Chapter 9

Tunneling and Firewall Evasion

Contents

9.1 Introduction

There are situations where firewalls are too restrictive, making it inconvenient for users. For example, many companies and schools enforce egress filtering, which blocks users inside of their networks from reaching out to certain websites or Internet services, such as game and social network sites. There are many ways to evade firewalls. A typical approach is to use the tunneling technique, which hides the real purposes of network traffic. There are a number of ways to establish tunnels. The two most common tunneling techniques are Virtual Private Network (VPN) and port forwarding. In this chapter, we study both types of tunnels, and demonstrate how to use them to evade firewalls.

9.1.1 The General Ideas of Tunneling

The general idea of using a tunnel to evade firewall is simple. A tunnel is a communication channel between two programs, one on each side of the firewall. This communication channel is not blocked by the firewall. An application needs to communicate with its destination on the other side of the firewall, but the communication is blocked. To evade the firewall, the application sends its data or packets to the tunnel program at the same side of the firewall. The tunnel program then puts the data/packets inside its own communication channel (allowed by the firewall), and then sends to its counterpart at the other end of the firewall, which then releases the data/packets towards their final destinations. The return data/packets will follow the reverse path. Figure 9.1 depicts the entire process.

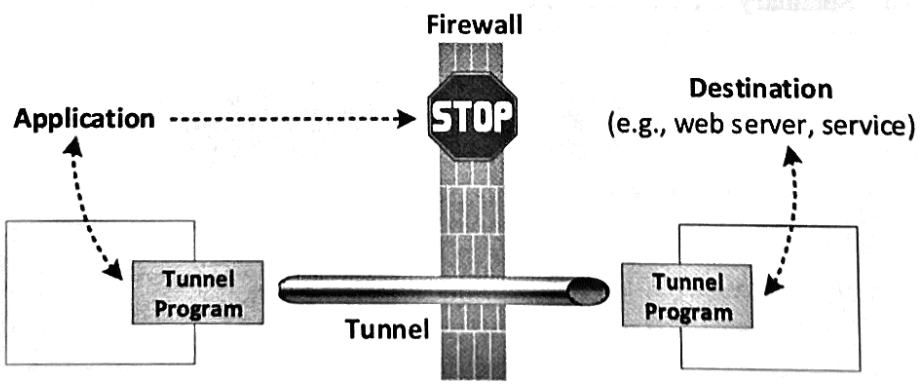

Figure 9.1: The general idea of tunneling

The firewall could be an ingress or egress type. If it is an ingress firewall, the application is on the outside; if it is an egress firewall, the application is on the inside. The way how the tunneling works is similar in both cases. Therefore, the tunneling technique can be used to evade both types of firewalls.

The key difference among different types of tunnel is how the data or packet from the application reaches the tunnel program (on the same side of the firewall). VPN tunnels are built on top of the network layer. It relies on the routing to get the packet from the application, as well as to release the packets to its final destination. VPN tunnels can also be built on top of the MAC layer, and relies on bridging to get the packets. Routing and bridging are transparent to the application. Port forwarding tunnels are built above the transport layer. For an application to send data to the tunnel program, the application needs to directly communicate with the tunnel

program, and thus this type of tunnel is not transparent. Both types of tunnels have their pros and cons. After we discuss them in details, we will compare them.

9.1.2 Network Setup

We will conduct a series of experiments in this chapter. These experiments need to use several computers in two separate networks. The experiment setup is depicted in Figure 9.2. We will use containers for these machines. Readers can find the container setup file from the website of this book. We will explain the roles for these containers later when they are used in experiments.

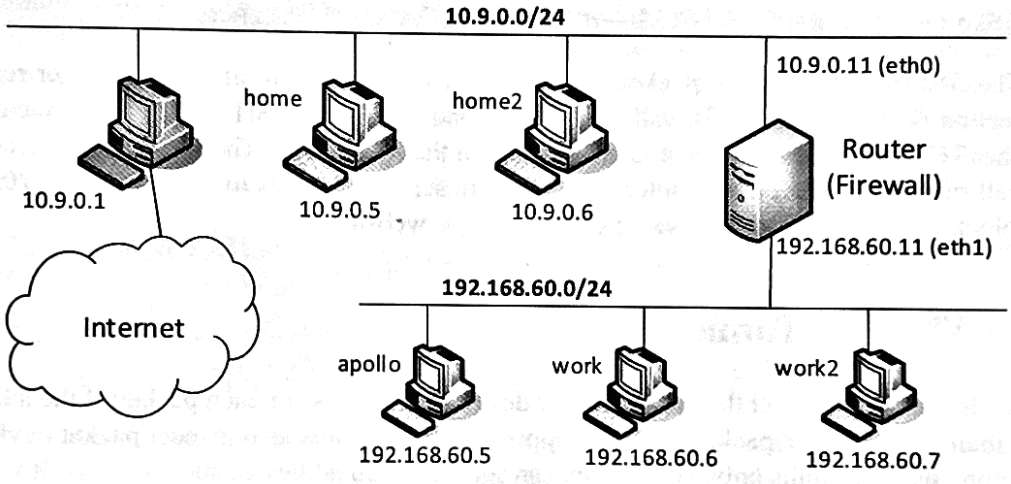

Figure 9.2: Network setup

Router configuration: setting up NAT. The following `iptables` command is included in the router configuration inside the `docker-compose.yml` file. This command sets up a NAT on the router for the traffic going out from its `eth0` interface, except for the packets to `10.9.0.0/24`. With this rule, for packets going out to the Internet, their source IP address will be replaced by the router's IP address `10.9.0.11`. Packets going to `10.9.0.0/24` will not go through NAT.

```
iptables -t nat -A POSTROUTING ! -d 10.9.0.0/24 -j MASQUERADE -o eth0
```

