TECHNION
Azrieli Continuing Education and External Studies Division

Python - Working with Scapy

Notebook

COPYRIGHT © DO NOT COPY WITHOUT WRITTEN PERMISSION

Introduction to Scapy

Scapy is a Python program that enables the user to send, sniff and dissect and forge network packets. This capability allows construction of tools that can probe, scan or attack networks. The power of Scapy is the network layer deep investigation abilities and crafting the network packets from scratch.

In other words, Scapy is a powerful interactive packet manipulation program. It can forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more. Scapy can easily handle most classical tasks like scanning, tracerouting, probing, unit tests, attacks, or network discovery. It can replace hping, arpspoof, arp-sk, arping, p0f and even some parts of Nmap, tcpdump, and tshark.

Other tools stick to the program-that-you-run-from-a-shell paradigm. The result is an awful syntax to describe a packet. For these tools, the solution adopted uses a higher but less powerful description, in the form of scenarios imagined by the tool's author. As an example, only the IP address must be given to a port scanner to trigger the port scanning scenario. Even if the scenario is tweaked a bit, you still are stuck to a port scan.

Scapy's paradigm is to propose a Domain Specific Language (DSL) that enables a powerful and fast description of any kind of packet. Using the Python syntax and a Python interpreter as the DSL syntax and interpreter has many advantages: there is no need to write a separate interpreter, users don't need to learn yet another language and they benefit from a complete, concise, and very powerful language.

Scapy enables the user to describe a packet or set of packets as layers that are stacked one upon another. Fields of each layer have useful default values that can be overloaded. Scapy does not oblige the user to use predetermined methods or templates. This alleviates the requirement of writing a new tool each time a different scenario is required. In C, it may take an average of 60 lines to describe a packet. With Scapy, the packets to be sent may be described in only a single line with another line to print the result. 90% of the network probing tools can be rewritten in 2 lines of Scapy.

Using Scapy from the CLI environment

When Scapy installed on your system (Kali buit-in) it comes with a self-interpreter that's very similar to the python's one.

Getting help in Scapy is available with couple of options. Scapy is layering protocols together - the protocols separated with "**/**" (slash) character. In Scapy we are defining each protocol layer with certain configurations when the layers that don't configured by the user coming pre-configured by default.

The **ls()** command will list you all the protocol layers supported by Scapy and there are many of them.

```
>>> ls()
AH          : AH
AKMSuite    : AKM suite
ARP         : ARP
ASN1P_INTEGER : None
ASN1P_OID   : None

[..snip..]
```

COPYRIGHT © DO NOT COPY WITHOUT WRITTEN PERMISSION

We could use the **ls()** command with an argument of a protocol layer to see available configurations

```
>>> ls(Ether)
dst        : DestMACField                        = (None)
src        : SourceMACField                      = (None)
type       : XShortEnumField                     = (36864)

>>> ls(IP)
version    : BitField  (4 bits)                  = (4)
ihl        : BitField  (4 bits)                  = (None)
tos        : XByteField                          = (0)
len        : ShortField                          = (None)
id         : ShortField                          = (1)
flags      : FlagsField  (3 bits)                = (<Flag 0 ()>)
frag       : BitField  (13 bits)                 = (0)
ttl        : ByteField                           = (64)
proto      : ByteEnumField                       = (0)
chksum     : XShortField                         = (None)
src        : SourceIPField                       = (None)
dst        : DestIPField                         = (None)
options    : PacketListField                     = ([])
```

Let’s build a packet

Building packets done very easy, for example let’s build an ICMP packet with a custom payload. As known ICMP protocol used to just confirm if a packet gets to the destination or face some issues. ICMP uses to send an “echo-request” packet with a bunch of data that mostly generated from [A-Za-z0-9!@#%%^&] characters.

