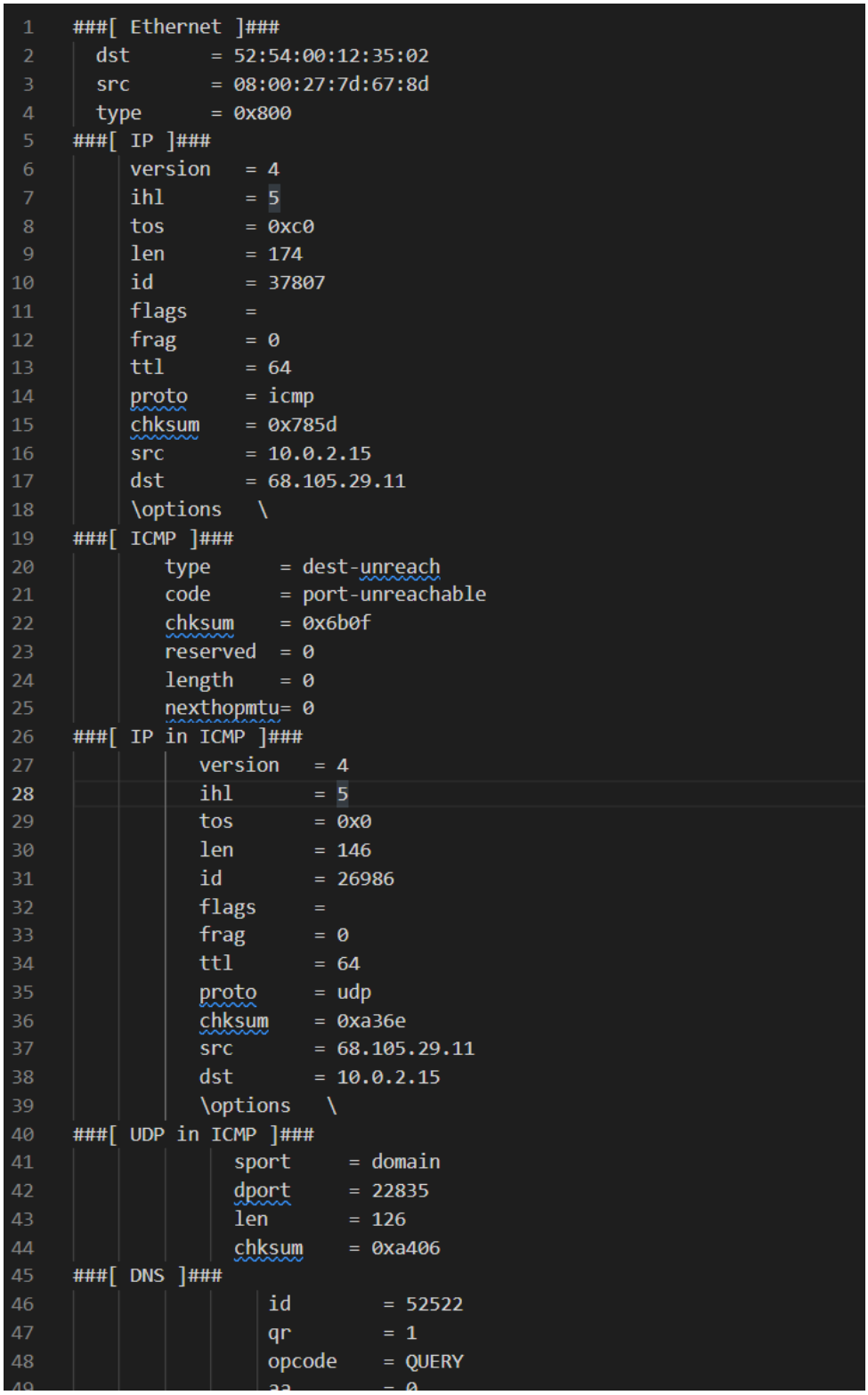
# ECE 571: Lab 5

> *Date: 2022-04-18*
>
> *Author: Zhaohui Yang*

## Task 1.1: Sniffing Packets

### Task 1.1 A

The captured ICMP packets are like the following.



```
###[ Ethernet ]###
  dst       = 52:54:00:12:35:02
  src       = 08:00:27:7d:67:8d
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0xc0
     len       = 174
     id        = 37807
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x785d
     src       = 10.0.2.15
     dst       = 68.105.29.11
     \options   \
###[ ICMP ]###
        type      = dest-unreach
        code      = port-unreachable
        chksum    = 0x6b0f
        reserved  = 0
        length    = 0
        nexthopmtu= 0
###[ IP in ICMP ]###
           version   = 4
           ihl       = 5
           tos       = 0x0
           len       = 146
           id        = 26986
           flags     =
           frag      = 0
           ttl       = 64
           proto     = udp
           chksum    = 0xa36e
           src       = 68.105.29.11
           dst       = 10.0.2.15
           \options   \
###[ UDP in ICMP ]###
              sport     = domain
              dport     = 22835
              len       = 126
              chksum    = 0xa406
