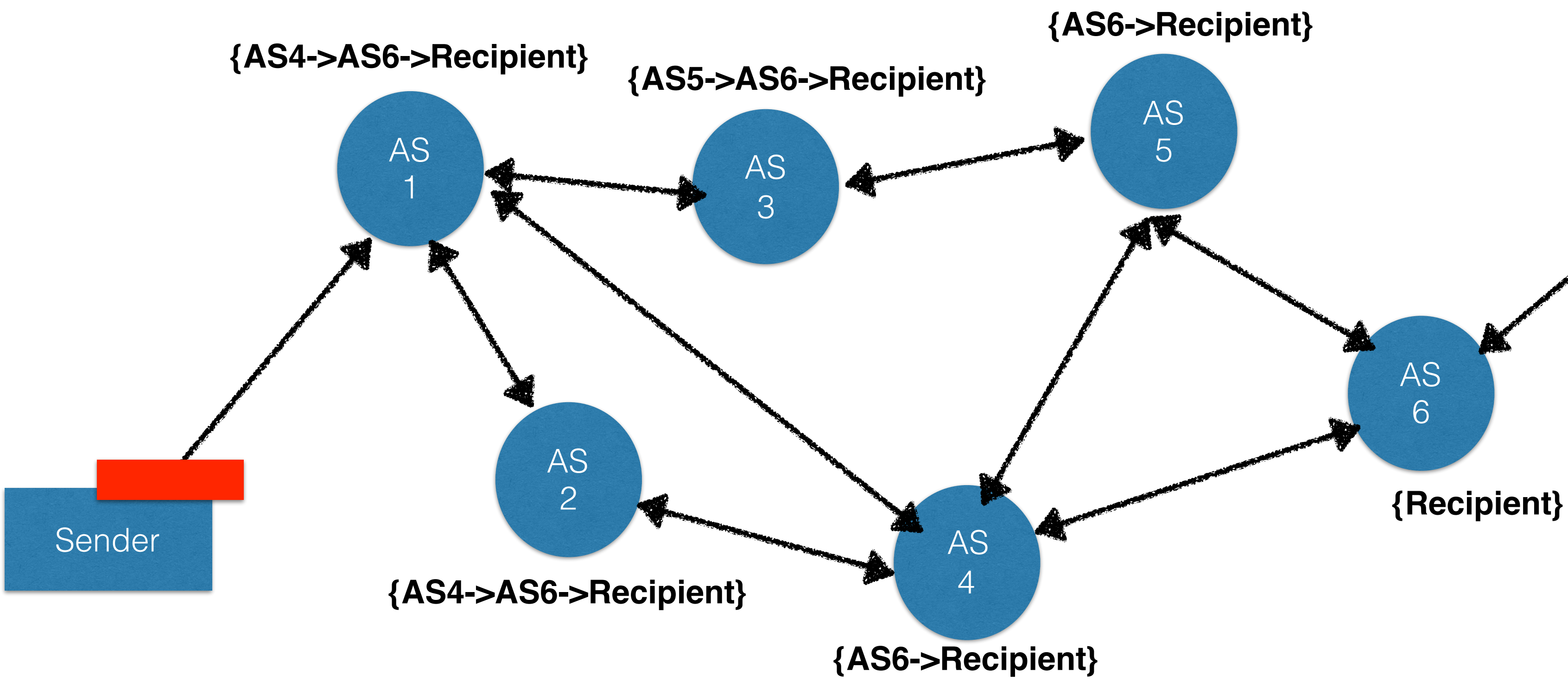
The Net Part 5: TCP & Trust

IP Routing: Autonomous Systems

- Your system sends IP packets to the gateway...
  - But what happens after that?
- Within a given network its routed internally
- But the key is the Internet is a network-of-networks
  - Each "autonomous system" (AS) handles its own internal routing
  - The AS knows the next AS to forward a packet to
- Primary protocol for communicating in between ASs is BGP:
  - Each router announces what networks it can provide and the path onward
  - ***Most precise*** route with the shortest path and no loops preferred

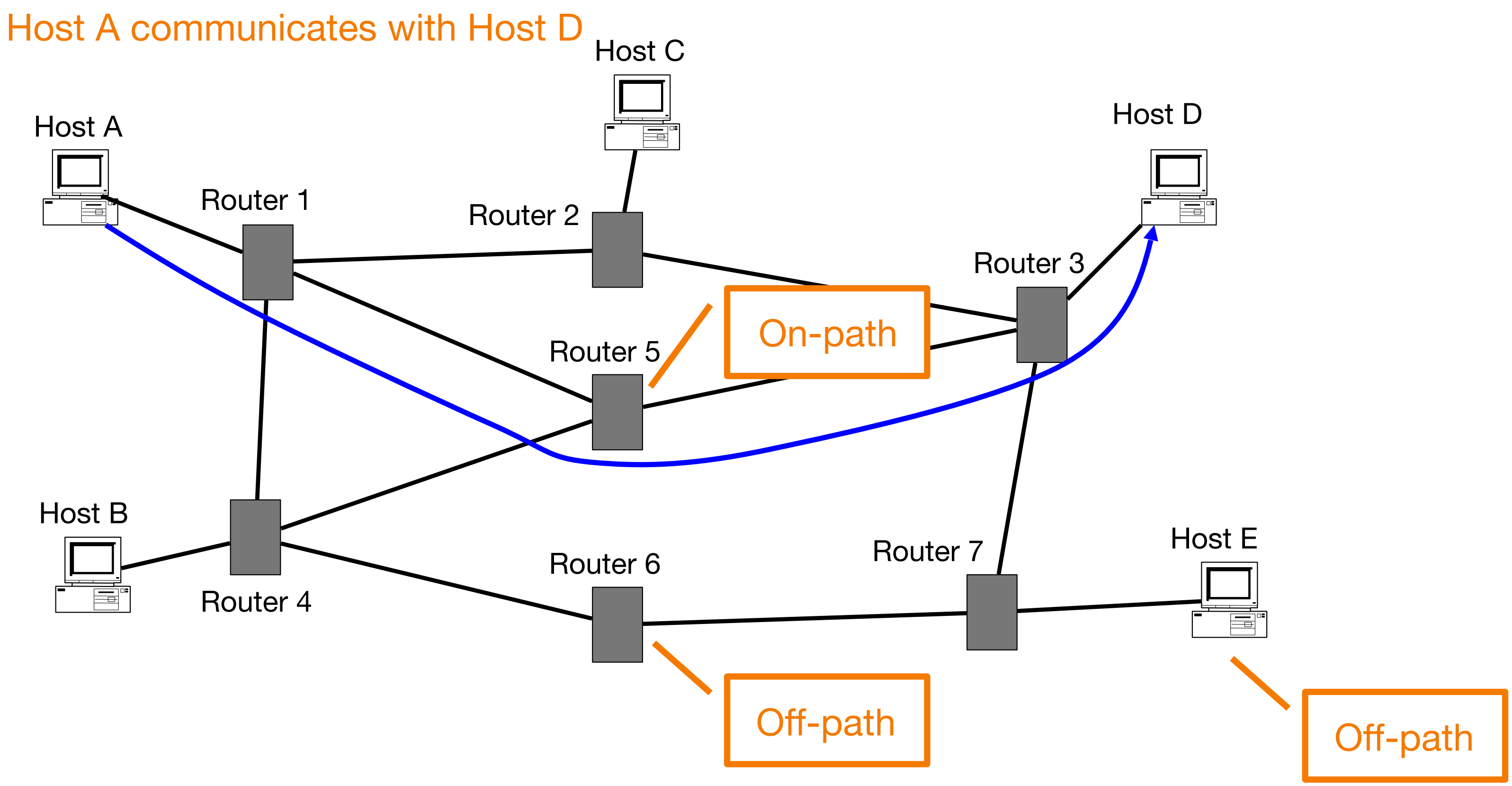
Packet Routing on the Internet:
Border Gateway Protocol & Routing Tables



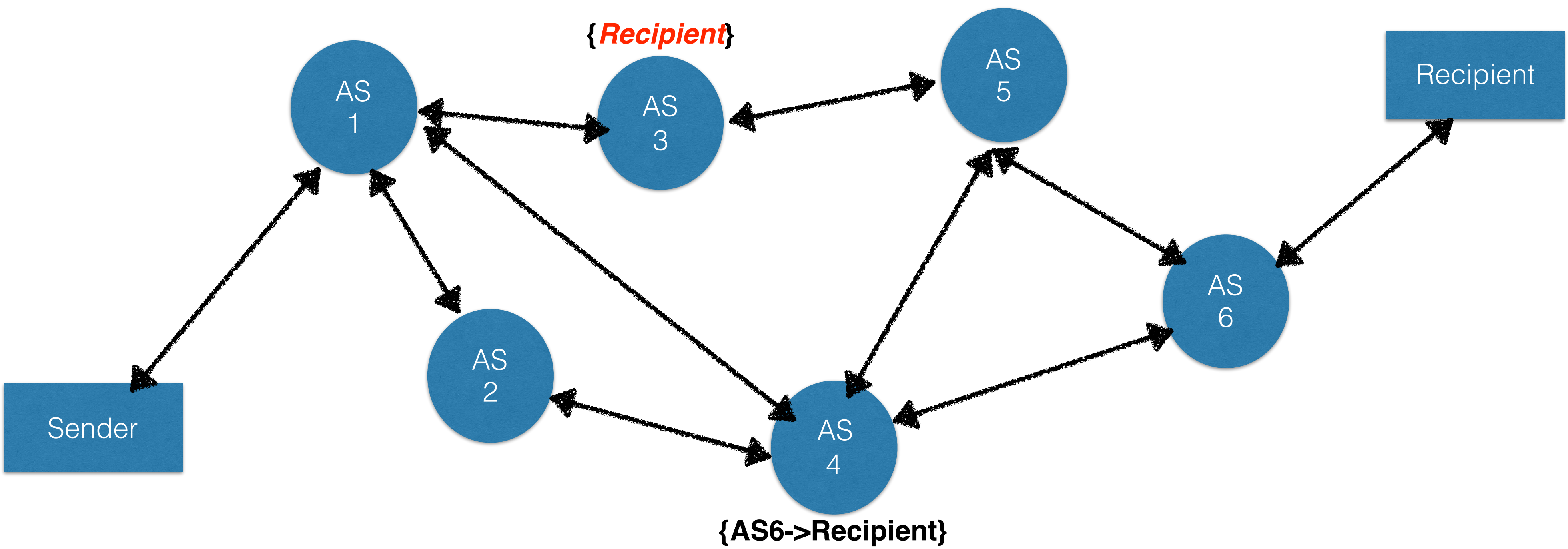
IP Spoofing
And Autonomous Systems

- The edge-AS where a user connects ***should*** restrict packet spoofing
  - Sending a packet with a different sender IP address
- But about 25% of them don't...
  - So a system can simply lie and say it comes from someplace else
- This enables blind-spoofing attacks
  - Such as the Kaminski attack on DNS
- It also enables "reflected DOS attacks"

On-path Injection vs Off-path Spoofing



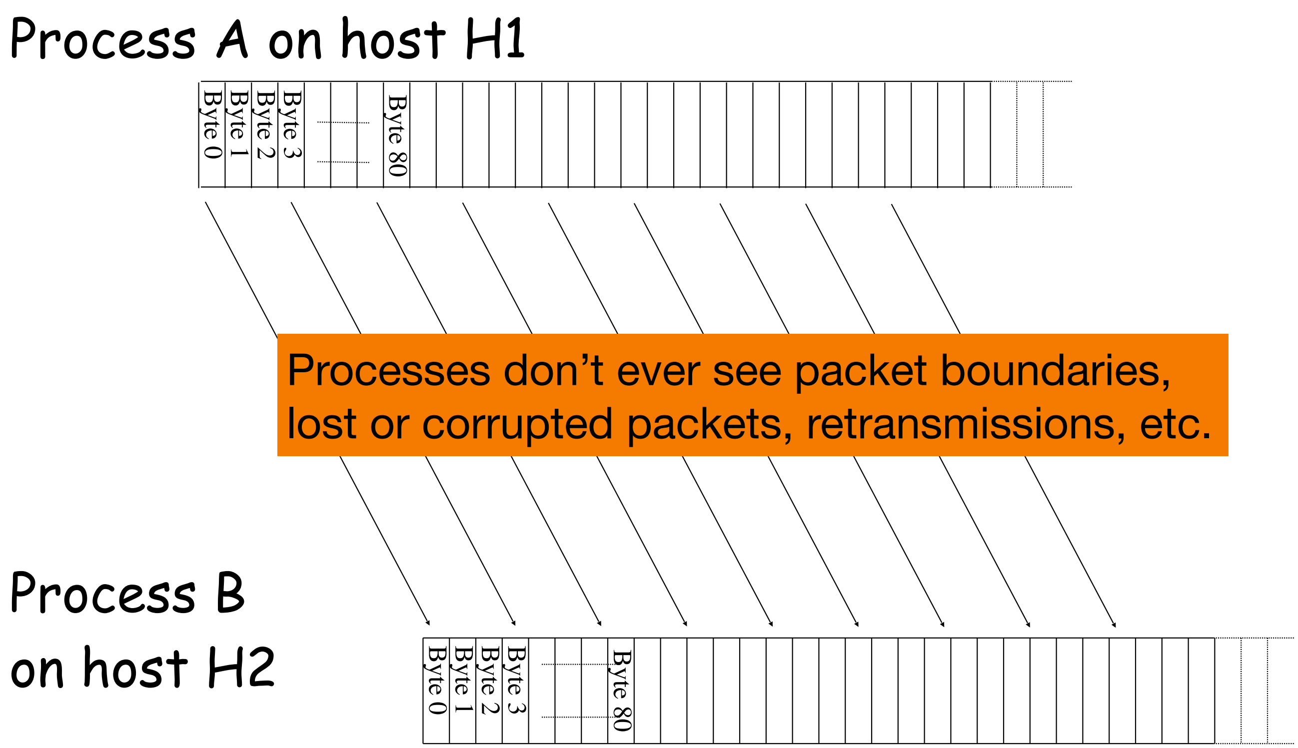
Lying in BGP


{Recipient}
AS 1
AS 3
AS 5
Recipient
AS 6
AS 2
Sender
AS 4
{AS6->Recipient}

“Best Effort” is Lame! What to do?

