Lab 6 - Networking Monitoring (Linux)

Why is **Networking Important**?
Having a well-established network has become an important part of our lives. The easiest way to expand your network is to build on the relationships with people you know; family, friends, classmates and colleagues. We are all expanding our networks daily.

Objectives

- Offer an introduction to Network monitoring.
- Get you acquainted with a few Linux standard monitoring tools and their outputs, for monitoring the impact of the Network on the system.
- Provides a set of insights related to understanding networks and connection behavior.

Contents

Tasks

Introduction

01. Ethernet Configuration Settings

Unless explicitly changed, all Ethernet networks are auto negotiated for speed. The benefit of this is largely historical when there were multiple devices on a network at **different speeds and duplexes**.
Most enterprise Ethernet networks run at either 100 or 1000BaseTX. Use **ethtool** to ensure that a specific system is synced at this speed.
In the following example, a system with a 100BaseTX card is running auto negotiated in *10BaseT*.

```
# ethtool eth0
Settings for eth0:
        Supported ports: [ TP MII ]
        Supported link modes:   10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
        Supports auto-negotiation: Yes
        Advertised link modes:  10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
        Advertised auto-negotiation: Yes
        Speed: 10Mb/s
        Duplex: Half
        Port: MII
        PHYAD: 32
        Transceiver: internal
        Auto-negotiation: on
        Supports Wake-on: pumbg
        Wake-on: d
        Current message level: 0x00000007 (7)
        Link detected: yes
```

The following command can be used to force the card into 100BaseTX:

```
# ethtool -s eth0 speed 100 duplex full autoneg off
```

02. Monitoring Network Throughput

It is impossible to control or tune the switches, wires, and routers that sit in between two host systems. The best way to test **network throughput** is to send **traffic** between two systems and measure statistics like **latency** and **speed**.

Using iptraf for Local Throughput

The **iptraf** utility (http://iptraf.seul.org) provides a **dashboard** of **throughput** per Ethernet interface. (Use: *# iptraf –d eth0*)

Using netperf for Endpoint Throughput

Unlike **iptraf** which is a **passive interface** that monitors traffic, the **netperf** utility enables a system administrator to perform **controlled tests** of **network throughput**. This is extremely helpful in determining the throughput from a client workstation to a heavily utilised server such as a file or web server. The **netperf** utility runs in a **client/server** mode.
To perform a basic controlled throughput test, the **netperf** server must be running on the server system (*server# netserver*).
There are multiple tests that the **netperf** utility may perform. The most basic test is a standard throughput test. The following test initiated from the client performs a 30 second test of TCP based throughput on a LAN. The output shows that the throughput on the network is around 89 mbps. The server (192.168.1.215) is on the same LAN. This is exceptional performance for a 100 mbps network.

```
client# netperf -H 192.168.1.215 -l 30
TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to
192.168.1.230 (192.168.1.230) port 0 AF_INET
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec

 87380  16384  16384    30.02      89.46
```

Another useful test using **netperf** is to monitor the amount of **TCP request and response** transactions taking place per second. The test accomplishes this by creating a single TCP connection and then sending multiple request/response sequences over that connection (ack packets back and forth with a byte size of 1). This behavior is similar to applications such as RDBMS executing multiple transactions or mail servers piping multiple messages over one connection.
The following example simulates TCP request/response over the duration of 30 seconds.

```
client# netperf -t TCP_RR -H 192.168.1.230 -l 30
TCP REQUEST/RESPONSE TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET
to 192.168.1.230 (192.168.1.230) port 0 AF_INET
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec

16384  87380  1        1       30.00    4453.80
16384  87380
```

In the previous output, the network supported a transaction rate of 4453 psh/ack per second using 1 byte payloads. This is somewhat unrealistic due to the fact that most requests, especially responses, are greater than 1 byte.
In a more realistic example, a **netperf** uses a **default size** of **2K** for requests and **32K** for responses.

```
client# netperf -t TCP_RR -H 192.168.1.230 -l 30 -- -r 2048,32768
TCP REQUEST/RESPONSE TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to
192.168.1.230 (192.168.1.230) port 0 AF_INET
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec

16384  87380  2048     32768   30.00     222.37
16384  87380
```

The transaction rate reduces significantly to 222 transactions per second.

