Python Scripting with Scapy

BLOSSOM
Manchester Metropolitan University
(Funded by Higher Education Academy)
l.han@mmu.ac.uk

1. Learning Objectives

This lab aims to learn how we use Scapy and python to programme the network monitor tools (manipulating, sending, receiving and sniffing packets.

2. Preparation

1) Under Linux environment

2) Some documents that you may need to refer to:

- 'Virtual-MachineGuide.pdf'
- ‘Linux-Guide.pdf’
- ‘BLOSSOM-UserGuide.pdf’
- ‘Packet Analysis & Introduction to Scapy.pdf’

3. Tasks

Setup & Installation:

- Start a single virtual machine as you have done with previous exercises (see Virtual Machine Guide):

```
# kvm -cdrom /var/tmp/BlossomFiles/blossom-0.98.iso -m 512 -net
nic,macaddr=52:54:00:12:34:57 -net vde -name node-one
```

Task 1 Basic Python Scripting with Scapy

1.1 In previous labs we have learnt how to use many different functions of Scapy, but now we will learn how Scapy can be as a library within Python which will allow us to write scripts or programs to perform tasks such as sending and receiving packets or sniffing packets.

First of all, we will open up gedit or your preferred text editor and write a basic Python script to sniff for packets:

NOTE: The text following the command to open gedit should be entered within the text editor. Also, '>' signifies the start of a line and should not be included in the script writing.

```
# gedit scapysniff.py
```

```
#! /usr/bin/env python

from scapy.all import *
a=sniff(count=10)
a.nsummary()
```

Save the script, change the mode of the file to be an executable and then execute it:

```
# chmod +x scapysniff.py
# ./scapysniff.py
```

This will sniff for 10 packets and as soon as 10 packets have sniffed, it will print a summary of the 10 packets that were discovered.

1.2 Next, we will look at a basic script that allows for the sending of packets:

```
# gedit scapysend.py
```

```
#! /usr/bin/env python

from scapy.all import *

send(IP(dst="1.2.3.4")/ICMP())
sendp(Ether()/IP(dst="1.2.3.4",ttl=(1,4)),
iface="eth1")
```

The two main lines of code feature different sending functions. Send() is used to send packets at the 3rd protocol layer, whereas Sendp() is used to send packets at the 2nd protocol layer. The difference is very important as some packets, such as ICMP are specific to certain layers, and it is up to us to know which packets can be used at which layer.

1.3 Scapy also has an array of commands for sending and receiving packets at the same time, which can be utilised in a python script as follows:

```
# gedit scapysendrec.py
```

```
#! /usr/bin/env python

 from scapy.all import *

 ans,unans=sr(IP(dst="192.168.86.130",ttl=5)/ICMP())
 ans.nsummary()
 unans.nsummary()
 p=sr1(IP(dst="192.168.86.130")/ICMP()/"XXXXXX")
 p.show()
```

The sr() function is for sending packets and receiving answers, which returns a couple of packets with answers, and also the unanswered packets which can be displayed as shown above. The function sr1() is a variant that only returns on packet that answered the packet that was sent.

sr() and sr1() are for layer 3 packets only. If you wish to send and receive layer 2 packets, you must use srp() or srp1().

Create a python script that sends and receives layer 2 packets, and then displays the information pertaining to the packets sent and received.

Task 2 Advanced Python Scripting with Scapy

2.1 Now that we understand the basics of sniffing packets, sending packets and receiving packets within python scripts, we can now learn some more advanced scripting.

```
# gedit scapysr.py
```

```
#! /usr/bin/env python

import sys
from scapy.all import sr1,IP,ICMP

p=sr1(IP(dst=sys.argv[1])/ICMP())
if p:
    p.show()
```

NOTE: Remember that indentation is extremely important when writing python scripts.

The previous script starts to introduce system arguments as an input. The sys.argv[1] as the destination address states that after executing the script, the first argument to follow the execution of the script will be used for the destination address, for example:

```
# ./scapysr.py 192.168.86.130
```

Using this, we now don't have to edit the source file every time we want to use a different IP address.

2.2 Scapy can also make use of methods so that we can make entire programs dedicated to certain functions, such as the live sniffing of packets:

```
# gedit scapylivearp.py
```

```
#! /usr/bin/env python

from scapy.all import *

def arp_monitor_callback(pkt):
        if ARP in pkt and pkt[ARP].op in (1,2): #who-has or is-at
                return pkt.sprintf("%ARP.hwsrc% %ARP.psrc%")

sniff(prn=arp_monitor_callback, filter="arp", store=0)
```

This will create a live packet sniffer that will return any ARP requests that are seen on all interfaces. The entire method basically states that if a packet is both an ARP packet, and the operation of that packet is either who-has or is-at, then it will return a printed line stating the source MAC address and source IP address of that ARP packet.

The method is applied to the sniff command using the prn function. Another important thing to notice is that 'store=0' is applied to the sniff command as well, and this is so that scapy avoids storing all of the packets within its memory.

2.3 There are countless other useful tools that we could create using Scapy as a library within python, such as the following example:

```
# gedit arping2tex.py
```

```
#! /usr/bin/env python
 import sys
 from scapy.all import srp,Ether,ARP,conf

 if len(sys.argv) != 2:
        print "Usage: arping2tex <net>\n eg: arping2text 192.168.1.0/24"
        sys.exit(1)

 conf.verb=0
 ans,unans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst=sys.argv[1]),
timeout=2)

 print r"\begin{tabular}{|l|l|}"
 print r"\hline"
 print r"MAC & IP\\"
 print r"\hline"
 for snd,rcv in ans:
        print rcv.sprintf(r"%Ether.src% & %ARP.psrc%\\")
 print r"\hline"
 print r"\end{tabular}"
```

This script will perform an ARP ping and then report whatever it finds out in LaTeX formatting, so if we were writing a report using LaTeX on a network, we could just copy the result and paste it into our report.

Task 3 Extending Scapy with Add-ons

3.1 If we ever need to add some new protocols or functions to Scapy, they can be written directly into the Scapy source file.

```
# gedit test_interact.py
```

```
#! /usr/bin/env python

# Set log level to benefit from Scapy warnings
import logging
logging.getLogger("scapy").setLevel(1)

from scapy.all import *

class Test(Packet):
    name="Test packet"
    fields_desc = [ ShortField("test1", 1),
                    ShortField("test2", 2) ]

def make_test(x,y):
    return Ether()/IP()/Test(test1=x,test2=y)

if __name__ == "__main__":
    interact(mydict=globals(), mybanner="Test add-on")
```

After this has been saved and made executable, execute the script and we should be confronted with a Scapy shell displaying the banner "Test add-on". Use the following command to see the add-on we created in action:

```
# make_test(42,666)
```

This lab begins to show the potential the Scapy can have when used in combination with Python to perform even more complex tasks, such as crafting packets to perform network attacks, or performing extensive packet sniffing for specific protocols. The possibilities are endless, and the information contained within this document is only a fraction of what can be done using Scapy and Python together.