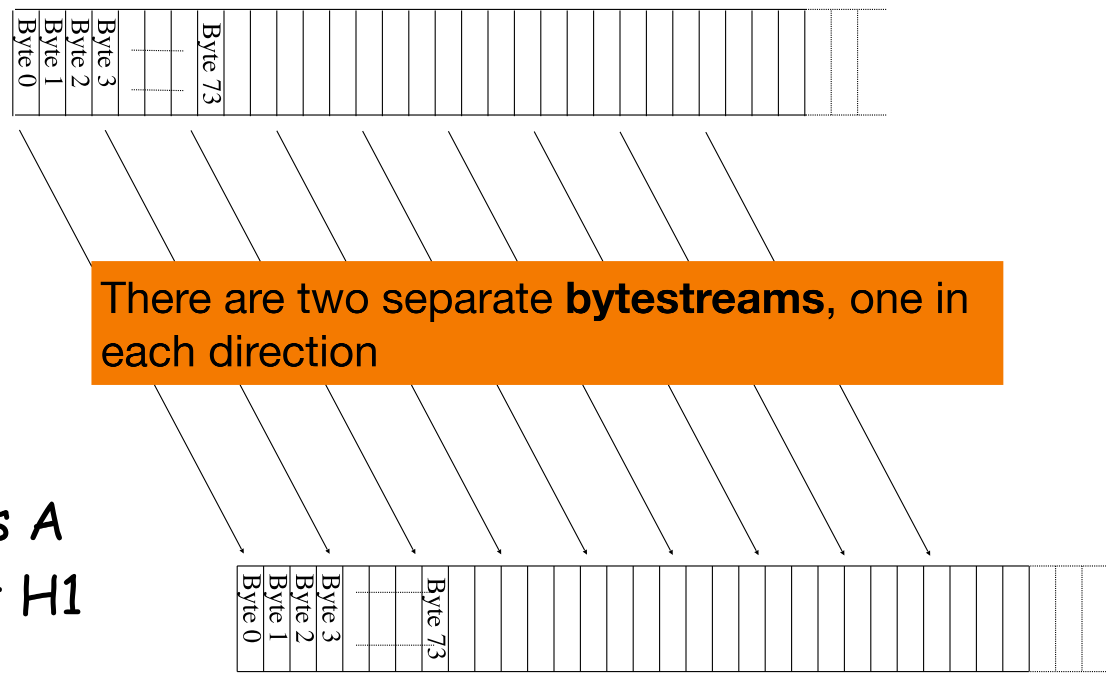
- It’s the job of our Transport (layer 4) protocols to build data delivery services that our apps need out of IP’s modest layer-3 service
- #1 workhorse: TCP (Transmission Control Protocol)
- Service provided by TCP:
  - Connection oriented (explicit set-up / tear-down)
    - End hosts (processes) can have multiple concurrent long-lived communication
  - **Reliable**, in-order, *byte-stream* delivery
    - Robust detection & retransmission of lost data

TCP “Bytestream” Service



Bidirectional communication:

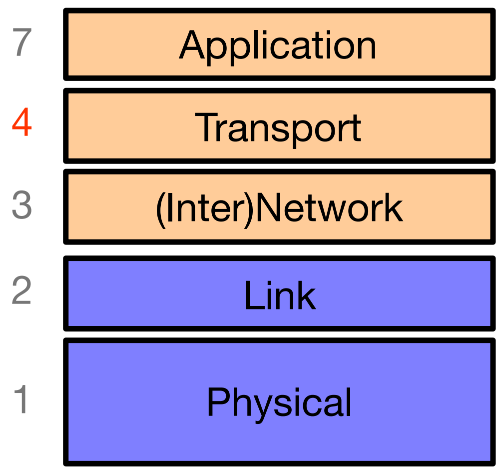
Process B on host H2

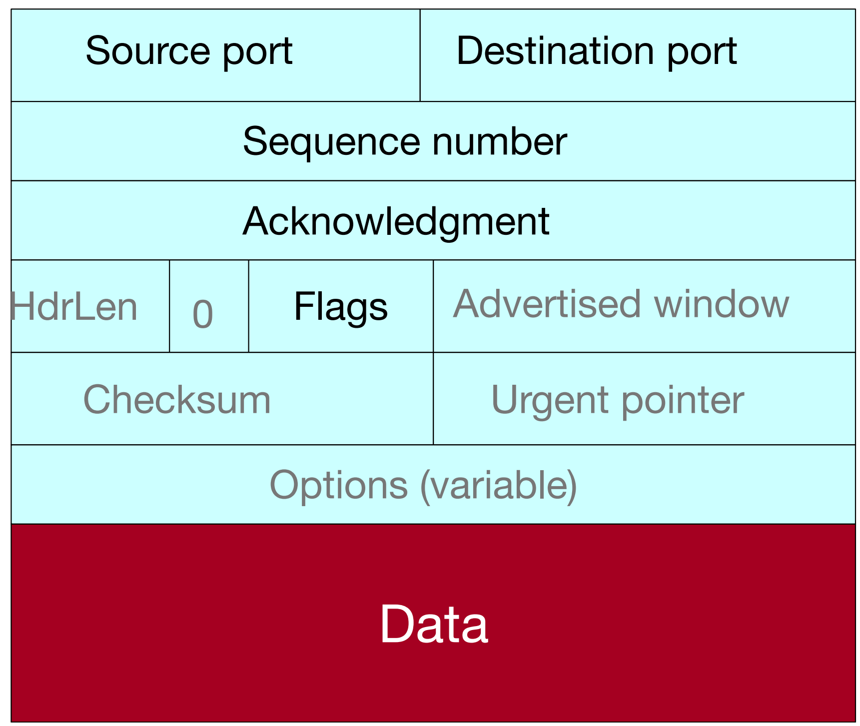


There are two separate **bytestreams**, one in each direction

Process A on host H1

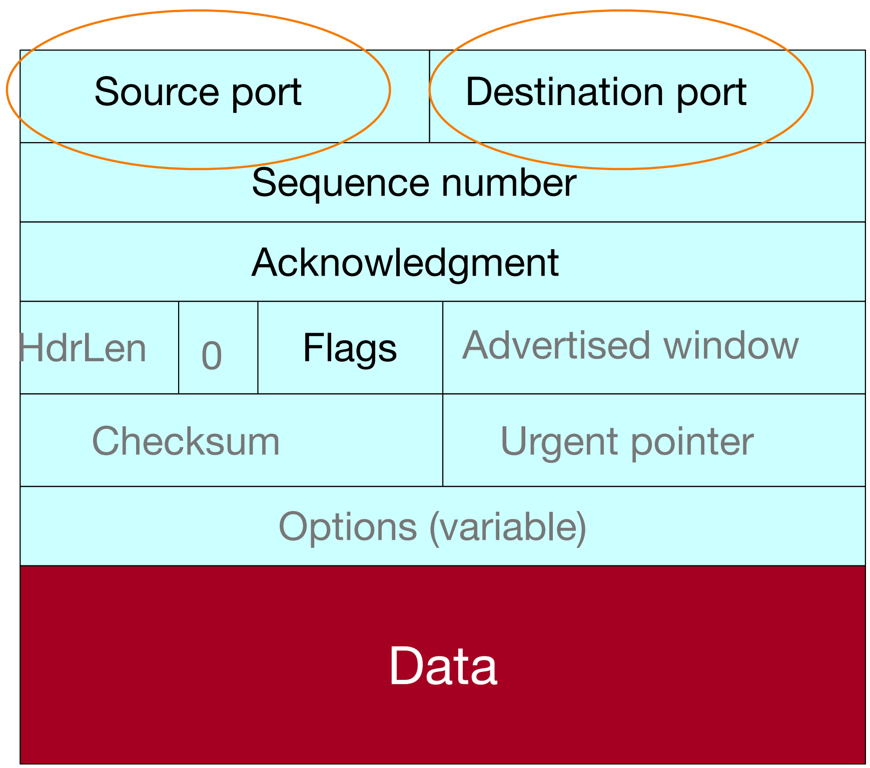
TCP


7 Application
4 Transport
3 (Inter)Network
2 Link
1 Physical

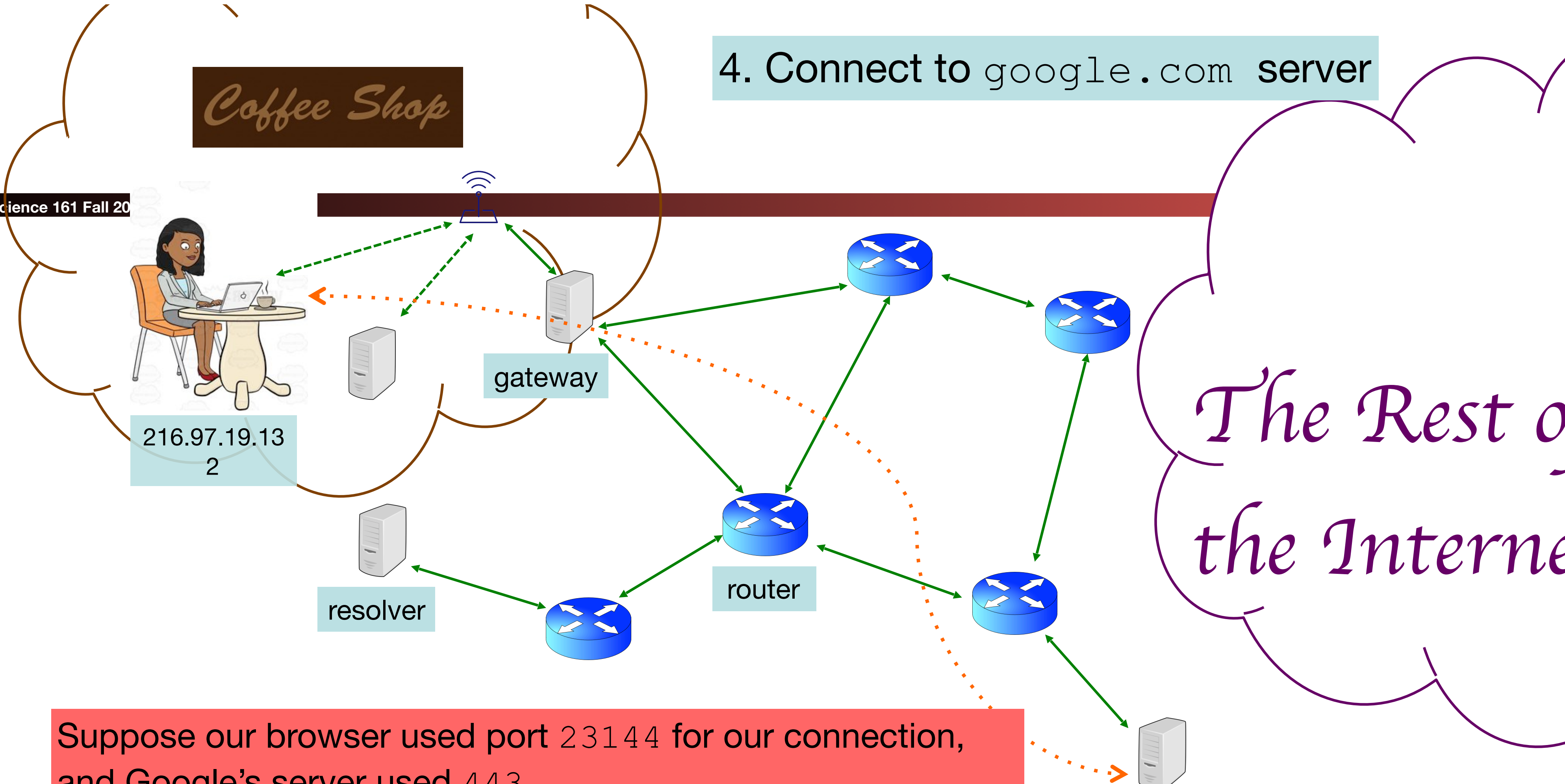

Source port
Destination port
Sequence number
Acknowledgment
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

TCP

These plus IP addresses define
a given connection



4. Connect to `google.com` server

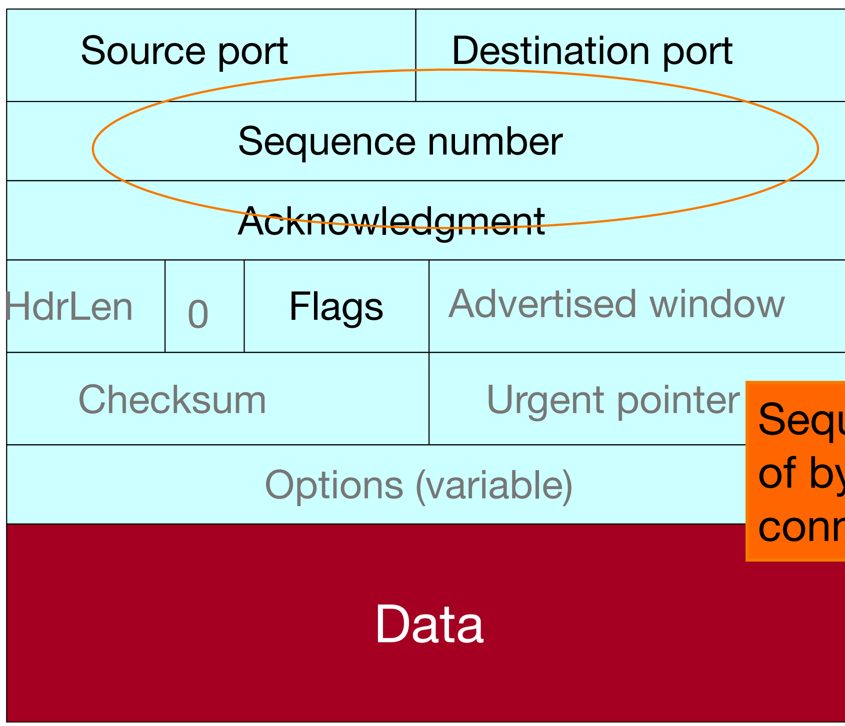


Suppose our browser used port `23144` for our connection, and Google's server used `443`.

Then our connection will be fully specified by the **single** tuple `<216.97.19.132,23144,172.217.6.78,443,TCP>`

TCP

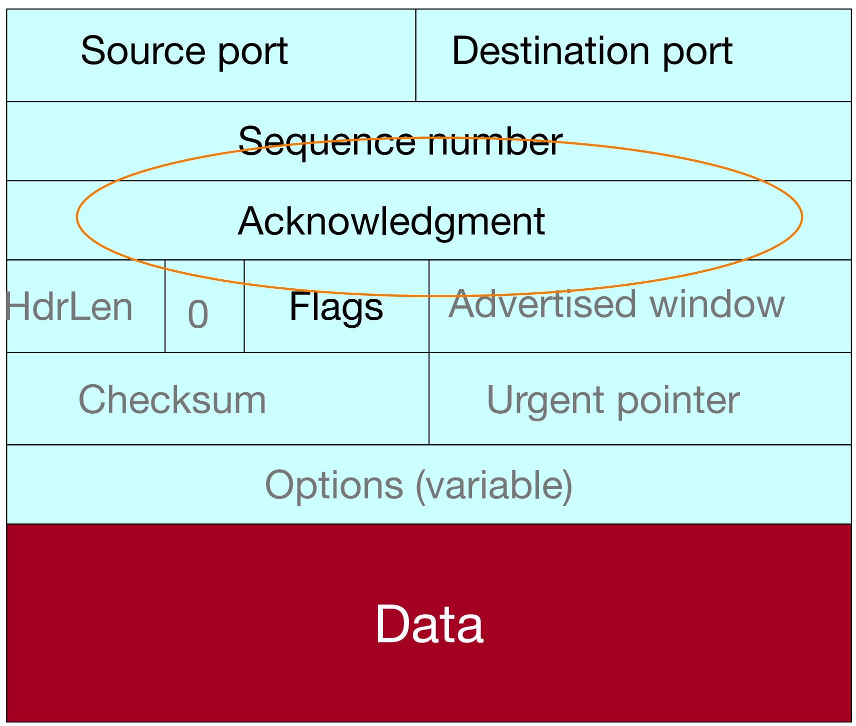
Used to order data in the connection: client program receives data ***in order***



Sequence number assigned to start of byte stream is picked when connection begins; **doesn't** start at 0

TCP

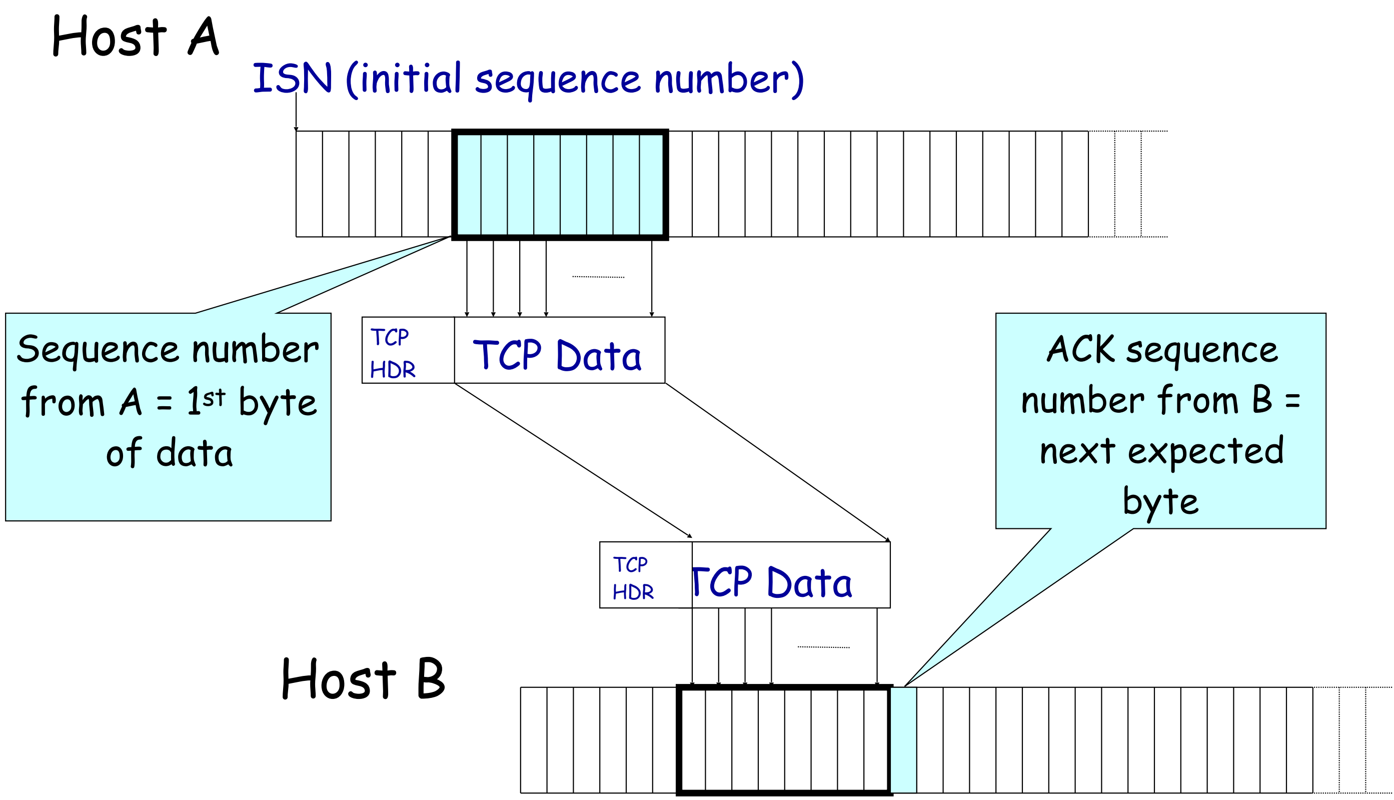
Used to say how much data has been received



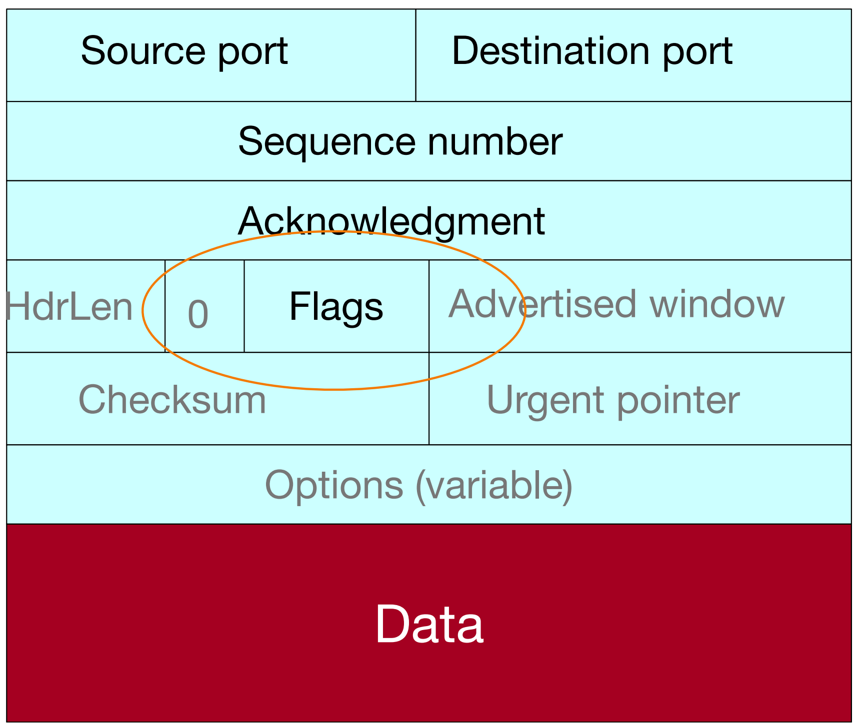
Acknowledgment gives seq # **just beyond** highest seq. received **in order**.

If sender successfully sends **N** bytestream bytes starting at seq **S** then “ack” for that will be **S+N**.

Sequence Numbers



TCP



Flags have different meaning:

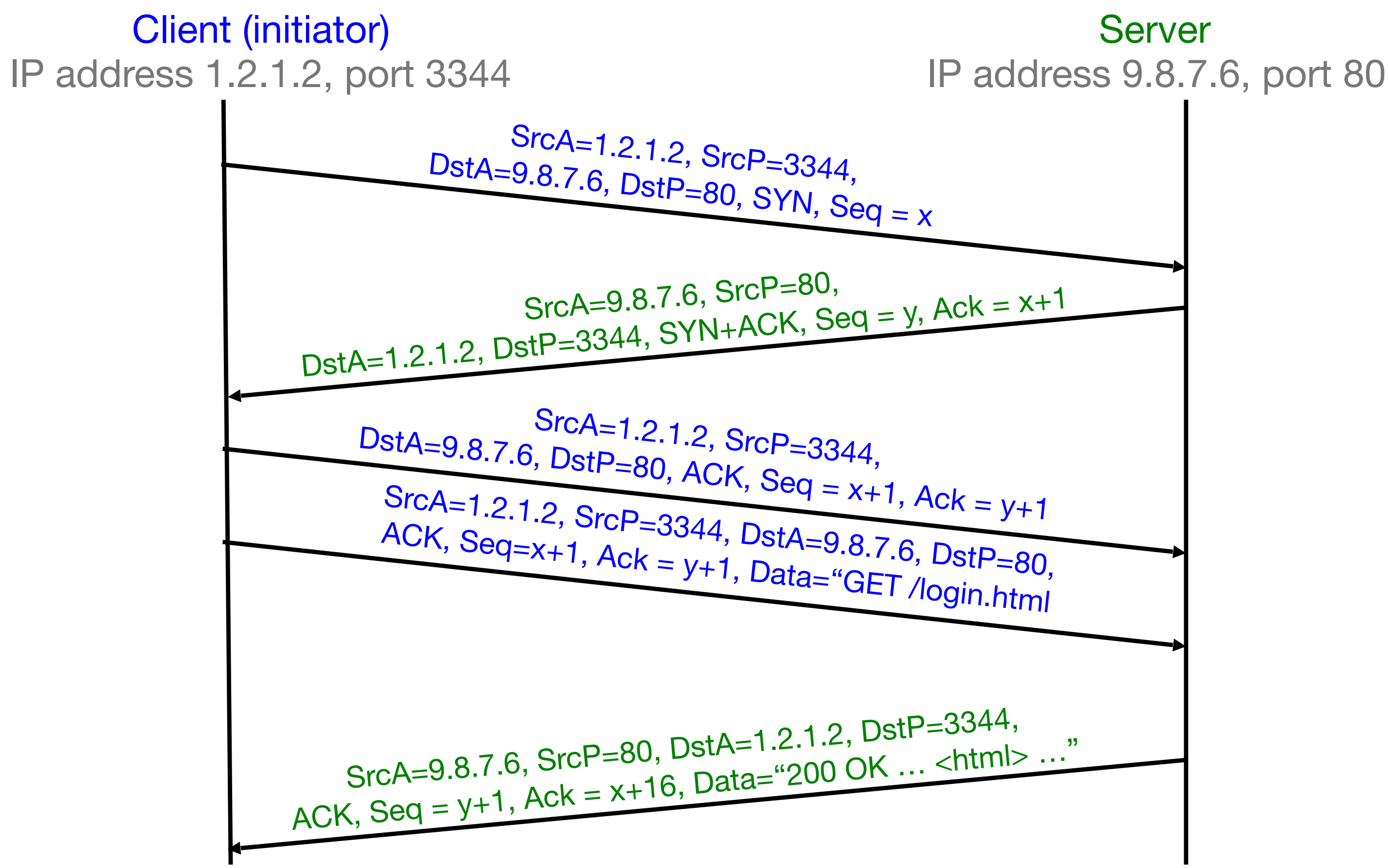
SYN: Synchronize,
used to initiate a connection

ACK: Acknowledge,
used to indicate acknowledgement of data

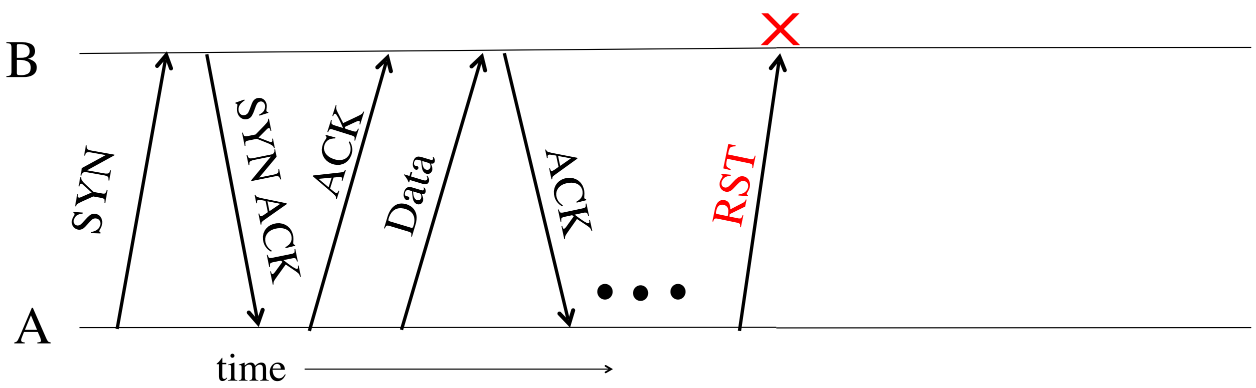
FIN: Finish,
used to indicate no more data will be sent (but can still receive and acknowledge data)

RST: Reset,
used to terminate the connection completely

TCP Conn. Setup & Data Exchange



Abrupt Termination



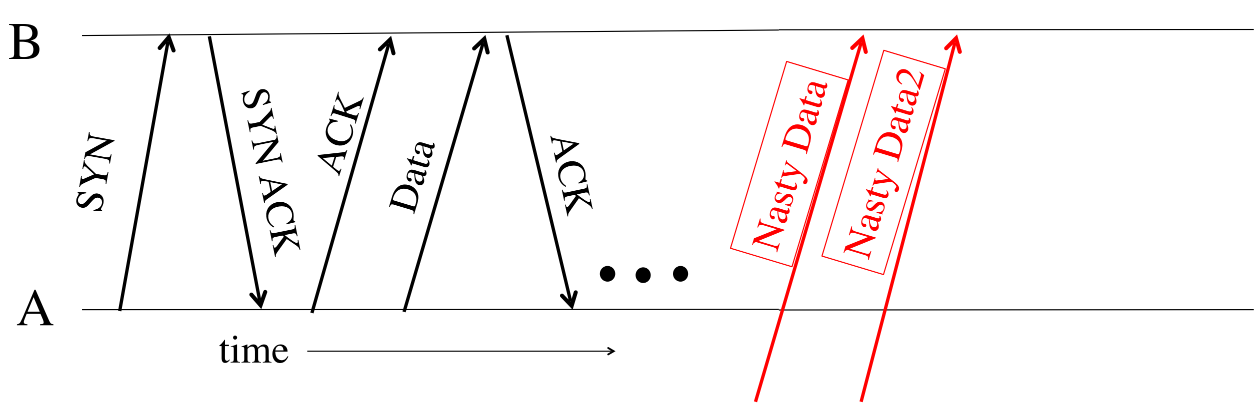
- A sends a TCP packet with RESET (**RST**) flag to B
  - E.g., because app. process on A crashed
  - (Could instead be that B sends a RST to A)
- Assuming that the sequence numbers in the **RST** fit with what B expects, That's It:
  - B's user-level process receives: `ECONNRESET`
  - No further communication on connection is possible

Disruption

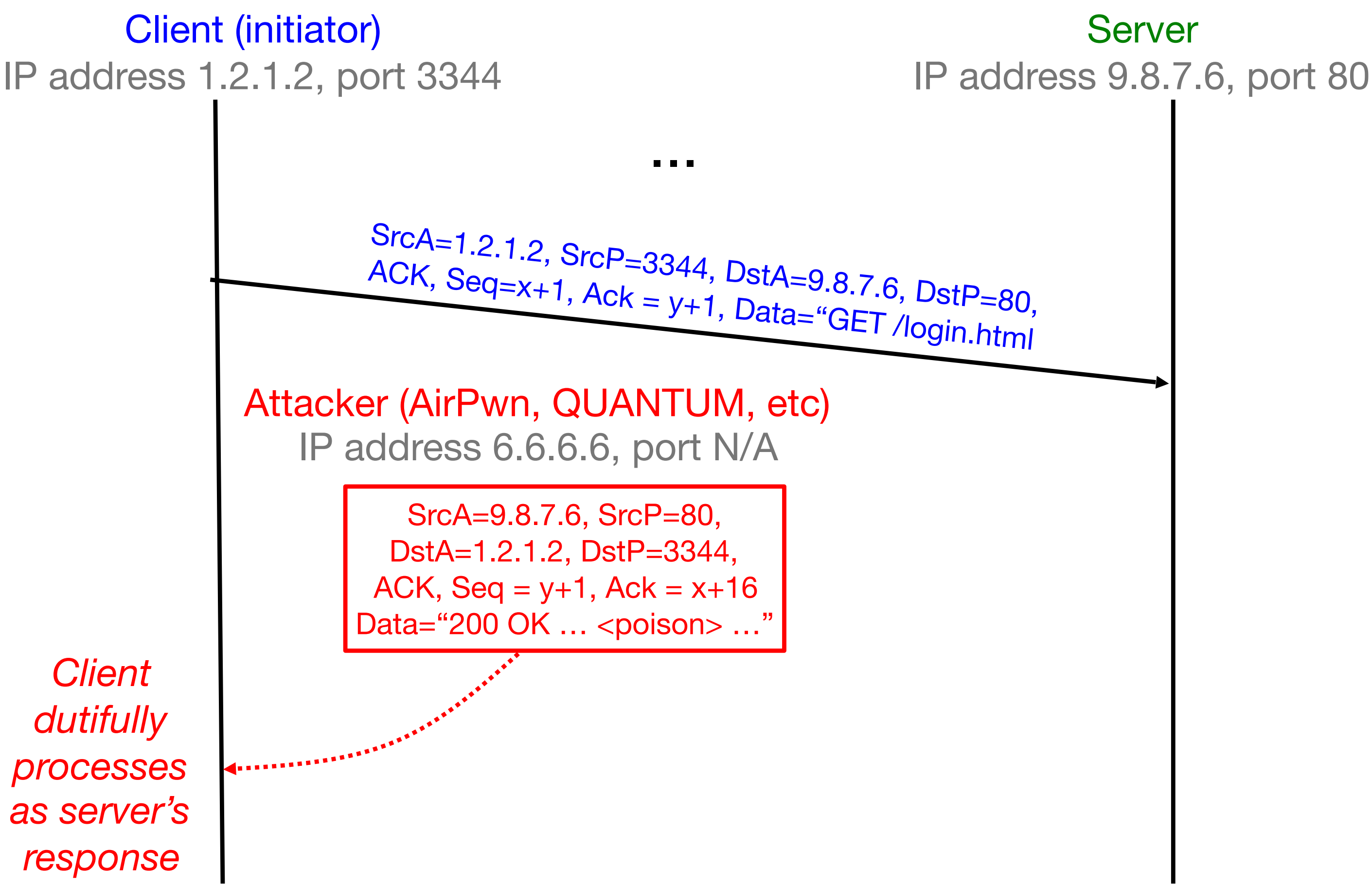
- Normally, TCP finishes ("closes") a connection by each side sending a `FIN` control message
  - Reliably delivered, since other side must ack
- But: if a TCP endpoint finds unable to continue (process dies; info from other "peer" is inconsistent), it abruptly terminates by sending a **`RST`** control message
  - Unilateral
  - Takes effect immediately (no ack needed)
  - Only accepted by peer if has correct* sequence number

TCP Threat: Data Injection

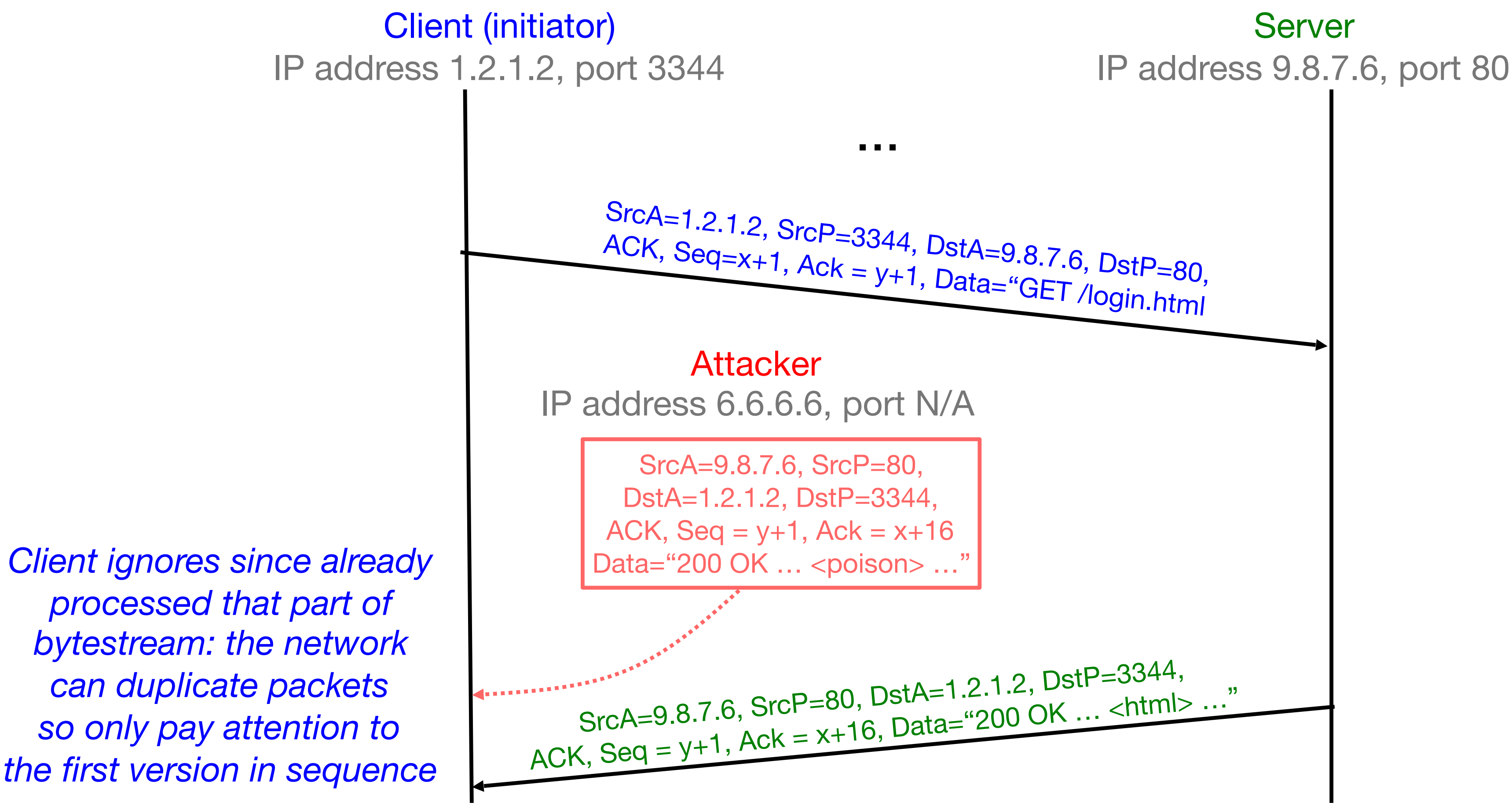
- If attacker knows ports & sequence numbers (e.g., on-path attacker), attacker can inject data into any TCP connection
  - Receiver B is *none the wiser!*
- Termed TCP connection hijacking (or "*session hijacking*")
  - A general means to take over an already-established connection!
- We are toast if an attacker can see our TCP traffic!
  - Because then they immediately know the port & sequence numbers



TCP Data Injection



TCP Data Injection



TCP Threat: Disruption aka RST injection

- The attacker can also inject RST packets instead of payloads
  - TCP clients must respect RST packets and stop all communication
    - Because its a real world error recovery mechanism
    - So "just ignore RSTs don't work"
- Who uses this?
  - China: The Great Firewall does this to TCP requests
  - A long time ago: Comcast, to block BitTorrent uploads
  - Some intrusion detection systems: To hopefully mitigate an attack in progress

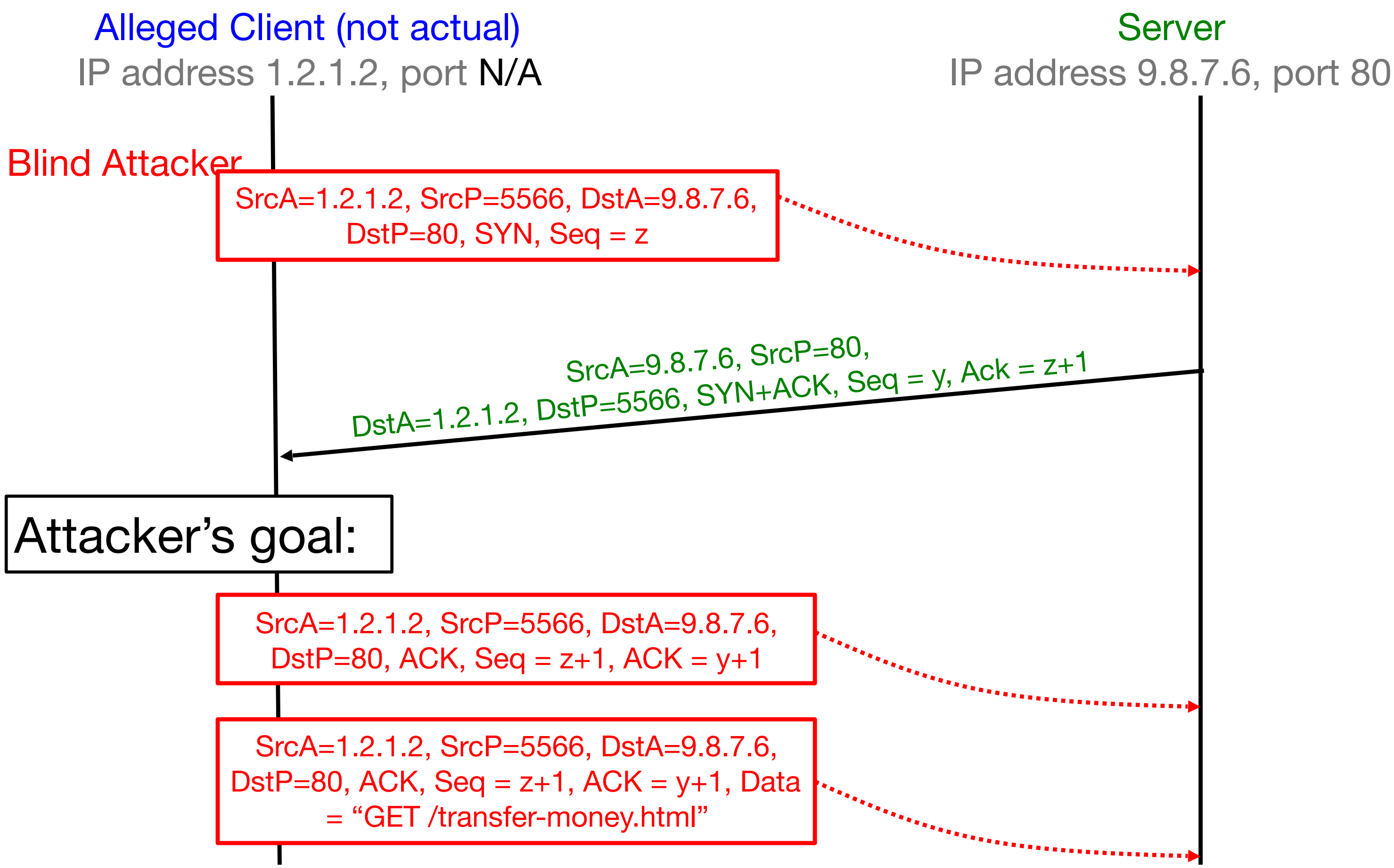
TCP Threat: Blind Hijacking

- Is it possible for an off-path attacker to inject into a TCP connection even if they can't see our traffic?
- YES: if somehow they can infer or guess the port and sequence numbers

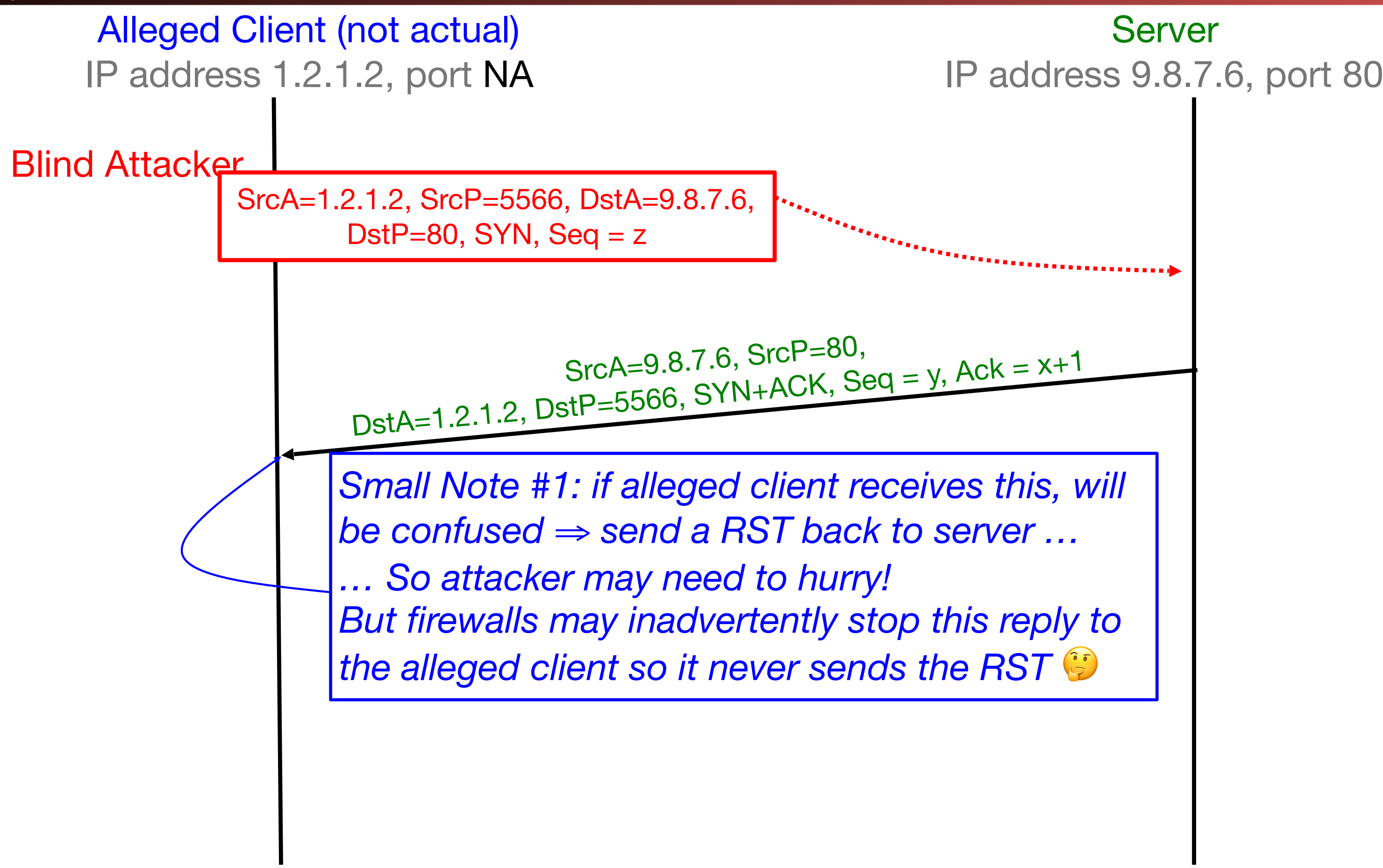
TCP Threat: Blind Spoofing

- Is it possible for an off-path attacker to create a fake TCP connection, even if they can't see responses?
- YES: if somehow they can infer or guess the TCP initial sequence numbers
- Why would an attacker want to do this?
  - Perhaps to leverage a server's trust of a given client as identified by its IP address
  - Perhaps to frame a given client so the attacker's actions during the connections can't be traced back to the attacker

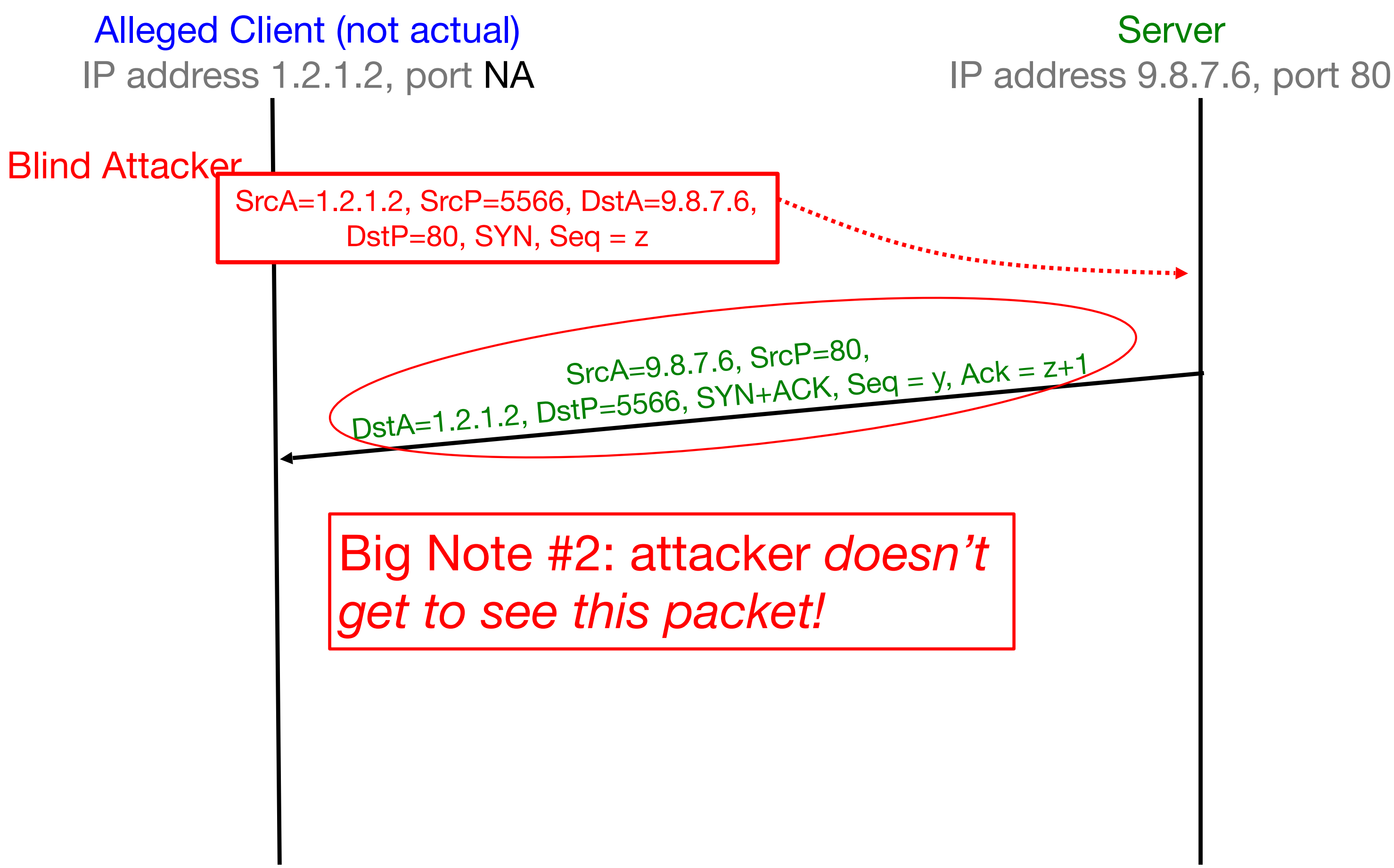
Blind Spoofing on TCP Handshake



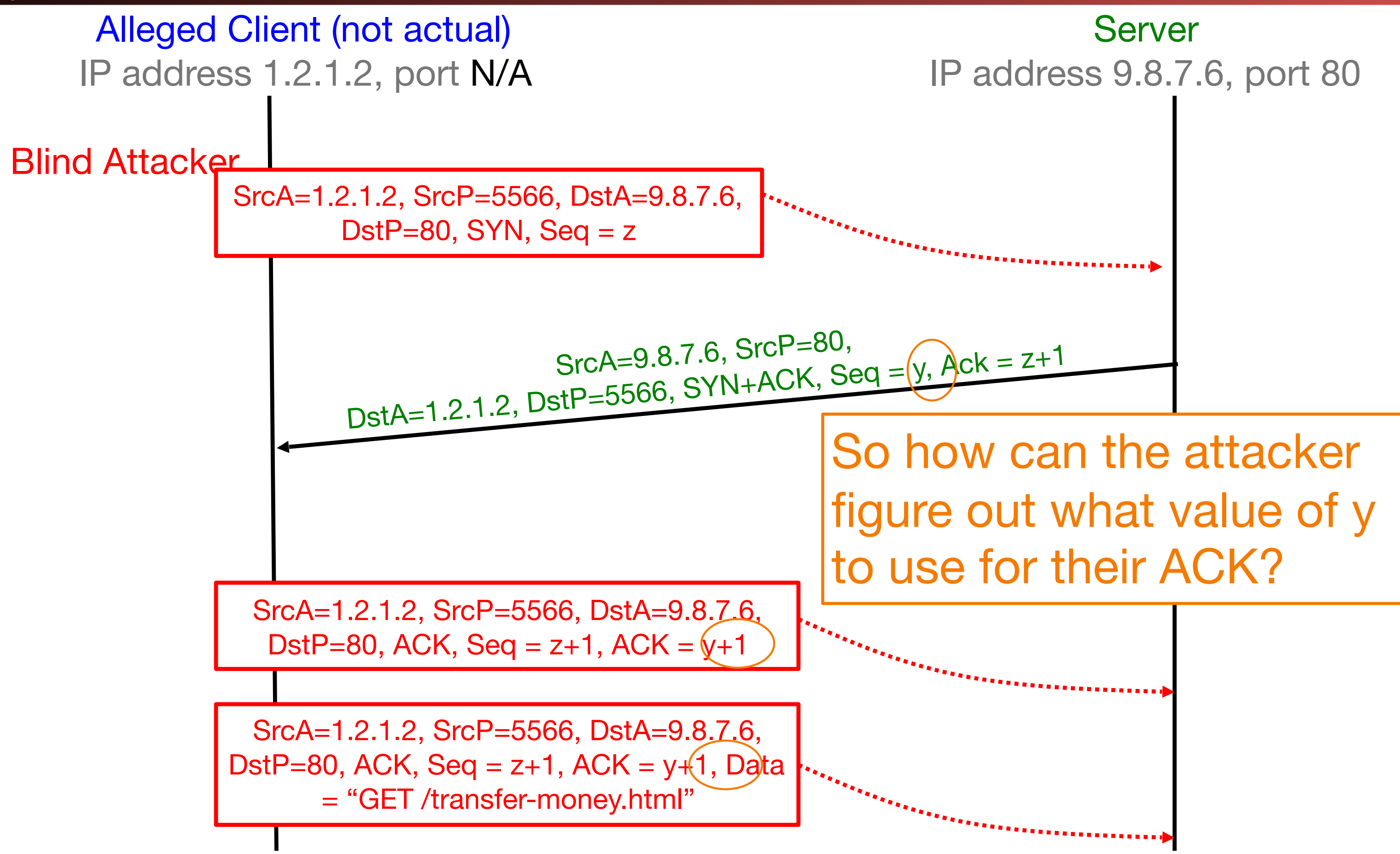
Blind Spoofing on TCP Handshake



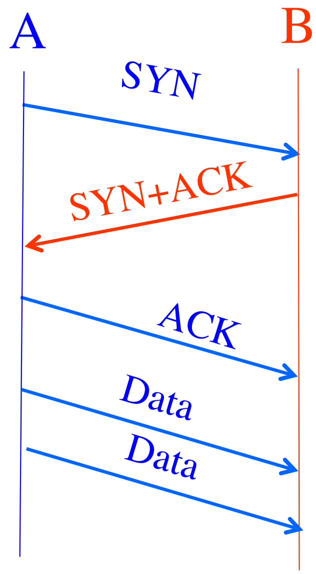
Blind Spoofing on TCP Handshake



Blind Spoofing on TCP Handshake



Reminder: Establishing a TCP Connection



Each host tells its *Initial Sequence Number* (ISN) to the other host.

(Spec says to pick based on local clock)

Hmm, any way for the attacker to know *this?*

How Do We Fix This?

Use a (Pseudo)-*Random* ISN

Sure – make a non-spoofed connection *first*, and see what server used for ISN y then!

Summary of TCP Security Issues

- An attacker who can observe your TCP connection can manipulate it:
  - Forcefully terminate by forging a RST packet
  - Inject (spoof) data into either direction by forging data packets
  - Works because they can include in their spoofed traffic the correct sequence numbers (both directions) and TCP ports
  - Remains a major threat today

Summary of TCP Security Issues

- An attacker who can observe your TCP connection can manipulate it:
  - Forcefully terminate by forging a RST packet
  - Inject (spoof) data into either direction by forging data packets
  - Works because they can include in their spoofed traffic the correct sequence numbers (both directions) and TCP ports
  - Remains a major threat today
- If attacker could predict the ISN chosen by a server, could “blind spoof” a connection to the server
  - Makes it appear that host ABC has connected, and has sent data of the attacker’s choosing, when in fact it hasn’t
  - Undermines any security based on trusting ABC’s IP address
  - Allows attacker to “frame” ABC or otherwise avoid detection
  - Fixed (mostly) today by choosing random ISNs

But wasn't fixed completely...

- CVE-2016-5696
  - "Off-Path TCP Exploits: Global Rate Limit Considered Dangerous" Usenix Security 2016
  - https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/cao
- Key idea:
  - RFC 5961 added some global rate limits that acted as an ***information leak***:
    - Could determine if two clients were communicating on a given port
    - Could determine if you could correctly guess the sequence #s for this communication
      - Required a third host to probe this and at the same time spoof packets
  - Once you get the sequence #s, you can then inject arbitrary content into the TCP stream (d'oh)

The SYN Flood DOS Attack...

- When a computer receives a TCP connection it decides to accept
  - It is going to allocate a significant amount of state
- So just send lots of SYNs to a server...
  - Each SYN that gets a SYN/ACK would allocate some state
  - So do a ***lot of them***
  - And ***spoof*** the source IP
- Attack is a resource consumption DOS
  - Goal is to cause the server to consume memory and CPU
- Requires that the attacker be able to spoof packets
  - Otherwise would just rate-limit the attacker's IPs

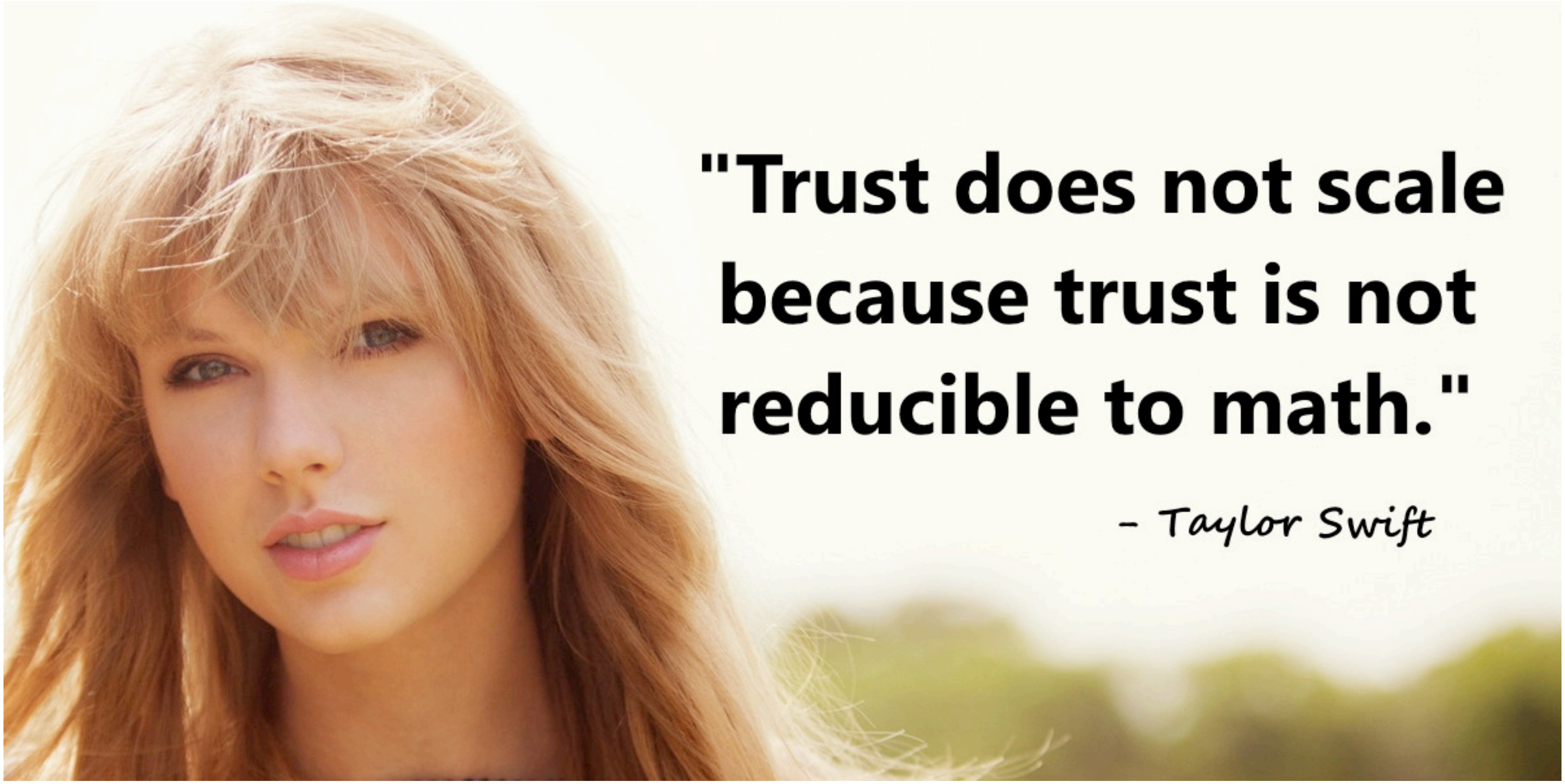
SYN-Flood Counter: SYN cookies

- Observation: Attacker needs to see or guess the server's response to complete the handshake
  - So don't allocate ***anything*** until you see the ACK...
    But how?
- Idea: Have our initial sequence ***not*** be random...
  - But instead have it be ***pseudo-***random
- So we create the SYN/ACK's ISN using the pseudo-random function
  - And then check than the ACK correctly used the sequence number

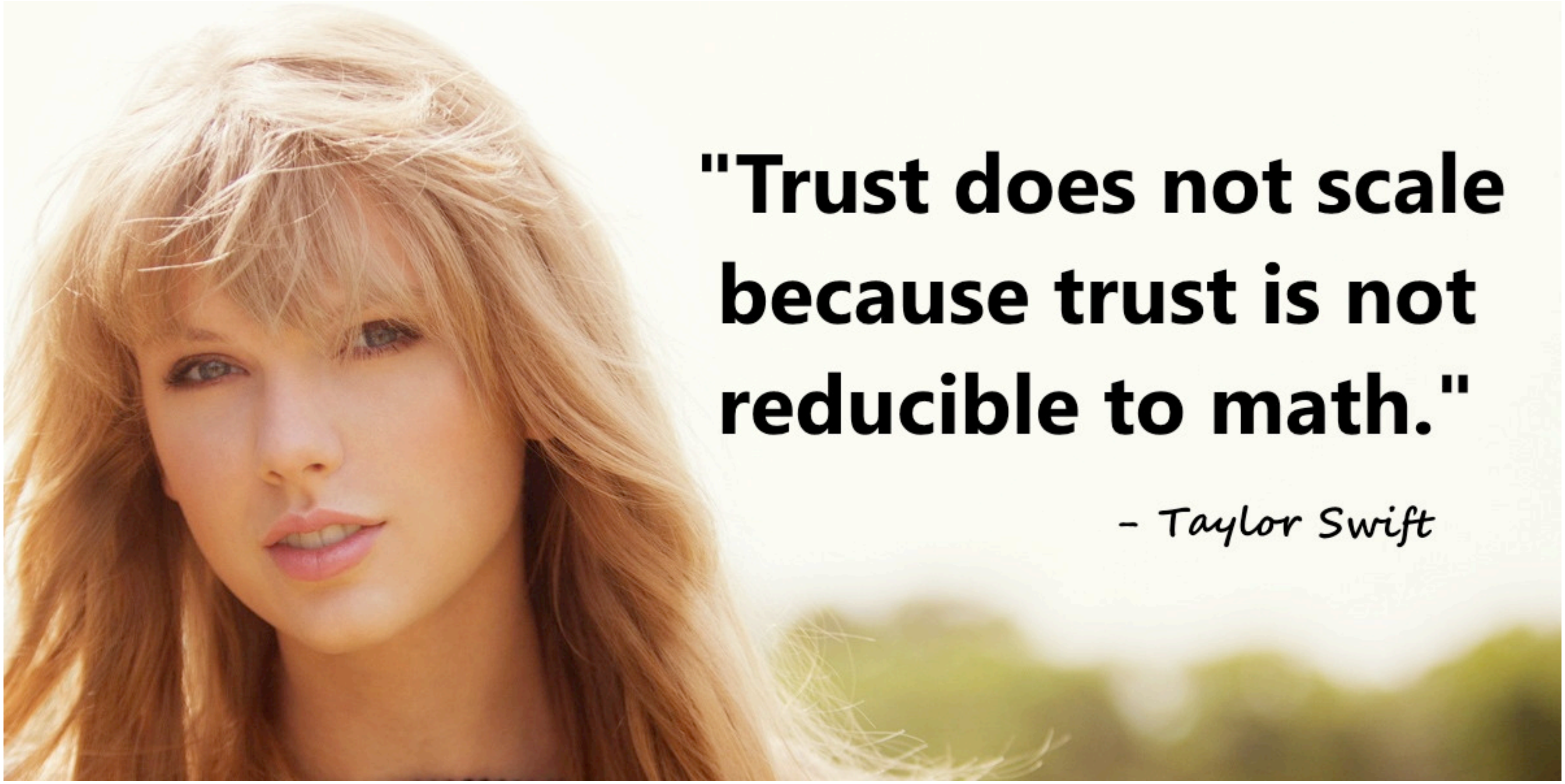
Easy SYN-cookies: HMAC

- On startup create a random key...
- For the server ISN:
  - $HMAC_k$(SIP|DIP|SPORT|DPORT|client_ISN)
- Upon receipt of the ACK
  - Verify that ACK is based off $HMAC_k$(SIP|DIP|SPORT|DPORT|client_ISN)
- Only ***then*** does the server allocate memory for the TCP connection
  - HMAC is very useful for these sorts of constructions:
    Give a token to a client, verify that the client presents the token later

Theme of The Rest Of This Lecture...



But Trust Can Be Delegated…



The Rest of Today's Lecture:

- Applying crypto technology in practice
- Two simple abstractions cover 80% of the use cases for crypto:
  - "Sealed blob": Data that is encrypted and authenticated under a particular key
  - Secure channel: Communication channel that can't be eavesdropped on or tampered with
- Today: TLS – a secure channel
  - In network parlance, this is an "application layer" protocol but…
  - designed to have any application over it, so really "layer 6.5" is a better description

Building Secure End-to-End Channels

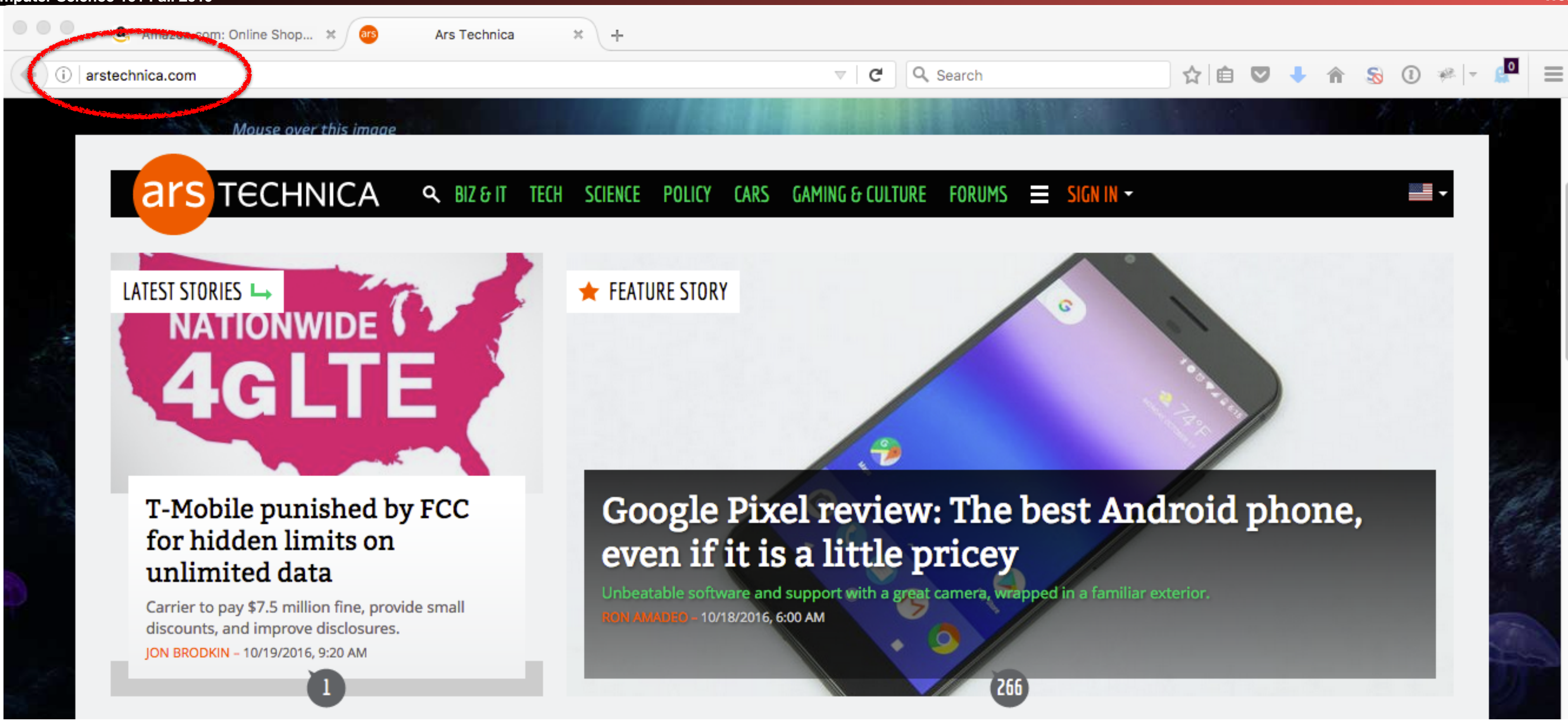
- End-to-end = communication protections achieved all the way from originating client to intended server
  - With no need to trust intermediaries
- Dealing with threats:
  - Eavesdropping?
    - Encryption (including session keys)
  - Manipulation (injection, MITM)?
    - Integrity (use of a MAC); replay protection
  - Impersonation?
    - Signatures

(What's missing? *Availability …*)

Building A Secure End-to-End Channel: SSL/TLS

- SSL = Secure Sockets Layer (predecessor)
- TLS = Transport Layer Security (standard)
  - Both terms used interchangeably
- Security for any application that uses TCP
  - Secure = encryption/confidentiality + integrity + authentication (of server, but not of client)
- Multiple uses
  - Puts the 's' in "https"
  - Secures mail sent between servers (STARTTLS)
  - Virtual Private Networks

An "Insecure" Web Page



A "Secure" Web Page

Lock Icon means:

"Your communication between your computer and the site is encrypted and authenticated"

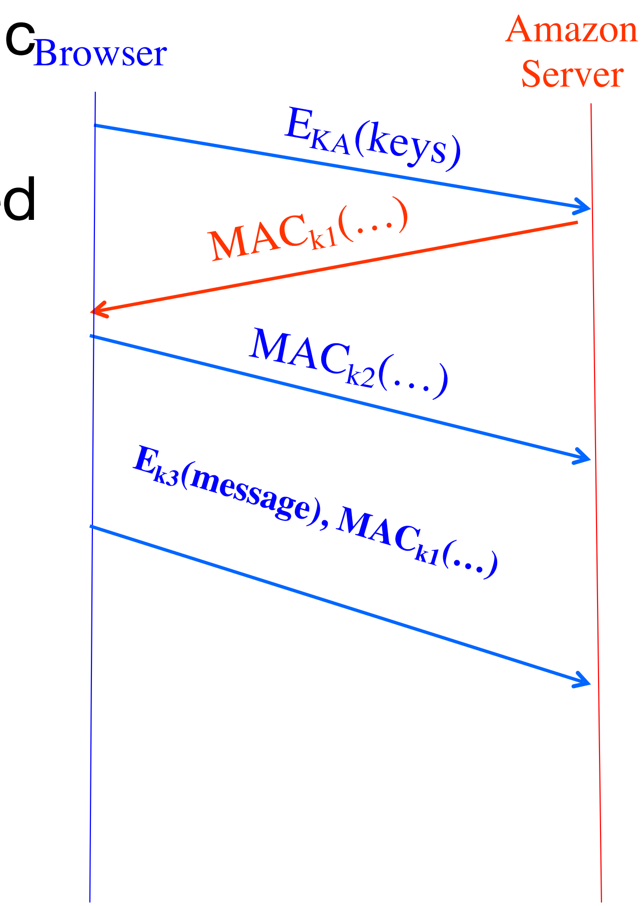
"Some other third party attests that this site belongs to Amazon"

"These properties hold not just for the main page, but any image or script is also fetched from a site with attestation and encryption"

People *think* lock icon means "Hey, I can trust this site" (no matter where the lock icon itself actually appears).

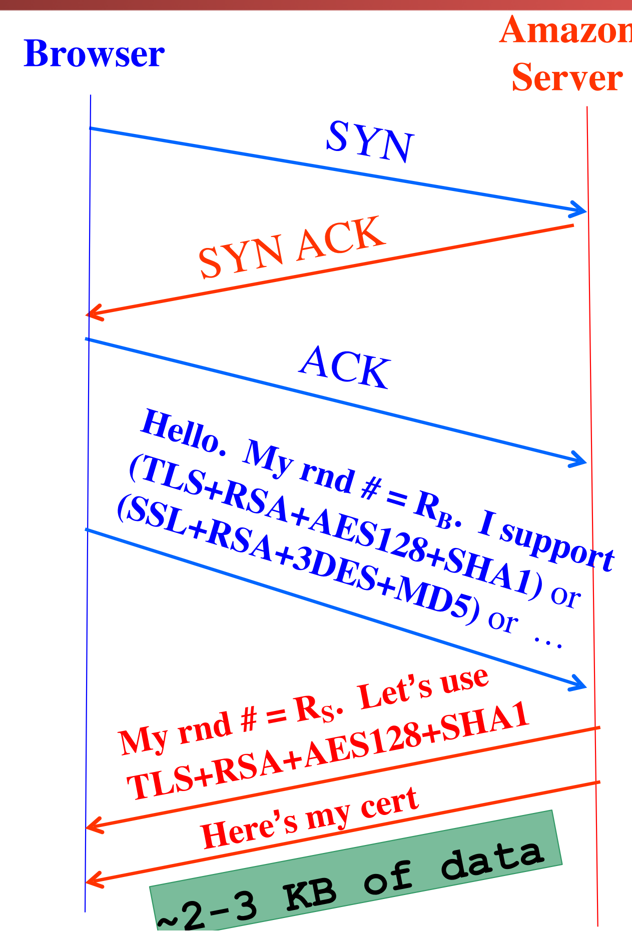
Basic idea

- Browser (client) picks some symmetric keys for encryption + authentication
- Client sends them to server, encrypted using RSA public-key encryption
- Both sides send MACs
- Now they use these keys to encrypt and authenticate all subsequent messages, using symmetric-key crypto



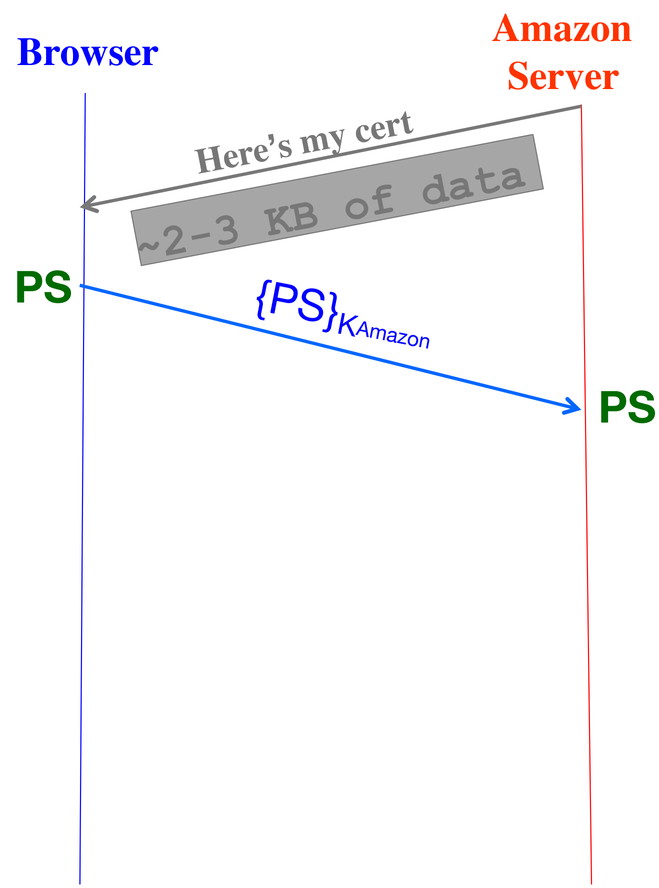
HTTPS Connection (SSL / TLS)

- Browser (client) connects via TCP to Amazon's HTTPS server
- Client picks 256-bit random number R_B, sends over list of crypto protocols it supports
- Server picks 256-bit random number R_S, selects protocols to use for this session
- Server sends over its certificate
  - (all of this is in the clear)
- Client now ***validates*** cert



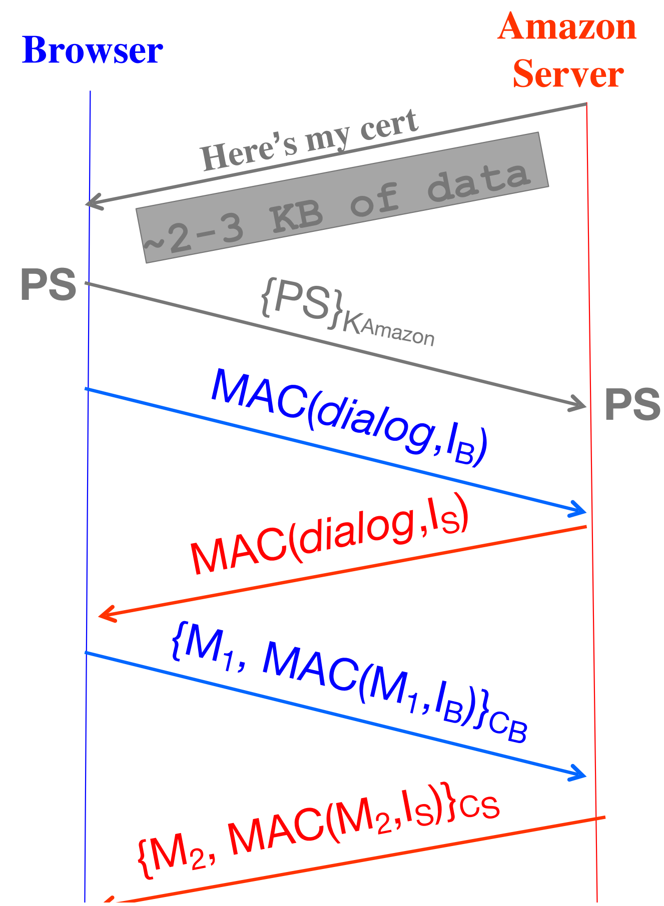
HTTPS Connection (SSL / TLS), cont.

- For RSA, browser constructs "Premaster Secret" PS
- Browser sends PS encrypted using Amazon's public RSA key K_{Amazon}
- Using PS, R_B, and R_S, browser & server derive symmetric cipher keys (C_B, C_S) & MAC integrity keys (I_B, I_S)
    - One pair to use in each direction
    - Done by seeding a pRNG in common between the browser and the server:
      Repeated calls to the pRNG then create the common keys



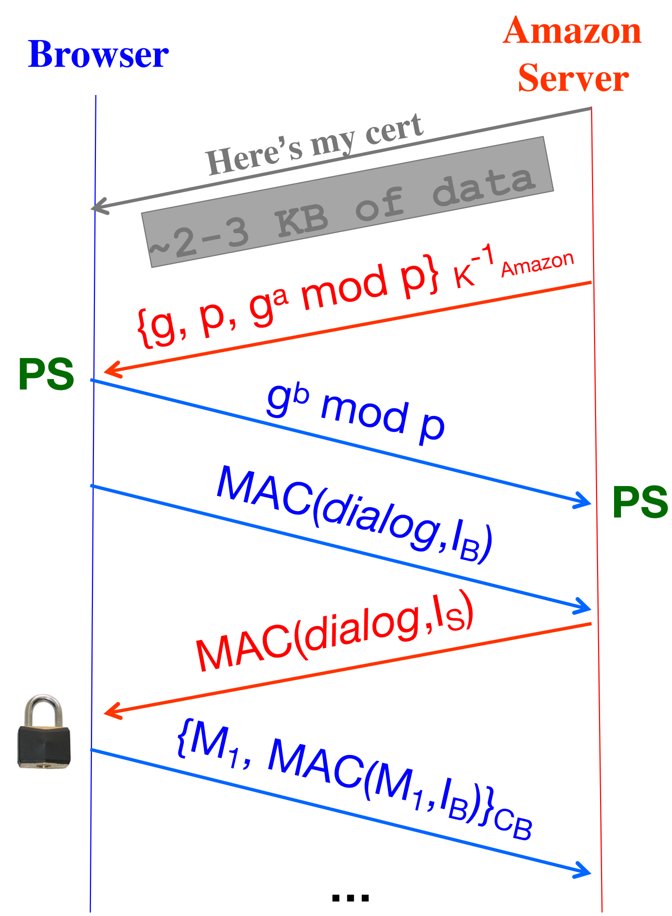
HTTPS Connection (SSL / TLS), cont.

- For RSA, browser constructs "Premaster Secret" PS
- Browser sends PS encrypted using Amazon's public RSA key KAmazon
- Using PS, R_B, and R_S, browser & server derive symm. cipher keys (C_B, C_S) & MAC integrity keys (I_B, I_S)
  - One pair to use in each direction
- Browser & server exchange MACs computed over entire dialog so far
- If good MAC, Browser displays 
- All subsequent communication encrypted w/ symmetric cipher (e.g., AES128) cipher keys, MACs
  - Sequence #'s thwart replay attacks, R_B and R_S thwart replaying handshake



Alternative: Ephemeral Key Exchange via Diffie-Hellman

- For Diffie-Hellman, server generates random a, sends public parameters and g^a mod p
  - Signed with server's private key
- Browser verifies signature
- Browser generates random b, computes PS = g^{ab} mod p, sends g^b mod p to server
- Server also computes PS = g^{ab} mod p
- Remainder is as before: from PS, R_B, and R_S, browser & server derive symm. cipher keys (C_B, C_S) and MAC integrity keys (I_B, I_S), etc…

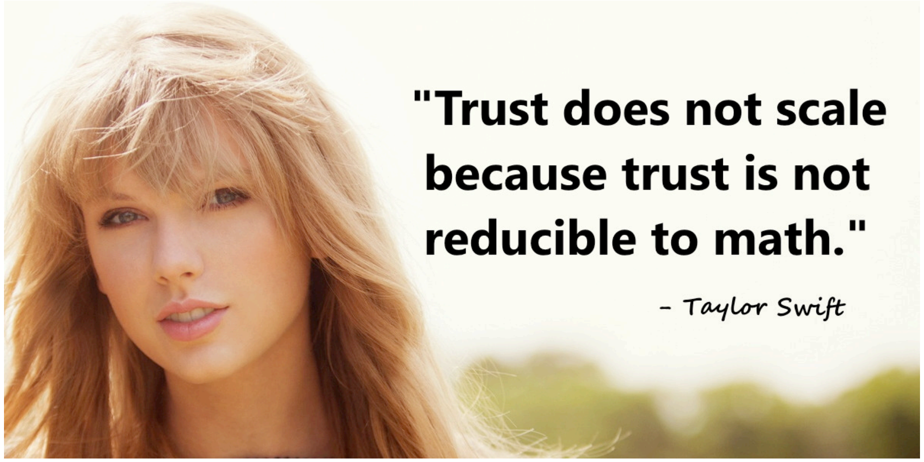


Big Changes for TLS 1.3
Diffie/Hellman and ECDHE only

- The RSA key exchange has a substantial vulnerability
  - If the attacker is ever able to compromise the server and obtain its RSA key… the attacker can decrypt any traffic captured
  - RSA lacks ***forward secrecy***
- So TLS 1.3 uses DHE/ECDHE only
  - Requires an attacker who steals the server's private keys to still be a MitM to decrypt data
- TLS 1.3 also speeds things up:
  - In the client hello, the client includes $\{g^b \bmod p\}$ for preferred parameters
    - If the server finds it suitable, the server returns $\{g^a \bmod p\}$
  - Saves a round-trip time
- Also only supports AEAD mode encryptions and limited ciphersuites (e.g. GCM)

But What About that "Certificate Validation"

- Certificate validation is used to establish a chain of "trust"
  - It actually is an ***attempt*** to build a scalable trust framework
- This is commonly known as a Public Key Infrastructure (PKI)
  - Your browser is trusting the "Certificate Authority" to be responsible…



Certificates

- Cert = signed statement about someone's public key
  - Note that a cert does not say anything about the identity of who gives you the cert
  - It simply states a given public key K_{Bob} belongs to Bob …
    - … and backs up this statement with a digital signature made using a different public/private key pair, say from Verisign (a "Certificate Authority")
- Bob then can prove his identity to you by you sending him something encrypted with K_{Bob} …
  - … which he then demonstrates he can read
- … or by signing something he demonstrably uses
- Works provided you trust that you have a valid copy of Verisign's public key …
  - … and you trust Verisign to use prudence when she signs other people's keys

Validating Amazon's Identity

- Browser compares domain name in cert w/ URL
  - Note: this provides an ***end-to-end*** property
    (as opposed to say a cert associated with an IP address)
- Browser accesses separate cert belonging to issuer
  - These are hardwired into the browser – ***and trusted***!
  - There could be a chain of these …
- Browser applies issuer's public key to verify signature **S**, obtaining the hash of what the issuer signed
  - Compares with its own SHA-1 hash of Amazon's cert
- Assuming hashes match, now have high confidence it's indeed Amazon's public key …
  - assuming signatory is trustworthy, didn't lose private key, wasn't tricked into signing someone else's certificate, and that Amazon didn't lose their key either…

End-to-End ⇒ Powerful Protections

- Attacker runs a sniffer to capture our WiFi session?
  - But: encrypted communication is unreadable
    - No problem!
- DNS cache poisoning?
  - Client goes to wrong server
  - But: detects impersonation
    - No problem!
- Attacker hijacks our connection, injects new traffic
  - But: data receiver rejects it due to failed integrity check since all communication has a mac on it
    - No problem!
- Only thing a ***full man-in-the-middle*** attacker can do is inject RSTs, inject invalid packets, or drop packets: limited to a ***denial of service***

Validating Amazon's Identity, cont.

- Browser retrieves cert belonging to the issuer
  - These are hardwired into the browser – and trusted!
- But what if the browser can't find a cert for the issuer?