# Lecture 16: TCP Vulnerabilities and the Denial-of-Service Attacks

## Lecture Notes on "Introduction to Computer Security"

by Avi Kak (kak@purdue.edu)

March 29, 2007



Goals:

- To review the TCP packet header
- TCP State Transition Diagram
- The SYN Flood Attack for Denial of Service
- TCP Vulnerabilities
- IP Spoofing
- Troubleshooting networks with the netstat utility

## TCP and IP

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

- IP stands for the Internet Protocol that deals with assigning addresses to computers and routing packets of data from one computer to another. TCP, on the other hand, deals with establishing reliable connections using the facilities provided by IP. A less reliable version of TCP is UDP (User Datagram Protocol). Despite the pejorative sense given by the phrase "less reliable", UDP is central to the workings of internet communications.

- 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.

## The TCP/IP Protocol Stack

- The four different layers of the TCP/IP protocol stack are

    1. **Application Layer**
       (HTTP, FTP, SMTP, SSH, POP3, TLS/SSL, DNS, etc.)

    2. **Transport Layer**
       (TCP, UDP, etc.)

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

    4. **Link Layer**
       (Ethernet, WiFi, PPP, SLIP, etc.)

  The "Network Layer" is also referred to as the "Internet Layer".

- Note that TCP (Transmission Control Protocol) and IP (Internet Protocol) are two separate protocols in the stack and that the entire stack is commonly referred to as the TCP/IP protocol suite. Of the various protocols shown, TCP and the IP were the first to the developed.

- The layered representation of the protocols shown above is somewhat lacking, especially with regard to the Application Layer. As a case in point, it is not reasonable to think of both HTTP and TLS/SSL at the same level because the former calls on the latter for security. In that sense, HTTP should be placed above TLS/SSL. [The acronym HTTP stands the HyperText Transport Protocol, TLS for the Transport Layer Security, and SSL for Secure Socket Layer.]

- A superior layering of the protocols is provided by the seven-layer OSI (Open System Interconnection) Model that splits the **Application Layer** of the TCP/IP stack into three finer grained layers: **Application Layer**, **Presentation Layer**, and **Session Layer**. In this model, TLS/SSL would belong to the Session Layer, whereas HTTP would stay in the Application Layer. The **Presentation Layer** includes protocols such as the SMB (Samba). (The OSI model also breaks the Link Layer of TCP/IP into two layers: the **Data Link Layer** and the **Physical Layer**. The **ethernet** protocol resides at the Data Link Layer. The **Physical Layer** would be represented by the propagation medium (cables, wireless, etc.) and the associated components like repeaters, amplifiers, etc. *Media Access Control* (MAC) works at the Data Link Layer using CSMA, CDMA, etc., types of algorithms.)

- Another commonly used protocol that does not conveniently fit into the 4-layer TCP/IP model is the ICMP protocol. 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:** 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 sends the ICMP *Source Quench* message back to the sender.

**Assist Troubleshooting:** The ICMP *Echo* messages are used by the commonly used **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.