Firstly, we add the protocols starting from the base - we will use Ether protocol as datalink layer protocol encapsulated with IP version 4 and the ICMP protocol.

```
>>> packet = Ether()/IP()/ICMP()
>>> packet
<Ether  type=IPv4 |<IP  frag=0 proto=icmp |<ICMP  |>>>
```

We could see detailed configuration fields with the **show()** command

```
>>> packet.show()
###[ Ethernet ]###
  dst= ff:ff:ff:ff:ff:ff
  src= 00:00:00:00:00:00
  type= IPv4
###[ IP ]###
     version= 4
     ihl= None
     tos= 0x0
     len= None
```

TECHNION
Azrieli Continuing Education and External Studies Division

COPYRIGHT © DO NOT COPY WITHOUT WRITTEN PERMISSION

```
     id= 1
     flags=
     frag= 0
     ttl= 64
     proto= icmp
     chksum= None
     src= 127.0.0.1
     dst= 127.0.0.1
     \options\
###[ ICMP ]###
        type= echo-request
        code= 0
        chksum= None
        id= 0x0
        seq= 0x0
```

Remember that Scapy is organized as object of arrays - that’s give us easy access to a certain configuration as well as easy new value assigning.

```
>>> packet[IP]
<IP  frag=0 proto=icmp dst=192.168.222.128 |<ICMP  |>>
>>> packet[IP].src
'192.168.222.128'
>>> packet[IP].dst
'192.168.222.128'
>>> packet[IP].dst = '192.168.222.131'
>>> packet[IP]
<IP  frag=0 proto=icmp dst=192.168.222.131 |<ICMP  |>>
>>> packet[IP].src
'192.168.222.128'
>>> packet[IP].dst
'192.168.222.131'
```

Adding a network payload done with a Raw layer (string).

```
>>> packet = packet/"My Uniq Message"
>>> packet
<Ether  type=IPv4 |<IP  frag=0 proto=icmp dst=192.168.222.131 |<ICMP  |<Raw
 load='My Uniq Message' |>>>>
```

Sending Packets

Let's get familiar with the **sr()**, **sr1()**, **srp()**, and **srp1()** functions. Just like the send(), function, the **p** at the end of the function name means that we're sending at Layer 2 instead of Layer 3. The functions with a **1** in them mean that Scapy will send the specified packet and end after receiving 1 answer/response instead of continuing to listen for answers/responses.

TECHNION
Azrieli Continuing Education and External Studies Division

COPYRIGHT © DO NOT COPY WITHOUT WRITTEN PERMISSION

Sending ICMP packet with unique payload

We build a network packet stacked with IP protocol, ICMP and Raw payload. When layer 2 protocols changed or mentioned by the user its force us to use the **srp()** function, in our example we keep it with default values.

```
>>> icmp_packet = IP()/ICMP()/"My Unique Message!"
>>> icmp_packet[IP].dst = "192.168.222.131"
>>> icmp_packet.show()
###[ IP ]###
  version= 4
  ihl= None
  tos= 0x0
  len= None
  id= 1
  flags=
  frag= 0
  ttl= 64
  proto= icmp
  chksum= None
  src= 192.168.222.128
  dst= 192.168.222.131
  \options\
###[ ICMP ]###
     type= echo-request
     code= 0
     chksum= None
     id= 0x0
     seq= 0x0
###[ Raw ]###
        load= 'My Unique Message!'
```

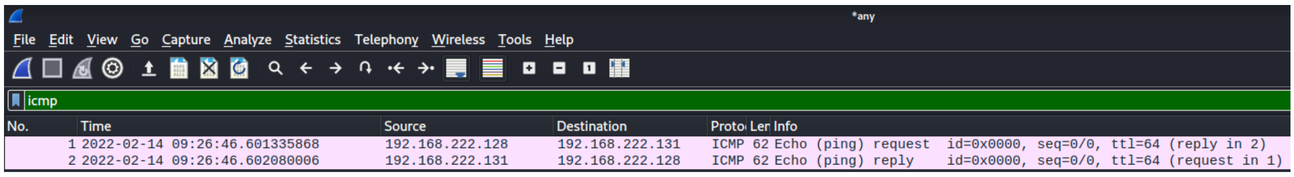
The **sr()** functions is a send-response function. When we like to send packet without waiting to a response, we could use the **send()** or **sendp()** function.

```
>>> sr1(icmp_packet)
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
<IP  version=4 ihl=5 tos=0x0 len=46 id=49503 flags= frag=0 ttl=64 proto=icmp
 chksum=0x7b1a src=192.168.222.131 dst=192.168.222.128 |<ICMP  type=echo-
reply code=0 chksum=0xc5d0 id=0x0 seq=0x0 |<Raw  load='My Unique Message!' |
>>>
```

