetration Testing Essentials

About the Author

Mohit (also known as Mohit Raj) is an application developer and Python programmer, with a keen interest in the field of information security. He has done his bachelor of technology in computer science from Kurukshetra University, Kurukshetra, and master of engineering (2012) in computer science from Thapar University, Patiala. He has written a thesis as well as a research paper on session hijacking, named COMPARATIVE ANALYSIS OF SESSION HIJACKING ON DIFFERENT OPERATING SYSTEMS, under the guidance of Dr Maninder Singh. He has also done the CCNA and Certified Ethical Hacking course from EC-Council (CEH) and has procured a CEH certification. He has published his article, How to disable or change web-server signature, in the eForensics magazine in December 2013. He has published another article on wireless hacking, named Beware: Its Easy to Launch a Wireless Deauthentication Attack! in Open Source For You in July 2014. He is also a certified Certified Security Analyst (ECSA). He has been working in IBM India for more than 2 years. He is also a freelance professional trainer for CEH and Python in CODEC Networks. Apart from this, he is familiar with Red Hat and CentOS Linux to a great extent, and also has a lot of practical experience of Red Hat. He can be contacted at mohitraj.cs@gmail.com.
First of all, I am grateful to the Almighty for helping me to complete this book. I would like to thank my mother for her love and encouraging support, and my father for raising me in a house with desktops and laptops. A big thanks to my teacher, thesis guide, and hacking trainer, Dr. Maninder Singh, for his immense help. I would like to thank my friend, Bhaskar Das, for providing me with hardware support. I would also like to thank everyone who has contributed to the publication of this book, including the publisher, especially the technical reviewers and also the editors Merwyn D'souza and Sonali Vernekar, for making me laugh at my own mistakes. Last but not least, I'm grateful to my i7 laptop, without which it would not have been possible to write this book.
Python Penetration Testing Essentials

This book is a practical guide that shows you the advantages of using Python for pentesting, with the help of detailed code examples. This book starts by exploring the basics of networking with Python and then proceeds to network and wireless pentesting, including information gathering and attacking. Later on, we delve into hacking the application layer, where we start by gathering information from a website, and then eventually move on to concepts related to website hacking, such as parameter tampering, DDOS, XSS, and SQL injection.

What This Book Covers

Chapter 1, Python with Penetration Testing and Networking, aims to complete the prerequisites of the following chapters. This chapter also discusses the socket and its methods. The server socket's method defines how to create a simple server.

Chapter 2, Scanning Pentesting, covers how network scanning is done to gather information on a network, host, and the service that are running on the hosts.

Chapter 3, Sniffing and Penetration Testing, teaches how to perform active sniffing, how to create a layer 4 sniffer, and how to perform layer 3 and layer 4 attacks.

Chapter 4, Wireless Pentesting, teaches wireless frames and how to obtain information such as SSID, BSSID, and the channel number from a wireless frame using a Python script. In this type of attack, you will learn how to perform pentesting attacks on the AP.

Chapter 5, Foot Printing of a Web Server and a Web Application, teaches the importance of a web server signature, and why knowing the server signature is the first step in hacking.

Chapter 6, Client-side and DDoS Attacks, teaches client-side validation as well as how to bypass client-side validation. This chapter covers the implantation of four types of DDoS attacks.

Chapter 7, Pentesting of SQLI and XSS, covers two major web attacks, SQL injection and XSS. In SQL injection, you will learn how to find the admin login page using a Python script.
Penetration 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. In this chapter, we will cover the following topics:

- The scope of pentesting
- The need for pentesting
- Components to be tested
- Qualities of a good pentester
- Approaches of pentesting
- Understanding the tests and tools you'll need
- Network sockets
- Server socket methods
- Client socket methods
• General socket methods
• Practical examples of sockets
• Socket exceptions
• Useful socket methods

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 his 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

As you must have seen, this book is divided into seven chapters. 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.

