Introduction to Scapy

1 Introduction

The goal of this tutorial is to get you comfortable with using Scapy. Scapy is a network packet manipulation program that allows you to create, modify and send network packets. Scapy is written in Python and thus, can also be used as a library. It is an extremely popular and powerful tool for parsing and creating network traffic. This tutorial introduces some aspects of scapy required for the accompanying assignment. This is by no means comprehensive. There are more tutorials mentioned at the end of the document if you are interested in exploring scapy further.

2 Installation

It is recommend that scapy be installed within a virtual environment using pip. A virtual environment is simply a Python installation that is isolated from the rest of the system. Thus, you will not have any conflicts or surprises because you were using the system installation.

Note: You will notice that there is a package "python-scapy" available for installation from the Ubuntu APT repositories. However, the version available for download from the repositories is quite old(2.2.0) when compared to what is available on PyPI(2.3.3). Your assignment solution will be tested using latest stable scapy available from PyPI. Hence, it is recommend to use scapy from PyPI for this tutorial and assignment than any other source.

As a first step, let's install PIP. PIP is a package manager similar to APT: it is used for managing Python packages available from PyPI and elsewhere. It is packaged in most Linux distro repositories and can be installed directly:

```
$ sudo apt install python-pip
```

This will install PIP and all required dependencies. However, the PIP version available in most repositories will be quite old and thus, it's recommended to update to the latest version using PIP itself:

```
$ sudo pip2 install --upgrade pip
```

For more information on pip, visit the homepage.

Now, let's install virtualenvwrapper. It is a set of wrapper scripts around virtualenv, which is the tool that greatly simplifies creating and managing Python virtual environments. Install virtualenvwrapper using PIP:

```
$ sudo pip2 install virtualenvwrapper
```

and enable virtualenvwrapper by adding the following line to your shell's RC file:

```
source /usr/local/bin/virtualenvwrapper.sh
```

For instance, if you use bash, your RC file will likely be *$HOME/.bashrc*. If you use zsh, it will be *$HOME/.zshrc* and so on. After adding the above line, restart your terminal.

After installing and setting up virtualenvwrapper is complete, let's create a new virtual environment for installing Scapy:

```
$ mkvirtualenv scapy-venv
```

This will create a new Python virtual environment installation in the folder *$HOME/.virtualenvs* with the name *scapy-venv*. If you wish to choose a different name for your virtual environment other than *scapy-venv*, substitute the desired name in the command. Throughout this tutorial, we will the name *scapy-venv* for the virtual environment.

On creating the virtual environment using mkvirtualenv, the virtual environment also gets activated automatically. This will be clear if you observe your shell's prompt: it would include the name of the virtual environment currently active. Inside this virtual environment, let's install scapy:

```
$ pip install scapy
```

This install packages required by scapy in the isolated virtual environment we just created above.

If you wish to have any additional packages, you can also install them inside the virtual environment. For instance, to use *ipython* within the *scapy-venv* environment, you can simply install *ipython* using *pip* within the virtual environment.

You can activate and deactivate virtual environments using the "*workon <insert environment name here>*" and *deactivate* commands respectively. You can also list, copy and remove virtual environments using *lsvirtualenv*, *cpvirtualenv* and *rmvirtualenv* respectively.

Tip: After setting up the virtual environment, you can export list of all libraries installed in the virtual environment to recreate the virtual more quickly next time. Export all libraries currently installed in the environment using the *pip freeze* command. The output of this command can be easily saved to a file which can be copied around and used to recreate the virtualenv environment:

```
# Saving list of installed packages
$ pip freeze > scapy-venv-requirements.txt
# Restoring installed packages
$ pip install -r scapy-venv-requirements.txt
```

3 Some scapy basics

There are two ways to use scapy:

1. **Interactive mode**: Here you will use scapy as a standalone program and work from within it's REPL(which is nothing but Python underneath). You can start scapy in interactive mode by running the command "scapy" from within scapy-venv.
2. **Embedded mode**: Here you can use scapy as a library from Python programs(and thus from the Python REPL too). For this, you can import *scapy.all* to access all scapy capabilities.

These above mode names aren't standard ones - I invented them to give an idea of how scapy can be used.