Using iperf to Measure Network Efficiency

The **iperf** tool is similar to the **netperf** tool in that it checks connections between two endpoints. The difference with **iperf** is that it has more in-depth checks around TCP/UDP efficiency such as window sizes and QoS settings. The tool is designed for administrators who specifically want to **tune TCP/IP stacks** and then test the effectiveness of those stacks. The **iperf** tool is a single binary that can run in either server or client mode. The tool runs on port **5001** by default. In addition to TCP tests, **iperf** also has UDP tests to measure packet loss and jitter.

03. Individual Connections with tcptrace

The **tcptrace** utility provides detailed TCP based information about specific connections. The utility uses **libpcap** based files to perform an analysis of specific TCP sessions. The utility provides information that is at times difficult to catch in a TCP stream. This information includes:

- **TCP Retransmissions** – the amount of packets that needed to be sent again and the total data size
- **TCP Window Sizes** – identify slow connections with small window sizes
- **Total throughput** of the connection
- **Connection duration**

For more information refer to pages 34-37 from Darren Hoch's Linux System and Performance Monitoring.

04. TCP and UDP measurments

"Time remaining: 12 Hours! What's wrong with the network?"

- This issue is all too common and it has nothing to do with the network.

TCP measurements: throughput, bandwidth

- **Capacity**: link speed
- Narrow link: link with the lowest capacity along a path
- Capacity of the end-to-end path: capacity of the narrow link
- **Utilized bandwidth**: current traffic load
- **Available bandwidth**: capacity – utilized bandwidth
- Tight link: link with the least available bandwidth in a path
- **Achievable bandwidth**: includes protocol and host issues
- Many things can **limit TCP throughput**:
- Loss
- Congestion
- Buffer Starvation
- Out of order delivery

TCP performance: window size

- In data transmission, TCP sends a certain amount of data and then pauses;

- To ensure **proper delivery of data**, it doesn't send more until it receives an acknowledgment from the remote host;

TCP performance: Bandwith Delay Product (BDP)

- The further away the two hosts, the longer it takes for the sender to receive the acknowledgment from the remote host, **reducing overall throughput**.
- To overcome BDP, we send more data at a time ⇒ **we adjust the TCP Window**. Telling TCP to send more data per flow than the default parameters.

To get **full TCP performance** the TCP window needs to be large enough to accommodate the Bandwidth Delay Product.

TCP performance: parallel streams - read/write buffer size

- TCP breaks the stream into pieces transparently
- **Longer writes often improve performance**
- Let TCP “do it’s thing”
- Fewer system calls
- How?
- -l <size> (lower case ell)
- Example –l 128K
- **UDP doesn’t break up writes, don’t exceed Path MTU**
- The –P option sets the number of streams to use

UDP measurements

- Loss
- Jitter
- Out of order delivery
- Use -b to specify target bandwidth (default is 1M)

Takeaways

- Check to make sure all **Ethernet interfaces** are running at proper rates.
- Check **total throughput** per **network interface** and be sure it is inline with network speeds.
- Monitor **network traffic** types to ensure that the **appropriate traffic** has precedence on the system.

Tasks

---

01. [10p] Local Network Scan

One of the first things that you have to do when trying to guide yourself in a network is to determine what devices you can contact. **arp-scan** is a tool that helps detect devices in a local network by sending ARP echo requests for IP addresses in an arbitrary range.

[5p] Task A - Using arp-scan

Analyse the output of the following command:

```
$ sudo arp-scan --interface eth0 --localnet
```

- **--interface** : specifies the interface used (if absent, uses the lowest numbered configured interface)

- **--localnet** : targets that are scanned are automatically determined from the interface's IP address and its network mask

[5p] Task B - ARP vs ICMP echo request

Use localnet-ping.sh to ping each host detected by **arp-scan**. Why do some of them not respond to your ping?

02. [20p] Traffic monitoring - Tcpdump

In most of the situations presented in all the laboratories we have already gone through, we have seen numerous tools whose output helps us to understand the behavior of the system we are analyzing. Next, we choose the most used Linux utility for **analyzing transferred packets** in a conversation between two or more systems.

Tcpdump captures and prints out a **description** of the contents of **packets** on a **network interface**. Tcpdump utilises the ***libpcap*** library for packet capturing. The packet details can either be displayed on the screen or saved to files.