Above you could see the response.

COPYRIGHT © DO NOT COPY WITHOUT WRITTEN PERMISSION

Wireshark view:


*any
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
icmp

```
No.  Time                           Source           Destination      Proto Len Info
   1 2022-02-14 09:26:46.601335868 192.168.222.128  192.168.222.131  ICMP 62 Echo (ping) request  id=0x0000, seq=0/0, ttl=64 (reply in 2)
   2 2022-02-14 09:26:46.602080006 192.168.222.131  192.168.222.128  ICMP 62 Echo (ping) reply    id=0x0000, seq=0/0, ttl=64 (request in 1)
```

Port scanner building with Scapy

Port scanning is an essential knowledge in cyber security area, the port scanning methodology and techniques are mostly based on TCP communication behavior, with Scapy we could examine the certain cases and provide a custom port scanner then the known tools. For the demonstration I run the ssh and apach2 services to open the 22 and 80 ports.

TCP packet assembling

```
>>> tcp_packet = IP(dst="192.168.222.131")/TCP(sport=49021,dport=80)
>>> tcp_packet.show()
###[ IP ]###
  version= 4
  ihl= None
  tos= 0x0
  len= None
  id= 1
  flags=
  frag= 0
  ttl= 64
  proto= tcp
  chksum= None
  src= 192.168.222.128
  dst= 192.168.222.131
  \options\
###[ TCP ]###
     sport= 49021
     dport= http
     seq= 0
     ack= 0
     dataofs= None
     reserved= 0
     flags= S
     window= 8192
     chksum= None
     urgptr= 0
     options= []
```

TECHNION Azrieli Continuing Education and External Studies Division

COPYRIGHT © DO NOT COPY WITHOUT WRITTEN PERMISSION

Send the packet

```
>>> sr1(tcp_packet)
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
<IP  version=4 ihl=5 tos=0x0 len=44 id=0 flags=DF frag=0 ttl=64 proto=tcp ch
ksum=0xfc76 src=192.168.222.131 dst=192.168.222.128 |<TCP  sport=http dport=
49021 seq=3478597759 ack=1 dataofs=6 reserved=0 flags=SA window=64240 chksum
=0x9f2a urgptr=0 options=[('MSS', 1460)] |<Padding  load='\x00\x00' |>>>
```

When the packet sends with waiting for 1 response, we got a response contains SYN and ACK flag like usual in TCP communication. This is an example of **open port**.

```
>>> tcp_packet[TCP].dport = 100
>>> sr1(tcp_packet)
Begin emission:
Finished sending 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
<IP  version=4 ihl=5 tos=0x0 len=40 id=0 flags=DF frag=0 ttl=64 proto=tcp ch
ksum=0xfc7a src=192.168.222.131 dst=192.168.222.128 |<TCP  sport=100 dport=4
9021 seq=0 ack=1 dataofs=5 reserved=0 flags=RA window=0 chksum=0xb198 urgptr
=0 |<Padding  load='\x00\x00\x00\x00\x00\x00' |>>>
```

When the port is **close** TCP will send back the RST and ACK flag.

```python
#!/usr/bin/python3

from Scapy.all import *

tcp_packet = IP(dst="192.168.222.131")/TCP()
ports = ['80','22','100','3306']

for port in ports:
  tcp_packet[TCP].dport = int(port)
  response = sr1(tcp_packet,verbose=0)
  if "R" in response[TCP].flags:
    print(f"Port {port} is open")
  else:
    print(f"Port {port} is close")
```