Scapy is an excellent packet parser: give it some bytes and ask it to create a packet out of it and it will do so very quickly and accurately. Let's try this example from the UDP segment parsing example:

```
$ ipython
In [1]: import scapy.all as scapy
In [2]: udp_packet = scapy.UDP("115c270f000cee3c74657374".decode('hex'))
In [3]: udp_packet
Out[3]: <UDP  sport=4444 dport=9999 len=12 chksum=0xee3c |<Raw  load='test' |>>
```

Notice how quickly scapy parsed the packet to extract the headers and the data section also. Repeat the above for the following UDP packets:

1. 115c270f000d3cae3132333435
2. 044b4e200015452c4142434445464748494a4b4c4d
3. A UDP packet you capture from wireshark. Also verify if the values of the fields displayed by scapy match what is shown in wireshark.

Similarly, you can also quickly build a packet using scapy. Here's how to build a UDP packet from port 10000 to port 20000 with data "testtest":

```
In [4]: udp_packet = scapy.UDP(sport=10000, dport=20000) / scapy.Raw("testtest")
In [5]: udp_packet
Out[5]: <UDP  sport=10000 dport=20000 |<Raw  load='testtest' |>>
```

Notice how layers get stacked up easily: the scapy.Raw data part becomes the payload for the UDP packet. In a similar manner, a complete packet consisting of data for all layers can be constructed by simply stack up all the layers one after another.

Note: To print out the nice representation of packet in a python script, use the *repr* function. Eg: *repr(udp_packet)*.

The interesting thing here is that optional fields(eg: checksum) need not be specified: they will be computed when the packet is sent or written out to a file. Thus, you can simply specify the essential fields while letting scapy compute values for the others. To view all fields for a protocol, simply inspect value of fields_desc member:

```
In [6]: udp_packet.fields_desc
Out[6]: [<Field (UDP,UDPerror).sport>, <Field (UDP,UDPerror).dport>,
         <Field (UDP,UDPerror).len>, <Field (UDP,UDPerror).chksum>]
```

Here, a UDP packet has 4 possible fields:

1. sport: Source port
2. dport: Destination port
3. len: The length of the UDP packet
4. chksum: The checksum of the UDP packet.

Similarly, other protocols would have their own fields, which can be found by inspecting the fields_desc field:

```
In [7]: scapy.TCP.fields_desc
Out[7]: [<Field (TCP,TCPerror).sport>, <Field (TCP,TCPerror).dport>,
         <Field (TCP,TCPerror).seq>, <Field (TCP,TCPerror).ack>,
         <Field (TCP,TCPerror).dataofs>, <Field (TCP,TCPerror).reserved>,
         <Field (TCP,TCPerror).flags>, <Field (TCP,TCPerror).window>,
         <Field (TCP,TCPerror).chksum>, <Field (TCP,TCPerror).urgptr>,
         <Field (TCP,TCPerror).options>]
```

4 Scapy test drive

Here are 3 demos on using scapy for sending network traffic: pinging a server, 3-way TCP handshake and sending a HTTP request.

4.1 Pre-requisites

In order to send and receive network traffic using scapy, there are steps required:

1. **Disable kernel RSTs**: By default, the kernel would send an RST packet if it receives packets on ports a connection is not active. This happens because we are bypassing the kernel when sending the packets using scapy and this can completely disconnect the connection on the other side. In order to prevent this from happening, we locally drop all RST packets originating from the kernel using kernel packet filter rules(iptables):

   ```
   $ sudo iptables -A OUTPUT -p tcp --tcp-flags RST RST -j DROP
   ```

   This step should be run after every reboot because iptables configuration isn't permanent. It is also recommended to not make this permanent and deleting it when not required.

2. **Communicating with localhost**: The localhost network interface(*lo*) is a special virtual interface that behaves differently from the regular network interfaces(eg: *eth0*). In order to send packets over localhost using scapy, set the interface scapy should use to *lo* and the socket type to raw sockets:

   ```
   >>> scapy.conf.iface = "lo"
   >>> scapy.conf.L3socket = scapy.L3RawSocket
   ```

   Remember this is only for communication with localhost only!

3. **Giving scapy and tcpdump network capability**: In order to send and receive packets using scapy, both scapy and tcpdump should have sufficient permissions. While the easiest way to achieve this is to run them as root, it is highly recommended to NOT run them as root. Instead, give them both capabilities to capture and send raw packets:

   ```
   $ sudo setcap 'CAP_NET_RAW+eip CAP_NET_ADMIN+eip' /usr/sbin/dumpcap
   $ sudo setcap cap_net_raw=eip $HOME/.virtualenvs/scapy-venv/bin/python
   ```

4.2 Sending ping packets using scapy

scapy has functions *sr1* and *srloop* respectively for sending 1 or multiple packets at layer 3 to a destination. These functions accept an IP packet as argument(including any higher layers), construct the layer 2 packet from the IP packet, send the packet to destination, wait till they receive a response for the packet sent and return it.