Supported options by tcpdump command:

Options	**Description**
–version	print the tcpdump and libpcap version strings and exit
-h, –help	print the tcpdump and libpcap version strings, print a usage message, and exit
-B buffer_size	set the operating system capture buffer size to *buffer_size*, in units of KiB
-c count	exit after receiving *count* packets
-D	print the list of the network interfaces on which tcpdump can capture packets
-i interface	report the results of compiling a filter expression on *interface*
-n	don't convert addresses (host addresses, port numbers) to names
-s snaplen	truncate *snaplen* bytes of data from each packet rather than the default
-t	don't print a timestamp on each dump line
-v	produce more verbose output
-w file	write the raw packets to *file* rather than parsing and printing them out
-r file	read packets from file
-A	print each packet in ASCII

- Check if **tcpdump** is installed and which **version** is installed.
- Check out the **network interfaces** available on your system.
- After starting a capture on all interfaces, you can always stop it using ***control + c***.

[10p] Task A - Understanding traffic

a) Start a capture that stops by itself after getting 10 packets on all interfaces.
b) Have a look at the output. You can notice that **host names** are used instead of **IP addresses**, and commonly known port are replaced with application names. Use a command to display the **IP addresses** and port numbers instead of these names.

Tcpdump triggers itself DNS traffic as it captures, if it is ran without the *-n* option. The utility will trigger reverse or PTR DNS lookups to find hostnames for IP addresses as it captures them. So, from now on, use *-n*.

- What is the capture size?

What does this mean? It means that tcpdump will keep all those bytes for analysis. We don't need all this information for now, so change the capture size to 96 bytes. The Ethernet, IP and TCP headers are the in the first 64 bytes of the packets, so capturing 96 bytes per packet is more than enough to capture these headers.

c) Do the capture again with the output limitation.

The **TCP flags** are **SYN**, **ACK**, **RESET**, **FIN**, **URGENT** and **PUSH**. All flags are represented by the first letter, with the exception of ACK which is represented by a **dot**.

d) Start a new capture only on the interface that connects you to the internet, without printing the timestamp on each dump line. Open a separate terminal and try to connect through ssh somewhere. Spot the **3-way handshake** in the capture.

e) Repeat what you did for the previous task, but add *-S* to your tcpdump command. Figure out what has changed, and why.

Check out the **window size** in the previous capture. Since window scaling is enabled, that is not the *actual* window size. Notice the window scaling factor (***wscale***) in the 3-way handshake output. The scaling factor translates in multiplying the receive window by 2 to the power of wscale. So the *real* window size is the window value shown in the capture, multiplied by 2 to the power of wscale.

The **length field** stands for packet length, and represents the number of bytes in the layer 4 headers, and it matches with the sequence numbers **(packet_length = larger_seq_no - smaller_seq_no)**.

[10p] Task B - DNS capture

If we are the victims of a possible cyber attack (DNS hijacking), the DNS request packages are investigated.

We will simulate the monitoring of all DNS packages.

a) Capture an output for a DNS request.

b) Save a capture to a file. Use the appropriate options so that:

- it displays the number of packets captured
- the capture stops after 30 packets

c) Read the contents of the capture file.

d) Using filters helps you view just the types of traffic that you are interested in and ignore the rest. Create short captures of up to 5 packets for the following cases:

- Capture traffic just from the IP 8.8.8.8
- Capture traffic having the source IP 8.8.8.8
- Capture traffic to or from your PC on port 80
- Capture traffic to or from your PC on port 80 or port 443

03. [15p] Transfer layer analysis

For this task, we want to analyze the network for the network traffic while using Proxychains and observe the effect on the performance.

Proxying refers to the technique of bouncing your Internet traffic through multiple machines to hide the identity of the original machine, or to overcome network restrictions. **ProxyChains** is a tool that hackers often use to accomplish this goal, it forces any TCP connection made by any given application to go through proxies.

[10p] Task A - Proxychains performance

First of all, we want to set up proxychains. For this, you can follow this link and use any proxy server. After this, use a tool (not **tcpdump**) to analyze the effects produced by proxying in comparison with normal traffic.

Some tcpdump alternatives:

- wireshark
- sysdig
- smartsniff
- tcpflow

04. [15p] Monitoring Bandwidth Used by Processes

Nethogs is a small 'net top' tool that shows the bandwidth used by individual processes and sorts the list putting the most intensive processes on top. Nethogs returns the PID, user and the path of the program.

