Firewall Lab Report

Mudit Vats
mpvats@syr.edu
1/31/2019

Table of Contents

Overview

This lab report presents observations and explanations for the tasks described in the Firewall Lab.

In the screen shot below, you can see three VMs.

VM Background	IP Address
Blue	10.0.2.9
Green	10.0.2.10
Red	10.0.2.11

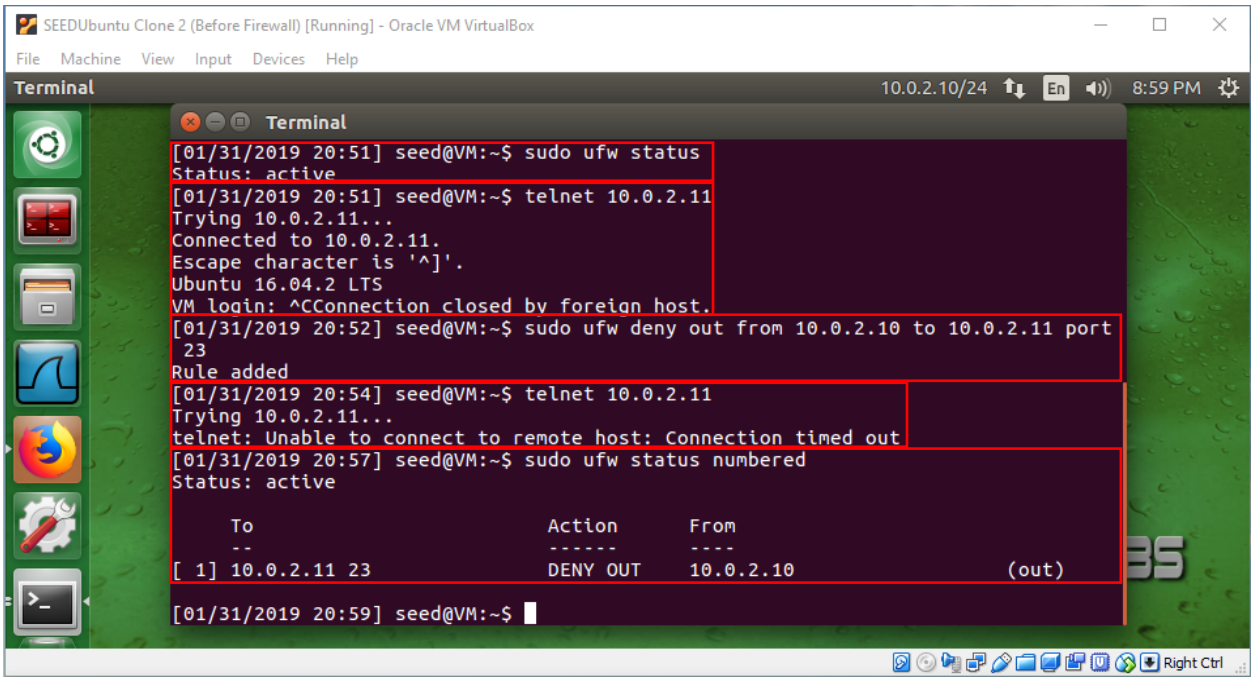
Task 1: Using Firewall

After you set up the two VMs, you should perform the following tasks:

Prevent A from doing telnet to Machine B

In the figure below we see the following sections (red rectangles):

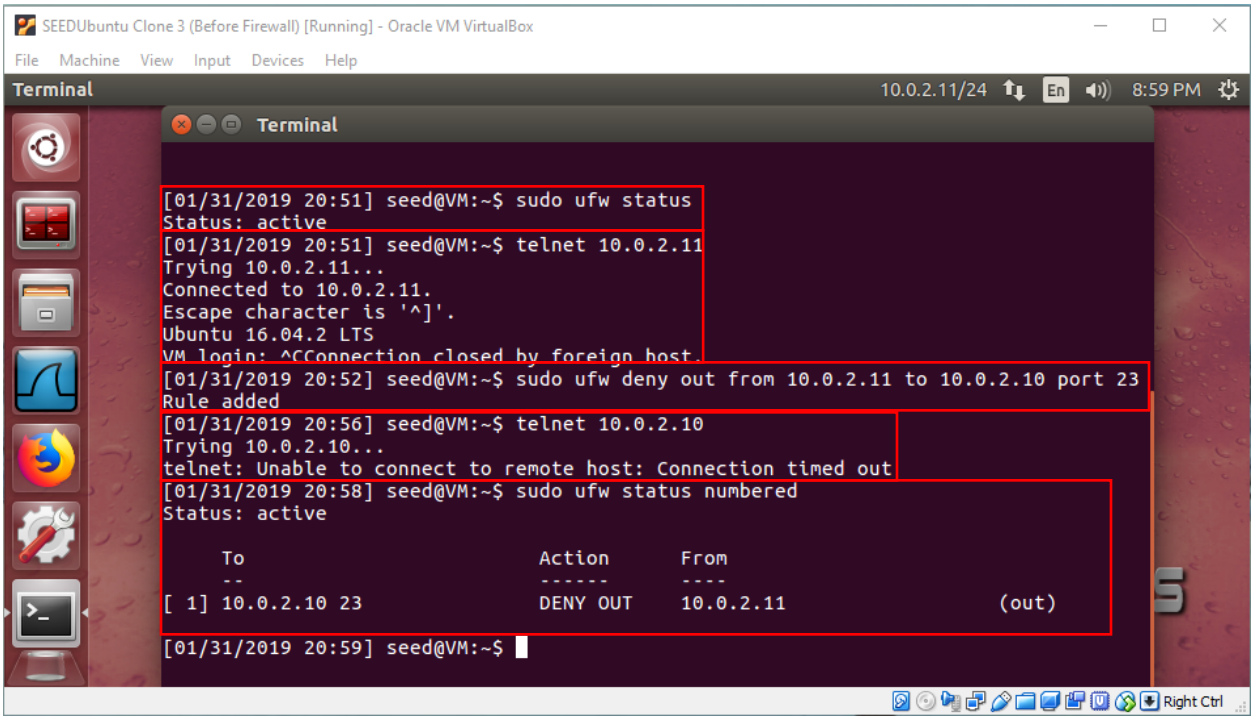
1. Status of the firewall (active) and the fact that there are no rules present since none are listed.
2. Telnet to Red VM 10.0.2.11. This is successful since there are no firewall rules in place to prevent this.
3. The UFW rule which is put in place to prevent outgoing telnet packets (deny from Green VM 10.0.2.10 port 23 to the Red VM 10.0.2.11).
4. Attempted telnet to the Red VM, which fails; i.e. times-out. This happens because we put in the rule (#3) above.
5. Status of the firewall, which shows the same active status and the rule that we previously inserted (#3).



Prevent B from doing telnet to Machine A

In the figure below we see the following sections (red rectangles):

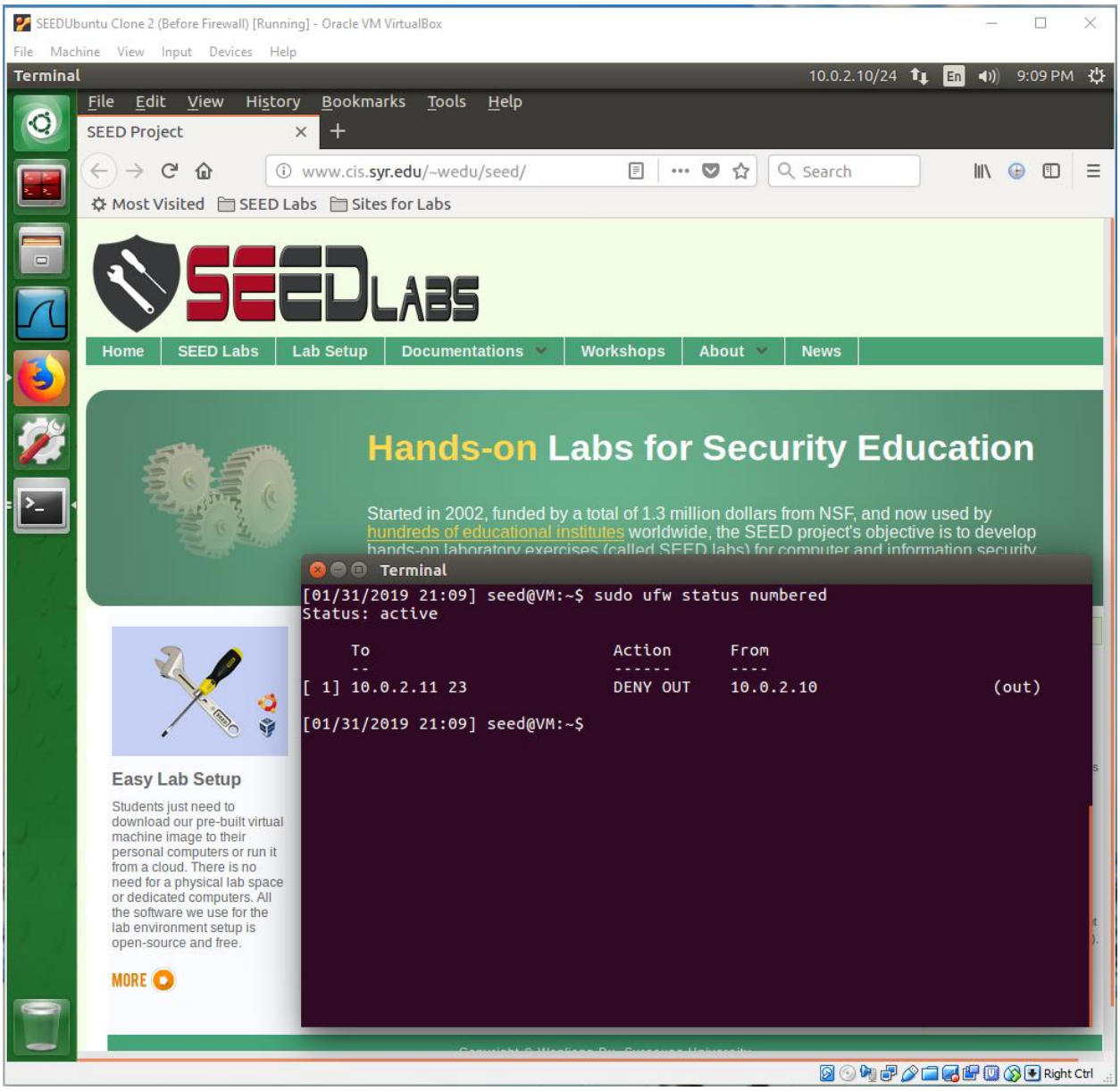
1. Status of the firewall (active) and the fact that there are no rules present since none are listed.
2. Telnet to Green VM 10.0.2.10. This is successful since there are no firewall rules in place to prevent this.
3. The UFW rule which is put in place to prevent outgoing telnet packets (deny from Red VM 10.0.2.11 port 23 to the Green VM 10.0.2.10).
4. Attempted telnet to the Red VM, which fails; i.e. times-out. This happens because we put in the rule (#3) above.
5. Status of the firewall, which shows the same active status and the rule that we previously inserted (#3).



Prevent A from visiting an external web site. You can choose any web site that you like to block, but keep in mind, some web servers have multiple IP addresses.

In the figure below, we see the current status of the firewall. We see that we can successfully reach the SEEDLabs web site at www.cis.syr.edu. This will be the web-site which we block, thus pointing that out.

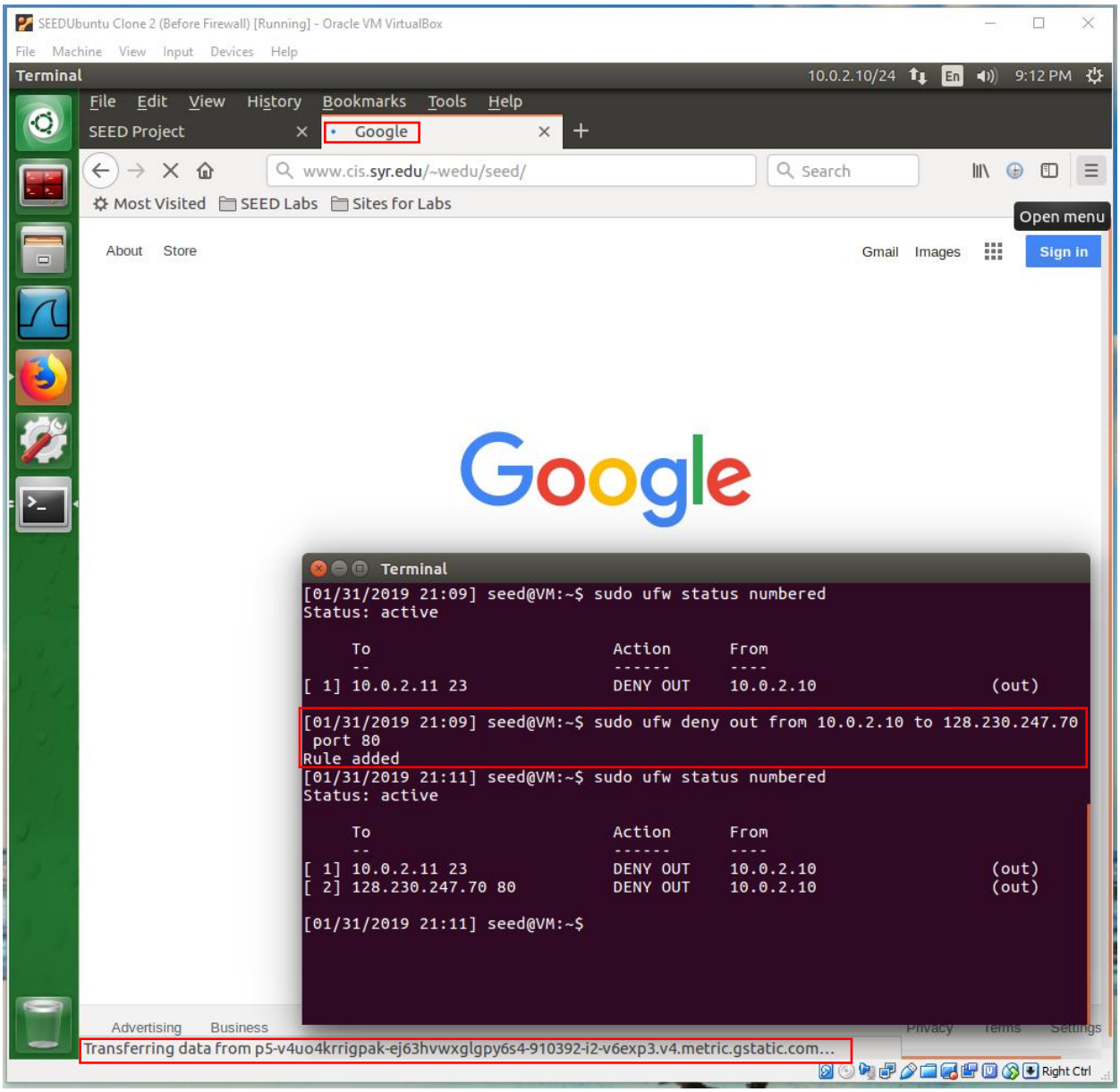
*Please note, we see a rule in place, **which is the rule for a previous exercise**. We will create a new rule for this exercise next so this rule can be ignored!*



