CS-340 Introduction to Computer Networking

Lecture 16: Authentication

Steve Tarzia

Last Lecture: Encryption and Anonymity

- Network security goals are:
  - **Confidentiality, Reliability, Integrity, Authentication & Anonymity**
- Routers and other participants on the network cannot be trusted.
- **AES** is a the standard **symmetric-key** encryption algorithm. Must somehow establish a shared session key, used by both parties.
- Public Key cryptography (**RSA**, ECC) uses a pair of related keys.
  - **Public key** is openly advertised and is used for encryption
  - **Private key** is secret and is used for decryption.
- Onion-routing/mix networks create routing **overlays** on the Internet.
  - Sender encrypts data many times. Relays decrypt one layer each.
  - This enables **anonymous** web browsing and even anonymous services.

Authentication definition

- Verifying the identity of the person/host I'm communicating with.

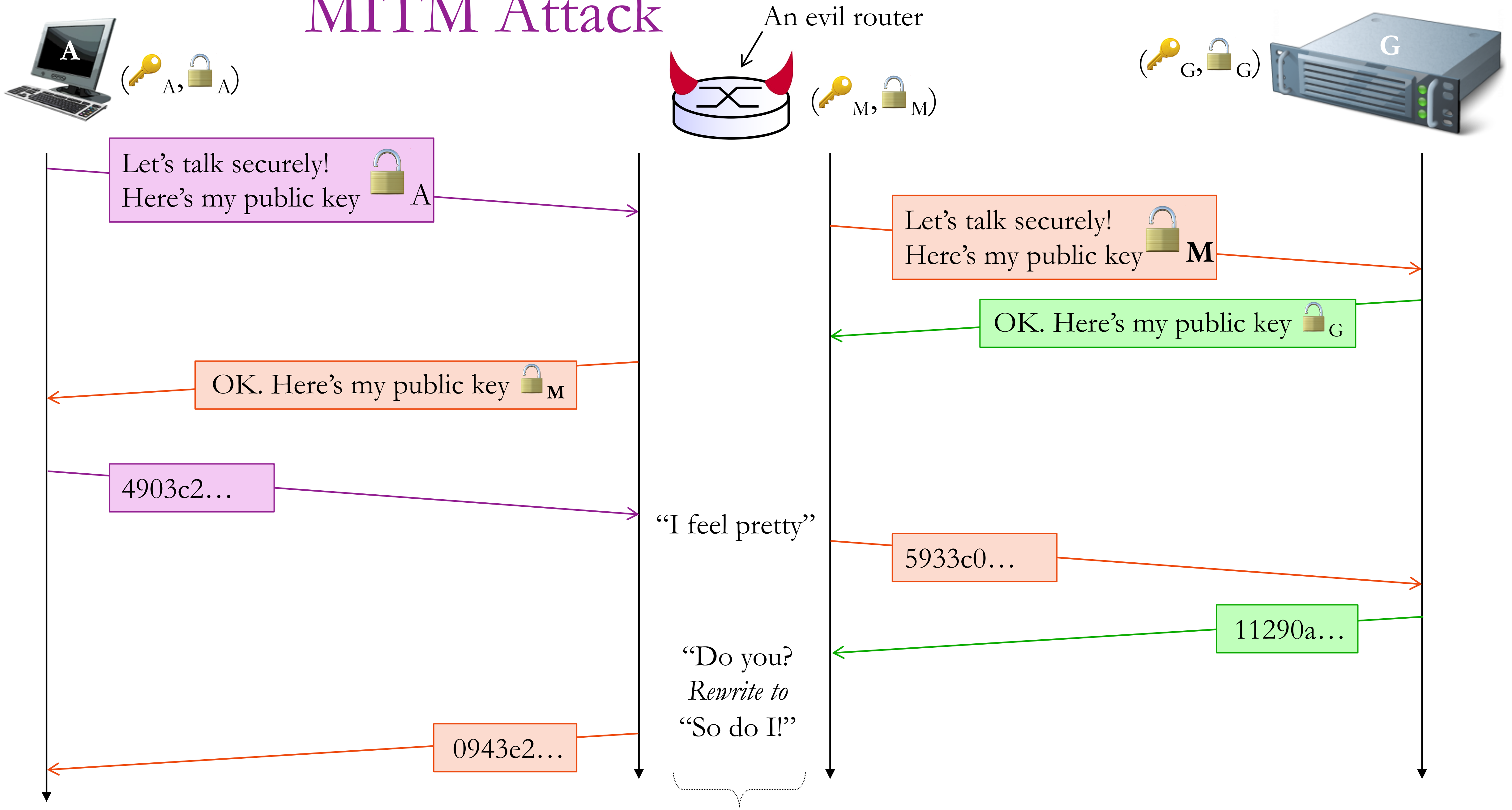
Why does **confidentiality** (keeping messages secret) require **authentication**?

- There are many ways in which Internet messages can be read by untrusted 3rd parties.
- In order to start encryption, we have to verify that we are exchanging public keys with the intended party, not a "man in the middle."

Man-in-the-middle (MITM) attack

- A major flaw remains in this communication scheme.
- We cannot be sure that the machine we contacted is actually who we intended to contact – messages may not be **authentic**.
- A malicious intermediate router may establish two different encrypted sessions between the communicating parties and relay messages.
  - MITM advertises a false public key (its own) to the sender
  - MITM is then able to decrypt sender's message and re-encrypt message with the receiver's true key before delivery.
- Messages can be viewed and altered before delivery.
- MITM can violate *confidentiality*, *integrity*, and *authentication*.

MITM Attack



MITM can **read** and **alter** messages!

How to avoid MITM attacks?

- Simple, but impractical, solution is to avoid public-key cryptography. Instead just use symmetric encryption with pre-shared key.
- To use public key cryptography, we must be sure that the public key we receive really belongs to the endpoint, and not some MITM.
  - If so, we can be assured that only the endpoint has the private key.
- Real-world solutions involve using **digital signatures** to verify public keys (certificates).

Why digital signatures?

- Want to produce a **public document** (not encrypted) that:
  - Makes some claim (or has a message, in general).
  - Anyone can verify the author of the document/message.
    - In other words, it's not possible that someone else *forged* the document.

- The **existence** of the signature on the document proves something.
  - We don't care who gave us a copy.
  - Unlike real-world ink signatures, an attacker cannot copy a signature from one document to another. Signature is somehow unique to the document.

- Notice this is a case when we need **authentication** and **integrity**, but *not* **confidentiality**.

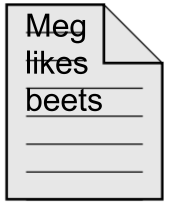
Digital Signatures

- A **digital signature** is a short bit-sequence generated from a digital document and a private key, with the following properties:

  - Like a hash function, it always produces the same result (for given data).
  - It produces different results for different documents and keys.
  - The document cannot be ***signed*** without a **private key**.
  - The signature can be ***verified*** using only the corresponding **public key**.

- Closely related to public key encryption: can use the same RSA keys.
- Changing a signed document will make the former signature invalid.

Signing



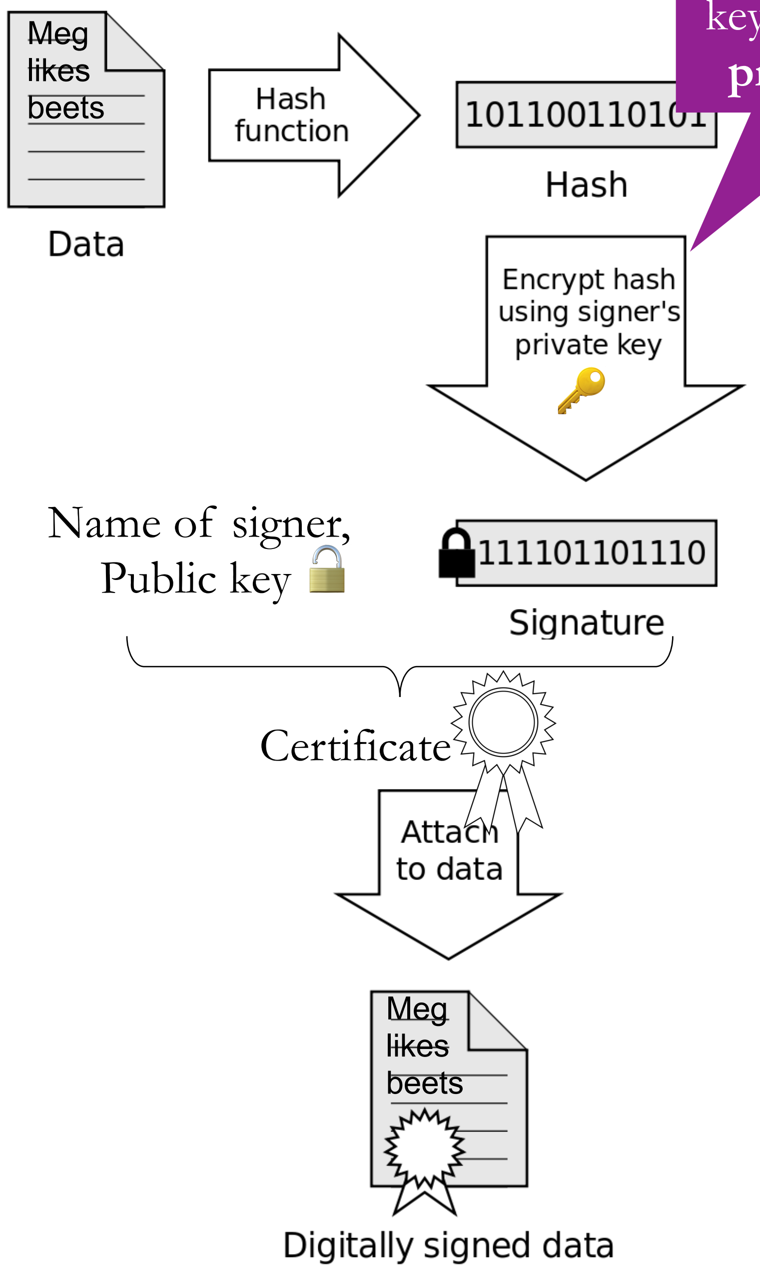
Data

If hashes are equal, the signature is valid

Signing: The signer uses the RSA private to encrypt the message hash, creating a *signature*.