In the above command, we assume that `eth0` is the name assigned to the interface connecting the router to the `10.9.0.0/24` network. This is not guaranteed. The router has two Ethernet interfaces; when the router container is created, the name assigned to this interface might be `eth1`. You can find out the correct interface name using the following command. If the name is not `eth0`, you should make a change to the command above inside the `docker-compose.yml` file, and then restart the containers.

```
# ip -br address
lo               UNKNOWN        127.0.0.1/8
eth1@if1907      UP             192.168.60.11/24
eth0@if1909      UP             10.9.0.11/24
```

Router configuration: Firewall rules. We have also added the following firewall rules on the router. Please make sure that `eth0` is the interface connected to the `10.9.0.0/24` network and that `eth1` is the one connected to `192.168.60.0/24`. If not, make changes accordingly.

```
// Ingress filtering: only allows SSH traffic
iptables -A FORWARD -i eth0 -p tcp -m conntrack \
        --ctstate ESTABLISHED,RELATED -j ACCEPT
iptables -A FORWARD -i eth0 -p tcp --dport 22 -j ACCEPT
iptables -A FORWARD -i eth0 -p tcp -j DROP

// Egress filtering: block www.example.com
iptables -A FORWARD -i eth1 -d 93.184.216.0/24 -j DROP
```

The first rule allows TCP packets to come in if they belong to an established or related connection. This is a stateful firewall rule. The second rule allows SSH, and the third rule drops all other TCP packets if they do not satisfy the first or the second rule. The fourth rule is an egress firewall rule, and it prevents the internal hosts from sending packets to `93.184.216.0/24`, i.e., blocking the access to the `www.example.com` website.

9.2 VPN: IP Tunneling

Firewalls typically inspect the source and/or destination address of each packet; if the address is on their blacklist, the packet will be dropped. Some firewalls also inspect packet payloads. Therefore, these firewalls only work if they can see the actual addresses and payloads. If we can hide those data items, we can bypass firewalls. Virtual Private Network (VPN) becomes a very natural solution for this purpose because it can hide a disallowed packet inside a packet that is allowed. With VPN, one can create a tunnel between two computers on two sides of a firewall. IP packets can be sent using this tunnel. Since the tunnel traffic is encrypted, firewalls cannot see what is inside the tunnel, so they cannot conduct the filtering. Details on how VPN works are given in Chapter 8 (VPN).

9.2.1 Creating VPN Using SSH

There are many ways that we can use to create a VPN tunnel. Chapter 8 shows how to write our own Python program to create a VPN. Its purpose is to help readers understand how VPN works. In this chapter, we will use an existing tool. OpenVPN is a powerful tool that we can use, but in this chapter, we will simply use OpenSSH, a tool for remote login with the SSH protocol. It can also create VPN tunnels. SSH is often called the poor man's VPN. We need to change some default SSH settings on the server to allow VPN creation. The changes made in `/etc/ssh/sshd_config` are listed in the following. They are already enabled inside the containers.

```
PermitRootLogin yes
PermitTunnel    yes
```

To create a VPN tunnel from a client to a server, we run the following `ssh` command. This command creates a TUN interface `tun0` on the VPN client and server machines, and then connect these two TUN interfaces using an encrypted TCP connection. Both zeros in option `0:0` means `tun0`. Detailed explanation of the `-w` option can be found in the manual of SSH.

```
# ssh -w 0:0 root@vpn-server
```

This command only creates a tunnel; further configuration is needed on both ends of the tunnel. We will discuss the configuration part when we create an actual VPN tunnel later. It should also be noted that creating TUN interfaces requires the root privilege, so we need to have the root privilege on both ends of the tunnel. That is why we run it inside the root account, and also SSH into the root account on the server.

9.2.2 Using VPN to Bypass Ingress Filtering

We show how to use VPN to bypass ingress filtering. Assume that we are working in a company, and we need to telnet to a machine called `work`. Sometimes, we need to work from home, so it is necessary to telnet from machine `home` to `work`. However, the company's firewall blocks all incoming TCP packets, unless (1) they are for SSH (port 22), or (2) they belong to an existing TCP connection. In our network setup, we have already set up the firewall rules. We can verify these rules by running the following command on the `home` machine (see Figure 9.2).

```
# telnet 192.168.60.6
Trying 192.168.60.6...

# telnet 192.168.60.7
Trying 192.168.60.7...
```

From outside, we are trying to telnet to the machines on the internal network protected by the firewall. We can see that the connections got blocked. To bypass the firewall rules, we create a VPN tunnel between the `home` machine outside and the `apollo` machine inside, and tunnel all our TCP packets through this VPN. See Figure 9.3. We run the following SSH command on the `home` machine (`192.168.60.5` is the IP address of `apollo`):

```
# ssh -w 0:0 root@192.168.60.5 \
      -o "PermitLocalCommand=yes" \
      -o "LocalCommand= ip addr add 192.168.53.88/24 dev tun0 && \
                        ip link set tun0 up" \
      -o "RemoteCommand=ip addr add 192.168.53.99/24 dev tun0 && \
                        ip link set tun0 up"
root@192.168.60.5's password: ****  ← Password:  dees
```

The `LocalCommand` entry specifies the command running on the VPN client side (i.e., on `home`). It configures the client-side TUN interface: assigning the `192.168.53.88/24` address to the interface and bringing it up. The `RemoteCommand` entry specifies the command running on the VPN server side (i.e., on `apollo`). It configures the server-side TUN interface.

