MH4921
Supervised Independent Study II

Linux Virtual Private Network (VPN) Lab

Brandon Goh Wen Heng

Academic Year 2018/19

Contents

1 Introduction

Organisation, Internet Service Providers (ISPs) and countries often block users from accessing certain external sites through what is known as egress filtering. This is used within organisations to reduce distractions, countries may make use of it to censor foreign web sites. However, these firewalls can easily be bypassed and there are multiple services/applications that aid in the circumvention.

The most common technology that is used to bypass these egress filtering are Virtual Private Networks (VPNs). VPNs are also commonly available on smartphone devices to help bypass these egress firewalls.

2 Overview

This lab will focus on how the VPN works and how it can be used to bypass egress firewalls. A VPN usually depends on two components, IP tunnelling and encryption. Tunnelling is crucial in helping to bypass these firewalls while the encryption helps to protect the content that is being transmitted through the VPN tunnel. For simplicity, encryption is not in the scope of this lab.

3 Exploration

Implementing a simple VPN for `Linux` is not trivial and is broken into multiple segments, with detailed observations for each step in the process. By setting up the VPN, a tunnel is established between two systems. When system A is behind a firewall and wants to access restricted resources, it will be blocked by egress filtering. Therefore, the packets are routed through to system B where it will send the packets out to the Internet. Similarly, the response from the Internet will be routed through to system B before sending it back to system A via the tunnel. This is how the VPN helps system A to bypass the firewall.

3.1 Virtual Machine (VM) Setup

3.1.1 VM Configuration

Two VMs are required for this lab. VM1 is behind a firewall while VM2 is outside the firewall. The objective is to help VM1 bypass the firewall so it is able to access blocked sites on the Internet.

VM1 and VM2 are connected to a separate NAT adapter, so both can access the Internet using the corresponding NAT servers. To emulate the Internet, where both VM1 and VM2 can communicate to each other, both VMs are connected to a host-only network adapter. This network emulates the facilitation of communication between the two VMs. The network setup is depicted in figure 1 for easier reference.

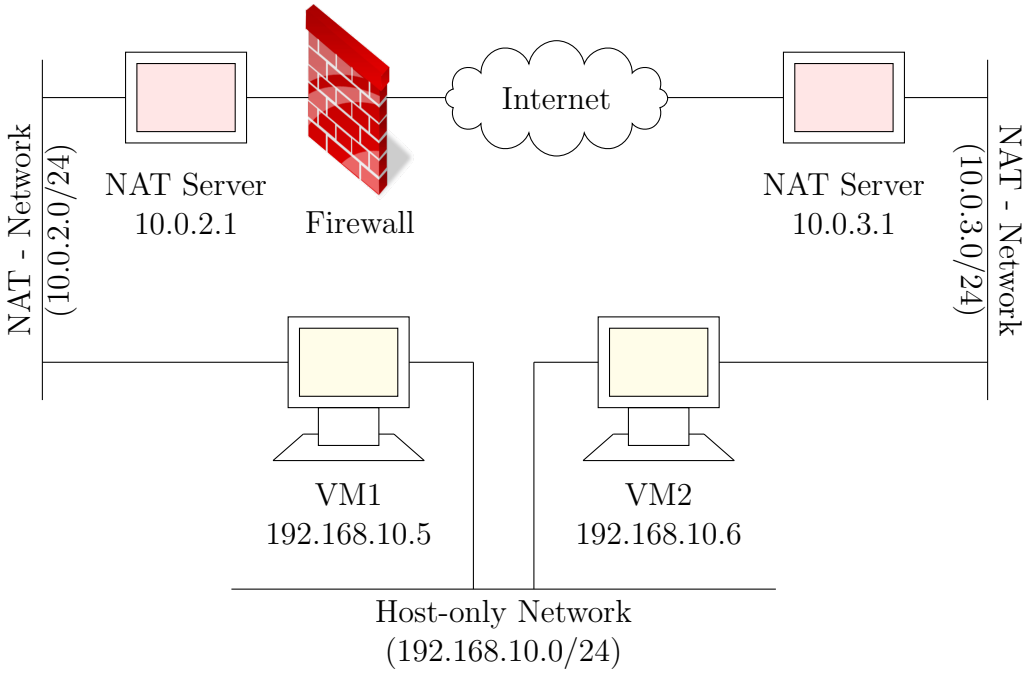


Figure 1: Network Topology

3.2 Task 1: Creating Host-to-Host Tunnel with TUN/TAP

TUN/TAP are essential and is widely implemented in modern computing systems. TUN and TAP are virtual network kernel drivers and are implemented entirely by software. TAP (as in network tap) simulates an Ethernet device and operates with layer-2 packets such as Ethernet frames. TUN (as in network TUNnel) simulates a network layer device and operates with layer-3 packets such as IP packets. With TUN/TAP, virtual network interfaces can be created.

A user-space program usually attaches to the TUN/TAP virtual network interface. Packets sent by an operating system via a TUN/TAP network interface are delivered to the user-space program. When packets are sent by the program via a TUN/TAP network interface, the packets are injected into the operating system network stack. On the operating system, the packets appear to originate from an external source through the virtual network interface.

When a program attaches to a TUN/TAP interface, the IP packets that the computer sends to this interface will be piped into the program. IP packets that are sent by the program instead will be piped into the computer, as if they came from the Internet through this virtual network interface. Standard `read()` and `write()` function calls can be made to send or receive packets to and from the virtual interface accordingly.

The source code to create a simple TUN/TAP has been provided by Davide Brini[1]

[1] `http://backreference.org/2010/03/26/tuntap-interface-tutorial`

and has been adapted by SEED Labs for this lab. The tutorial for the source code also shows how two computers can be connected using the TUN tunnelling technique. The adapted TUN/TAP code (`simpletun`) has been attached to Appendix A for reference.

The `simpletun` program can run as both a client and a server. To run as a client, the `-c` is used. To run it as a server instead, the `-s` flag is used. The following list details the requirements to create a tunnel between two computers using the `simpletun` program.

3.2.1 Setting Up Tunnel Point A

Tunnel point A is set to be the server side of the tunnel, which is on VM1. Once the tunnel between the two systems have been established, there is no difference between client and the server as the concept is only meaningful during the establishment of connection between two systems.