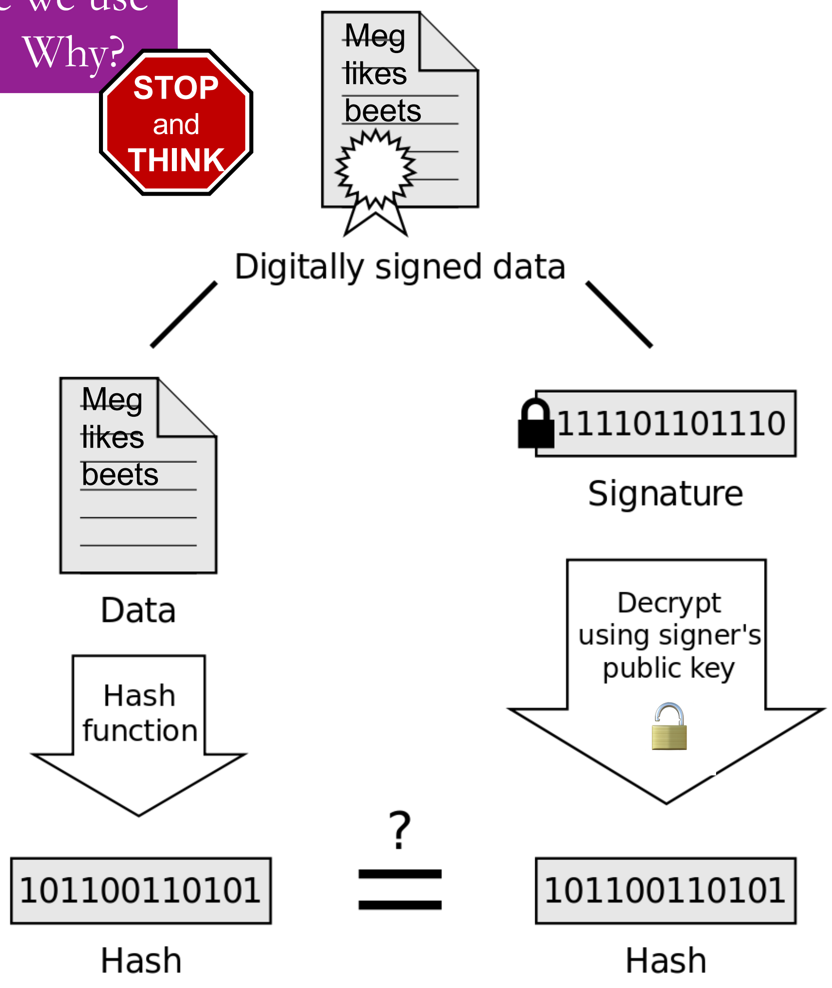
Verification: Anyone looking at the certificate can use the public key to try to decrypt the signature. If this leads to the true message hash, then the signature must have been generated using the true private key.

Signing



For confidentiality, we encrypt with **public** keys, but here we use **private** key. Why?

Verification



If hashes are equal, the signature is valid

Signing: The signer uses the RSA private to encrypt the message hash, creating a *signature*.

Verification: Anyone looking at the certificate can use the public key to try to decrypt the signature. If this leads to the true message hash, then the signature must have been generated using the true private key.

The meaning of signatures

- If I download a document with a statement ("Meg likes beets") **and** that document contains a valid digital signature for public key $_{\mathbf{M}}$, I know that:
  - Someone with access to the private key $_{\mathbf{M}}$ saw the document and chose to run the signing algorithm to compute the corresponding signature.
  - We know that only someone holding the private key would have any reasonable chance of computing the correct signature for that ⟨document, public-key⟩ pair.
- Therefore, if I already know that Meg's public key is $_{\mathbf{M}}$ then I can trust that she is the author of the statement "Meg likes beets".
- Meg cannot claim that she does not like beets (**non-repudiation**).

- But, how do I know $_{\mathbf{M}}$ is really her public key? *transitive trust*…

But what is the "signature" itself?

- Signing is generating and publishing a **number** (signature).
- The number is associated with a message (document), and a public key.
- The number is computationally infeasible to calculate without knowing the private key.
- Hence, the fact that the number is known by anyone implies that the holder of the private key chose to generate it.
- Anyone can verify that the number is correct using the document and the public key.

Digital signatures can provide ***transitive trust***

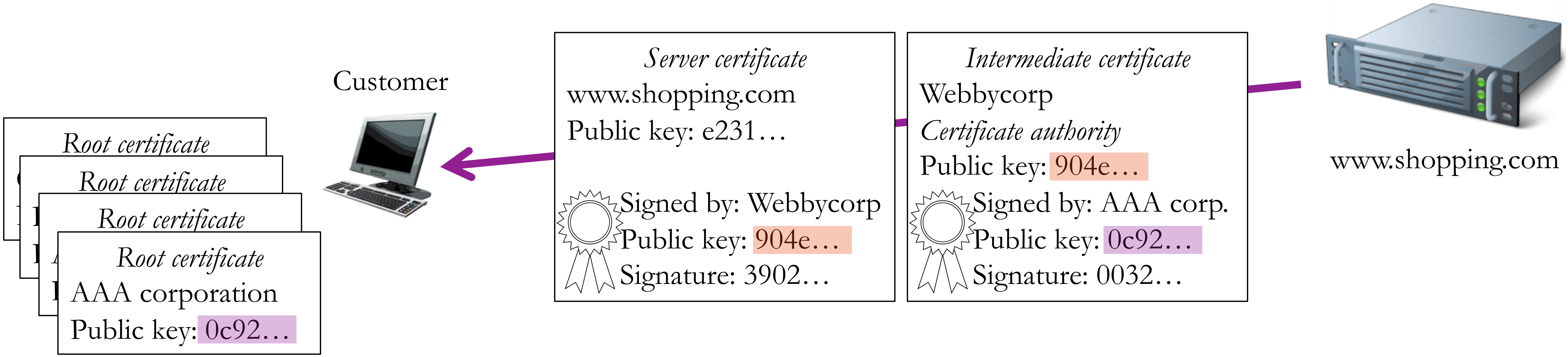
- Let's say I want to browse the web and securely visit thousands of websites over the lifetime of the computer.
- It's impractical to get a pre-shared key ahead of time from these thousands of websites, yet I still wish to communicate securely.
- However, I can easily get just a **few trusted public keys:**
    - These are called **root authorities**.
- Root authority can sign a certificate identifying another trustworthy **certificate authority**.
- Website operator must get their HTTPS public keys (certificates) signed by a certificate authority
    - Certificate authority does some kind of check that the person asking for the certificate really represents the domain & organization listed on the cert.
    - Website then advertises that certificate, proving that their public key is legit.

Demo of SSL chain of trust

https://northwestern.edu

Public Key Infrastructure (PKI)

- PKI creates, distributes, and verifies stranger's claims (certificates).
- A **distributed** and **scalable** way to verify public keys.
- At first, client does not trust server, however server provides two certificates which are sufficient to earn the customer's trust:
- Chain of trust links the advertised public key to an already-trusted key.



Getting a certificate

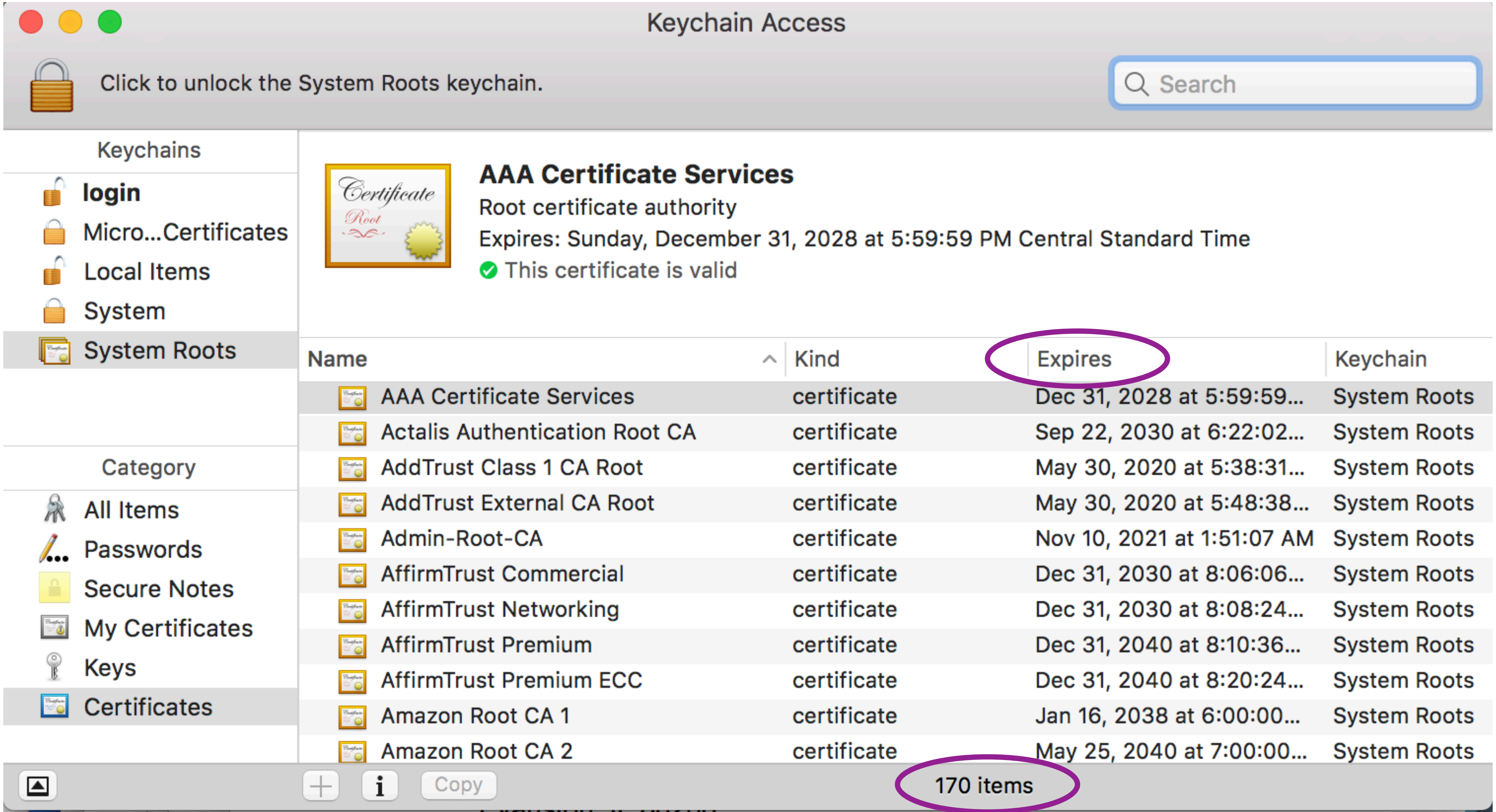
- Pay a fee to a certificate authority and send them a **certificate signing request** (CSR):

  "*Common name:* northwestern.edu; *Public key:* 3a203c…"

How to verify requester's identity?

