University of Pittsburgh

Applied Cryptography and Network Security
CS 1653

Summer 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Prof. Adam Lee's CS1653 slides.)

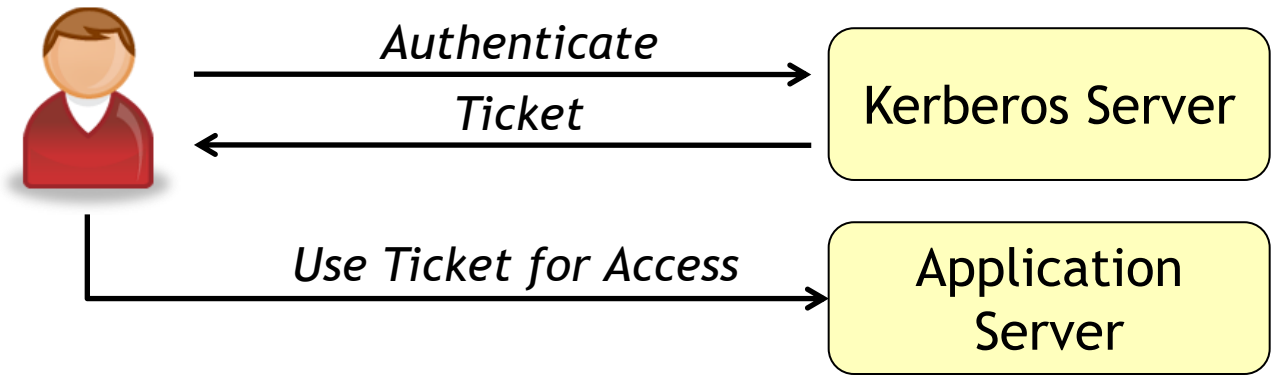
Announcements

- Please schedule a demo of Project Phase 2 this week with Pratik
- Homework 6 due this Friday @ 11:59 pm
- Programming Assignment 1 due this Friday
- Project Phase 3 Due on Monday 7/17 @ 11:59 pm
  - Your team must schedule a meeting with me on or before Thursday 7/13

Kerberos Overview

This is kind of like our term project, yeah?

Kerberos is a mediated authentication protocol



This protocol is based on the following assumptions:

- Server(s) used by Kerberos are highly secured (How?)
- Application servers are moderately secure, though may be compromised
- Client machines are untrusted

Kerberos uses **secret key cryptography** to allow users to authenticate to networked services from any location

Kerberos Design Goals

Main goal: Breaking into one host should not help the attacker compromise the overall security of the system

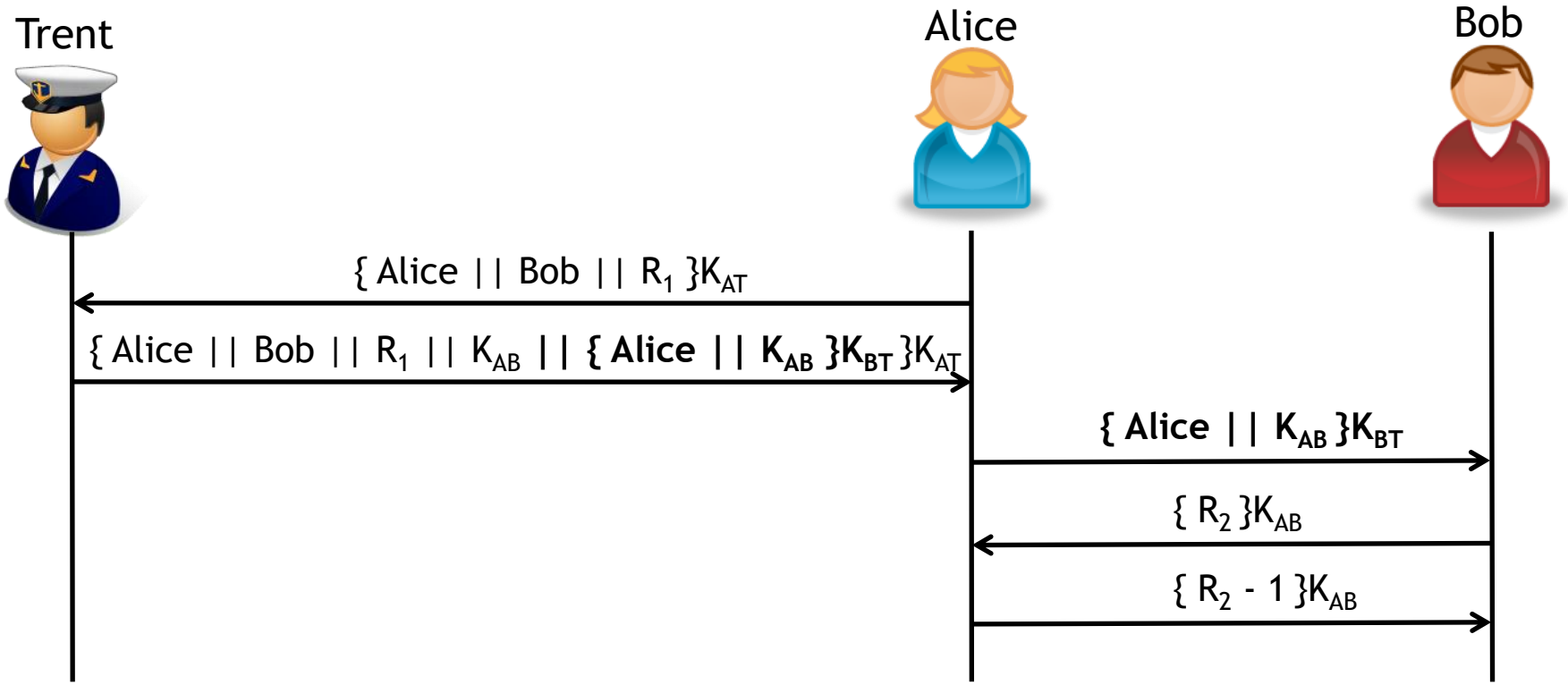
Client authenticator goals:

- Users cannot remember cryptographic keys, so keys should be derived from the user's **password**
- Passwords should not be sent in cleartext **(why?)**
- Passwords should not be stored on the server
- The client's password should be used as little as possible **(why?)**

The use of Kerberos should require only minimal modifications to existing applications

So, how does this work?

Kerberos is based on the Needham-Schroeder secret key authentication protocol



After message 2, Alice

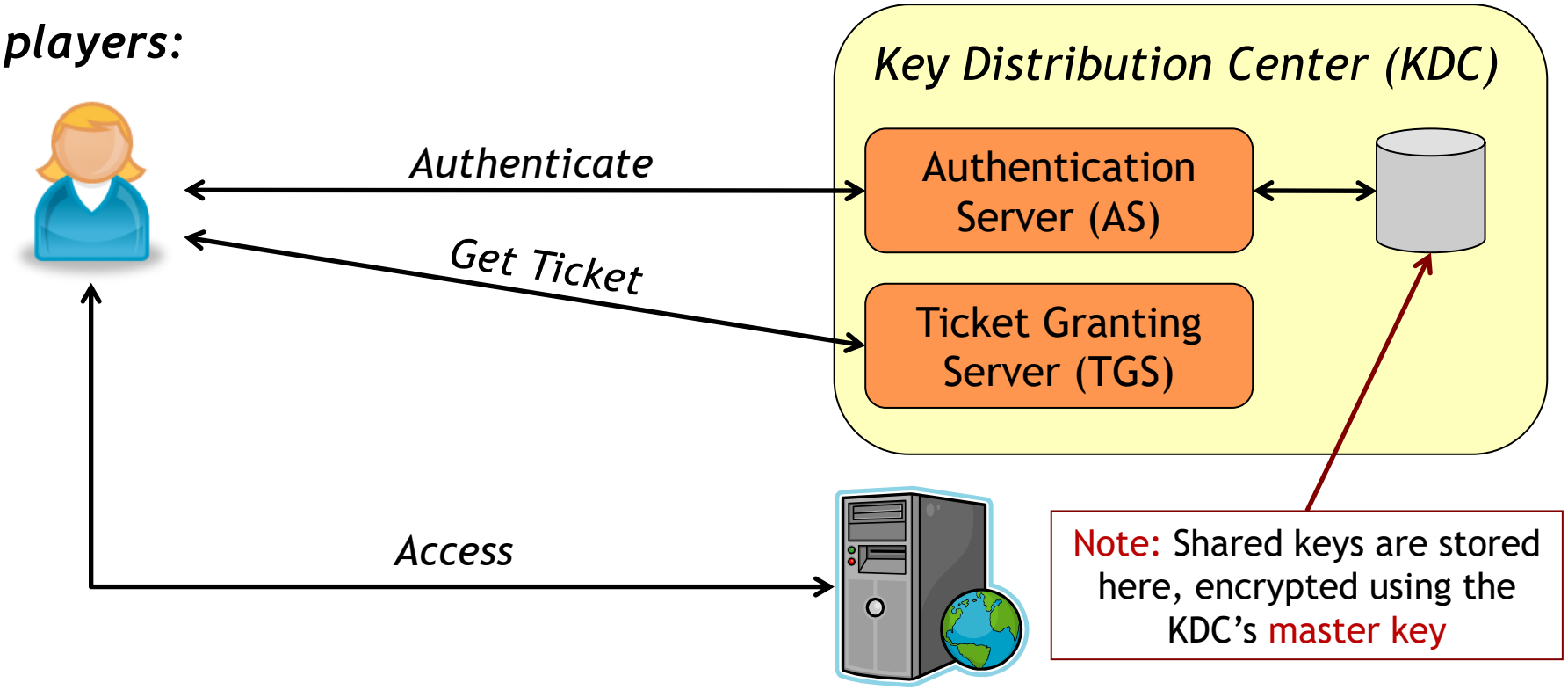
- Knows that this message is fresh **(why?)**
- Knows that the session key is to be shared with Bob **(why?)**

After message 3, Bob knows that he has a shared key with Alice

After message 5, Bob knows that this key is fresh

Kerberos v4: The Basics

The players:

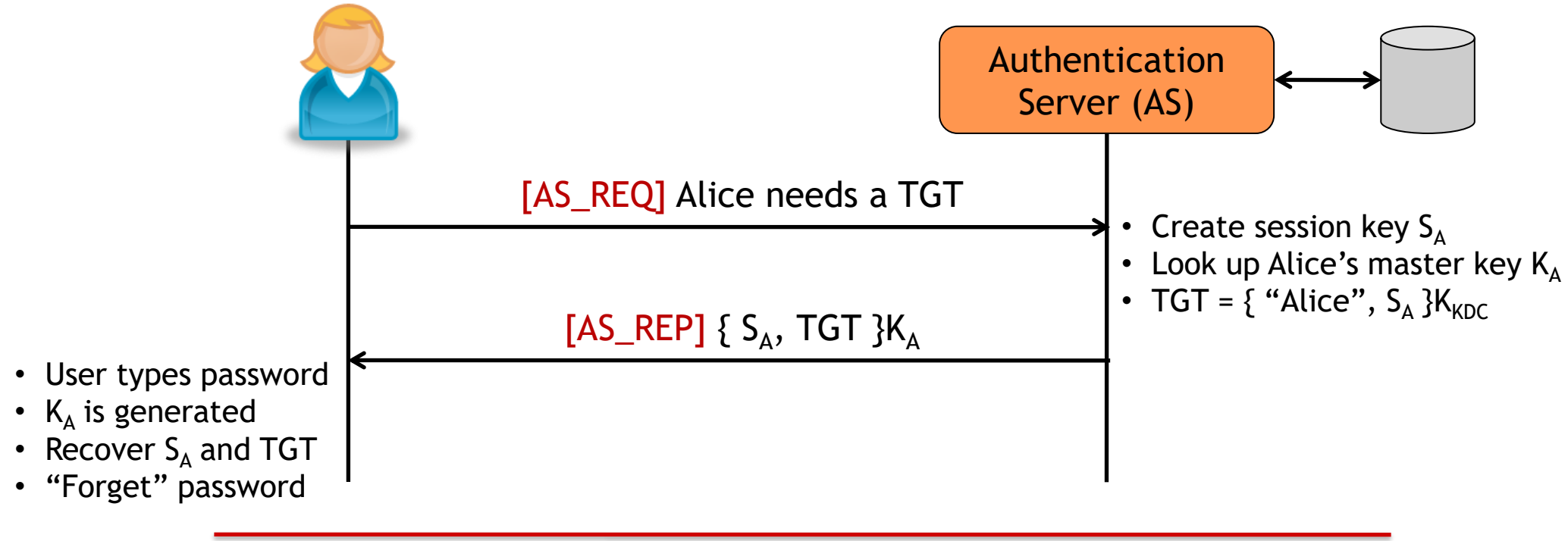


All principals in the system share a secret key with the AS

- User keys derived from their password
- Application server keys are random cryptographic keys

The cryptographic algorithm used by Kerberos is DES (**problem?**)

Step 1: Obtaining a ticket-granting ticket (TGT)



The above process is used to initiate a login session

- Password used to initiate the session
- S_A is used for subsequent exchanges

Note that the user password is not needed until after the TGT is obtained

- Why is this a good thing?
- Why is this a bad thing?