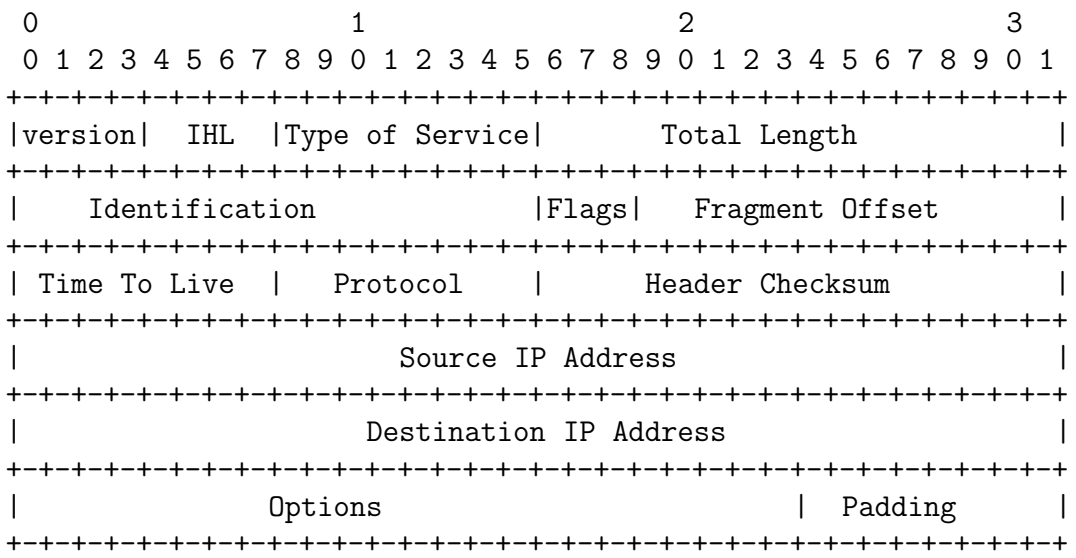
**Announce Timeouts:** When a packet's TTL (Time To Live) drops to zero, the router discarding the packet sends ICMP message back to the sender announcing this fact. (Every IP packet contains a TTL field that is decremented every time the packet passes through a router.)

The ICMP protocol is a bit of a cross between the 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.

- On the transmit side, as each packet descends down the protocol stack, each layer adds its own header to the packet.

## The Network Layer (also known as the Internet Layer or the IP Layer)

- The header added by the IP layer includes information as to which higher level protocol the packet came from. The header added to the packet by the IP layer also includes information on what host the packet is going to, and the host the packet came from. Shown below is the IP Header format (taken from RFC791, dated 1981):




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 the Internet Header Length is the length of the header in 32-bit words. The minimum value for this field is 5 for five 32-bit words.

– The **Type of Service** field (8 bits wide) is used to indicate the priority to be accorded to a packet. The routers may ignore this field.

– The **Total Length** field (16 bits wide) is the length of the datagram in bytes, including the header and the data. The minimum value for this field is 576.

– The **Identification** field (16 bits wide) 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 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 **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 **Time To Live** field (8 bits wide) determines how long the packet can live in the internet. As previously mentioned, each time a packet passes through a router, its TTL is decremented by one.

– The **Protocol** field (8 bits wide) is the integer identifier for the higher-level protocol that generated the data portion of this packet. (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) 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) is the IP address of the source.

– The **Destination Address** field (32 bits wide) 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.

- The IP protocol is also responsible for fragmenting a descending datagram and reassembling the packets into what would become an ascending datagram. Fragmentation is carried out so that the packets can fit the packet size as dictated by the hardware constraints of the destination network. *(But note that if the IP layer produces outgoing packets that are too small, any IP layer filtering 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.)*

- Note that, whereas the TCP protocol is a a connection-oriented protocol, the IP protocol that resides in the internet layer 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.

# The Transport Layer (TCP)

- TCP provides a connection-oriented, reliable byte-stream service.

