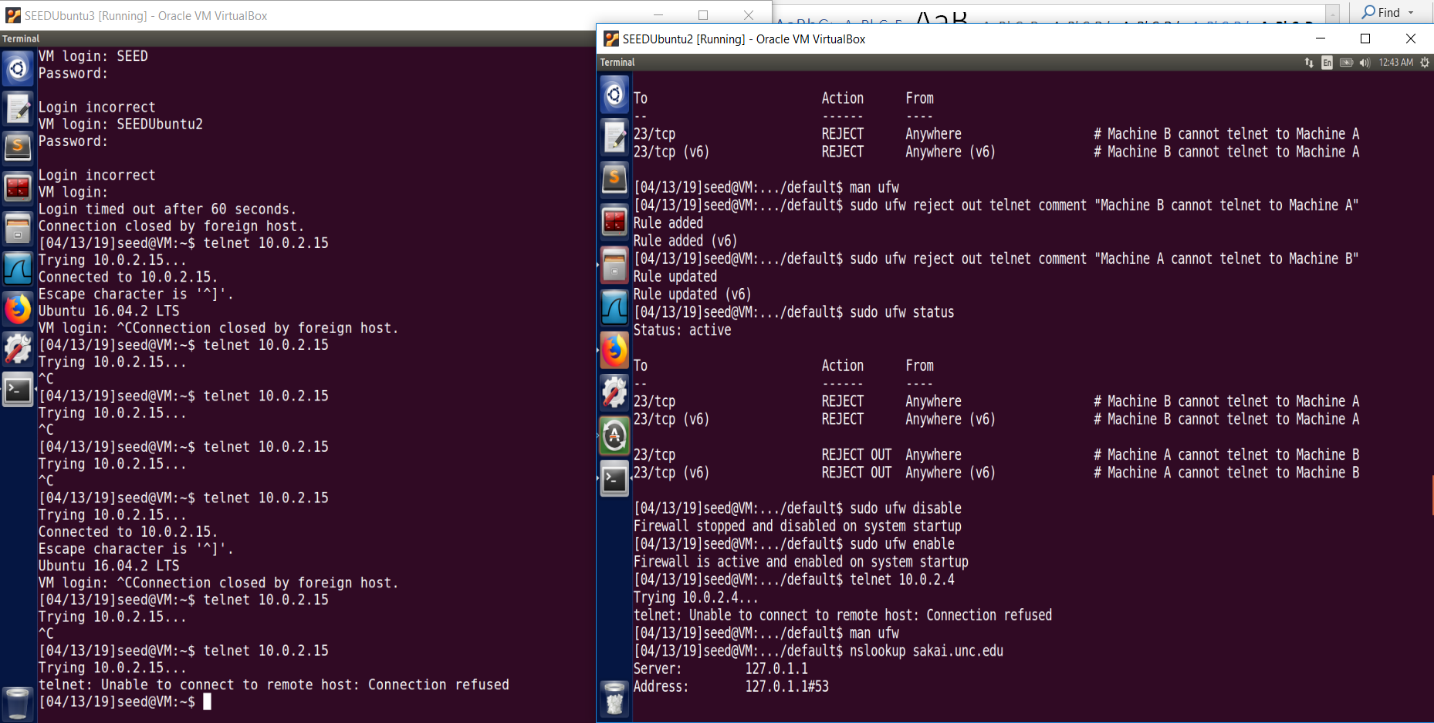
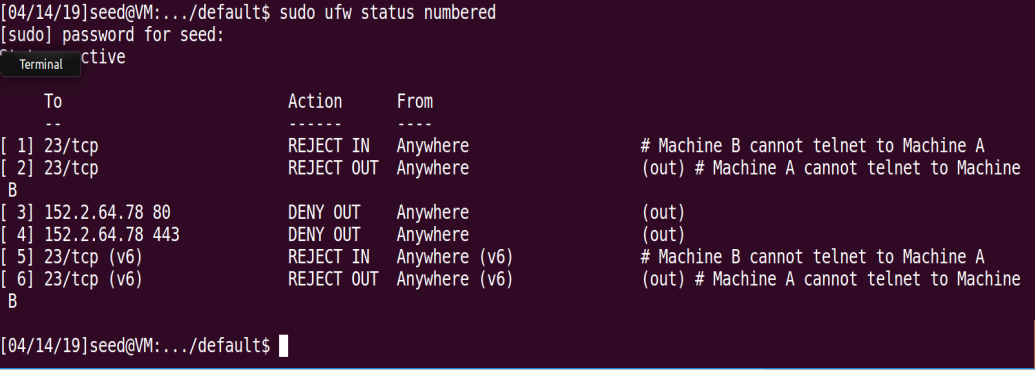
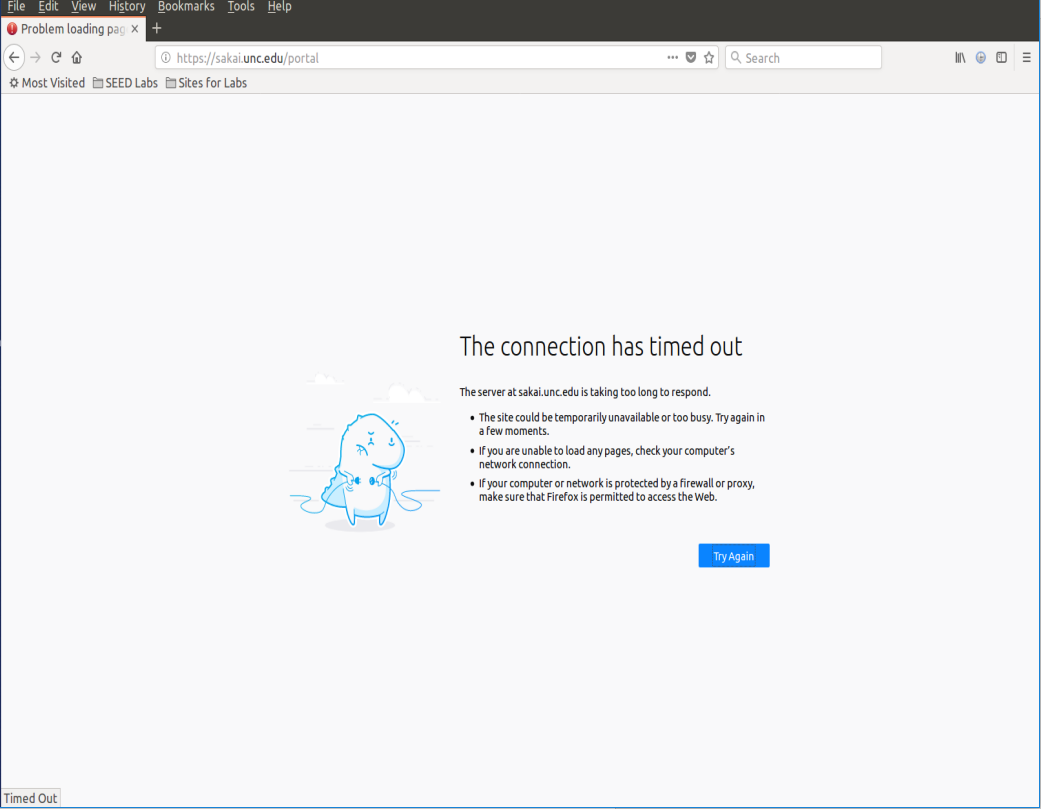
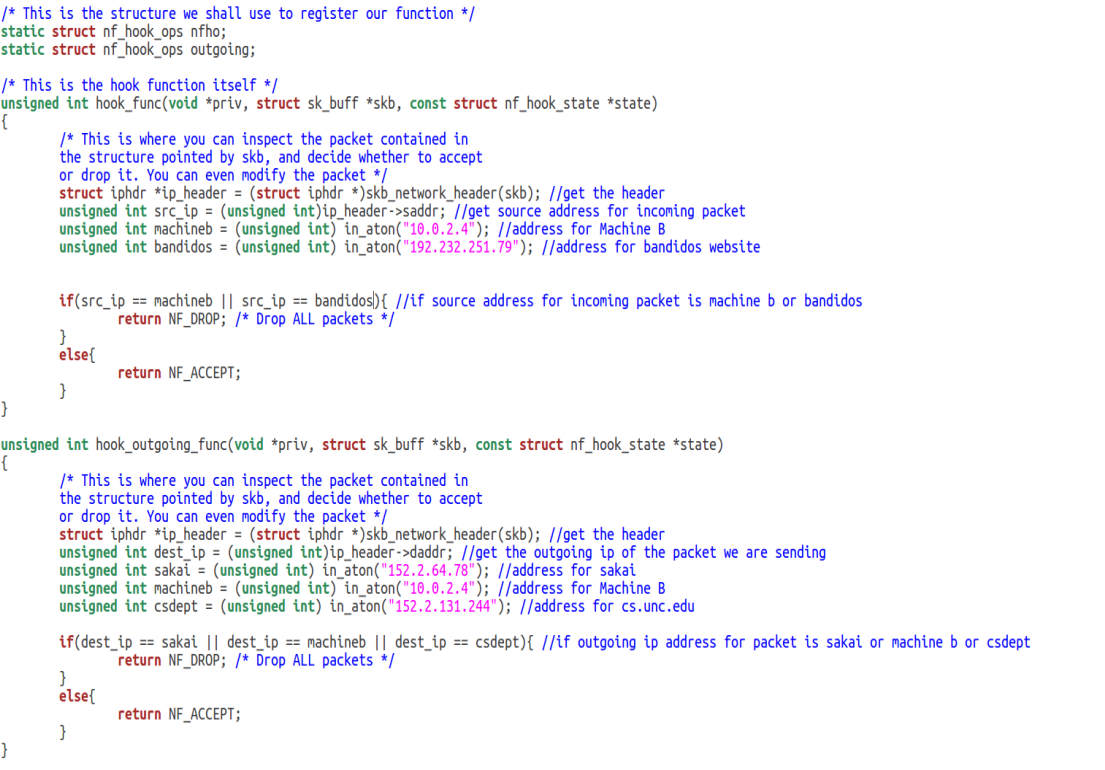
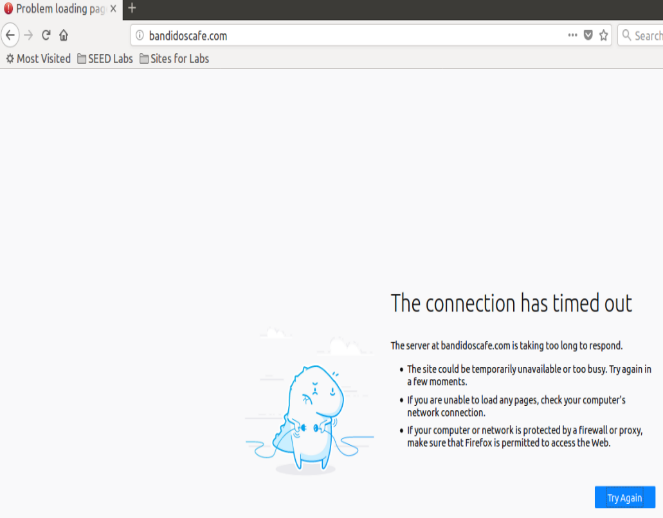
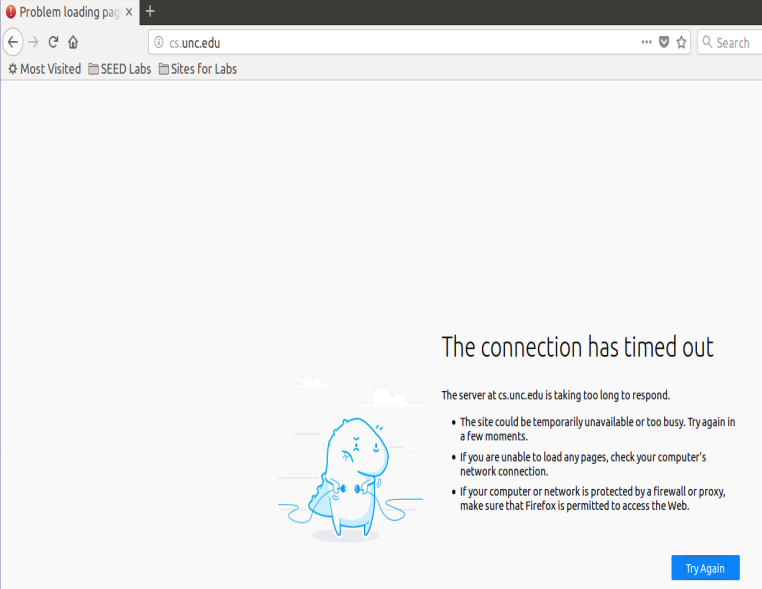
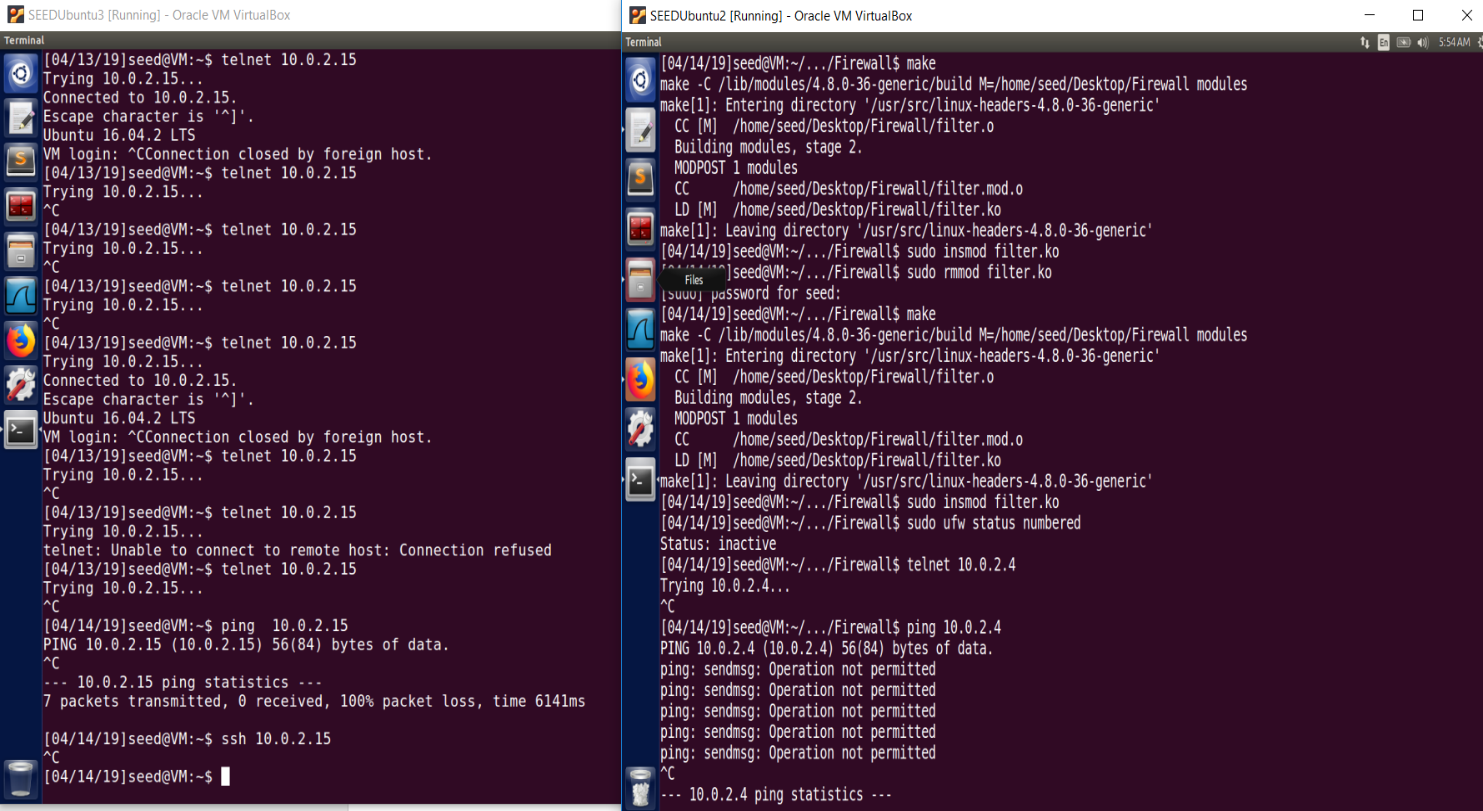
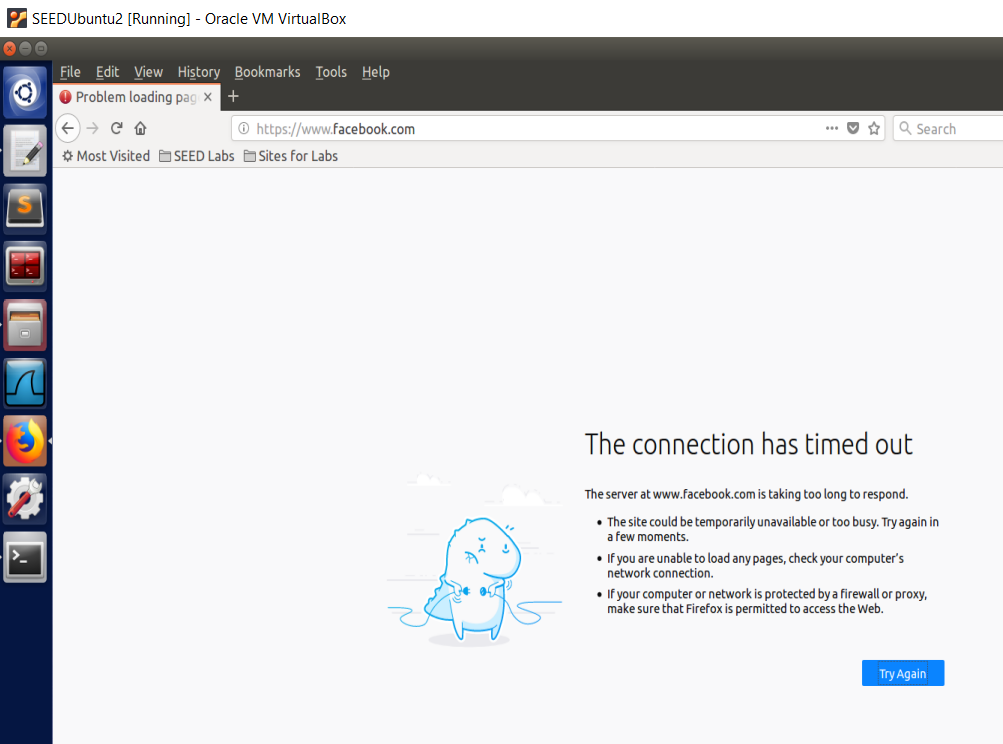
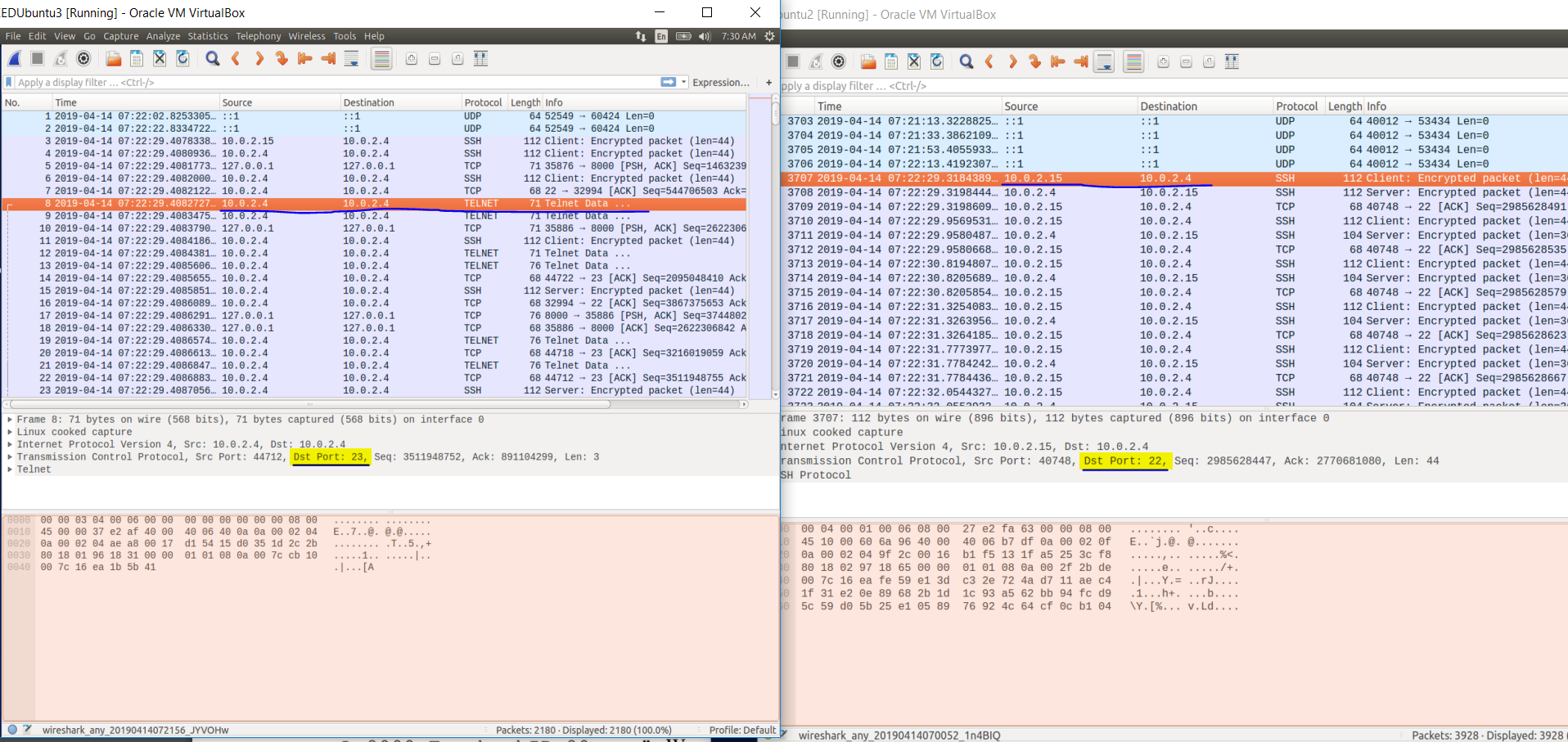
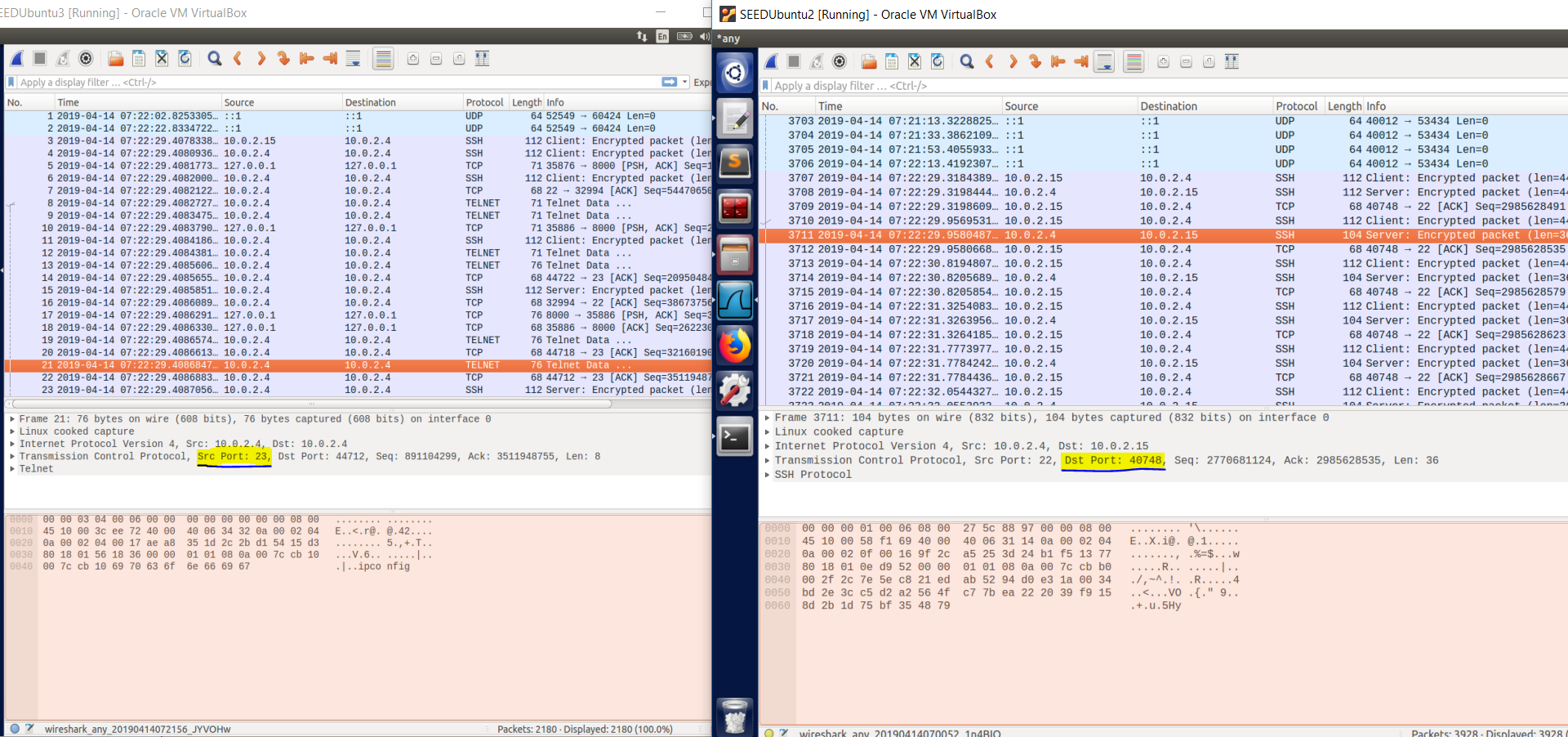
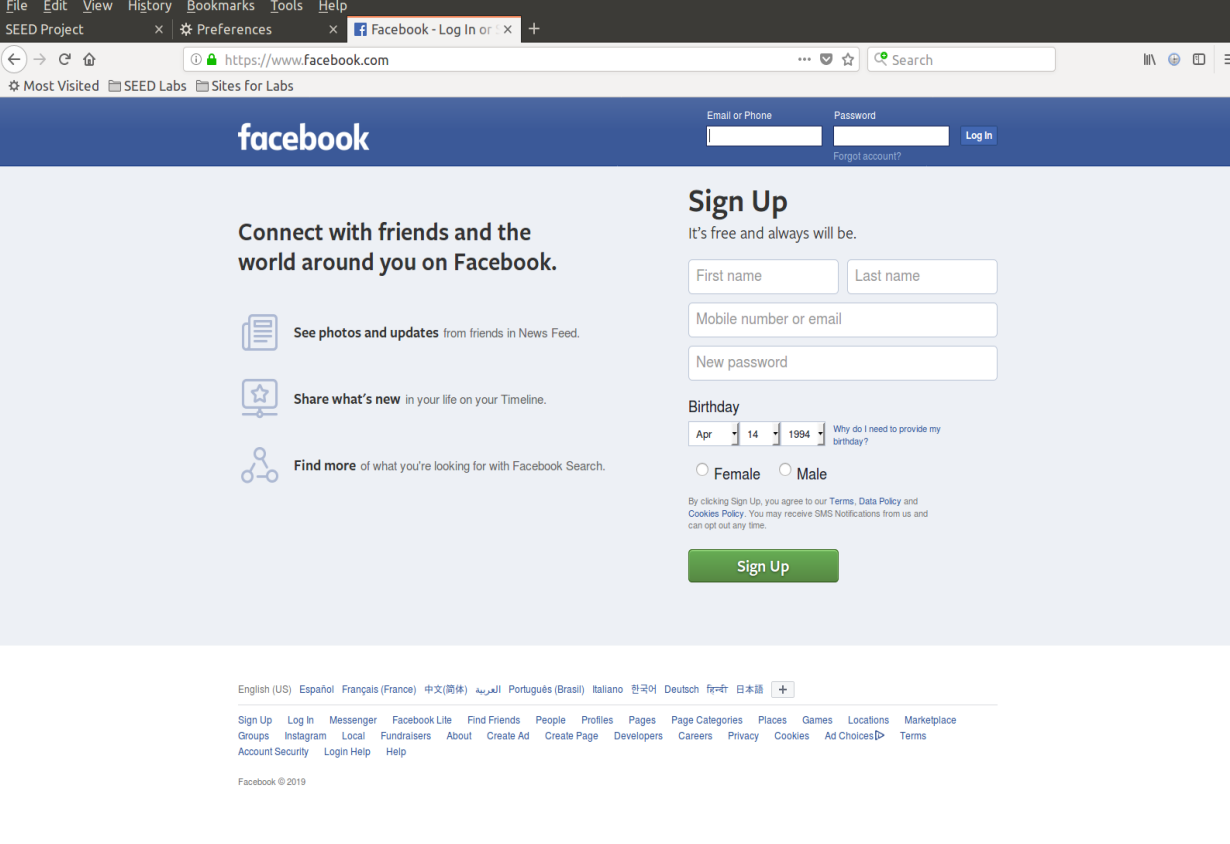