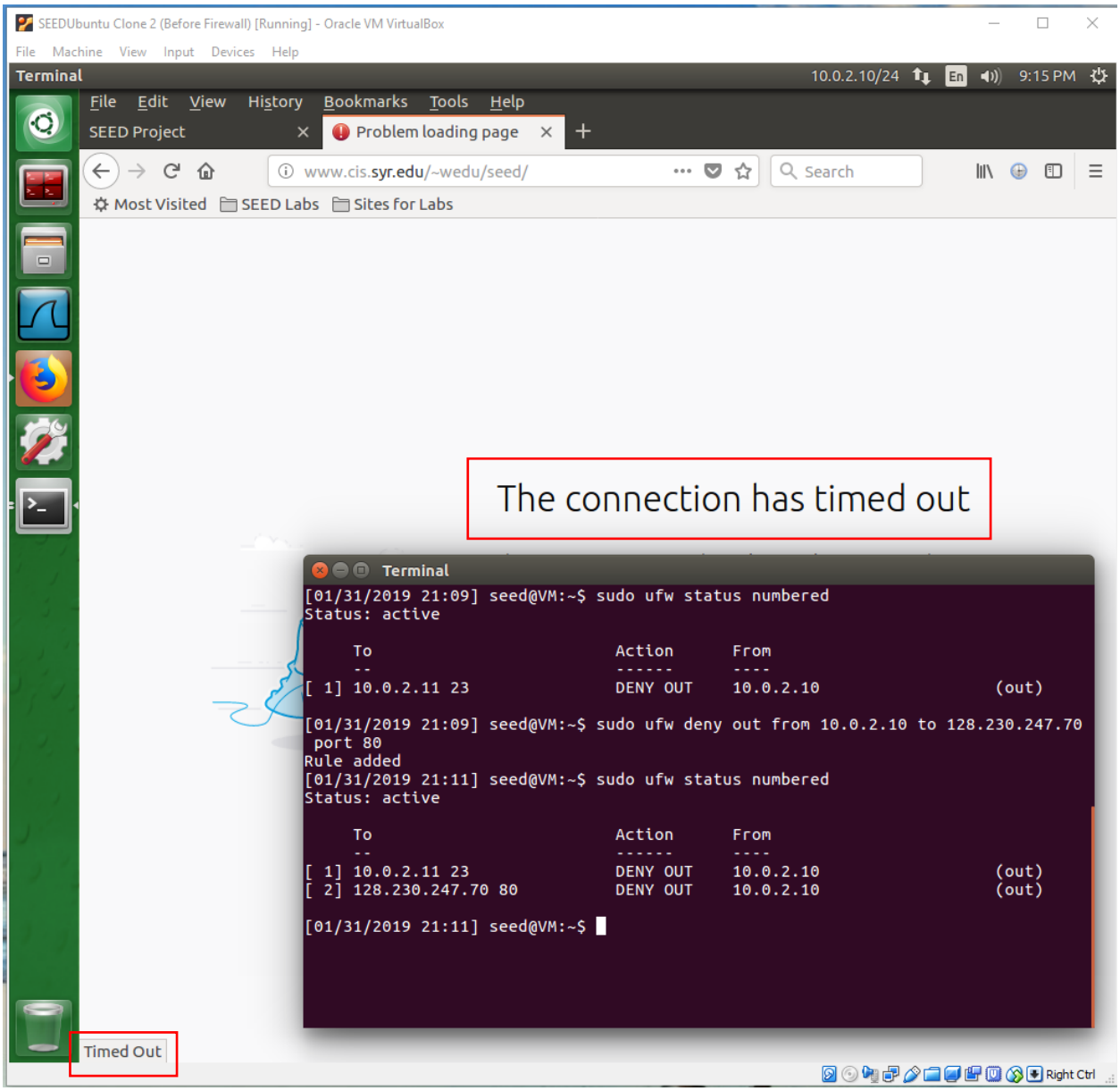
In the next figure, we see the insertion of the rule to deny the Green VM 10.0.2.10 from going out to the IP address of the SEEDLabs site, port 80. This is denoted by the red rectangle in the middle of the Terminal. Also, we use port 80 since the is the port designated for HTTP traffic (and the site is not HTTPS, so we don't use 443).

Also, in the figure you can in the address bar of Firefox which shows that we attempt to go to the SEEDLabs site, but it's not displaying the web-site. Please note that I previously went to Google to "reset" the browser to another site so we can better see whether we can or cannot go to the SEEDLabs web-site.

If you look at the top red rectangle, we can see a dot which is actually a dot that moves around in a circle indicating that it's trying to load the SEEDLabs web-site. We also see the lower red rectangle which shows Firefox attempted to get the web page.



In the figure below, you can see that the connection times-out which means that the browser was unable to go to the SEEDLabs web-site. The reason for this is because we specifically setup a rule to block outbound access for this IP address to access the IP address of the SEEDLabs web-site.



Observations / Explanations

In this series of exercises, we created and tested firewall rules. We used the UFW user-mode utility to create the firewall rules.

In the first two exercises, we created a rule to deny outbound traffic to a particular IP and port. Our rule took the form (using the Green VM as source and Red VM as destination; first exercise):

- sudo ufw deny out from 10.0.2.10 to 10.0.2.11 port 23

Explanation:

1. We use "sudo" since changing firewall rules is a privileged operation.
2. "ufw" is the firewall configuration tool.
3. "deny out" implies that we are denying outbound traffic.

4. "from 10.0.2.10" is our source IP address. This is the system we are preventing from going out.
5. "to 10.0.2.11 port 23" is the destination IP address and port (Telnet) that we are blocking.

In the last exercise, we block a particular web-site. The UFW rule is:

- sudo ufw deny out from 10.0.2.10 to 128.230.247.70 port 80

Explanation:

1. We use "sudo" since changing firewall rules is a privileged operation.
2. "ufw" is the firewall configuration tool.
3. "deny out" implies that we are denying outbound traffic.
4. "from 10.0.2.10" is our source IP address. This is the system we are preventing from going out.
5. "to 128.230.247.70 port 80" is the destination IP address (SEEDLabs web-site IP address) and port (HTTP traffic) that we are blocking.

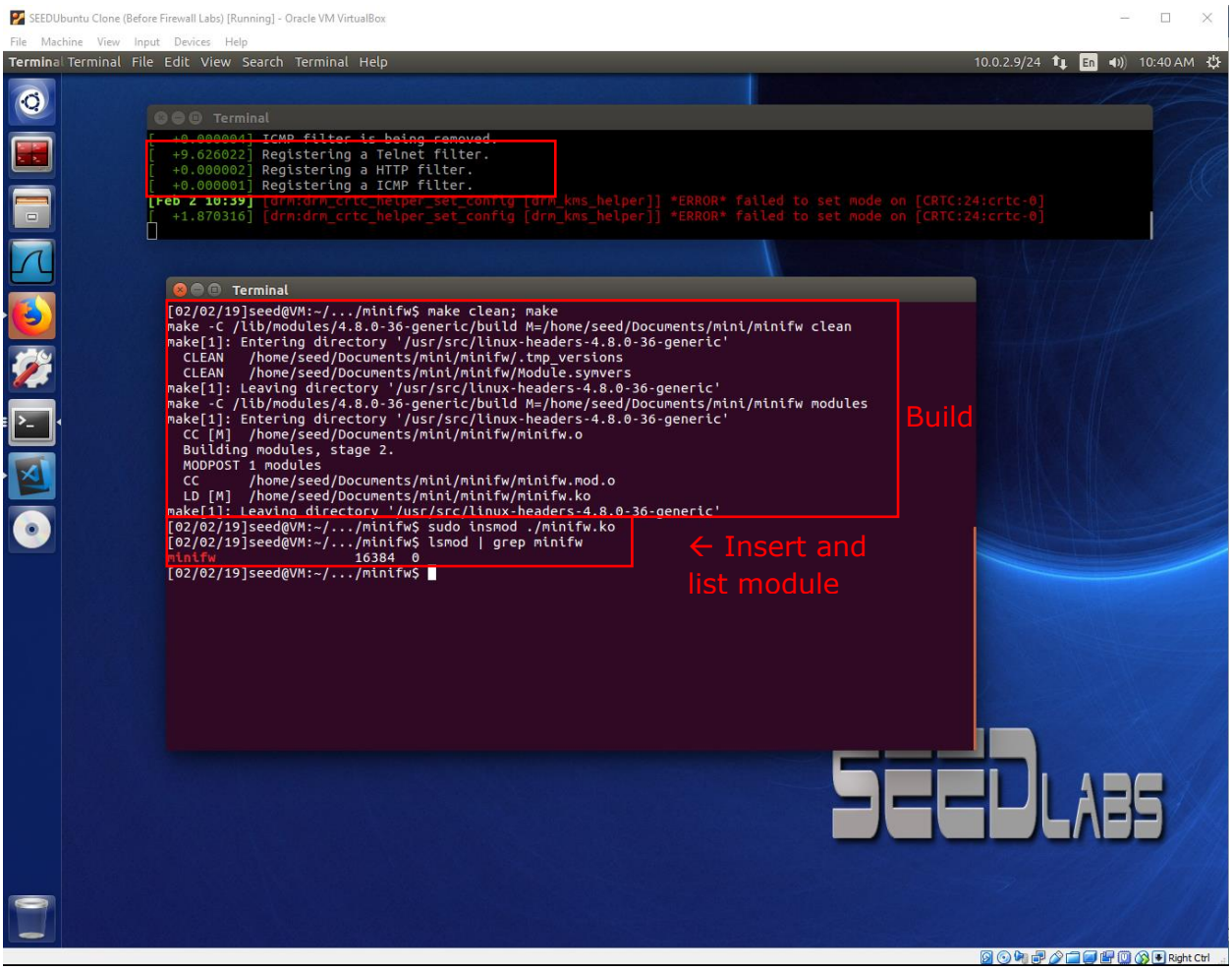
By setting firewall rules using UFW, we were able to control (deny) packets from reaching their destination for Telnet and HTTP. We can use the same methodology to create more interesting rules for our firewall to ALLOW / DENY network traffic. Since the firewall is in the kernel, once the rules are in place, all normal (non-root) users of the system/computer will be "locked down" to the rules that are in place. This can help make the system more secure and prevent access to unsecured services and malicious web-sites.

Task 2: Implementing a Simple Firewall

In this task, you need to use LKM and Netfilter to implement the packet filtering module. To make your life easier, so you can focus on the filtering part, the core of firewalls, we allow you to hardcode your firewall policies in the program. You should support at least two (per instructor) different rules, including the ones specified in the previous task.

For this task, I built the minifw.ko and added three filters. The figure below shows the build and insertion of the minifw.ko module. The three filters implemented are:

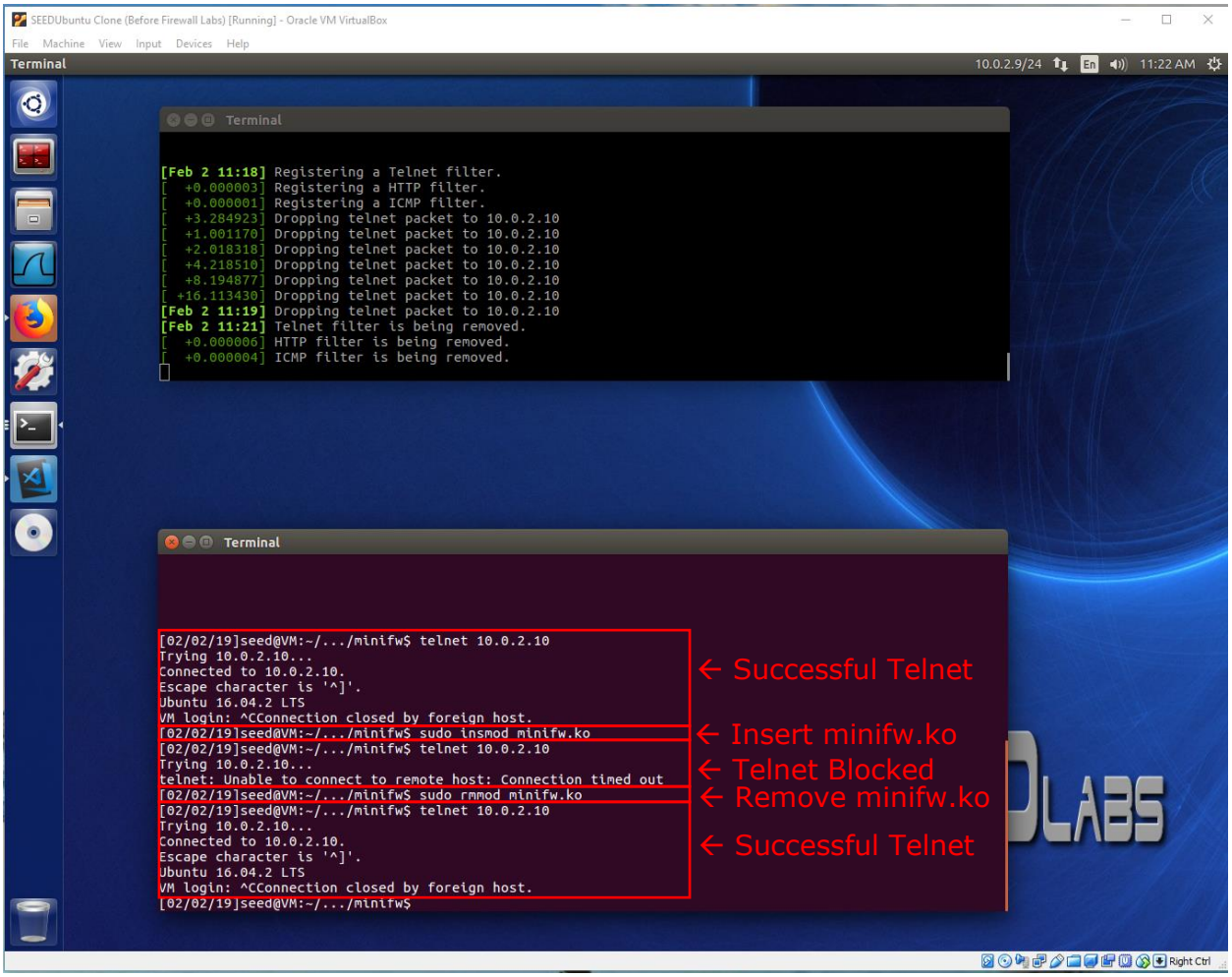
1. Telnet Filter – This filter will **drop** Telnet (port 23) packets destined for the Green VM (10.0.2.10) only.
2. HTTP Filter – This filter will **drop** HTTP (port 80) packets designed for the SEEDLabs website (128.230.247.70) only.
3. ICMP Filter – This filter will modify the ICMP source IP address destined for the Green VM (10.0.2.10) and change the source IP to 1.2.3.4. It will **accept** the packet, so that the modification carry on as it goes out to the network.