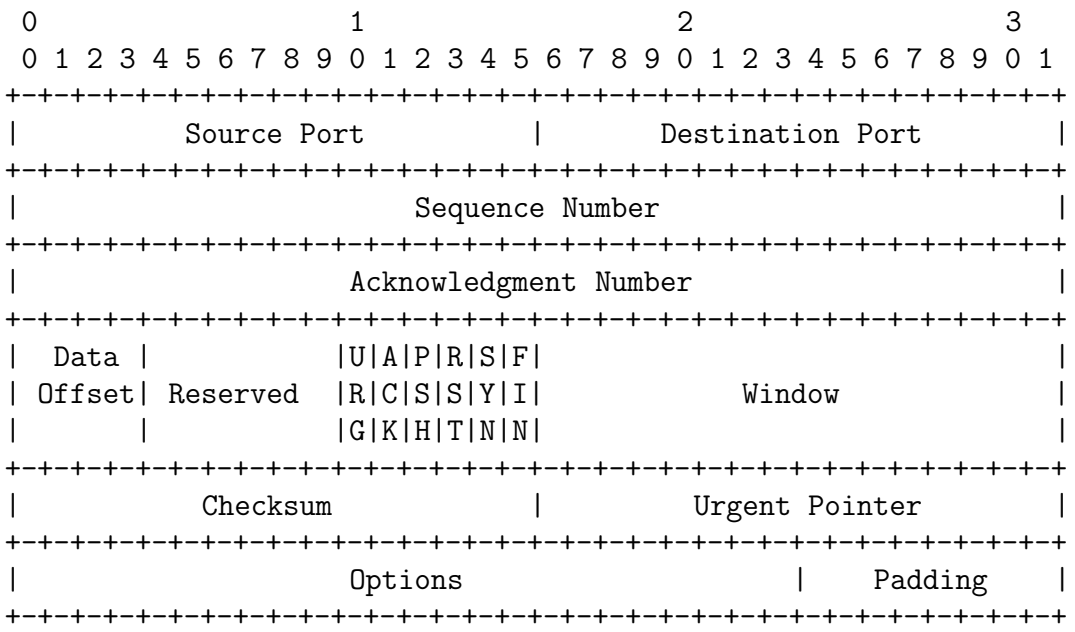
- The term connection-oriented means the applications using TCP must establish a TCP connection with each other before they can exchange data.

- 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 a flow-control mechanism for each of the two byte-streams mentioned above. **The flow-control mechanism allows the receiver to limit how much data the sender can transmit.**

- TCP also implements a congestion control mechanism that will be presented later.

- The header of a TCP segment is shown below (taken from RFC793, dated 1981):




The various fields of the 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, and

– The **Acknowledgment Number** field

with each field being 32 bits wide. These two fields 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 now exchanging data:

* When a host A first wants to expresses a desire to establish a TCP connection with a remote host B, 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. The remote host, B, must send back to A what is known as an `ACK` packet containing the same sequence number in its **Acknowledgment** field. That way A knows as to which connection request B is responding to.

* In an on-going connection between two parties A and B, as the parties are engaged in the exchange of data, each party associates a byte count with the outgoing bytes. Say that A's TCP sends to B's TCP a datagram containing 1000 bytes of data. The **sequence number** field of the TCP header for this outgoing packet will contain the byte-count associated with the first byte of the 1000-byte data block. When B receives this data block, its TCP acknowledgment packet must contain a value it expects to see in the **sequence number** field of the next TCP data block 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
  * **2nd flag bit:** ACK when set means acknowledgment
  * **3rd flag bit:** PSH when set means that the sender has invoked the push operation
  * **4th flag bit:** RST when set means that the receiver 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 has no further data to send

– The **Window** field (16 bits wide) indicates the maximum number of data bytes a receiver is willing to accept from a sender. When A sends a request-for-new-connection to B, B responds back with an acknowledgment packet whose **window** field tells the sender's TCP as to what maximum number

of data bytes that the receiver is willing to accept in a single packet. *We will see later how TCP achieves flow control by the receiver varying the value of the* **Window** *field in the acknowledgment packets sent back to the sender.*

– The **Checksum** field (16 bits wide) is computed as the 16-bit one's complement of the one's complement sum of all 16-bit words in the header and the data (including what is known as the *pseudo header* that consists of three 32-bit words. While computing the **checksum**, the **checksum** field itself is replaced with zeros.

– The **Urgent Pointer** field (16 bits wide) points to one past the last byte of the urgent data. The urgent data begins at the value in the sequence number field. This field is only interpreted if the URG control bit is set. (One application of the Urgent Point is in Telnet where an immediate response in the form of the echoing of a character is needed.)

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

## TCP vs IP

- IP provides a best-effort, connectionless, and *unreliable* packet delivery service for the higher layer (the TCP layer).

- 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. As mentioned above, TCP provides a reliable segment delivery service with flow 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 datagram to be received at a specific port at the receiving. 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 datagram.

- 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 sockets involved uniquely define a communication link.