Network sockets

A network socket address contains an IP address and port number. In a very simple way, a socket is a way to talk to other computers. By means of a socket, a process can communicate with another process over the network.
In order to create a socket, use the `socket.socket()` function that is available in the socket module. The general syntax of a socket function is as follows:

```python
s = socket.socket(socket_family, socket_type, protocol=0)
```

Here is the description of the parameters:

- **socket_family**: `socket.AF_INET`, `PF_PACKET`
  - `AF_INET` is the address family for IPv4.
  - `PF_PACKET` operates at the device driver layer. The pcap library for Linux uses `PF_PACKET`. You will see more details on `PF_PACKET` in *Chapter 3, Sniffing and Penetration Testing*. These arguments represent the address families and the protocol of the transport layer:

- **Socket_type**: `socket.SOCK_DGRAM`, `socket.SOCK_RAW`, `socket.SOCK_STREAM`
  - The `socket.SOCK_DGRAM` argument depicts that UDP is unreliable and connectionless, and `socket.SOCK_STREAM` depicts that TCP is reliable and is a two-way, connection-based service. We will discuss `socket.SOCK_RAW` in *Chapter 3, Sniffing and Penetration Testing*.

- **protocol**: Generally, we leave this argument; it takes 0 if not specified. We will see the use of this argument in *Chapter 3, Sniffing and Penetration Testing*.

### Server socket methods

In a client-server architecture, there is one centralized server that provides service, and many clients request and receive service from the centralized server. Here are some methods you need to know:

- **`socket.bind(address)`**: This method is used to connect the address (IP address, port number) to the socket. The socket must be open before connecting to the address.
- **`socket.listen(q)`**: This method starts the TCP listener. The `q` argument defines the maximum number of lined-up connections.
- **`socket.accept()`**: The use of this method is to accept the connection from the client. Before using this method, the `socket.bind(address)` and `socket.listen(q)` methods must be used. The `socket.accept()` method returns two values: `client_socket` and `address`, where `client_socket` is a new socket object used to send and receive data over the connection, and `address` is the address of the client. You will see examples later.
Client socket methods
The only method dedicated to the client is the following:

- `socket.connect(address)`: This method connects the client to the server. The `address` argument is the address of the server.

General socket methods
The general socket methods are as follows:

- `socket.recv(bufsize)`: This method receives a TCP message from the socket. The `bufsize` argument defines the maximum data it can receive at any one time.
- `socket.recvfrom(bufsize)`: This method receives data from the socket. The method returns a pair of values: the first value gives the received data, and the second value gives the address of the socket sending the data.
- `socket.recv_into(buffer)`: This method receives data less than or equal to `buffer`. The `buffer` parameter is created by the `bytearray()` method. We will discuss it in an example later.
- `socket.recvfrom_into(buffer)`: This method obtains data from the socket and writes it into the buffer. The return value is a pair (nbytes, address), where nbytes is the number of bytes received, and the address is the address of the socket sending the data.

Be careful while using the `socket.recv from_into(buffer)` method in older versions of Python. Buffer overflow vulnerability has been found in this method. The name of this vulnerability is CVE-2014-1912, and its vulnerability was published on February 27, 2014. Buffer overflow in the `socket.recvfrom_into` function in modules/socketmodule.c in Python 2.5 before 2.7.7, 3.x before 3.3.4, and 3.4.x before 3.4c1 allows remote attackers to execute arbitrary code via a crafted string.

- `socket.send(bytes)`: This method is used to send data to the socket. Before sending the data, ensure that the socket is connected to a remote machine. It returns the number of bytes sent.
Chapter 1

- **socket.sendto(data, address):** This method is used to send data to the socket. Generally, we use this method in UDP. UDP is a connectionless protocol; therefore, the socket should not be connected to a remote machine, and the address argument specifies the address of the remote machine. The return value gives the number of bytes sent.

- **socket.sendall(data):** As the name implies, this method sends all data to the socket. Before sending the data, ensure that the socket is connected to a remote machine. This method ceaselessly transfers data until an error is seen. If an error is seen, an exception would rise, and `socket.close()` would close the socket.

Now it is time for the practical; no more mundane theory.

**Moving on to the practical**

First, we will make a server-side program that offers a connection to the client and sends a message to the client. Run `server1.py`:

```python
import socket
host = "192.168.0.1" #Server address
port = 12345  #Port of Server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((host,port)) #bind server
s.listen(2)
conn, addr = s.accept()
print addr, "Now Connected"
conn.send("Thank you for connecting")
conn.close()
```

The preceding code is very simple; it is minimal code on the server side.