Telnet Filtering

For Telnet Filtering, I modified the original filtering code to specifically check for the Green VM's IP address (10.0.2.10). The figure below shows two windows. On the top is the DMESG output to see our printk's from the minifw kernel module. The one below it is our terminal. The steps within the terminal are:

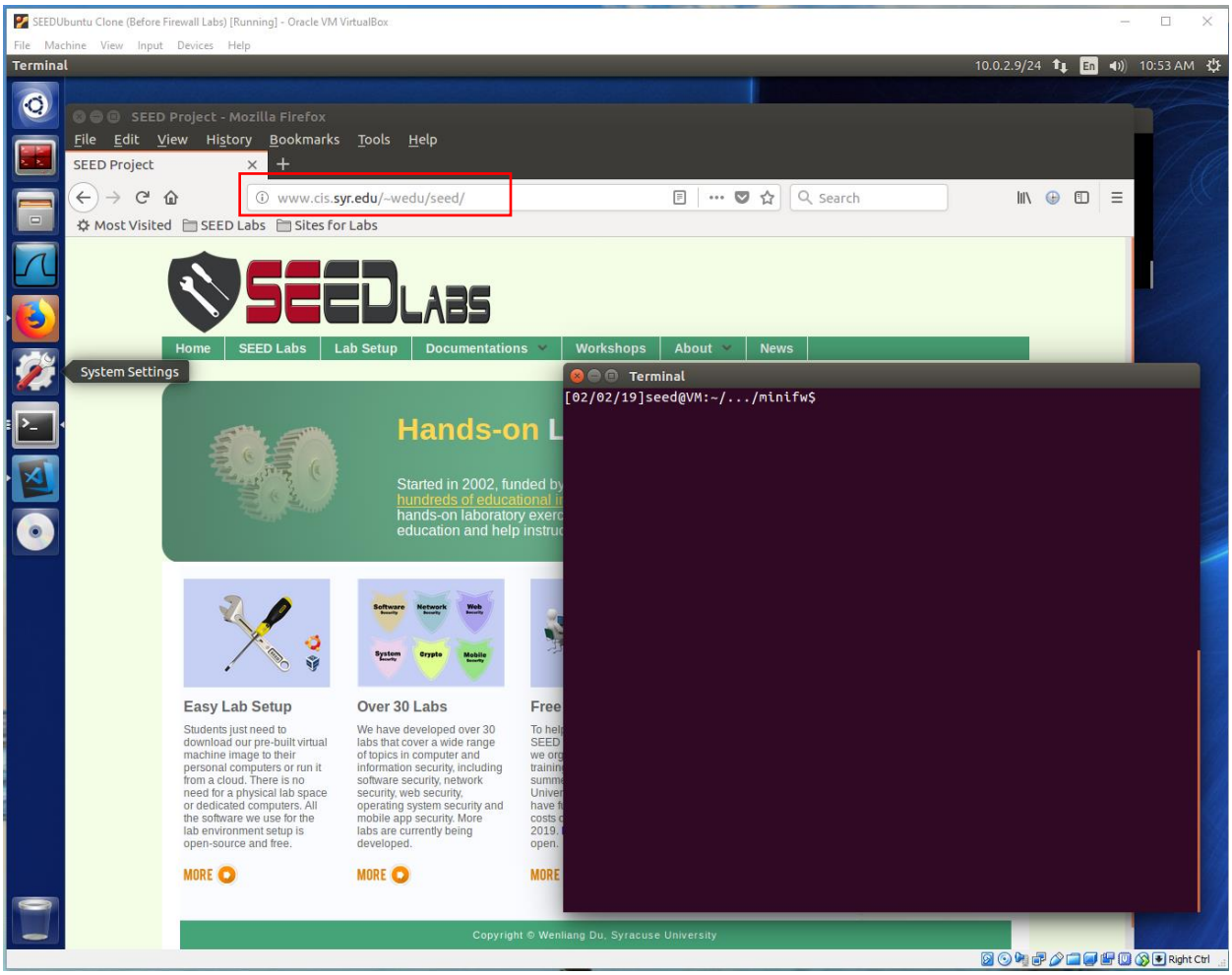
1. Telnet to 10.0.2.10 without the minifw.ko module inserted; i.e. no Netfilter rules inserted. As you can see, the Telnet was successful.
2. Insert the minifw.ko module. This registers the Telnet filter.
3. Telnet to 10.0.2.10. In this case, the Telnet fails. This can also be seen in the DMESG window where it says "Dropping telnet packet to 10.0.2.10".
4. Remove the minifw.ko kernel module.
5. Telnet to 10.0.2.10 is working again.



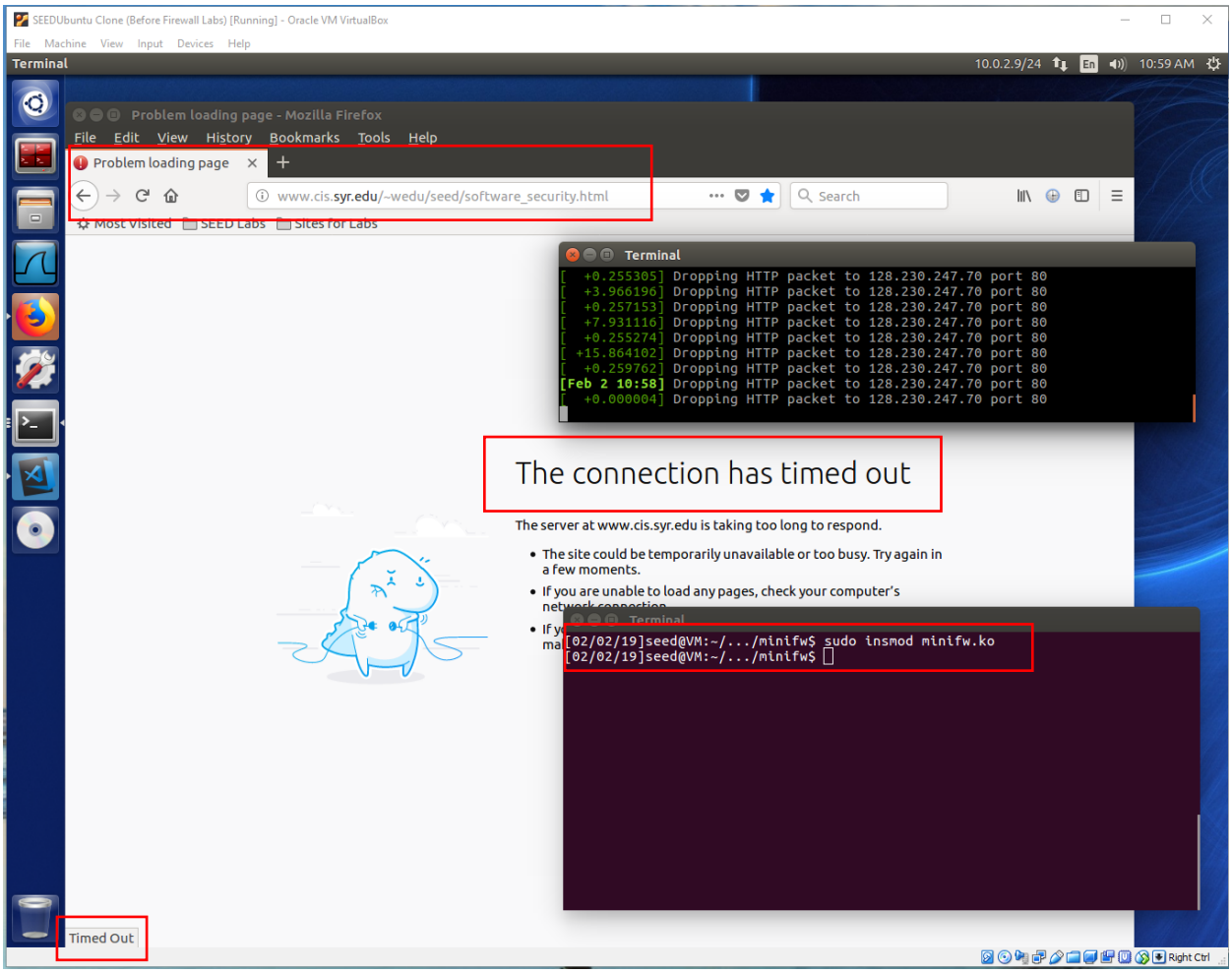
Web-Site Filtering

For Web-Site Filters, I created a new filter which filters for port 80, HTTP, traffic going to the SEEDLabs web-site.

The first figure below shows the SEEDLabs site working. Here, the minifw.ko is not inserted.

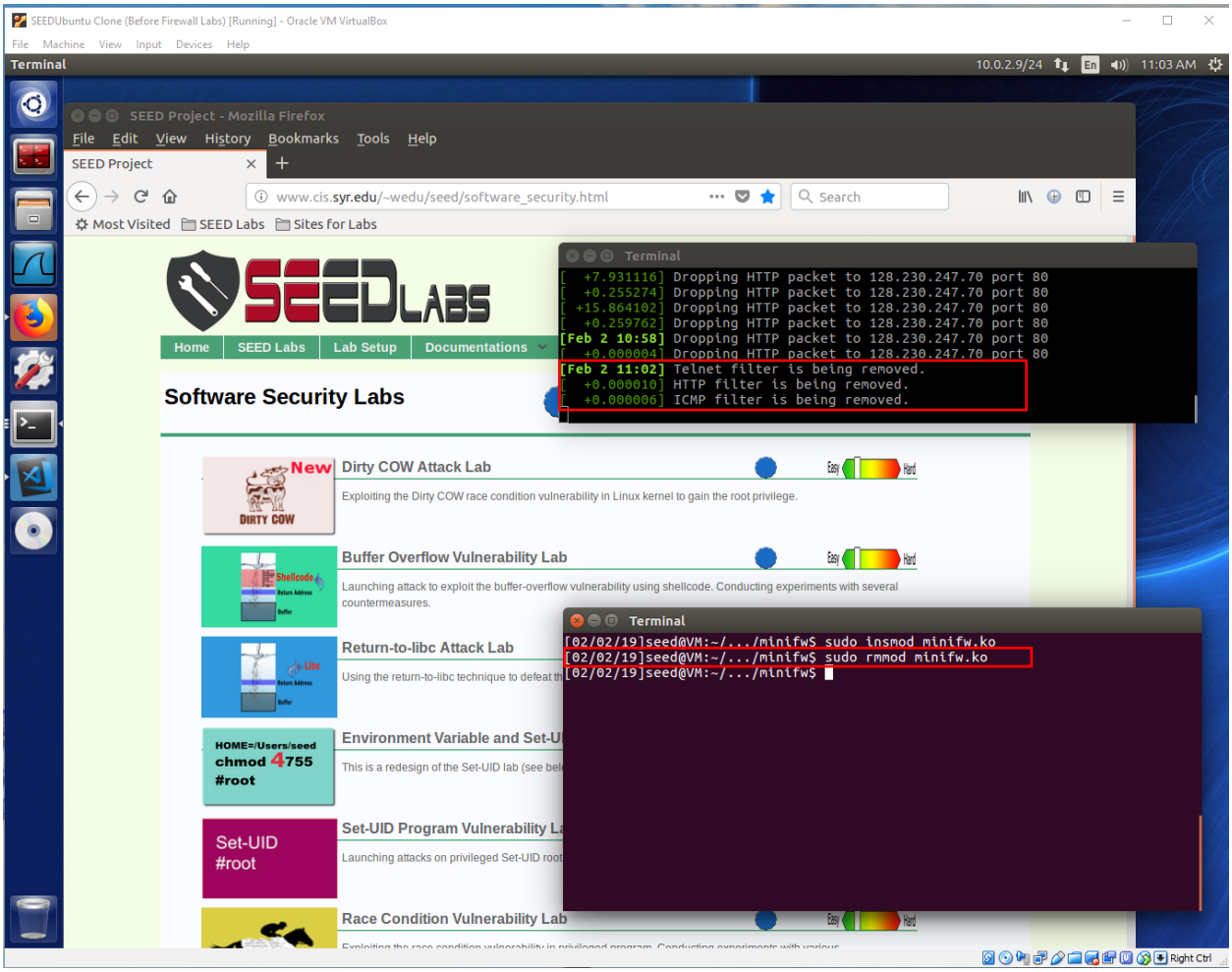


The next figure, shows the insertion of the mififw.ko kernel module and an attempt to access the SEEDLabs web site. As you can see, the web-site times-out. This is shown by the "timed out" web page messages and the Load Page error. It can also be observed in the DMESG window that packets destined for 128.230.247.70 port 80, which is the SEEDLabs site, are being dropped. This is the minifw.ko module who is printing these messages as the packets filtered and dropped.



The final figure below shows the minifw.ko kernel module removed. This can be seen in the terminal window via the "sudo rmmod minifw.ko" and the DMESG messages which show the filters being removed/unregistered.

Finally, we can see that we are now able access the SEEDLabs web-site.



ICMP Filtering

For the ICMP Filtering, I created a filter which looks for ICMP packets that are destined for the Green VM (10.0.2.10) and change the source IP address to 1.2.3.4. I also updated the IP header checksum so that the IP packet is well-formed.