- Certificate authority (CA) somehow verifies the claim in the certificate:
  - Email the registrant listed in the WHOIS database.
  - Look up the phone number of the requester and call them.
  - Send a letter to the requester and wait for a reply.
  - Visit the requester's office and verify the public key in-person.
  - Challenge the requester to post a random number/document on their webpage or in a DNS record.
- If the CA is satisfied, it will compute a digital signature **certifying** the claims in the CSR, and send you the certificate (CSR + signature).

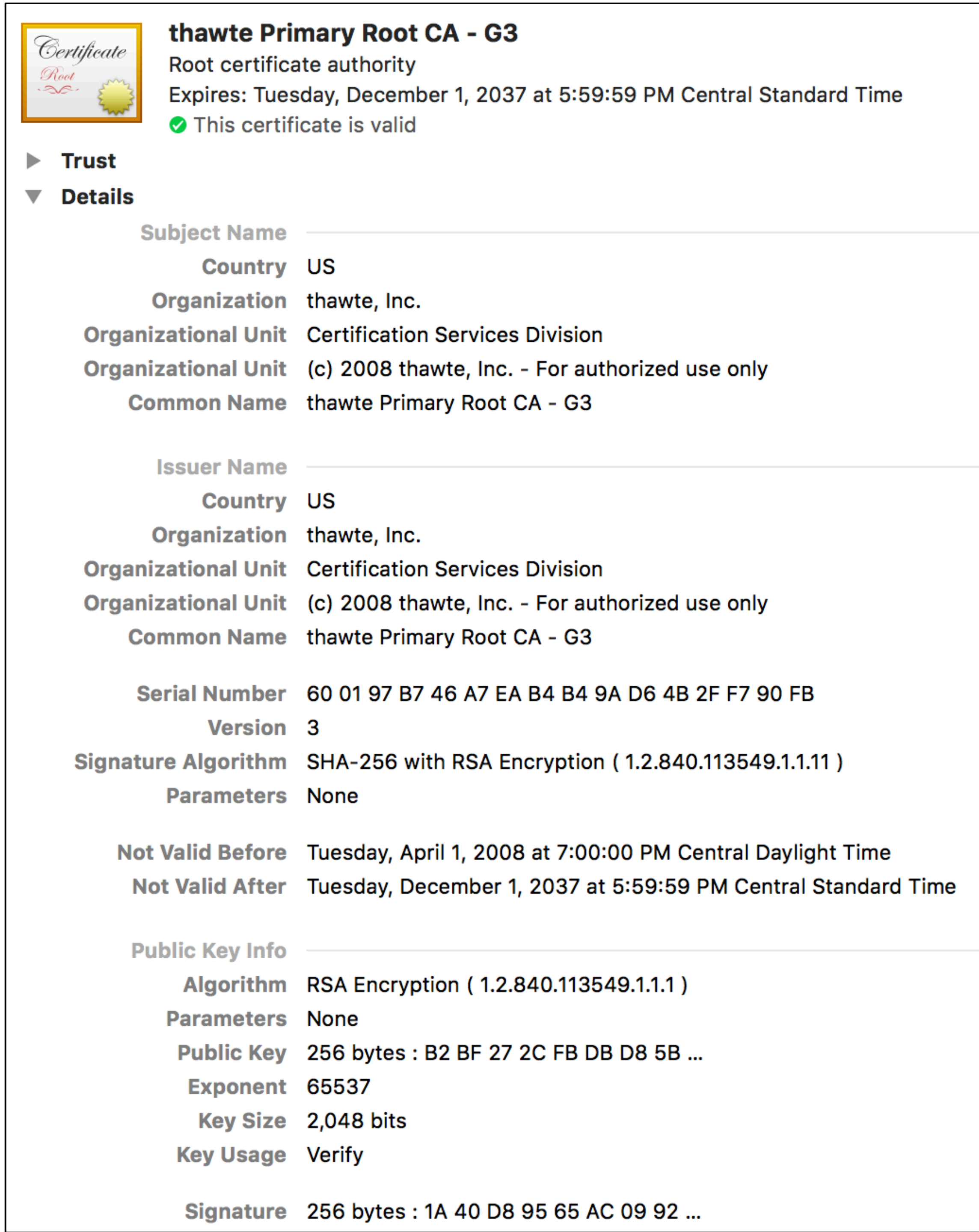
Mac OS comes with 170 root certificates



Root certificate components

- Expiration date
- Subject's name
- Issuer name
- Issuer's signature algorithm
- Public key of subject
- Issuer's signature

Root certificates are *self-issued* and *self-signed*. The user must have some outside reason to trust it.



Intermediate certificate

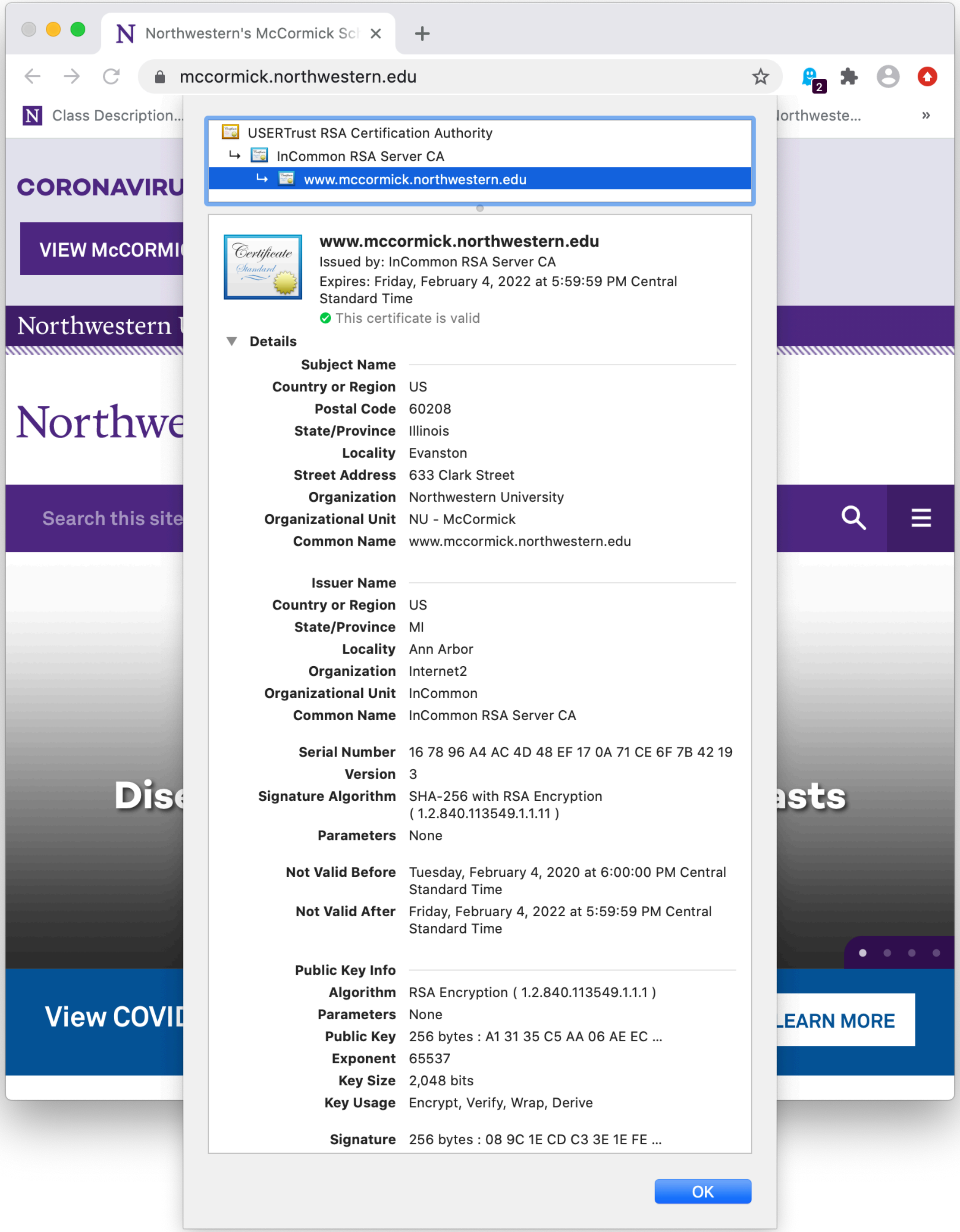
- Has same basic components as root certificate.
- Must be signed by a trusted *issuer*.
- This certificate was issued by a root authority at the same company:

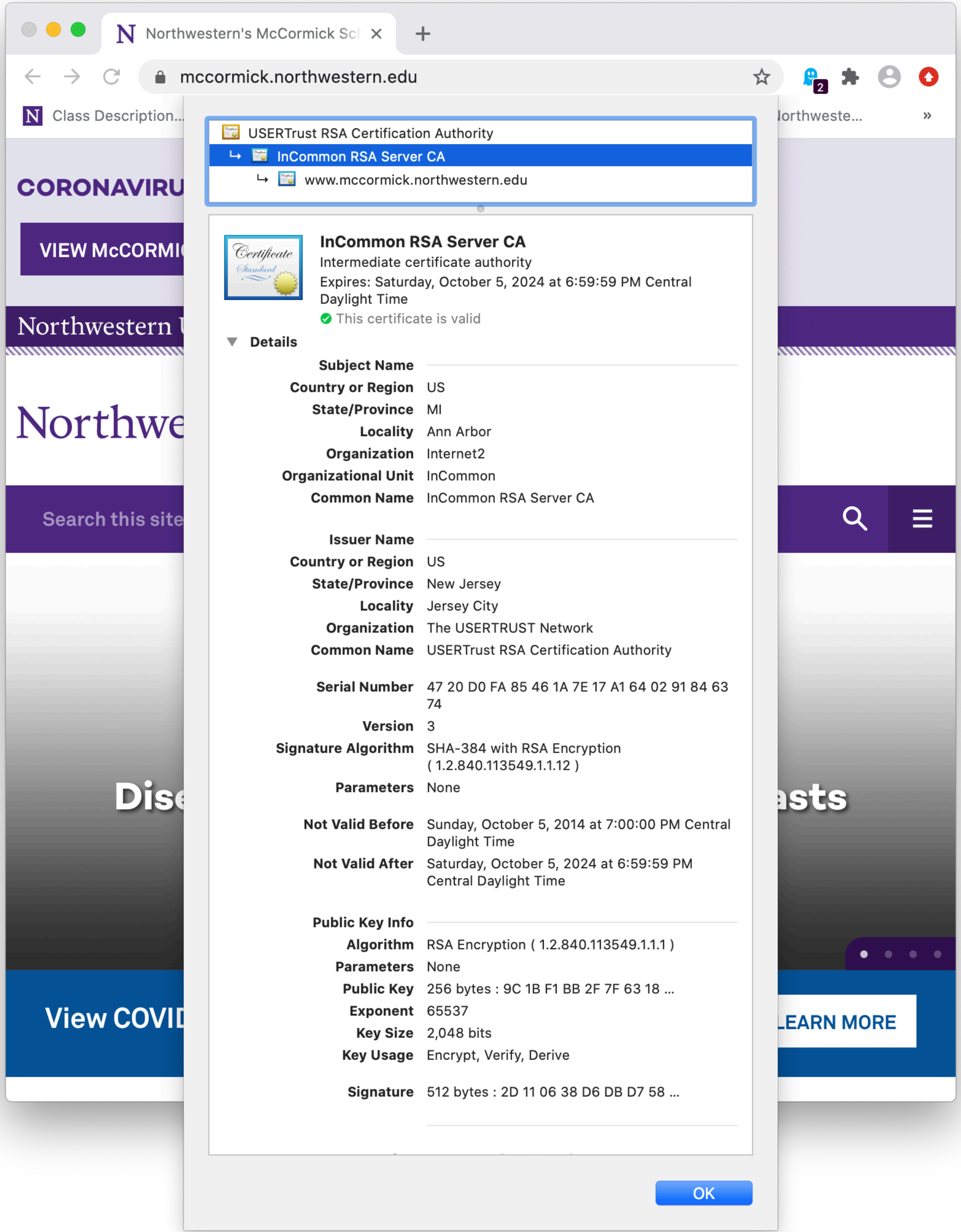
  "thawte SHA256 SSL CA"
  is signed by
  "thawte Primary Root CA - G3"

thawte SHA256 SSL CA
Intermediate certificate authority
Expires: Monday, May 22, 2023 at 6:59:59 PM Central Daylight Time
✓ This certificate is valid

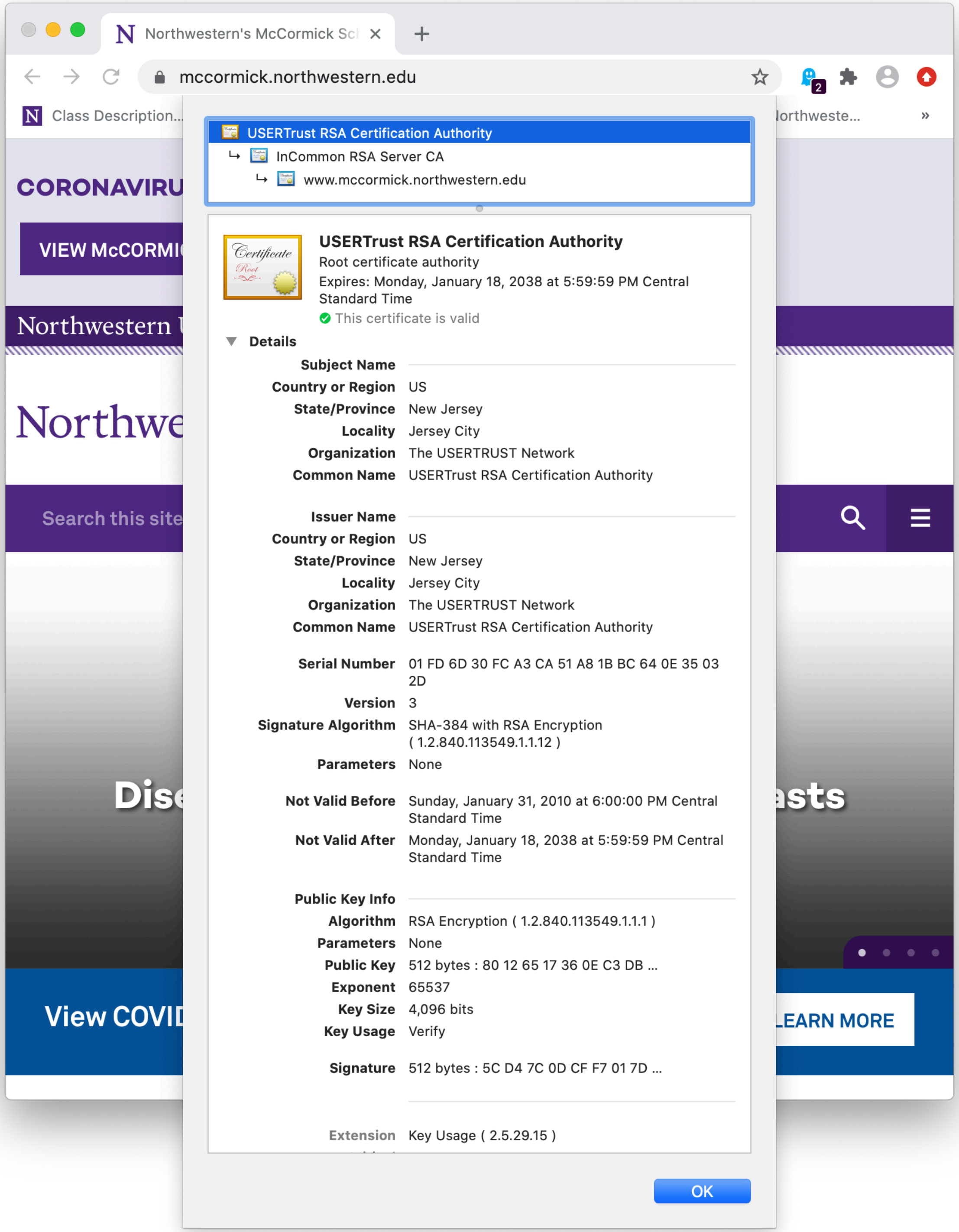
▶ **Trust**
▼ **Details**

Subject Name	
Country	US
Organization	thawte, Inc.
Common Name	thawte SHA256 SSL CA
Issuer Name	
Country	US
Organization	thawte, Inc.
Organizational Unit	Certification Services Division
Organizational Unit	(c) 2008 thawte, Inc. - For authorized use only
Common Name	thawte Primary Root CA - G3
Serial Number	36 34 9E 18 C9 9C 26 69 B6 56 2E 6C E5 AD 71 32
Version	3
Signature Algorithm	SHA-256 with RSA Encryption (1.2.840.113549.1.1.11)
Parameters	None
Not Valid Before	Wednesday, May 22, 2013 at 7:00:00 PM Central Daylight Time
Not Valid After	Monday, May 22, 2023 at 6:59:59 PM Central Daylight Time
Public Key Info	
Algorithm	RSA Encryption (1.2.840.113549.1.1.1)
Parameters	None
Public Key	256 bytes : A3 63 2B D4 BA 5D 38 AE ...
Exponent	65537
Key Size	2,048 bits
Key Usage	Verify
Signature	256 bytes : 74 A6 56 E8 AF 93 96 19 ...


Northwestern's McCormick Sc
mccormick.northwestern.edu
Class Description...
lorthweste...
CORONAVIRU
VIEW McCORMI
Northwestern
Northwe
Search this site
Dise
asts
View COVI
LEARN MORE
USERTrust RSA Certification Authority
InCommon RSA Server CA
www.mccormick.northwestern.edu
Certificate
Standard
www.mccormick.northwestern.edu
Issued by: InCommon RSA Server CA
Expires: Friday, February 4, 2022 at 5:59:59 PM Central Standard Time
This certificate is valid
Details
Subject Name
Country or Region US
Postal Code 60208
State/Province Illinois
Locality Evanston
Street Address 633 Clark Street
Organization Northwestern University
Organizational Unit NU - McCormick
Common Name www.mccormick.northwestern.edu
Issuer Name
Country or Region US
State/Province MI
Locality Ann Arbor
Organization Internet2
Organizational Unit InCommon
Common Name InCommon RSA Server CA
Serial Number 16 78 96 A4 AC 4D 48 EF 17 0A 71 CE 6F 7B 42 19
Version 3
Signature Algorithm SHA-256 with RSA Encryption (1.2.840.113549.1.1.11)
Parameters None
Not Valid Before Tuesday, February 4, 2020 at 6:00:00 PM Central Standard Time
Not Valid After Friday, February 4, 2022 at 5:59:59 PM Central Standard Time
Public Key Info
Algorithm RSA Encryption (1.2.840.113549.1.1.1)
Parameters None
Public Key 256 bytes : A1 31 35 C5 AA 06 AE EC ...
Exponent 65537
Key Size 2,048 bits
Key Usage Encrypt, Verify, Wrap, Derive
Signature 256 bytes : 08 9C 1E CD C3 3E 1E FE ...
OK



