Synprobe

Testing

TCP Server Initiated Protocol

Testing for tcp server initiated protocol was done by using ssh server built into kali. SSH runs over TCP and is a server initiated protocol
because it first sends the banner to the client and then excepts response bytes from the client. To start the ssh server on kali, run the following command

```
sudo service ssh start
```

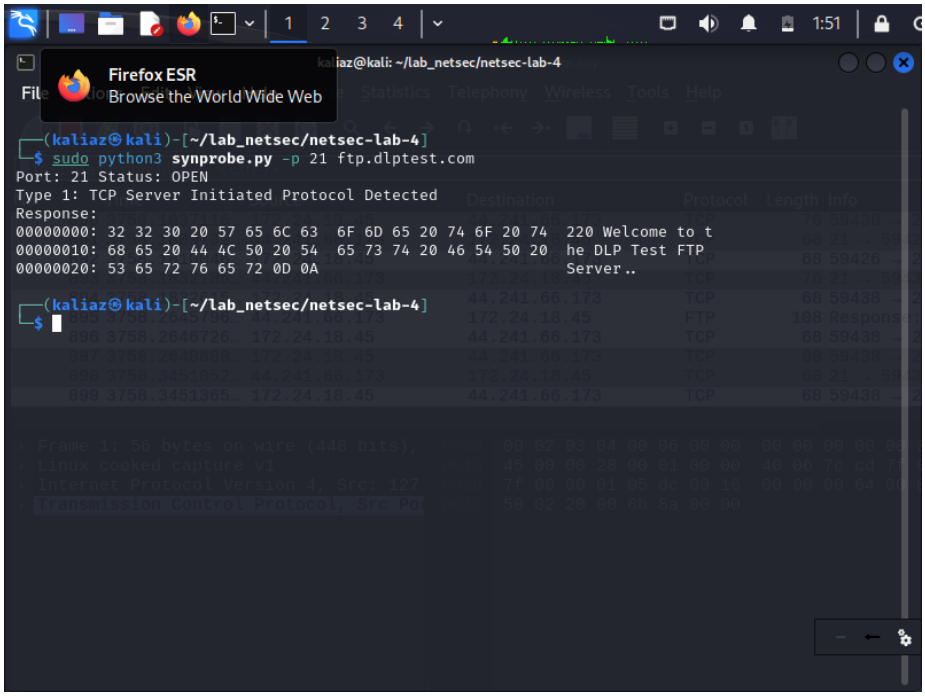

Figure 1: tcp_server_initiated.png

TCP Generic Server

Testing for generic tcp server was done by ncat. To run a TCP ncat server, run the below command

```
ncat -lkv -p 9090
```

