This time

Digging into Networking Protocols

With a particular focus on TCP details, attacks, and defenses
Layer 3: (Inter)network layer

- Bridges multiple “subnets” to provide *end-to-end internet* connectivity between nodes
- Provides *global* addressing (IP addresses)
- Only provides *best-effort* delivery of data (i.e., no retransmissions, etc.)
- Works across different link technologies
### IP packet “header”

<table>
<thead>
<tr>
<th>4-bit Version</th>
<th>4-bit Header len</th>
<th>8-bit Type of service (TOS)</th>
<th>16-bit Total length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>16-bit Identification</td>
<td>3-bit Flags</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8-bit Time-to-live (TTL)</td>
<td>13-bit Fragment offset</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8-bit Protocol</td>
<td>16-bit Header checksum</td>
</tr>
<tr>
<td>32-bit</td>
<td>32-bit</td>
<td>32-bit</td>
<td>Source IP address</td>
</tr>
<tr>
<td>Protocol</td>
<td>Destination IP address</td>
<td>32-bit</td>
<td>Destination IP address</td>
</tr>
</tbody>
</table>

The IP packet header consists of a 20-byte header, which includes the version number, header length, type of service (TOS), total length, identification, flags, fragment offset, time-to-live (TTL), protocol, and header checksum fields.
IP Packet Header Fields (1)

- **Version number** (4 bits)
  - Indicates the version of the IP protocol
  - Necessary for knowing what fields follow
  - “4” (for IPv4) or “6” (for IPv6)

- **Header length** (4 bits)
  - How many 32-bit words (rows) in the header
  - Typically 5
  - Can provide IP options, too

- **Type-of-service** (8 bits)
  - Allow packets to be treated differently based on different needs
  - Low delay for audio, high bandwidth for bulk transfer, etc.
IP Packet Header Fields (2)

- Two IP addresses
  - Source (32 bits)
  - Destination (32 bits)

- **Destination address**
  - *Unique* identifier/locator for the receiving host
  - Allows each node (end-host and router) to make forwarding decisions

- **Source address**
  - Unique identifier/locator for the sending host
  - Recipient can decide whether to accept the packet
  - Allows destination to *reply* to the source
IP: “Best effort” packet delivery

• Routers inspect destination address, determine “next hop” in the forwarding table

• Best effort = “I’ll give it a try”
  • Packets may be lost
  • Packets may be corrupted
  • Packets may be delivered out of order

Fixing these is the job of the transport layer!
Layer 4: Transport layer

- End-to-end communication between processes
- Different types of services provided:
  - UDP: unreliable datagrams
  - TCP: reliable byte stream
- “Reliable” = keeps track of what data were received properly and retransmits as necessary
TCP: reliability

- Given best-effort deliver, the goal is to ensure 
  *reliability*
  - All packets are delivered to applications
  - ... in order
  - ... unmodified (with reasonably high probability)

- Must robustly detect and retransmit lost data
TCP’s bytestream service

- Process A on host 1:
  - Send byte 0, byte 1, byte 2, byte 3, ...

- Process B on host 2:
  - Receive byte 0, byte 1, byte 2, byte 3, ...

- The applications do not see:
  - packet boundaries (looks like a stream of bytes)
  - lost or corrupted packets (they’re all correct)
  - retransmissions (they all only appear once)
TCP bytestream service

Abstraction: Each byte reliably delivered in order

Process A on host H1

byte1 byte2 byte3 byte4 byte5 byte6 byte7 byte8

Process B on host H2

byte1 byte2 byte3 byte4 byte5 byte6 byte7 byte8
TCP bytestream service

Reality: *Packets* sometimes retransmitted, sometimes arrive out of order

Packet 1

Needs to be retransmitted

Packet 2

Needs to be buffered

Packet 3

TCP’s first job: achieve the abstraction while hiding the reality from the application
How does TCP achieve reliability?

Bytes 1000-1500

Expecting byte 1000

Expecting byte 1501

Reliability through acknowledgments to determine whether something was received.
How does TCP achieve reliability?

A

Bytes 1000-1500
Bytes 1501-2000
Bytes 2001-3000

Still expecting byte 1000

Expecting packet 3001

Still expecting byte 1000

Buffer these until

Expecting packet 3001
TCP congestion control

TCP’s second job: don’t break the network!

• Try to use as much of the network as is safe (does not adversely affect others’ performance) and efficient (makes use of network capacity)

• Dynamically adapt how quickly you send based on the network path’s capacity

• When an ACK doesn’t come back, the network may be beyond capacity: slow down.
TCP header

<table>
<thead>
<tr>
<th>16-bit Source port</th>
<th>16-bit Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit Sequence number</td>
<td></td>
</tr>
<tr>
<td>32-bit Acknowledgment</td>
<td></td>
</tr>
<tr>
<td>4-bit Header Length</td>
<td>Reserved</td>
</tr>
<tr>
<td>16-bit Checksum</td>
<td></td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
</tr>
</tbody>
</table>
TCP ports

- Ports are associated with **OS processes**
- Sandwiched between IP header and the application data
- \{src IP/port, dst IP/port\} : this 4-tuple uniquely identifies a TCP connection
- Some port numbers are well-known
  - 80 = HTTP
  - 53 = DNS
TCP header

<table>
<thead>
<tr>
<th>4-bit</th>
<th>Reserved</th>
<th>6-bit Flags</th>
<th>16-bit Advertised window</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header Length</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit</td>
<td>Checksum</td>
<td></td>
<td>16-bit Urgent pointer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Options (variable)</th>
<th>Padding</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>32-bit Sequence number</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>16-bit</th>
<th>Source port</th>
<th>16-bit</th>
<th>Destination port</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>4-bit</th>
<th>Reserved</th>
<th>6-bit Flags</th>
<th>16-bit Advertised window</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header Length</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit</td>
<td>Checksum</td>
<td></td>
<td>16-bit Urgent pointer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Options (variable)</th>
<th>Padding</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>32-bit Acknowledgment</th>
</tr>
</thead>
</table>
TCP seqno

• Each byte in the byte stream has a unique “sequence number”
  • Unique for both directions

• “Sequence number” in the header = sequence number of the \textit{first} byte in the packet’s data

• Next sequence number = previous seqno + previous packet’s data size

• “Acknowledgment” in the header = the \textit{next} seqno you expect from the other end-host
TCP header

<table>
<thead>
<tr>
<th>IP Header</th>
<th>16-bit Source port</th>
<th>16-bit Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit</td>
<td>Sequence number</td>
<td></td>
</tr>
<tr>
<td>32-bit</td>
<td>Acknowledgment</td>
<td></td>
</tr>
<tr>
<td>4-bit</td>
<td>Reserved</td>
<td>6-bit Flags</td>
</tr>
<tr>
<td>16-bit</td>
<td>Checksum</td>
<td>16-bit Advertised window</td>
</tr>
<tr>
<td>16-bit</td>
<td>Urgent pointer</td>
<td></td>
</tr>
<tr>
<td>Options (variable)</td>
<td>Padding</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
TCP flags

• SYN
  • Used for setting up a connection

• ACK
  • Acknowledgments, for data and “control” packets

• FIN

• RST
Setting up a connection

Three-way handshake

A

SYN

SYN + ACK

ACK

Data

Data

Data

B

Let's SYNchronize sequence numbers

Got yours; here's mine

Got yours, too

Waterfall diagram

Time
Setting up a connection

Three-way handshake

Let's SYNchronize sequence numbers
Got yours; here's mine
Got yours, too

A

SYN seqno=x

SYN seqno=y
+ACK x+1

ACK y+1

Data

Data

Data

B

Waterfall diagram

Time
TCP flags

- SYN
- ACK
- FIN: Let’s shut this down (two-way)
  - FIN
  - FIN+ACK
- RST: I’m shutting you down
  - Says “delete all your local state, because I don’t know what you’re talking about"
Attacks

• SYN flooding
• Injection attacks
• Opt-ack attack
SYN flooding
SYN flooding

Recall the three-way handshake:

At this point, B allocates state for this new connection (incl. IP, port, maximum segment size).

B will hold onto this local state and retransmit SYN+ACK’s until it hears back or times out (up to 63 sec).
SYN flooding

The attack

Exhaust memory at the victim B.

New connections will fail (insufficient memory)
SYNflooding details

• Easy to detect many incomplete handshakes from a single IP address

• **Spoof** the source IP address
  • It’s just a field in a header: set it to whatever you like

• Problem: the host who really owns that spoofed IP address may respond to the SYN+ACK with a RST, deleting the local state at the victim

• Ideally, spoof an IP address of a host you know won’t respond
SYN cookies
The defense

Rather than store this data, send it to the host who is initiating the connection and have him return it to you

Store the necessary state in your seqno

Check that $f(data)$ is valid for this connection. Only at that point do you allocate state.
SYN cookie format

The secure hash makes it difficult for the attacker to guess what $f()$ will be, and therefore the attacker cannot guess a correct ACK if he spoofs.
Injection attacks

• Suppose you are on the path between src and dst; what can you do?
  • Trivial to inject packets with the correct sequence number

• What if you are not on the path?
  • Need to guess the sequence number
  • Is this difficult to do?
Initial sequence numbers

- Initial sequence numbers used to be deterministic

- What havoc can we wreak?
  - Send RSTs
  - Inject data packets into an existing connection (TCP veto attacks)
  - *Initiate and use an entire connection without ever hearing the other end*
Mitnick attack

1. SYN flood the trusted server
2. Spoof trusted server’s IP addr in SYN to X-terminal
3. Trusted server too busy to RST
4. ACK with the guessed seqno
5. Grant access to all sources
6. RSTs to trusted server (cleanup)
Defenses

• Initial sequence number must be difficult to predict!
Opt-ack attack

TCP uses ACKs not only for reliability, but also for congestion control: the more ACKs come back, the faster I can send
Opt-ack attack

If I could convince you to send REALLY quickly, then you would effectively DoS your own network!

But to get you to send faster, I need to get data in order to ACK, so I need to receive quickly …or do I?
Opt-ack attack

If I can predict what the last seqno will be and when A will send it

Then I could ACK early! (“optimistically”)

A will think “what a fast, legit connection!”

Eventually, A’s outgoing packets will start to get dropped.

But so long as I keep ACKing correctly, it doesn’t matter.
Amplification

• The big deal with this attack is its Amplification Factor
  • Attacker sends x bytes of data, causing the victim to send many more bytes of data in response
  • Recent examples: NTP, DNSSEC

• Amplified in TCP due to cumulative ACKs
  • “ACK x” says “I’ve seen all bytes up to but not including x”
Opt-ack’s amplification factor

- Max bytes sent by victim per ACK:
  \[
  \text{Max window size} \times \frac{\text{MSS}}{\text{MSS}} \times (14 + 40 + \text{MSS})
  \]

- Max ACKs attacker can send per second:
  \[
  \frac{\text{Attacker bandwidth (bytes/sec)}}{(14 + 40)}
  \]
Opt-ack’s amplification factor

- Boils down to max window size and MSS
  - Default max window size: 65,536
  - Default MSS: 536
- Default amp factor: $65536 \times (1/536 + 1/54) \sim 1336\times$
- Window scaling lets you increase this by a factor of $2^{14}$
- Window scaling amp factor: $\sim1336 \times 2^{14} \sim 22M$
- Using minimum MSS of 88: $\sim 32M$
Opt-ack defenses

• Is there a way we could defend against opt-ack in a way that is still compatible with existing implementations of TCP?

• An important goal in networking is *incremental deployment*: ideally, we should be able to benefit from a system/modification when even a subset of hosts deploy it.
Next time

• We now know how to communicate *reliably* and *securely* with a given destination IP address

• Next up: **Naming**

• How do we *get* an IP address, and can we attack that process?
  • DHCP

• Given *bank.com*, how do we get its IP address?
  • DNS
  • Kaminsky attack