What is the purpose of this process?

In a single session, a user may want to access many different machines

For example…

- Download a file from a secured web server
- SSH into another machine to compare results with experimental data
- Email a colleague to inquire about an oddity in the file
- FTP an updated file to the secured server
- …

By obtaining a single session key S_A, Alice only uses her password once!

- This minimizes the amount of time that the password is exposed
- If S_A is cracked, the password is still safe

Furthermore, the TGT frees the KDC from maintaining any state

- Recall that TGT = {"Alice", S_A }K_{KDC}
- **No need to track S_A at the server**, just ask Alice for her TGT

Obtaining a TGT: Message Detail

AS_REQ

# Bytes	Content
1	Version of Kerberos (`4`)
1	Message Type (`1`)
≤ 40	Alice's name
≤ 40	Alice's **instance**
≤ 40	Alice's realm
4	Alice's timestamp
1	Desired ticket lifetime
≤ 40	Service name (`krbtgt`)
≤ 40	Service **instance**

In multiples of 5 minutes (up to about 21.5 hours)

Helps Alice match request/reply pairs

Note: This message is sent unencrypted

Obtaining a TGT: Message Detail

AS_REP

Bytes	*Content*
1	Version of Kerberos (4)
1	Message type (2)
≤ 40	Alice's name
≤ 40	Alice's instance
≤ 40	Alice's realm
4	Alice's timestamp
1	Number of tickets (1)
4	Ticket expiration time
1	Alice's **key version number**
2	Credentials length
var	Credentials

Bytes	*Content*
8	S_A
≤ 40	TGS name
≤ 40	TGS instance
≤ 40	TGS realm
1	Ticket lifetime
1	TGS key version number
1	Length of ticket
var	Ticket
4	Timestamp
var	**Padding of 0s**

Note: The "Credentials" field is encrypted with Alice's master key K_A

Obtaining a TGT: Message Detail

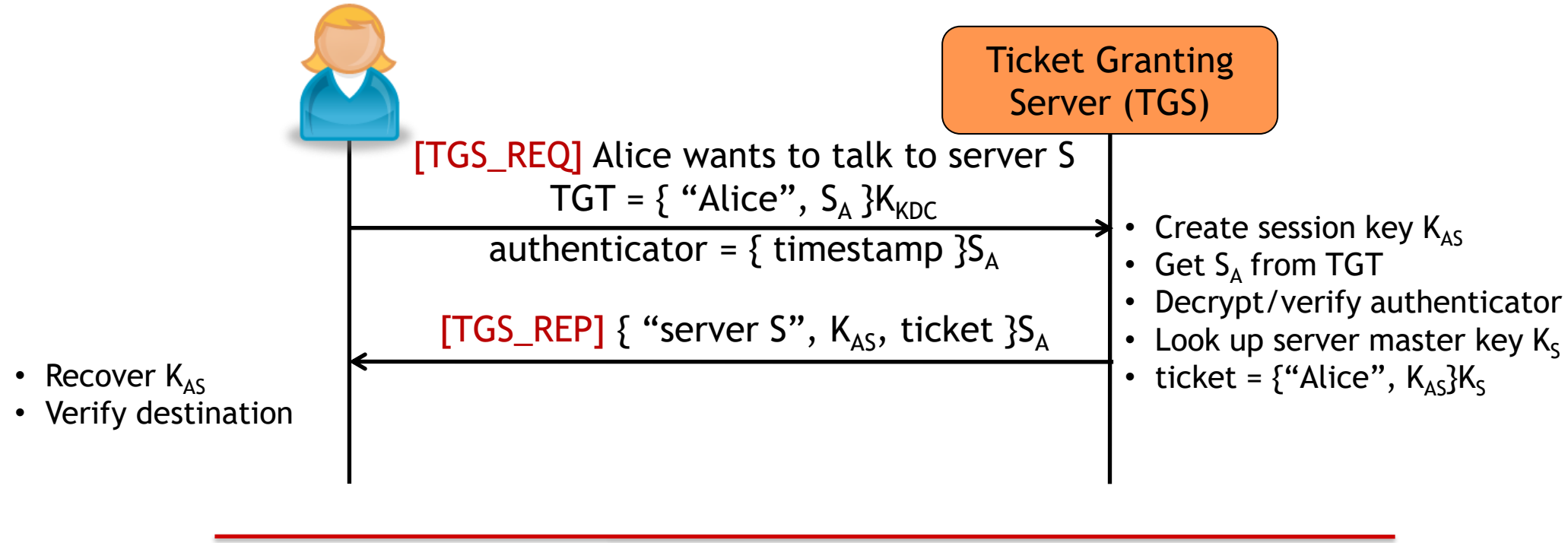
This is the credentials field…

Bytes	*Content*
8	S_A
≤ 40	TGS name
≤ 40	TGS instance
≤ 40	TGS realm
1	Ticket lifetime
1	TGS key version number
1	Length of ticket
var	Ticket
4	Timestamp
var	Padding of 0s

Bytes	*Content*
≤ 40	Alice's name
≤ 40	Alice's instance
≤ 40	Alice's realm
4	Alice's IP address
8	Session key S_A
1	Ticket lifetime
4	KDC timestamp
≤ 40	TGS name
≤ 40	TGS instance
var	Padding of 0s

Note: The "Ticket" field is encrypted with the TGS's master key K_{KDC}

Step 2: Obtaining a service ticket



Interesting notes:

- Alice did not user her password to authenticate!
- The TGS did not need to maintain any state to verify Alice's identity

The authenticator attests to the freshness of the current exchange

- This means that Kerberos requires synchronized clocks (usually ~5 mins)
- Authenticator not actually needed in this exchange (Why?)

Obtaining a Service Ticket: Message Detail

TGS_REQ

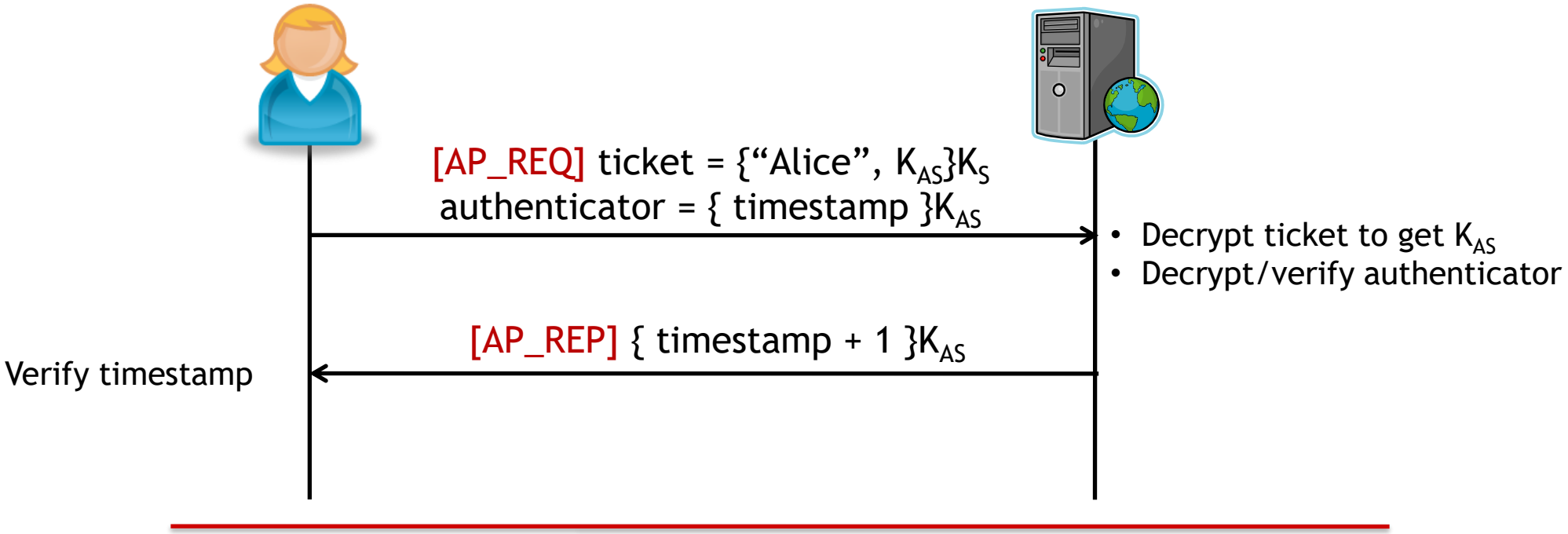
Bytes	*Content*
1	Version of Kerberos (4)
1	Message Type (3)
1	KDC key version number
≤ 40	KDC realm
1	Length of TGT
1	Length of authenticator
var	TGT
var	authenticator
4	Alice's timestamp
1	Desired ticket lifetime
≤ 40	Service name
≤ 40	Service instance

Copied from credentials field of AS_REP

Bytes	*Content*
≤ 40	Alice's name
≤ 40	Alice's instance
≤ 40	Alice's realm
4	checksum
1	5ms timestamp
4	Timestamp
var	Padding

Note: The TGS_REP message is the same format as AS_REP

Step 3: Using a service ticket



The AP_REQ message authenticates Alice to the server

- Only the KDC knows K_S, so the ticket for Alice is authentic
- If timestamp is recent, then this message is fresh and sent by Alice

The AP_REP message authenticates the server to Alice

- Only Alice, the server, and the KDC know K_{AS}
- If the timestamp is as expected, then this message is fresh

Question: How can we prevent replay attacks?