## How does TCP Break Up the Byte-Stream that Needs to be Sent to a Receiver?

- As mentioned on Slides 12 and 13, 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 into what is called a **TCP segment**. A TCP segment along with its TCP header constitutes a **TCP datagram**. It is the TCP datagrams that are passed to the IP Layer for onward transmission.

- Suppose an Application Layer protocol wants to send 10,000 bytes of data to a remote host. TCP will decide how to segment this data into TCP segments. As you will see later, this decision by TCP depends on the **Window** field sent by the receiver when the connection was first established (see Slide 14). The value of the **Window** field indicates the maximum number of bytes the receiver can accept in each TCP segment. The receiver sets a value for this field depending on the amount of memory allocated to the connection for the purpose of buffering the received data.

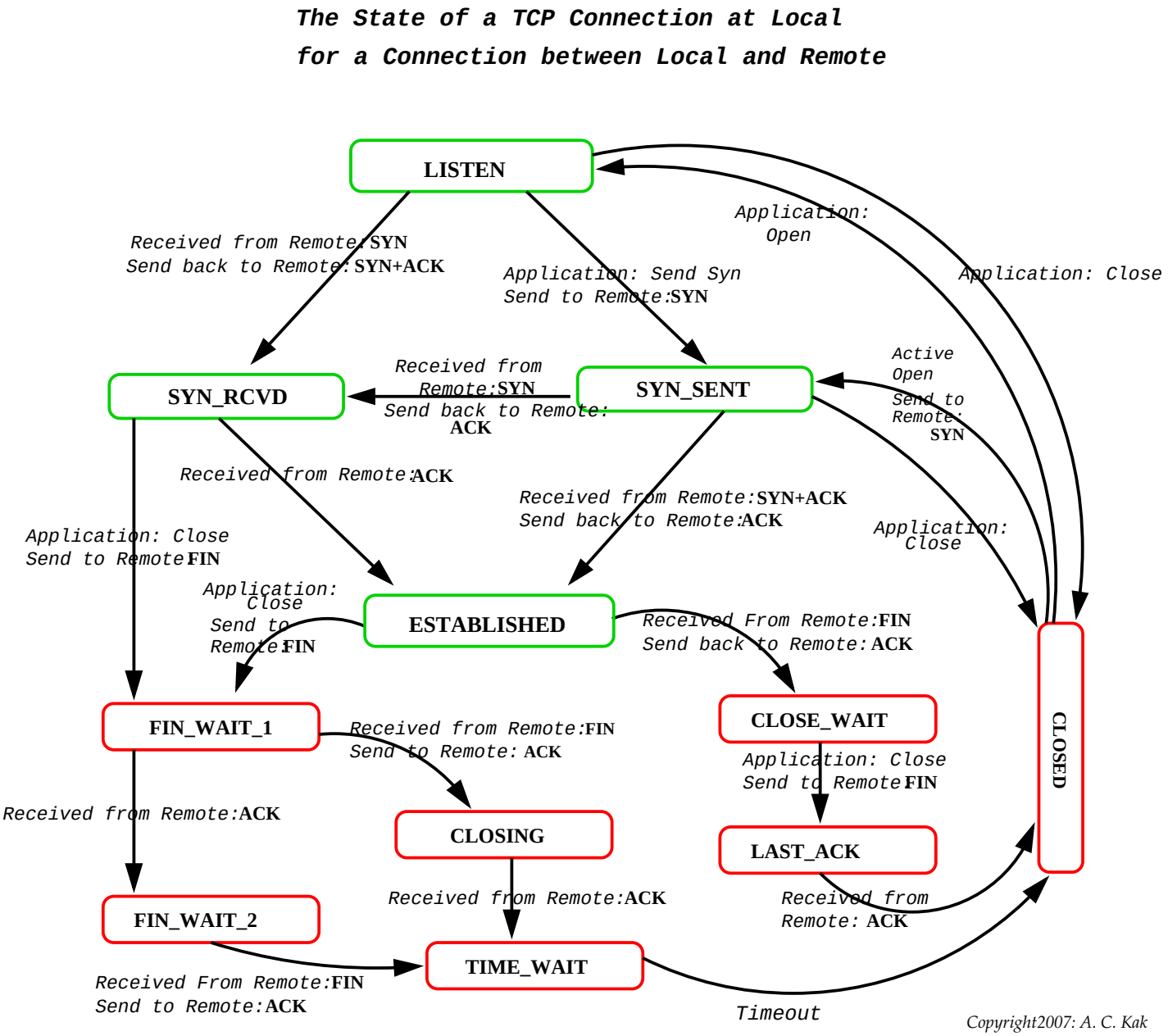
- The receiver sending back a value for the **Window** field is the main **flow control** mechanism used by TCP. This is also referred