Note: Without updating the checksum, I wasn't getting ICMP Echo Replies. Of course it makes sense that the checksum needs updating, but it took some investigation / debug to get this right. 😊

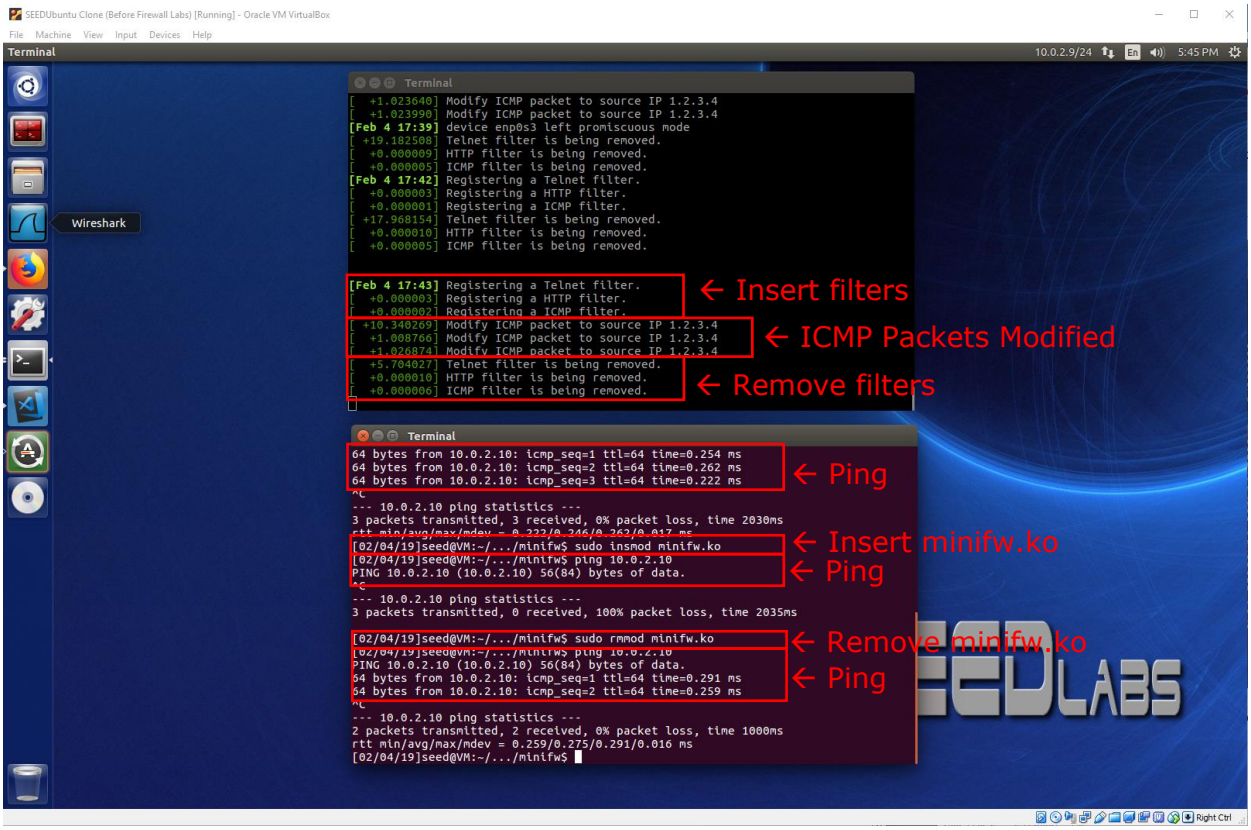
The figure below, shows two windows. On the top is the DMESG window and on the bottom is the Terminal. The Terminal shows:

1. ping 10.0.2.10, which successfully receives responses from 10.0.2.10.
2. insmod of the minifw.ko. The DMESG windows shows the module being inserted and our filters being registered.
3. ping 10.0.2.10. As can be seen, the pings are sent to 10.0.2.10 and in the DMESG we see the message "Modify ICMP packet to source IP 1.2.3.4" which means that the ICMP request that was sent had its source IP address changed to 1.2.3.4. We do not get a response that is viewable via ping since 1.2.3.4 is not

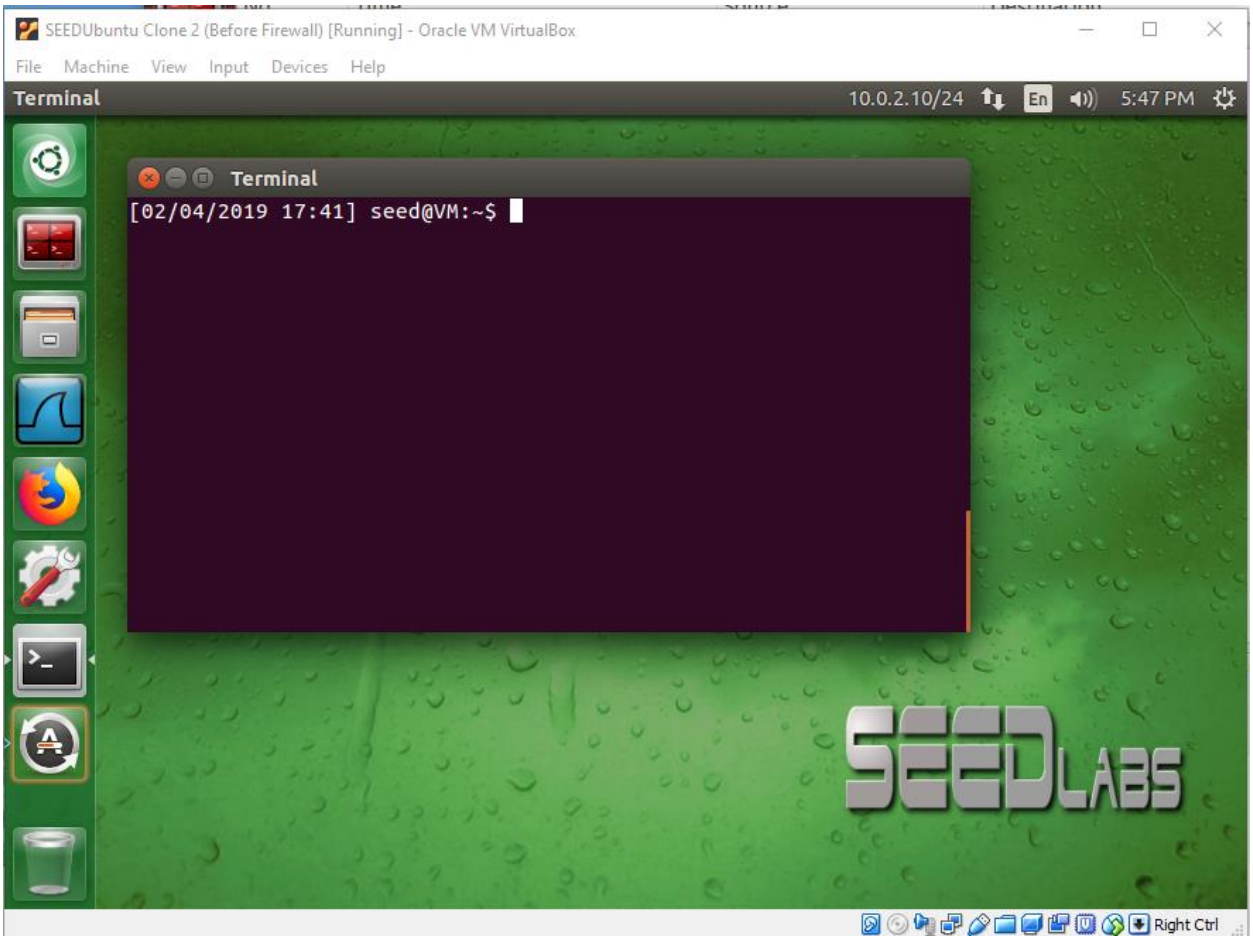
on the network/NAT and replies would be sent to that (1.2.3.4) address. ***We do see the responses via WireShark which will shown below.***

4. rmmod minifw.ko to remove the module. We see the DMESG that shows that the filters are being removed.
5. ping 10.0.2.10 works again.

The WireShark captures are show and explained after the following two figures.



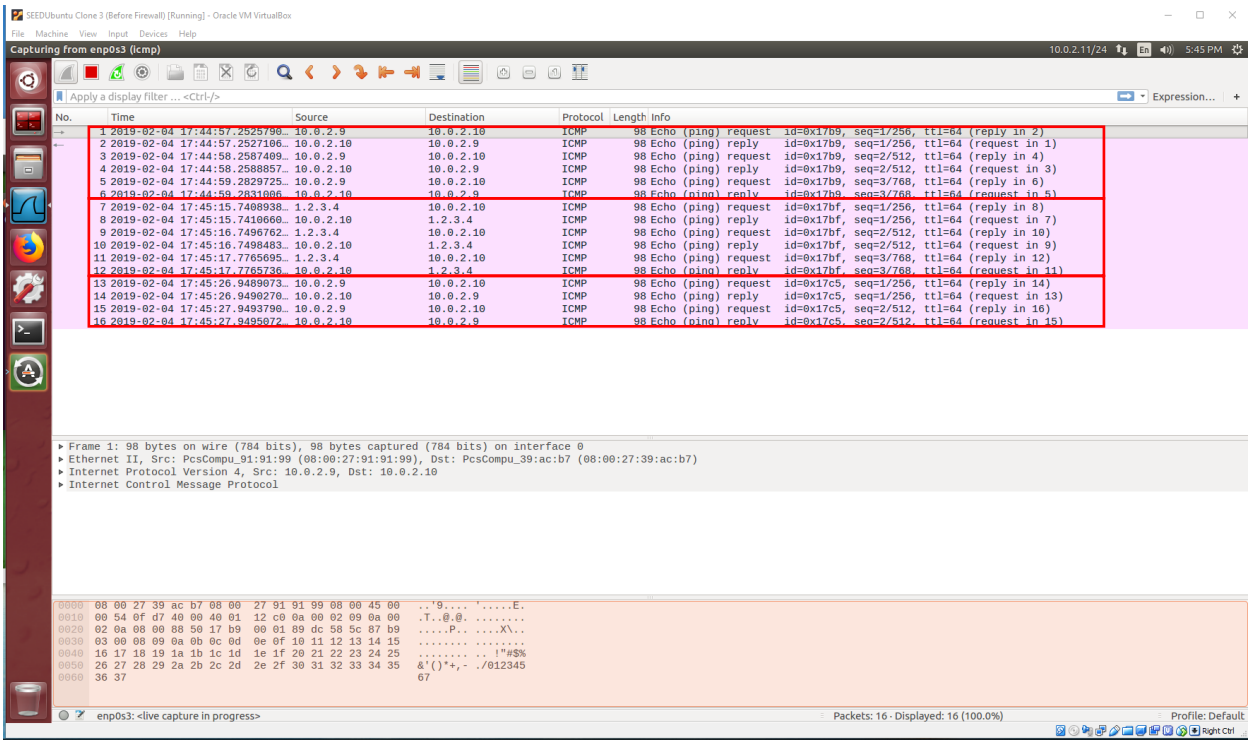
In the screen shot below, we simply see the Green VM 10.0.2.10 which is the VM that we're pinging.



Below is the WireShark capture of ICMP packets. We ran WireShark on the Red VM 10.0.2.11. For this exercise, we this VM was only used for WireShark.

We can see three ping sessions (red rectangles in the figure) which correspond to the ICMP Echo (ping) requests / responses shown in the Blue VM figure above the Green VM figure:

1. The first is the successful ping request and response from Blue VM 10.0.2.9 to Green VM 10.0.2.10.
2. We then see the "modified by our minifw kernel module filter" ping which shows that pings occurred from 1.2.3.4 to 10.0.2.10. Please recall, that we actually pinged from 10.0.2.9, but since we modified the source IP, it now shows as 1.2.3.4. Of course, that IP does not exist on the network, but 10.0.2.10 does receive the Request and responds with an ICMP Echo Reply to 1.2.3.4. Since there is no 1.2.3.4 the only way to view the response is to view it via WireShark, which is shown here.
3. Successful ping request and response from Blue VM 10.0.2.9 to Green VM 10.0.2.10.



Observations / Explanations

We created filters which register to NF_INET_POST_ROUTING (telnetFilter and httpFilter) hook and filters which registers to NF_INET_LOCAL_OUT (icmpFilter).

The filters get called whenever these hooks are encountered as part of the packet processing flow.

We insert all of this code into a kernel module which we insert into the kernel space. This allows use to interact and register our hooks with the NetFilter infrastructure.

By creating a kernel module with our NetFilter hooks, we are able to filter packets and choose what to do with them. In the case of telnet, we simply dropped them. In the case of HTTP, we were able to drop packets going to a particular web-site; i.e. SEEDLabs. Finally, we were able to modify an outbound ICMP packet and change its source IP address.

Using NetFilter hooks in kernel space provides a low-level way to filter packets. Certainly very effective and gives a great deal of flexibility and control, but more complex then using UWF.

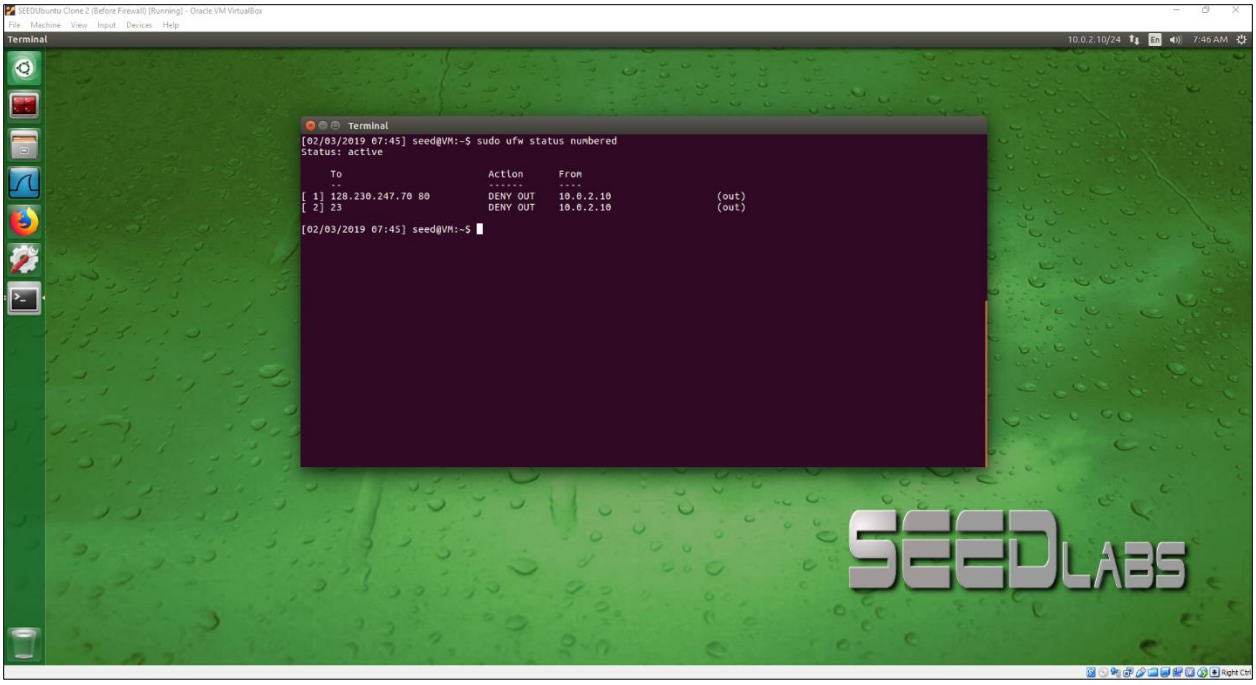
Full code listing of all the minifw.c module with all three filters for Telnet (telnetFilter), Http (httpFilter), and ICMP (icmpFilter) listed below in the Code Listing section.