###[ DNS ]###
                 id        = 52522
                 qr        = 1
                 opcode    = QUERY
                 aa        = 0
```

## Task 1.1 B

Similarly, use the Python program `sniff_pkt.py` to sniff packets with particular types on the other hand. Corresponding results are as anticipated.

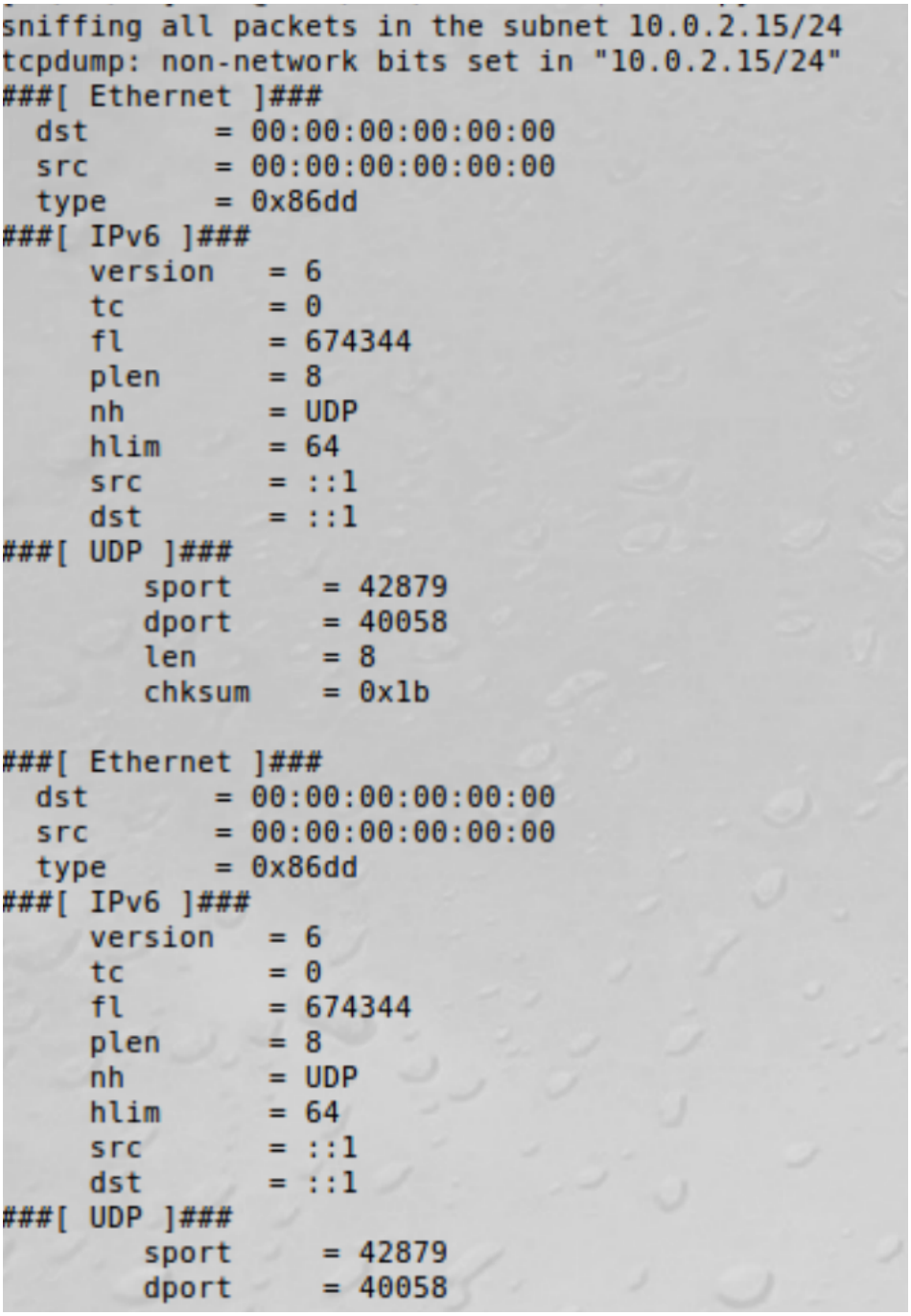
1. ICMP packets

```
sniffing all ICMP packets ...
###[ Ethernet ]###
  dst       = 52:54:00:12:35:02
  src       = 08:00:27:7d:67:8d
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0xc0
     len       = 100
     id        = 57256
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x2cae
     src       = 10.0.2.15
     dst       = 68.105.29.11
     \options   \
###[ ICMP ]###
        type      = dest-unreach
        code      = port-unreachable
        chksum    = 0x6ac5
        reserved  = 0
        length    = 0
        nexthopmtu= 0
###[ IP in ICMP ]###
           version   = 4
           ihl       = 5
           tos       = 0x0
           len       = 72
           id        = 11685
           flags     =
           frag      = 0
           ttl       = 64
           proto     = udp
           chksum    = 0xdf7d
           src       = 68.105.29.11
```