[10p] Task A - Monitoring the behaviour

Open a data streaming website (example: youtube.com) and start downloading/playing content. Use nethogs (sudo apt-get install nethogs) to find the process that uses **most of the bandwidth** and kill it.

05. [30p] NetfilterQueue

A tool that you have previously used is **iptables**. This user space tool interacts with the packet filtering framework that is implemented in the kernel (i.e. netfilter) in order to allow you to specify your firewall rules.

Netfilter is a framework for packet mangling, outside the normal Berkeley socket interface. It has four parts. Firstly, each protocol defines “hooks” (IPv4 defines 5) which are well-defined points in a packet’s traversal of that protocol stack. At each of these points, the protocol will call the netfilter framework with the packet and the hook number.

In this exercise, we introduce **Netfilter Queues**, an **iptables** target (also part of the netfilter project). What's interesting about this tool is that the hook that it inserts in the network protocol stack does not immediately decide what happens to the packet, based only on the usual vectors (protocol, src/dst IP, src/dst port, etc). Instead, it sends the packet to a user space process for analysis. All packets are buffered and the user space process can retrieve them, evaluate their contents and decide whether to **ACCEPT, DROP or RETURN** them. The messages do not need to be evaluated in the order that they were received. Moreover, the process can even alter the packets and ask the kernel to use the modified version (this happens to be very useful when implementing MitM attacks).

Next, we will look at two examples of **Netfilter Queue** analysis programs that also alter the traffic. First, download the demo scripts. Also, make sure that you install all dependencies before starting the tasks:

```
$ sudo apt update
$ sudo apt install -y python3-pip wireshark openssh-server libnetfilter-queue1 libnetfilter-queue-dev nfqueue-bindings-python python3-scapy
$ pip3 install NetfilterQueue
```

[15p] Task A - DNS hijack

In this task we will intercept **all DNS responses** and alter the returned IP address for a certain domain name. Before proceeding with the following commands, make sure you have a ssh server

running on your machine (we will mess around with that later on). The main goal is to understand how it all works, so make sure you read the script.
If you feel that you need a better understanding of the DNS message format, check out Let's hand write DNS messages.