After the tunnel has been established, on the VPN client (`home`), we need to re-route all the `192.168.60.0/24`-bound traffic towards `tun0`. This way, the packets going to this destination will go through the tunnel, not through the router `10.9.0.11`. However, we do need to make an exception for the packets to `192.168.60.5`, because this is the VPN server's address, and we do not want to route the traffic to this address towards `tun0`. Packets to this destination should still go through the router, but since we will only access port `22` on this destination, the packets will not be blocked.

```
# ip route replace 192.168.60.0/24 dev tun0  ← reroute to TUN
# ip route add 192.168.60.5/32 via 10.9.0.11 ← do not reroute
```

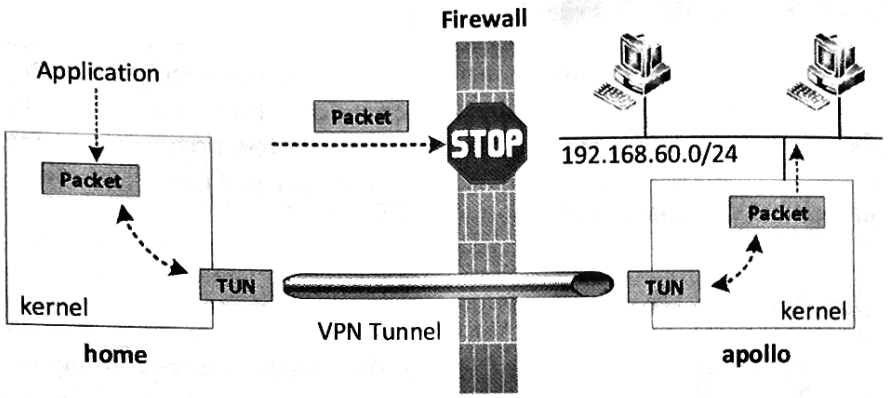

Figure 9.3: Bypassing firewall using VPN tunnel

On the VPN server (`apollo`), we need to set a NAT server, so packets going out of the `eth0` interface will take this interface's IP address. This is important for the traffic coming from the TUN interface, because their source IP addresses will be `192.168.53.88`. When they get routed out to the `192.168.60.0/24` network via `eth0`, if the source IP address is preserved, the reply from the destination will be sent to `192.168.53.88`, which will not come to the VPN server, unless we change the routing tables on all the hosts on the `192.168.60.0/24` network. Setting this NAT ensures that the reply will come to the VPN server without changing the routing tables.

```
# iptables -t nat -A POSTROUTING -j MASQUERADE -o eth0
```

On the VPN server, we also need to have IP forwarding enabled. The VPN server is just a proxy; when it receives a packet from the VPN tunnel, it needs to release the packet towards the final destination. If the IP forwarding is not enabled, the packet will be dropped, not be forwarded. IP forwarding is already enabled on the `apollo` container.

Testing. Once everything is set up, we can try to telnet to a host on `192.168.60.0/24` network (other than the host `192.168.60.5`). From the results, we can see that the connection is successful, and we have successfully bypassed the firewall.

```
# telnet 192.168.60.7
Trying 192.168.60.7...
Connected to 192.168.60.7.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
32ee031be16b login:
```

If we turn on `tcpdump` on the router, we will only see the SSH traffic between the VPN client and server. We will not be able see the telnet packets, because they are put inside the payload of the SSH packets, encrypted.

```
# tcpdump -n -i eth0
21:03:32.287328 IP 10.9.0.5.44432 > 192.168.60.5.22: ...
21:03:32.287405 IP 192.168.60.5.22 > 10.9.0.5.44432: ...
```

```
21:03:32.287559 IP 10.9.0.5.44432 > 192.168.60.5.22: ...
21:03:32.287405 IP 192.168.60.5.22 > 10.9.0.5.44432: ...
```

9.2.3 Using VPN to Bypass Egress Firewall

Many organizations or countries conduct egress filtering on their firewalls to prevent their internal users from accessing certain websites, for a variety of reasons, including discipline, safety, and politics. For instance, many K-12 schools in the United States block social network sites, such as Facebook, from their networks, so students do not get distracted during school hours. Another example is the "Great Firewall of China", which blocks a number of popular sites, including Google, YouTube and Facebook.

This type of firewall typically checks the destination IP address of the packet. Using VPN, we can create a tunnel with a VPN server outside of the firewall, and send our packet to the server via the tunnel. The VPN server will release our packets to the Internet from its end, which is outside of the firewall. Although the tunnel traffic still goes through the firewall, what the firewall sees is only the VPN server's IP address, our proxy, not our true destination.

In our network setup, we have added a rule on the firewall to block the hosts on the `192.168.60.0/24` network from accessing the `example.com` website. We will show how to use VPN to bypass this firewall rule. We set up a VPN between `apollo` and `home`, but this time, we use `apollo` as the VPN client and `home` as the VPN server. Again, we use SSH to establish such a VPN tunnel. We run the following commands on `apollo` (VPN client), and the other end of the tunnel (VPN server) is `10.9.0.5`.

```
# ssh -w 0:0 root@10.9.0.5 \
      -o "PermitLocalCommand=yes" \
      -o "LocalCommand= ip addr add 192.168.53.88/24 dev tun0 && \
                        ip link set tun0 up" \
      -o "RemoteCommand=ip addr add 192.168.53.99/24 dev tun0 && \
                        ip link set tun0 up"
```