The following command is executed in terminal (with root privileges).

```
# ./simpletun -i tun0 -s -d
```

At this point, the virtual network device `tun0` has been created and the system now has multiple network interfaces. It will not show up if `ifconfig` is used as the interface is currently not active. Instead, the command `ip addr show` will list all physical and virtual network interfaces attached to the system regardless of status, as shown in Figure 2.

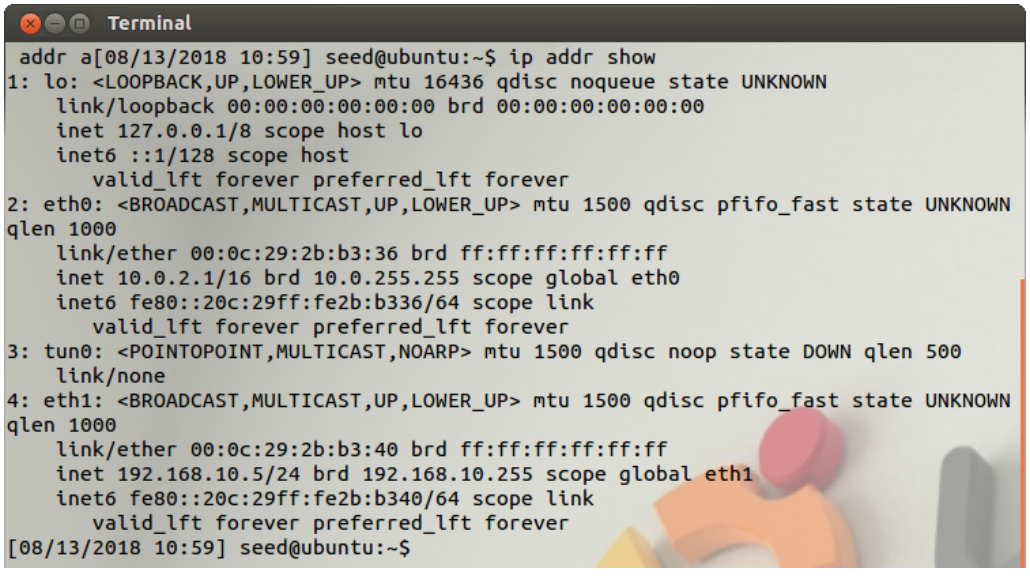


Figure 2: Virtual Network Details

The new virtual network device is not fully configured, as evident by the absence of an IP address. The IP address that we will be assigning will be from the reserved IP address space 10.0.0.0/8. It is also important to note that the `Terminal` window that

was used to create the virtual network interface is currently waiting for connections and cannot be used currently. Another window needs to be opened to assign an IP address to the virtual network interface `tun0`, using the following commands.

```
# ip addr add 10.0.3.15/24 dev tun0
# ifconfig tun0 up
```

If `ifconfig` is used now, it appears as a valid network interface with its configuration displayed.

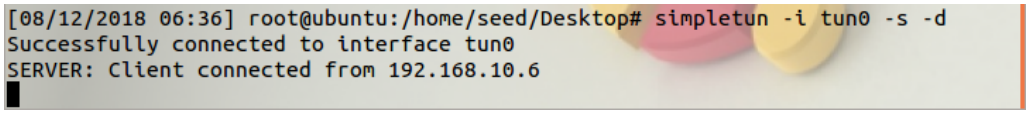
Figure 3: Valid Network Interfaces

3.2.2 Setting Up Tunnel Point B

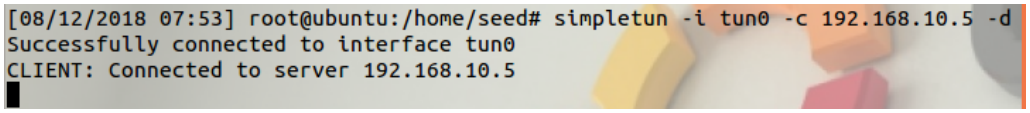
Tunnel point B is used as the client side of the tunnel, which is to be on VM2. The setup is similar to VM1. The first command used will connect our client to the server (VM1), which is tunnel point A.

```
# ./simpletun -i tun0 -c 192.168.10.5 -d
```

Executing the line above will immediately trigger the `Terminal` window on both VM1 and VM2 to state that the connection between each other has been established. This shows that the configuration is correct.



(a) Server Response



(b) Client Response

Figure 4: Both Systems Connected

Again, the previous command will prevent any further input in the current window, hence the following commands to initialise the virtual network interface must be done in a separate window.

```
# ip addr add 10.0.3.16/24 dev tun0
# ifconfig tun0 up
```

Once it has been executed, the virtual network interface will have all the configurations required to be a functional network adapter.

3.2.3 Establishing Routing Path

By the previous steps, a tunnel has been established between the two systems. However before it can be used to transmit data, the routing path needs to be set up so that both machines will direct the intended outgoing traffic through the tunnel. The following routing table directs all packs for the 10.0.3.0/24 network through the interface `tun0`, where the packets will travel through the tunnel. The following must be executed on **both** VMs:

```
# route add -net 10.0.3.0 netmask 255.255.255.0 dev tun0
```

3.2.4 Using The Tunnel

Using VM1, we are now able to access VM2 through the `tun0` adapter. Likewise, VM2 can access VM1 through the `tun0` adapter as well. The tunnel connection can be tested using the `ssh` or `ping` programs.

