tasks

March 22, 2022

0.1 Introduction

The goals of this lab are:

- Set up and analyze a firewall.
- Understand symmetric and public-key encryption by setting up and analyzing TLS for secure TCP connections.
- Set up and analyze VPN for secure network tunneling.
- Understand the principles of sender authentication and message integrity by applying and verifying digital signatures.

The lab has several **milestones**. Make sure you reach each one before advancing to the next but try to skim the whole lab at the beginning.

For delivery, submit a PDF report where you answer **only** those steps that are marked with **REPORT**.

0.2 System Setup

In this lab, you will use a system setup as shown in Section **??**. It is similar to lab5, but the servers now have private IP addresses. Your job is to set up a firewall to allow secure access from external networks. Then set up a website with TLS support. Finally, you will setup a VPN to establish a secure network tunnel between two private networks.

Figure 1: Lab6 System Setup

- To copy files from or to containers, use `scp` or the mounted volumes. Read through the "docker-compose.yml" to find out which directories are mounted.
- The OSPF routing protocol is already configured. NAT is also configured in "router1" and "router2".

[ ]:
```
import sys
!{sys.executable} -m pip install cryptography
from test_lab6 import TestLab6
check_progress = TestLab6()
```

1 Milestone 1 – Firewall

In lab5, for simplicity, we assigned public IP address to the webserver, which makes it available to any external network (i.e., the IP address is publically routable across the entire internet), and anyone can access it in an unconstrained manner. But this is wrong!

We should place our resources in an internal network (Section **??**) and should carefully scrutinize any access from an external network through a firewall.

1.1 1.1 Port Forwarding

The first step in building the firewall is to allow access to certain services residing on a private (protected) network. We can use port forwarding to remap the public IP and port to that service's private IP address and port.

- In **"pclient2"**, verify that you can **not** access the webserver (neither through the private IP of "webserver" nor through the public IP or "router0"):

```
wget http://10.20.30.3 --header "Host: www.ttm4200.com"
wget http://129.168.1.2 --header "Host: www.ttm4200.com"
```

- In **"router0"**, set up port forwarding to redirect any HTTP or HTTPS requests to the webserver. That means to set up a packet filter to inspect incoming packets: if a packet has a destination port of 80 (HTTP) or 443 (HTTPS), forwarded it to the webserver of the private IP 10.20.30.3:

```
sudo nft add table nat
sudo nft 'add chain nat postrouting { type nat hook postrouting priority 100 ; }'
sudo nft 'add chain nat prerouting { type nat hook prerouting priority -100; }'
sudo nft 'add rule nat prerouting ip daddr 129.168.1.2 tcp dport { 80, 443 } dnat to 10.20.30.3
sudo nft add rule nat postrouting masquerade
```

  See this tutorial or this tutorial to fully understand the previous commands.

- In **"pclient2"**, verify that you can access the webserver by retrieving the content of its HTML page:

```
wget http://129.168.1.2 --header "Host: www.ttm4200.com"
```

```
[ ]: check_progress.test_1_1()
```

1.2 1.2 Prevent Probing/Scanning a Network

One way to prevent probing/scanning your network is to block ICMP ping:

- In **router0**, drop all incoming ICMP echo requests, using nftables.

  To solve this, you need to understand nftables. These are some suggested source to read more (link1, link2, Link3)

- Then, make sure you can not ping "router0".
- **Optional:** What is the downside of disabling ICMP? >HINT

```
# TA: uncomment the following line to see the solution
# %load ./solutions/nft_1_2
```

```
check_progress.test_1_2()
```

1.3 1.3 Filtering Inbound Traffic

A firewall on a home router allows you to initiate communications from internal devices but restricts any communication initiation from external networks. Your task in this section is to implement this firewall.

- Set up a firewall n **"router1"** that:
  - Allow all connections initiated from internal devices.
  - Allows only established or related connections from external networks
  - Otherwise, drop all incoming connections from external networks. >HINT

```
# TA: uncomment the following line to see the solution
# %load ./solutions/nft_1_3
```

```
check_progress.test_1_3()
```

Q1. **REPORT:** Briefly explain the packet filters above (1.1, 1.2. 1.3). Your explanation should address:

- Whether the packet filter is statefull or stateless, and why?
- Where the packet filter applies in the network stack, and why?