Using a Service Ticket: Message Detail

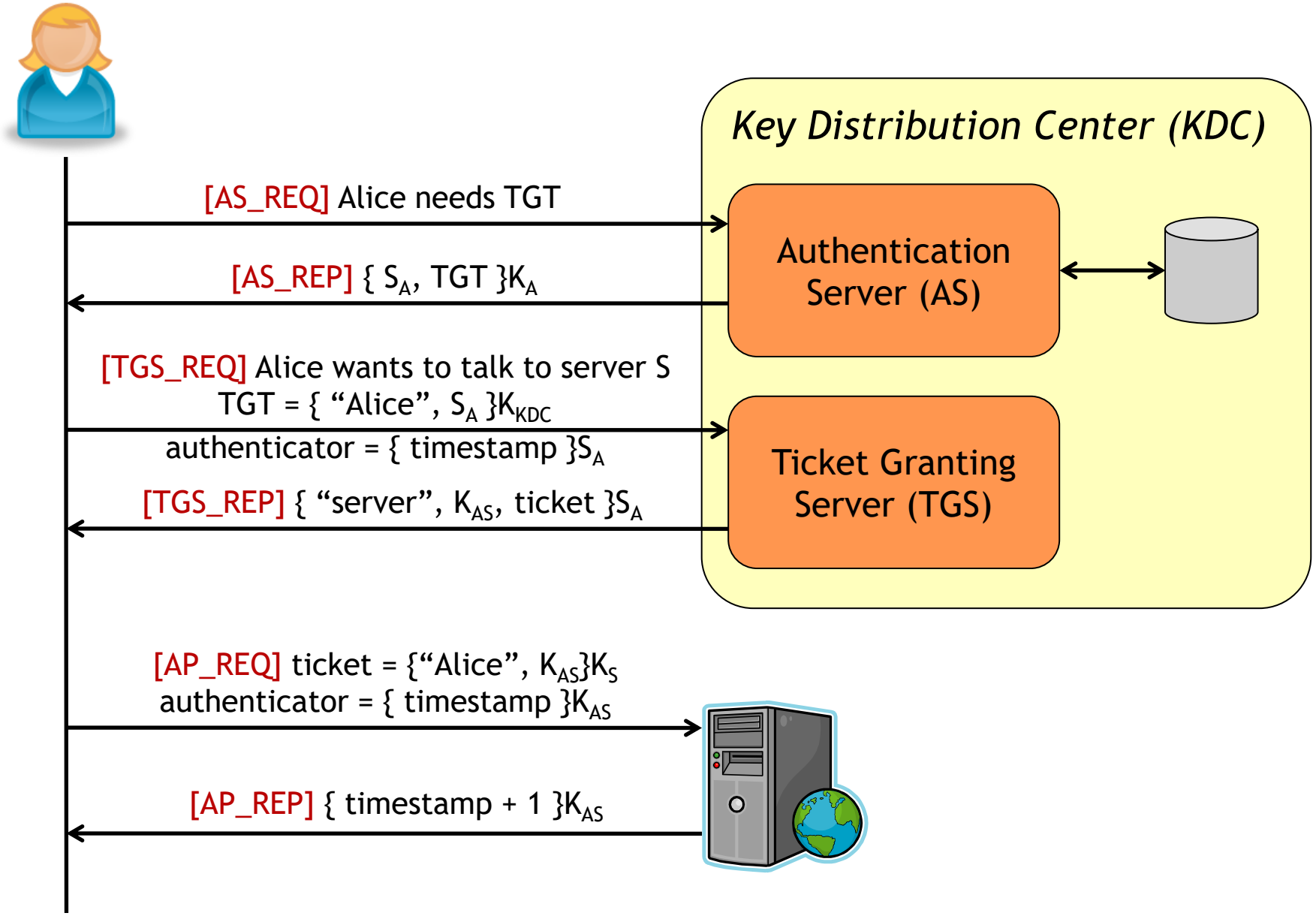
AP_REQ

Bytes	*Content*
1	Version of Kerberos (4)
1	Message Type (8)
1	Server's key version number
≤ 40	Server's realm
1	Length of ticket
1	Length of authenticator
var	ticket
var	authenticator

Copied from credentials field of TGS_REP

The ticket and authenticator follow the same format as in the TGS_REQ messages

Putting it all together…



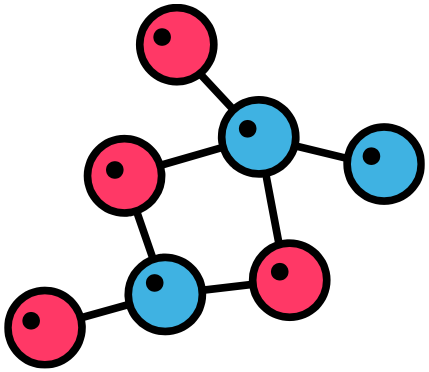
Is the assumption of a global KDC *really* realistic?

Problem: The KDC knows all keys!

- Probably not reasonable in mutually-distrustful organizations
- Scalability as number of users increases is poor
- Very valuable single point of attack

Problem: Reliability and fault tolerance

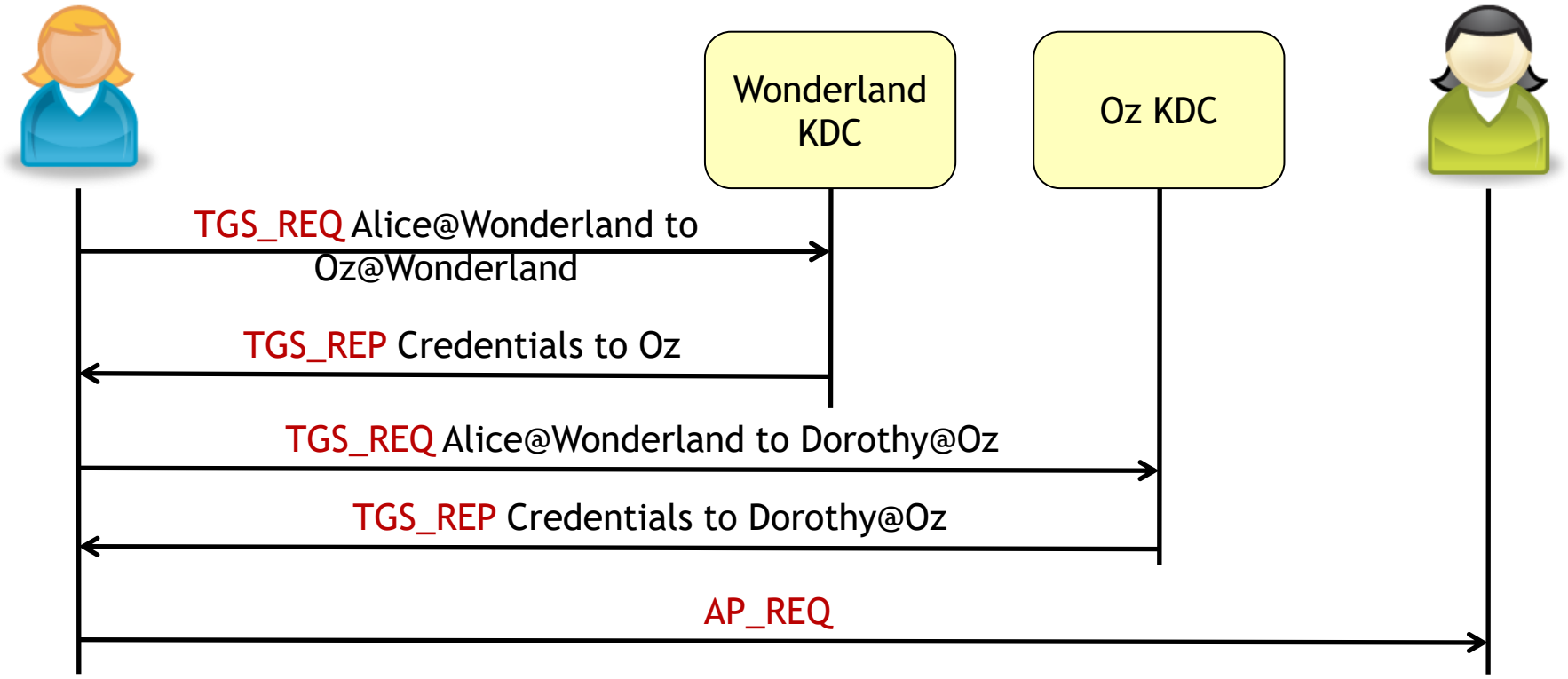
- A single KDC is a single point of failure
- If users cannot authenticate, they cannot work!
- Even if KDC is up all the time and everyone trusts it, it will probably not be able to serve all requests in a timely manner…



Kerberos is widely used, right? How does it address these problems?

Kerberos solves the untrusted KDC problem by allowing inter-realm authentication

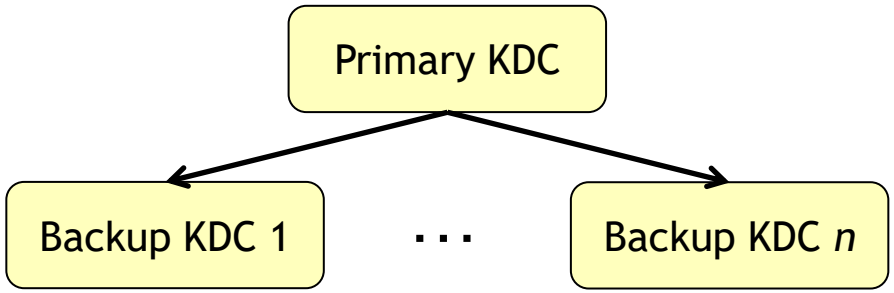
Example: Alice from the realm "Wonderland" wants to talk to Dorothy in the realm "Oz". Clearly, the Wonderland KDC knows nothing about principals managed by the Oz KDC. How do we proceed?



Note: For inter-realm authentication to work, the KDCs for each realm must agree to share keys a priori

Kerberos solves the failure and bottleneck problems by replicating the KDC's database

Goal: Any KDC should be able to service any client request



For this to work:

- The KDCs must all share the same master key
- The databases managed by each replica must be consistent with the primary

How can the replicated databases maintain consistency with the primary?

- Primary DB contents are periodically downloaded by the backups
- Backups are used exclusively for read only operations (Is this a problem?)

How is the DB protected during transmission?

- Confidentiality: Provided "for free" since DB is stored encrypted
- Integrity: Keyed hash using shared master key

Despite being a widely-deployed authentication solution, Kerberos v4 is far from perfect

Security

- Kerberos v4 is based on DES
- Integrity provided using non-standard techniques
- Does not support the use of other algorithms

Clocks and timestamps

- Maximum ticket lifetime is ~21.5 hours **(why?)**
- Cannot renew tickets
- Cannot obtain tickets in advance

Networks and naming

- 4 bytes used for network address
- What about IPv6??
- Names constrained to 40 characters

Kerberos v5 fixes these problems!

Supports extensible security suites. New algorithms can be added to the protocol as they are discovered. Standard techniques used for integrity protection.

Security

- Kerberos v4 is based on DES
- Integrity provided using non-standard techniques
- Does not support the use of other algorithms

ASN.1 used to encode names and addresses. Much more flexible.

Tickets have start and end times, can be renewed, and use a different timestamp format.

Clocks and timestamps

- Maximum ticket lifetime is ~21.5 hours
- Cannot renew tickets
- Cannot obtain tickets in advance

Networks and naming

- 4 bytes used for network address
- What about IPv6??
- Names constrained to 40 characters

Summary of Kerberos

Kerberos is a widely-used authentication paradigm based on the Needham-Schroeder authentication protocol

Authentication via Kerberos is a three-step process

1. Password-based authentication to the AS
2. TGT returned by the AS is used to request a service ticket from the TGS
3. Service ticket is used to mutually authenticate with a service

Inter-realm authentication allows users in different administrative domains to mutually authenticate

Many KDCs are replicated to prevent bottlenecks in the event of failure

Next: Public key infrastructure (PKI) models

Public Key Infrastructure

What is a public key infrastructure (PKI)?

Some common PKI models

- Monopoly
- Delegated CAs
- Oligarchy
- Anarchy

Case study: PKIX/X.509

Case study: PGP

What a **digital certificate** anyway?

Keeping track of public keys is simply not enough!