To use `ssh`, the `ssh` server must be started first.

```
$ sudo service ssh start
```

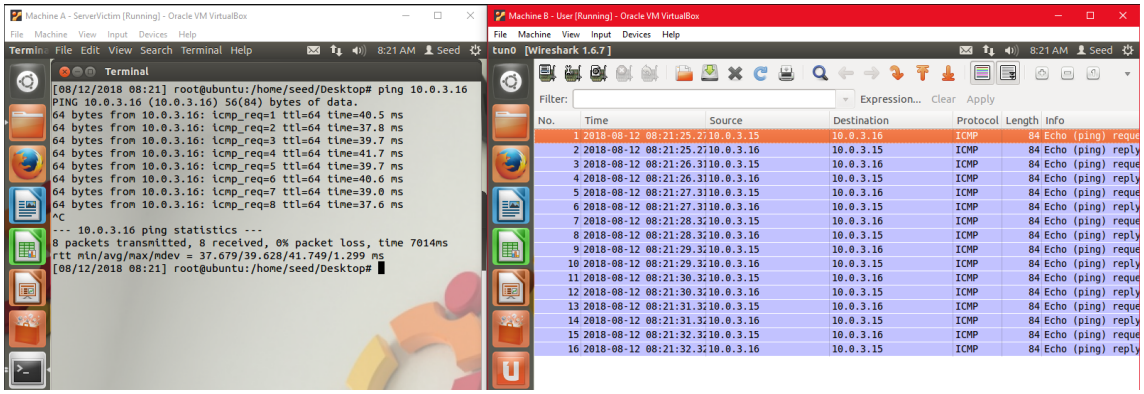
To test the tunnel from VM1:

```
$ ping 10.0.3.16
$ ssh 10.0.3.16
```

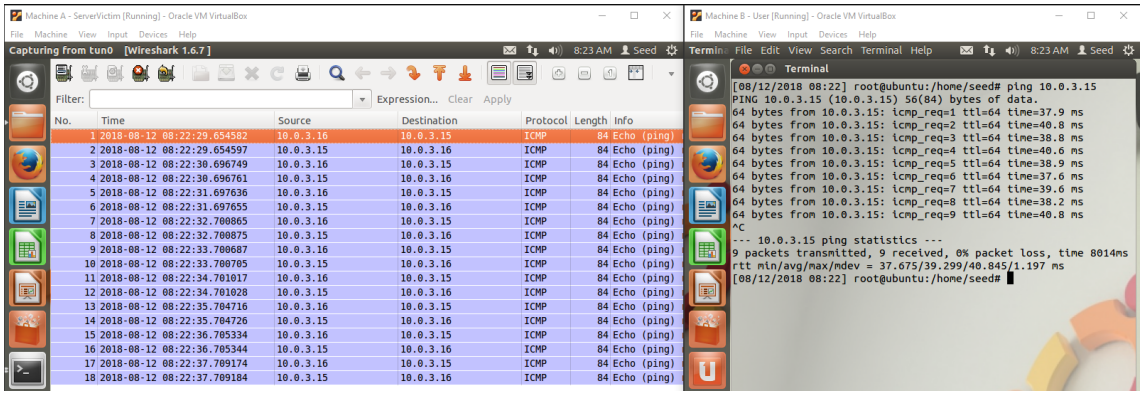
To test the tunnel from VM2:

```
$ ping 10.0.3.15
$ ssh 10.0.3.15
```

If the tunnel is successful, both `ssh` and `ping` will display valid responses from the other system. The figure below shows the packet capture from Wireshark for `ssh` in both directions is proof that the tunnel works as expected.

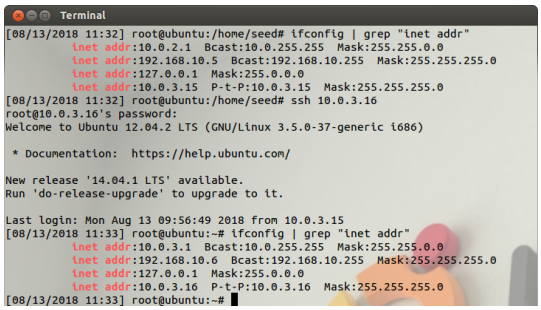


(a) `Ping` from A to B

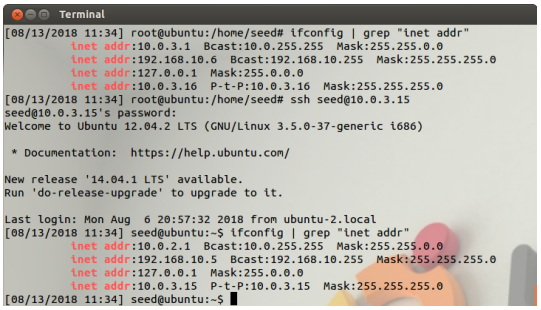


(b) `Ping` from B to A

Figure 5: Both Systems Contactable



(a) `SSH` from A to B



(b) `SSH` from B to A

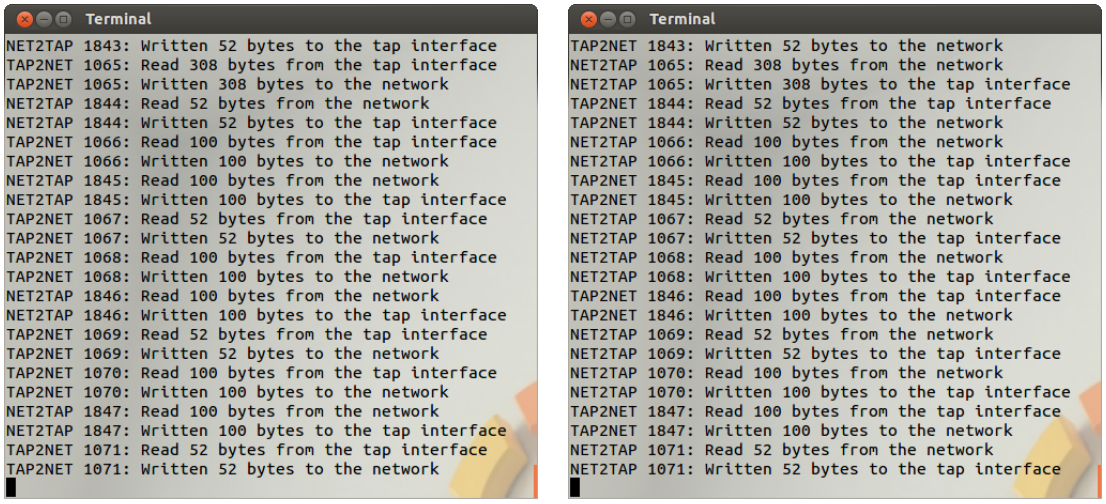
Figure 6: Both Systems Connected

Also of interest while the two systems are connected is that data that goes through the tunnel are displayed in the first Terminal window that was opened in VM1 and VM2 (the window that was used to start the `simpletun` program). Both display the data that was sent to/from the tap interface and the network. It is also noteworthy

to notice that the logs on both systems corroborate with the details mentioned in the introduction, which has been summarised into the table below with the corresponding screenshots from the VMs.