  HINT: There are 5 points in the network stack (aka, hooks), where you can apply a rule of a packet filter:

  -prerouting: inspects all packets entering the system.

  -input: inspects packets delivered to the local system.

  -forward: inspects packets forwarded to a different host.

  - output: inspects packets sent by local processes.

  - postrouting: inspects all packets leaving the system.

2 Milestone 2 – Securing TCP Connection: TLS

In this section, you will set up a secure TCP connection using Transport Layer Security(TLS), commonly confused with its predecessor Secure Sockets Layer (SSL), which is deprecated. Then you will analyze a trace of TLS records to investigate the various records types and the fields in these records.

You will set up a web server that supports TLS (HTTPS: HTTP over TLS). Then investigate how cryptography enhances HTTP with security (confidentiality, data integrity, and end-point authentication).

2.1 2.1 Eavesdropping Insecure Connection

- The "webserver" is set up with *nginx* but unsecured without TLS. Thus any established connection is susceptible to eavesdropping and tampering by an outside party.

To verify that:

- Start packet capturing on **"router3"** (as the outside party, or "Trudy") on **"ether0"** interface. Retrieve the content of the webserver from "pclient2". Stop packet capturing and open the capture file in Wireshark.
- Find the HTTP response (HTTP/1.1 200 OK) sent from the "webserver" to "pclient2". Can you read the response in cleartext (line-based text data)?
- Additionally, you can check that a web server is not secure through a browser, as in Figure 2.

Figure 2: Insecure TCP connection

2.2 2.2 Creating TLS Certificate

The first step in setting up a website with TLS support is to get a digital certificate signed by a certification authority(CA). However, we can create a Self-Signed SSL Certificate using OpenSSL.

> **NOTE:** Using OpenSSL to generate certificates doesn't mean we are using the deprecated SSL protocol. In fact, we can generate *certificates* that can be used by both SSL and TLS *protocols*.

- First, in the **"webserver"**, generate the server's private key with the following command:

```
sudo openssl genrsa -out /etc/ssl/private/ttm4200-selfsigned.key 2048
```

This will generate an RSA key, 2048 bits long. This key is required to sign the certificate. Typically, a certificate is signed by a certificate authority (CA) using CA's private key, which will allow our browsers and other equipment to trust such keys (recall the importance of CAs on page 647 of the book, 7th edition). However, we will use the private key to sign the certificate in our case.

> **Extra:** If you are interested in freely generating a valid worldwide key, you may try Let's Encrypt. However, take care that some of the steps may differ.

- Now, you will generate the TLS certificate. To do it, run the following command and provide the public information that will be embedded in the certificate. Feel free to change the information except for the Common Name, which must correspond to the server's domain name.

```
sudo openssl req -x509 -sha256 -nodes -days 60 -key /etc/ssl/private/ttm4200-selfsigned.key -ou

Country Name (2 letter code) [AU]:NO
State or Province Name (full name) [Some-State]:Trøndelag
Locality Name (eg, city) []:Trondheim
Organization Name (eg, company) [Internet Widgits Pty Ltd]:NTNU
Organizational Unit Name (eg, section) []:IIK
```

```
Common Name (e.g. server FQDN or YOUR name) []:www.ttm4200.com
Email Address []:murad@ttm4200.com
```

More information about the command:

-x509: define the format of public-key certificates. (https://en.wikipedia.org/wiki/X.509)

-sha256: use a stronger hash algorithm

-nodes: skip securing the certificate with a passphrase.

-days 60: specify the validity of the certificate

-key: specify the private key to sign this certificate

-out: specify the path and name of the certificate.

-Emai Address: Use an email address with your team number (e.g., alex@team10A.com). We will check this in the capture file to prove that your team generated the certificate.

- Create a Diffie–Hellman (D-H) Parameter:

```
sudo openssl dhparam -out /etc/ssl/certs/dhparam.pem 2048
```

"Using Diffie–Hellman key exchange will generate a random and unique session key for encryption and decryption that has the additional property of forward secrecy: if the server's private key is disclosed in future, it cannot be used to decrypt the current session, even if the session is intercepted and recorded by a third party." Reference.

2.3 2.3 Configuring the Web Server to Support TLS

The *nginx* service installed in the "webserver" is configured at "/etc/nginx/site-available/ttm4200" to support HTTP. You need to adjust the configuration to use TLS and redirect HTTP requests to HTTPS, thus enforcing the use of the TLS certificate.