For instance, what if

- A transient fault switches Alice's and Bob's keys
- You and Trent have different opinions about who "Alice" is
- Trent is actually malicious and provides you with incorrect information
- Trent becomes compromised and the attacker provides false data
- …

In reality, key distribution servers manage digital certificates, rather than simple (name, key) pairs

Digital certificates are verifiable and unforgeable bindings between a user, a public key, and a trusted certifier (a certificate authority, or CA)

Terminology

If Alice issues a certificate vouching for Bob's name and key, then Alice is considered the issuer, and Bob is the subject

If Alice is verifying a certificate or chain of certificates, then she is called the verifier or relying party

Any user, machine, or service that has a certificate is called a principal

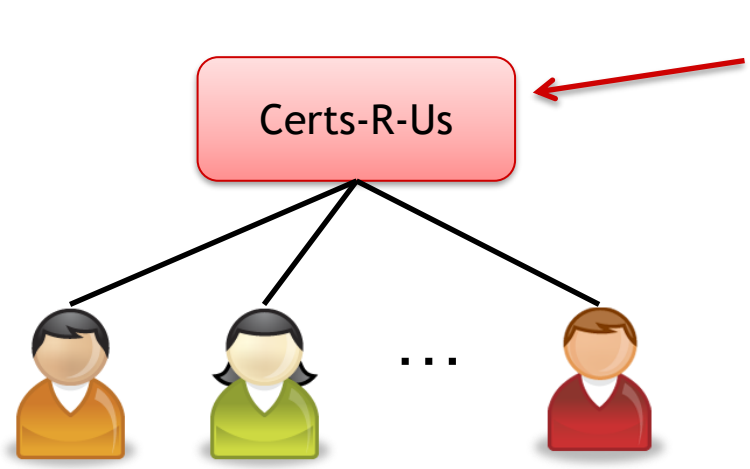
A trust anchor is a public key that is trusted by a verifier to certify the public keys of principals

---

Now that we have a basic vocabulary, let's explore a few ways that PKIs can be organized…

Most PKI models are hierarchical

A monopoly is the simplest form of PKI



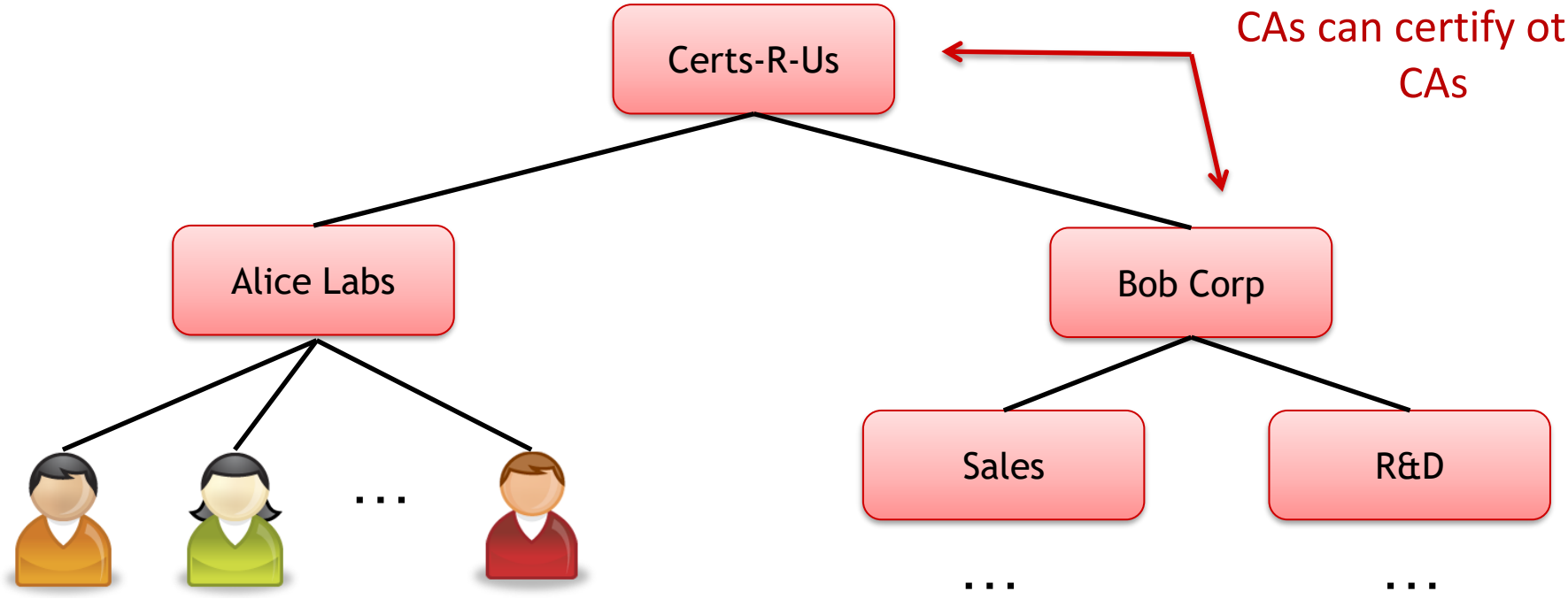
Pros:

- Very simple to model
- Only need to know a single public key to authenticate anyone

Cons:

- In real life, no organization is trusted by everyone
- Changing the key of that CA would be a nightmare
- How would a single CA verify the identity associated with every key?

Allowing delegation fixes some of these problems



Pros:

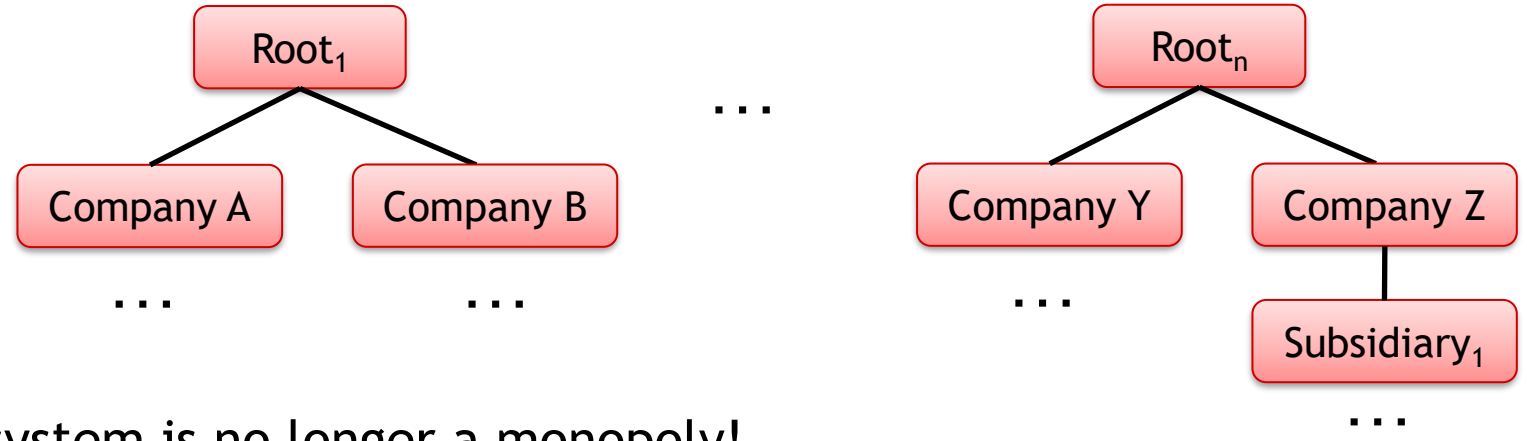
- Fairly simple to model
- Only need to know a single public key to verify a certificate chain that authenticates anyone
- Delegation of control makes identity verification more tractable

Cons:

- In real life, no organization is trusted by everyone
- Changing the key of that CA would be a nightmare

An oligarchy is a collection of hierarchies

Rather than pre-configuring a single trust anchor, allow applications to have a number of trust anchors



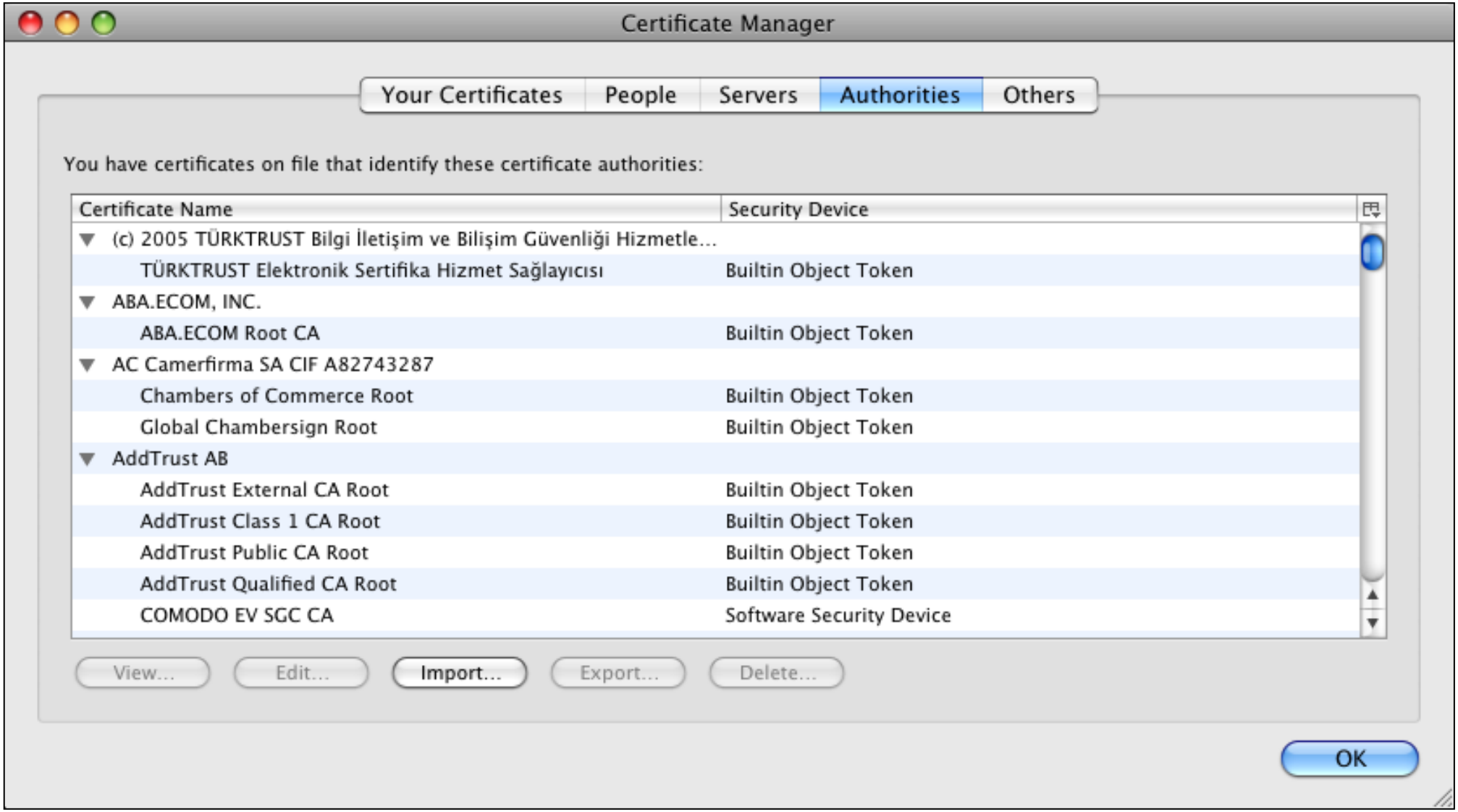
The system is no longer a monopoly!

- In a free market, this will make for more competitive certificate pricing
- There is no single point of attack

Unfortunately, this model is not without its problems

- Trust anchors are often chosen by application vendors, not users!
- If trust anchors are user configurable, it may be possible to trick users into adding a malicious root of trust!
- Managing a large set of trust anchors is complicated…

Your web browser uses the oligarchy model



In any PKI model, certificate issuance is a tricky issue

Say you find a certificate for Sherif Khattab, who is it *really* for?

- A lecturer at Pitt? (http://www.cs.pitt.edu/~skhattab/)
- A plastic surgeon? (https://www.Khattab.com)
- A pharmaceutical strategist? (https://www.linkedin.com/in/sherifkhattab/)
- many more …

- A certificate is only helpful if it unambiguously vouches for an identity
  - Am I really talking to Amazon.com, or is this a phishing site?
  - Did I just take CS course advice from a plastic surgeon?

Question: How can we *unambiguously* specify identities?

In our case studies, we'll see how this can be accomplished using unique identifiers and social connections

How can we deal with revoked certificates?