Once the VPN tunnel is set up, on the VPN client (`apollo`), we need to redirect the `example.com`-bound traffic to the TUN interface; otherwise, the traffic will be routed to `192.168.60.11` (router/firewall) and gets dropped by the firewall.

```
# ip route add 93.184.216.0/24 dev tun0
```

After the above setup, if we go visit `example.com` from `apollo`, we can see that packets are now going through our tunnel, and can successfully reach VPN server. Although the tunnel traffic still goes through the router/firewall, the firewall does not block these packets because their destination is VPN server, not `example.com`. The following `ping` command shows that our packets have successfully passed the firewall, but nothing has come back.

```
# ping www.example.com
...
```

From the wireshark trace, we can see that our VPN server has actually forwarded the packets to `example.com`, which has also replied, but the reply never reaches the VPN server. The reason is the source IP address of the packet. As we have discussed before, when a packet is sent out via the VPN tunnel from the user machine, the packet takes the TUN interface's IP address as its source IP address, which is `192.168.53.88` in our setup. When this packet is

sent out by the VPN server, it will go through VirtualBox's NAT (Network Address Translation) server, where the source IP address will be replaced by the IP address of the host computer. The packet will eventually arrive at `example.com`, and the reply packet will come back to our host computer, and then be given to the same NAT server, where the destination address is translated back `192.168.53.88`. This is where the problem comes up.

VirtualBox's NAT server knows nothing about the `192.168.53.0/24` network, because this is the one that we create internally for our TUN interface, and VirtualBox has no idea how to route to this network, much less knowing that the packet should be given to VPN server. As a result, the reply packet from `example.com` will be dropped. That is why we do not see anything back from the VPN tunnel.

To solve this problem, we will set up our own NAT server on VPN server, so when packets from `192.168.53.88` go out, their source IP addresses are always replaced by the VPN server's IP address (`10.9.0.5`). We can use the following command to create a NAT server on the `eth0` interface of the VPN server (the `home` machine).

```
# iptables -t nat -A POSTROUTING -j MASQUERADE -o eth0
```

Once the NAT server is set up, we should be able to connect to `www.example.com` from the user machine. The following results show that we successfully bypassed the firewall.

```
# ping www.example.com
PING www.example.com (93.184.216.34) 56(84) bytes of data.
64 bytes from 93.184.216.34 (93.184.216.34): icmp_seq=1 ttl=53 ...
64 bytes from 93.184.216.34 (93.184.216.34): icmp_seq=2 ttl=53 ...
64 bytes from 93.184.216.34 (93.184.216.34): icmp_seq=3 ttl=53 ...

# curl www.example.com
<!doctype html>
<html>
<head>
    <title>Example Domain</title>
...
</html>
```

For other hosts. If we want to also allow the other hosts on `192.168.60.0/24` to bypass firewall, we should reroute their `example.com`-bound traffic to `apollo`, so they can enter the VPN tunnel. We run the following commands on all the hosts inside `192.168.60.0/24`.

```
# ip route add 93.184.216.0/24 via 192.168.60.5
```

We may also want to run a NAT server on the TUN interface of `apollo`, so all the packets going through this interface will have their source IP addresses changed to the TUN interface's IP address. Without doing this, the return packets will not come through the tunnel. Instead, they will be directly given by the host VM to the hosts on the `192.168.60.0/24`, because the host VM is also attached to this network. Although this does not affect our experiment outcome, it does make the returning traffic look abnormal.

```
# iptables -t nat -A POSTROUTING -j MASQUERADE -o tun0
```

Discussion. In the real world, once a VPN server is identified as being used for bypassing firewalls, it will be put on the blacklist. Therefore, VPN servers used for firewall-bypassing purposes need to change their addresses constantly, so if one server gets blocked, another one will come up with a different IP address. This is a like a never-ending cat-and-mouse game.

9.2.4 Bypassing Geo-Restriction

Geo-restriction is a common type of firewall, which is used to decide whether to provide services or not based on the geolocation of the IP address of the request. This is quite common for media service providers, such as YouTube, Netflix, etc., because of the licenses of the content. For example, when Netflix purchases the license for some content, the license may only allow Netflix to stream the content within the US. Therefore, Netflix has to set up the geo-restriction to block the accesses to these contents if the accesses are not from within the US.

Another example is the College Board. Every year, when the Advanced Placement (AP) scores are made available, to prevent everybody from accessing the scores on the same day and overload the server, the College Board spreads the score releases over a few days. When students can see their scores depends on their geographic regions. For example, in 2016 all the states on the east coast got their scores first, whereas those in the northwest got theirs last. However, the checking is not based on where you took the exam, but based on where you are when you check the score, i.e., based on the geolocation of the source IP address of the packet.

In the past, students who were eager to learn their scores had to ask their friends in another region to check the scores for them. This requires students to give the login credentials to their friends, who can not only learn the scores, but can also look at all the other sensitive information inside the account. This is actually one type of proxy, human proxy.

These days, many students have learned to use a VPN server in another region as their proxy. When you connect to a VPN server, your packets will be sent to the VPN server, from where, they will be released to the Internet. The source IP address of the packets is not your machine's IP address; instead, it belongs to the VPN server's network. When College Board checks the geolocation of the IP address, it will be the geolocation of the VPN server. Therefore, if only the states on the east coast can get their scores today, but you do not live there, you can find a VPN server on the east coast, and use it to check your score. The communication between you and the College Board is encrypted end to end, so the VPN server in the middle will not be able to see your scores. This is much better than the human proxy.