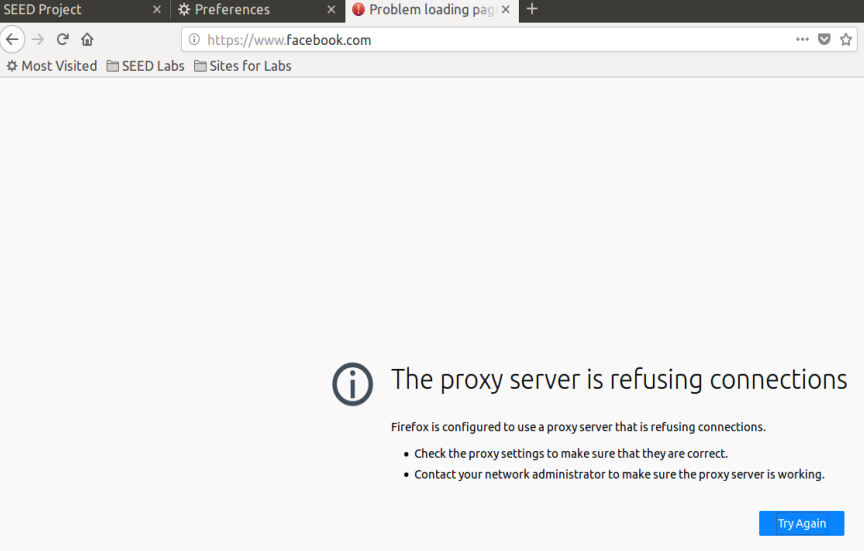
**Linux Firewall Exploration Lab**

* Task #1: Using Firewall
  + First we setup and open two VM’s, one for Machine A and one for Machine B. We modify the firewall on Machine A. To prevent Machine B from being able to telnet to Machine A, we can add a rule to the ufw firewall for telnet connections. To deny all incoming telnet connections, we can add the rule:
  + sudo ufw reject telnet comment “Machine B cannot telnet to Machine A”
  + To deny all outgoing telnet connections, we can add the rule:
  + sudo ufw reject out telnet comment “Machine A cannot telnet to Machine B”
  + After adding both rules to ufw and enabling it as a firewall, no one can telnet to Machine A and Machine A cannot telnet to anyone else.
  + Machine A on the right has IP 10.0.2.15, Machine B on the left has IP 10.0.2.4. Neither can telnet to each other.
  + To prevent Machine A from being able to access an external server, we can add a rule specifying no outgoing connection on port 80 for HTTP or port 443 for HTTPS to the external server. I performed an nslookup on sakai.unc.edu and found the address to be 152.2.64.78. We can add rules blocking connection to this server with:
  + sudo ufw deny out to 152.2.64.78 port 80
  + sudo ufw deny out to 152.2.64.78 port 443
  + Then we can try connecting to sakai.unc.edu and we will not be able to.
  + The rules can be seen in ufw status. The connection is denied when previously it connected with no problems.
* Task #2: Implementing a Simple Firewall
  + First we create a Makefile and rules files with templates given in section 3 of the lab. We can make the module with ruleset of the file we indicate with the make function. The following code contains the makefile and five rules for this simple firewall.