Over time, it may become necessary to revoke a certificate. For example,

- The private key becomes compromised (and the owner finds out)
- The binding between the name and certificate becomes invalidated

The revocation process can be handled in an offline manner using a certificate revocation list (CRL)

- Digitally-signed list of revoked certificates
- Issued periodically by the CA
- Always consulted prior to accepting a certificate

Note: This is how credit cards were checked for revocation in the "old" days

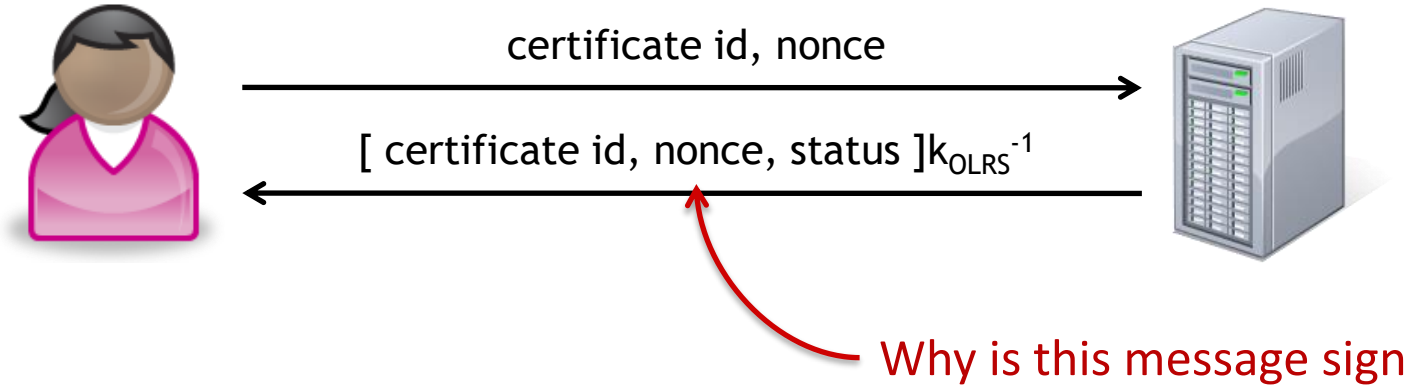
---

Question: Why do CRLs need to be signed?

Question: How can we make the distribution of large CRLs more efficient?

If we assume a reliable online service, we can implement a more timely revocation system

An online revocation server (OLRS) is an online service that is queried to check the validity of certificates



Usually, the OLRS is not the CA itself, but instead a service operating on behalf of the CA (Why?)

Alternatively, some OLRS pre-compute validity responses offline

- E.g., "Valid as of 08:00 on 10/19/2017"

Question: What are the tradeoffs between these two approaches?

Case study: X.509

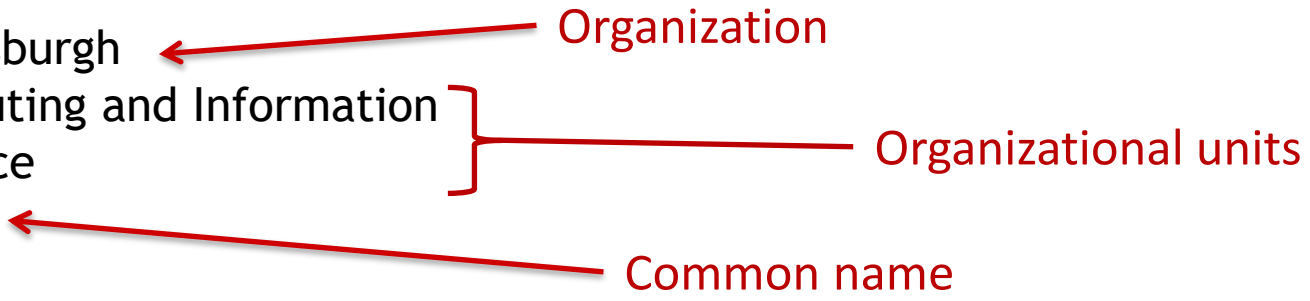
The IETF's PKIX working group is tasked with developing standards for deploying public key infrastructures based on the X.509 standard

This group has released a number of RFCs describing things like

- Certificate contents (RFC 5280)
- Certificate path validity checking (RFC 3280)
- Handling revocation lists (RFC 5280)
- Online certificate status protocols (RFC 2560)
- …

X.509 certificates uniquely identify principals through the use of X.500 distinguished names (DNs)

- /O=University of Pittsburgh
  /OU=School of Computing and Information
  /OU=Computer Science
  /CN=Sherif Khattab



X.509 Certificates

An X.509 certificate contains quite a bit of information

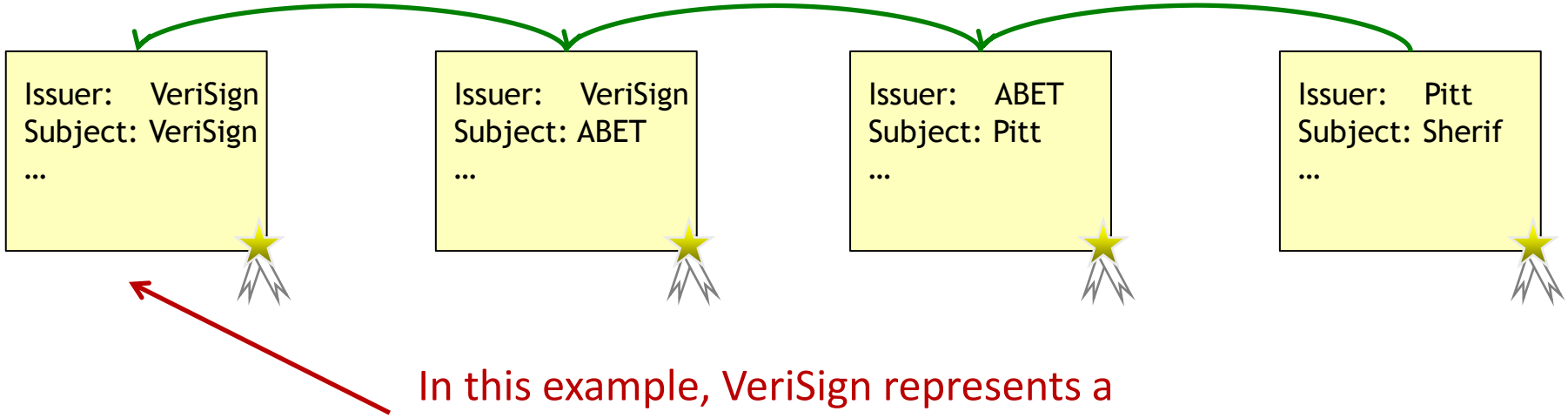
- Version
- Serial number: Must be unique amongst all certificates issued by the same CA
- Signature algorithm ID: What algorithm was used to sign this certificate?
- Issuer's distinguished name (DN): Who signed this certificate
- Validity interval: Start and end of certificate validity
- Subject's DN: Who is this certificate for?
- Subject's public key information: The public key of the subject
- Issuer's unique ID: Used to disambiguate issuers with the same DN
- Subjects unique ID: Used to disambiguate subjects with the same DN
- Extensions: Typically used for key and policy information
- Signature: A digital signature of (a hash of) all other fields

Note that each certificate has exactly one issuer

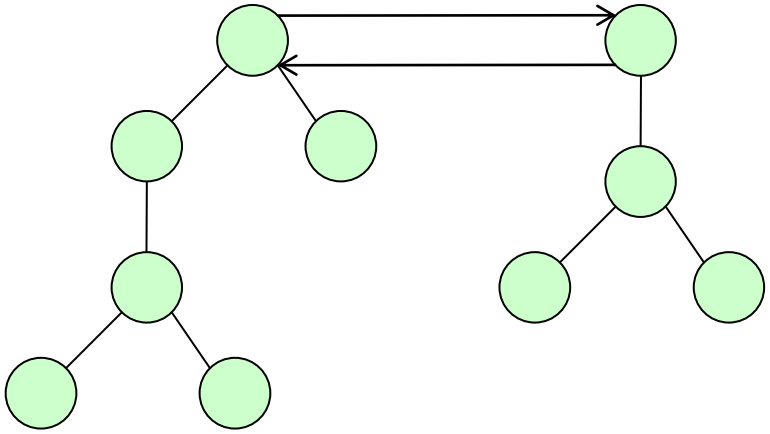
Question: What if you wanted a certificate to be authorized by more than one entity, but didn't want to deviate from the X.509 standard?

How can we validate X.509 certificates?

Since each certificate has exactly one issuer, we can build chains of trust



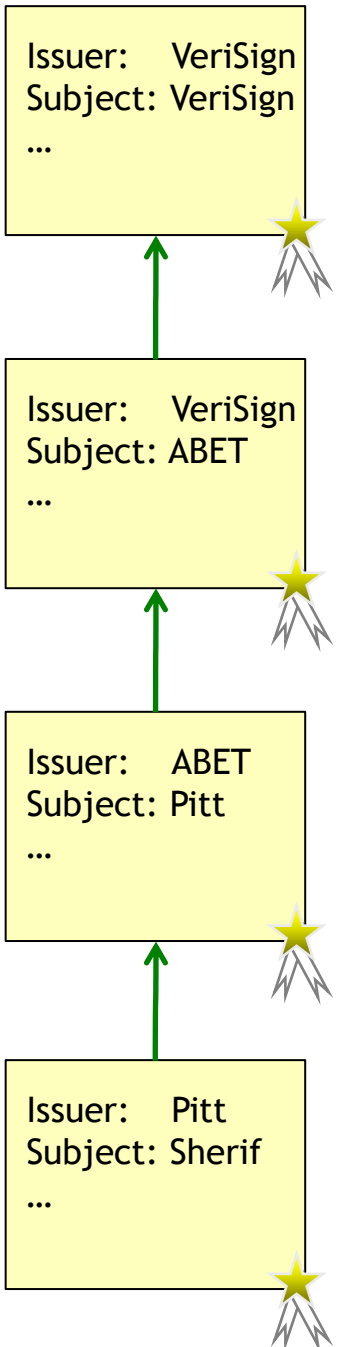
As you might expect, X.509 CAs form a hierarchy



Note: Cross-certification allows certificates from one domain to be interpreted in another

Question: Is there any value to allowing the use of self-signed client certificates within the X.509 model?

RFC 3280 describes how X.509 certificate chains are supposed to be validated



Checks run on certificate 1:

- Is this certificate a trust anchor?
- Is this certificate valid at the current time?
- Has this certificate been revoked?
- Is the self signature on this certificate valid?

Checks run on certificates 2 through *n*-1:

- Are these certificates CA certificates?
- Are these certificates currently valid?
- Have any of these certificate been revoked?
- Was certificate *i* signed by certificate *i*-1?

Checks run on certificate n:

- Is this certificate valid at the current time?
- Has this certificate been revoked?
- Was this certificate signed by certificate *n*-1?

Question: How do we find certificate chains in the first place?

Case study: PGP

Pretty Good Privacy (PGP) is a hybrid encryption program that was first released by Phil Zimmerman in 1991

In the PGP model

- Users are typically identified by their email address
- Users create and manage their own digital certificates
- Certificates can be posted and discovered by using volunteer "key servers"
- Key servers also serve a function similar to an OLRS

Email addresses are GUIDs, so they effectively disambiguate identities

However, note that there are no CAs in the system! As such, users can create certificates for any email address or identity that they want!

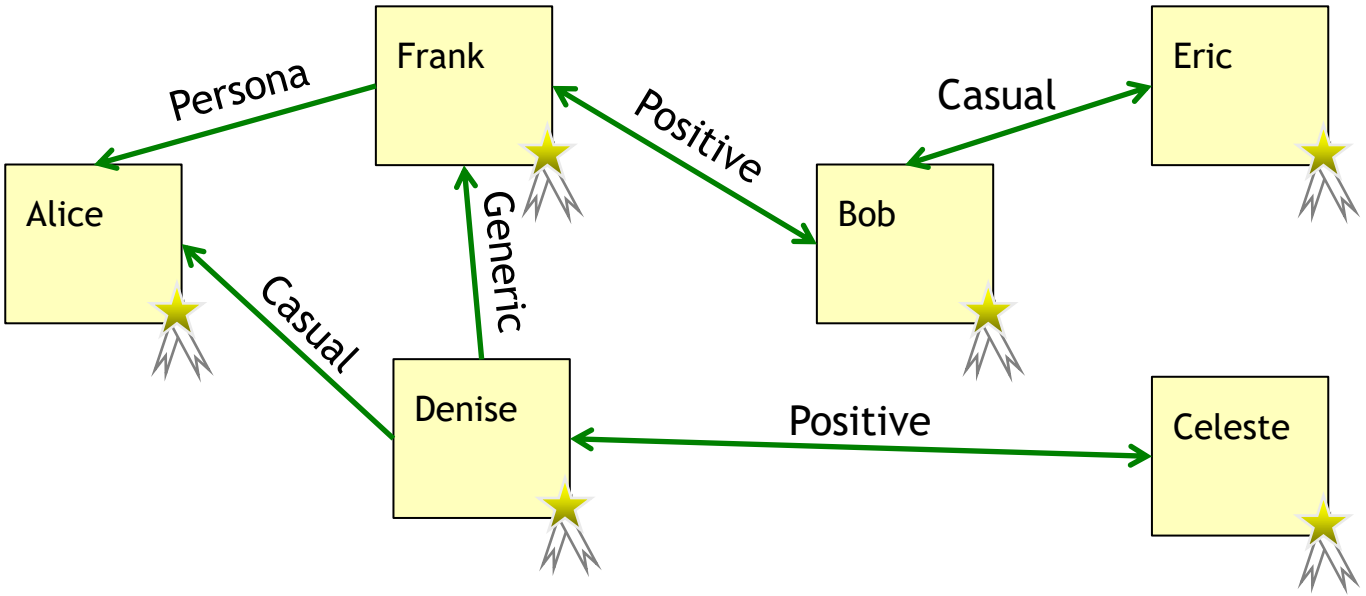
How can we deal with this anarchy?!