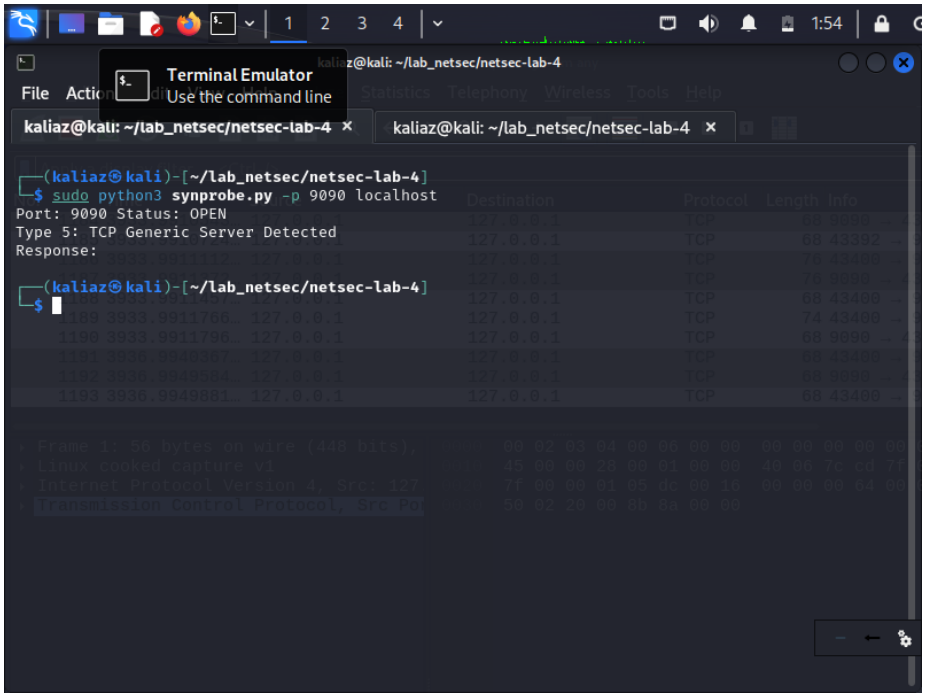

Figure 2: tcp_generic_server.png

TLS Server Initiated Protocol Testing

Testing for TLS connection, where the communication is initiated by the server is done the server **imap.gmail.com:993**. Please note that the aforementioned server uses IMPAS which is the TLS Secured version of IMAP. Sometimes over port 143 one may find an IMAP server with STARTTLS. The STARTTLS method starts with a connection on the standard unencrypted IMAP port (143), and then uses the STARTTLS command to upgrade to
a secure, encrypted connection. STARTTLS is an extension to plain text communication protocols, which offers a way to upgrade to TLS
after the connection is established. Once the STARTTLS command is issued and the server responds positively, the connection becomes encrypted.

An alternative way would be to run a custom tls_server via the script or via ncat

```
# Generate the ssl cert and key required
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365 -nodes
# Run the tls server
python3 tls_server.py
```

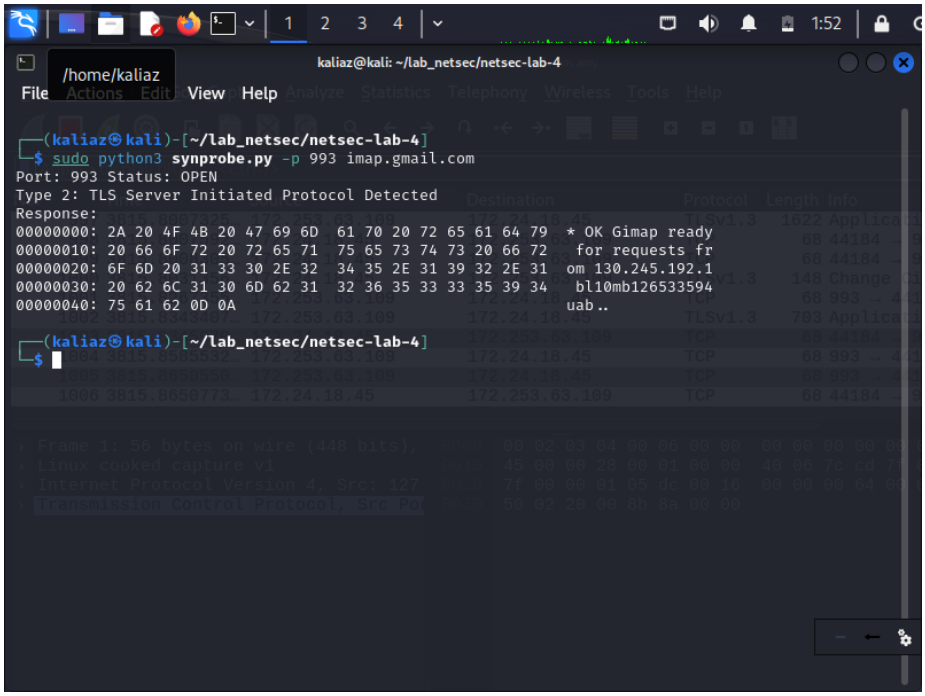

Figure 3: tls_server_initiated.png

Generic TLS Server

To test for a generic TLS server we first create our own cert.pem and key.pem file using the below command.
Remember to replace your ip address in the end of the below command

```
openssl req -newkey rsa:4096 -nodes -keyout key.pem -x509 -days 365 -out cert.pem -subj "/CN
```

Now we have two options, 1. Easier for testing option : Directly trust the cert file by adding it into cacerts list. This option needs to be used if you have a self-signed certificate, because in case of self-signed certificate the client will never know which certificate authroity to use to establish the authenticity of the presented certificate. This lack of trusted path will result in an error like "SSL: CERTIFICATE_VERIFY_FAILED" 2. Production Testing Option : The cert file presented by the server must be issued by a valid Certificate Authority whose certificate is already present in the cacerts list