Task 3: Evading Egress Filtering

From exercise one, we setup two rules via UFW on the Green VM:

1. Block all the outgoing traffic to external telnet (port 23) servers.
    a. "sudo ufw deny out from 10.0.2.10 to any port 23"
2. Block all the outgoing traffic to http://www.cis.syr.edu (IP 128.230.247.70, port 80).
    a. "sudo ufw deny out from 10.0.2.10 to 128.230.247.70 port 80"

In the figure below, the UFW rules can be seen, including the firewall status which is Active.

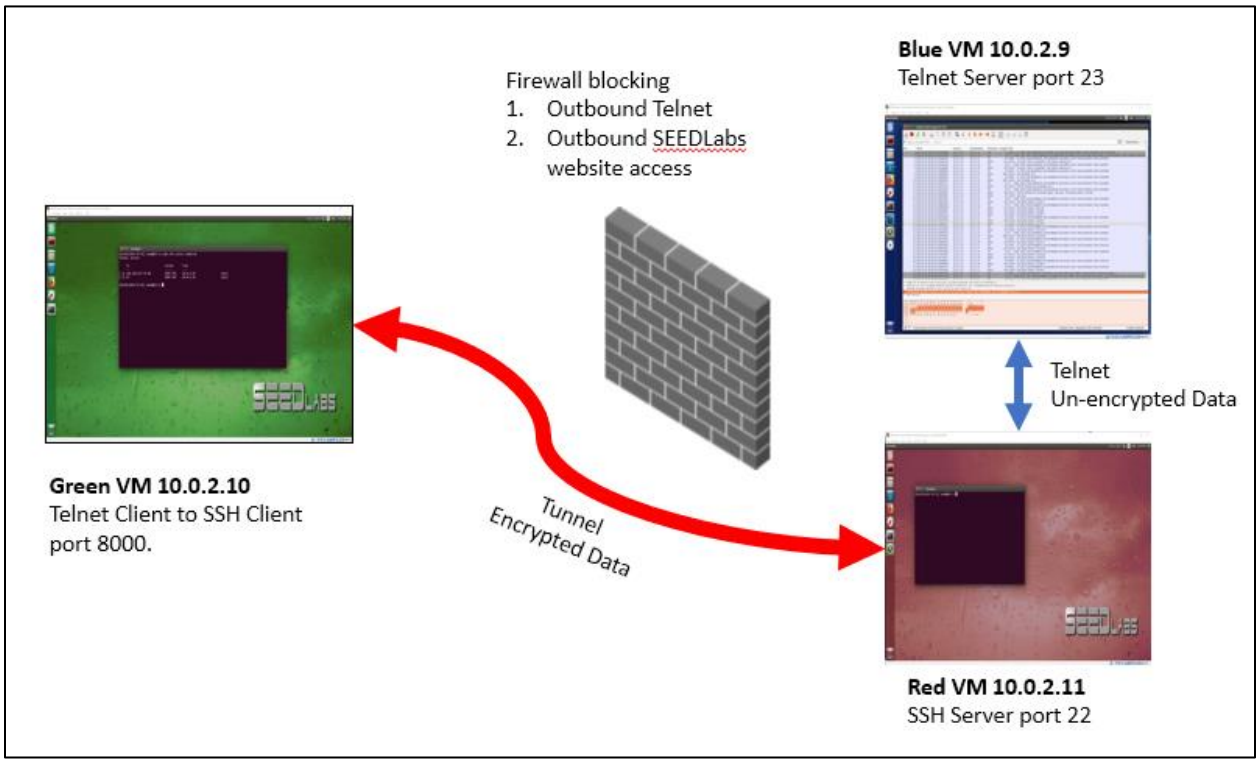


Task 3.a: Telnet to Machine B though the Firewall

For this task, we are using three VMs:

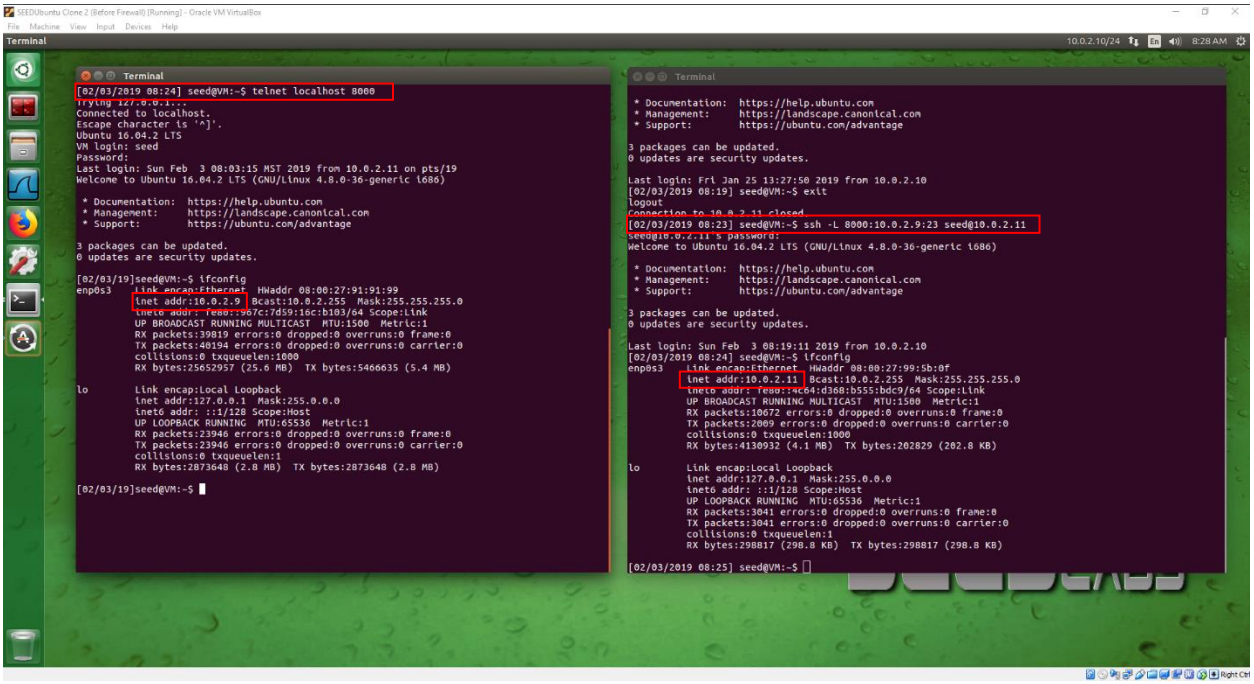
1. **Green VM 10.0.2.10**
    a. This VM will have our firewall rules as described in the section above.
    b. Setup the SSH tunnel to encrypt data between this Green VM and Red VM 10.0.2.11.
    c. Act as the client which telnets to the Blue VM 10.0.2.9 through the SSH tunnel (port 8000).
2. **Red VM 10.0.2.11**
    a. This VM is running the SSH server. It will receive encrypted packets from Green VM and forward them, unencrypted, to Blue VM where the Telnet server is running. The reverse occurs when the Telnet server sends data back to the Green VM.
3. **Blue VM 10.0.2.9**
    a. This VM is running the Telnet server.
    b. This VM is running WireShark, looking for "portrange 22-23", which includes SSH and Telnet packets.

Below is a picture describing this scenario:

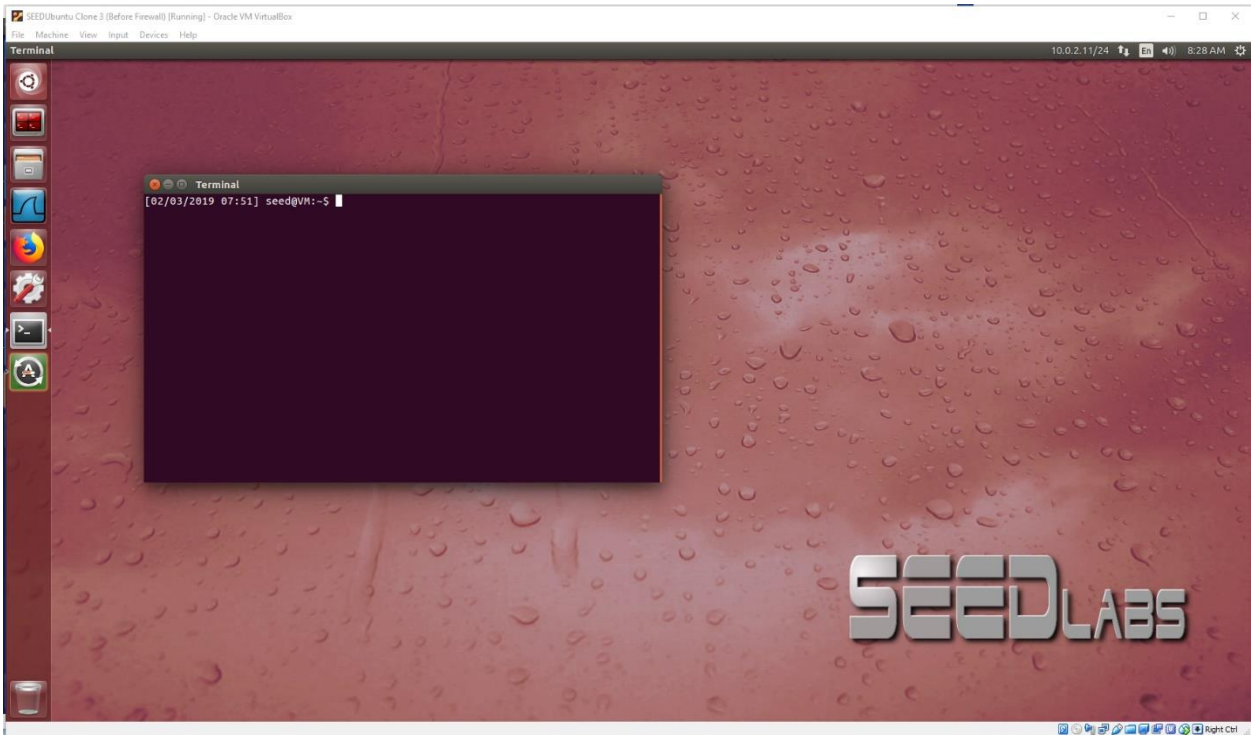


The figure below shows the Green VM with two terminals:

1. Right Terminal. Here we can see:
    a. The SSH session setup. "ssh -L 8000:10.0.2.9:23 seed@10.0.2.11"
        i. "ssh -L" specifies an SSH Tunnel with Local Port Forwarding. So, packets are sent from the client SSH (Green VM) to the server SSH (Red VM) and then FORWARDED to the Telnet Server (Blue VM).
        ii. "8000:10.2.9:23" specified our local port 8000 for the SSH client and destination (forwarding) IP and port – 10.0.2.9:23 (Telnet server).
        iii. "seed@10.0.2.11" specifies the host machine; i.e. SSH server machine. Username "seed" is used since we login with seed user and pass to this machine.
    b. Login into the SSH shell.
        i. Notice the IP Address in the shell is 10.0.2.11. Since we created an SSH tunnel from Green VM 10.0.2.10 to Red VM 10.0.2.11, seeing Red VM's IP address is appropriate since we are SSH'ed to this destination.
2. Left Terminal. Here we can see:
    a. Telnet'ing into the localhost port 8000. We use this port since that is the local port we specified when creating the SSH session.
    b. Upon successfully telnet'ing, we see our IP address is 10.0.2.9 which is the Blue VM. This is our ultimate destination.

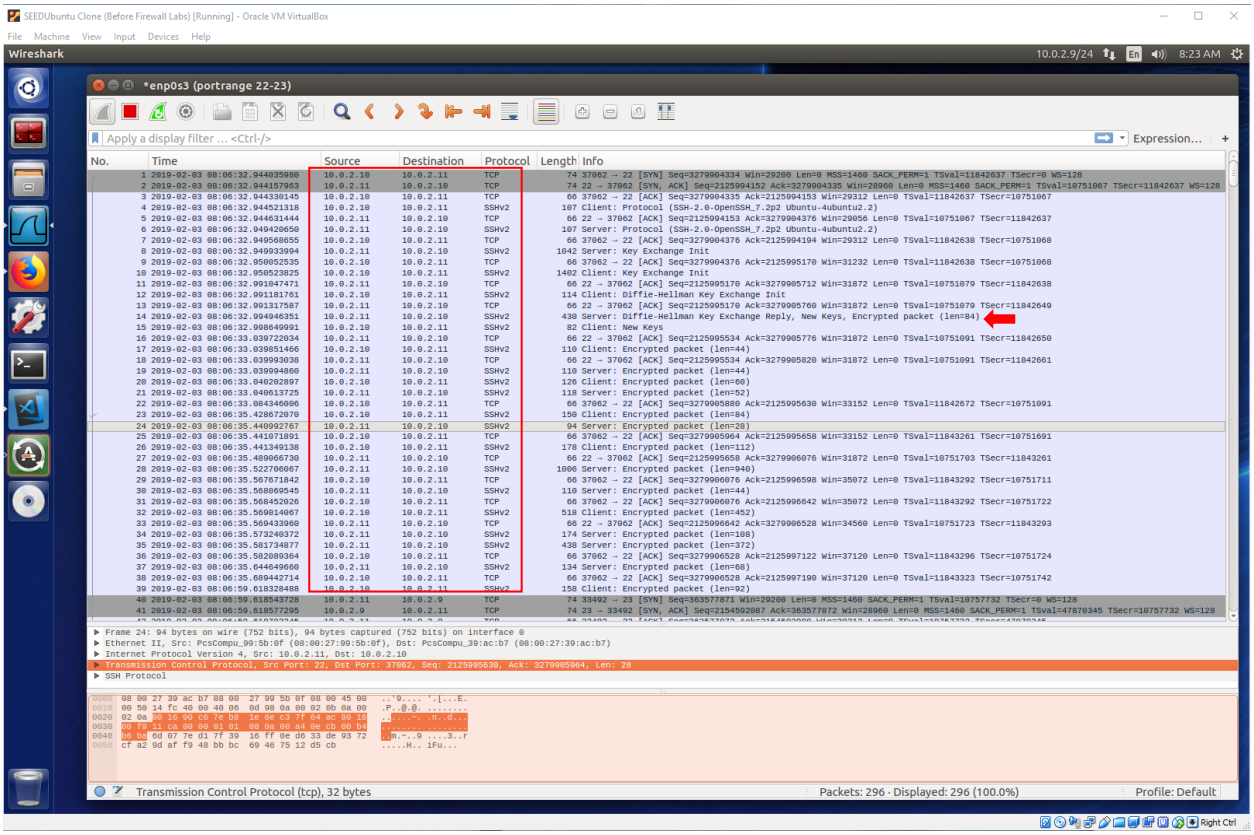


The figure below is the Red VM which is the SSH Server VM. Recall this is 10.0.2.11 and serves as the SSH server and packet forwarder to Blue VM. SSH is a server, so nothing to see here, but it is running and configured to allow local port forwarding.