- Start by filling the following template. Keep in mind that HTTPS uses a different port number than HTTP.

  HINT

```
server {
    #enable ssl on listening socket (IPv4, default_server)
    listen ====fill in here====;

    #enable ssl on listening socket (IPv6, default_server)
    listen ====fill in here====;

    #location of the server certificate
    ssl_certificate ====fill in here====;

    #location of the private key
    ssl_certificate_key ====fill in here====;
```

```
    #location of the dh parameter
    ssl_dhparam ====fill in here====;

    root /var/www/ttm4200;
    index ttm4200_index.html;
    server_name www.ttm4200.com;
    location / {
       try_files $uri $uri/ =404;
       }
    error_page 500 502 503 504 /50x.html;
    location = /50x.html {
          root /var/www/ttm4200;
       index 50x.html;
       }
}

 # catch-all http requests and redirect them to https
 server{
    listen 80 default_server;
    listen [::]:80 default_server;
    server_name www.ttm4200.com;
    return 301 https://www.ttm4200.com$request_uri;
    }
```

- Check that the configuration's syntax (`sudo nginx -t`) and if so, restart *nginx*.
- In your VM, access the website through a browser. Because we are using a self-signed certificate (i.e., not signed by any trusted CA), the browser will be unable to verify the certificate presented by the webserver. It will issue a warning, as in Figure 3.

Figure 3: Self-signed certificate warning

- Click on *Advanced* → *View Certificate* to display certificate information.

  Note: If you already pressed *Accept the Risk and Continue* you can still view the certificate by clicking on the lock on the address bar.
- **Self-Check:** Why are the "Subject Name" and "Issuer Name" the same?
- **Self-Check:** What algorithm is used for public-key encryption?
- **Self-Check:** What is the signature algorithm?
- **Self-Check:** What hash function is used to produce the certificate fingerprints?

[ ]:
```
# TA: uncomment the following line to see the solution
# %load ./solutions/nginx_tls
```

[ ]:
```
check_progress.test_2_3()
```

2.4 2.4 Analyzing TLS protocol.

To analyze TLS, you need to capture packets in a TLSsession.

- Start packet capturing in **"router3"** on "ether0" interface, and dump the capture to a file named "https_tls.pcap".

```
sudo tcpdump -i ether0 -w https_tls.pcap
```

- In **"pclient2"**, retrieve the content from the "webserver" using HTTPS and skip the certificate verification:

```
wget https://129.168.1.2 --header "Host: www.ttm4200.com" --no-check-certificate
```

- Stop packet capturing in "router3". Copy the capture file to the "lab6" directory, then open it in Wireshark. Apply the display filter `tls` to show only the frames that have TLS records. Then answer the following questions:

  NOTE: Keep in mind that a single Ethernet frame may contain one or more TLS records.

- Locate the first message in the TLS handshake (*ClientHello* record), which the client sends to initiate a session with the server. Then answer the following:
  - What is the packet number of *ClientHello* record?
  - What is the value of the content type?
  - Locate the nonce (also known as a "Random"), consisting of a random number and a Unix timestamp. What is the value of the nonce? How long (in bytes) is the nonce?
  - Locate the cipher suite list, which contains the combinations of cryptographic algorithms supported by the client in order of the client's preference (favorite choice first). How many cipher suites are advertised in the Client Hello record?
  - **Self Check:** Locate the "supported_versions" Extension. Why is the client requesting a TLS 1.2 handshake when it can support TLS 1.3? HINT

[ ]:
```
# What is the packet number of _ClientHello_ record?
ClientHello_packet_number = 7 # your answer (as integer)

# What is the value of the content type?
ClientHello_content_type = 22 # your answer (as integer)

# Locate the nonce (also known as a "Random"), consisting of a random number␣
 ↪and a Unix timestamp. What is the value of the nonce?
ClientHello_nonce_value =␣
 ↪'fb4761d7f426a6faa389d307615502a21984aec62d9313fa4021a007467f892f' # your␣
 ↪answer (as a string of hexadecimal value)
# How long (in bytes) is the nonce?
ClientHello_nonce_length = 32 # your answer (as integer)

# How many cipher suites are advertised in the Client Hello record?
Number_of_cipher_suites =75
```

```
check_progress.test_2_4_1(ClientHello_packet_number, ClientHello_content_type,␣
↪ClientHello_nonce_value , ClientHello_nonce_length, Number_of_cipher_suites)
```

