The exercises are designed for students to finish in an individual capacity. The exercises are not designed to be completed in tutorial sessions but rather to give you some tasks and a starting point to continue and complete on your own.

---

1 Lab Setup

1.1 Installing Project

To complete this week's lab tasks, we will be using a new network topology. Run the command below on the GNS3 VM shell to download the project. If you SSH into the VM from your host OS terminal, you can simply copy and paste the command instead of typing it manually.

```
gdown 1sbyoIvG1ynO343PhS5hqSQSMp4YZZZqa ; sudo bash ./install_EmailSecurity.sh
```

Alternatively, you can use the link below to download the same project. However, if you are connected to the **Monash Wi-Fi**, this method may not work. In that case, please use a mobile hotspot. (single command)

```
wget https://sniffnrun.com/install_EmailSecurity.sh --no-check-certificate ; \
sudo bash ./install_EmailSecurity.sh
```

1.2 Topology

Open the EmailSecurity project on GNS3 and start all nodes. The topology includes two email servers: one for **redmail.com** and another for **bluemail.com**. Each mail domain has the following email addresses configured:

- alice@redmail.com
- albert@redmail.com
- bob@bluemail.com
- bobby@bluemail.com

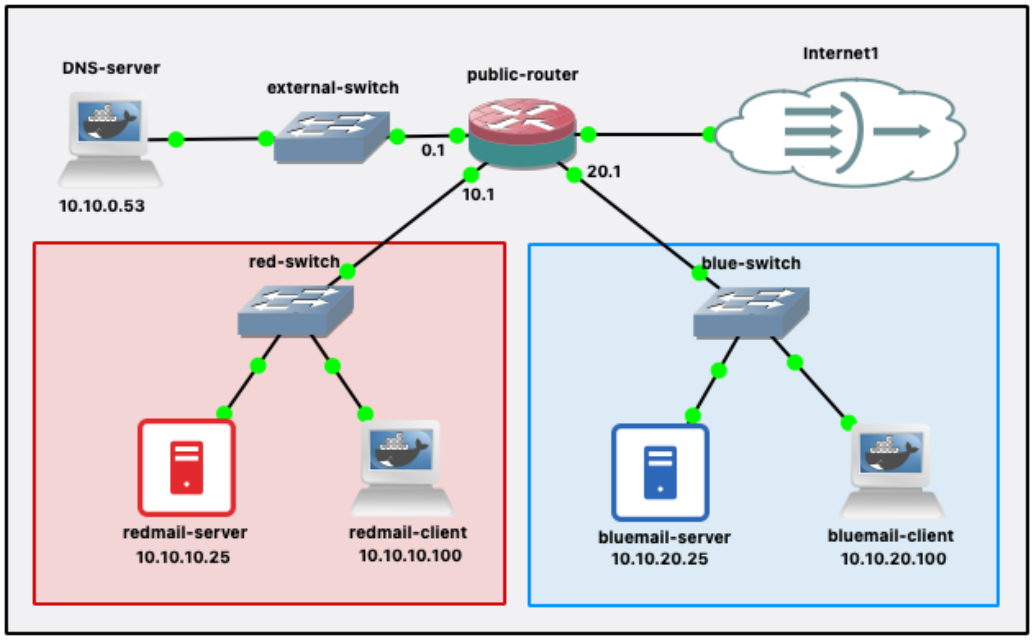

Figure 1: Topology

2 Sending Emails

Let's exchange emails between the two domains and analyze the traffic to understand the flow of messages and the use of protocols.

Before we begin, **start capturing traffic** on the link between the redmail-server and the red-switch using Wireshark.

2.1 Sending text-only Emails

We will use a Python-based email client to send an email from `alice@redmail.com` to `bob@bluemail.com`. The `email-client.py` file is located in the `/home` directory of **redmail-client**.

Open the terminal on **redmail-client** and run the following commands to change the working directory to `/home`.

```
cd /home
```

Now run the following command to send an email using `email-client.py`. **Note:** This is a single command. Copy all lines into the terminal and press `Enter`.

```
python3 email-client.py \
  --server mail.redmail.com --sender alice@redmail.com --recipient bob@bluemail.com \
  --subject "Plain text email" --body "This email is plain text because SMTP supports \
  only text by default."
```

Open the terminal of the bluemail-server and run the below command to view the received email.

```
cat /home/bob/mailbox
```

1. Did you notice any DNS traffic? What is the role of DNS in sending email?
2. What protocol is used between the email client and the **redmail-server**? Is the traffic encrypted?
3. What protocol is used between the **redmail-server** and the **bluemail-server**? Is the traffic encrypted?
4. Can you read the content of the email? Is the email encrypted while it is stored on the destination email server?