The figures below show the Blue VM. This is IP Address 10.0.2.9. This is the final destination for the telnet packets since that is where the SSH server forwards them to. It's also the VM where we are running WireShark.

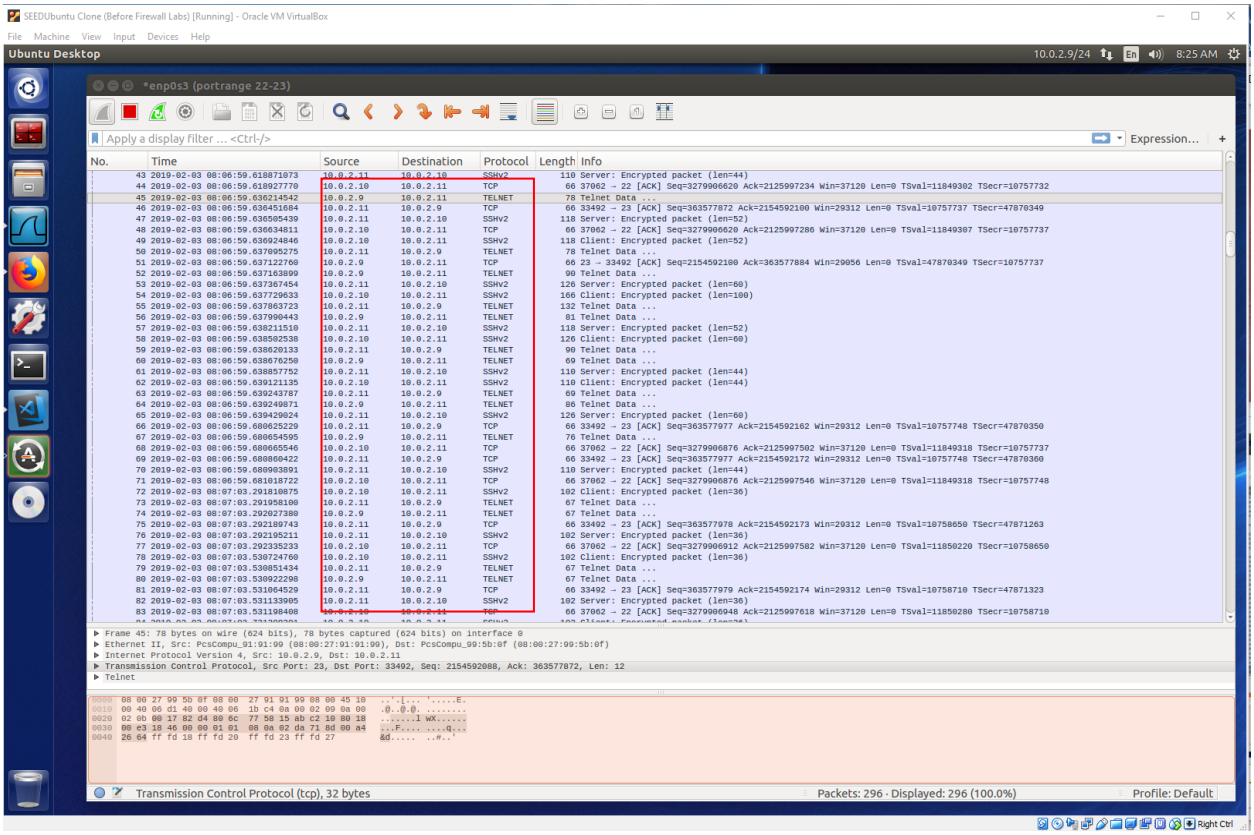
In the figure below, we see the WireShark output. While not dissecting each packet, you can see in the rectangle below, the SSH connection establishment between client SSH 10.0.2.10 and server SSH 10.0.2.11. We can also see the Server and Client key exchange / sharing to create a secure encrypted channel.



In the figure below, which overlaps a little from the previous WireShark capture, we can see the Telnet from Green VM 10.0.2.10 to Blue VM 10.0.2.9. It's interesting since we can see how data is exchanged between the client SSH and the destination Telnet server. The data flow can generally be described as:

1. SSH data from 10.0.2.10 to 10.0.2.11. This is encrypted data send to the SSH server. It is actually Telnet data, but since it is encrypted in an TCP SSH packet.
2. Telnet data from 10.0.2.11 to 10.0.2.9. This is the decrypted packet which has been forward to 10.0.2.9.
3. Telnet data from 10.0.2.9 to 10.0.2.11.
4. SSH data which is the encrypted data from #3 from 10.0.2.11 to 10.0.2.10.

A more simpler way to think about it is that, the client SSH sends encrypted Telnet data through the SSH tunnel which then forwards the decrypted Telnet data to the Telnet server. The Telnet server responds back the same way, but in reverse.

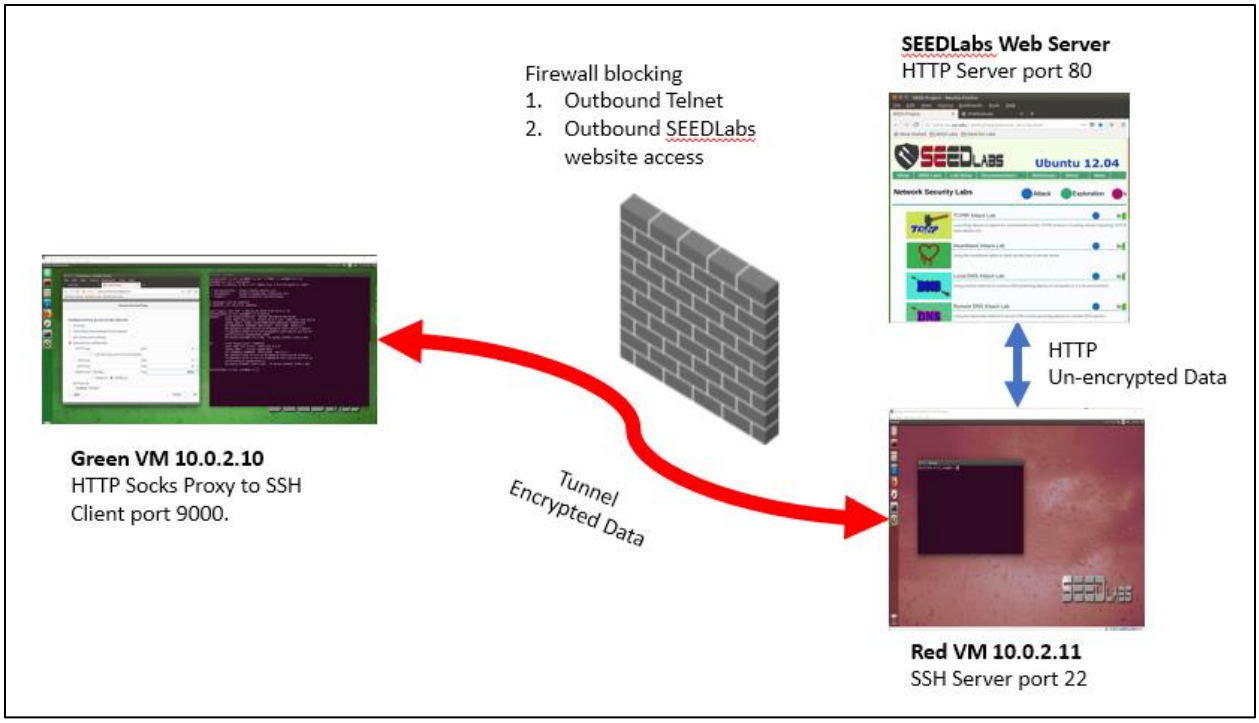


Task 3.b: Connect to SEEDLabs website using SSH Tunnel

For this task, we are using three VMs:

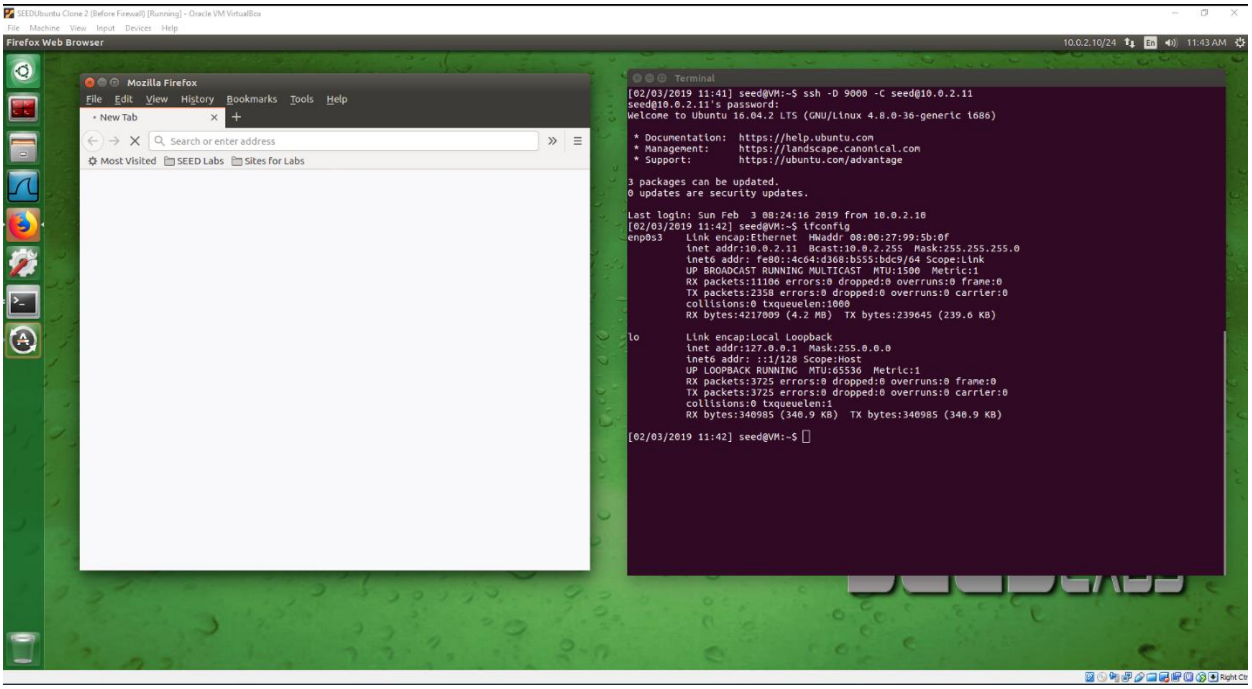
1. **Green VM 10.0.2.10**
    a. This VM will have our firewall rules as described in the beginning of this section.
    b. Setup the SSH tunnel to encrypt data between this Green VM and Red VM 10.0.2.11.
    c. Act as the Socks Proxy Server sends HTTP traffic to the SEEDLabs web-site through the SSH tunnel (port 9000).
2. **Red VM 10.0.2.11**
    a. This VM is running the SSH server. It will receive encrypted packets from Green VM and forward them, unencrypted, to their destination (in our case, we are choosing to go to SEEDLabs web-site). The reverse occurs when the SEEDLabs web server sends data back to the Green VM.
3. **Blue VM 10.0.2.9**
    a. This VM is running WireShark, with no filter, so all packets are shown.

Below is a picture describing this scenario:

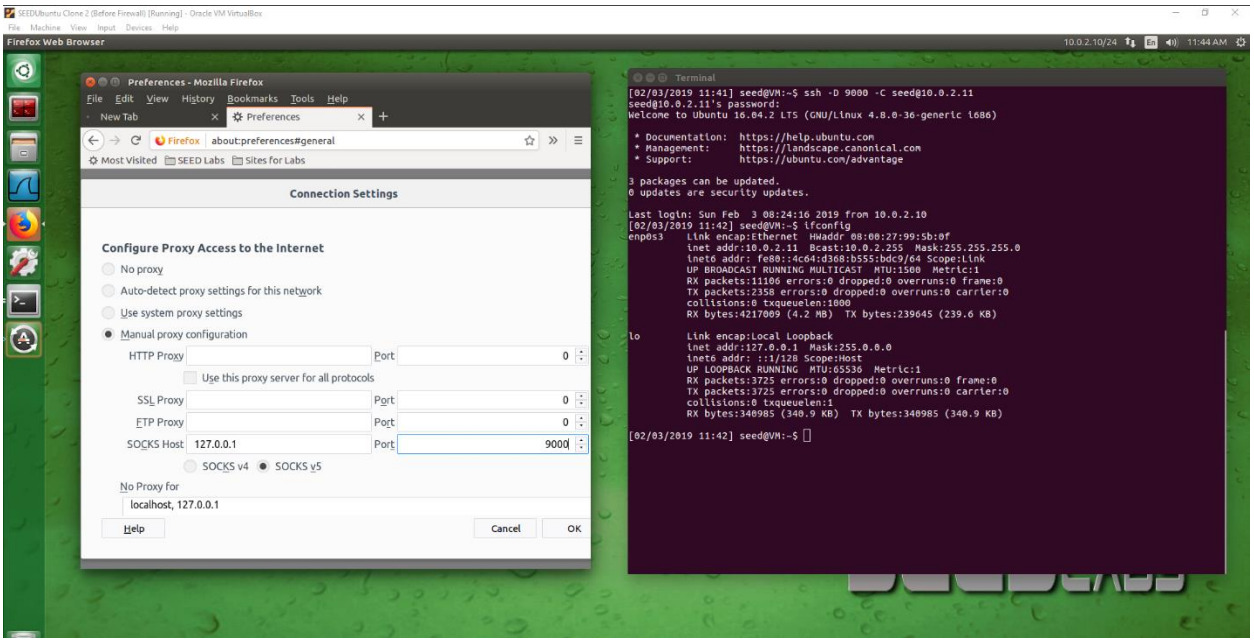


The figure below, shows two windows.