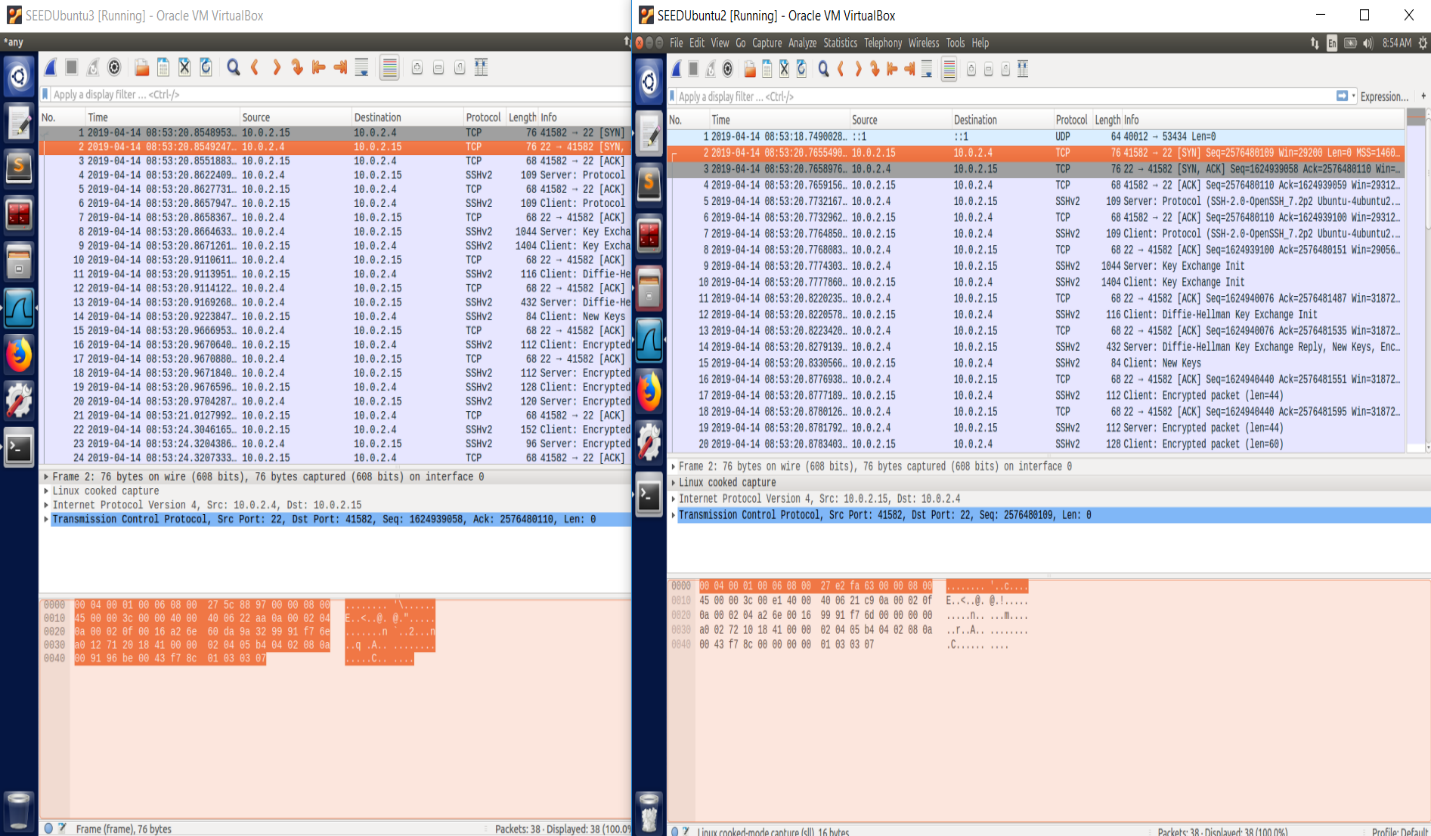


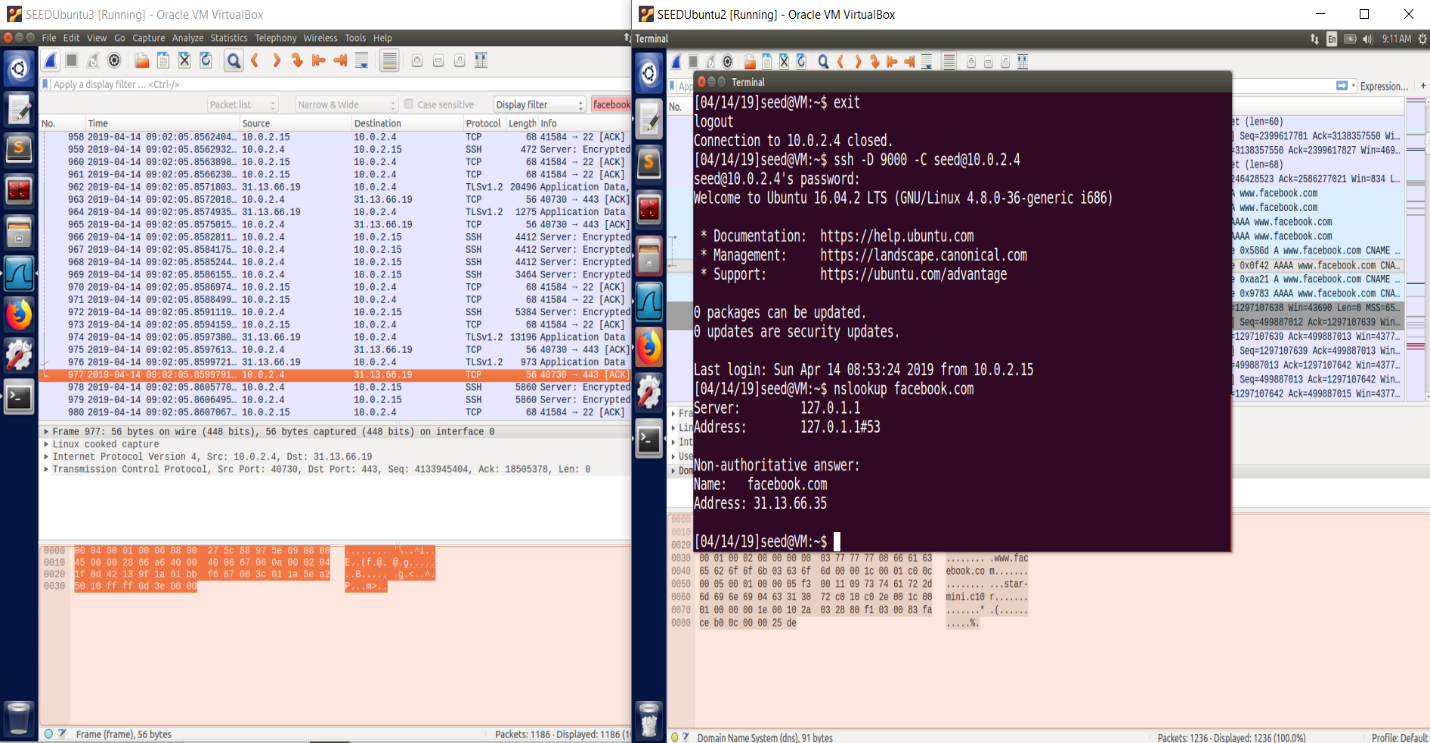
* + Necessary libraries are included. Three of the six rules implemented are the same ones from the first task. To stop Machine A from telnet to specifically Machine B, we get the address for Machine B and check the outgoing packet’s destination address, and if it matches Machine B then we drop it. Same goes for sakai.unc.edu, if the destination address for the packet matches its ip address then the packet gets dropped. This requires a new register hook function which I call hook\_outgoing\_func and setup with requisite traits similar to the given nfho. Then to stop Machine B from telnet to Machine A, we use the hook\_func and in it we also get the header of the packet but this time check the source ip address. Since this is ingress traffic, the source destination is the place this packet is being sent to us from. Thus, if the source IP in the packet header matches Machine B’s address we drop the packet.