This represents another important application of VPN: hiding the sender's real IP address. The same idea has also been used by users who want to watch videos from Netflix and YouTube that are not available to their regions.

9.3 Port Forwarding: SSH Tunneling

Running VPN requires the root privilege, because creating the TUN interface and changing the routing table both require the root privilege. If this is not allowed, we can use another type of tunneling, port forwarding. This tunnel works above the transport layer, so it does not require the root privilege. The tradeoff is that it becomes less transparent compared to VPN. In this section, we will use SSH to establish a port forwarding to bypass ingress and egress filtering.

9.3.1 Port Forwarding: Evading Ingress Firewall

To evade the ingress firewall, we run the following command on `home`. This establishes a port forwarding SSH tunnel between `home` and `apollo`, forwarding the traffic at port `8000` (on `home`) to port `23` (on `192.168.60.6`), through the SSH tunnel.

```
home$ ssh -4NT -L 8000:192.168.60.6:23 seed@192.168.60.5
                  ↗    ---------------     ------------
               home        work                apollo

// -4: use IPv4 only, or we will see some error message.
// -N: do not execute a remote command.
// -T: disable pseudo-terminal allocation (save resources).
```

Once the tunnel is established, if we want to telnet to `work` through the tunnel, we need to telnet to the port `8000` on the `home` machine; otherwise the telnet traffic will not go through the tunnel. See the following command.

```
// On home
home$ telnet localhost 8000
```

The way how the port forwarding works is depicted in Figure 9.4. On the `home` end, at port `8000`, SSH receives the TCP packets from the `telnet` client. It forwards the TCP data to `apollo` through the SSH tunnel. Once the data reaches the other end, it is put in another TCP packet, which is sent towards the machine `work`. The entire process consists of three separate TCP connections (see ① - ③ in the figure). The company firewall only sees the `ssh` traffic from `home` to `apollo`, not the `telnet` traffic from `apollo` to `work`. Moreover, the `ssh` traffic is encrypted, so the firewall will not be able to see what is inside.

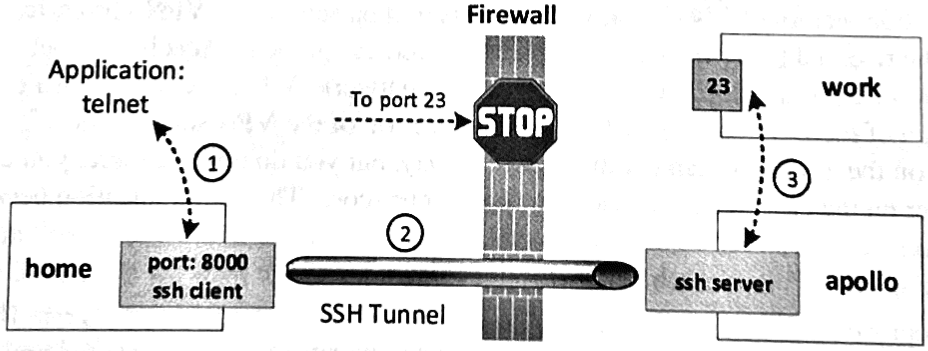

Figure 9.4: Evade firewall using SSH port forwarding tunnel

Using the SSH tunnel from another host. If we are on another machine (e.g., `home2`), and want to use the SSH tunnel on `home`, we will see the following error message:

```
home2$ telnet 10.9.0.5 8000
Trying 10.9.0.5...
telnet: Unable to connect to remote host: Connection refused
```

This is because in our setup command, we did not put a hostname before the port number 8000, so a default hostname, localhost, is used. Namely, the port forwarding is actually between localhost:8000 and 192.168.60.6:23. In this setup, SSH only listens to port 8000 on the loopback interface, so the connections from other interfaces will not reach this port. We can see this from the netstat command.

```
home$ netstat -nat
Active Internet connections (servers and established)
Proto  Recv-Q  Send-Q  Local Address     Foreign Address    State
tcp       0       0     127.0.0.1:8000    0.0.0.0:*          LISTEN
```

To solve this problem, we just need to modify the SSH command. This time, we put 0.0.0.0 in front of the port number 8000, indicating that our port forwarding will listen to the connection from all the interfaces. We can verify that using the netstat command. After this, the tunnel can now be used by another machine to reach the telnet server on work.

```
home$ ssh -4NT -L 0.0.0.0:8000:192.168.60.6:23  seed@192.168.60.5
                  ------------ ---------------      ------------
                        home          work              apollo

apollo$ netstat -nat
Active Internet connections (servers and established)
Proto  Recv-Q  Send-Q  Local Address     Foreign Address    State
tcp       0       0     0.0.0.0:8000      0.0.0.0:*          LISTEN
```

Comparison with VPN tunnel. Port forwarding is different from VPN. In port forwarding, the telnet application only establishes a connection with the home machine. To the telnet client, it is only communicating with home. To the telnet server on work, it sees that the client is on apollo, which is only a proxy, not the actual client. Namely, the telnet client talks to one end of the tunnel, while the telnet server talks to the other end. The tunnel is not transparent to the application. The client has to change its behavior: instead of sending packets directly to the server, it has to send packets to a middle man.

In VPN, the telnet application directly communicates with the work machine, not through a middle man. However, their packets are indeed routed to a middle man, i.e., the VPN tunnel, but routing occurs at the network layer, transparent to applications. Neither telnet client nor server is even aware of the existence of the tunnel. Therefore, the VPN tunnel is transparent to the client and server applications. This is the main difference between VPN and port forwarding.