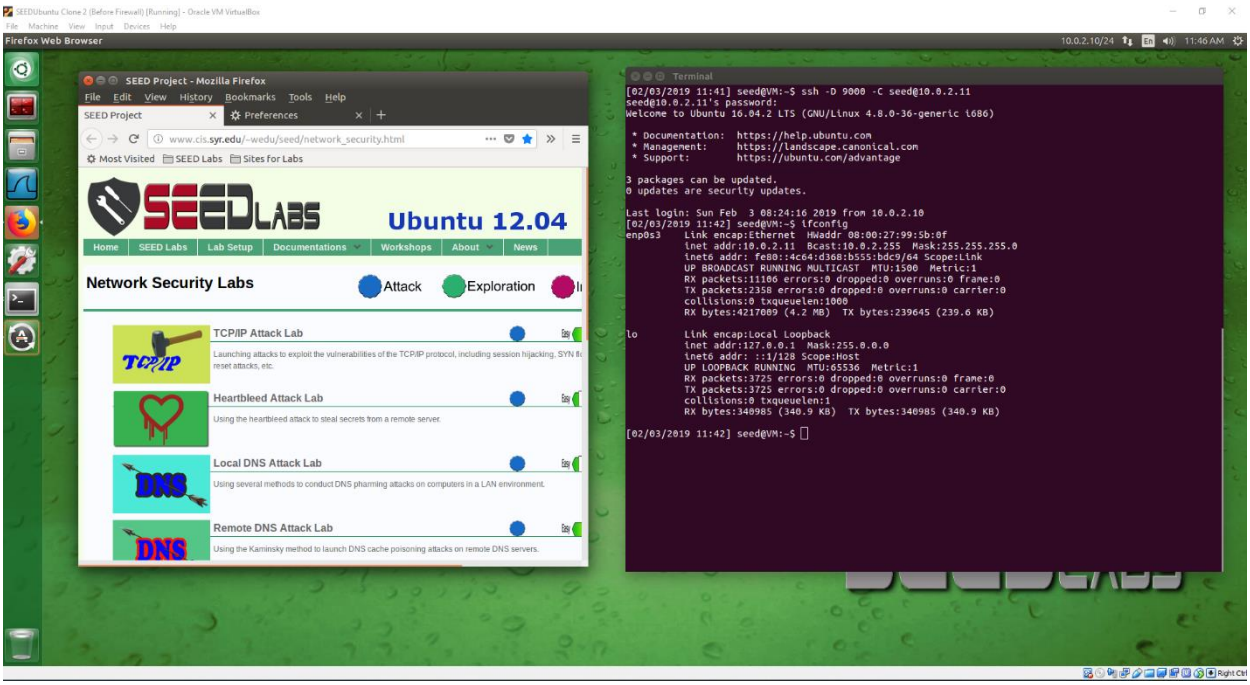
1. Right Terminal. Here we can see:
    a. The SSH session setup. "ssh -D 9000 -C seed@10.0.2.11"
        i. "ssh -D" specifies an SSH Tunnel with Dynamic Port Forwarding. So, packets are sent as a SSH Socks Proxy Server from the client SSH (Green VM) to the server SSH (Red VM) and then FORWARDED to the requested destination (as specified in the packet).
        ii. "-C" specifies that compression should be used during data transmission for better efficiency.
        iii. "seed@10.0.2.11" specifies the host machine; i.e. SSH server machine. Username "seed" is used since we login with seed user and pass to this machine.
    b. Login into the SSH shell.
        i. Notice the IP Address in the shell is 10.0.2.11. Since we created an SSH tunnel from Green VM 10.0.2.10 to Red VM 10.0.2.11, seeing Red VM's IP address is appropriate since we are SSH'ed to this destination.
2. Left Window. Here we can see:
    a. Firefox trying to access SEEDLabs web-site.
    b. At this time, we have not added the proxy settings to the browser, therefore it cannot access the web-site – recall, that the UFW firewall is in place to block outbound traffic form this Green VM to the SEEDLabs web-site.



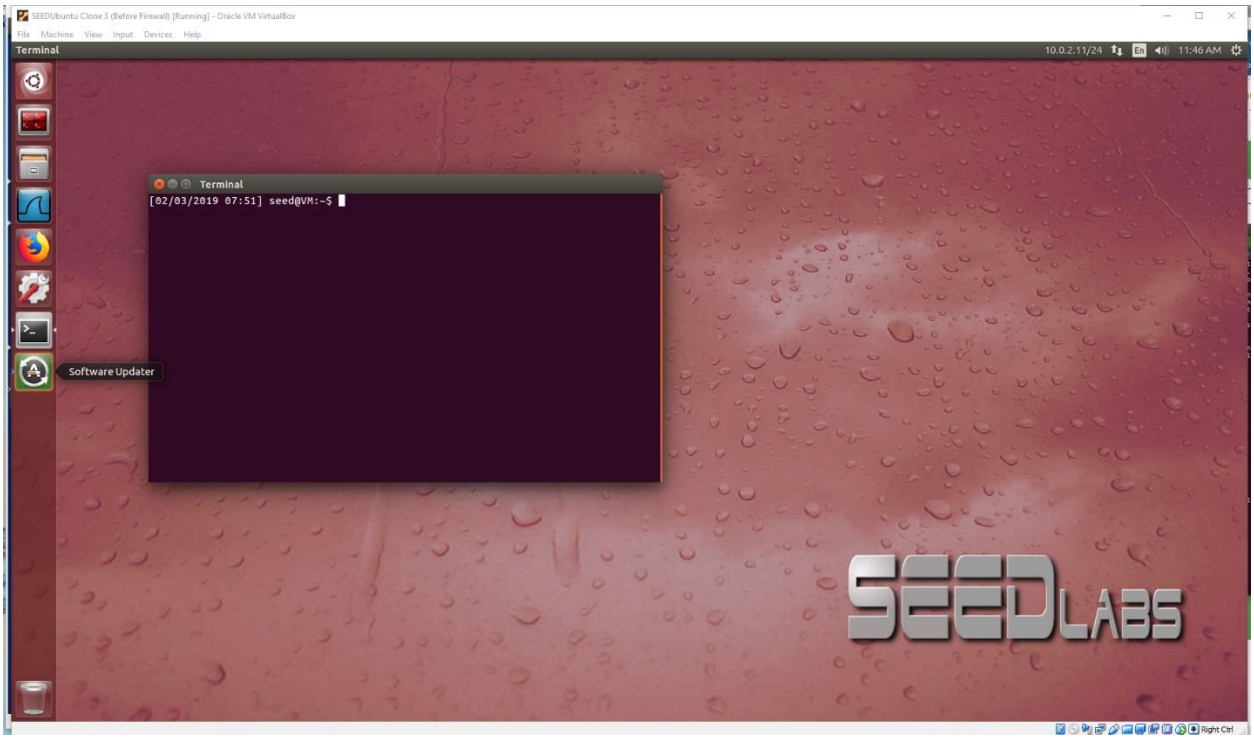
This next figure shows the Manual Proxy Configuration being placed into the Connection Settings of Firefox. Note, this Green VM is still connected to the SSH server 10.0.2.11.



After setting the proxy settings, we are able to access the SEEDLabs web-site!

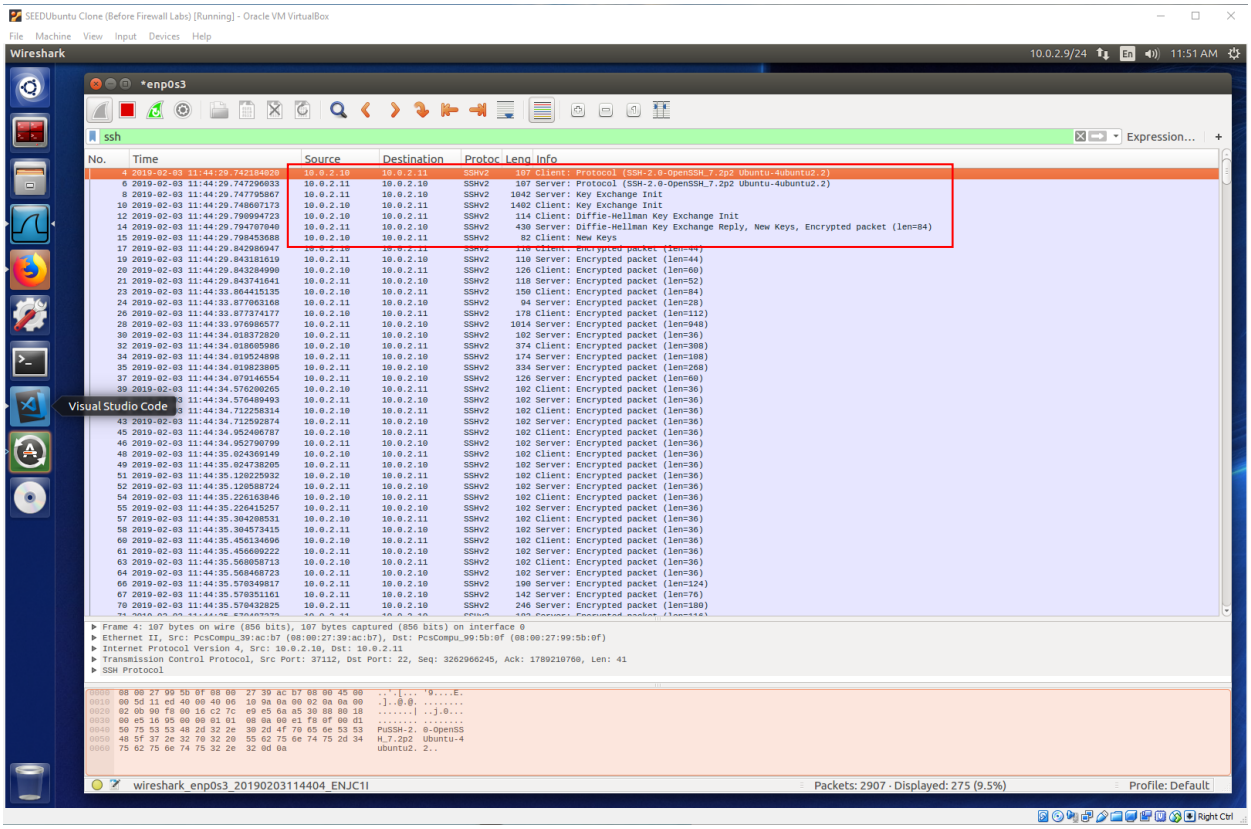


The figure below is the Red VM which is the SSH Server VM. Recall this is 10.0.2.11 and serves as the SSH server and packet forwarder to Blue VM. SSH is a server, so nothing to see here, but it is running and configured to allow local port forwarding.



The figures below show the Blue VM. This is IP Address 10.0.2.9. For this exercise, it's only purpose is to run WireShark.

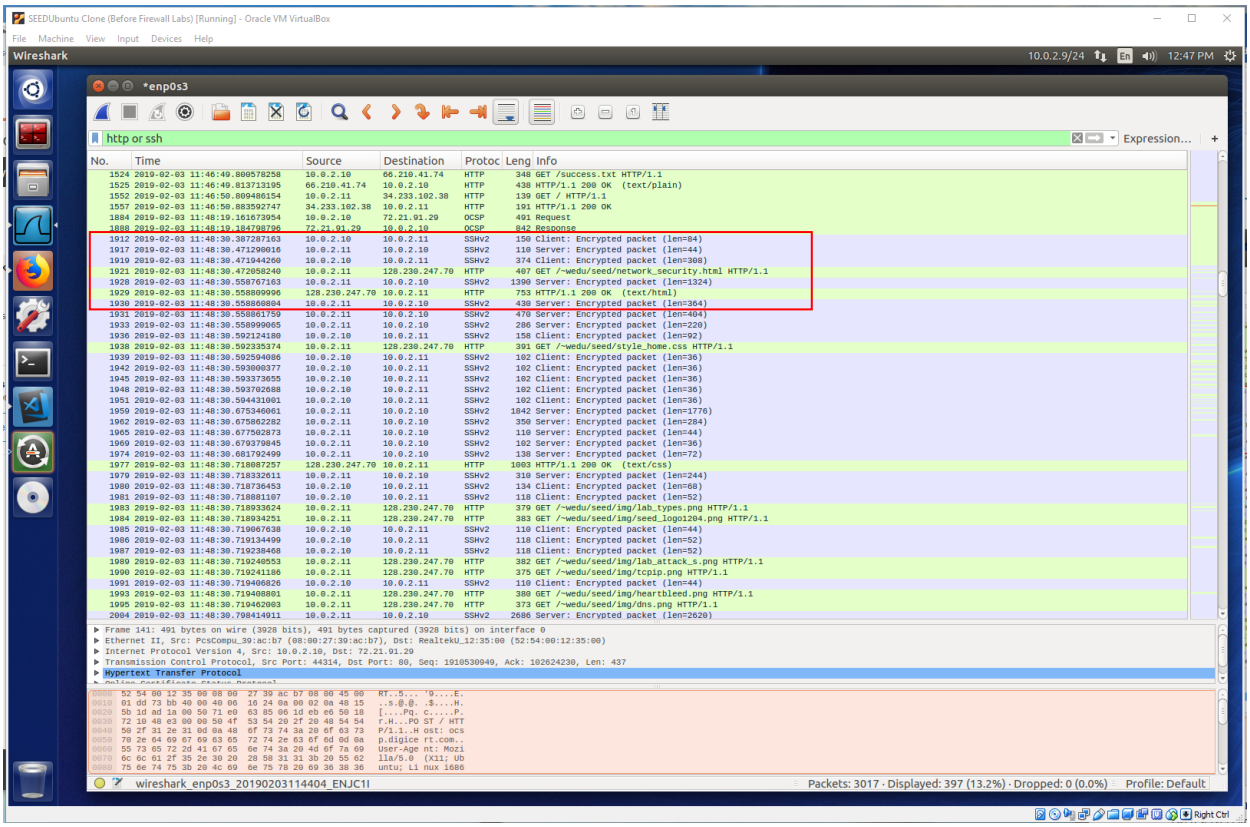
In the figure below, we see the WireShark output. While not dissecting each packet, you can see in the rectangle below, the SSH connection establishment between client SSH 10.0.2.10 and server SSH 10.0.2.11. We can also see the Server and Client key exchange / sharing to create a secure / encrypted channel. For this capture, all TCP packets are included, but I filtered for SSH so we can see the connection establishment messages.



In the figure below, we can see HTTP data from Green VM 10.0.2.9 to Blue VM 10.0.2.11. It's interesting since we can see how data is exchanged between the client SSH and the destination HTTP server. The data flow can generally be described as:

1. SSH data from 10.0.2.10 to 10.0.2.11. This is encrypted data sent to the SSH server. It is actually HTTP data, but it is encrypted in a TCP SSH packet.
2. HTTP data from 10.0.2.11 to SEEDLabs web-site 128.230.247.70. This is the decrypted packet which has been forward to 128.230.247.70.
3. HTTP data from SEEDLabs web-site 128.230.247.70 to 10.0.2.11.
4. SSH data which is the encrypted data from #3 from 10.0.2.11 to 10.0.2.10.

A more simpler way to think about it is that, the client SSH sends encrypted HTTP data through the SSH tunnel which then forwards the decrypted HTTP data to the HTTP/Web server. The HTTP/Web server responds back the same way, but in reverse.