Let's try sending 1 ping packet to 8.8.8.8 and get a response:

```
In [8]: scapy.sr1(scapy.IP(dst="8.8.8.8")/scapy.ICMP())
Out[8]: <IP  version=4L ihl=5L tos=0x0 len=28 id=8632 flags= frag=0L ttl=19
         proto=icmp chksum=0x6a0b src=8.8.8.8 dst=10.0.2.15 options=[] |
         <ICMP type=echo-reply code=0 chksum=0xffff id=0x0 seq=0x0 |
          <Padding
        load='\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'|>>>
```

In the above example, a single ICMP echo request is sent to 8.8.8.8 and a response is sought. Notice how no details of ICMP packet were provided: scapy automatically filled in whatever fields it could. Scapy fills up most fields with default values as much as possible. Repeat this above step while simultaneously capturing packets using wireshark.

To send multiple packets, the *srloop* function can be used. Let's send 5 ICMP echo requests to 8.8.8.8 and get the response. It is recommended to capture packets using wireshark to observe the scapy generated traffic and response to it.

```
In [9]: responses = scapy.srloop(scapy.IP(dst="8.8.8.8")/scapy.ICMP(), count=5)
RECV 1: IP / ICMP 8.8.8.8 > 10.0.2.15 echo-reply 0 / Padding
RECV 1: IP / ICMP 8.8.8.8 > 10.0.2.15 echo-reply 0 / Padding
RECV 1: IP / ICMP 8.8.8.8 > 10.0.2.15 echo-reply 0 / Padding
RECV 1: IP / ICMP 8.8.8.8 > 10.0.2.15 echo-reply 0 / Padding
RECV 1: IP / ICMP 8.8.8.8 > 10.0.2.15 echo-reply 0 / Padding

Sent 5 packets, received 5 packets. 100.0% hits.
In [10]: for response in responses[0]:
    ...:     print(response)

<Output of repr() for all responses>
```

4.3 TCP 3 way handshake using scapy

Let's try to exchange packets involved in the TCP 3 way handshake using scapy.

Before you proceed, ensure that you

1. **Configured iptables to drop TCP RST packets generated by the kernel.**
2. **Enabled scapy and tcpdump to be able to capture packets.**
3. **Use raw sockets and configured scapy to use the *lo* interface and raw sockets.**

Let's start a simple TCP server using nc:

```
  $ nc -lvv 7000
```

Let's create a TCP SYN packet in scapy and send it to the server started above:

```
In [12]: tcp_syn_packet = scapy.TCP(sport=10000, dport=7000, seq=1234567890, flags="S")
In [13]: ip_syn_packet = scapy.IP(src="127.0.0.1", dst="127.0.0.1")/a
In [14]: ip_syn_ack_packet = scapy.sr1(ip_syn_packet)
In [15]: print(repr(ip_syn_ack_packet.getlayer(scapy.TCP)))
Out [15]: <Output of repr() of the SYN-ACK packet from the server>
```

If you respond to this SYN-ACK packet with the correct values for sequence and acknowledgement numbers, the TCP 3 way handshake will be completed and the TCP connection will be established. It is highly recommended to complete the connection by responding with an ACK packet to the above SYN-ACK packet. Please attempt doing it on your own. You can verify if the connection was successful by looking for a verbose message from nc, similar to

```
  $ nc -lvv 7000
  Listening on [0.0.0.0] (family 0, port 7000)
 Connection from [127.0.0.1] port 7000 [tcp/afs3-fileserver] accepted (family 2, sport 10000)
```

indicating that a connection was successfully established. Alternatively, you can also verify by capturing packets using wireshark. Please complete this before you move on to the next section.

Warning: The above example is for illustration only. You must always choose initial sequence number at random as discussed in lectures!

4.4 Sending an HTTP GET request using scapy

This tutorial by a member of the Eindbazen CTF team is an excellent illustration how a complete HTTP GET request can be sent using scapy. This further builds upon the previous section to issue a full HTTP GET request by establishing a TCP connection with a webserver.

5 Conclusion

We have done a very quick tour of scapy and some of it's features. We have also seen some examples of how scapy can be used to generate and communicate over network protocols by carefully generating the packets. Here are some more resources on scapy for your reference:

1. http://www.secdev.org/projects/scapy/demo.html
2. https://scapy.readthedocs.io/
3. https://theitgeekchronicles.files.wordpress.com/2012/05/scapyguide1.pdf
4. http://packetlife.net/blog/2011/may/23/introduction-scapy/
5. https://bt3gl.github.io/black-hat-python-infinite-possibilities-with-the-scapy-module.html