Northwestern's McCormick Sc
mccormick.northwestern.edu
Class Description...
CORONAVIRU
VIEW McCORMI
Northwestern
Search this site
Dis
asts
View COVI
LEARN MORE
USERTrust RSA Certification Authority
InCommon RSA Server CA
www.mccormick.northwestern.edu
InCommon RSA Server CA
Intermediate certificate authority
Expires: Saturday, October 5, 2024 at 6:59:59 PM Central Daylight Time
This certificate is valid
Details
Subject Name
Country or Region US
State/Province MI
Locality Ann Arbor
Organization Internet2
Organizational Unit InCommon
Common Name InCommon RSA Server CA
Issuer Name
Country or Region US
State/Province New Jersey
Locality Jersey City
Organization The USERTRUST Network
Common Name USERTrust RSA Certification Authority
Serial Number 47 20 D0 FA 85 46 1A 7E 17 A1 64 02 91 84 63 74
Version 3
Signature Algorithm SHA-384 with RSA Encryption (1.2.840.113549.1.1.12)
Parameters None
Not Valid Before Sunday, October 5, 2014 at 7:00:00 PM Central Daylight Time
Not Valid After Saturday, October 5, 2024 at 6:59:59 PM Central Daylight Time
Public Key Info
Algorithm RSA Encryption (1.2.840.113549.1.1.1)
Parameters None
Public Key 256 bytes : 9C 1B F1 BB 2F 7F 63 18 ...
Exponent 65537
Key Size 2,048 bits
Key Usage Encrypt, Verify, Derive
Signature 512 bytes : 2D 11 06 38 D6 DB D7 58 ...
OK


Northwestern's McCormick Sc
mccormick.northwestern.edu
Class Description...
lorthweste...
CORONAVIRU
VIEW McCORMI
Northwestern U
Northwe
Search this site
Dise
asts
View COVID
LEARN MORE
USERTrust RSA Certification Authority
InCommon RSA Server CA
www.mccormick.northwestern.edu
USERTrust RSA Certification Authority
Root certificate authority
Expires: Monday, January 18, 2038 at 5:59:59 PM Central Standard Time
This certificate is valid
Details
Subject Name
Country or Region US
State/Province New Jersey
Locality Jersey City
Organization The USERTRUST Network
Common Name USERTrust RSA Certification Authority
Issuer Name
Country or Region US
State/Province New Jersey
Locality Jersey City
Organization The USERTRUST Network
Common Name USERTrust RSA Certification Authority
Serial Number 01 FD 6D 30 FC A3 CA 51 A8 1B BC 64 0E 35 03 2D
Version 3
Signature Algorithm SHA-384 with RSA Encryption (1.2.840.113549.1.1.12)
Parameters None
Not Valid Before Sunday, January 31, 2010 at 6:00:00 PM Central Standard Time
Not Valid After Monday, January 18, 2038 at 5:59:59 PM Central Standard Time
Public Key Info
Algorithm RSA Encryption (1.2.840.113549.1.1.1)
Parameters None
Public Key 512 bytes : 80 12 65 17 36 0E C3 DB ...
Exponent 65537
Key Size 4,096 bits
Key Usage Verify
Signature 512 bytes : 5C D4 7C 0D CF F7 01 7D ...
Extension Key Usage (2.5.29.15)
OK

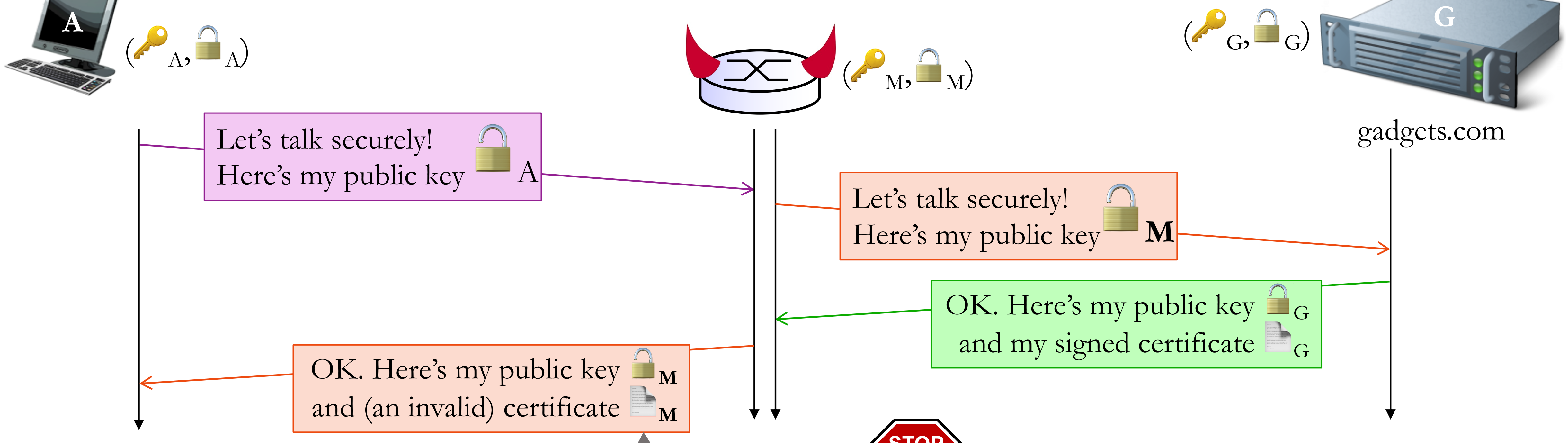
User can add and remove root certificates

- This controls which organizations (and public keys) are trusted to vouch for others.
- What would happen if you removed lots of these root certificates?

  - Many HTTPS websites (and other SSL connections) would stop working.
- What would happened if you added a “bad” root certificate?

  - Your web browser would trust public keys that may be invalid, making you vulnerable to a man-in-the-middle or other impersonation attack.

MITM Attack Revisited



Client notices that *certificate is fake or missing*. A real certificate authority would not have signed a certificate listing M as the public key for gadgets.com.
Client drops the connection.

STOP and THINK

Why?

M must have one of these problems:

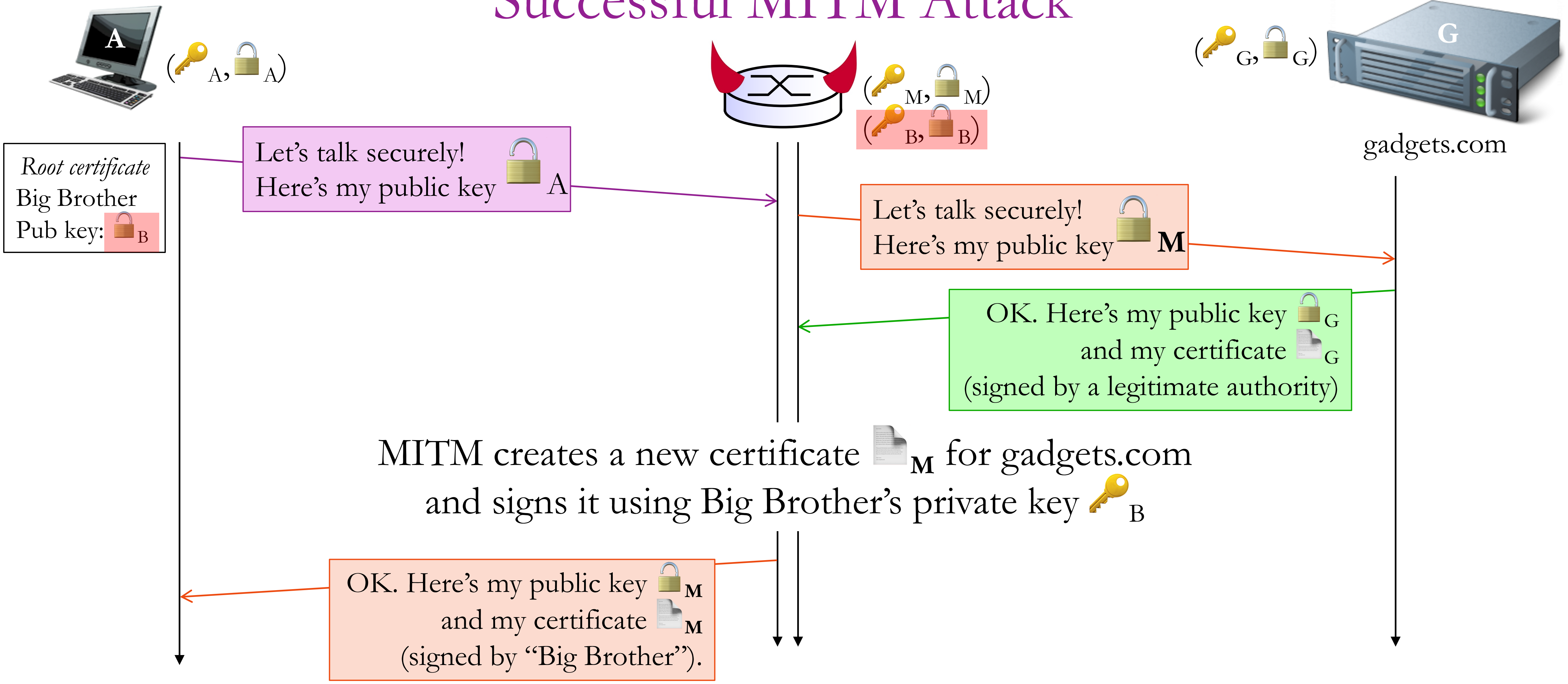
- Domain name is not "gadgets.com"
- Signature is invalid.
- Issuer is not trusted.

Server *does not* notice the attack because it doesn't expect a signed certificate from client. Instead, *passwords* are usually used to authenticate clients.

Sketchy root certificates allow MITM attacks

- Some corporate and campus networks require machines to install a new root certificate to connect to networked services.
- By installing a single malicious root certificate, all of client's encrypted network traffic can be read and modified.
- This technique is also used by some legitimate debugging tools (eg. Charles Proxy) to sniff HTTPS traffic.
  - Normally, Wireshark cannot view HTTPS traffic because it's encrypted at the application layer.
  - Charles Proxy is a MITM running on your machine that shows decrypted HTTPS streams.

Successful MITM Attack



Client accepts the connection because Big Brother has been installed as a trusted root authority.

PKI failed because the client installed a root certificate from a malicious party who is willing to sign fake certificates.