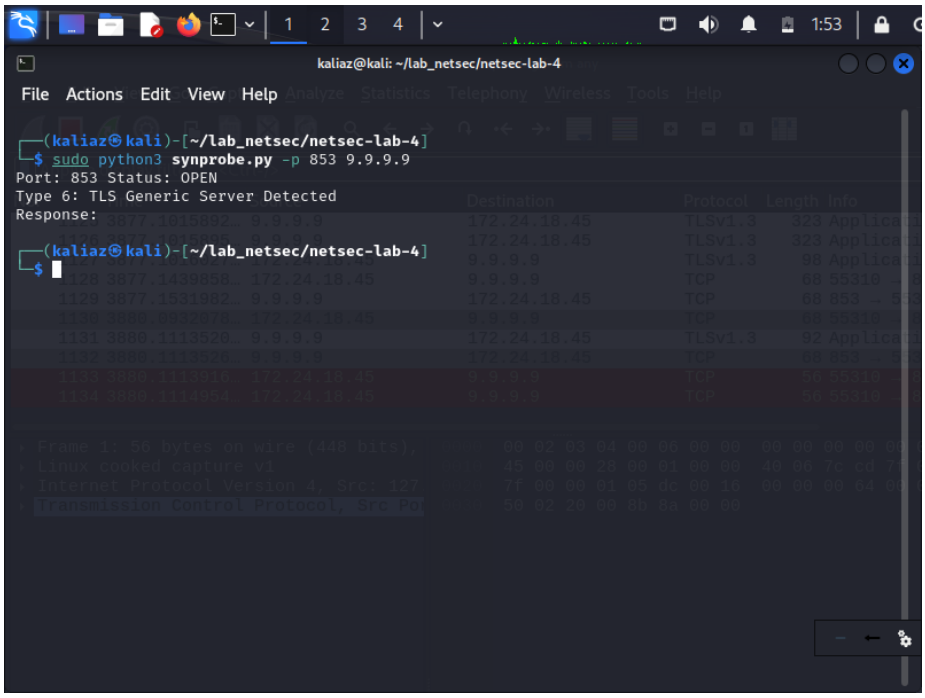

Figure 4: tls_generic_server.png

HTTP Server Testing

Below is an image capturing the testing done for http server

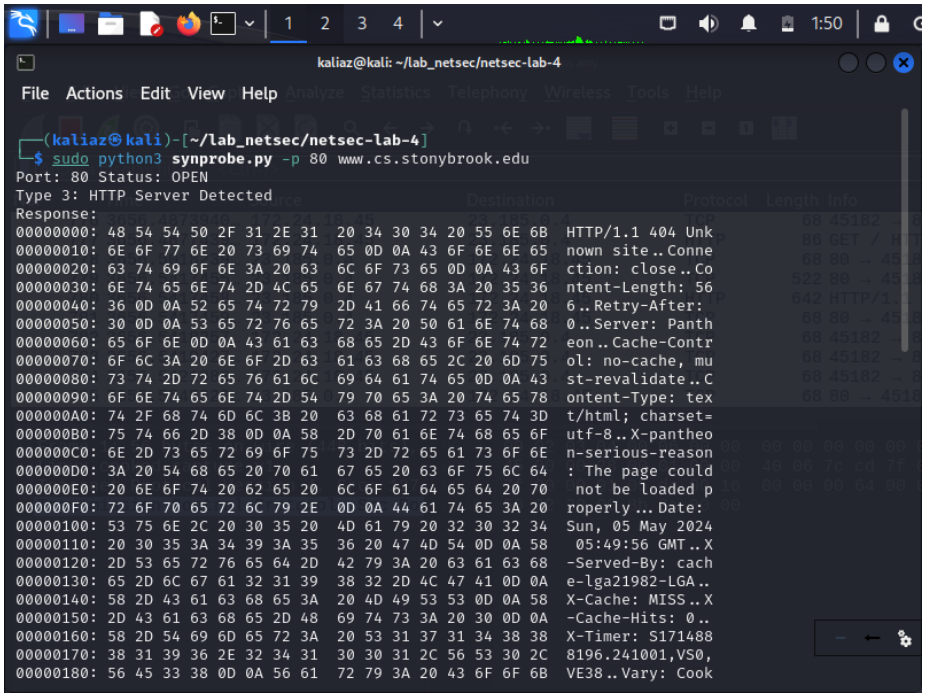

Figure 5: http_server.png

HTTPS Server Testing

Below is an image capturing the testing done for https server

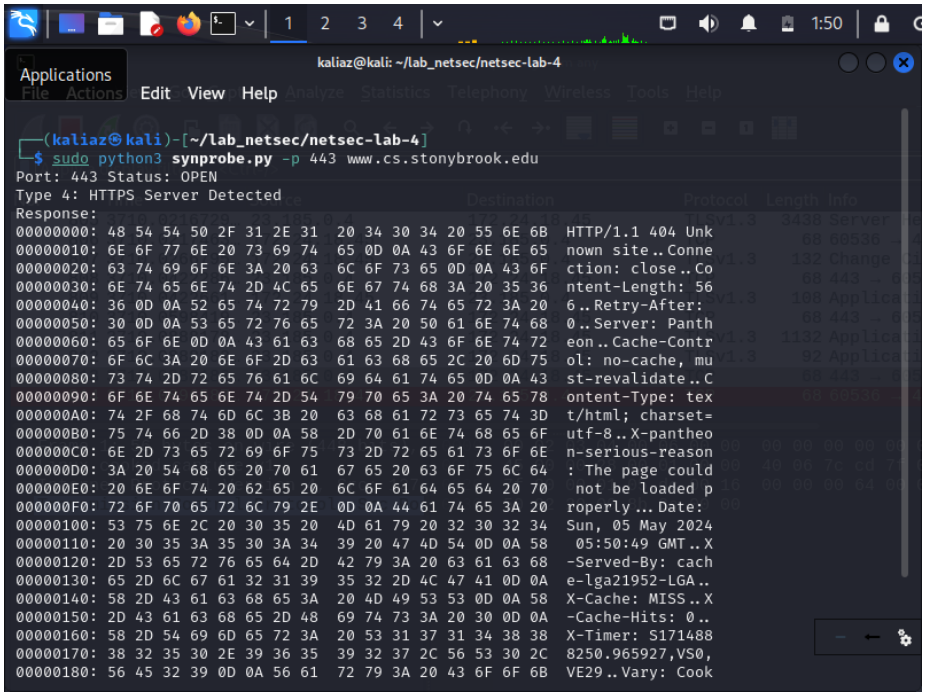

Figure 6: https_server.png

Blocked/Filtered Port

iptables is the firewall for linux. By default, there are no firewall rules set on linux.
To check the firewall rules on linux, use the below command

```
sudo iptables -L
```

To simulate a filtered port or a firewall we need to add a rule to drop all incoming packets on
a port say 9092. To add a rule to iptables to drop all tcp packets on 9092, run the below command

```
sudo iptables -A INPUT -p tcp --dport 9092 -j DROP
```

For enabling closed port or filtererd port detection you must use the self.syn_scanning(target_port)
function that will enable the full syn scanning instead of the current connect scanning.

Debugging Experience

When I ran my synprobe for the first time, the machine on which I ran the synprobe ended up sending
two RST packets instead of the expected 1 RST packet that I was creating. Furthermore, if I sent out any syn packet and recieved a syn-ack packet, the machine would automatically send one RST packet to the machine who sent the RST packet. This effectively meant that I could not complete even a single handshake via scapy. This was happening because of the