Observations / Explanations

In this lab, we used a tunnel to bypass the firewall. For telnet, we created an SSH tunnel which receives encrypted telnet packets, decrypts them and forwards them to telnet server. For http, we created an SSH Socks Proxy Server, which accomplishes something similar by taking packets generated form the browser, securely sending them to the proxy server, which then forwards them to the web server their destined for. Of course, responses are routed the same way, but in reverse.

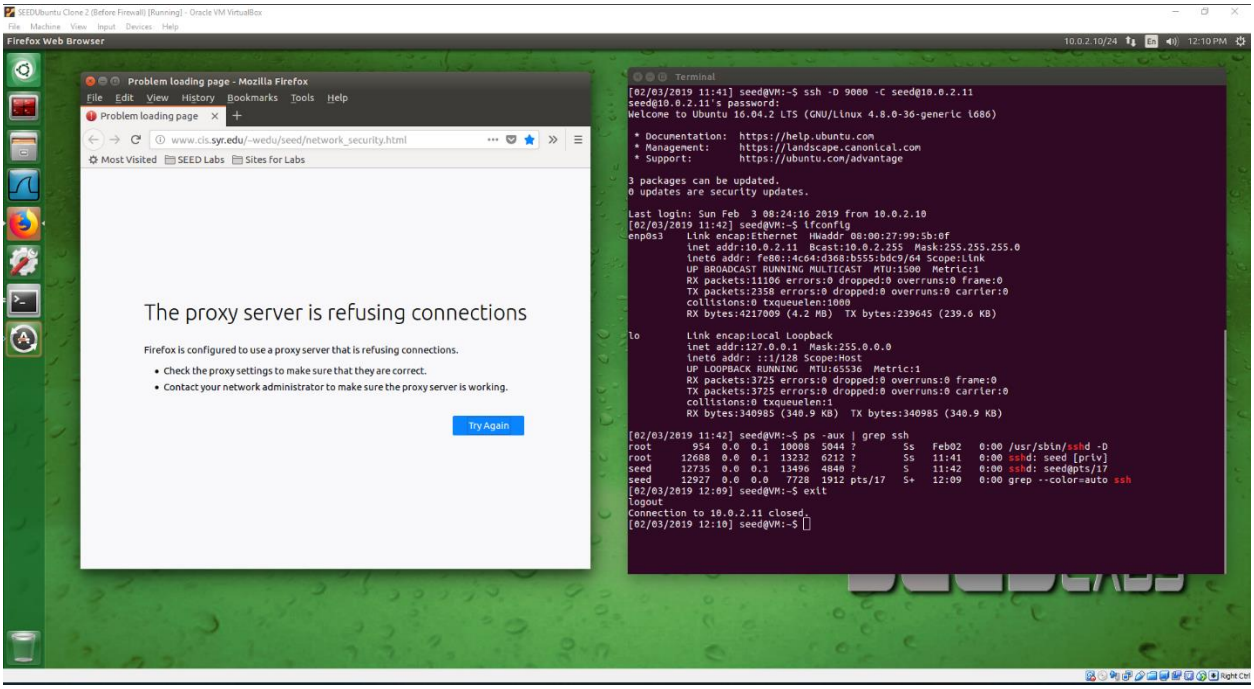
The interesting part is that we were able to use SSH to bypass our firewall rules. We were able to do this because by created the SSH tunnel, which is allowed by the firewall. We used this SSH tunnel to "tunnel" our telnet or HTTP payload in an encrypted format. Since these packets are encrypted, they are treated as "data" and not identified as telnet or HTTP packets, simply SSH packets with some encrypted data payload. Because of this, the firewall processing flow cannot filter these packets out and, as such, we can bypass the firewall.

Of course, for this to work, an SSH server must exist someplace which we can get to that can forward the packets to their destination. Knowing this, if I was an admin for a company, I'd probably block SSH!

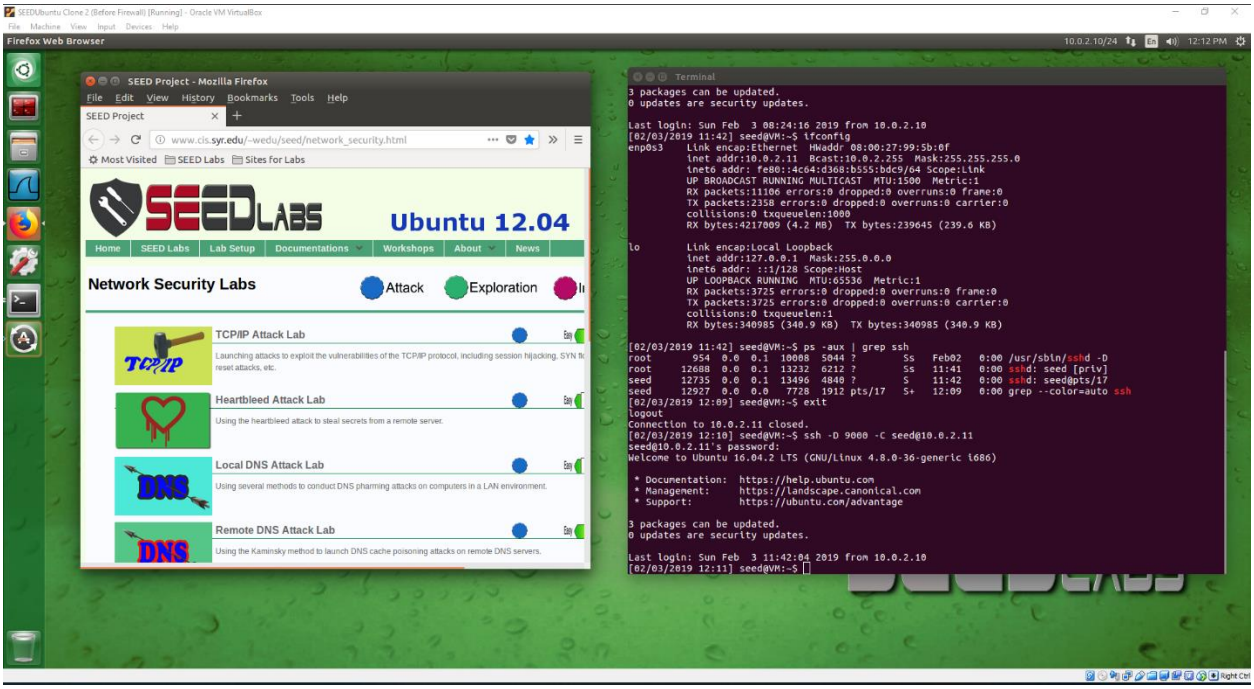
Questions

1. Run Firefox and go visit the Facebook page. Can you see the Facebook page? Please describe your observation.
    a. Yes. Observations shown above (using SEEDLabs web-site).

2. After you get the facebook page, break the SSH tunnel, clear the Firefox cache, and try the connection again. Please describe your observation.
    a. After breaking the SSH connection, I see the "The proxy server is refusing connections". See below.



3. Establish the SSH tunnel again and connect to Facebook. Describe your observation.
    a. I was able to see the web page. Since the SSH tunnel now exists, the proxy setting in the web browser can send HTTP requests to the localhost and port 9000 as we've configured.



4. Please explain what you have observed, especially on why the SSH tunnel can help bypass the egress filtering. You should use Wireshark to see what exactly is happening on the wire. Please describe your observations and explain them using the packets that you have captured.
    a. I've included WireShark captures in this and the previous exercise and included explanation on how data travels from client SSH to server SSH to the service (Telnet / HTTP). Please see that text above.
    b. The main idea is that, the Telnet or HTTP packets get encrypted and sent as data in the SSH packet payload. Since SSH is allowed by the firewall, it thinks that the packets are "ok" to travel outside the firewall (because it cannot inspect the encrypted packets). Once the packets get to the SSH server, they are decrypted and forwarded to the appropriate service; i.e. Telnet or HTTP (it can be any type of packet, but those are the kind we're sending).
    c. Bottom-line, firewall cannot see / inspect the encrypted payload and as long as SSH is allowed, it can be used as a method to bypass the firewall.

Code Listing

minifw.c

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/icmp.h>

static struct nf_hook_ops telnetFilterHook;
static struct nf_hook_ops httpFilterHook;
static struct nf_hook_ops icmpFilterHook;

unsigned int telnetFilter(void *priv, struct sk_buff *skb,
                          const struct nf_hook_state *state)
{
    struct iphdr *iph;
    struct tcphdr *tcph;

    iph = ip_hdr(skb);
    tcph = (void *)iph + iph->ihl * 4;

    // Filter Telnet port 23
    if (iph->protocol == IPPROTO_TCP && tcph->dest == htons(23))
    {
        // Drop packets to IP 10.0.2.10
        if ((((unsigned char *)&iph->daddr)[0] == 10) &&
            (((unsigned char *)&iph->daddr)[1] == 0) &&
            (((unsigned char *)&iph->daddr)[2] == 2) &&
            (((unsigned char *)&iph->daddr)[3] == 10))
        {
            printk(KERN_INFO "Dropping telnet packet to %d.%d.%d.%d\n",
                   ((unsigned char *)&iph->daddr)[0],
                   ((unsigned char *)&iph->daddr)[1],
                   ((unsigned char *)&iph->daddr)[2],
                   ((unsigned char *)&iph->daddr)[3]);
            return NF_DROP;
        }
    }
    return NF_ACCEPT;
}

unsigned int httpFilter(void *priv, struct sk_buff *skb,
                        const struct nf_hook_state *state)
{
    struct iphdr *iph;
    struct tcphdr *tcph;

    iph = ip_hdr(skb);
    tcph = (void *)iph + iph->ihl * 4;

    // Filter HTTP port 80
    if (iph->protocol == IPPROTO_TCP && tcph->dest == htons(80))
    {
        // Block packets to IP lcs-vc-wwwlinux.syr.edu (128.230.247.70)
        if ((((unsigned char *)&iph->daddr)[0] == 128) &&
            (((unsigned char *)&iph->daddr)[1] == 230) &&
            (((unsigned char *)&iph->daddr)[2] == 247) &&
            (((unsigned char *)&iph->daddr)[3] == 70))
        {
            printk(KERN_INFO "Dropping HTTP packet to %d.%d.%d.%d port %d\n",
                   ((unsigned char *)&iph->daddr)[0],
                   ((unsigned char *)&iph->daddr)[1],
                   ((unsigned char *)&iph->daddr)[2],
                   ((unsigned char *)&iph->daddr)[3],
                   htons(tcph->dest));
            return NF_DROP;
```

```
        }
    }

    return NF_ACCEPT;
}

/*****************************************
 * Listing 12.9: Calculating Internet Checksum
 *****************************************/

unsigned short in_cksum (unsigned short *buf, int length)
{
   unsigned short *w = buf;
   int nleft = length;
   int sum = 0;
   unsigned short temp=0;

   /*
    * The algorithm uses a 32 bit accumulator (sum), adds
    * sequential 16 bit words to it, and at the end, folds back all
    * the carry bits from the top 16 bits into the lower 16 bits.
    */
   while (nleft > 1)  {
       sum += *w++;
       nleft -= 2;
   }

   /* treat the odd byte at the end, if any */
   if (nleft == 1) {
        *(u_char *)(&temp) = *(u_char *)w ;
        sum += temp;
   }

   /* add back carry outs from top 16 bits to low 16 bits */
   sum = (sum >> 16) + (sum & 0xffff);  // add hi 16 to low 16
   sum += (sum >> 16);                  // add carry
   return (unsigned short)(~sum);
}

unsigned int icmpFilter(void *priv, struct sk_buff *skb,
                        const struct nf_hook_state *state)
{
    struct iphdr *iph;
    struct icmphdr *icmp;

    iph = ip_hdr(skb);
    icmp =  (struct icmphdr *)iph + iph->ihl*4;

    // Filter ICMP
    if (iph->protocol == IPPROTO_ICMP)
    {
        // Modify source to 1.2.3.4
        ((unsigned char *)&iph->saddr)[0] = 1;
        ((unsigned char *)&iph->saddr)[1] = 2;
        ((unsigned char *)&iph->saddr)[2] = 3;
        ((unsigned char *)&iph->saddr)[3] = 4;

        // Calculate the checksum for integrity
        iph->check = 0;
        iph->check = in_cksum((unsigned short *)iph,
                              sizeof(struct iphdr));

        printk(KERN_INFO "Modify ICMP packet to source IP %d.%d.%d.%d\n",
                ((unsigned char *)&iph->saddr)[0],
                ((unsigned char *)&iph->saddr)[1],
                ((unsigned char *)&iph->saddr)[2],
                ((unsigned char *)&iph->saddr)[3]);
    }

    return NF_ACCEPT;
}
```

```
int setUpICMPFilter(void)
{
    printk(KERN_INFO "Registering a ICMP filter.\n");
    icmpFilterHook.hook = icmpFilter;
    icmpFilterHook.hooknum = NF_INET_LOCAL_OUT;
    icmpFilterHook.pf = PF_INET;
    icmpFilterHook.priority = NF_IP_PRI_FIRST;

    // Register the hook.
    nf_register_hook(&icmpFilterHook);
    return 0;
}

void removeICMPFilter(void)
{
    printk(KERN_INFO "ICMP filter is being removed.\n");
    nf_unregister_hook(&icmpFilterHook);
}

int setUpHTTPFilter(void)
{
    printk(KERN_INFO "Registering a HTTP filter.\n");
    httpFilterHook.hook = httpFilter;
    httpFilterHook.hooknum = NF_INET_POST_ROUTING;
    httpFilterHook.pf = PF_INET;
    httpFilterHook.priority = NF_IP_PRI_FIRST;

    // Register the hook.
    nf_register_hook(&httpFilterHook);
    return 0;
}

void removeHTTPFilter(void)
{
    printk(KERN_INFO "HTTP filter is being removed.\n");
    nf_unregister_hook(&httpFilterHook);
}

int setUpTelnetFilter(void)
{
    printk(KERN_INFO "Registering a Telnet filter.\n");
    telnetFilterHook.hook = telnetFilter;
    telnetFilterHook.hooknum = NF_INET_POST_ROUTING;
    telnetFilterHook.pf = PF_INET;
    telnetFilterHook.priority = NF_IP_PRI_FIRST;

    // Register the hook.
    nf_register_hook(&telnetFilterHook);
    return 0;
}

void removeTelnetFilter(void)
{
    printk(KERN_INFO "Telnet filter is being removed.\n");
    nf_unregister_hook(&telnetFilterHook);
}

int setUpFilters(void)
{
    setUpTelnetFilter();
    setUpHTTPFilter();
    setUpICMPFilter();

    return 0;
}

void removeFilters(void)
{
    removeTelnetFilter();
    removeHTTPFilter();
```

```
    removeICMPFilter();
}

module_init(setUpFilters);
module_exit(removeFilters);

MODULE_LICENSE("GPL");
```