VM1	VM2
TAP2NET <id>: Read x bytes from the tap interface	NET2TAP <id>: Read x bytes from the network
TAP2NET <id>: Written x bytes to the network	NET2TAP <id>: Written x bytes to the tap interface
NET2TAP <id>: Read x bytes from the network	TAP2NET <id>: Read x bytes to the tap interface
NET2TAP <id>: Written x bytes to the tap interface	TAP2NET <id>: Written x bytes to the network

Table 1: Differences between logs of both VMs



(a) Logs from VM1 (b) Logs from VM2

Figure 7: Logs in Sync

To this point, the connection of both VMs using a tunnel has been successful. Figure 8 is a diagrammatic representation on the current state of the network between the two VMs.

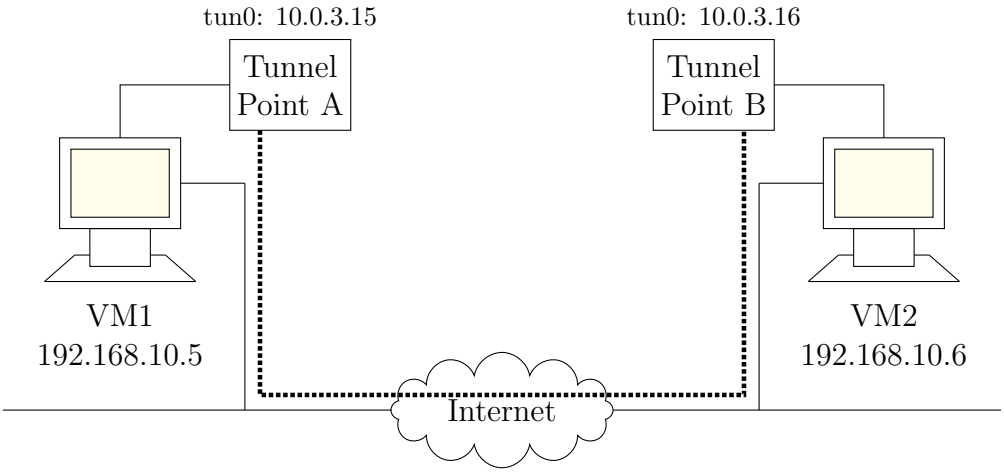


Figure 8: Tunnel Network over Emulated Internet

3.3 Task 2: Creating Host-to-Gateway Tunnel

Completing the previous task is insufficient as accessing VM2 through the tunnel is a secondary objective. Instead, the packet sent through the VPN tunnel are to be routed out to the Internet. In other words, VM2 must function as a gateway. This will make our tunnel a host-to-gateway tunnel. As a continuation of the previous task, the following additional tasks must be completed.

3.3.1 Setting Up IP Forwarding

Unless the system is explicitly configured, a computer will only as a host and not a gateway. To do so, the command below enables IP forwarding, allowing the computer to behave like a gateway.

```
$ sudo sysctl net.ipv4.ip_forward=1
```

3.3.2 Getting Around The Limitation of NAT

When the destination sends the reply packets back to the machine on the private network, it will reach the NAT first (as the source IP of all outgoing packets are changed to the NAT's external IP address). The NAT will usually replace the destination IP address with the IP address of the original packet and send it to the system to whoever owns the IP address.

Before the NAT sends out the packet, it needs to know the MAC address of the machine that own the IP address 10.0.3.15. To solve the limitation, an extra NAT is created on VM2 so that all packets sent out of VM2 will have VM2's IP address as its source IP. To reach the Internet, the packets will need to go through the NAT adapter that has already been provisioned when the VMs were set-up. The following sets of commands will enable the NAT on VM2.

The first step is to clean all of `iptables` rules.

```
$ sudo iptables -F
$ sudo iptables -t nat -F
```

Next, we need to add a rule on the postrouting position to the NAT adapter, in this instance with `eth0` (this network adapter may change depending on the interface identified within the VM). The network adapter selected for postrouting must be outward-facing.

```
$ sudo iptables -t nat -A POSTROUTING -j MASQUERADE -o eth0
```

3.3.3 Setting Up Firewall

The firewall is set up on VM1 to block the access of a target website. Setting up the firewall requires superuser privileges, as with the setup of the VPN tunnel. With reference to the previously completed Firewall lab, popular social media sites Facebook and Twitter are blocked to prevent distractions within organisations.

Setting up the firewall on VM1 invokes an issue where the firewall should not be able to block packets over the virtual network interface or the VPN will not be able to function at all. Therefore, the firewall rule cannot be set before the routing, nor can it be set on the virtual interface.

The firewall rule only needs to be set for the physical network interface so it will not affect the virtual network interface. The following `iptables` command can help to overcome this problem by preventing the packets with the affected IP addresses from going through the physical network adapter `eth0`.

```
# iptables -t mangle -A POSTROUTING -d 155.69.7.173/24 -o eth0 -j DROP
```

In the above code, the NTU site (including NTULearn, iNTU etc.) are all blocked as one IP address serves the entire domain. Since NTU is the only site is blocked by the firewall rule, accessing other sites is not an issue.