COPYRIGHT © DO NOT COPY WITHOUT WRITTEN PERMISSION

TECHNION
Azrieli Continuing Education and External Studies Division

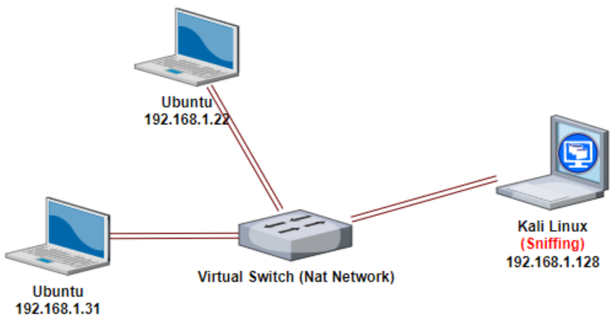
Packet Sniffing and Analysis

Packet sniffing is a technique whereby packet data flowing across the network is detected and observed. Network administrators use packet sniffing tools to monitor and validate network traffic, while hackers my use similar tools for nefarious purposes.

Packet sniffing is available with the very popular tool called Wireshark - don't worry its part of course. The most important thing to understand in packet sniffing is how it happened and when it available.

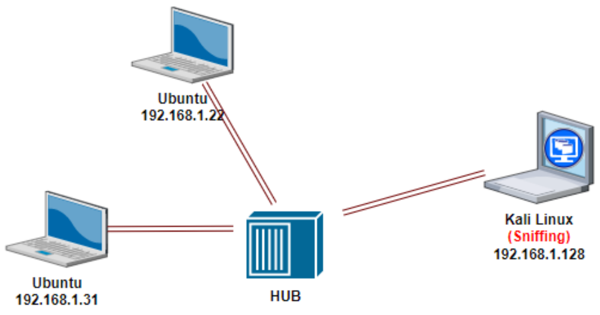
Basically, sniffing is a configuration done to your NIC (network interface card) that allow it overcome the regular network procedure - regular when packet comes to your NIC, its defined by the DST IP address and if its belong to you, you will receive it - to overcome this you enter your NIC to a **promiscuous mode** that allow every packet to be collected regardless the destination.

When the network is managed by a Switch - sniffing is possible but very limited because the Switch device know how to destinate packets by physical ports. Switched network allow communication between two entities only and don't broadcasting the packet through the others. In the case below if my sniffing station will be activated it would collect only the communications of himself with others.



In real-world, when we would like to collect network traffic, we will connect our sniffing station to a specified port called **SPAN** port - when this port is enabled its just collect copy from all the other ports of the network traffic.

When the network managed by a Hub - sniffing is possible in the best level. Hub networks are broadcast networks what means that all the packets will be transmitted to all the network members.



Wireless networks work like HUB - so its less expected that anybody today manage their network using a HUB but if you are part of a Wi-Fi - you could get a great packet trace and investigate the traffic.

TECHNION
Azrieli Continuing Education and External Studies Division

COPYRIGHT © DO NOT COPY WITHOUT WRITTEN PERMISSION

Sniffing with Scapy

Scapy has a built-in function called **sniff()** that accept various arguments:

- **count=<integer>**
  This will count the number of packets we would like to collect.

- **filter=<filter option>**
  Filtering the captured packets and collect only the asked.

- **iface=<interface string>**
  As mentioned above, we need to specify the interface name to allow it to become a "sniffer"

There are more arguments that considered as **advanced**.

Because Scapy able to read most of the packet structure details it become overpowered tool to investigate network traffic. When it's combined with a code it become a monster.