2. TCP packets

```
sniffing all TCP packets from IP address 192.168.1.1 and port 23
tcpdump: 'tcp' modifier applied to host
###[ Ethernet ]###
  dst       = 00:00:00:00:00:00
  src       = 00:00:00:00:00:00
  type      = 0x86dd
###[ IPv6 ]###
     version   = 6
     tc        = 0
     fl        = 674344
     plen      = 8
     nh        = UDP
     hlim      = 64
     src       = ::1
     dst       = ::1
###[ UDP ]###
        sport     = 42879
        dport     = 40058
        len       = 8
        chksum    = 0x1b
```

3. packets in the particular subnet




## Task 1.2: Spoofing ICMP Packets

The packet spoofing program is like this

```
from scapy.all import *
a = IP()
a.dst = '10.0.2.3'
send(a / ICMP())
```

and the detailed content is in the `spoof_icmp_pkt.py` file.

In the spoofing & observing experiment, we opened two terminal windows. In the first terminal, we monitor packets by using `scapy` command line interface.

```
>>> data = sniff()
```

After the spoofing program is executed for a while in the other terminal, we manually intercepted the sniffing process and observe sniffed packets by executing

```
>>> data.show()
```

The result as following demonstrated there is exactly echo-request to `10.0.2.3` and echo-reply returned from `10.0.2.3`.

```
0000 Ether / ARP who has 10.0.2.3 says 10.0.2.15
0001 Ether / ARP is at 52:54:00:12:35:03 says 10.0.2.3 / Padding
0002 Ether / IP / ICMP 10.0.2.15 > 10.0.2.3 echo-request 0
0003 Ether / IP / ICMP 10.0.2.3 > 10.0.2.15 echo-reply 0 / Padding
```

## Task 2.1: ARP Cache Poisoning

Herein I use a laptop to conduct ARP cache poisoning to my PC computer. Similar results occurs like below. Below the figure shows how the ARP cache table has been changed after ARP poisoning.


```
(base) PS E:\Git-Projects\NLP-GitHub-Classroom\assignment-3-Youngcius> arp -a | Select-String 192.168.1.1

接口: 192.168.131.1 --- 0x8
  192.168.131.255       ff-ff-ff-ff-ff-ff     静态
接口: 192.168.1.110 --- 0x1a
  192.168.1.1           80-8f-1d-1a-d5-9a     动态
  192.168.1.107         50-5b-c2-d3-bb-fd     动态


(base) PS E:\Git-Projects\NLP-GitHub-Classroom\assignment-3-Youngcius> arp -a | Select-String 192.168.1.1

接口: 192.168.131.1 --- 0x8
  192.168.131.255       ff-ff-ff-ff-ff-ff     静态
接口: 192.168.1.110 --- 0x1a
  192.168.1.1           f0-2f-4b-0b-65-f9     动态
  192.168.1.104         f0-2f-4b-0b-65-f9     动态
  192.168.1.107         50-5b-c2-d3-bb-fd     动态
```

## Task 2.1 A (using ARP request)

Detailed content is in `task2a.py`.

## Task 2.1 B (using ARP reply)

Detailed content is in `task2b.py`.

## Task 2.1 C (using ARP gratuitous message)

Detailed content is in `task2c.py`.