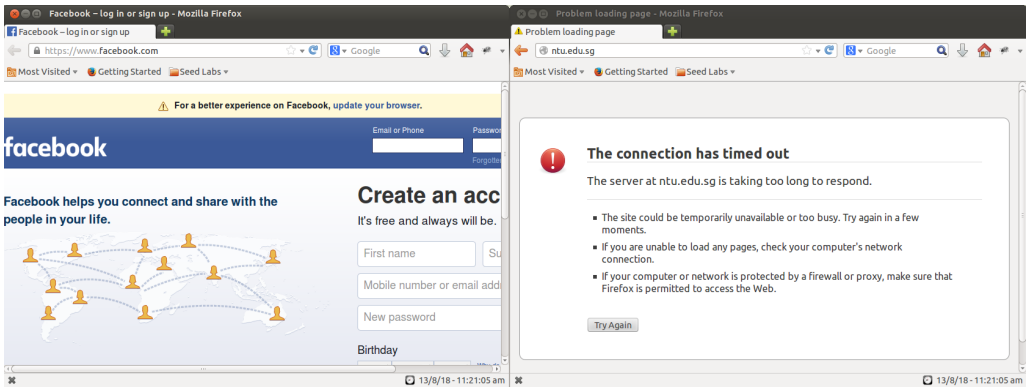


Figure 9: NTU Site Blocked

To manually check whether the IP address was properly stored in `iptables`, the following line of code can be used.

```
# iptables -L -t mangle --line-numbers
```

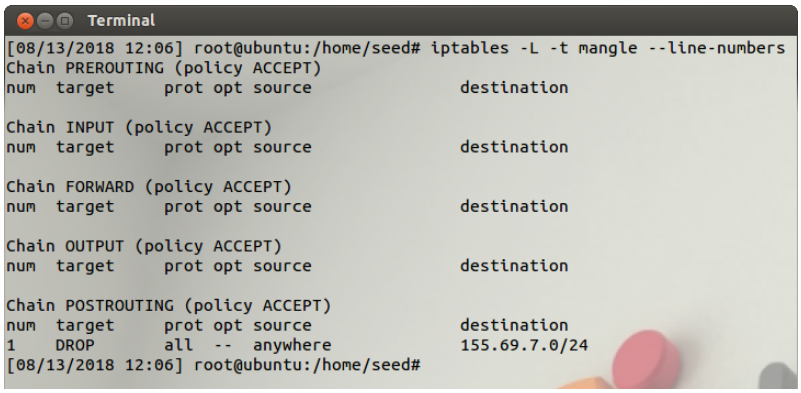


Figure 10: List of `mangle` entries

3.3.4 Bypassing Firewall

To bypass the egress filtering by the firewall, a SSH tunnel is established between the two VMs using the existing VPN tunnel. For web filtering, the dynamic port forwarding (`-D`) switch can be used as packets will automatically be forwarded to the destinations.

```
ssh -D 3000 seed@10.0.3.16
```

After SSH has been established, the browser must be configured to make use of the dynamic port forwarding mechanism. To do so, we must navigate to the following: `Edit` >`Preferences` >`Advanced` >`Network` >`Settings`. The "Manual Proxy Configuration" option must be selected. As we are using SSH, the SOCKS proxy should be used. For the host, it should be `localhost` with port `3000`. SOCKS v5 needs to be selected as well. The rest of the options will remain as the default. Figure 11 is a screenshot of the proxy settings in Firefox.

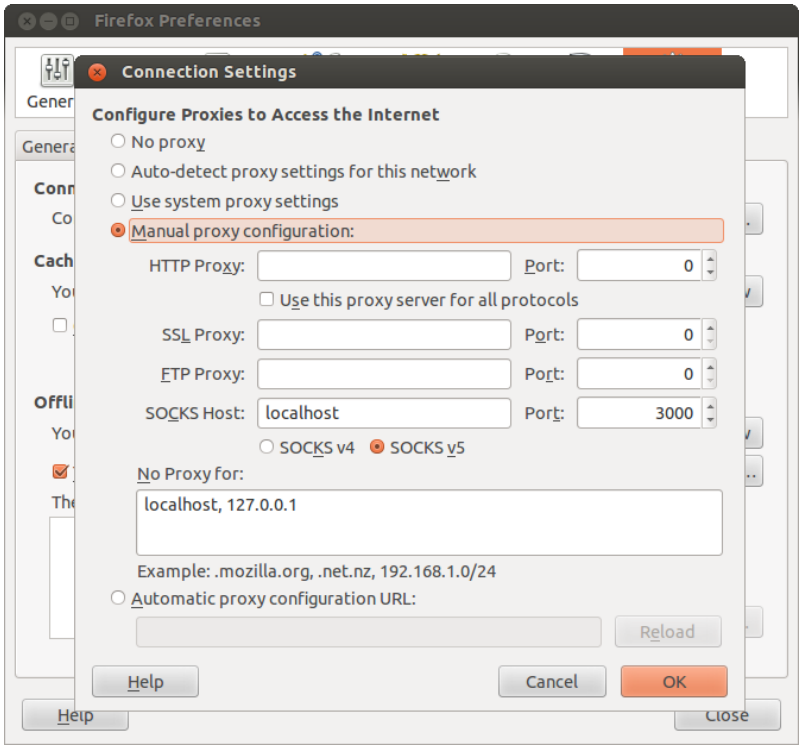


Figure 11: Proxy Configuration

Once completed, the *Connection Settings* & *Preferences* windows can be closed and the new settings will take effect immediately. If the proxy has been properly configured, then refreshing the site will allow the page to load up successfully, bypassing the firewall rule.

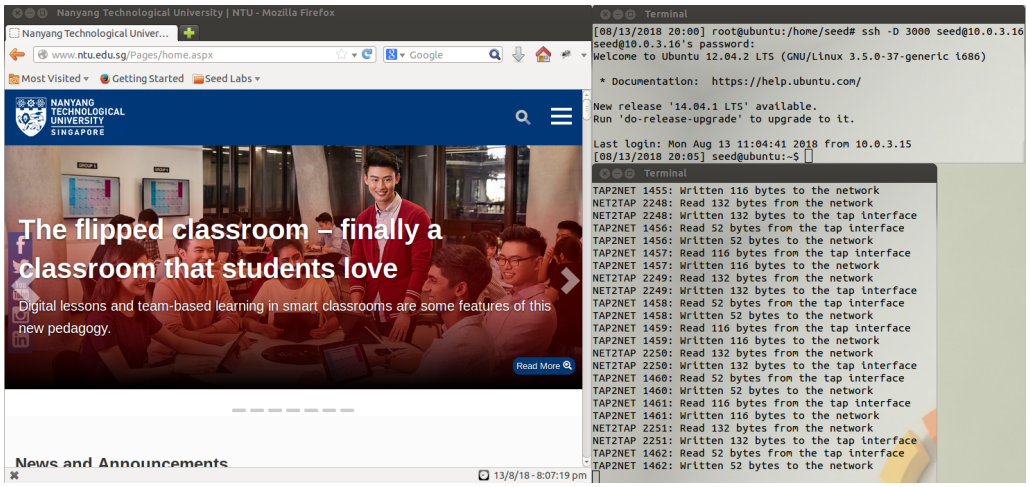


Figure 12: Firewall Bypassed For Websites

To demonstrate how applications can bypass the firewall, `Telnet` is used as a classic example. To block the `Telnet` protocol, the packet filtering firewall `ufw` can be used and it complements `iptables`. The following lines are executed in superuser privileges.

```
# ufw deny out to any port 23
# ufw deny in from any port 23
# ufw disable
# ufw enable
```

If there is any attempt to initiate `Telnet` from VM1, then the `Telnet` requests will timeout and no connection will be established since all packets are dropped by `ufw`.