Certificate Revocation Lists (CRLs)

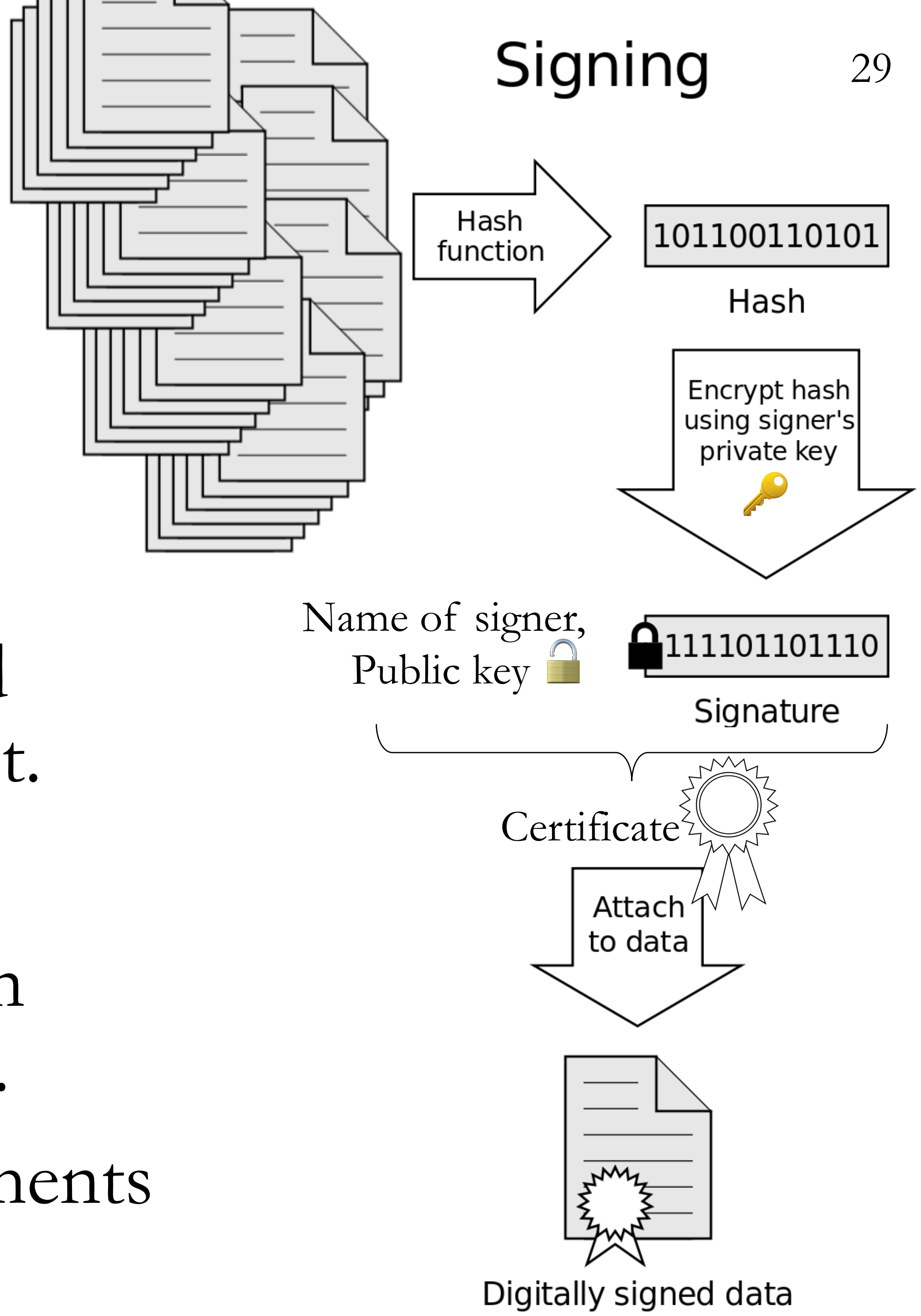
- Private keys are supposed to be kept private, but mistakes happen.
- What happens if someone steals the private key on the www.mccormick.northwestern.edu webserver?
  - A trusted certificate authority has already issued a certificate saying the corresponding public key is valid for that domain until February 4, 2022.
  - Using the private key and a copy of the certificate, the attacker can run a webserver impersonating www.mccormick.northwestern.edu.
- Certificate authorities maintain Certificate Revocation Lists (CRLs) listing **revoked** (but unexpired) certificates. CRL web address is listed in CA cert.
- Client **may** consult CRL before trusting a certificate, but this is slow.
- PKI's scalability (through transitive trust) is lost if you always double-check with a central authority, so CRLs are usually not checked.
  - In practice, losing a private key can have serious security implications.

2011 Comodo Hack

- Comodo is a root certificate authority, but in 2011 its certificate-signing server was hacked.
- Attacker got a username/password for a system that Comodo had built to allow their trusted affiliates to request digital signatures.
- Allowed attacker to generate new certificates for popular services like Gmail, Yahoo Mail, and Hotmail.
- https://www.csoonline.com/article/2623707/hacking/the-real-security-issue-behind-the-comodo-hack.html
- Bogus certificates were revoked and some browsers considered dropping the Comodo root certificate.
    - This would have required all their past customers to buy new certificates from another vendor!

Hashing before signing

- Notice that the digital signature algorithm encrypts a **hash** of the document's data.
- RSA can only encrypt integers $m < n$.
  - Plaintext must be less than 1024 or 2048 bits.
- **Hashing** maps a large document to a fixed integer range, small enough to RSA-encrypt.
- But hashing must be done carefully!
- By signing a hash, we are actually signing an *infinite set of documents* that map to that hash.
- We must be confident that the other documents are random, not useful to attackers.



Hashing example

- If the hash is 256 bits, it can take 2^{256} different values.
- HTTPS certificates are about 2000 bits long: 2^{2000} possible certs.
- We expect $2^{2000}/2^{256} \cong 2^{1744}$ such documents to share each hash value.

128-bit hash function

Name: gadgets.com Public key: 930a…	940294a9bc9233eb5829848ecb9e2482
Name: 3kjD|Sj3;… Public key: a329…	940294a9bc9233eb5829848ecb9e2482
Name: \(we(CC1… Public key: 44a0…	940294a9bc9233eb5829848ecb9e2482
…	
Name: shop.com Public key: 201e…	45b798e10349ae8750cd45209809ec94

Hash *collisions*

Cryptographic Hash functions

If H(x) is a *cryptographic hash* function, it should be *computationally infeasible* to:

- Map backwards from hash output to input: find **x** given **H(x)**
- Find two inputs **x** and **y** that map to the same hashed value: **H(x) = H(y)**

  - We know that there is an infinite set of such (x,y) pairs, but the hash function is designed to make them nearly impossible to find.
  - In particular, if we know *x*, we should *not* be able to find *y* in polynomial time such that H(x) = H(y)
  - Like a good symmetric encryption algorithm, a cryptographic hash must have good *confusion* and *diffusion*. It must behave very randomly.
  - If input is called the *message*, the output is sometimes called the *message digest.*

- **SHA-1** and **MD5** are examples of cryptographic hash functions.

Back to digital signatures

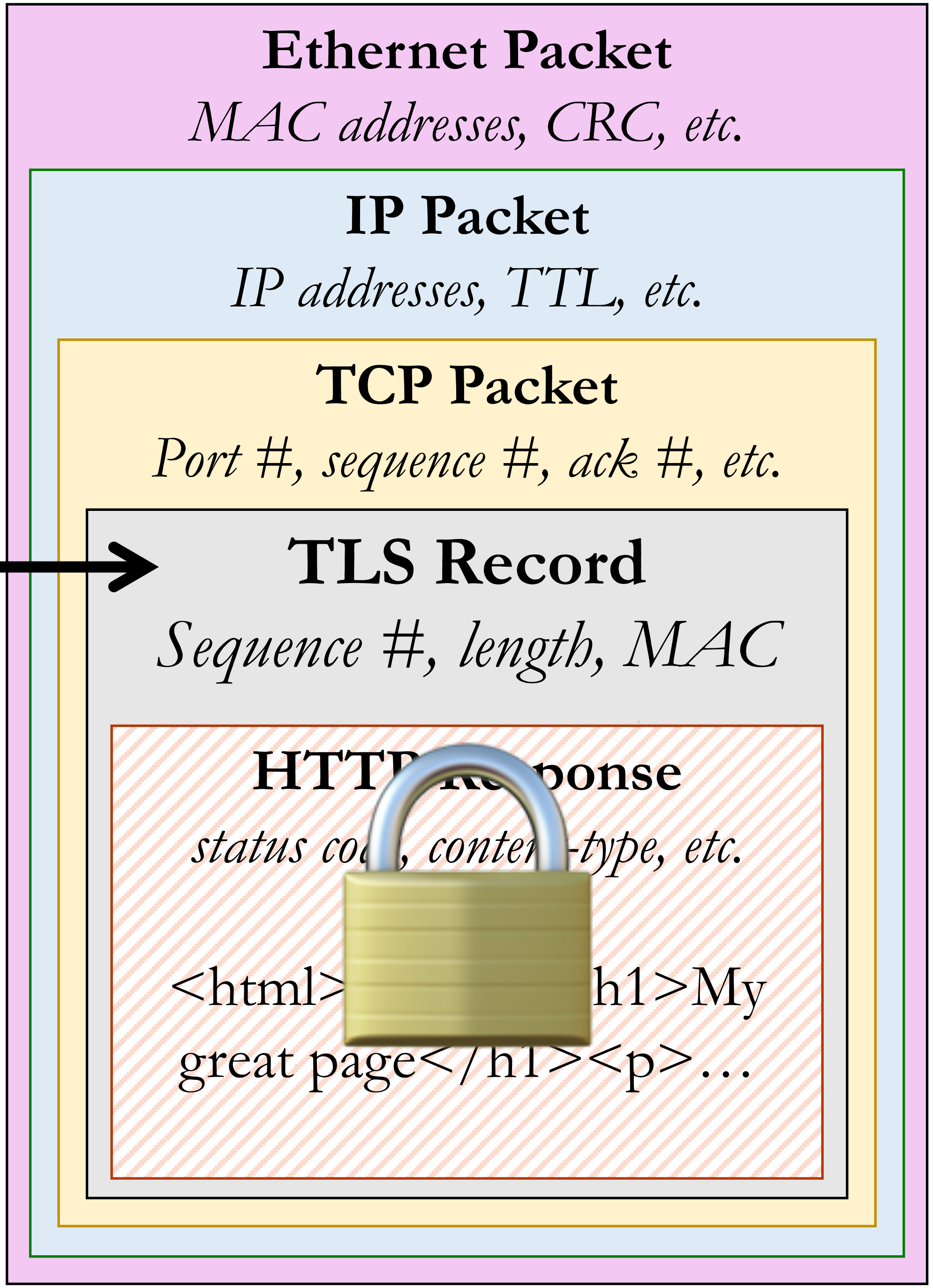
- If I sign a SHA-1 hash of a document and publish that signature, it will be difficult for an attacker to construct a second document with the same SHA-1 hash as the original document that I signed.
- Thus, it's difficult for that signature to be used to falsely verify other documents that I have not seen and signed.
- If I used a dumb hash function, like "the sum of all bytes," forgery would be easy:
  - SUM("fun and cats") == SUM("gun and bats")
  - The change **f+1→g** is cancelled by the change **c-1→b**
  - Using this really bad hash function, the signature of "fun and cats" would also be valid for "gun and bats"