Note: In this lab, we are directly examining the email received and stored on the destination email server. In the real world, you would use either the **IMAP** or **POP3** protocols to download or view emails from the email server.

2.2 Sending emails with attachments

The **SMTP** protocol, by default, only supports sending text-based emails. If we need to send an attachment, an extension on top of SMTP is required. This extension is called **MIME**.

Let us now send another email with an attachment. First, we need to convert the file to **Base64** text so that it can be attached using the MIME extension.

Open the `monash.png` file located in the `/home` directory of the **redmail-client** using the terminal-based image viewer **Chafa**.

```
chafa monash.png
```

Run the following command to convert the `monash.png` image to **Base64** and save it as `monash.b64`.

```
base64 monash.png > monash.png.b64
```

You can view the **Base64** conversion of the image using the following command.

```
cat monash.png.b64
```

Now that we have a file converted to **Base64** text, let us send the Base64-encoded image as an attachment over the **SMTP** protocol using **MIME** (Multipurpose Internet Mail Extensions).

```
python3 email-client.py \
--server mail.redmail.com --sender alice@redmail.com --recipient bob@bluemail.com \
--subject "Plain text email" --body "This email is an email with an attachment" \
--b64 "$(cat monash.png.b64)" --b64-type image/png
```

If you run the following command on the **bluemail-server**, you should be able to see the attachment in the email as Base64-encoded text.

```
cat /home/bob/mailbox
```

Since we do not have an email client that supports **MIME**, we must manually decode the Base64-encoded attachment.

Copy the Base64 text of the attachment to the clipboard from the email. Then, run the following command to create a new file named `monash.png.b64`, paste the content from the clipboard into the file, and save it.

```
nano monash.png.b64
```

Decode the Base64 file to retrieve the image file.

```
base64 -d -i monash.png.b64 > monash.png
```

Now view the decoded image using the **Chafa** tool.

```
chafa monash.png
```

You should now see the same image that you viewed on the **redmail-client**, successfully sent to the **bluemail-server** as an email attachment over **SMTP** using **MIME**.

3 Securing Emails on the Wire

In the previous tasks, you may have noticed that emails are transmitted in plain text over the public Internet by default. Let us now look at how to configure security at each step of the email flow.

3.1 Securing SMTP Between Servers

Edit the `/etc/postfix/main.cf` file on **BOTH** the **redmail-server** and the **bluemail-server** using the **nano** text editor.

```
nano /etc/postfix/main.cf
```

Locate the following configuration lines and set them to always encrypt traffic when sending and receiving emails over **SMTP**.

```
smtpd_tls_security_level = encrypt
smtp_tls_security_level = encrypt
```

Run the following command on **BOTH** the **redmail-server** and the **bluemail-server** to restart the Postfix service.

```
service postfix restart
```

3.2 Securing Client-to-Server SMTP

SMTP traffic between the client and the server can be secured using two methods: **STARTTLS** (xplicit TLS, port 587) and **SMTPS** (implicit TLS, port 465). Let us configure both on the **redmail-server**.

Open the `/etc/postfix/master.cf` file.

```
nano /etc/postfix/master.cf
```

Uncomment the last few lines of the file as shown below, then press `CTRL+X` to save and exit.

```
submission inet n       -       y       -       -       smtpd
  -o syslog_name=postfix/submission
  -o smtpd_tls_security_level=encrypt

smtps     inet  n       -       y       -       -       smtpd
  -o syslog_name=postfix/smtps
  -o smtpd_tls_wrappermode=yes
  -o smtpd_tls_security_level=encrypt
```

Restart the **Postfix** service.

```
service postfix restart
```

3.2.1 Sending Emails Using STARTTLS

Our Python-based email client supports both **STARTTLS/SMTP** and **SMTPS**. Let us first send an email using **STARTTLS**. Use the `--tls` parameter to use STARTTLS.