Sniffing ARP

```
>>> packets = sniff(filter="arp", count=10)
>>> packets
<Sniffed: TCP:0 UDP:0 ICMP:0 Other:10>
>>> packets.summary()
Ether / ARP who has 192.168.222.1 says 192.168.222.128 / Padding
Ether / ARP is at 00:50:56:c0:00:08 says 192.168.222.1 / Padding
Ether / ARP who has 192.168.222.1 says 192.168.222.128 / Padding
Ether / ARP is at 00:50:56:c0:00:08 says 192.168.222.1 / Padding
Ether / ARP who has 192.168.222.1 says 192.168.222.128 / Padding
Ether / ARP is at 00:50:56:c0:00:08 says 192.168.222.1 / Padding
Ether / ARP who has 192.168.222.1 says 192.168.222.128 / Padding
Ether / ARP is at 00:50:56:c0:00:08 says 192.168.222.1 / Padding
Ether / ARP who has 192.168.222.1 says 192.168.222.128 / Padding
Ether / ARP is at 00:50:56:c0:00:08 says 192.168.222.1 / Padding
```

Above **sniff()** function is used to catch 10 ARP packets, to get an overview of caught packets you could use the **summary()** function.

The packet sniffing could be saved into a PCAP file (standard of Packet Capture file extension) using **wrpcap()** function.

```
>>> wrpcap("arp-capture.pcap",packets)
```

PCAP files could be read by Scapy using **rdpcap()** function

```
>>> rdpcap("arp-capture.pcap")
<arp-capture.pcap: TCP:0 UDP:0 ICMP:0 Other:10>
```

TECHNION Azrieli Continuing Education and External Studies Division

COPYRIGHT © DO NOT COPY WITHOUT WRITTEN PERMISSION

Network Forensics

Network forensics falls under the digital forensics umbrella and is related to the investigation of evidence left behind on a network following a cyber-attack. This evidence provides clues as to what weaknesses led to the breach and who may be behind it.

Regardless of the 'who', it's the 'how' question that gets answered with a thorough sweep of the network - knowing how a breach occurred allows businesses to draw actionable conclusions about the state of its security and apply fixes accordingly.

Scapy could be used instead of traditional tools to accomplish this task, especially when the form of investigation is repeated on and on. Most of network forensics aim in the cyber security is about to "sign" the attack and find valuable IoC or IoA.

Indicators of attack (**IOA**) focus on detecting the intent of what an attacker is trying to accomplish, regardless of the malware or exploit used in an attack. Just like AV signatures, an IOC-based detection approach cannot detect the increasing threats from malware-free intrusions and zero-day exploits.

An Indicator of Compromise (**IOC**) is often described in the forensics world as evidence on a computer that indicates that the security of the network has been breached. Investigators usually gather this data after being informed of a suspicious incident, on a scheduled basis, or after the discovery of unusual call-outs from the network. Ideally, this information is gathered to create "smarter" tools that can detect and quarantine suspicious files in the future. IoC could defined as an IP address or a domain name or even a TCP port connection.

Scapy could be a great deal when automation applied and not only for investigation of certain static PCAP file, but we could also use Sapy to "run" the certain communication once again using its **"offline"** abilities.

Discovering unique IP addresses

```
#!/usr/bin/python3
import time
from scapy.all import *

packets = rdpcap("case001.pcap")

participated_ips = []

start = time.time()
for packet in packets:
  participated_ips.append(packet[IP].src)
  participated_ips.append(packet[IP].dst)
end = time.time()

with open("uniq-ips.txt","w") as f:
  for ip in set(participated_ips):
    f.write(ip+"\n")
```

TECHNION
Azrieli Continuing Education and External Studies Division

COPYRIGHT © DO NOT COPY WITHOUT WRITTEN PERMISSION

```
print(f"Time: {rount((end-start),2)} seconds")
print(f"Total of unique IP addresses: {len(set(participated_ips))}")
```

The output:

```
Time: 0.54 seconds
Total of unique IP addresses: 185
```

The file created:

```
root💀kali# cat uniq-ips.txt
209.15.36.32
152.163.50.3
157.240.2.38
13.32.80.25
75.98.70.72
172.217.10.134
23.49.188.45
52.23.15.29
52.6.168.107
[..snip..]
```

After we gather all these IP addresses its possible to send it over the AbuseIPDB API using the requests library.