Hash-based Message Authentication Code *(HMAC)*

- Public-key cryptography & digital signatures are computationally expensive.
- HMAC provides a more efficient way to authenticate public messages:
- **HMAC steps:**
  - Assume sender and receiver have a **shared secret**: 🗝️ ← *a new requirement :(*
  - MAC = hash(message + 🗝️)
  - Send ⟨message, MAC⟩
  - Anyone can read the message.
  - Receiver with 🗝️ can also compute the MAC to verify the received MAC.
- Again, we must use a strong cryptographic hash, like SHA-1
- We could have used 🗝️ to encrypt with AES, but this is slower than a SHA-1 hash (and maybe we want 3rd parties to see the message).
- HMAC is often used to authenticate API calls (eg., AWS REST API).

SSL/TLS

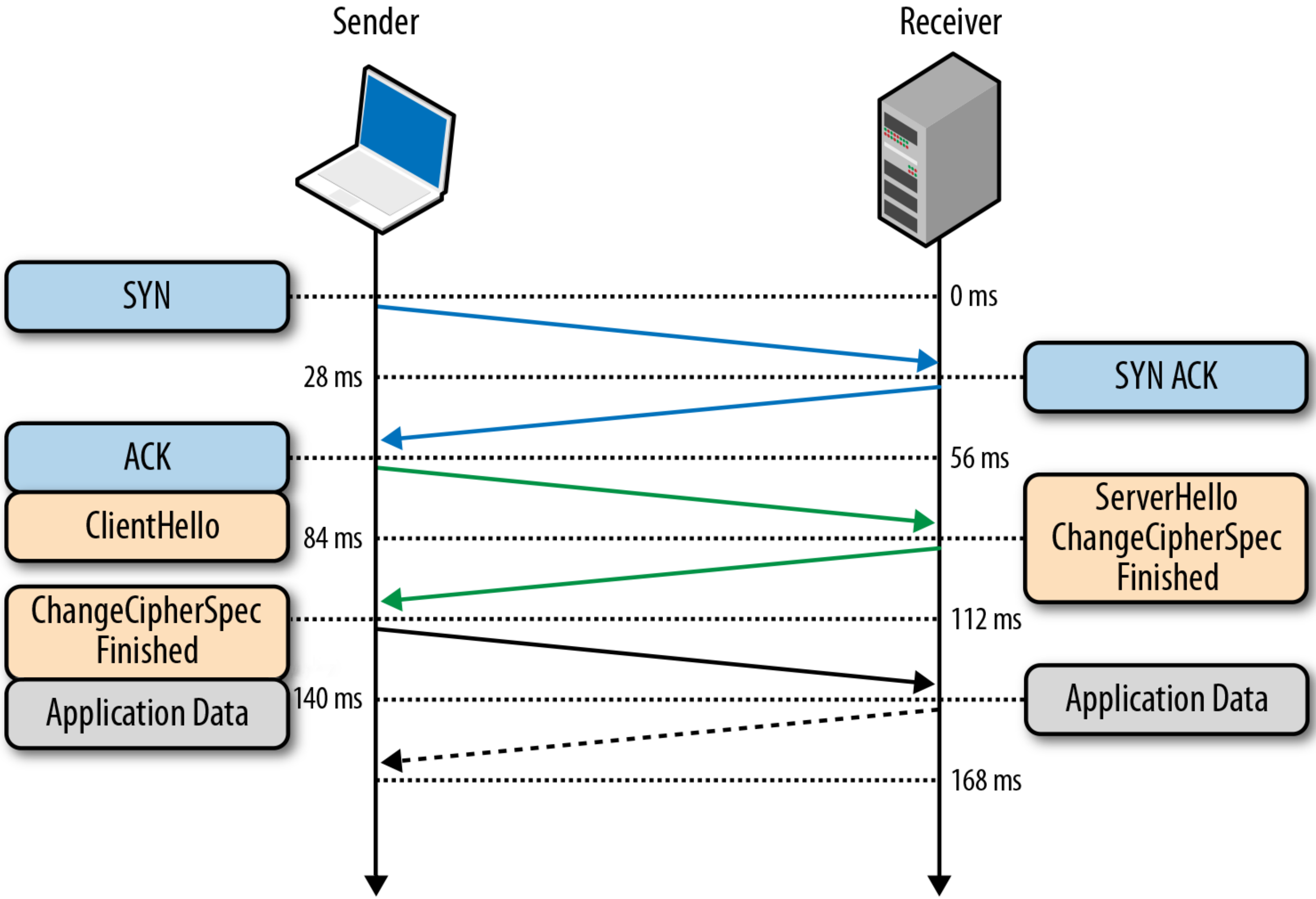
- **Transport Layer Security (TLS)** is the Internet standard for encrypted communication, formerly called *Secure Sockets Layer (SSL)*.
    - A real-world implementation of public-key encryption and auth.
- It's built on top of TCP, sitting below the application layer.
- TLS payload is encrypted. Eg., **this** could be an encrypted segment of an HTML document.
- Defined in [RFC 5246](RFC 5246).



TLS handshake (after TCP handshake)

Sender and receiver must agree on:

- encryption algorithms eg., RSA+AES
- Shared keys. TLS actually uses four different keys (sender, receiver)×(encryption, MAC)
- Other random values:
  - Initialization Vector used by AES
  - Nonce

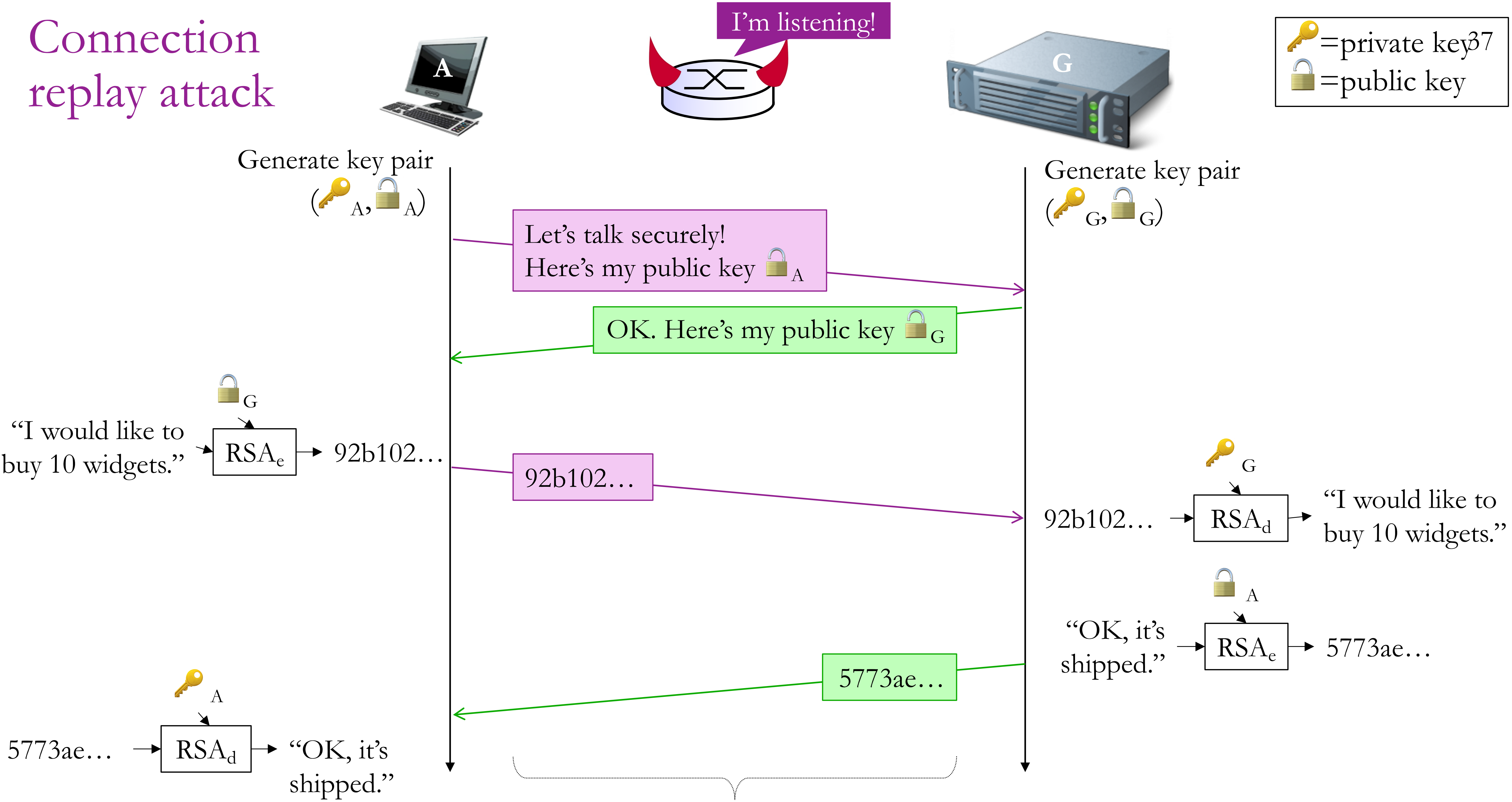


Packet replay attack

- Attacker cannot decrypt packets, but it can intercept and **replay** any packet.
- Receiver might think that it's valid, since it decrypts just fine.
- **Solution**: TLS records include *sequence numbers.*
- Replayed packet would be dropped