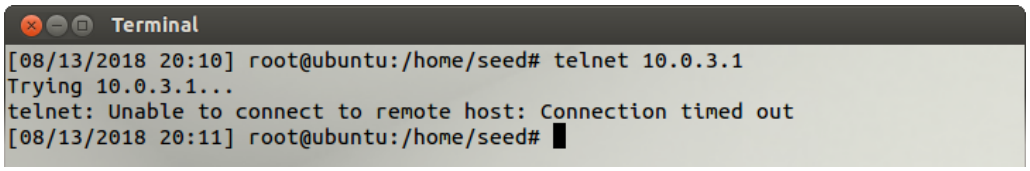


Figure 13: `Telnet` Timeout

To use `Telnet`, the SSH tunnel is made use of again. However, this time static port forwarding is used.

```
$ ssh -L 3000:10.0.3.16:23 seed@10.0.3.16
```

When the SSH tunnel has been established, the `Telnet` program has to make to use of the static port forwarding rule. To do so, `Telnet` connects to the source port and `localhost` to initiate the connection.

```
$ telnet localhost 3000
```

Successful connection through the SSL tunnel will prompt the user login and password of the remote system, in this case VM2. When `ifconfig | grep "inet addr"` is executed, the IP address that should show up will be for VM2.

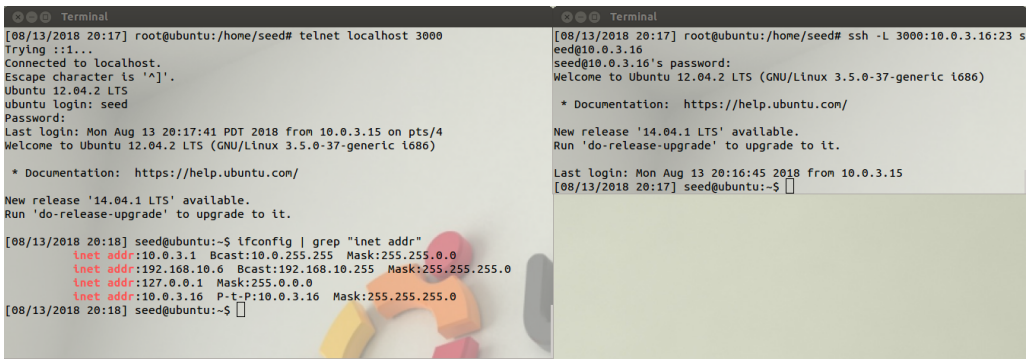


Figure 14: Proof of Successful `Telnet` via `ifconfig` Checking

Using Wireshark to analyse the packets between the two VMs, it can be noticed that packets that are from VM1 are sent to `tun0` with the source and the destination being the host-only network IP address (eth1). This packet is "repackaged" and sent through the tunnel by SSH. This is reflected as the source and destination IP of the packet is the IP address of the TUN/TAP that was set in task 1. Again when the data is sent back, the packet headers are modified by the TUN/TAP adapter before it travels through the tunnel.