  + Another rule I created was to block outgoing traffic to cs.unc.edu, similar to blocking traffic to sakai and Machine B. Another was to block any incoming traffic from bandidos website.
  + This simple firewall drops only the packets that breaks one of our rules, and otherwise accepts all packets, hence the else return nf\_accept code.
  + After checking, I couldn’t telnet or ssh between machines, couldn’t access sakai, cs dept, or bandidos.
* Task #3: Evading Egress Filtering
  + Task 3.a: Telnet to Machine B through the firewall
  + First we are to make a firewall with two rules. No external telnet traffic and no traffic to facebook. I removed my module and used ufw for this task. Enable and then type the commands for the rules:
  + sudo ufw enable
  + sudo ufw deny out 23
  + sudo ufw deny out to 31.13.66.35 port 80
  + sudo ufw deny out to 31.13.66.35 port 443
  + First enable ufw firewall, deny external telnet connections which happen on port 23 by default, deny access to 31.13.66.35 (facebook’s address found from nslookup) on ports 80 and 443 (HTTP and HTTPS).
  + Of course, we now can’t access facebook.com and cannot telnet.
  + The task provides code to create an ssh tunnel to Machine B from localhost port 8000 to Machine B default port 22. Then the code telnet’s to Machine C by which the packets travel in a reverse fashion from Machine C to Machine B by telnet and back to Machine A through the tunnel. Thus we can essentially telnet Machine C.
  + ssh -L 8000:10.0.2.4:23 seed@10.0.2.4
  + IP addresses are the same because I can only reasonably open two VM’s. Now that we have ssh’d into Machine B we can telnet to Machine C via the tunnel.
  + telnet localhost 8000
  + By telnet Machine C, packets travel back in reverse by the process I explained.
  + Now we take an in depth to see this happening on the wire and to explain how egress filtering is bypassed. We do this by using WireShark.
  + Here we have the two virtual machines running wireshark at the time I telnet localhost 8000 after having ssh’d into Machine B. On the right is Machine A (after ssh’d into Machine B), and on the left is Machine B. When I execute the telnet command on Machine A, packets are sent from 10.0.2.15 (Machine A) to 10.0.2.4 (Machine B). This is exactly what we expect from our explanation above. First, the packets travel through the ssh tunnel from Machine A port 40748 to Machine B on port 22. We can verify that it is port 22 on Machine B by looking at the selected packet on the right image. Then we can look to the left image and we’ll see a telnet data packet. This packet is going from source 10.0.2.4 to 10.0.2.4 because I am using the same vm for Machine B and Machine C. But we can look at the source port 44712 and the destination port which is 23. This telnet data produced by our telnet execution on Machine A was sent to Machine B (left image) and in these packets are being forwarded to Machine C (which is using the same ip address) on port 23. To complete the traversal we look a few packets ahead.
  + This time starting from the left image (Machine B) then going to the right (Machine A) we can see the telnet data packet coming back from source port 23 (Machine C) to destination port 44712 (Machine B). Then finally on the right image we have the ssh data packet going through the ssh tunnel from source port 22 (Machine B) back to destination port 40748 (Machine A). We can verify the source and destination ports using the previous images that they are 22, 23, 44712 and 40748.
  + Task 3.b: Connect to Facebook using SSH Tunnel
  + First we ssh on port 9000 to Machine B using:
  + ssh -D 9000 -C seed@10.0.2.4
  + Then we configure firefox to push traffic to localhost on port 9000 so that it goes through the ssh tunnel to Machine B. This is done by going to firefox network settings and using manual proxy configuration for a SOCKSv5 proxy localhost 127.0.0.1 port 9000. Now we can connect to facebook.

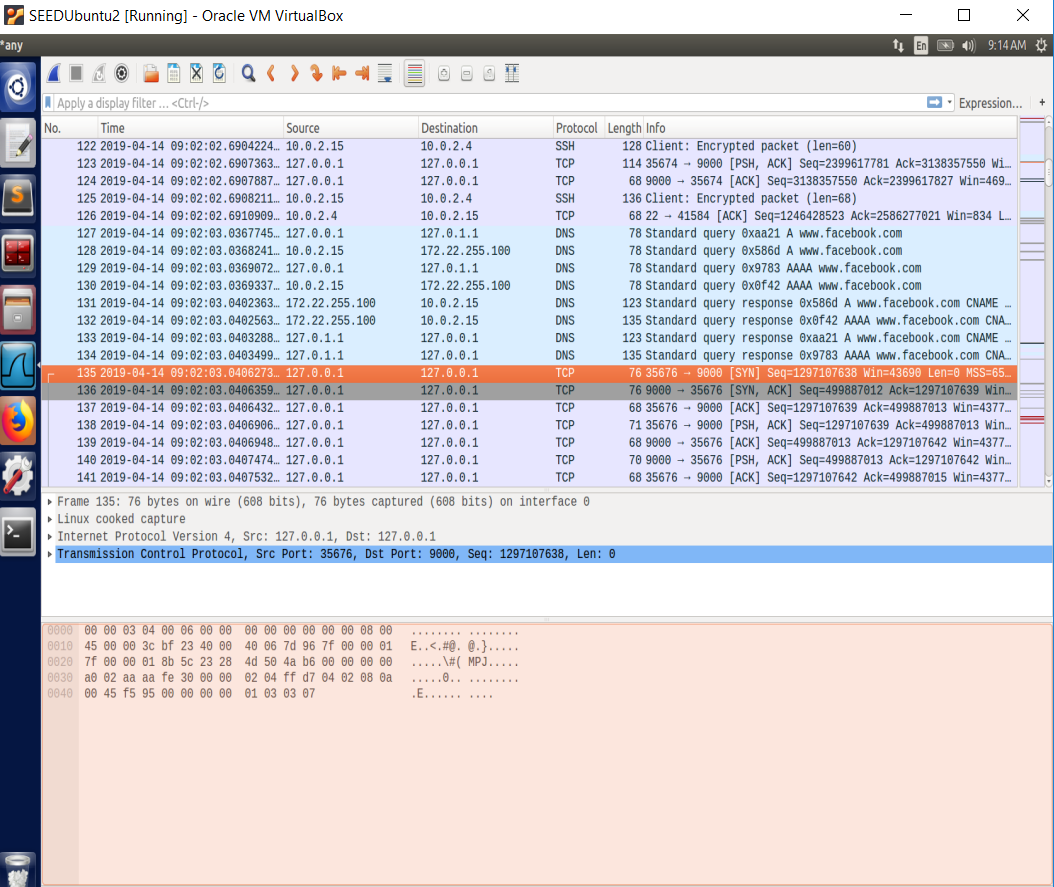
1. I can see the facebook page, we seem to have sent our request through the proxy successfully and received back facebook.