Evading egress firewalls. The example shown above evade an ingress filter firewall. The same technique can be used to evade egress filtering. Assume that this time, we are inside the company, working on work or apollo. We would like to visit example.com, but the company has blocked it. We will use an outside machine called home to bypass such a firewall. This time, we establish an SSH tunnel from apollo to home, opposite to the direction in the previous example..

```
apollo$ ssh -4NT -L 8000:www.example.com:80  seed@10.9.0.5
                  ↗  ------------------      --------
              apollo    target website          home
```

To test the tunnel, we need to visit `www.example.com` from a browser. Since we cannot run a browser inside a container, we will use a command-line "browser" called `curl`. We run the following command on `apollo`. The `curl` program will sends its HTTP request to the proxy `localhost:8000`, which forwards the HTTP request to the other end of the SSH tunnel on `home`, and `home` will forward the request to the final destination `www.example.com`. The responses will come back in the reverse direction. The firewall only sees the SSH traffic between `apollo` and `home`, not the actual web traffic between `apollo` and `example.com`.

```
apollo$ curl --proxy localhost:8000 http://www.example.com
<!doctype html>
<html>
<head>
    <title>Example Domain</title>
    ....
```

Using the SSH tunnel from another host. If we want to use the SSH tunnel from another machine (e.g., `work` or `work2`) to reach the blocked site, just like what we have discussed earlier, we need to modify our command. See the following:

```
apollo$ ssh -4NT -L 0.0.0.0:8000:www.example.com:80  seed@10.9.0.5
                    ------------ ------------------      --------
                      apollo      target website           home
```

After that, we can use the IP address of `apollo` as our proxy. See the following:

```
work$ curl --proxy 192.168.60.5:8000 http://www.example.com
```

9.3.2 Reverse SSH Tunneling

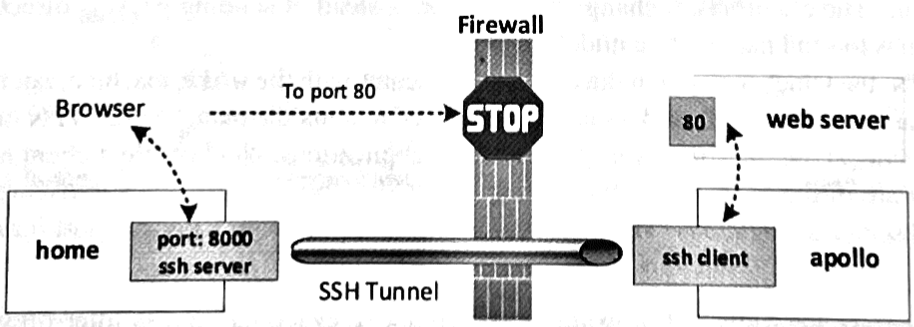

Figure 9.5: Use reverse SSH tunneling to access an internal web server

In the previous examples, the direction of the port forwarding is from SSH client to SSH server. Therefore, if we want to port forward from side A to side B, we need to SSH from side A to side B. What if SSH from A to B is blocked by firewalls, can we still port forward from A to B?

For example, assume that we have an internal website that nobody can access from outside, either because it uses a private IP address or the firewall blocks the access. Our goal is to evade

the firewall rule and allow others to access this internal website from the outside. We also assume that the incoming SSH traffic is also blocked; otherwise, we can create a SSH tunnel from outside, and use this tunnel to do port forwarding. We do assume that firewall does not prevent us from establishing a connection with outside servers.

We can create a reverse SSH tunnel, which allows us do SSH from A to B, but port forwarding traffic from B to A. It is called reverse because the direction of SSH is different from that of port forwarding. We can run the following command to set up a reverse SSH tunnel from the inside machine `apollo`. We assume that the web server is on `192.168.60.6`.

```
apollo$ ssh -4NT -R 8000:192.168.60.6:80 seed@10.9.0.5
                  ↗  ---------------     --------
               home        web server          home
```

The above command creates a reverse SSH tunnel from `apollo` using the `-R` option, so when users send an HTTP request to the port `8000` on machine `home`, the SSH tunnel will forward the request to `apollo`, which further forwards the request to the port `80` of `192.168.60.6`.

By default, in the reverse SSH tunnel, the port `8000` only binds to the loopback interface. This means that we can only use this tunnel from the local host, i.e., `10.9.0.5`. Unlike in the local port forwarding, even if we put `0.0.0.0` before the port number, SSH will not allow us to bind `8000` to all the interfaces. This is a security consideration in the SSH server configuration. We need to change the following option from the default `no` to `clientspecified` in `/etc/ssh/sshd_config` on the SSH server (`home` in this case). This change has already been made in all the containers.

```
GatewayPorts clientspecified
```

After making this configuration change, we can add `0.0.0.0` before the port number. Now, all the machines from the outside can access the web server on `192.168.60.6` via the proxy at `10.9.0.5:8000`. Here is the modified SSH command.

```
apollo$ ssh -4NT -R 0.0.0.0:8000:192.168.60.6:80   seed@10.9.0.5
                  ------------ ---------------      --------
                       home       web server          home
```

9.4 Dynamic Port Forwarding and SOCKS Proxy

In the previous examples, each port-forwarding tunnel forwards the data to a particular destination. If we want to forward data to multiple destinations, we need to set up multiple tunnels. For example, using port forwarding, we can successfully visit the blocked `example.com` website, but what if the firewall blocks many other sites, do we have to tediously establish one SSH tunnel for each site? In this section, we show how to use one port-forwarding tunnel to reach multiple destinations.