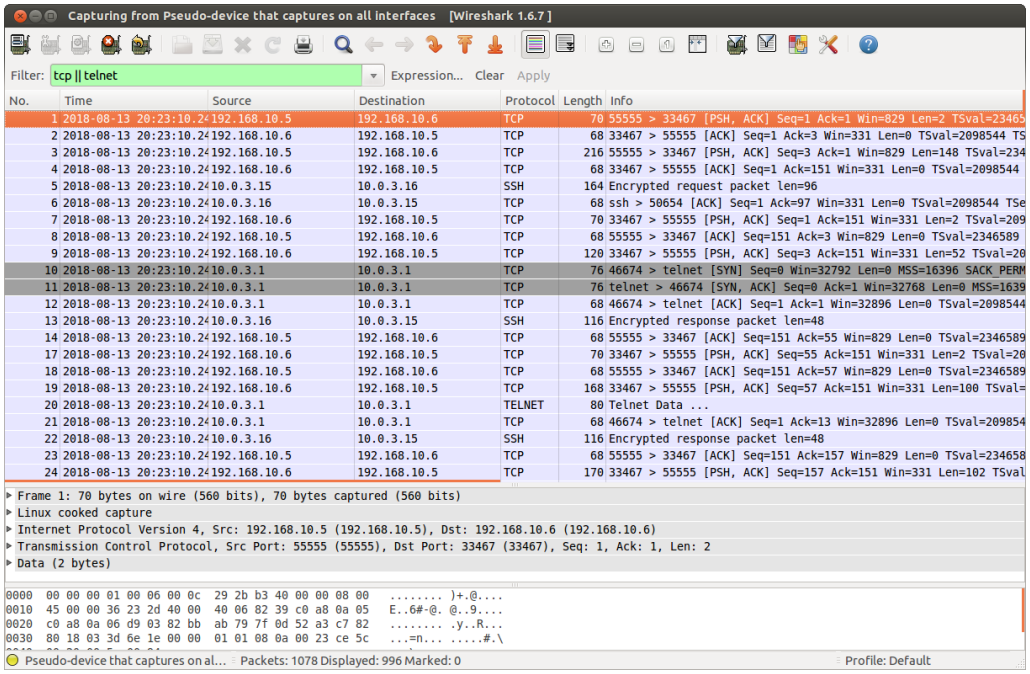


Figure 15: Packet Flow in Wireshark

4 Appendix A

```c
/**************************************************************************
 * simpletun.c                                                            *
 *                                                                        *
 * A simplistic, simple-minded, naive tunnelling program using tun/tap    *
 * interfaces and TCP. Handles (badly) IPv4 for tun, ARP and IPv4 for     *
 * tap. DO NOT USE THIS PROGRAM FOR SERIOUS PURPOSES.                     *
 *                                                                        *
 * You have been warned.                                                  *
 *                                                                        *
 * (C) 2009 Davide Brini.                                                 *
 *                                                                        *
 * DISCLAIMER AND WARNING: this is all work in progress. The code is      *
 * ugly, the algorithms are naive, error checking and input validation    *
 * are very basic, and of course there can be bugs. If that's not enough, *
 * the program has not been thoroughly tested, so it might even fail at   *
 * the few simple things it should be supposed to do right.               *
 * Needless to say, I take no responsibility whatsoever for what the      *
 * program might do. The program has been written mostly for learning     *
 * purposes, and can be used in the hope that is useful, but everything   *
 * is to be taken "as is" and without any kind of warranty, implicit or   *
 * explicit. See the file LICENSE for further details.                    *
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <linux/if.h>
#include <linux/if_tun.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <arpa/inet.h>
#include <sys/select.h>
#include <sys/time.h>
#include <errno.h>
#include <stdarg.h>

/* buffer for reading from tun/tap interface, must be >= 1500 */
#define BUFSIZE 2000
#define CLIENT 0
#define SERVER 1
#define PORT 55555

/* some common lengths */
#define IP_HDR_LEN 20
#define ETH_HDR_LEN 14
#define ARP_PKT_LEN 28
```

```
51
52  int debug;
53  char *progname;
54
55  /**************************************************************************
56   * tun_alloc: allocates or reconnects to a tun/tap device. The caller     *
57   *            needs to reserve enough space in *dev.                      *
58   **************************************************************************/
59  int tun_alloc(char *dev, int flags) {
60
61    struct ifreq ifr;
62    int fd, err;
63
64    if( (fd = open("/dev/net/tun", O_RDWR)) < 0 ) {
65      perror("Opening /dev/net/tun");
66      return fd;
67    }
68
69    memset(&ifr, 0, sizeof(ifr));
70
71    ifr.ifr_flags = flags;
72
73    if (*dev) {
74      strncpy(ifr.ifr_name, dev, IFNAMSIZ);
75    }
76
77    if( (err = ioctl(fd, TUNSETIFF, (void *)&ifr)) < 0 ) {
78      perror("ioctl(TUNSETIFF)");
79      close(fd);
80      return err;
81    }
82
83    strcpy(dev, ifr.ifr_name);
84
85    return fd;
86  }
87
88  /**************************************************************************
89   * cread: read routine that checks for errors and exits if an error is    *
90   *        returned.                                                       *
91   **************************************************************************/
92  int cread(int fd, char *buf, int n){
93
94    int nread;
95
96    if((nread=read(fd, buf, n))<0){
97      perror("Reading data");
98      exit(1);
99    }
100   return nread;
101 }
102
```

```
103 /**************************************************************************
104  * cwrite: write routine that checks for errors and exits if an error is  *
105  *         returned.                                                      *
106  **************************************************************************/
107 int cwrite(int fd, char *buf, int n){
108 
109   int nwrite;
110 
111   if((nwrite=write(fd, buf, n))<0){
112     perror("Writing data");
113     exit(1);
114   }
115   return nwrite;
116 }
117 
118 /**************************************************************************
119  * read_n: ensures we read exactly n bytes, and puts those into "buf".    *
120  *         (unless EOF, of course)                                        *
121  **************************************************************************/
122 int read_n(int fd, char *buf, int n) {
123 
124   int nread, left = n;
125 
126   while(left > 0) {
127     if ((nread = cread(fd, buf, left))==0){
128       return 0 ;
129     }else {
130       left -= nread;
131       buf += nread;
132     }
133   }
134   return n;
135 }
136 
137 /**************************************************************************
138  * do_debug: prints debugging stuff (doh!)                                *
139  **************************************************************************/
140 void do_debug(char *msg, ...){
141 
142   va_list argp;
143 
144   if(debug){
145         va_start(argp, msg);
146         vfprintf(stderr, msg, argp);
147         va_end(argp);
148   }
149 }
150 
151 /**************************************************************************
152  * my_err: prints custom error messages on stderr.                        *
153  **************************************************************************/
154 void my_err(char *msg, ...) {
```

```
155
156   va_list argp;
157
158   va_start(argp, msg);
159   vfprintf(stderr, msg, argp);
160   va_end(argp);
161 }
162
163 /**********************************************************************
164  * usage: prints usage and exits.                                     *
165  **********************************************************************/
166 void usage(void) {
167   fprintf(stderr, "Usage:\n");
168   fprintf(stderr, "%s -i <ifacename> [-s|-c <serverIP>] [-p <port>] [-u|-a]
      ↪  [-d]\n", progname);
169   fprintf(stderr, "%s -h\n", progname);
170   fprintf(stderr, "\n");
171   fprintf(stderr, "-i <ifacename>: Name of interface to use (mandatory)\n");
172   fprintf(stderr, "-s|-c <serverIP>: run in server mode (-s), or specify server
      ↪  address (-c <serverIP>) (mandatory)\n");
173   fprintf(stderr, "-p <port>: port to listen on (if run in server mode) or to
      ↪  connect to (in client mode), default 55555\n");
174   fprintf(stderr, "-u|-a: use TUN (-u, default) or TAP (-a)\n");