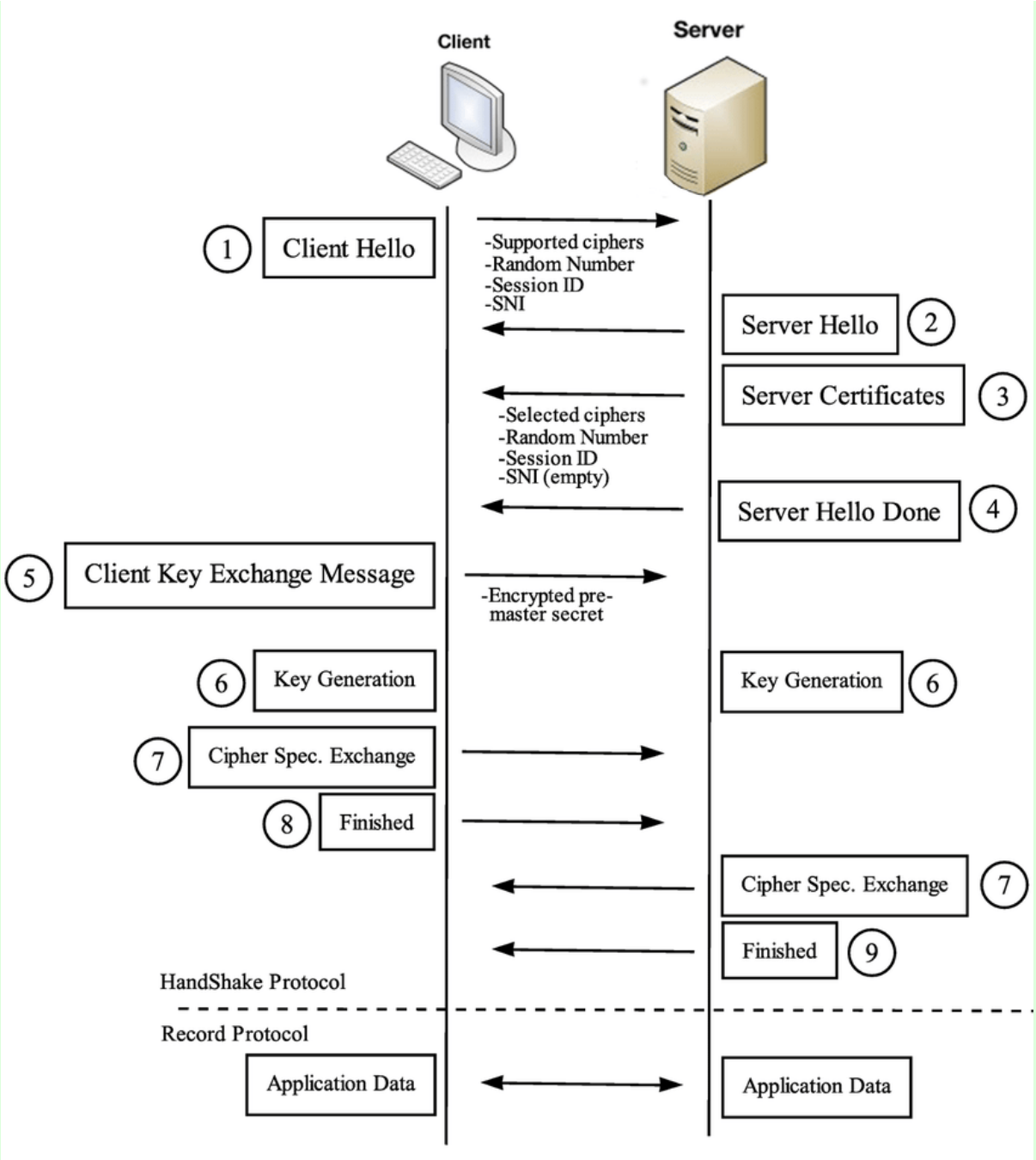
```
$ dig +short fep.grid.pub.ro
$ sudo iptables -I INPUT -p udp --sport 53 -j NFQUEUE --queue-num 1
$ sudo ./mitm-dns fep.grid.pub.ro. 127.0.0.1

$ dig +short fep.grid.pub.ro
$ ssh student@fep.grid.pub.ro
   Password: student
$ sudo iptables -D INPUT 1
```

Let's take a look at what just happened:

- first, we use **dig** to obtain the IP address of *fep.grid.pub.ro* and it should be *141.85.241.99* or something along those lines
- next, we add an **iptables** rule: *redirect all UDP incoming traffic that originated from port 53 to queue number 1 for validation in userspace*
- then, we run the python3 script that will subscribe to queue number 1 and will replace all DNS reponses to fep's domain with 127.0.0.1
- after checking the IP again with **dig**, we notice that the script worked
- what may be a possible implication? Well… let's try connecting to *fep* via **ssh**
- finally, we stop the script and delete the **iptables** rule

[15p] Task B - TLS downgrade & Wireshark

TLS is a cryptographic protocol meant to provide secure communication over a network (e.g. *https*). All sessions that use this protocol start with a *TLS handshake* (see figure below). When the client contacts the server, it sends a *Client Hello*. This message includes a list of supported encryption algorithms. The server receives this message, chooses the best encryption algorithm (that it also knows) and responds with a *Server Hello*, containing the chosen algorithm.



So we know how **Netfilter Queues** and **TLS** work. In this task we will use **wireshark** to detect abnormal traffic. This time, our script will intercept all *Client Hello* messages and replace the supported cipher suite list with a single (weaker) item that the server will be forced to select. Let's try to connect to *ocw.cs.pub.ro* and see what cipher suite it normally chooses:

```
$ echo | openssl s_client -connect ocw.cs.pub.ro:443
```

The answer should be *ECDHE-RSA-AES256-GCM-SHA384*. Now, make sure you have **wireshark** installed and get a network capture of this unaltered handshake. Save it for later. Next, set up the **iptables** rule and run the process:

```
$ sudo iptables -I OUTPUT -p tcp --dport 443 -j NFQUEUE --queue-num 1
$ sudo ./mitm-tls_downgrade.py

$ sudo iptables -D OUTPUT 1
```

Try to capture the *Client Hello* once again with **wireshark**. Place the two captures side by side and identify the cipher suites lists. Which algorithm did our script force the server to accept?