- Locate the *ServerHello* record and expand it. Then answer the following:
  - What is the packet number containing the *ServerHello* record?
  - Which TLS version does the server support?
  - Find the chosen cipher suite (the server-chosen cipher suite from the client's advertised list) and specify the selected algorithms:
    * The symmetric-key algorithm (bulk encryption algorithm) is used to encrypt the sent data.
    * The has algorithm (message authentication code algorithm)
- **Self Check:** Can you locate the *Certificate*, *Server Key Exchange*, and *Client Key Exchange* records? Why?

[ ]:
```
# What is the packet number containing the _ServerHello_ record?
ServerHello_packet_number = 9  # your answer (as integer)

# Which TLS version does the server support?
Server_suppored_TLS_version = 1.3  # your answer (either 1.0, 1.1, 1.2, 1.3)

# Find the chosen cipher suite in the ServerHello_and specify the selected␣
↪algorithms:
symmetric_key_algorithm = 'AES_256_GCM' # your answer (as string)
hash_algorithm = 'SHA384' # your answer (as string)


check_progress.test_2_4_2(ServerHello_packet_number,␣
↪Server_suppored_TLS_version, symmetric_key_algorithm, hash_algorithm)
```

Q2. **REPORT:** Draw a sequence diagram of all TLS **records** exchanged during TLS handshake. Then:

- Explain the functionality of each TLS record.
- Explain how the application data is encrypted.

  NOTE: Keep in mind that a single Ethernet frame may contain one or more records. The question is about TLS **records**, not packets or frames.

3 Milestone 3 – Virtual Private Network (VPN)

To establish a protected network over a public internet connection, we can set up a VPN to encrypt the traffic and allow access to services on private networks. For example, the "Openstack" VMs are only accessible via NTNU private network, and you had to set up a VPN connection to access them.

This lab will use WireGuard, free and open-source VPN software, yet it is faster, simpler, and more efficient than IPsec and OpenVPN. It is already installed in all containers of this lab. You **must** read this page (https://www.wireguard.com/) to get a conceptual overview of WireGuard.

- First, remove port forwarding from **"router0"** to prevent external access to internal resources in "ttm4200_private_net":

```
sudo nft flush ruleset
```

- In **"pclient2"**, verify that you can **not** access the webserver:

```
wget http://129.168.1.2 --header "Host: www.ttm4200.com"
wget http://10.20.30.3 --header "Host: www.ttm4200.com"
```

[ ]:
```
check_progress.test_3_0()
```

3.1 3.1 VPN Setup: WireGuard

In the following steps, you will establish a VPN connection from **"pclient2"** (WireGuard client) to **"router0"** (WireGuard server). Refer back to Section **??**:

- Generate public and private keys for both the sever (**"router0"**) and the client (**"pclient2"**) (i.e., these commands need to be executed in both containers):

```
umask 077
wg genkey | tee privatekey | wg pubkey > publickey
```

More information about the command (https://man7.org/linux/man-pages/man8/wg.8.html):

- umask 077: Set file permission for newly created files to only the user has "read," "write," and "execute" permissions (completely private).

- wg genkey: Generates a random private key in base64.

- | tee privatekey: Read the output of the generated private key and write it to a file named "privatekey" .

- | wg pubkey > publickey: Calculates a public key from the corresponding private key and write it to a file named "publickey"

- In the server (**"router0"**), create a configuration file "/etc/wireguard/wg0.conf", and add the following lines::

```
[Interface]
PrivateKey = <Private key of "router0">
#We will use a VPN subnet of 10.20.30.128/25, use 10.20.30.130 for the sever
#and 10.20.30.131 for the client.
Address = 10.20.30.130/25
#Standard port for WireGuard (optional)
ListenPort = 51820
#Accept every packet on the tunnel interface, and masquerade outgoing
#packets with a public IP
PostUp = iptables -A FORWARD -i wg0 -j ACCEPT
```

```
PostUP = iptables -t nat -A POSTROUTING -o ether1 -j MASQUERADE
#Delete the NAT rule when removing the VPN
PostDown = iptables -D FORWARD -i wg0 -j ACCEPT
PostDown = iptables -t nat -D POSTROUTING -o ether1 -j MASQUERADE

[Peer]
PublicKey = <Public key of "pclinet2">
AllowedIPs = 10.20.30.131
#Send a keepalive packet every 25 seconds
PersistentKeepalive = 25
```

