Experiment No .7 Packet Crafting

Aim: Packet Crafting using Scapy

Objectives:

1. Gain hands-on experience with Scapy, a Python-based packet manipulation tool.
2. Understand the functionality and significance of protocols at the Application and Transport Layers.
3. Analyze and dissect packets to comprehend the structure and contents of different protocols.
4. Investigate the interaction between Application and Transport Layer protocols.
5. Develop skills in crafting custom packets for specific networking scenarios.

Theory:
The Scapy hosts the website with all the documentation. Kindly visit(https://scapy.net/) for reads.

Problem Statement:
Kindly craft the following packets. Take a screenshot of results. Also take screenshots of the crafted packet you send.

Ping (ICMP Echo Request):

- Craft an ICMP Echo Request packet using Scapy.
- Send the packet to a target IP address.
- Expect an ICMP Echo Reply packet in response from the target.

UDP Datagram:

- Craft a UDP packet with custom payload using Scapy.
- Send the UDP packet to a target listening on a specific UDP port.
- Expect a response from the target if the port is open and reachable.

DNS Query:

- Craft a DNS query packet using Scapy to query a DNS server for a specific domain.
- Send the DNS query packet to the DNS server.
- Expect a DNS response containing the IP address associated with the queried domain.

HTTP GET Request:

- Craft an HTTP GET request packet using Scapy to retrieve a specific web page from a web server.
- Send the HTTP GET request to the web server.
- Expect an HTTP response containing the requested web page content.

Experiment No .7 Packet Crafting

Traceroute:

- Craft UDP packets with increasing TTL (Time-to-Live) values using Scapy.
- Send these packets towards a destination IP address.
- Observe the ICMP Time Exceeded messages returned by intermediate routers to map the network path to the destination.

Sending Packets

Creating and sending a packet:

```
>>> packet =
IP(dst="4.5.6.7",src="1.2.3.4")/
TCP(dport=80, flags="S")
```

Send a packet, or list of packets without custom ether layer:

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

Other send functions:

sr() sends and receives without a custom ether() layer
sendp() sends with a custom ether() layer
srp() sends and receives at with a custom ether() layer
sr1() sends packets without custom ether() layer and waits for first answer
sr1p() sends packets with custom ether() layer and waits for first answer

Send function options:

filter = <Berkley Packet Filter>
retry = <retry count for unanswered packets>
timeout = <number of seconds to wait before giving up>
iface = <interface to send and receive>

```
>>> packets = sr(packet, retry=5,
timeout=1.5, iface="eth0", filter="host
1.2.3.4 and port 80")
```

Sniffing and pcaps

To sniff using Berkley Packet Filters:

```
>>> packets = sniff(filter="host
1.1.1.1")
```

Sniffing using counts:

```
>>> packets = sniff(count=100)
```

Reading packets from a pcap:

```
>>> packets = rdpcap("filename.pcap")
```

Writing packets to a pcap:

```
>>> wrpcap("filename.pcap", packets)
```

Receiving and Analyzing Packets

Received packets can be stored in a variable when using a send/receive function such as sr(), srp(), sr1() sr1p():

```
>>> packet = IP(dst="10.10.10.20")/
TCP(dport=0,1024)
>>> unans, ans = sr(packet)
Received 1086 packets, got 1024 answers,
remaining 0 packets
```

“ans” will store the answered packets:

```
>>> ans
<Results: TCP:1024 UDP:0 ICMP:0 Other:0>
```

To see a summary of the responses:

```
>>> ans.summary()
IP / TCP 10.1.1.15:ftp_data >
10.10.10.20:netbios_ssn S ==> IP / TCP
10.10.10.20:netbios_ssn > 10.1.1.15:ftp_data
SA / Padding
```

Note: this is the output from port 139 (netbios_ssn). Notice how this port was open and responded with a SYN-ACK.