OS kernel. Scapy bypasses the networks stack of the OS. When scapy sends a SYN packet and the machine recieves a SYN-ACK packet, that SYN-ACK packet is sent to two places. First, it is sent to the OS Kernel, who did not expect this SYN-ACK packet because it was unaware of what scapy is doing. Hence it sends its own RST packet. Second, it reached the synprobe.py python process. Now no matter what the synprobe.py process sends to the target machine, heuristic obeservation is that the RST from the kernel always reached first, hence that connection was terminated half-way. To avoid the kernel sending any information to the target machine, we want to avoid sending any packets sent out by the synprobe.py process to the kernel. To do this, there are OS dependent solutions. #### For MacOS Use pf to drop any incoming packets to port 1500 and 1600 becuase these are the ports that synprobe.py is using.

```
sudo vim /etc/pf.conf
```

Add the below line

block in proto tcp to any port {1500, 1600}

```
sudo pfctl -f /etc/pf.conf # To make pf use the rules that we just defined.
sudo pfctl -e # To enable pf to perform packet filtering in case it is disabled
sudo pfctl -sr # To verify the rules set
```

For Linux Use IP Tables to drop any incoming packets to port 1500 and 1600 becuase these are the ports that synprobe.py is using. Run the below commands on the terminal of your linux machine

```
sudo iptables -A INPUT -p tcp --dport 1500 -j DROP
sudo iptables -A INPUT -p tcp --dport 1600 -j DROP
```