What is AbuseIPDB?

AbuseIPDB is a project dedicated to helping combat the spread of hackers, spammers, and abusive activity on the internet.

Our mission is to help make Web safer by providing a central blacklist for webmasters, system administrators, and other interested parties to report and find IP addresses that have been associated with malicious activity online.

You can report an IP address associated with malicious activity or check to see if an IP address has been reported, by using the search box above.

TECHNION
Azrieli Continuing Education and External Studies Division

COPYRIGHT © DO NOT COPY WITHOUT WRITTEN PERMISSION

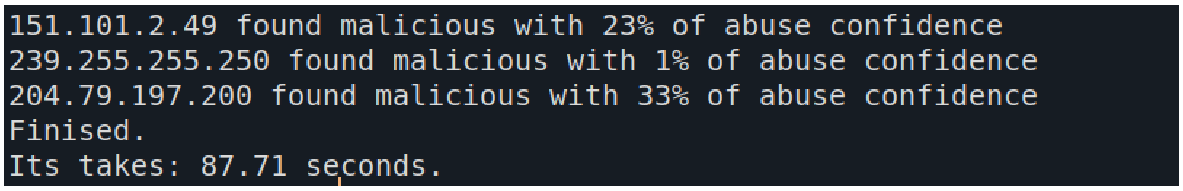
```python
#!/usr/bin/python3

import requests
import json
import time

url = "https://api.abuseipdb.com/api/v2/check"
api = {"Key":"CENSORED"}

start = time.time()
with open("/root/Downloads/uniq-ips.txt","r") as f:
  for ip in f.read().splitlines():
    r = requests.get(url,headers=api,params={"ipAddress":ip})
    response = json.loads(r.text)
    if int(response["data"]["abuseConfidenceScore"]) > 0:
      print(f'{response["data"]["ipAddress"]} found malicious with {response
["data"]["abuseConfidenceScore"]}% of abuse confidence')
      with open("malicious-ips.txt","w") as f:
        f.write(response["data"]["ipAddress"]+"\n")
end = time.time()
print(f"Finised.\nIts takes: {round((end-start),2)} seconds.")
```

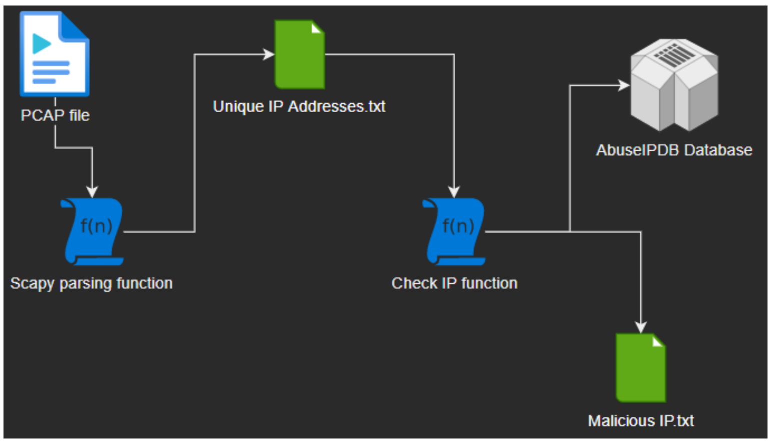
Output:



```
151.101.2.49 found malicious with 23% of abuse confidence
239.255.255.250 found malicious with 1% of abuse confidence
204.79.197.200 found malicious with 33% of abuse confidence
Finised.
Its takes: 87.71 seconds.
```

The above scripts could be combined and with some work it could become more efficient. This is our fully automated process to extract IP addresses from PCAP files and check their maliciously rate perspective.

Short flow:



TECHNION
Azrieli Continuing Education and External Studies Division
COPYRIGHT © DO NOT COPY WITHOUT WRITTEN PERMISSION