Unlike X.509, PGP takes a more "grassroots" approach to certificate validation

Users create their own certificates, and sign the certificates of others

- "I am Alice and I have verified that this certificate belongs to Bob"
- Four levels of certification: Generic, Persona, Casual, and Positive

This is essentially a cryptographic social network!



Question: Can Celeste trust Frank? Can Eric trust Frank?

PGP can handle two types of certificate formats

In addition to supporting X.509 certificates, PGP also has its own native certificate format

A PGP certificate contains the following information:

- PGP version number
- Public key: e.g., an RSA, DH, or DSA public key
- The certificate holder's "information": Free-form contents
- Signature: A self-signature on the certificate
- Validity period: When was this certificate created? How long is it valid?
- Preferred algorithms: What cipher suites should be used?

Question: If PGP keys are self-issued, why is a signature needed?

Discussion

Discussion Question

Which approach do you think is better? Why?

So Far …

Digital certificates act as verifiable and unforgeable bindings between an identity and a public key

There are many PKI models out there

- Some based on hierarchies (e.g., X.509)
- Some more ad-hoc (e.g., PGP)

In any deployment model, it is essential to unambiguously identify users

Handling revoked certificates can be done either online or offline

Next: Real-time communication security

Real-time communication security

Now we're going to talk about real-time security issues

Issues related to session key disclosure

- Perfect forward secrecy
- Forward secrecy
- Backward secrecy

Deniable authentication protocols

Denial of service

- Computational puzzles
- Guided tour puzzles

Real-time what?

Real-time security is a "catch all" term defined in the textbook

- i.e., Any interactive security protocol
- Basically: most authentication protocols

Note that the security protocols that we've discussed to date can be viewed as "soft" real-time protocols

- We require liveness (read: progress) in the system
- However, we do not usually have hard deadlines

We'll look at two classes of "real-time" properties today:

Key negotiation properties

Timeliness properties

Session keys: recap

Parties in cryptographic protocols typically have long-lived secrets

- Kerberos: K_A shared between Alice and the KDC
- PKI: Public/private key pairs (k_A, k_A^{-1})

Even so, it is good practice to generate new session keys regularly (Why?)

- Help limit compromises due to overuse
- Prevent breaks of one key from compromising all sessions
- …

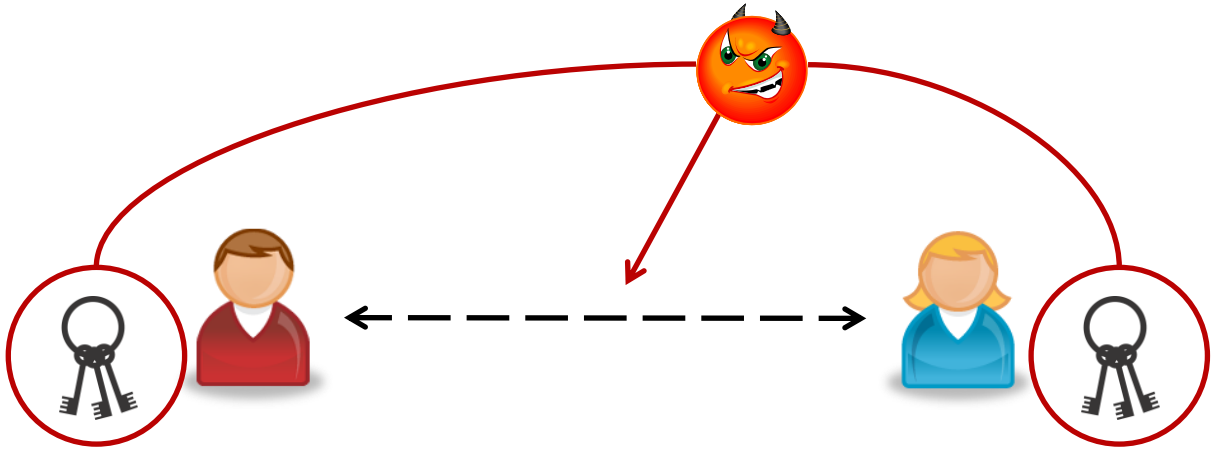
In general, session keys should be used for only a single session

Typically both parties contribute to the session keys for a protocol (Why?)

- If either party can generate "good" pseudorandom numbers, the derived key will be reasonably secure
- Less likely to have replay attacks lead to successful impersonations

Ideally, the session keys that we choose should have a property known as perfect forward secrecy

A key exchange protocol is said to have perfect forward secrecy if an adversary that (i) watches all messages exchanged and (ii) later compromises both parties cannot recover the agreed-upon session key



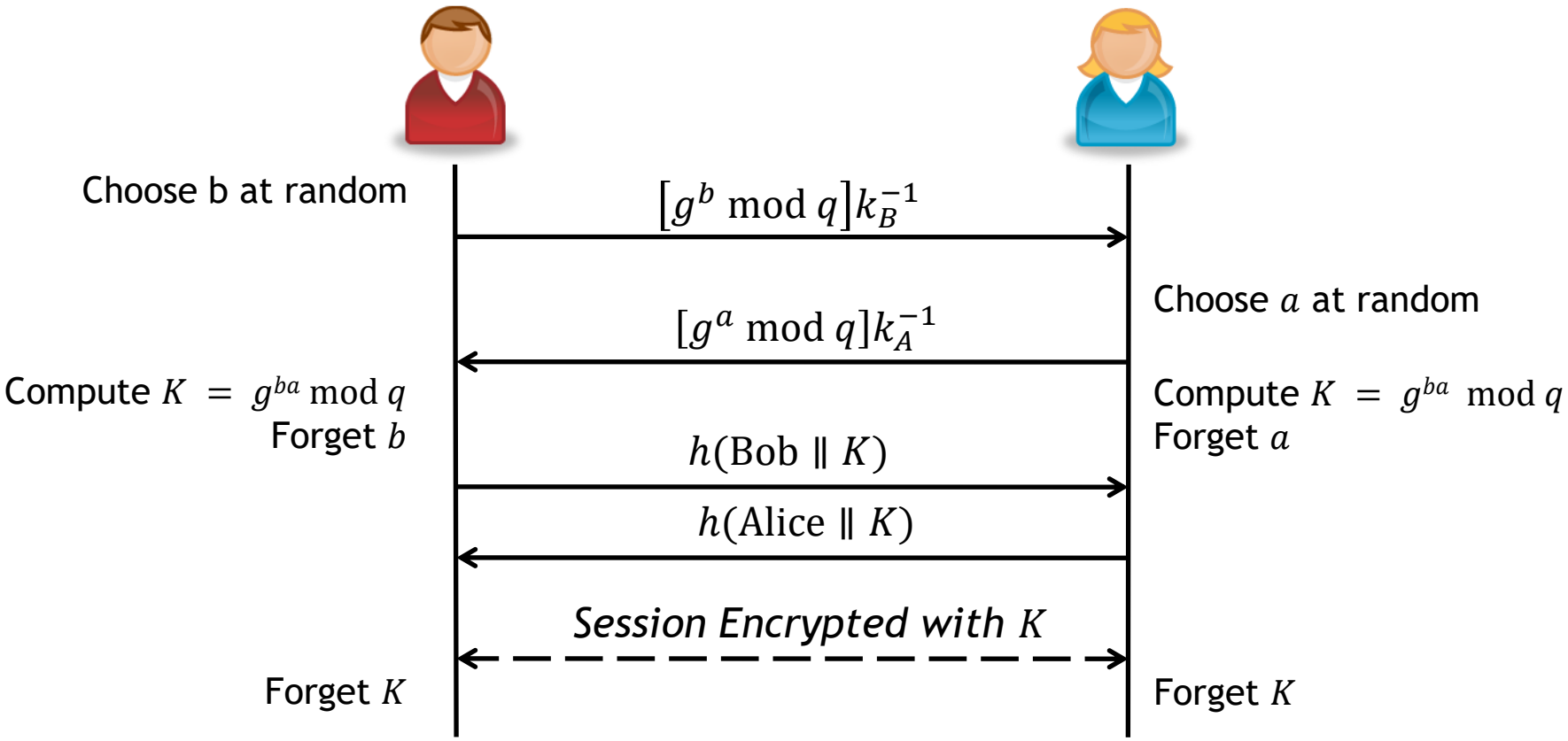
How do we achieve perfect forward secrecy?

- Generate a temporary secret
- Use only information that is not stored at the node long-term

---

This sounds good, but what protocols actually give us this property?

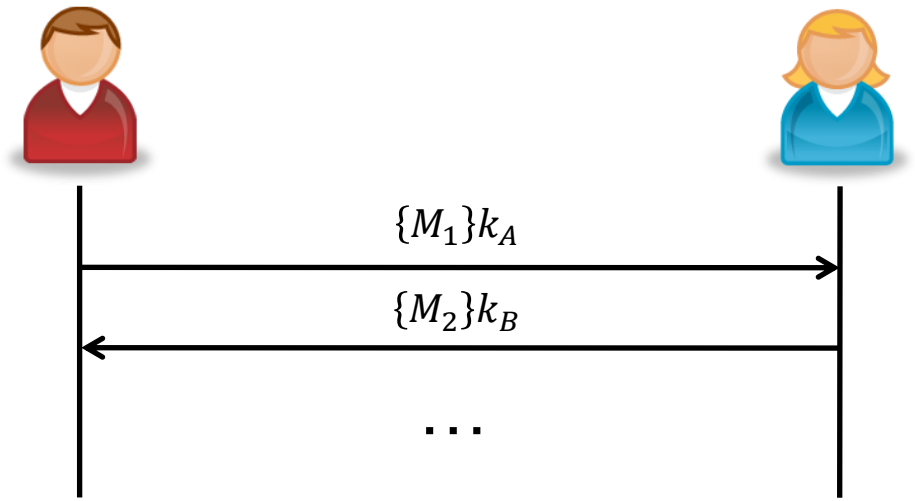
Signed Diffie-Hellman key exchange!



Question: Why does this protocol provide perfect forward secrecy?