9.4.1 Dynamic Port Forwarding

The technique that we used in the previous section is static port forwarding, i.e., the destination of the port forwarding is fixed. The `ssh` program provides another way to do port forwarding, the dynamic port forwarding. We will first get some experience with such type of port forwarding,

and then discuss how it actually works. We set a tunnel between apollo and home, so we can browse any external website from the internal network (we did not block all the websites on the firewall, but let us pretend they are all blocked). We run the following command on apollo.

```
apollo$ ssh -4NT -D 9000 seed@10.9.0.5
                  ↗          --------
          port on apollo       home
```

After the tunnel is set up, we can test it using the curl command. We specify a proxy option, so curl will send its HTTP request to the proxy, which listens on port 9000. The proxy forwards the data received on this port to home via the established ssh tunnel. The type of proxy is called SOCKS, which will be discussed later.

```
// On apollo
$ curl --proxy socks5h://localhost:9000 https://www.google.com
$ curl --proxy socks5h://localhost:9000 https://www.example.com
```

If we want to allow other hosts to use the tunnel/proxy, we should add 0.0.0.0 before the port number, and then provide the proxy's IP address, instead of localhost, when we use the proxy. See the following changes.

```
// Set up the tunnel/proxy
apollo$ ssh -4NT -D 0.0.0.0:9000 seed@10.9.0.5

// Use the proxy from another machine
$ curl --proxy socks5h://192.168.60.5:9000 https://www.example.com
```

9.4.2 Testing the Proxy Using Browser

We can also test the tunnel using a browser. Although it is hard to run a browser inside a container, in the docker setup, by default, the host machine is always attached to any network created inside docker, and the first IP address on that network is assigned to the host machine. For example, in our setup, the host machine is the SEED VM; its IP address on the internal network 192.168.60.0/24 is 192.168.60.1. Therefore, we can test the proxy using Firefox on the VM.

We first need to configure Firefox's proxy setting. Type about:preferences in the URL field (or click Preference menu item). On the General page, find the "Network Settings" section, click the Settings button, and a window will pop up. Follow Figure 9.6 to set up the SOCKS proxy.

Once the proxy is configured, we can then browse any website. The requests and replies will go through the SSH tunnel. Since the host VM can reach the Internet directly, to make sure that our web browsing traffic has gone through the tunnel, we can run tcpdump on the router/firewall. We will be able to see the tunnel traffic every time we browse a website. That indicates that our traffic does go through the proxy/tunnel. If we break the SSH tunnel and then try to browse a website, Firefox will show the following error message: "The proxy server is refusing connections". After the experiment, make sure to check the "No proxy" option to remove the proxy setting from Firefox.

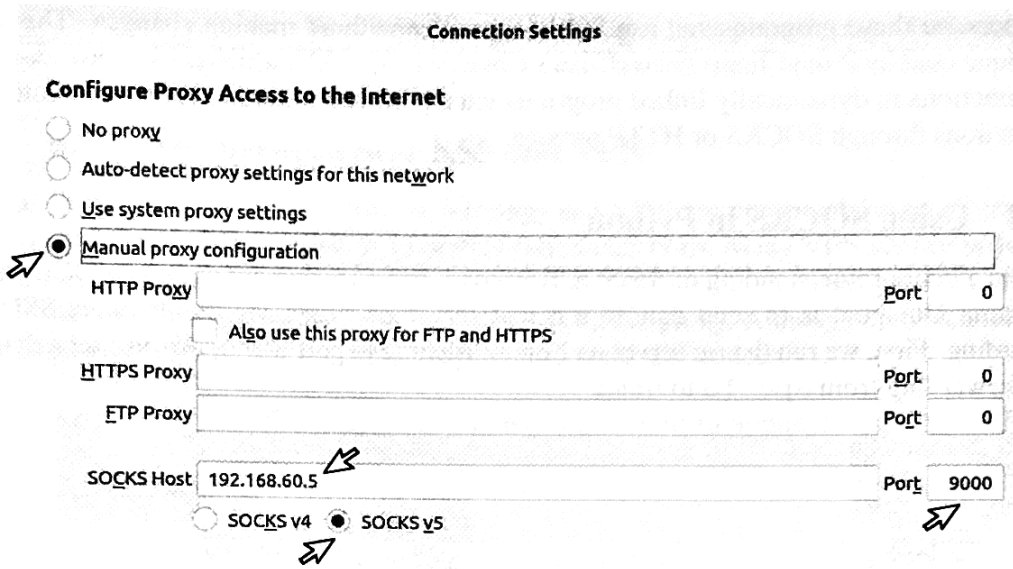

Figure 9.6: Configure the SOCKS Proxy

9.4.3 The SOCKS Protocol

For port forwarding to work, we need to specify where the data should be forwarded to (the final destination). In the static case, this piece of information is provided when we set up the tunnel, i.e., it is hard-wired into the tunnel setup. In the dynamic case, the final destination is dynamic, not specified during the setup, so how can the proxy know where to forward the data?

Applications using a dynamic port forwarding proxy must tell the proxy where to forward their data. This is done through an additional protocol between the application and the proxy. A common protocol for such a purpose is the SOCKS (Socket Secure) protocol, which becomes a de facto proxy standard. The protocol was originally developed/designed by Koblas and Koblas [1992]. It was later extended to version 4, and then version 5. Details of the SOCKS protocol (Version 5) are specified in RFC 1928 [Leech et al., 1996].

The details of the SOCKS protocol is beyond the scope of this book; we will only summarize its key features. In the following, the application is referred to as the client, and the proxy is referred to as the server:

- The client and server conduct handshake. If the server requires authentication, the client will send the credentials during the handshake.
- The client sends the destination information to the server, including the destination IP address or domain name, and the destination port number. Using the destination information, the server (proxy) can now set up the port forwarding.