To view a specific answer as a stream in array form:

```
>>> ans[15]
```

To view the first packet in the stream:

```
>>> ans[15][0]
```

(this will be packet the Scapy sent)

```
<IP  frag=0 proto=tcp dst=10.10.10.20 |<TCP
dport=netstat flags=S |>>
```

To view the response from the distant end:

```
>>> ans[15][1]
<IP  version=4L ihl=5L tos=0x0 len=40 id=16355
flags=DF frag=0L ttl=128 proto=tcp
chksum=0x368c src=10.10.10.20 dst=10.1.1.15
options=[] |<TCP  sport=netstat dport=ftp_data
seq=0 ack=1 dataofs=5L reserved=0L flags=RA
window=0 chksum=0x2b4c urgptr=0 |<Padding
load='\x00\x00\x00\x00\x00\x00' |>>>
```

To view the TCP flags in the response packet:

```
>>> ans[15][1].sprintf("%TCP.flags%")
'RA'
```

SANS INSTITUTE

Scapy Cheat Sheet

POCKET REFERENCE GUIDE
Ver 0.2

http://www.sans.org

Purpose

The purpose of this cheat sheet is to describe some common options and techniques for using Scapy.

Scapy Overview

Scapy Background

Scapy is a Python module created by Philippe Biondi that allows extensive packet manipulation. Scapy allows packet forgery, sniffing, pcap reading/writing, and real-time interaction with network targets.

Scapy can be used interactively from a Python prompt or built into scripts and programs.

Launching Scapy

Once Scapy is installed it can be launched interactively by typing "sudo scapy" or from the command prompt.

Additionally Scapy can be imported either interactively or in a script with:

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

Note: Scapy requires root privileges to sniff or send packets!

Scapy Basics

To list supported layers:

```
>>> ls()
```

Some key layers are:

```
arp, ip, ipv6, tcp, udp, icmp
```

To view layer fields use ls(layer):

```
>>> ls(IPv6)

>>> ls(TCP)
```

To list available commands:

```
>>> lsc()
```

Some key commands for interacting with packets:

```
rdpcap, send, sr, sniff, wrpcap
```

Getting help with commands use help(command):

```
>>> help(rdpcap)
```

Basic Packet Crafting / Viewing

Scapy works with layers. Layers are individual functions linked together with the "/" character to construct packets. To build a basic TCP/IP packet with "data" as the payload:

```
>>> packet = IP(dst="1.2.3.4")/
TCP(dport=22)/"data"
```

Note: Scapy allows the user to craft all the way down to the ether() layer, but will use default values if layers are omitted. To correctly pass traffic layers should be ordered lowest to highest from left to right e.g. (ether -> IP -> TCP).

To get a packet summary:

```
>>> packet.summary()
```

To get more packet details:

```
>>> packet.show()
```

Ethernet Layer Fields / Default Values

```
>>> ls(Ether)
Field       Type            Default Value
dst      : DestMACField         = (None)
src      : SourceMACField       = (None)
type     : XShortEnumField      = (0)
```

IPv4 Layer Fields / Default Values

```
>>> ls(IP)
Field       Type            Default Value
version : BitField            = (4)
ihl     : BitField            = (None)
tos     : XByteField          = (0)
len     : ShortField          = (None)
id      : ShortField          = (1)
flags   : FlagsField          = (0)
frag    : BitField            = (0)
ttl     : ByteField           = (64)
proto   : ByteEnumField       = (0)
chksum  : XShortField         = (None)
src     : Emph                = (None)
dst     : Emph            = ('127.0.0.1')
options : PacketListField     = ([])
```

TCP Layer Fields / Default Values

```
>>> ls(TCP)
Field          Type            Default Value
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     = ({})
```

Altering Packets

Packet layer fields are Python variables and can be modified.
Example packet:

```
>>> packet = IP(dst="10.10.10.50")/
TCP(sport=80)
```

Viewing a field's value like the source port:

```
>>> packet.sport
80
```

Setting the source port:

```
>>> packet.sport = 443
>>> packet.sport
443
```

Setting port ranges:

```
>>> packet[TCP].dport = (1,1024)
```

Setting a list of ports:

```
>>> packet[TCP].dport = [22, 80, 445]
```

Setting the TCP flags (control bits):

```
>>> packet[TCP].flags="SA"
>>> packet[TCP].flags
18  (decimal value of CEUAPRSF bits)
>>> packet.sprintf("%TCP.flags%")
'SA'
```

Note! For ambiguous fields, like "flags", you must specify the target layer (TCP).

Setting destination IP address(es):

```
>>> packet[IP].dst = "1.2.3.4"
>>> packet[IP].dst = "sans.org"
```

Using CIDR:

```
>>> packet[IP].dst = "1.2.3.4/16"
```

Multiple Destinations:

```
>>> packet[IP].dst = ["1.2.3.4",
"2.3.4.5", "5.6.7.8"]
```