> *HINT: you can run* ***cat privatekey****, then copy the key and paste it into the config file. The same goes for the publickey.*

- In the client (**"plient2"**), create a configuration file "/etc/wireguard/wg0.conf", and add the following lines:

```
[Interface]
PrivateKey = <Private key of "pclient2">
Address = 10.20.30.131/25

[Peer]
PublicKey = <Public key of "router0">
#Public ip address of the server "router0" and ListenPort
Endpoint = 129.168.1.2:51820
# Route all traffic with destination IP address in the subnet
# of (10.20.30.0/24) through this Wireguard tunnel
AllowedIPs = 10.20.30.0/24
```

- Bring up the WireGuard tunnel interface, in **both** the server and the client:

```
sudo wg-quick up /etc/wireguard/wg0.conf
```

- Verify that there is a new interface "wg0" added:

```
ip address
```

- Verify that your VPN is set up correctly by pinging the webserver from **"pclient2"**:

```
ping 10.20.30.3
```

[ ]:
```
check_progress.test_3_1()
```

3.2 3.2 Analyzing VPN

- Start packet capturing on **"router3"** (as the outside party, or "Trudy") on **"ether0"** interface (refer back to Section **??**), and dump the capture to a file named "wireguard_https.pcap":
- Bring down WireGuard tunnel interface, then bring it up to capture the handshake (in **both** the server and the client):

```
sudo wg-quick down /etc/wireguard/wg0.conf
sudo wg-quick up /etc/wireguard/wg0.conf
```

- In **“pclient2”**, retrieve the content from the ”webserver, and then stop packet capturing.

```
wget https://10.20.30.3 --header "Host: www.ttm4200.com" --no-check-certificate
```

Copy the capture file to the “lab6” directory, then open it in Wireshark. Apply the display filter `wg` to show only WireGuard packets. Then answer the following questions:

- Locate the “Handshake Initiation” packet, which corresponds to the first message from Initiator (“pclient2” in our case) to Responder (“router0” in our case). Then answer the following:
  - What is the packet number of the “Handshake Initiation” packet?
  - What is the used transport protocol?
  - What is the destination port?
  - **Self Check:** Does the destination port corresponds to the “ListenPort” in the WireGuard server?
  - How many WireGuard fields are in the “Handshake Initiation” packet?
  - **Self Check:** What are these fields? HINT
- Locate the “Handshake Response” packet, which corresponds to the second message from Responder to Initiator.
  - What is the packet number of the “Handshake Response” packet?
  - How many WireGuard fields are in the “Handshake Response” packet?
  - **Self Check:** What are these fields?
- Locate a “Transport Data” packet, corresponding to the encapsulated data.
  - What is the packet number of the “Transport Data” packet?
  - How many WireGuard fields are in the “Transport Data” packet?
  - **Self Check:** What are these fields?
- Locate a “Keepalive” message.
  - What is the packet number of a “Keepalive” packet?
  - How many WireGuard fields are in the “Keepalive” packet?
  - **Self Check:** What are these fields?

[ ]:
```
################  Handshake Initiation ########################################
# What the packet number of the "Handshake Initiation" packet?
Handshake_Initiation_packet_number = 12  # your answer (as integer)

#  What is the used transport protocol?
wireguard_transport_protocol= 'UDP' # your answer (as string)

# What is the destination port?
Handshake_Initiation_destination_port= 51820# your answer (as integer)
```

```
# How many WireGuard fields are in the "Handshake Initiation" packet?
Handshake_Initiation_num_fields = 8 # your answer (as integer)


################  Handshake Response ########################################
# What is the packet number of the "Handshake Response" packet?
Handshake_Response_packet_number = 13  # your answer (as integer)

# How many WireGuard fields are in the "Handshake Response" packet?
Handshake_Response_num_fields = 8  # your answer (as integer)


################  Transport Data ###########################################
# What is the packet number of a "Transport Data" packet?
Transport_Data_packet_number = 14  # your answer (as integer)

# How many WireGuard fields are in the "Transport Data" packet?
Transport_Data_num_fields = 5  # your answer (as integer)


################  Keepalive ################################################
# What is the packet number of a "Keepalive" packet?
Keepalive_packet_number = 15  # your answer (as integer)

# How many WireGuard fields are in the "Keepalive" packet?
Keepalive_num_fields = 5

check_progress.test_3_2(Handshake_Initiation_packet_number,
                        wireguard_transport_protocol,
                        Handshake_Initiation_destination_port,
                        Handshake_Initiation_num_fields,
                       Handshake_Response_packet_number,
                       Handshake_Response_num_fields,
                       Transport_Data_packet_number,
                       Transport_Data_num_fields,
                       Keepalive_packet_number,
                       Keepalive_num_fields)
```