Do not forget to continue capturing traffic using **Wireshark**.

```
python3 email-client.py \
  --server mail.redmail.com --sender alice@redmail.com --recipient bob@bluemail.com \
  --subject "Using STARTTLS" --body "This email is using STARTTLS" --port 587 --tls
```

3.2.2 Sending Emails Using SMTPS

Now let us send another email using our Python email client, but this time use the `--smtps` parameter to enable **SMTPS**.

```
python3 email-client.py \
  --server mail.redmail.com --sender alice@redmail.com --recipient bob@bluemail.com \
  --subject "Using SMTPS" --body "This email is using SMTPS" --port 465 --smtps
```

- Compare the traffic between the two approaches of using TLS for SMTP. Do you notice any differences? Is one method more secure than the other?
- Which of these modes were used when sending data between servers?

Note: In this lab, we do not cover securing the **POP3** and **IMAP** protocols. Both of these protocols support the same two methods of using TLS:

- **POP3S** (port 995)
- **STARTTLS with POP3** (port 110)
- **IMAPS** (port 993)
- **STARTTLS with IMAP** (port 143)

4 End-to-End security for emails

TLS secures the channel between two mail servers or between a client and a server. This prevents eavesdropping in transit on that specific link. However, once the email arrives at the server, it is decrypted and stored in plain text unless additional measures are taken. If the server is compromised, or if the administrator, government, or ISP requests access to the mail, they can read everything.

To ensure end-to-end security for emails, there are two widely used approaches: **S/MIME** and **PGP**. For this lab, we will use **PGP** to send end-to-end encrypted emails.

Note: PGP (Pretty Good Privacy) was the original proprietary encryption software, while GPG (GNU Privacy Guard) is its free, open-source implementation based on the OpenPGP standard. Both handle encryption and digital signatures, but today on Linux, GPG is the go-to tool (gpg) since it's open, standardized, and widely supported.

4.1 Create and share PGP Public Key

Alice is sending a **PGP**-encrypted email to Bob. To encrypt the message, Alice needs Bob's public key.
Let us create a PGP key for Bob. Open the terminal on the **bluemail-client**. When prompted, provide Bob's name and email address (`bob@bluemail.com`). The unique identifier for PGP keys is the associated email address.

```
gpg --gen-key
```

You can view the created key in the local **PGP** keyring.

```
gpg --list-keys
```

Now let us export Bob's public key in an encoded format so that we can copy it to Alice's computer.

```
gpg --export --armor bob@bluemail.com > bob-public.asc
```

Run the following command and copy the contents of `bob-public.asc`. On Alice's computer (**redmail-client**), create a new file using **nano**, paste the content, and save the file.

```
cat bob-public.asc
```

Create the new file on the **redmail-client**.

```
nano bob-public.asc
```

Run the following command on the **redmail-client** to import Bob's public key into Alice's keyring.

```
gpg --import bob-public.asc
```

List the keys to confirm that Bob's public key has been successfully imported.

```
gpg --list-keys
```

4.2 Sending a PGP encrypted email

As we already know, **SMTP** only supports text-based messages by default. To carry encrypted content, **MIME** must be used. Let us create a text file named `email.txt` and write the content of the email body in this file.

```
nano email.txt
```

Now let us encrypt the `email.txt` file using Bob's public key. The following command will generate an output file named `email.txt.gpg`.

```
gpg --encrypt --recipient Bob email.txt
```

Now we need to attach this file to an email. Let's encode it using base64.

```
base64 email.txt.gpg > email.txt.gpg.b64
```

Now send the file via email to Bob. **Note:** We are using **SMTPS** to send the file.

```
python3 email-client.py \
--server mail.redmail.com --sender alice@redmail.com --recipient bob@bluemail.com \
--subject "PGP Encrypted Email" --body "" \
--b64 "$(cat email.txt.gpg.b64)" --b64-type application/octet-stream --port 465 --smtps
```

Open the email on the **bluemail-server** and copy the encoded file content from the email to the clipboard.

```
cat /home/bob/mailbox
```

Create a new file on the **bluemail-client** named `email.txt.gpg.b64` and paste the copied content into it.

```
nano email.txt.gpg.b64
```

Decode the email file before decrypting it.

```
base64 -d -i email.txt.gpg.b64 > email.txt.gpg
```

Decrypt the email using Bob's private key.

```
gpg --decrypt email.txt.gpg > email.txt
```

View the content of the decrypted email.

```
cat email.txt
```

Congratulations! You have successfully sent an end-to-end encrypted email using **PGP**.