2. After breaking ssh tunnel, clearing cache, reconnecting to facebook.

We receive an error stating that the proxy server is refusing connections. It appears that firefox is still trying to push traffic through port 9000 but there is nothing to connect to.

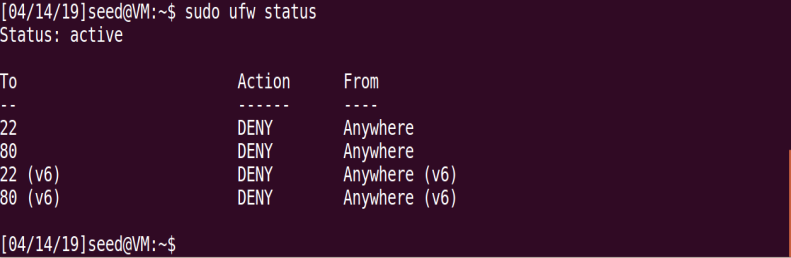
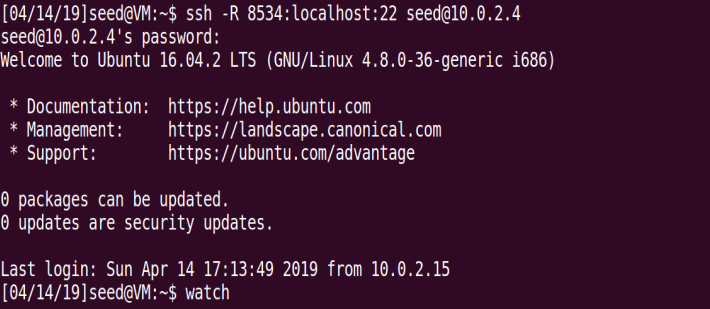
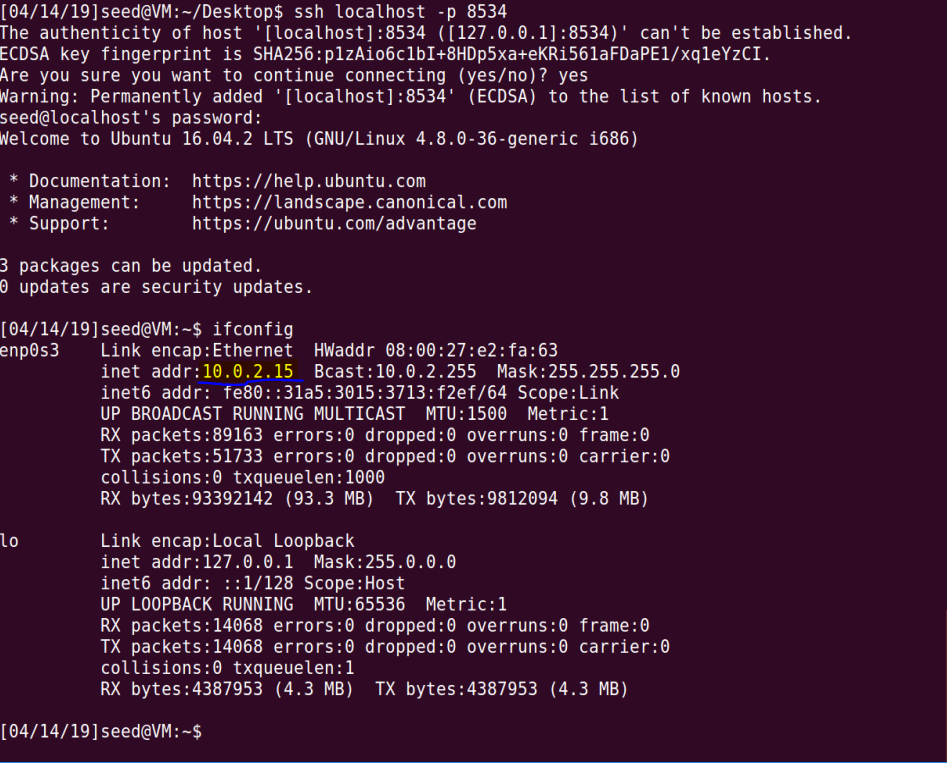
1. We establish the ssh tunnel again and connect to facebook successfully as in step one
2. Now let’s take a look at WireShark as before to see what is happening.

This is as soon as we execute the ssh tunnel command to establish an ssh tunnel from Machine A (on the right) to Machine B (on the left) on port 9000. First, they perform a TCP handshake and send data back and forth with Machine B’s ssh port 22.

Then once we make the request, from Machine B we see SSH packets flowing to and from port 22, all to and from 10.0.2.15 (Machine A). Then also we can see TCP and TLSv1.2 packets to a facebook-like domain space. The TLSv1.2 packets only flow to and from 10.0.2.4 (Machine B) and show that they are using the secure sockets layer application data protocol to send application data back and forth on http-over-tls. This makes sense because Machine A cannot send any packets to facebook, only Machine B can.

Then on Machine A we can see

A DNS lookup for www.facebook.com and following tcp packets destination port 9000. We escape egress filtering when we use an ssh tunnel proxy to Machine B which we can give requests to that B can fulfill but A cannot. The ssh tunnel has all encrypted packets that aren’t being filtered by the packet filter because they are not external telnet connections nor direct facebook connections and port 22 is not blocked, thus we escape the egress filtering we specified on the firewall. The packets sent to and from Machine B at this time are all encrypted as well, including application data. No egress filter is happening because B is fulfilling the requests and sending packets back to Machine A through SSH tunnel.

* Task 4: Evading Ingress Filtering
  + First we setup Machine A to not allow any incoming SSH connections, we do this by
  + sudo ufw deny in 22
  + Port 22 is the port incoming SSH connection will connect to so we deny it and subsequently deny any incoming SSH connections. We also do the same for port 80 (web server). Then our ufw status should look like this.
  + We need to setup a reverse SSH tunnel on Machine A so that we can access its web server from home (Machine B).
  + Since the firewall allows outgoing SSH connections, we can first execute an SSH tunnel on Machine A (10.0.2.15) to Machine B (10.0.2.4). The code
  + ssh -R 8534:localhost:22 seed@10.0.2.4
  + Will ssh to Machine B port 22 from Machine A using an open port I specified 8534.
  + Now we can go home (to Machine B) and SSH from home to work (Machine A) using the SSH tunnel on the port 8534.
  + On Machine B we can run
  + ssh localhost -p 8534
  + We have now setup the reverse tunnel from Machine B to Machine A and have accessed Machine A’s internal web server from Machine B.