to as the TCP's sliding window algorithm for flow control.

- If the receiver sends a 0 for the **Window** field, the sender stops sending data and starts what is known as the **Persist Timer** that is used to protect TCP from a possible deadlock situation that can occur if an updated value for **Window** is lost, the receiver has nothing further to send to the sender, and the sender is waiting for an updated value for **Window**. When the **Persist Timer** expires, the sender sends a small segment to the receiver (without any data what is optional anyway in a TCP datagram) with the expectation that the ACK received in response will contain an updated value for the **Window** field.

# TCP State Transition Diagram

- The figure shown below is the TCP state transition diagram.



- 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.

- 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 connec-

tion, note that 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 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).

## TCP Re-Transmissions

- Since TCP must guarantee reliability in communications, it retransmits a TCP segment if an ACK is not received in a certain period of time. This time period is referred to as the **Retransmission Timeout** ($RTO$).

- How $RTO$ is set is specified in RFC2988. It depends on a measured value for the **Round-Trip Transmission Time** ($RTT$). But if $RTT$ cannot be measured, $RTO$ must be set to be close to 3 seconds, with backoffs on repeated retransmissions.

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

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

  where $SRTT$ is the Smoothed Round Trip Time and $RTTVAR$ is the 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:

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

  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.

## TCP Congestion Control

- The TCP flow control already described and the TCP congestion control to be presented now can interfere with each other.

- TCP congestion control consists of the two phases described next:

- **PHASE 1: The Slow-Start Phase: (RFC2581)**

  - When a connection is first established, TCP assumes that the new segments should be injected into the network at the same rate at which acknowledgments are received from the receiver.

  - The rate at which TCP injects segments into the network is controlled by an optional TCP header field called the **CWND** field (for Congestion Window field). Initially, CWND is set to one unit of SMSS, which typically translates into a TCP segment size of 512 bytes sent at the rate of one segment per RTT, where RTT stands for the Round-Trip Transmission time. **SMSS stands for Sender Maximum Segment Size.**

  - Next, each time an ACK is received from the receiver, the CWND field is incremented by one SMSS for each ACK. This

can result in the desirable exponential ramp-up because of the following reason: When ACK is received for the first TCP segment transmitted, the CWND value will change to $2 \times SMSS$. Now the sender will transmit 2 TCP segments per RTT. When the sender receives ACKs for both these segments, the CWND value will have gotten incremented to $4 \times SMSS$. Now the sender will try to send 4 TCP segments one after another. Upon the receipt of ACKs for all these four segments, the CWND value will be get set to $8 \times SMSS$; and so on.

– But note that the receiver is not required to send ACK for every TCP segment. Not receiving an ACK for every transmitted segment does not create any confusion for the sender as long as the value of the 32-bit ACKNOWLEDGMENT NUMBER for the sequence number of the next expected byte is correct. So if the receiver chose to acknowledge only every other segment received, the growth in the rate at which the sender is injecting the packets into the network will be a slower exponential.

– The overall packet flow rate obviously depends both on the ADVERTISED WINDOW size set by the receiver (to take care of the buffer size limitations at the receiver) and the CWND value set by the sender (according the traffic congestion perceived by the sender).

– As the sender ramps up the packet injection rate, at some

point it would hit the capacity of the internet and the intermediate routers will start discarding the packets. This would indicate to the sender that the value of the congestion window, CWND, has become too large. There exist two mechanisms by which the sender perceives packet loss: timeout for receiving ACK for a given packet and the receipt of duplicate ACKs.

– In summary, during slow-start, TCP increments CWND by at most SMSS bytes for each ACK received that acknowledges new data.

• **PHASE 2: Congestion Avoidance Phase: (RFC2581)**

– The slow-start phase ends when either congestion is detected as explained earlier or when the current value of CWND exceeds a threshold called SSTHRESH (for Slow-Start Threshold).

– During congestion avoidance, CWND is incremented in a way that does not exceed one SMSS per RTT (Round-Trip Time). More commonly, the formula used to update CWND during the congestion avoidance phase is

$$CWND \quad += \quad \frac{SMSS \times SMSS}{CWND}$$

This formula is calculated on every incoming non-duplicated ACK. This provides an approximation to calculating an updated for each RTT.

– The initial value for SSTHRESH is 65535 bytes. When congestion is first detected, the minimum of the current value of CWND and the size of the ADVERTISED WINDOW is saved in SSTHRESH with the stipulation that the value saved will at least twice the value of the latest TCP segment.

– If CWND is less than or equal to SSTHRESH, TCP is in the slow-start mode; otherwise TCP is performing congestion avoidance.

## TCP Timers

As the reader should have already surmised from the discussion so far, there are various timers associated with connection establishment or termination, flow control, and retransmission of data:

**Connection-Establishment Timer:** This timer is set when a SYN packet is sent to a 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.

**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 ker-

nel 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.

## The Much-Feared SYN Flood Attack for Denial of Service

- 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, packet whose SYN flag bit is set. The purpose of sending a SYN segment is to synchronize the connection.

- **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 SYN flood attack:

  - The hostile client repeatedly sends SYN segments to every port on the server using fake IP addresses.

  - 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 sort of a 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. It can modify its firewall rules so that all SYN packets arriving from the intruder will be simply discarded. Nevertheless, if the SYN flood is strong enough and coming from multiple sources (especially if the source IP addresses in the incoming SYN packets are being spoofed), there may be no protection against such an attack.

• 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.

## TCP Vulnerabilities

- There exist network security vulnerabilities in both the TCP specification and in its various implementations.

- There exist security-related deficiencies in the TCP/IP protocol suite that make networks vulnerable to intruders. Intruder can use the following methods to create various security problems:
  - use SYN floods for DoS attacks (*as already explained*)
  - use of ICMP packet floods for DoS attacks (*similar to SYN floods*)
  - IP address spoofing to gain one-way access to victim machines (*will be explained shortly*)
  - TCP sequence number prediction

- In addition to exploiting the vulnerabilities in the TCP protocol itself, intruders can also exploit vulnerabilities in a particular TCP implementation to cause the TCP on a machine to make illegal state transitions that may seriously compromise the performance of a victim machine.

- Here are some specific TCP security vulnerabilities:

1. **DoS caused by SYN Floods:** This was explained previously.

2. **Unauthorized State Transitions:** We will now talk about a problem associated with the TCP state transitions: extraneous state transitions that can be caused by packets whose headers are (intentionally) malformed. These transitions are extraneous in the sense that they are not permitted by the TCP specification but can nonetheless be found in some TCP implementations, especially the older ones.

   – Let's say an intruder sends a packet to a host with both the SYN and FIN flags set.

   – Assuming that the receiving host is in the **LISTEN** state, the host will first see the SYN flag and transition to the **SYN_RCVD** state. Next, the TCP engine on the host will see the FIN flag.

   – But, according to the specification, there is no outgoing arc for the FIN flag at the **SYN_RCVD** state. This however does not translate into FIN being a prohibited condition

out of the **SYN_RCVD** state **in some of the older implementations of TCP**.

- These defective implementations put the the connection in the **CLOSE_WAIT** state (as if the connection was already in the **ESTABLISHED** state). In the **CLOSE_WAIT** state, the TCP engine on the victim machine expects to receive a "close" signal from the application so that a FIN packet can be sent to the remote for a termination of the connection.

- But since the connection never actually transitioned into **ESTABLISHED**, the application is not really aware of the fact that a connection is open with the remote.

- Not receiving a close signal from the higher-level application, the victim host's TCP machine gets stuck in the **CLOSE_WAIT** state. If the keep_alive timer is enabled on the victim host, TCP will be able to reset the connection and perform a transition to **CLOSED** after a period of usually two hours.

3. **Problems Caused by the Switch-Off of a Timer:** This problems is caused by the attacker switching off the Connection-Establishment timer during attempts at simultaneous open connection. Guha and Mukherjee point out the following sce-

nario that can cause an intruder X to successful DoS attack on a victim A. Let's say that an intruder X and a host A are engaged in the following interaction for the establishment of an FTP link.

- X sends an FTP request to A and a TCP connection is established between X and A to transfer control signals. Now host A sends a SYN packet to X in order to start a TCP connection for data transfer. After sending the SYN packet, A will transition to the **SYN_SENT** state.

- When X receives the SYN packet from A, it sends a SYN packet back in response. (Ordinarily, X will respond back with a SYN+ACK packet in response to A's SYN packet.)

- When host A receives this packet, it assumes that a SIMULTANEOUS OPEN CONNECTION is in progress. A therefore sends a SYN+ACK packet to X and, at the same time, IT SWITCHES OFF ITS CONNECTION ESTABLISHMENT TIMER (presumably under the belief that X has already expressed its intention to make a connection). It then transitions into the **SYN_RCVD** state.

- Host X receives the SYN+ACK packet from A but does

NOT respond back with any packets.

- But host A is expecting an ACK packet from X. Not receiving one, it gets stalled in the **SYN_RCVD** state.

- **In this manner, X successfully mounts a DoS attack on A.**

4. **IP Spoofing:** IP spoofing refers to an intruder using a forged source IP address 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 particular dangerous if there exists a trusted relationship between the victim machine and the host that the intruder is masquerading as.

# IP Spoofing

- 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 IP spoofing attacks.

- To explain how IP spoofing works (a la Guha and Mukherjee), let's assume there are two hosts A and B and another host X controlled by an adversary.

- 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:

$$B \quad --> \quad A \quad : \quad SYN + ACK$$
$$(sequence\, num:\ N,\ acknowledgment\, num:\ M)$$

- 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.

$$X\ \ (posing\ as\ A) \quad --> \quad B \quad : \quad ACK$$
$$(guessed\ acknowledgment\ num:\ N)$$

- Should the guess happen to be right, X will have a one-way connection with A. X will now be able to send commands to B and B will execute these commands assuming that they were sent by the trusted host A.

- 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. This X can do 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 get 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 (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 my 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 using a shortcut function 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.

- 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**.

- The **birthday paradox** states that given a group of 23 or more random 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%.

- The above statements of birthdays appear to be paradoxical in light of the fact that there are 365 different possible birthdays and you would think that it would take many more than 60 people for two people to have the same birthday with near certainty. (It is NOT a paradox in the sense of being a logical contradiction.)

- One way to accept the 'paradox' intuitively is that you can construct

$$C(23,2) = \begin{pmatrix} 23 \\ 2 \end{pmatrix} = \frac{23!}{21!2!} = \frac{22 \times 23}{2} = 253$$

different possible pairs from a group of 23 people. Since this number, 253, is rather comparable to 365, the total number of different birthdays, the conclusion is not surprising.

- For a probabilistic assessment of the odds, it is easiest to first calculate the probability of ALL the individuals in a group of N people having *different* birthdays:

$$\begin{aligned} p'(N) &= 1(1-t)(1-2t).....(1-(N-1)t) \\ &= \prod_{k=1}^{N-1}(1-kt) \\ &\approx 1.e^{-t}.e^{-2t}\ldots e^{-(N-1)t} \\ &= e^{-N(N-1)t/2} \end{aligned}$$

where t = 1/365. The above result follows from the fact that the first individual can have any of the 365 out of 365 birthdays, but the second individual can only have 364 out of 365 birthdays, the third individual only 363 out of 365, etc. etc.

- The probability that at least two individuals out of N will have the same birthday is then given by

$$\begin{aligned} p(N) &= 1 - p'(N) && (P) \\ &\approx 1 - e^{-N(N-1)t/2} \\ &\approx 1 - \left(1 - \frac{N(N-1)t}{2}\right) \\ &= \frac{N(N-1)t}{2} && (Q) \end{aligned}$$

- We can invert this expression to write for N:

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

where 'ln' is the natural logarithm. Now for a given probability of "collision" $p$, we can find the smallest $N$ that will contain at least two members with the same value of the random variable, in this case the birthday. Note again that for the real birthday problem we have $t = 1/365$.

- As a digression, note that given $N$ individual in a room, the probability that any of them has the same birthday as you is very significantly smaller than what is given by the formulas in lines (P) and (Q) above. In fact, this probability would be give by $1 - (1 - t)^N$. That is because the probability of a second individual having a birthday different from yours is $1 - t$ where $t = 1/365$.

- Using the formula at (Q) above, let's set $t$ as follows for 32 bit sequence numbers:

$$t \quad = \quad 2^{-32}$$

- So if we construct a spoofing set with $N = 10,000$, 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.

- 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 Lecture10 slides) 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. A phase analysis of the random numbers

produced by such PRNGs shows high structured surfaces in the state space.

- To see how these structured surfaces result in small spoofing sets, let's review how the phase space is constructed:

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

$$\begin{array}{lcl} x(t) & = & seq(t) - seq(t-1) \\ y(t) & = & seq(t-1) - seq(t-2) \\ z(t) & = & seq(t-2) - seq(t-3) \end{array}$$

  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: $seq(t-1)$ and $seq(t-2)$ and we want to predict $seq(t)$. We now construct the two differences:

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

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 like 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.

## 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
```

  which just after a page being viewed in the firefox browser was closed returns

```
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**.

- 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 possible causes. You may have a problem even when both the local and the remote may both be in **ESTABLISHED** but yet the remote server may not be 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 stuck in while the data is going back and forth between the local and the remote. But if TCP state at either end is stuck in this state while there is no interaction at the application level, then 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 be-

tween 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 on the right above 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. On Unix machines, 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 ath0
  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 ath0
  ```

  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 destination address (since the first column entry is 0.0.0.0), the host 192.168.1.1 is a gateway and it is up.

- 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. (The first row says that the gateway for the network address 192.168.1.0 is 0.0.0.0, what stands for all IPv4 addresses.) Now try pinging the router 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 at the local

```
tracerout 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.

## For Further Reading .....


- W. Richard Stevens, and Gary R. Wright, "TCP/IP Illustrated, 3 Volume Set", Addison-Wesley Professional, 1993.

- Douglas E. Comer, "Internetworking with TCP/IP Vol. 1: Principles, Protocols, and Architecture (4th Edition)", Prentice-Hall, 2000.

- S. M. Bellovin, "Security Problems in the TCP/IP Protocol Suite," Computer Communications Review, Vol. 19, No. 2, April 1989. See also: S. M. Bellovin, "A Look Back at "Security Problems in the TCP/IP Protocol Suite"," 2004.

- Michal Zalewski, "Strange Attractors and TCP/IP Sequence Number Analysis," `http://www.bindview.com/Services/Razor/Papers/2001/tcpseq.cfm`, Dec. 2002.

- Biswaroop Guha and Biswanath Mukherjee, "Network Security via Reverse Engineering of TCP Code: Vulnerability Analysis and Proposed Solutions," IEEE Network, vol. 11, July-August 1997, pp. 40-48.

- Noah Davids, "TCP Connection States — A Clue to Network Health",

`http://www.samag.com/documents/sam9907d/` [*The material on how to troubleshoot networks by looking at the TCP connection states was drawn from this source.*]