- Adversary learns $g^a \bmod q$ and $g^b \bmod q$ by snooping
- Adversary learns k_B^{-1} and k_A^{-1} by compromise
- The important pieces are a, b, and K
  - These are not stored in the long term

Not all seemingly reasonable key exchange protocols provide perfect forward secrecy



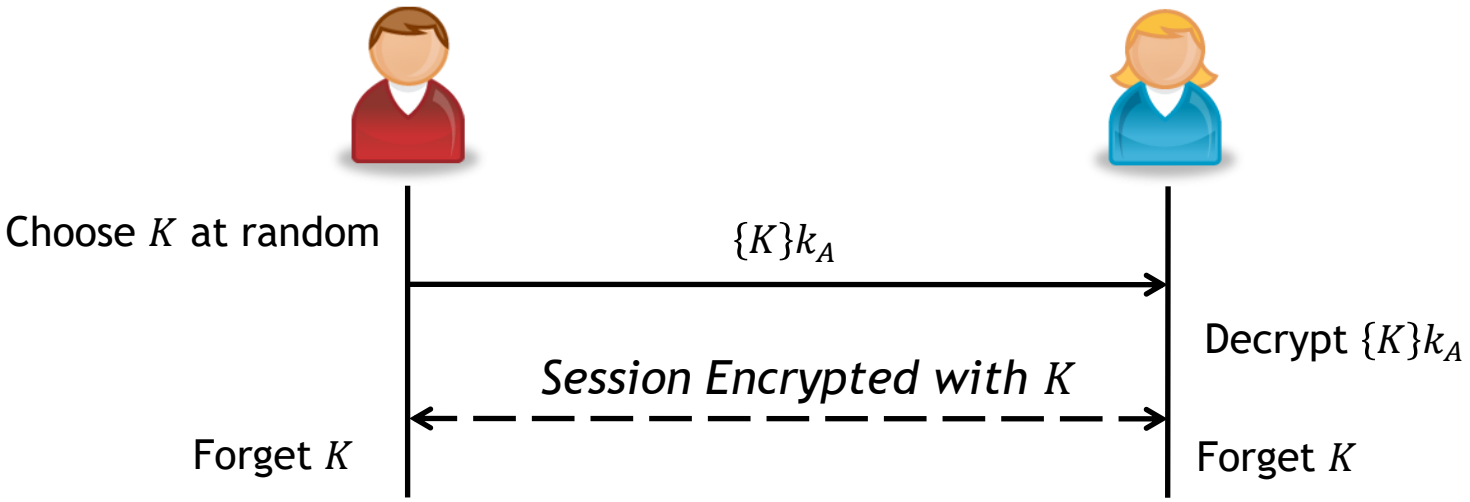
This protocol is inefficient and fails to provide perfect forward secrecy

What does the adversary learn?

- Monitoring: Nothing
- Compromising Alice and Bob: k_A^{-1} and k_B^{-1}

This compromises the protocol, as all messages can be recovered

Not all seemingly reasonable key exchange protocols provide perfect forward secrecy



This completely reasonable hybrid cryptosystem also fails to provide perfect forward secrecy (Why?)

What does the adversary learn?

- Monitoring: $\{K\}k_A$
- Compromising Alice and Bob: k_A^{-1} and k_B^{-1}

Given k_A^{-1} and $\{K\}k_A$, the adversary can recover K even though Alice and Bob have both forgotten it!

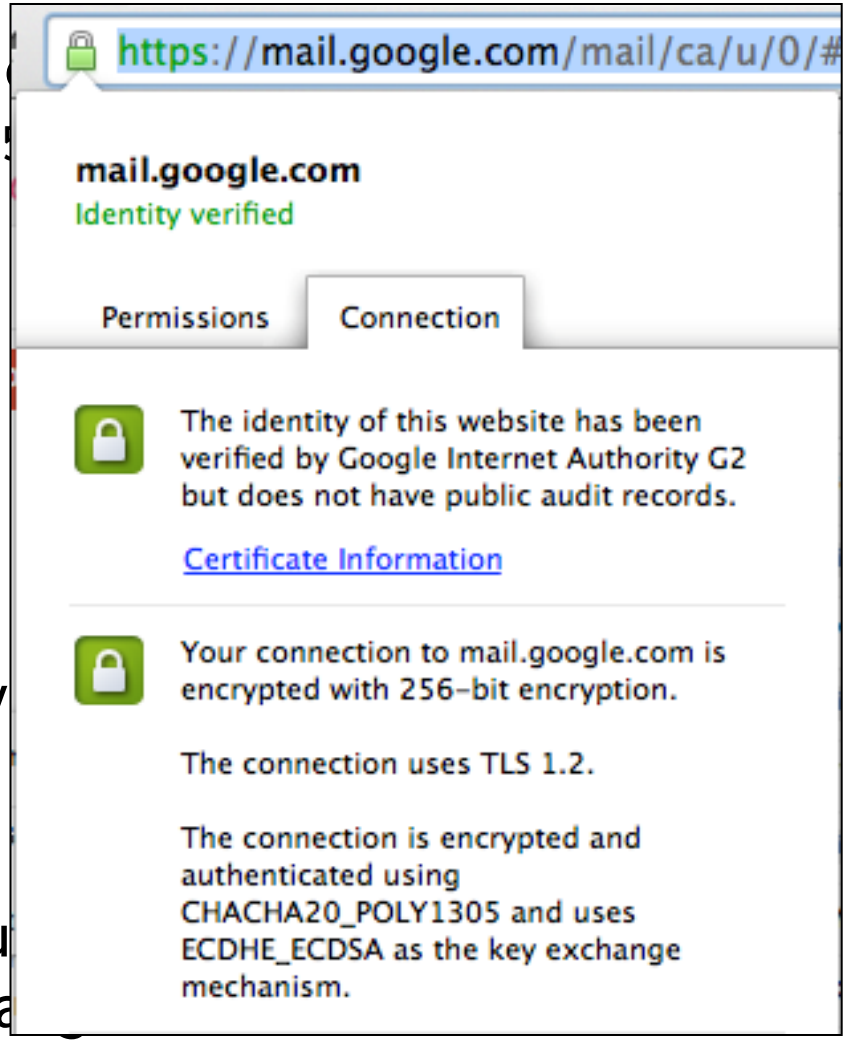
The previous protocol is similar to the “RSA key exchange” method supported in SSL/TLS

Many TLS cipher suites use RSA key

- TLS_RSA_EXPORT_WITH_RC4_40_MD5
- TLS_RSA_WITH_RC4_128_MD5
- TLS_RSA_WITH_RC4_128_SHA
- TLS_RSA_WITH_IDEA_CBC_SHA
- TLS_RSA_WITH_DES_CBC_SHA
- TLS_RSA_WITH_3DES_EDE_CBC_SHA
- …

Using these cipher suites, one party
with the RSA key of the other party

In November 2011, Google added su
using the ECDHE family of key excha



So far, we've only talked about pairwise keys… What about group keys?

Scenario: Secure chat rooms

Assume that we want a way for geographically distributed people to carry out secure conversations.

By secure, we mean that only parties that are part of the chat room can understand what is being said, even though all messages are exchanged over the public Internet.

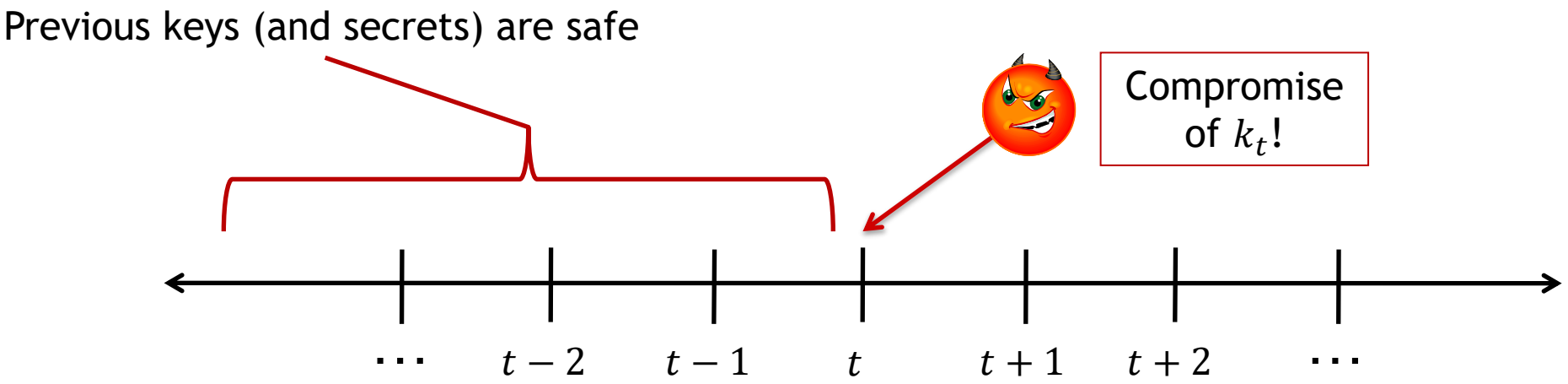
Simple idea: Set up a shared key for the chat room, and encrypt all messages using this key

Questions:

- Should people that join the group be able to decrypt old messages?
- Should people that leave the group be able to decrypt new messages?

Forward secrecy protects old information

A group communication scheme has forward security if learning the key k_t for time t does not reveal any information about keys k_i for all $i < t$.

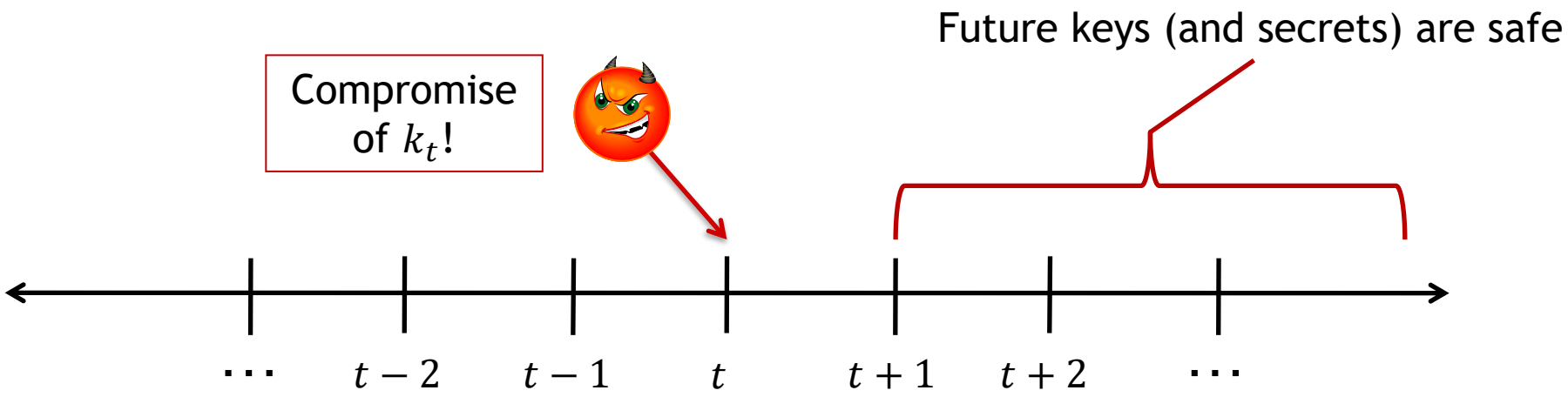


Informally: Later compromise does not reveal earlier keys

How can we achieve forward secrecy?

Backward secrecy is the complement of forward secrecy

A group communication scheme has backward security if learning the key k_t for time t does not reveal any information about keys k_i for all $i > t$.



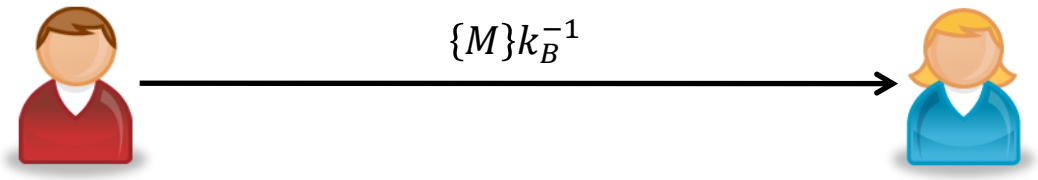
Informally: Earlier compromise does not reveal later keys

Question: Does our previous protocol also provide backward secrecy?

Deniability is another interesting property of authentication/key exchange protocols

Recall that a digitally signed message provides non-repudiability

- Alice can use k_B to verify that Bob signed the message
- But so can anyone else!