175   fprintf(stderr, "-d: outputs debug information while running\n");
176   fprintf(stderr, "-h: prints this help text\n");
177   exit(1);
178 }
179
180 int main(int argc, char *argv[]) {
181
182   int tap_fd, option;
183   int flags = IFF_TUN;
184   char if_name[IFNAMSIZ] = "";
185   int header_len = IP_HDR_LEN;
186   int maxfd;
187   uint16_t nread, nwrite, plength;
188 //  uint16_t total_len, ethertype;
189   char buffer[BUFSIZE];
190   struct sockaddr_in local, remote;
191   char remote_ip[16] = "";
192   unsigned short int port = PORT;
193   int sock_fd, net_fd, optval = 1;
194   socklen_t remotelen;
195   int cliserv = -1;    /* must be specified on cmd line */
196   unsigned long int tap2net = 0, net2tap = 0;
197
198   progname = argv[0];
199
200   /* Check command line options */
201   while((option = getopt(argc, argv, "i:sc:p:uahd")) > 0){
202     switch(option) {
203       case 'd':
```

```
204           debug = 1;
205           break;
206         case 'h':
207           usage();
208           break;
209         case 'i':
210           strncpy(if_name,optarg,IFNAMSIZ-1);
211           break;
212         case 's':
213           cliserv = SERVER;
214           break;
215         case 'c':
216           cliserv = CLIENT;
217           strncpy(remote_ip,optarg,15);
218           break;
219         case 'p':
220           port = atoi(optarg);
221           break;
222         case 'u':
223           flags = IFF_TUN;
224           break;
225         case 'a':
226           flags = IFF_TAP;
227           header_len = ETH_HDR_LEN;
228           break;
229         default:
230           my_err("Unknown option %c\n", option);
231           usage();
232       }
233     }
234
235     argv += optind;
236     argc -= optind;
237
238     if(argc > 0){
239       my_err("Too many options!\n");
240       usage();
241     }
242
243     if(*if_name == '\0'){
244       my_err("Must specify interface name!\n");
245       usage();
246     }else if(cliserv < 0){
247       my_err("Must specify client or server mode!\n");
248       usage();
249     }else if((cliserv == CLIENT)&&(*remote_ip == '\0')){
250       my_err("Must specify server address!\n");
251       usage();
252     }
253
254     /* initialize tun/tap interface */
255     if ( (tap_fd = tun_alloc(if_name, flags | IFF_NO_PI)) < 0 ) {
```

```
256       my_err("Error connecting to tun/tap interface %s!\n", if_name);
257       exit(1);
258     }
259
260     do_debug("Successfully connected to interface %s\n", if_name);
261
262     if ( (sock_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
263       perror("socket()");
264       exit(1);
265     }
266
267     if(cliserv==CLIENT){
268       /* Client, try to connect to server */
269
270       /* assign the destination address */
271       memset(&remote, 0, sizeof(remote));
272       remote.sin_family = AF_INET;
273       remote.sin_addr.s_addr = inet_addr(remote_ip);
274       remote.sin_port = htons(port);
275
276       /* connection request */
277       if (connect(sock_fd, (struct sockaddr*) &remote, sizeof(remote)) < 0){
278         perror("connect()");
279         exit(1);
280       }
281
282       net_fd = sock_fd;
283       do_debug("CLIENT: Connected to server %s\n", inet_ntoa(remote.sin_addr));
284
285     } else {
286       /* Server, wait for connections */
287
288       /* avoid EADDRINUSE error on bind() */
289       if(setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR, (char *)&optval,
          ↪  sizeof(optval)) < 0){
290         perror("setsockopt()");
291         exit(1);
292       }
293
294       memset(&local, 0, sizeof(local));
295       local.sin_family = AF_INET;
296       local.sin_addr.s_addr = htonl(INADDR_ANY);
297       local.sin_port = htons(port);
298       if (bind(sock_fd, (struct sockaddr*) &local, sizeof(local)) < 0){
299         perror("bind()");
300         exit(1);
301       }
302
303       if (listen(sock_fd, 5) < 0){
304         perror("listen()");
305         exit(1);
306       }
```

```
307
308       /* wait for connection request */
309       remotelen = sizeof(remote);
310       memset(&remote, 0, remotelen);
311       if ((net_fd = accept(sock_fd, (struct sockaddr*)&remote, &remotelen)) < 0){
312         perror("accept()");
313         exit(1);
314       }
315
316       do_debug("SERVER: Client connected from %s\n", inet_ntoa(remote.sin_addr));
317     }
318
319     /* use select() to handle two descriptors at once */
320     maxfd = (tap_fd > net_fd)?tap_fd:net_fd;
321
322     while(1) {
323       int ret;
324       fd_set rd_set;
325
326       FD_ZERO(&rd_set);
327       FD_SET(tap_fd, &rd_set); FD_SET(net_fd, &rd_set);
328
329       ret = select(maxfd + 1, &rd_set, NULL, NULL, NULL);
330
331       if (ret < 0 && errno == EINTR){
332         continue;
333       }
334
335       if (ret < 0) {
336         perror("select()");
337         exit(1);
338       }
339
340       if(FD_ISSET(tap_fd, &rd_set)){
341         /* data from tun/tap: just read it and write it to the network */
342
343         nread = cread(tap_fd, buffer, BUFSIZE);
344
345         tap2net++;
346         do_debug("TAP2NET %lu: Read %d bytes from the tap interface\n", tap2net,
           ↪  nread);
347
348         /* write length + packet */
349         plength = htons(nread);
350         nwrite = cwrite(net_fd, (char *)&plength, sizeof(plength));
351         nwrite = cwrite(net_fd, buffer, nread);
352
353         do_debug("TAP2NET %lu: Written %d bytes to the network\n", tap2net, nwrite);
354       }
355
356       if(FD_ISSET(net_fd, &rd_set)){
357         /* data from the network: read it, and write it to the tun/tap interface.
```

```
358           * We need to read the length first, and then the packet */
359
360         /* Read length */
361         nread = read_n(net_fd, (char *)&plength, sizeof(plength));
362         if(nread == 0) {
363           /* ctrl-c at the other end */
364           break;
365         }
366
367         net2tap++;
368
369         /* read packet */
370         nread = read_n(net_fd, buffer, ntohs(plength));
371         do_debug("NET2TAP %lu: Read %d bytes from the network\n", net2tap, nread);
372
373         /* now buffer[] contains a full packet or frame, write it into the tun/tap
            ↪  interface */
374         nwrite = cwrite(tap_fd, buffer, nread);
375         do_debug("NET2TAP %lu: Written %d bytes to the tap interface\n", net2tap,
            ↪  nwrite);
376       }
377     }
378
379     return(0);
380   }
```