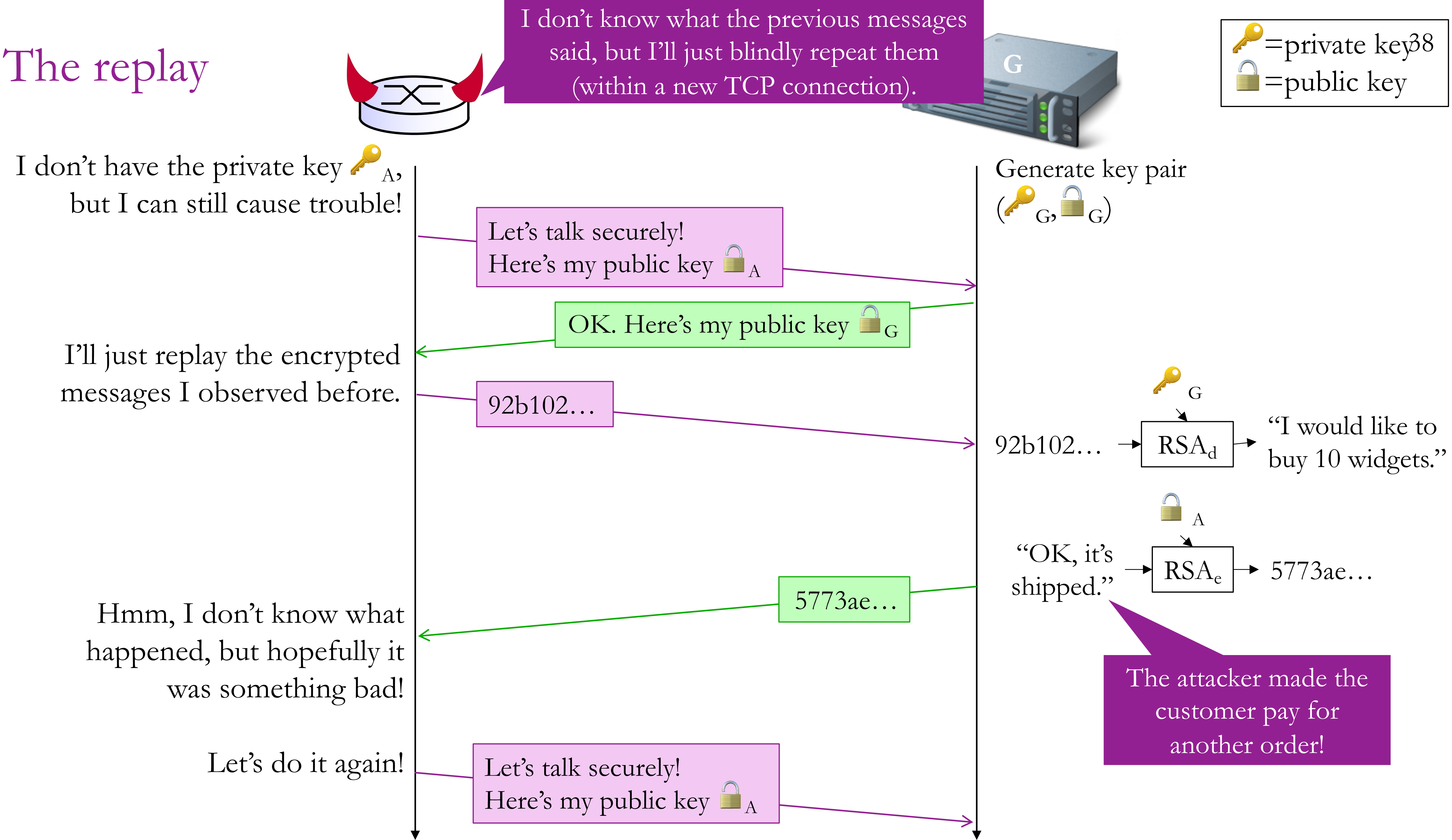
How to protect against packet replay?

Connection replay attack



Network observer only sees public keys and encrypted messages. A & G are never seen.

The replay


I don't know what the previous messages said, but I'll just blindly repeat them (within a new TCP connection).
G
=private key
=public key
I don't have the private key A, but I can still cause trouble!
Generate key pair
(G, G)
Let's talk securely!
Here's my public key A
OK. Here's my public key G
I'll just replay the encrypted messages I observed before.
92b102…
G
92b102…
RSA$_d$
"I would like to buy 10 widgets."
A
"OK, it's shipped."
RSA$_e$
5773ae…
5773ae…
Hmm, I don't know what happened, but hopefully it was something bad!
The attacker made the customer pay for another order!
Let's do it again!
Let's talk securely!
Here's my public key A

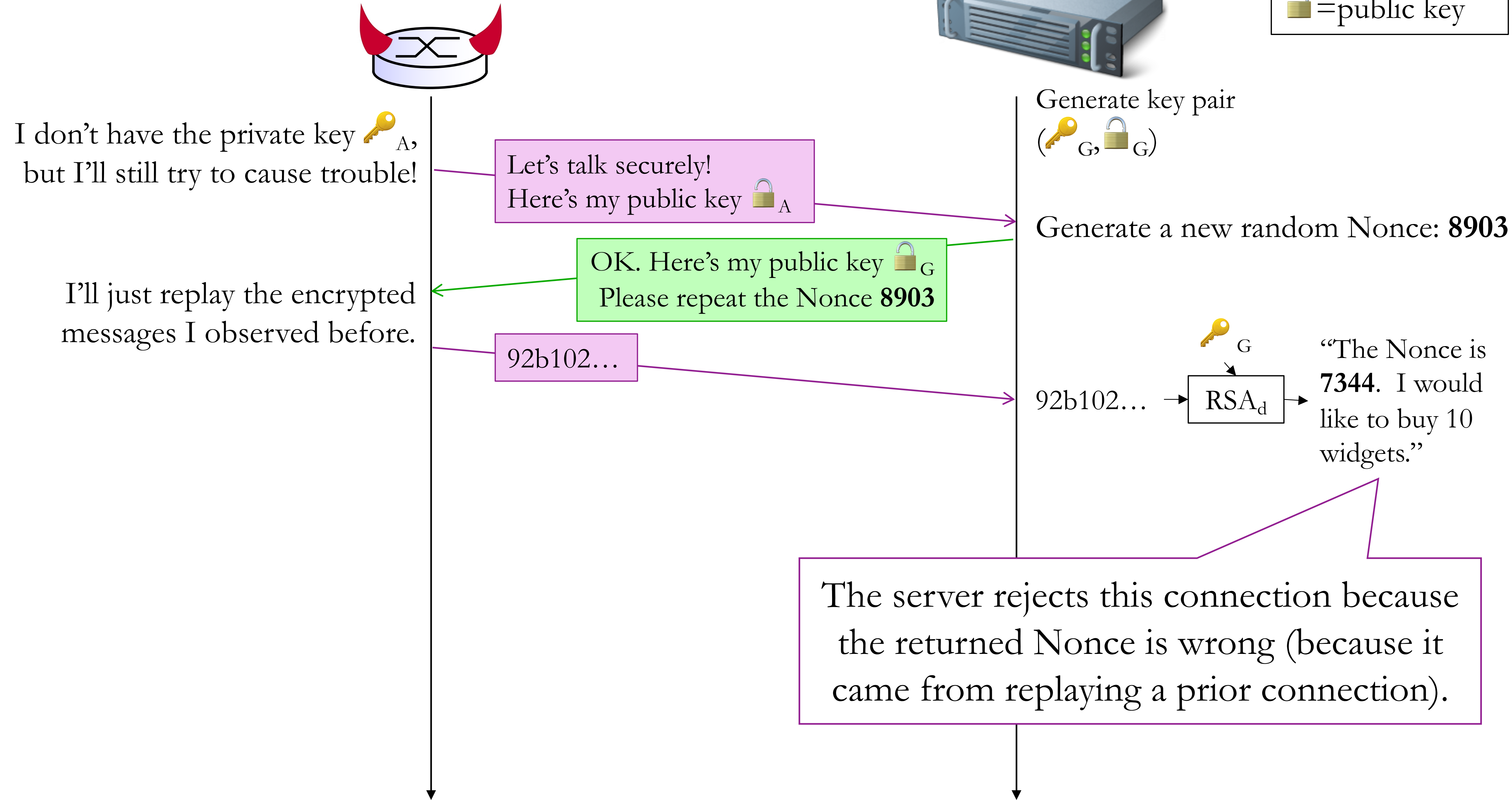
Connection replay attack

How to prevent this?

- Attacker can observe entire client-server interaction and **replay** it.
- **Solution**: receiver sends a random **nonce** in handshake message.
- Sender must include encrypted nonce in the next message.
- This requires each connection's data to be at least slightly different.
- Replay will not work because original connection had a different nonce.

Add a **Nonce** to prevent replay

=private key
=public key



Fundamental network security lessons

- Secure communication involves many considerations.
- Encryption primitives are not enough, they must be **used** carefully.
- TLS must be carefully designed to avoid all kinds of clever attacks, like replay attacks (and many others!)
- Authentication is still not a fully-solved problem, (Public Key Infrastructure has many drawbacks).
    - Learn more in CS-396 Cryptography

Lessons for the software/network engineer:

- Don't try to build your own encryption scheme from scratch.
- Just use the *latest version* of TLS.
- Know the meaning of PKI/certificates, and keep private keys safe!

Recap

- **Digital signatures** are special bit sequences attached to documents that can only be computed by the holder of a private key.
  - Signatures are used to establish **transitive trust** and verify new public keys, thus preventing **Man In The Middle** and other attacks.
  - **Certificate authorities** verify public keys with digitally signed certificates.
  - MITM with root authority's private key can forge arbitrary certificates.
- **Cryptographic hash** functions are irreversible and unpredictable.
  - Used to create a small summary of a document than can be signed with RSA.
  - Also used in **Message Authenticate Codes** (HMAC) to verify that sender has a shared secret: MAC = hash(message + 🗝️)
- **Transport Layer Security (TLS)** encrypts a TCP stream.
  - Details are complex, to allow many different systems to interoperate and to mitigate a variety of attacks: Eg., packet replay, connection replay.