Since the application needs to interact with the proxy using the SOCKS protocol, the application software must have a native SOCKS support in order to use SOCKS proxies. Both Firefox and `curl` have such a support, but we cannot directly use this type of proxy for the `telnet` program, because `telnet` does not provide a native SOCKS support. If these programs are dynamically linked programs, we can use the function interposition technique to replace their calls to network-related functions in dynamic libraries with calls to user-defined

wrappers, so these programs can use SOCKS proxies without making changes. This is the technique used by ProxyChains [ProxyChains Contributors, 2022], which hooks network-related libc functions in dynamically linked programs via a preloaded shared library, redirecting the connections through SOCKS or HTTP proxies.

9.4.4 Using SOCKS in Python

To gain a better understanding of the SOCKS proxy, let us implement our own SOCKS client program. Our goal is to send data to a netcat server on `home` via the dynamic SSH port forwarding. First, we run the `nc` server on `home`, listening to port `8080`. We also set a dynamic port forwarding from `apollo` to `home`.

```
// On home (10.9.0.5)
home$ nc -lv 8080

// On apollo
apollo$ ssh -4NT -D 9000 seed@10.9.0.5
```

Python has a module called `socks`, which implements the SOCKS protocol. We will use this module to communicate with the SOCKS proxy. The program is listed in the following.

Listing 9.1: The SOCKS client program (`socks_client.py`)

```
#!/bin/env python3

import socks

s = socks.socksocket()
s.set_proxy(socks.SOCKS5, "localhost", 9000)  ①
s.connect(("10.9.0.5", 8080))                 ②

s.sendall(b"hello\n")
s.sendall(b"hello again\n")
print(s.recv(4096))
```

In the program, at Line ①, we first tell the program where the proxy is (`localhost` and port `9000`). We then invoke `s.connect(("10.9.0.5", 8080))` at Line ②, and this is where the following actions are performed:

- A TCP connection is established between our Python program and the proxy, i.e., the SSH process listening at port `9000` on `localhost`.
- Inside the TCP connection, our Python program and the proxy initiate the SOCKS protocol. This is when the Python program tells the proxy that the final destination of the port forwarding is `10.9.0.5:8080`. Therefore, the port forwarding setup is complete.
- After the SOCKS protocol, the SSH proxy will forward the data received from the Python program to the other end of the tunnel, from where the data will be further forwarded to the final destination, i.e., the `nc` server. The response will be forwarded back through the reverse path.

We run the Python program on `apollo`. We can see that the two `hello` messages can be successfully sent to the `nc` server. If we type a message on the `nc` server side, the message will

be sent to our client program. To confirm that the messages have gone through the tunnel, we can break the tunnel and run the client program again. The communication will fail.

9.4.5 Difference Between SOCKS5 and VPN

Both SOCKS5 proxy (dynamic port forwarding) and VPN are commonly used in creating tunnels to bypass firewalls, as well as to protect communications. Many VPN service providers provide both types of services. Sometimes, when a VPN service provider tells you that it provides the VPN service, but in reality, it is just a SOCKS5 proxy. Although both technologies can be used to solve the same problem, they do have some differences.

- *Transparency:* The SOCKS5 proxy is not transparent to applications, while VPN is. This means that to use a SOCKS5 proxy, the application needs to be able to support the SOCKS5 protocol.
- *Setup:* Setting up a VPN is more complicated than setting up a SOCKS5 proxy. It involves installing/running the VPN client program, creating a TUN/TAP interface, and modifying the routing table. Some of these steps require the superuser privilege. For SOCKS5 proxy, we only need to install/run the proxy program, and there is no need for special privileges. It should be noted that VPN's transparency is achieved due to its setup.
- *Application-Specific:* A port-forwarding tunnel established between a client and a proxy using SOCKS5 can only be used by the client, not by other applications. For VPN, once the tunnel is established, all applications can use it.
- *Encryption:* VPN by definition encrypts your traffic, but whether a SOCKS5 proxy encrypts the traffic or not depends on the proxy implementation. If we use SSH for the SOCKS5 proxy, the traffic is encrypted.

9.5 Other Tunneling Methods

Other than the VPN and port forwarding, several other types of tunnels are also used to evade firewalls. HTTP tunneling and ICMP tunneling are two examples. HTTP tunneling disguises the blocked communication between two remote computers as the HTTP traffic, as most firewalls do allow the web traffic to go through. Similarly, ICMP tunneling disguises the blocked communication using ICMP echo request and reply packets. With this type of tunneling, the communication between two remote computers is put inside the payload area of the ICMP echo request/reply packets. As long as ICMP is allowed by the firewall, the communication can get through. These types of tunneling are less generic than VPN or port forwarding, so we will not discuss them in this book.

9.6 Summary

Firewall is not a perfect security solution. There are many ways to bypass firewalls. Tunneling is one of the primary techniques used to bypass firewalls. In this chapter, we study two types of tunneling techniques, VPN and port forwarding. We explain how they work and demonstrate how to use them to evade firewalls. These two techniques are widely used in practice. Their differences are also discussed.

❒ Hands-on Lab Exercise

We have developed a SEED lab for this chapter. The lab is called *Firewall Evasion Lab*, and it is hosted on the SEED website: `https://seedsecuritylabs.org`.

❒ Problems and Resources

The homework problems, slides, and source code for this chapter can be downloaded from the book's website: `https://www.handsonsecurity.net/`.