Q3. **REPORT:** Which of the following security requirements does WireGuard provide/satisfy in this lab? Briefly explain why?

- Confidentiality?
- Integrity?
- Authentication?
- Forward Secrecy?
- Traffic Type Obfuscation?

Q4. **REPORT:** What do you think would happen if you use the public IP address (i.e., `wget`

`https://129.168.1.2 --header "Host: www.ttm4200.com" --no-check-certificate`) while using VPN? What would happen if you also perform port forwarding on "router0" (as you did in Section **??**)? Briefly explain why? >HINT

Extra Credit: WireGuard supports optional Pre-shared Symmetric key Mode, which provides sufficient protection for post-quantum cryptography. According to the WireGuard protocol description: "If an additional layer of symmetric-key crypto is required (for, say, post-quantum resistance), WireGuard also supports an optional pre-shared key that is mixed into the public key cryptography. When the pre-shared key mode is not in use, the pre-shared key value is assumed to be an all-zero string of 32 bytes." (ref:`https://www.wireguard.com/protocol/`). Your **task** is to reconfigure WireGuard with a pre-shared key (use the same setup as before, but include a pre-shared key). Submit a screenshot of your configuration, a screenshot of the output of `sudo wg show`. Briefly explain how using a Pre-shared key augments WireGuard security in case of quantum computing advances.

Optional If you are interested, see how some of your colleagues used WireGuard in their master thesis Link, Link.

4 Milestone 4 – Digital Signature

In this milestone, you will apply a digital signature to provide document authentication (verifying that a known sender created the document) and integrity (verifying that the document was unaltered in transit). We will use Pretty Good Privacy (PGP) protocol for creating and verifying signatures.

One common and free implementation of PGP is GNU Privacy Guard (GPG), also known as GnuPG. We will use the command-line tool, which can be installed in your VM (no need for `docker` in this milestone):

```
sudo apt-get install gnupg
```

4.1 4.1 Public/Private Keys

- Create a PGP public/private key pair with the following command:

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

Then enter a name (e.g., your team's name), email address (e.g., one of your emails), and a passphrase to protect your private key (remember this passphrase!). By default, it will create an RSA key with 3072 bits. You will use the private key to sign a document, while others can use your public key to verify your signature.

- Export your public key as an ASCII file so you can share it with others:

```
gpg --export --armor youremail > teamName_pubkey.asc
```

- Import and trust others' public keys. A public key belongs to this course at lab6 directory ("lab6/pgp/ttm4200_pubkey.asc"). You should import it, and trust it:

```
gpg --import ttm4200_pubkey.asc
gpg --edit-key ttm4200
trust
5
```

```
y
quit
```

4.2 4.2 Verifying Digital Signature

Q5. **REPORT:** There is a pdf document at "lab6/pgp/verify_me.pdf" that was digitally signed with the private key of ttm4200, where the signature file is at "lab6/pgp/verify_me.pdf.asc". Verify the document's signature, then explain how this verification proves (or does not prove) the document's **integrity** and **authenticity**.

> **Note:** assume the public-key hasn't been tampered with.
>
> **HINT**

4.3 4.3 Creating Digital Signature & Encryption

Q6. **REPORT:** When you finish writing the delivery report, digitally sign your report (**detatched signature**). Then encrypt your report such that only the owner of the private key of ttm4200 can read the encryption. Submit the following files:

- Your report (unencrypted version).
- Your public key.
- The signature file.
- The encrypted report.

[ ]:

[ ]: