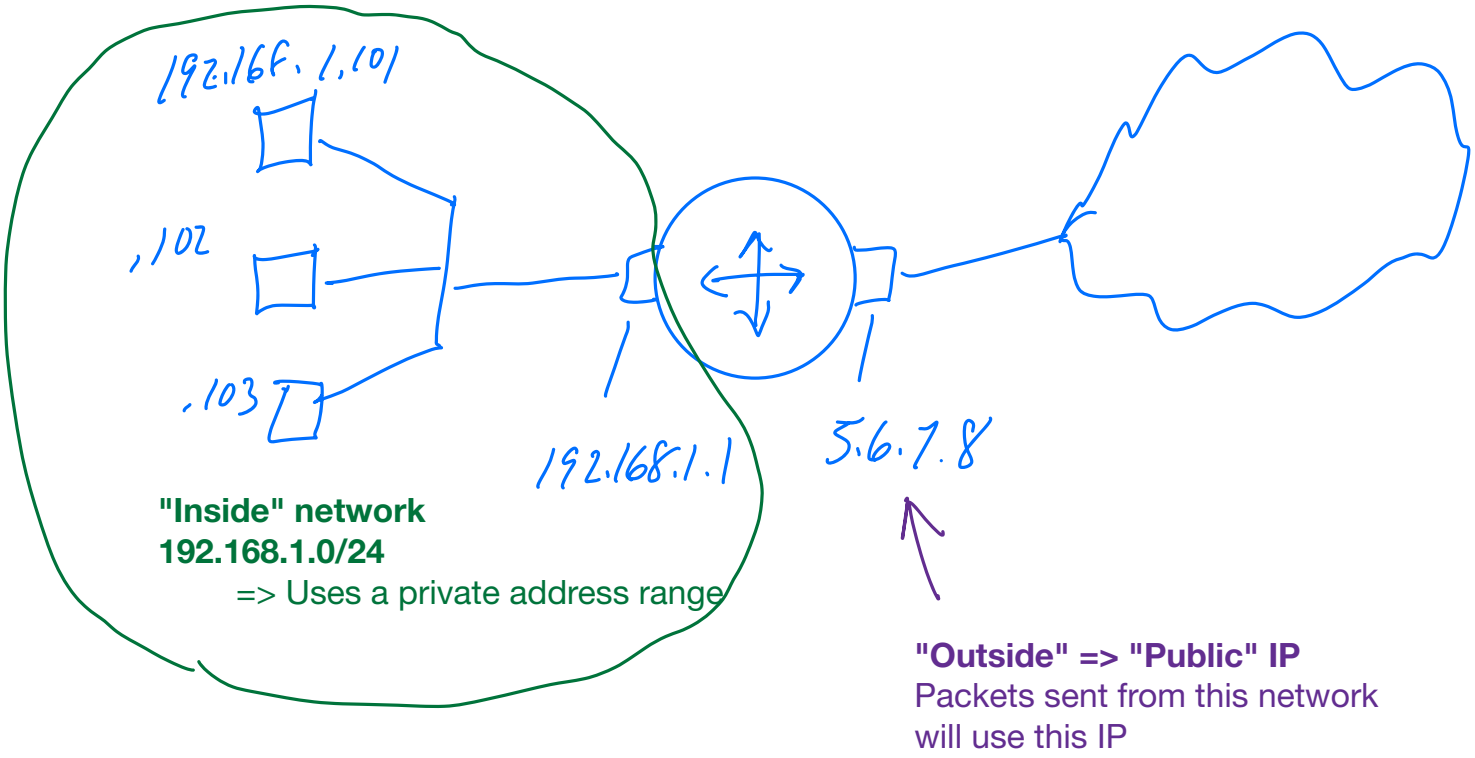
Lecture 24

Today
- NAT wrapup
- TLS

Administrivia
- Look for announcement about grading meetings
- Fill out project team form

Recap: NAT

A way to share a public IP address among multiple devices



"Private" IP ranges (RFC1918)

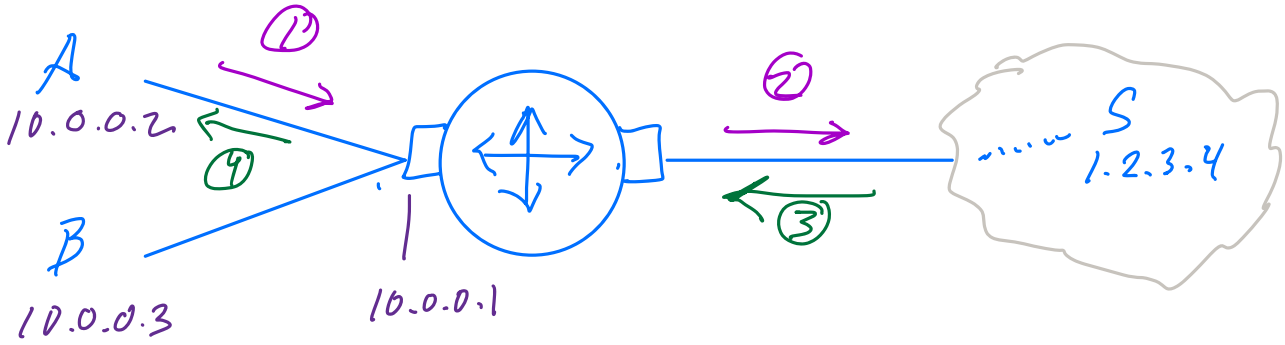
=> Should never be routed over the Internet!

Prefix	Purpose
127.0.0.1/8	LOOPBACK ADDRESS
10.0.0.0/8 ← BIG NETWORKS (BROWN WIFI)	PRIVATE NETWORKS
192.168.0.0/16	
172.16.0.0/12	

MOST HOME NETWORKS

DOCKER

NAT translation: an example



INSIDE: 10.0.0.0/24

Suppose A wants to connect to S on port 80:

	INSIDE		OUTSIDE	
	SRC	DST	SRC	DST
①	10.0.0.2:5555	1.2.3.4:80		
NAT → ②			5.6.7.8:7777	1.2.3.4:80
NAT ← ③	10.0.0.2:5555	1.2.3.4:80		
④	1.2.3.4:80		5.6.7.8:7777	

ROUTER MUST REMEMBER:

5.6.7.8:7777 → 10.0.0.2:5555

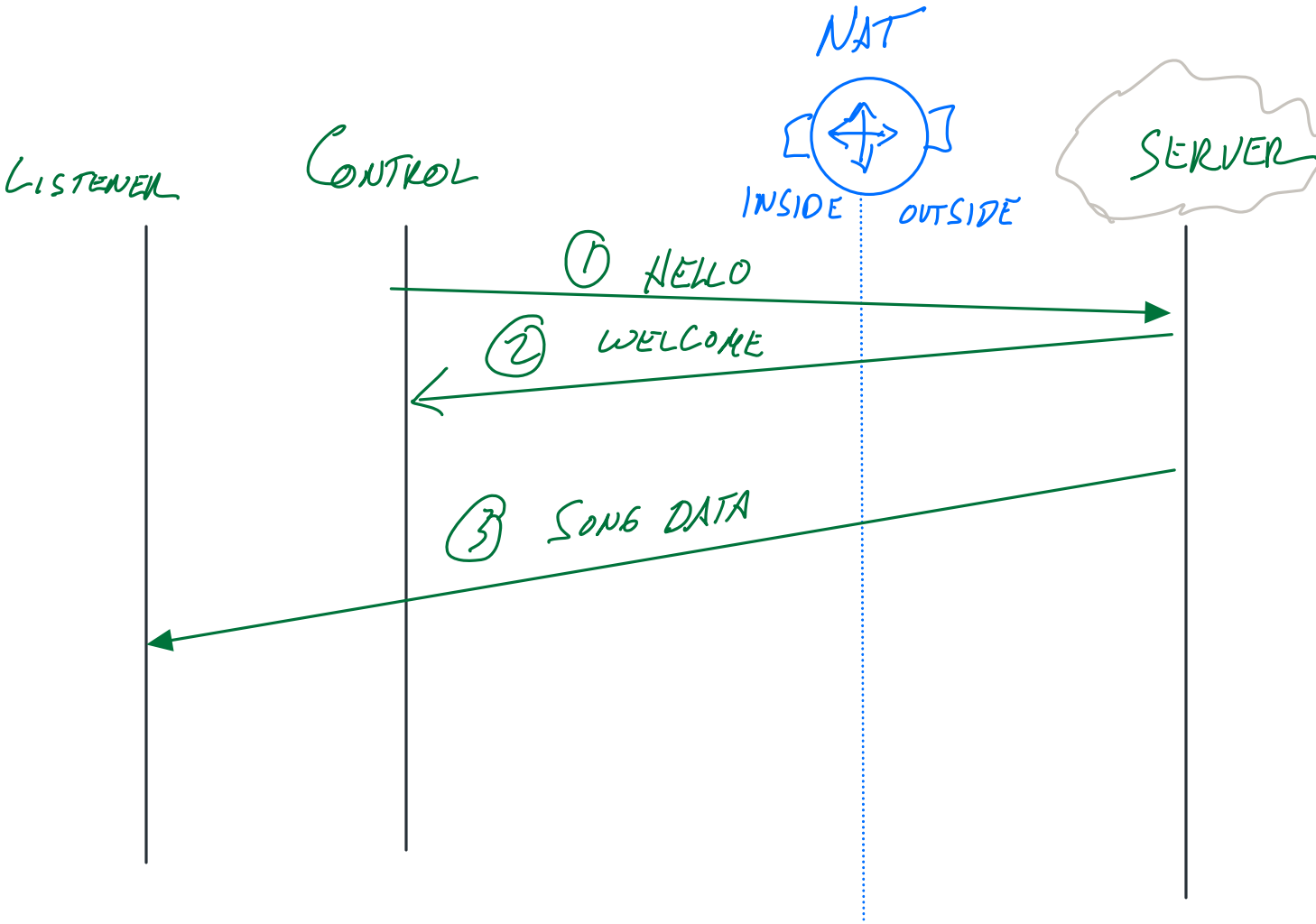
Key points

-- Router needs to keep track of state to remember how to "translate" packets

-- Can run out of possible translations (number of ports, space in table)
=> Big NATs (eg. Brown) have multiple outside IPs to increase number of possible translations

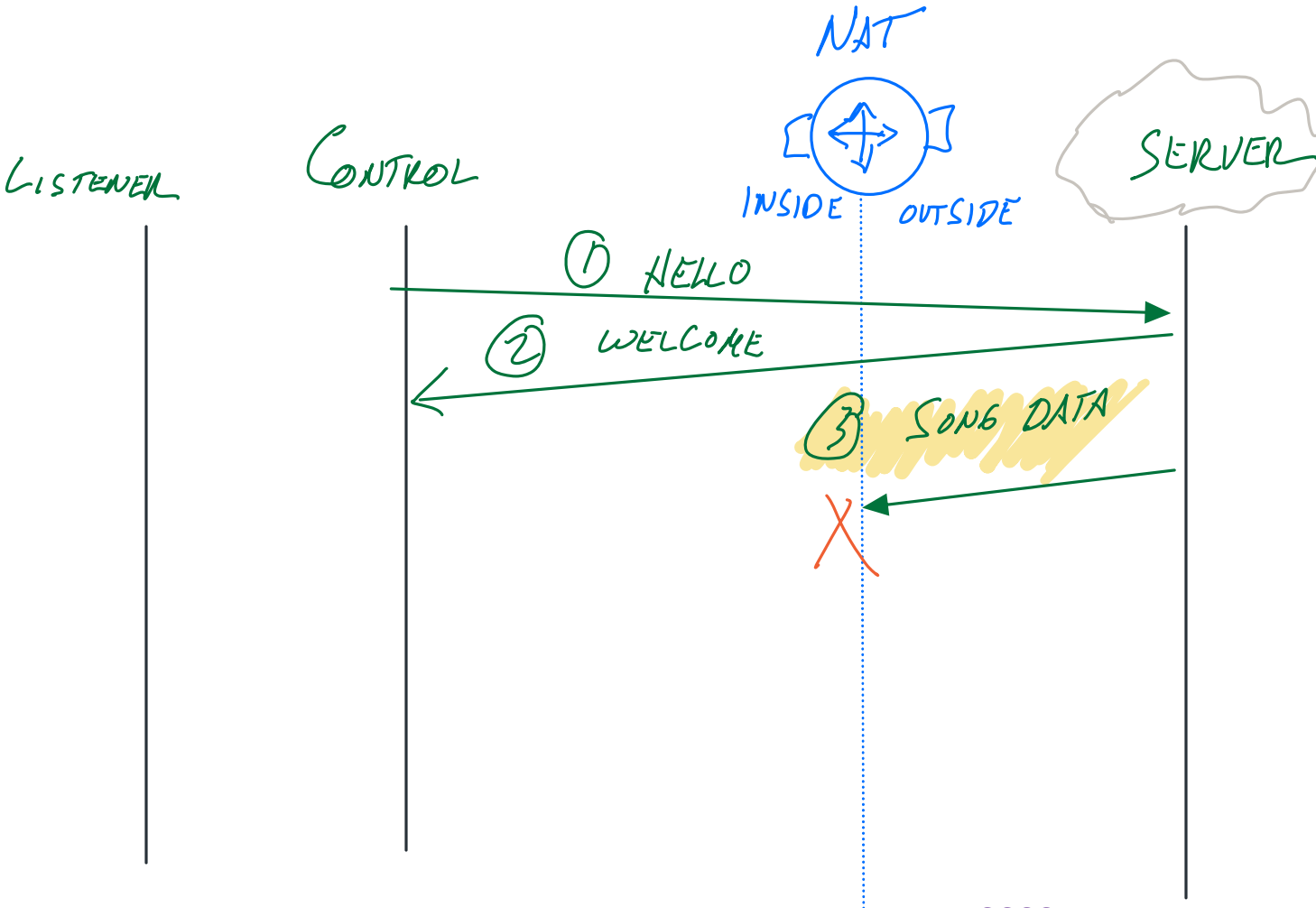
-- Router needs to figure out when connections start and end, so it can clean up table
=> Kinda works for TCP (protocol has well-defined start and end)
=> Sketchy for UDP (need to make assumptions based on timing)

NAT R. SNOWCAST



Which of these steps will not work if the client is behind a NAT????

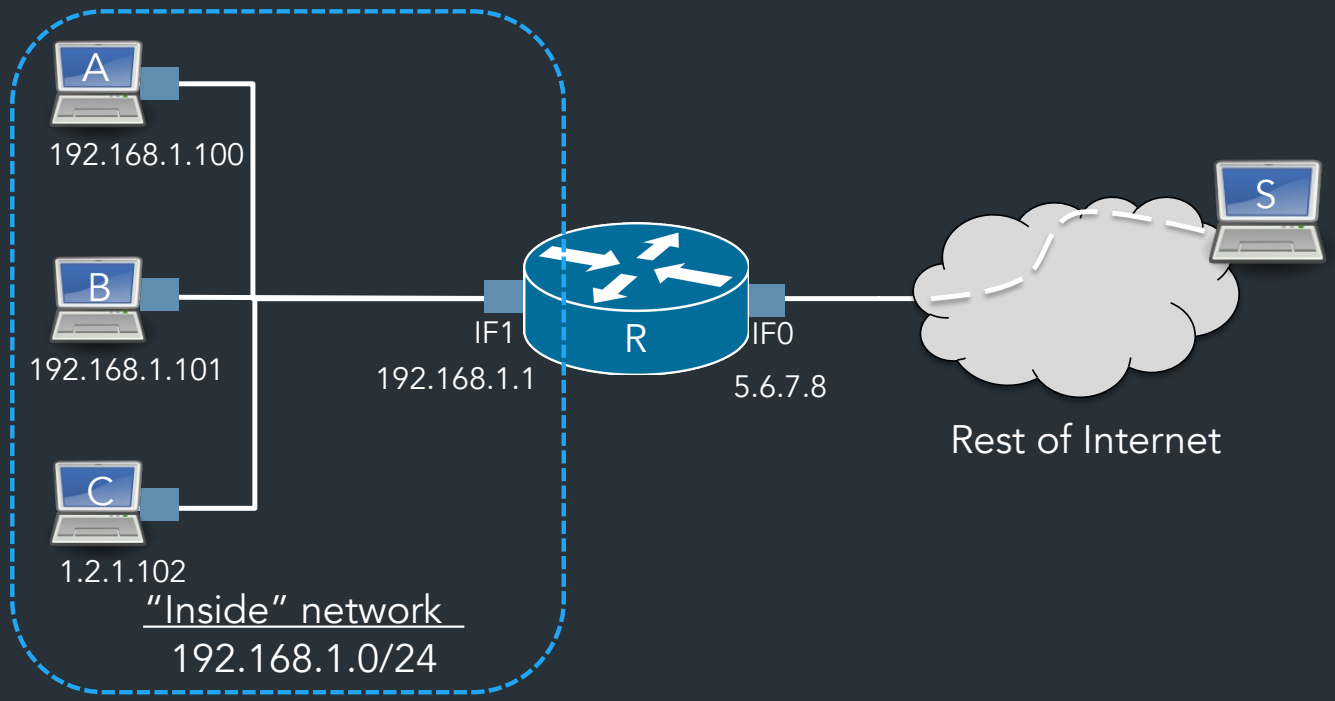
NAT R. SNOWCAST



Which of these steps will not work if the client is behind a NAT????

=> Server won't be able to send data to the listener! When the server sends a packet to the listener client, there's no translation rule for the listener's port, so the packet will get dropped!

=> In general, an outside host can't send packets to an inside host unless the inside host has made a connection first



What happens when outside host S wants to connect to inside host A?

Can't do it (at least without special setup)!

⇒ By default, R only knows how to translate packets for connections originating from INSIDE the network

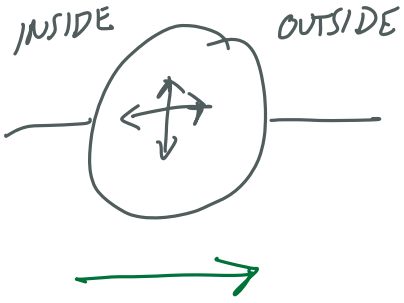
⇒ Breaks end to end connectivity!!!

Why is this bad?

NAT is used in just about every consumer network
=> In general, can't connect directly to an inside host unless it connects to you first

=> Need extra work for any protocols that need a direct connection between hosts
=> E.g., Protocols that aren't strictly client-server, or require the server to make a connection to the client (like snowcast)

=> Latency-critical applications: voice/video calls, games



How to get around this?

- Manual method: add a static mapping on the router to always forward traffic on a certain port to a certain host => "Port forwarding"

- A class of methods/protocols called **"NAT traversal"**
  Examples: ICE (RFC 8445), STUN (RFC5389)
  One idea: connect to external server via UDP, it tells you the address/port of the other host that wanted to connect to you

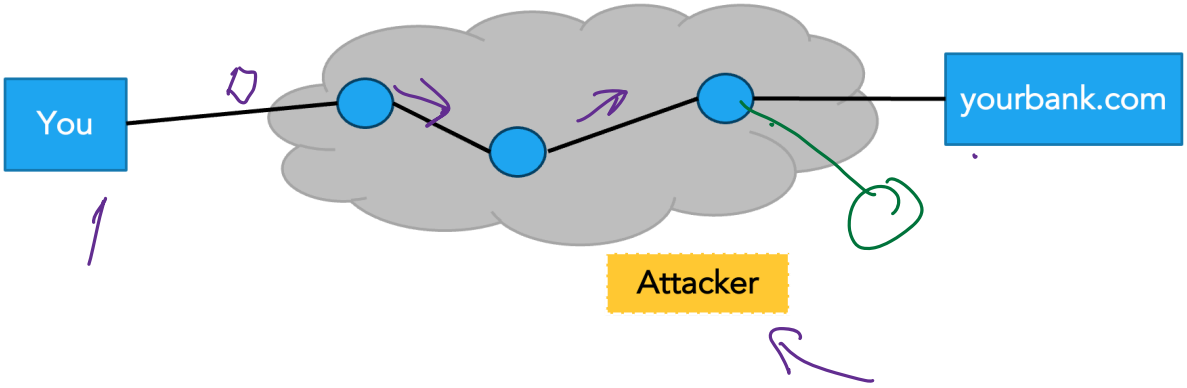
Q: Is NAT good for security?

=> NAT implicitly blocks incoming connections to internal hosts, which is often a common security policy to protect devices. However, NAT alone isn't designed to provide security. NAT routers need to make some assumptions about when to add/remove rules from their translation tables, which can be exploited by an attacker.
=> There are various techniques for this--look up "NAT hole punching" for more info

TLS

The Internet was not designed to be secure => in this class, we've seen several ways various components can be compromised...

Thinking about security: what can go wrong?



Knowing what we now know about the network, what can an attacker do to disrupt us trying to connect to some high-value website, e.g., yourbank.com?

Ideas from class
- Eavesdropping - intercepting or reading traffic not meant for us
- Drop packets/sinkhole route etc.
- IP spoofing/forge fake packets
- Denial of service

- IP prefix hijacking => influence BGP to get traffic routed to you
- DNS hijacking => make DNS queries resolve to your IP

Key security properties: We often think of security in terms of properties we want to provide. Here are some of the most important properties (and the ones that matter for TLS);

Confidentiality	`prevent adversary from reading data => prevent eavesdropping`
Integrity	`making sure messages arrive in original format`
Authentication	`verifying identity of message or entity => Protection against spoofing and impersonation`

Others: availability, provenance, anonymity, ...

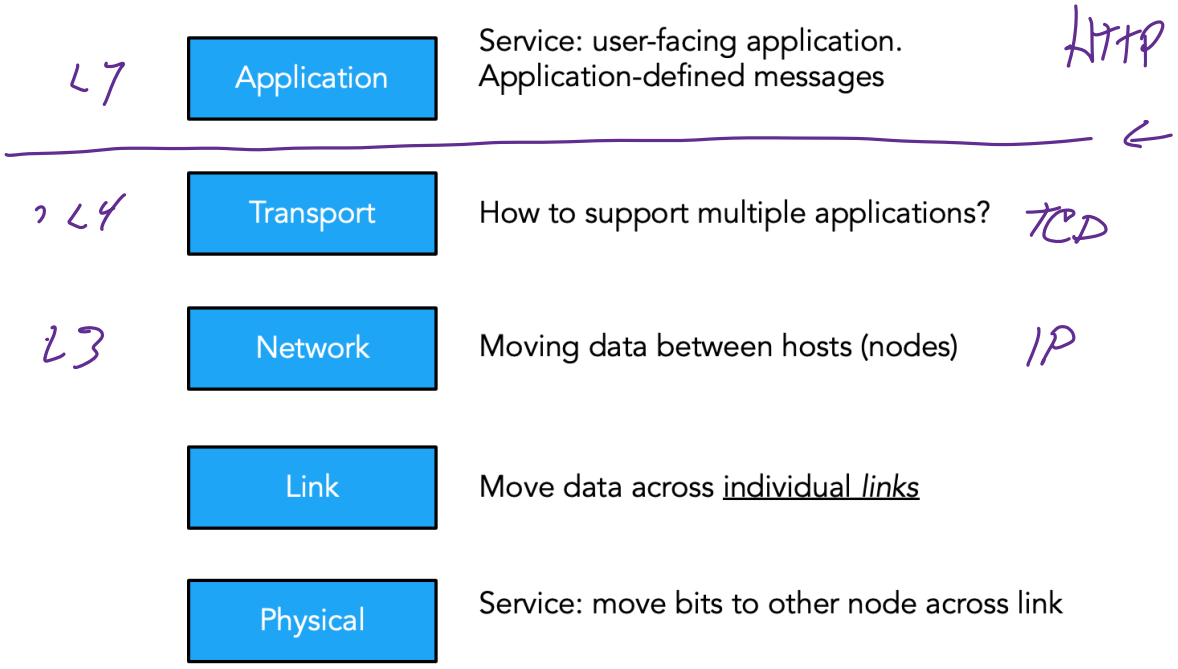
TLS: Transport Layer Security

TLS 1.0 (1999) => TLS 1.3 (2018)

Bidirectional pipe between two parties providing

- Confidentiality
- Integrity
- Authentication

Where does TLS go?



So how does TLS work?

- Builds atop TCP
- Sets up a secure connection using cryptography. Two parts to this:
    - Confidentiality: setting up a key to use to encrypt/decrypt traffic
    - Authentication: verifying the server's identity

Technical challenges (first):
=> the cryptography itself, what algorithms to use, etc.

Social challenges (second):
=> what it means to verify the server's identity
=> what to do when things go wrong

For our class: we don't need to learn the details on all of the crypto, but...
- I want you to appreciate the principles involved (what matters for TLS, and elsewhere)
- I want you to understand the social challenges (which are the most important thing in practice)

Want to know more?
- **CS1660 (Spring): Intro to Computer Systems Security**
    - Broad-spectrum security course, hands-on

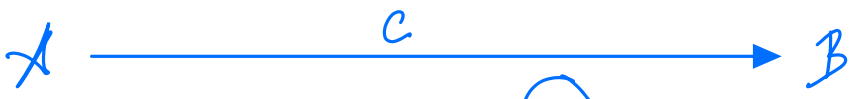
- **CS1515 (Spring): Applied Cryptography**
    - How to implement cryptographic protocols

- **CS1510 (Fall): Intro to Cryptography**
    - Underlying theory of cryptography

Brief overview: symmetric crypto

Setup:

- A wants to send message m
- A, B agree on secret key K
  => Must be exchanged beforehand using some other secure method



1

A encrypts message with key k, sends ciphertext c
c = Enc(k, m)

2

B decrypts c to recover original message
m = Dec(k, c)

This provides: confidentiality
Attacker can only see ciphertext. If encryption scheme is strong, adversary can't learn anything about m
- (Unless they can steal k somehow. . .)

Examples: AES, DES (old, insecure), Serpent, Whirlpool, …

Key properties
- Symmetric crypto is fast (relative to other crypto in our toolkit). Modern CPUs even have hardware support for AES, the most common symmetric scheme
  => Most data is protected using symmetric crypto

However: need to exchange a shared key beforehand
How to do this if you 1) can't send k over the network in the clear, and 2) need keys for everyone you might talk to?

=> There exist crypto protocols to establish a shared key without sending it over the network (beyond this course—eg. look up Diffie-Hellman Key Exchange (DHKE) for one)

=> Also need to verify the sender's identity before you start communicating (ie, how do you know if they are who they say they are?)

Brief overview: asymmetric crypto

Setup:

- **A and B each have a public/private key pair (eg. Kpub_A, Kpriv_A)**
- **Kpub is known to everyone, Kpriv is kept secret**

From here, there are two fundamental operations we can perform:

- **Encryption/Decryption: like before, but now taking advantage of two keys**
- **Signing and verification: used to verify a message came from a specific party (next page)**

I) Asymmetric encryption

1. **A encrypts message with B's public key, Kpub,B**
2. **B can decrypt with its private key Kpriv,B**

This is pretty powerful: can encrypt a message for B without a shared key!
However: asymmetric crypto is very slow (orders or magnitude slower than symmetric crypto)

=> Used for authenticating at connection start (more on this in a moment)
=> Use asymmetric crypto to establish a symmetric key at session start, use for rest of connection

II) Signatures and verification

Idea: want to use public key crypto to verify a message came from a certain party

1. **A signs message (or hash of message) with its private key**
   => Produces a signature: a small value like a hash

2. **B can verify the signature using A's public key**
   => Outputs a true if message was signed with Kpriv_A, otherwise false

s = Sign(Kriv_A, m) **b = Verify(Kpub_A, m, s)**

Also pretty powerful: anyone can verify that m was signed by A's private key
=> Assuming that A's private key is indeed private (only A has it), this means only A could have signed the message

Some notes

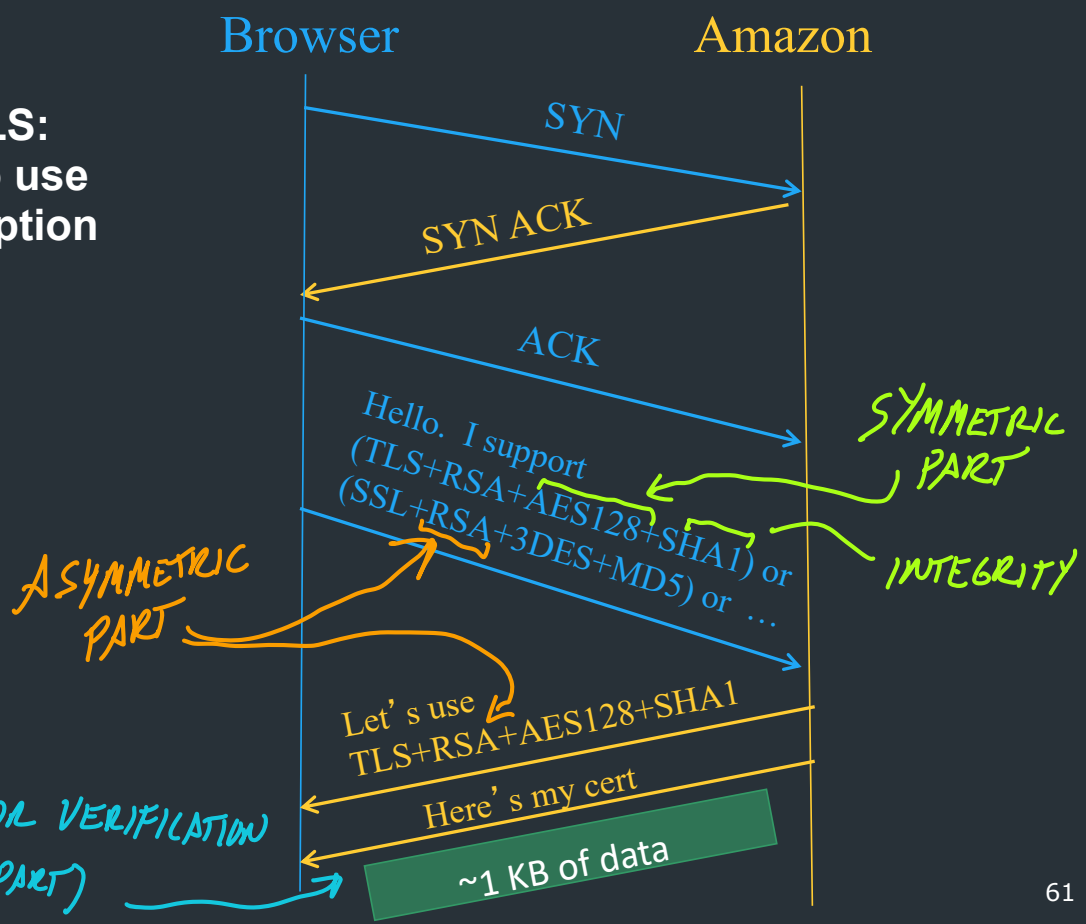
- This is also slow, like asymmetric encryption/decryption
- Could also encrypt the message, but also important to sign public info: for example, an open-source developer might sign a software update to prove it's legitimate

- *(not necessary for this class) Signing has another important crypto property: non-repudiation, meaning A can't prove it didn't sign message m*

TLS: setup

First step: TCP handshake

Then, a separate handshake step for TLS:
- **Need to agree on which algorithms to use**
- **Set up a session key to use for encryption**



But there's one more part: authentication => how do we verify the server is who they claim they are?

A list of crypto algorithms used in TLS
Not all clients/servers support all algorithms, so need to agree on the best available to both...

SYMMETRIC PART.

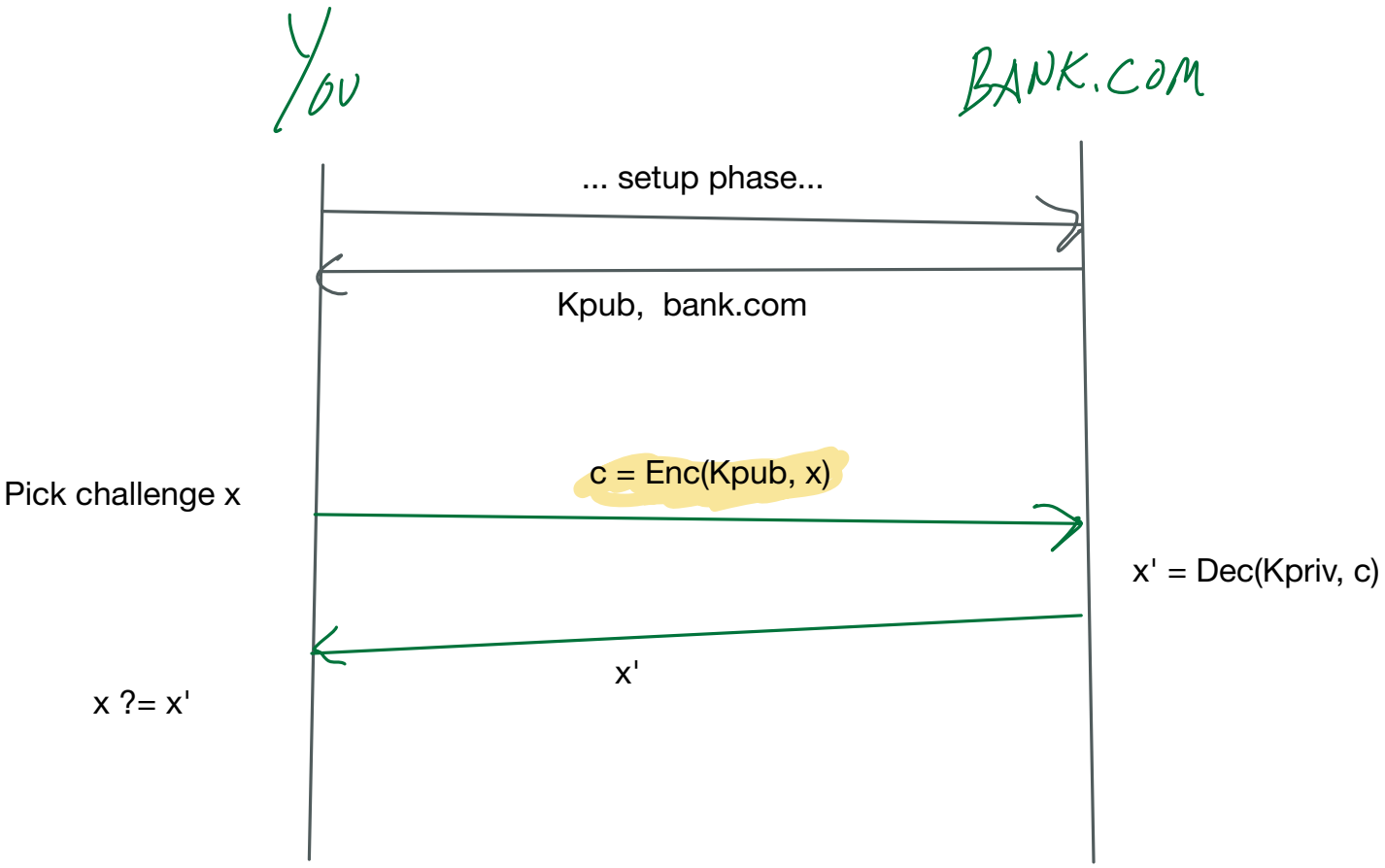
0x00,0xA0	TLS_DH_RSA_WITH_AES_128_GCM_SHA256	Y	N	[RFC5288]
0x00,0xA1	TLS_DH_RSA_WITH_AES_256_GCM_SHA384	Y	N	[RFC5288]
0x00,0xA2	TLS_DHE_DSS_WITH_AES_128_GCM_SHA256	Y	N	[RFC5288]
0x00,0xA3	TLS_DHE_DSS_WITH_AES_256_GCM_SHA384	Y	N	[RFC5288]
0x00,0xA4	TLS_DH_DSS_WITH_AES_128_GCM_SHA256	Y	N	[RFC5288]
0x00,0xA5	TLS_DH_DSS_WITH_AES_256_GCM_SHA384	Y	N	[RFC5288]
0x00,0xA6	TLS_DH_anon_WITH_AES_128_GCM_SHA256	Y	N	[RFC5288]
0x00,0xA7	TLS_DH_anon_WITH_AES_256_GCM_SHA384	Y	N	[RFC5288]
0x00,0xA8	TLS_PSK_WITH_AES_128_GCM_SHA256	Y	N	[RFC5487]
0x00,0xA9	TLS_PSK_WITH_AES_256_GCM_SHA384	Y	N	[RFC5487]
0x00,0xAA	TLS_DHE_PSK_WITH_AES_128_GCM_SHA256	Y	Y	[RFC5487]
0x00,0xAB	TLS_DHE_PSK_WITH_AES_256_GCM_SHA384	Y	Y	[RFC5487]
0x00,0xAC	TLS_RSA_PSK_WITH_AES_128_GCM_SHA256	Y	N	[RFC5487]
0x00,0xAD	TLS_RSA_PSK_WITH_AES_256_GCM_SHA384	Y	N	[RFC5487]
0x00,0xAE	TLS_PSK_WITH_AES_128_CBC_SHA256	Y	N	[RFC5487]
0x00,0xAF	TLS_PSK_WITH_AES_256_CBC_SHA384	Y	N	[RFC5487]

ASYMMETRIC PART

TLS authentication: how do we verify the entity on the other end of the connection is who they claim to be?

=> This is the most conceptually important part

Technical part: send a "challenge" to the server, ask server to decrypt it with its private key



What have we learned from this???

=> Challenge proves that the server for bank.com holds Kpriv

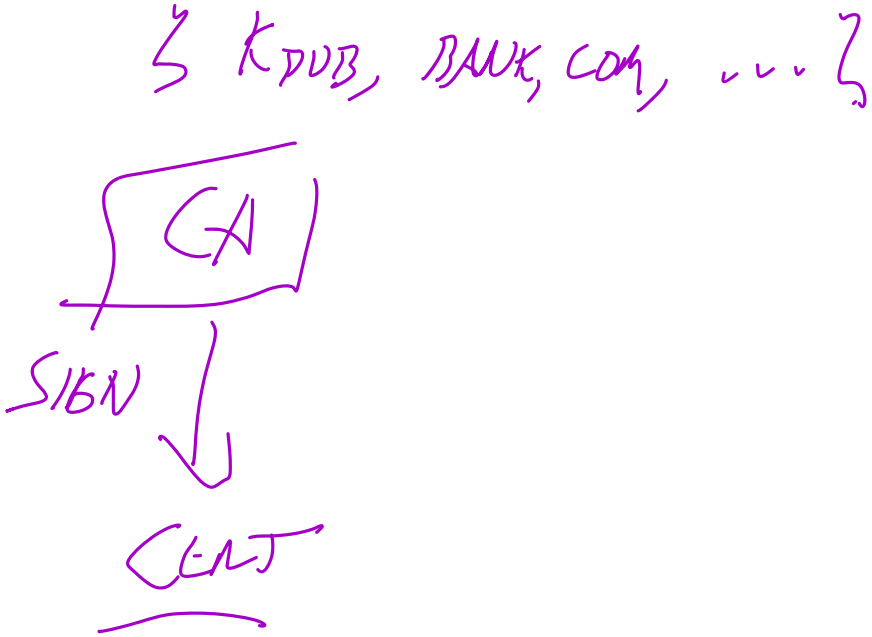
=> Does NOT prove that the server belongs to your bank, the real-life bank with your money

How can we trust K_pub is your bank's public key?

Trust distribution
=> Hard to do in real life
=> Hard to scale

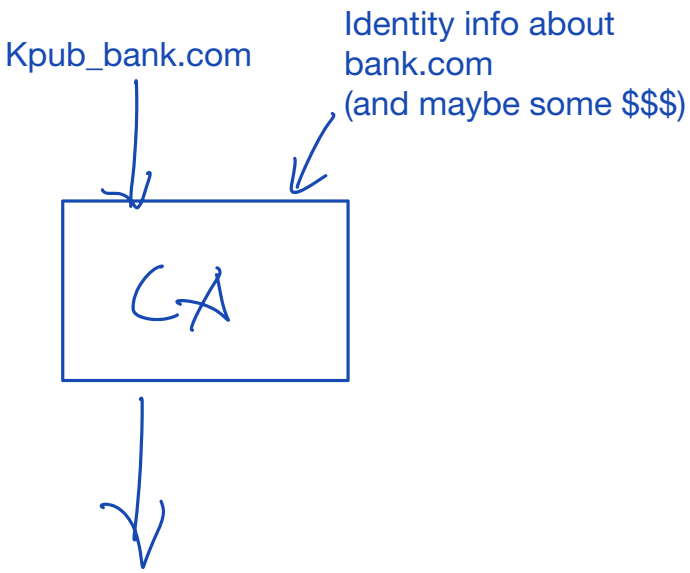
TLS relies on PKI (Public Key Infrastructure)
Idea: "delegated trust"
=> can't trust everyone, can't trust everyone, so trust a few well-known entities to verify everyone else
=> Certificate Authorities (CAs)

If a trusted CAs trusts them => OK



PKI - the gist: We didn't get as far as we would have liked in lecture, so here's the gist. See the next few pages for more notes (which we'll talk about next class).

Setup:
- Everyone has Kpub of CAs they trust (installed on their computer, in their browser)
- bank.com asks CA for a *certificate*



CA verifies bank.com's identity, signs their public key with the CA's private key:

s = Sign(Kpriv_CA, Kpub_bank, metadata)

Issues *certificate* to website:

cert = {Kpub_bank, s}

The certificate proves that the CA, whom we we trust, signed bank.com's public key. When we connect to the server, it presents the certificate to the client, who can verify it was signed by the CA. If CA did their job in checking bank.com's identity, this means we can trust that the Kpub in the certificate belongs to the real bank.com.

What's next? Here's what you'll find in the rest of the notes (and what we'll discuss next class):

- Certificates also contain metadata (e.g., domain name of the server, validity dates, etc.), which also signed. Your browser checks this info during TLS setup; if any part is wrong, validation fails.

- There's actually a *hierarchy* of CAs, to help protect against a CA getting compromised. We'll talk more about this!

- Some cool demos and case studies 😎

Read on for more info, or tune in for next lecture!

Preview notes on CAs and PKI....

We'll talk about this next class

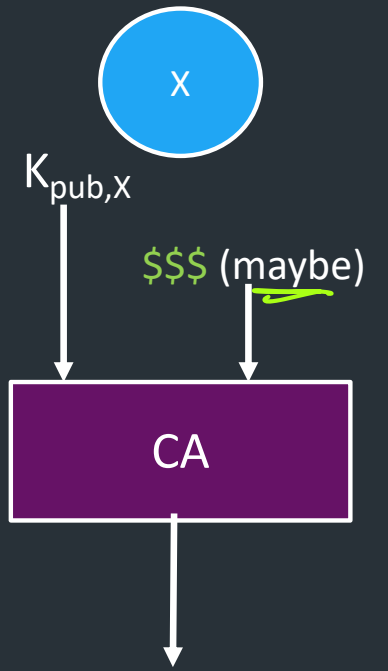
Problem: How can we trust K_pub is Your Bank's public key?

PKI: The main idea

Public keys managed by Certificate Authorities (CAs)

- Everyone knows public key for some root CAs
  - Pre-installed into browser/OS
- If X wants a public key, request from CA
  - CA validates X's identity => if OK signs X's public key
  - Generates certificate
- Client can verify $K_{pub,X}$ from CA's signature:

  Verify($K_{pub,CA}$ Cert) => True/False



=> Delegates trust for individual entity to a more trusted authority

What's in a certificate?

- Public key of entity (eg. yourbank.com)
- Common name: DNS name of server (yourbank.com)
- Contact info for organization
- VALIDITY DATES.
- ...

K_{PUB}

↳ OR A LIST OF NAMES

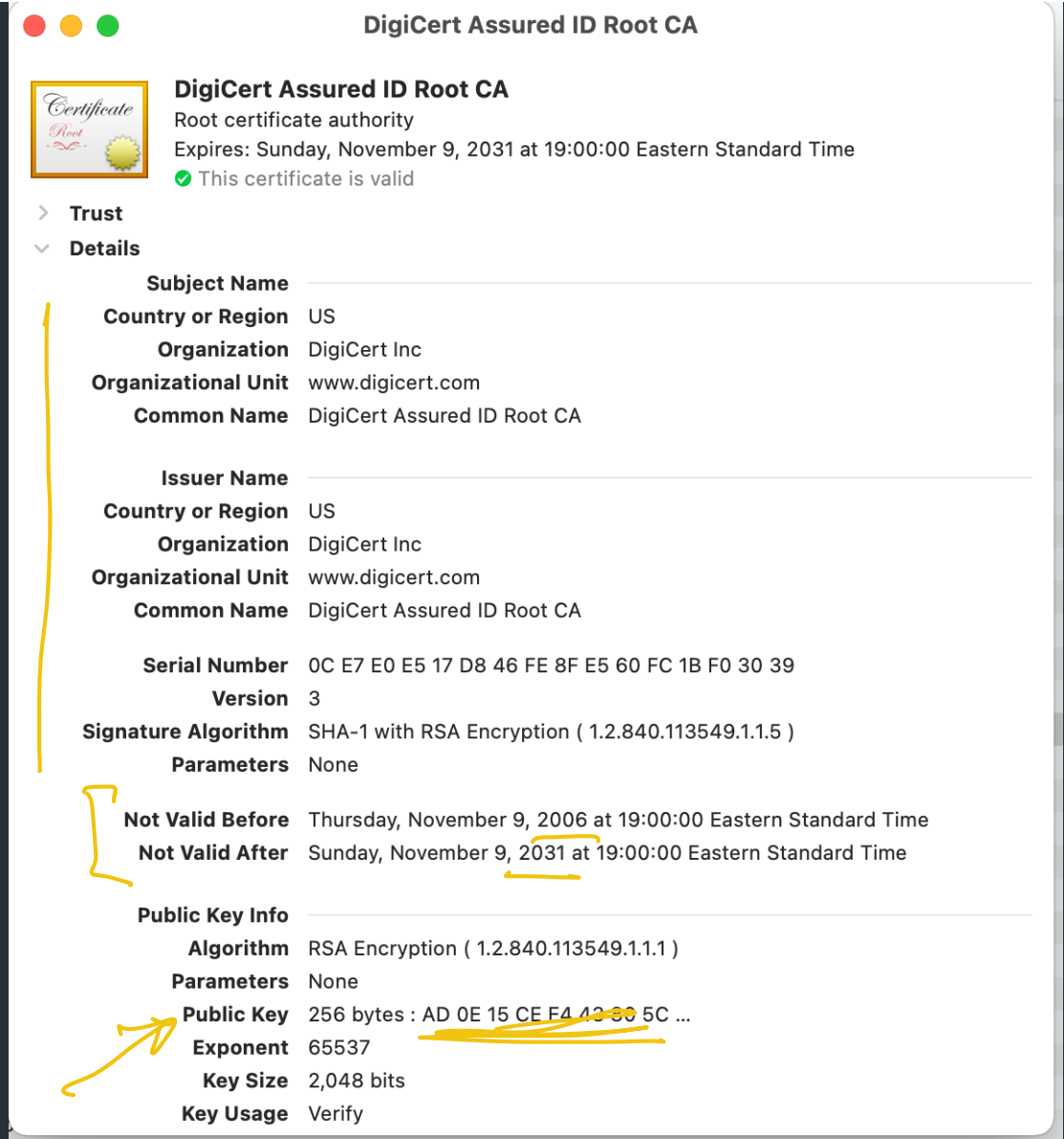
ALL SIGNED BY CA

CERT IS INVALID IF ANY METADATA ALTERED!

What's in a certificate?

- Public key of entity (eg. yourbank.com)
- Common name: DNS name of server (yourbank.com)
- Contact info for organization
- Validity dates (start date, expire date)
- URL of *revocation center* to check if key has been revoked

All of this is part of the data signed by the CA
=> Critical to check all parts during TLS startup!



Given Cert = {Kpub,bank, s, ..}

Browser will: Verify(s, Kpub,CA)

↳

{T, F}

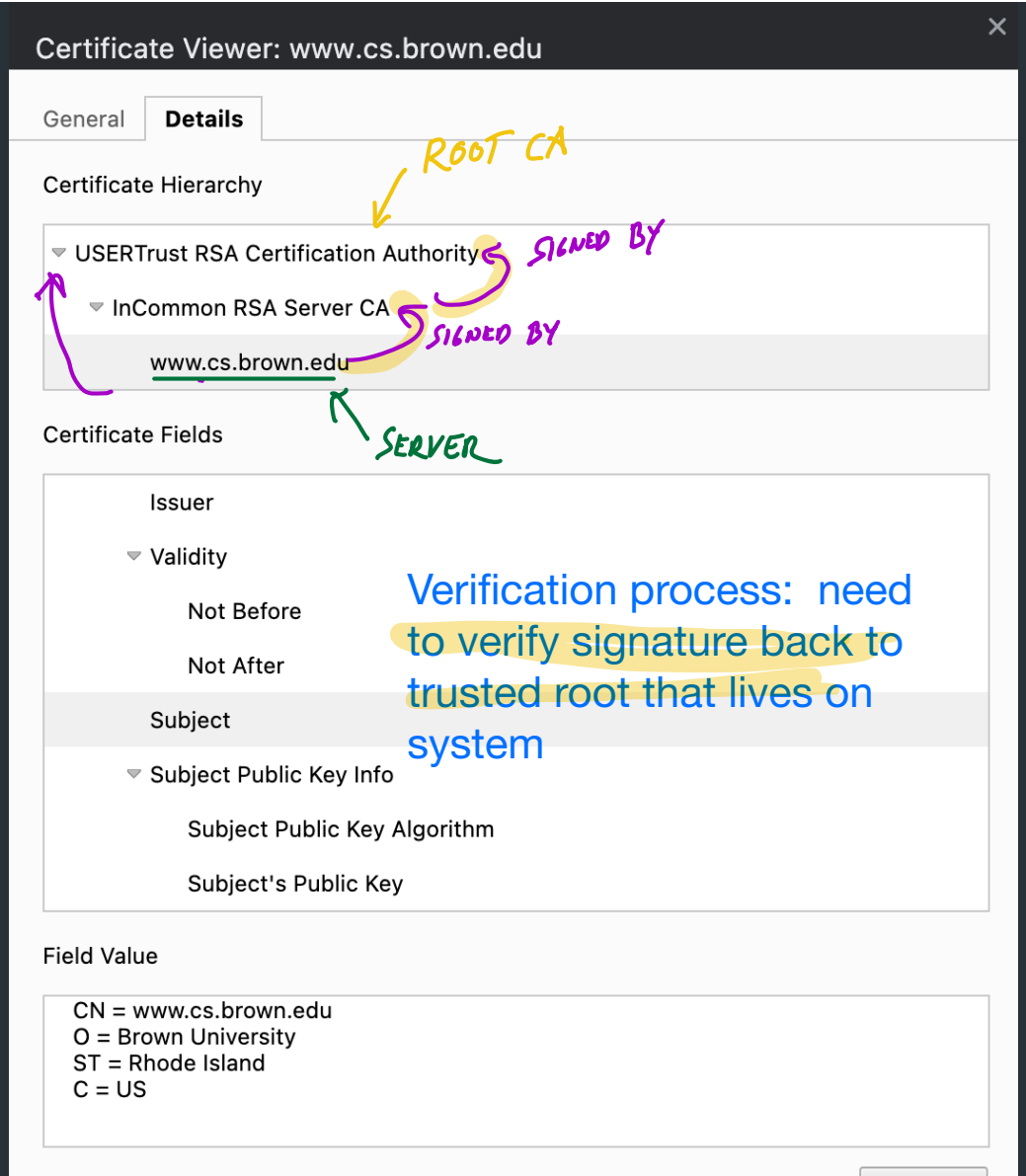
Keychain Access

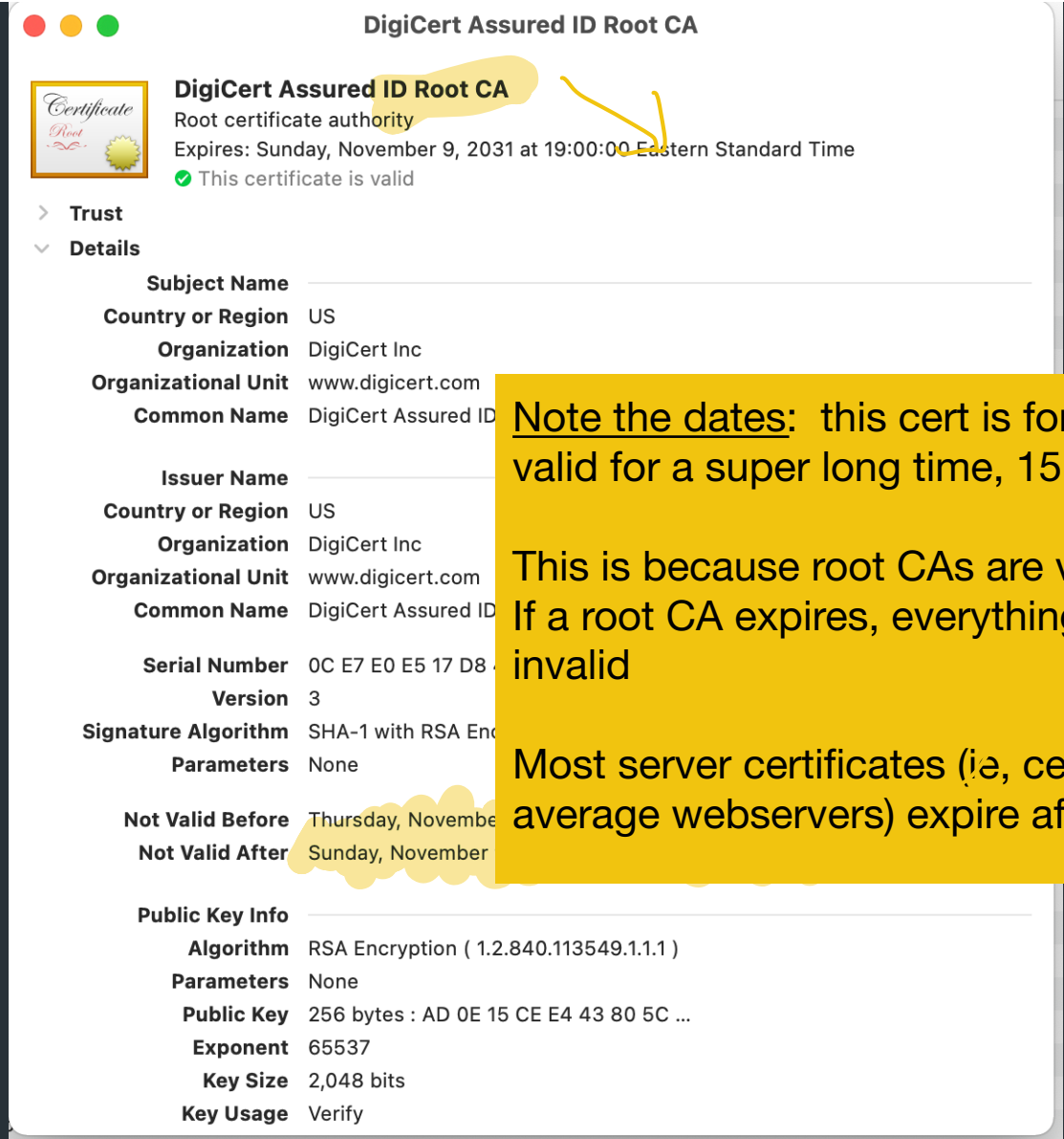
Search

All Items | Passwords | Secure Notes | My Certificates | Keys | Certificates

Amazon Root CA 1
Root certificate authority
Expires: Saturday, January 16, 2038 at 19:00:00 Eastern Standard Time
This certificate is valid

Name	Kind	Date Modified	Expires	Keychain
AAA Certificate Services	certificate	--	Dec 31, 2028 at 18:59:59	System Roots
AC RAIZ FNMT-RCM	certificate	--	Dec 31, 2029 at 19:00:00	System Roots
Actalis Authentication Root CA	certificate	--	Sep 22, 2030 at 07:22:02	System Roots
AffirmTrust Commercial	certificate	--	Dec 31, 2030 at 09:06:06	System Roots
AffirmTrust Networking	certificate	--	Dec 31, 2030 at 09:08:24	System Roots
AffirmTrust Premium	certificate	--	Dec 31, 2040 at 09:10:36	System Roots
AffirmTrust Premium ECC	certificate	--	Dec 31, 2040 at 09:20:24	System Roots
Amazon Root CA 1	certificate	--	Jan 16, 2038 at 19:00:00	System Roots
Amazon Root CA 2	certificate	--	May 25, 2040 at 20:00:00	System Roots
Amazon Root CA 3	certificate	--	May 25, 2040 at 20:00:00	System Roots
Amazon Root CA 4	certificate	--	May 25, 2040 at 20:00:00	System Roots
ANF Global Root CA	certificate	--	Jun 5, 2033 at 13:45:38	System Roots
Apple Root CA	certificate	--	Feb 9, 2035 at 16:40:36	System Roots
Apple Root CA - G2	certificate	--	Apr 30, 2039 at 14:10:09	System Roots
Apple Root CA - G3	certificate	--	Apr 30, 2039 at 14:19:06	System Roots
Apple Root Certificate Authority	certificate	--	Feb 9, 2025 at 19:18:14	System Roots
Atos TrustedRoot 2011	certificate	--	Dec 31, 2030 at 18:59:59	System Roots
Autoridad de Certificacion Firmaprofesional CIF A62634068	certificate	--	Dec 31, 2030 at 03:38:15	System Roots
Autoridad de Certificacion Raiz del Estado Venezolano	certificate	--	Dec 17, 2030 at 18:59:59	System Roots
Baltimore CyberTrust Root	certificate	--	May 12, 2025 at 19:59:00	System Roots
Buypass Class 2 Root CA	certificate	--	Oct 26, 2040 at 04:38:03	System Roots
Buypass Class 3 Root CA	certificate	--	Oct 26, 2040 at 04:28:58	System Roots
CA Disig Root R1	certificate	--	Jul 19, 2042 at 05:06:56	System Roots
CA Disig Root R2	certificate	--	Jul 19, 2042 at 05:15:30	System Roots
Certigna	certificate	--	Jun 29, 2027 at 11:13:05	System Roots
Certinomis - Autorité Racine	certificate	--	Sep 17, 2028 at 04:28:59	System Roots
Certinomis - Root CA	certificate	--	Oct 21, 2033 at 05:17:18	System Roots
Certplus Root CA G1	certificate	--	Jan 14, 2038 at 19:00:00	System Roots
Certplus Root CA G2	certificate	--	Jan 14, 2038 at 19:00:00	System Roots
certSIGN ROOT CA	certificate	--	Jul 4, 2031 at 13:20:04	System Roots
Certum CA	certificate	--	Jun 11, 2027 at 06:46:39	System Roots
Certum Trusted Network CA	certificate	--	Dec 31, 2029 at 07:07:37	System Roots


Certificate Viewer: www.cs.brown.edu
General
Details
Certificate Hierarchy
ROOT CA
USERTrust RSA Certification Authority
SIGNED BY
InCommon RSA Server CA
SIGNED BY
www.cs.brown.edu
SERVER
Certificate Fields
Issuer
Validity
Not Before
Not After
Subject
Subject Public Key Info
Subject Public Key Algorithm
Subject's Public Key
Verification process: need to verify signature back to trusted root that lives on system
Field Value
CN = www.cs.brown.edu
O = Brown University
ST = Rhode Island
C = US



Note the dates: this cert is for a root CA, so it's valid for a super long time, 15 years!

This is because root CAs are very hard to change. If a root CA expires, everything signed by it is invalid

Most server certificates (ie, certs installed on average webservers) expire after 1 year, or less

PKI hierarchy

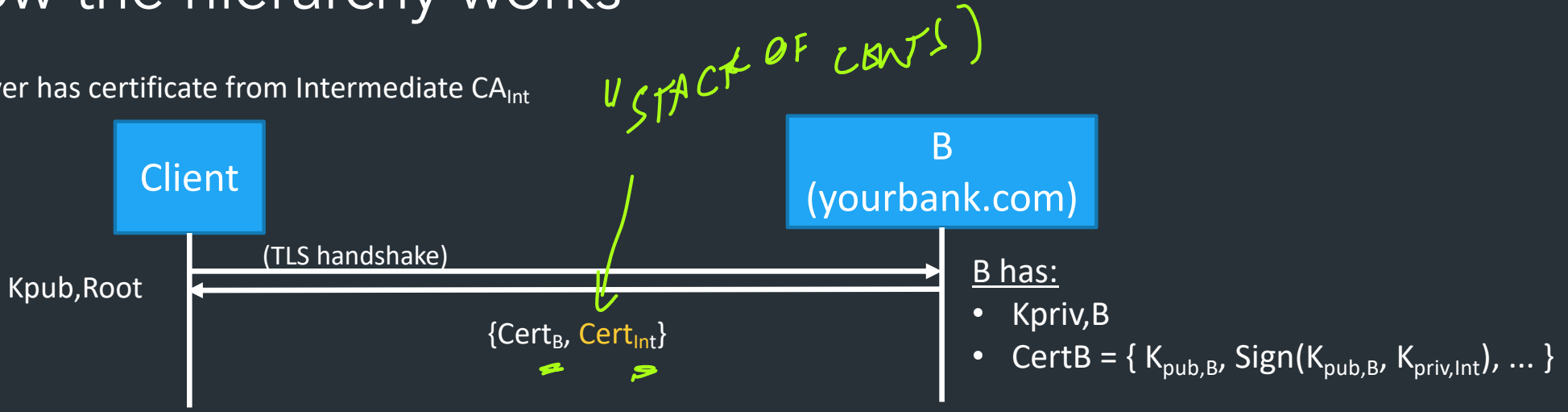
In reality, PKI creates a hierarchy of trust:

- Root CAs: k_{pub} stored in virtually every browser, OS
  - Private keys protected by most stringent security measures (software, hardware, physical)
- Intermediate CAs: k_{pub} signed by root CA
  - Sign certificates for general use (ie, regular websites)
  - Doesn't require same protections as root
- General-use certificates: for a specific webserver

What happens if a root is compromised?

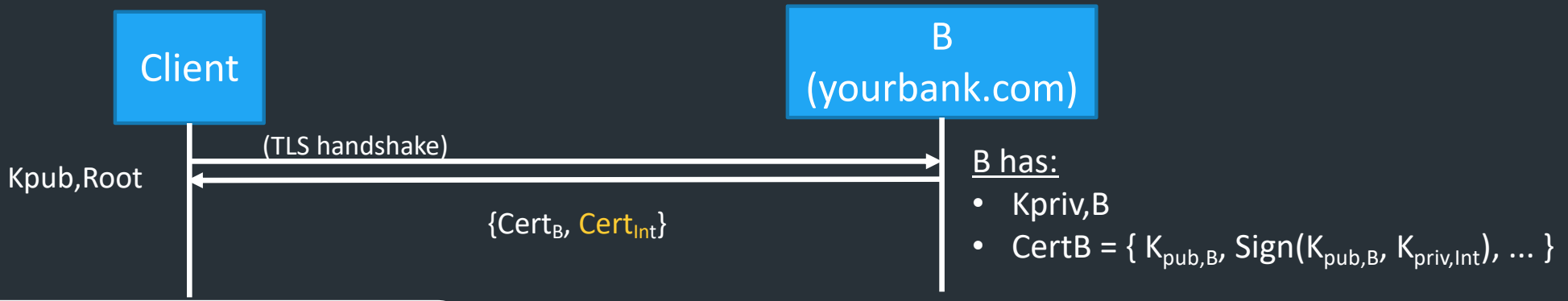
How the hierarchy works

Ex. Server has certificate from Intermediate CA_{Int}



How the hierarchy works

Ex. Server has certificate from Intermediate CA_{Int}

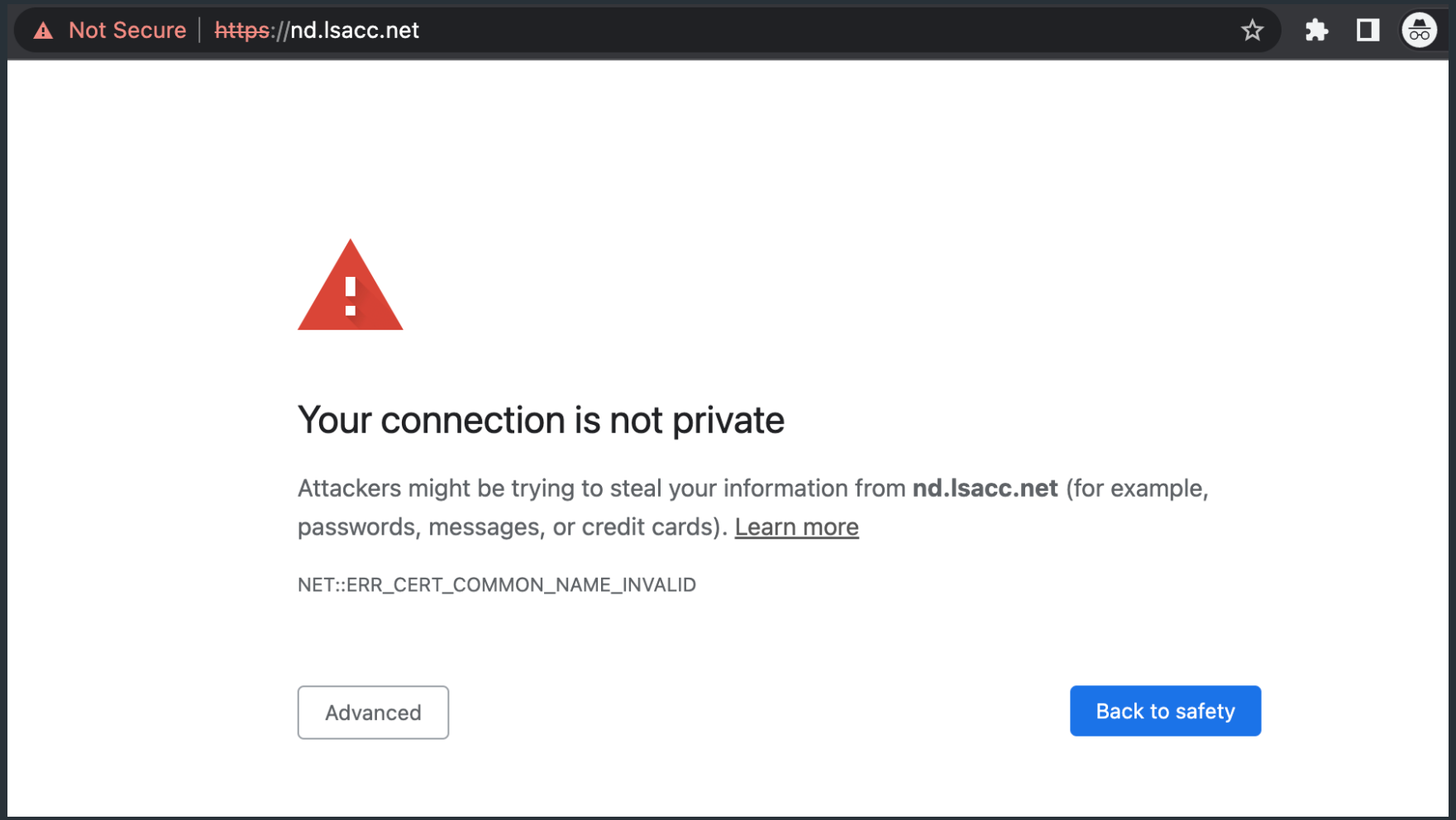


Client's workflow:

- Checks metadata ✅
- Verify($Cert_B$, $K_{pub,Int}$) ✅
- Verify($Cert_{Int}$, $K_{pub,Root}$) ✅

=> To verify integrity, need to verify certificates back to (trusted) root certificate

=> OK if verification passes and metadata correct: 🔒


Not Secure | https://nd.lsacc.net
Your connection is not private
Attackers might be trying to steal your information from nd.lsacc.net (for example, passwords, messages, or credit cards). Learn more
NET::ERR_CERT_COMMON_NAME_INVALID
Advanced
Back to safety

Most common TLS errors you might see

- Common name (eg. yourbank.com) invalid
- Certificate expired => 3mos – 1yr for server certs
- Bad chain of trust (can't verify back to trusted root)

=> Usually a sign of something sketchy, or something wrong with the webserver

When is it okay to click "proceed"? What happens if you do?

=> *Might occur if webserver configured properly, or if you're setting up a system, but not okay for your bank (or Brown ...)*

Most common TLS errors you might see

- Common name (eg. yourbank.com) invalid
- Certificate expired
- Bad chain of trust (can't verify to trusted root cert)
- "Certificate is self-signed"???

Kpub,X
CertX = {Kpub,X, Sign(Kpub,X, Kpriv,X)}

Self-signed: certificate that signs itself
=> Common for demo services
=> Root CAs are self-signed (that's okay because we trust them)

Q: are there other methods of delegating trust?

- Web of trust: small group of parties that sign each other's keys
  => Have a threshold on how many signatures you need to be "trusted"
  => Doesn't scale to entire internet, but exists for small communities (esp. open-source software projects)

- Trust on first use (TOFU)
  - ON first connection, ask user if they trust the public key (y/n)
  - If user says yes, trust key for all time
  - If public key changes later, something sketchy is happening => trust error
  => SSH (by default)

Also: PKI comes up in other ways outside of TLS:
- DNSSEC has a similar hierarchy (root zone ~= trusted CA)
- Similar certificates used for secure email (S/MIME) or some other related authentication standards