First, import the socket module and define the host and port number: `192.168.0.1` is the server's IP address. `socket.AF_INET` defines the IPv4 protocol's family. `Socket.SOCK_STREAM` defines the TCP connection. The `s.bind((host,port))` statement takes only one argument. It binds the socket to the host and port number. The `s.listen(2)` statement listens to the connection and waits for the client. The `conn, addr = s.accept()` statement returns two values: `conn` and `addr`. The `conn` socket is the client socket, as we discussed earlier. The `conn.send()` function sends the message to the client. Finally, `conn.close()` closes the socket. From the following examples and screenshot, you will understand `conn` better.
This is the output of the `server1.py` program:

```
G:\Python\Networking>python server1.py
G:\Python\Networking>python server1.py
```

Now, the server is in the listening mode and is waiting for the client:

Let's see the client-side code. Run `client1.py`:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = "192.168.0.1"  # server address
port =12345  #server port
s.connect((host,port))
print s.recv(1024)
s.send("Hello Server")
s.close()
```

In the preceding code, two methods are new: `s.connect((host, port))`, which connects the client to the server, and `s.recv(1024)`, which receives the strings sent by the server.

The output of `client.py` and the response of the server is shown in the following screenshot:

![Screenshot of Python output](image)

The preceding screenshot of the output shows that the server accepted the connection from `192.168.0.11`. Don't get confused by seeing the port `1789`; it is the random port of the client. When the server sends a message to the client, it uses the `conn` socket, as mentioned earlier, and this `conn` socket contains the client IP address and port number.
The following diagram shows how the client accepts a connection from the server. The server is in the listening mode, and the client connects to the server. When you run the server and client program again, the random port gets changed. For the client, the server port **12345** is the destination port, and for the server, the client random port **1789** is the destination port.

![Diagram of client and server connection](image)

You can extend the functionality of the server using the `while` loop, as shown in the following program. Run the `server2.py` program:

```python
import socket
host = "192.168.0.1"
port = 12345
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((host, port))
s.listen(2)
while True:
    conn, addr = s.accept()
    print addr, "Now Connected"
    conn.send("Thank you for connecting")
    conn.close()
```

The preceding code is the same as the previous one, just the infinite `while` loop is added.

Run the `server2.py` program, and from the client, run `client1.py`. 
The output of `server2.py` is shown here:

```bash
G:\Python\Networking>python server2.py
('192.168.0.11', 1791) Now Connected
('192.168.0.11', 1792) Now Connected
('192.168.0.11', 1793) Now Connected

C:\> Command Prompt
C:\net1>client1.py
Thank you for connecting
C:\net1>client1.py
Thank you for connecting
C:\net1>client1.py
Thank you for connecting
C:\net1>client1.py
Thank you for connecting
C:\net1>
```

One server can give service to many clients. The `while` loop keeps the server program alive and does not allow the code to end. You can set a connection limit to the `while` loop; for example, set `while i>10` and increment `i` with each connection.

Before proceeding to the next example, the concept of `bytearray` should be understood. The `bytearray` array is a mutable sequence of unsigned integers in the range of 0 to 255. You can delete, insert, or replace arbitrary values or slices. The `bytearray` array’s objects can be created by calling the built-in `bytearray` array.

The general syntax of `bytearray` is as follows:

```python
bytearray([source[, encoding[, errors]]])
```

Let’s illustrate this with an example:

```python
>>> m = bytearray("Mohit Mohit")
>>> m[1]
111
>>> m[0]
77
>>> m[:5]= "Hello"
>>> m
bytearray(b'Hello Mohit')
>>> 
```
This is an example of the slicing of `bytearray`.

Now, let's look at the splitting operation on `bytearray()`:  

```python
>>> m = bytearray("Hello Mohit")
>>> m
bytearray(b'Hello Mohit')
>>> m.split()
[bytearray(b'Hello'), bytearray(b'Mohit')]
```

The following is the append operation on `bytearray()`:

```python
>>> m.append(33)
>>> m
bytearray(b'Hello Mohit!')
```

The next example is of `s.recv_into(buff)`. In this example, we will use `bytearray()` to create a buffer to store data.

First, run the server-side code. Run `server3.py`:

```python
import socket
host = "192.168.0.1"
port = 12345
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((host, port))
s.listen(1)
conn, addr = s.accept()
print "connected by", addr
conn.send("Thanks")
conn.close()
```

The preceding program is the same as the previous one. In this program, the server sends `Thanks`, six characters.

Let's run the client-side program. Run `client3.py`:

```python
import socket
host = "192.168.0.1"
port = 12345
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))
buf = bytearray("-" * 30) # buffer created
print "Number of Bytes ", s.recv_into(buf)
print buf
s.close
```
In the preceding program, a buf parameter is created using bytearray(). The s.recv_into(buf) statement gives us the number of bytes received. The buf parameter gives us the string received.

The output of client3.py and server3.py is shown in the following screenshot:

![Screenshot showing the output of client3.py and server3.py.]

Our client program successfully received 6 bytes of string, Thanks. Now, you must have got an idea of bytearray(). I hope you will remember it.

This time I will create a UDP socket.

Run udp1.py, and we will discuss the code line by line:

```python
import socket
host = "192.168.0.1"
port = 12346
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind((host, port))
data, addr = s.recvfrom(1024)
print "received from ", addr
print "obtained ", data
s.close()
```

socket.SOCK_DGRAM creates a UDP socket, and data, addr = s.recvfrom(1024) returns two things: first is the data and second is the address of the source.
Now, see the client-side preparations. Run `udp2.py`:

```python
import socket
host = "192.168.0.1"
port = 12346
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
print s.sendto("hello all",(host,port))
s.close()
```

Here, I used the UDP socket and the `s.sendto()` method, as you can see in the definition of `socket.sendto()`. You know very well that UDP is a connectionless protocol, so there is no need to establish a connection here.

The following screenshot shows the output of `udp1.py` (the UDP server) and `udp2.py` (the UDP client):

```
C:\Project Snake\First Chapter\programs>python udp1.py
received from ("192.168.0.1", 1814) obtained hello all
```

The server program successfully received data.

Let us assume that a server is running and there is no client start connection, and that the server will have been listening. So, to avoid this situation, use `socket.settimeout(value)`.

Generally, we give a value as an integer; if I give 5 as the value, it would mean wait for 5 seconds. If the operation doesn't complete within 5 seconds, then a timeout exception would be raised. You can also provide a non-negative float value.
For example, let's look at the following code:

```python
import socket
host = "192.168.0.1"
port = 12346
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind((host,port))
s.settimeout(5)
data, addr = s.recvfrom(1024)
print "received from ", addr
print "obtained ", data
s.close()
```

I added one line extra, that is, `s.settimeout(5)`. The program waits for 5 seconds; only after that it will give an error message. Run `udptime1.py`.

The output is shown in the following screenshot:

The program shows an error; however, it does not look good if it gives an error message. The program should handle the exceptions.

**Socket exceptions**

In order to handle exceptions, we'll use the try and except blocks. The next example will tell you how to handle the exceptions. Run `udptime2.py`:

```python
import socket
host = "192.168.0.1"
port = 12346
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
try:
```

s.bind((host, port))
s.settimeout(5)
data, addr = s.recvfrom(1024)
print "received from ", addr
print "obtained ", data
s.close()

except socket.timeout :
    print "Client not connected"
s.close()

The output is shown in the following screenshot:

C:\Windows\system32\cmd.exe
C:\Project Snake\Chapter 1\First Chapter\programs>python udptime2.py
Client not connected
C:\Project Snake\Chapter 1\First Chapter\programs>

In the try block, I put my code, and from the except block, a customized message is printed if any exception occurs.

Different types of exceptions are defined in Python's socket library for different errors. These exceptions are described here:

- exception socket.herror: This block catches the address-related error.
- exception socket.timeout: This block catches the exception when a timeout on a socket occurs, which has been enabled by settimeout(). In the previous example you can see that we used socket.timeout.
- exception socket.gaierror: This block catches any exception that is raised due to getaddrinfo() and getnameinfo().
- exception socket.error: This block catches any socket-related errors. If you are not sure about any exception, you could use this. In other words, you can say that it is a generic block and can catch any type of exception.

Downloading the example code
You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
Useful socket methods

So far, you have gained knowledge of socket and client-server architecture. At this level, you can make a small program of networks. However, the aim of this book is to test the network and gather information. Python offers very beautiful as well as useful methods to gather information. First, import socket and then use these methods:

- `socket.gethostbyname(hostname)`: This method converts a hostname to the IPv4 address format. The IPv4 address is returned in the form of a string. Here is an example:
  ```python
  >>> import socket
  >>> socket.gethostbyname('thapar.edu')
  '220.227.15.55'
  >>>
  >>> socket.gethostbyname('google.com')
  '173.194.126.64'
  >>>
  ```
  I know you are thinking about the `nslookup` command. Later, you will see more magic.

- `socket.gethostbyname_ex(name)`: This method converts a hostname to the IPv4 address pattern. However, the advantage over the previous method is that it gives all the IP addresses of the domain name. It returns a tuple (hostname, canonical name, and IP_addrlist) where the hostname is given by us, the canonical name is a (possibly empty) list of canonical hostnames of the server for the same address, and IP_addrlist is a list all the available IPs of the same hostname. Often, one domain name is hosted on many IP addresses to balance the load of the server. Unfortunately, this method does not work for IPv6. I hope you are well acquainted with tuple, list, and dictionary. Let's look at an example:
  ```python
  >>> socket.gethostbyname_ex('thapar.edu')
  ('thapar.edu', [], ['14.139.242.100', '220.227.15.55'])
  >>>
  >>> socket.gethostbyname_ex('google.com')
  ```
  ```python
  >>>
  ('google.com', [], ['173.194.36.64', '173.194.36.71',
  '173.194.36.73', '173.194.36.70', '173.194.36.78',
  '173.194.36.66', '173.194.36.65', '173.194.36.68',
  '173.194.36.69', '173.194.36.72', '173.194.36.67'])
  ```
It returns many IP addresses for a single domain name. It means one domain such as `thapar.edu` or `google.com` runs on multiple IPs.

- `socket.gethostname()`: This returns the hostname of the system where the Python interpreter is currently running:

  ```python
  >>> socket.gethostname()
  'eXtreme'
  ```

  To glean the current machine's IP address by socket module, you can use the following trick using `gethostbyname(socket.gethostname())`:

  ```python
  >>> socket.gethostbyname(socket.gethostname())
  '192.168.10.1'
  ```

  You know that our computer has many interfaces. If you want to know the IP address of all the interfaces, use the extended interface:

  ```python
  >>> socket.gethostbyname_ex(socket.gethostname())
  ('eXtreme', [], ['10.0.0.10', '192.168.10.1', '192.168.0.1'])
  ```

  It returns one tuple containing three elements: first is the machine name, second is a list of aliases for the hostname (empty in this case,) and third is the list of IP addresses of interfaces.

- `socket.getfqdn([name])`: This is used to find the fully qualified name, if it’s available. The fully qualified domain name consists of a host and domain name; for example, `beta` might be the hostname, and `example.com` might be the domain name. The **fully qualified domain name (FQDN)** becomes `beta.example.com`.

  ```python
  >>> socket.getfqdn('facebook.com')
  'edge-star-shv-12-frc3.facebook.com'
  ```

  In the preceding example, `edge-star-shv-12-frc3` is the hostname, and `facebook.com` is the domain name. In the following example, FQDN is not available for `thapar.edu`:

  ```python
  >>> socket.getfqdn('thapar.edu')
  'thapar.edu'
  ```

  If the name argument is blank, it returns the current machine name:

  ```python
  >>> socket.getfqdn()
  'eXtreme'
  ```
socket.gethostbyaddr(ip_address): This is like a "reverse" lookup for the name. It returns a tuple (hostname, canonical name, and IP_addrlist) where hostname is the hostname that responds to the given ip_address, the canonical name is a (possibly empty) list of canonical names of the same address, and IP_addrlist is a list of IP addresses for the same network interface on the same host:

```python
>>> socket.gethostbyaddr('173.194.36.71')
('del01s06-in-f7.1e100.net', [], ['173.194.36.71'])

>>> socket.gethostbyaddr('119.18.50.66')
```

Traceback (most recent call last):
File "<pyshell#9>", line 1, in <module>
socket.gethostbyaddr('119.18.50.66')
herror: [Errno 11004] host not found

It shows an error in the last query because reverse DNS lookup is not present.

socket.getservbyname(servicename[, protocol_name]): This converts any protocol name to the corresponding port number. The Protocol name is optional, either TCP or UDP. For example, the DNS service uses TCP as well as UDP connections. If the protocol name is not given, any protocol could match:

```python
>>> import socket
>>> socket.getservbyname('http')
80
>>> socket.getservbyname('smtp','tcp')
25

>>> socket.getservbyname('http')
```

socket.getservbyport(port[, protocol_name]): This converts an Internet port number to the corresponding service name. The protocol name is optional, either TCP or UDP:

```python
>>> socket.getservbyport(80)
'http'
>>> socket.getservbyport(23)
'telnet'
>>> socket.getservbyport(445)
'microsoft-ds'
```
socket.connect_ex(address): This method returns an error indicator. If successful, it returns 0; otherwise, it returns the errno variable. You can take advantage of this function to scan the ports. Run the `connet_ex.py` program:

```python
import socket
rmip = '127.0.0.1'
portlist = [22, 23, 80, 912, 135, 445, 20]

for port in portlist:
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    result = sock.connect_ex((rmip, port))
    print(port, ':', result)
    sock.close()
```

The output is shown in the following screenshot:

![Screenshot showing the output of the program](image)

The preceding program output shows that ports 80, 912, 135 and 445 are open. This is a rudimentary port scanner. The program is using the IP address 127.0.0.1; this is a loop back address, so it is impossible to have any connectivity issues. However, when you have issues, perform this on another device with a large port list. This time you will have to use `socket.settimeout(value)`:

```python
socket.getaddrinfo(host, port[, family[, socktype[, proto[, flags]]]])
```

This socket method converts the host and port arguments into a sequence of five tuples.

Let's take a look at the following example:

```python
>>> import socket
>>> socket.getaddrinfo('www.thapar.edu', 'http')
[(2, 1, 0, '', ('220.227.15.47', 80)), (2, 1, 0, '', ('14.139.242.100', 80))]
>>>```
output 2 represents the family, 1 represents the socket type, 0 represents the protocol, "," represents canonical name, and 
(('220.227.15.47', 80) represents the socket address. However, this number is difficult to comprehend. Open the directory of the socket.

Use the following code to find the result in a readable form:

```python
import socket

def get_protnumber(prefix):
    return dict((getattr(socket, a), a)
                for a in dir(socket)
                if a.startswith(prefix))

proto_fam = get_protnumber('AF_')
types = get_protnumber('SOCK_')
protocols = get_protnumber('IPPROTO_')

for res in socket.getaddrinfo('www.thapar.edu', 'http'):
    family, socktype, proto, canonname, sockaddr = res
    print 'Family     :', proto_fam[family]
    print 'Type       :', types[socktype]
    print 'Protocol   :', protocols[proto]
    print 'Canonical name:', canonname
    print 'Socket address:', sockaddr
```

The output of the code is shown in the following screenshot:
The upper part makes a dictionary using the AF_, SOCK_, and IPPROTO_ prefixes that map the protocol number to their names. This dictionary is formed by the list comprehension technique.

The upper part of the code might sometimes be confusing, but we can execute the code separately as follows:

```python
>>> dict((getattr(socket,n),n) for n in dir(socket) if n.startswith('AF_'))
```

Now, this is easy to understand. This code is usually used to get the protocol number:

```python
for res in socket.getaddrinfo('www.thapar.edu', 'http'):
```

The preceding line of code returns the five values, as discussed in the definition. These values are then matched with their corresponding dictionary.

**Summary**

Now, you have got an idea of networking in Python. The aim of this chapter is to complete the prerequisites of the upcoming chapters. From the start, you have learned the need for pentesting. Pentesting is conducted to identify threats and vulnerability in the organization. What should be tested? This is specified in the agreement; don't try to test anything that is not mentioned in the agreement.

Agreement is your jail-free card. A pentester should have knowledge of the latest technology. You should have some knowledge of Python before you start reading this book. In order to run Python scripts, you should have a lab setup, a network of computers to test a live system, and dummy websites running on the Apache server.

This chapter discussed the socket and its methods. The server socket method defines how to make a simple server. The server binds its own address and port to listen to the connections. A client that knows the server address and port number connects to the server to get service. Some socket methods such as `socket.recv(bufsize)`, `socket.recvfrom(bufsize)`, `socket.recv_into(buffer)`, `socket.send(bytes)`, and so on are useful for the server as well as the client. You learned how to handle different types of exceptions. In the Useful socket methods section, you got an idea of how to get the IP and hostname of a machine, how to glean the IP address from the domain name, and vice versa.

In the next chapter, you will see scanning pentesting, which includes IP address scanning to detect the live hosts. To carry out IP scanning, ping sweep and TCP scanning are used. You will learn how to detect services running on a remote host using port scanner.
Where to buy this book

You can buy Python Penetration Testing Essentials from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.