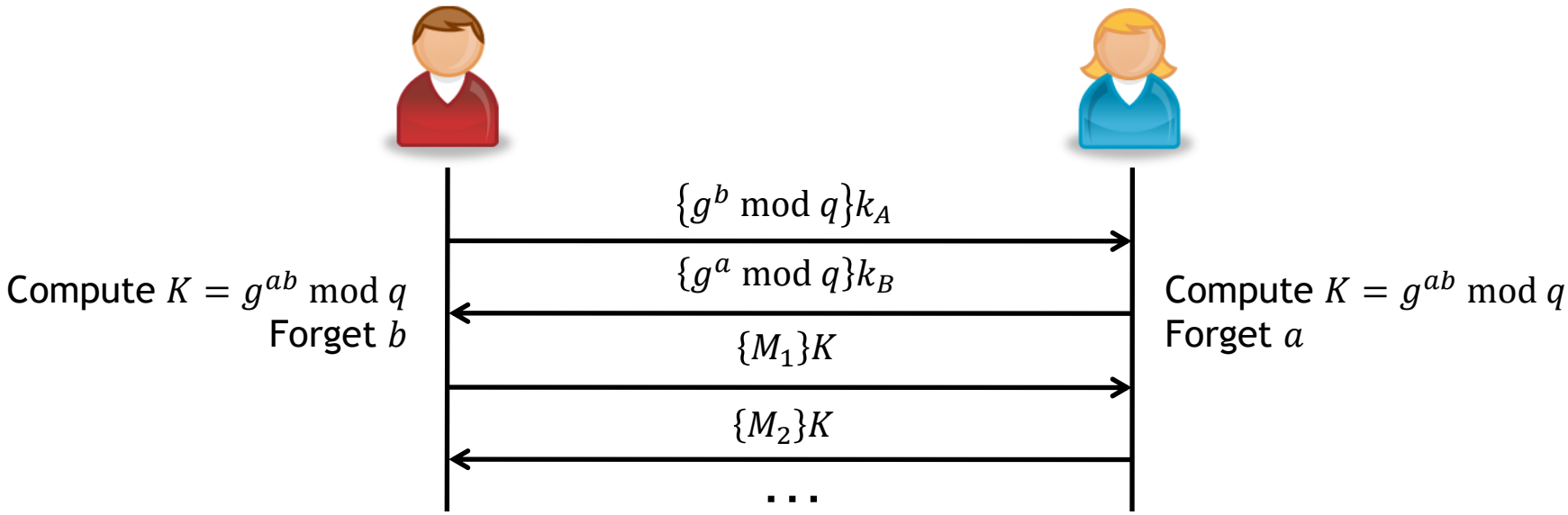


Sometimes, we would like Alice to be able to verify the authenticity of a message without being able to convince a third party that the message came from Bob

In essence, Bob would like some form of plausible deniability

More formally, a protocol between two parties provides plausible deniability if the transcript of messages exchanged could have been generated by a single party

How can we provide plausible deniability?



This protocol mutually authenticates Alice and Bob

- Alice knows that only Bob could have derived K
- Bob knows that only Alice could have derived K
- Each party knows which M_is they sent

Anyone could have generated the transcript!

- Only need k_A and k_B, which are public
- The rest is random numbers!

Lesson: Causality is important. Notions of "I sent" or "I received" are *not* part of the transcript, but are part of real life.

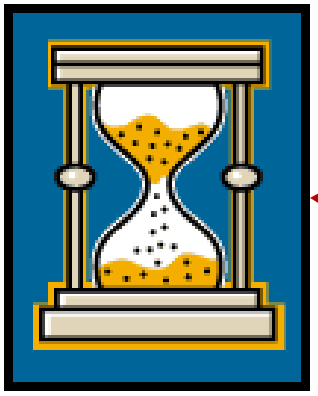
Computer security is typically defined with respect to three types of properties

How do I ensure that my secrets remain secret?

Confidentiality

Can I trust the services that I use?

Integrity

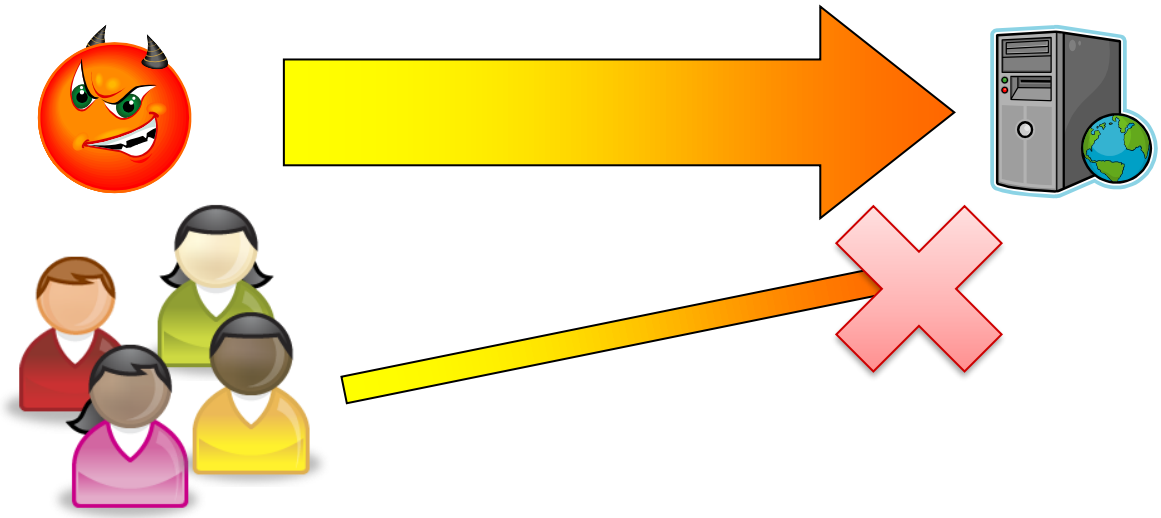


Availability

Am I able to do what I need to do?

Denial of Service (DoS)

A denial of service (DoS) attack occurs when a malicious client is able to overwhelm a server, thereby preventing service to legitimate clients



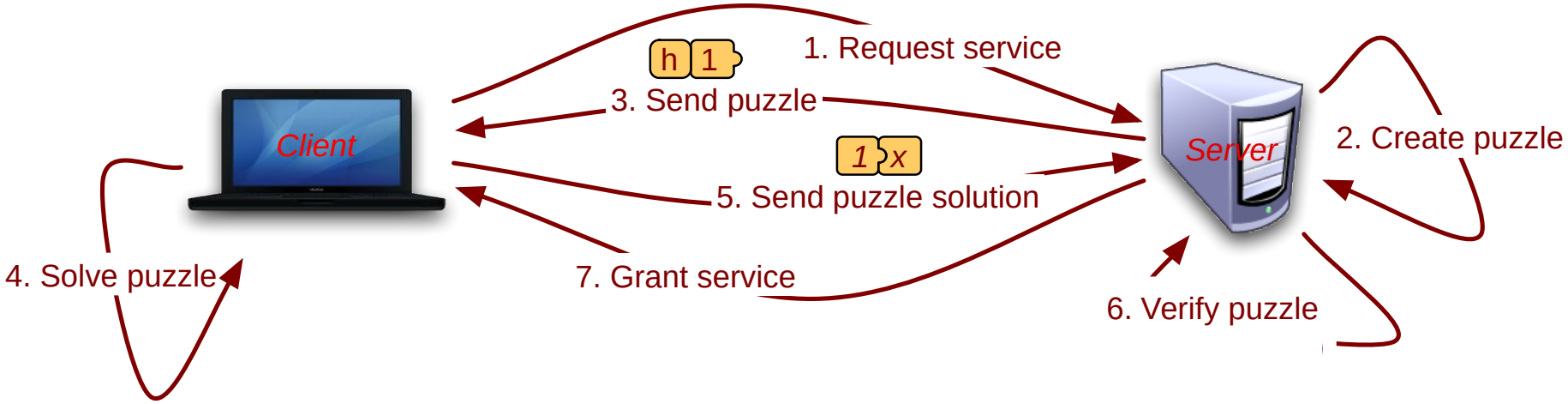
Denial of service attacks are typically abusing a resource disparity

- Easy to generate lots of network requests, hard to maintain server state
- Easy to create random application data, decryption takes time/cycles
- …

Avoiding resource disparity is a good design principle, but is not always easy to do in practice

Computational puzzles can be used to mitigate DoS attacks

Idea: Make clients pay for their requests by solving a hard puzzle first



Computational puzzles must satisfy three requirements:

1. Puzzles should be easy for the server to generate
2. Puzzles should be hard for the client to solve
3. Puzzle solutions should be easy for the server to verify

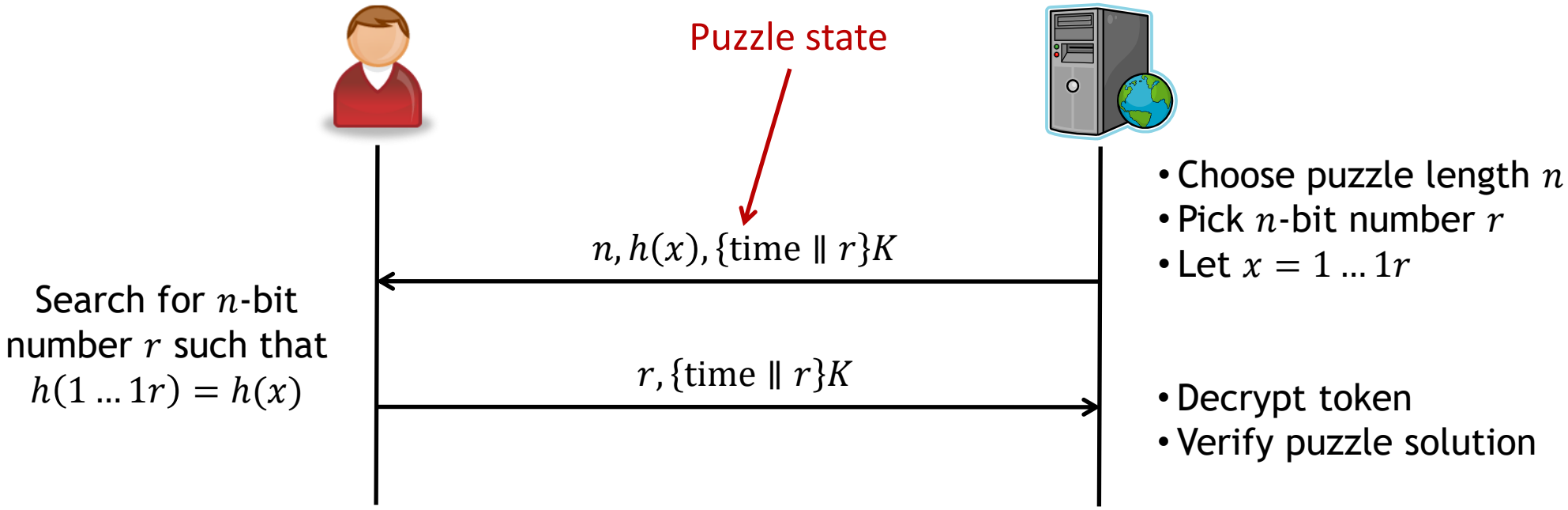
Question: Why do computational puzzles have these requirements?

Example: Hash inversion

Intuition: It should be very hard to invert a cryptographic hash function

- Recall that "hard" means $O(2^m)$ work where m is the hash bit length

Hash inversion puzzles work as follows:



Note: Puzzle hardness can be tuned by using different length values (n)

Question: Why is the puzzle state offloaded to the client?

Discussion

When would computational puzzles be effective for mitigating DoS attacks? When are they ineffective?

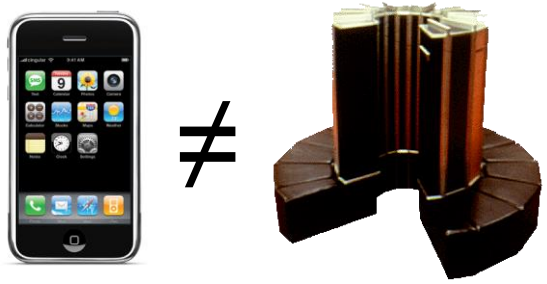
Strengths and weaknesses of computational puzzles

When do computational puzzles work well?

- All clients have approximately similar computational ability
- Puzzles cannot be parallelized between multiple clients
- Puzzles can be efficiently generated

Unfortunately, computational puzzles are not a panacea…

- Not all clients are created equal
- Clients have better things to do than burn cycles solving puzzles
- Attackers often have a means of parallelizing puzzles



Recent work in our department is attempting to address this problem

Observation: Attackers can fairly easily control:

- Their own computational abilities
- A subset of nodes in the network (e.g., via compromise)
- The quality of their connection to the network

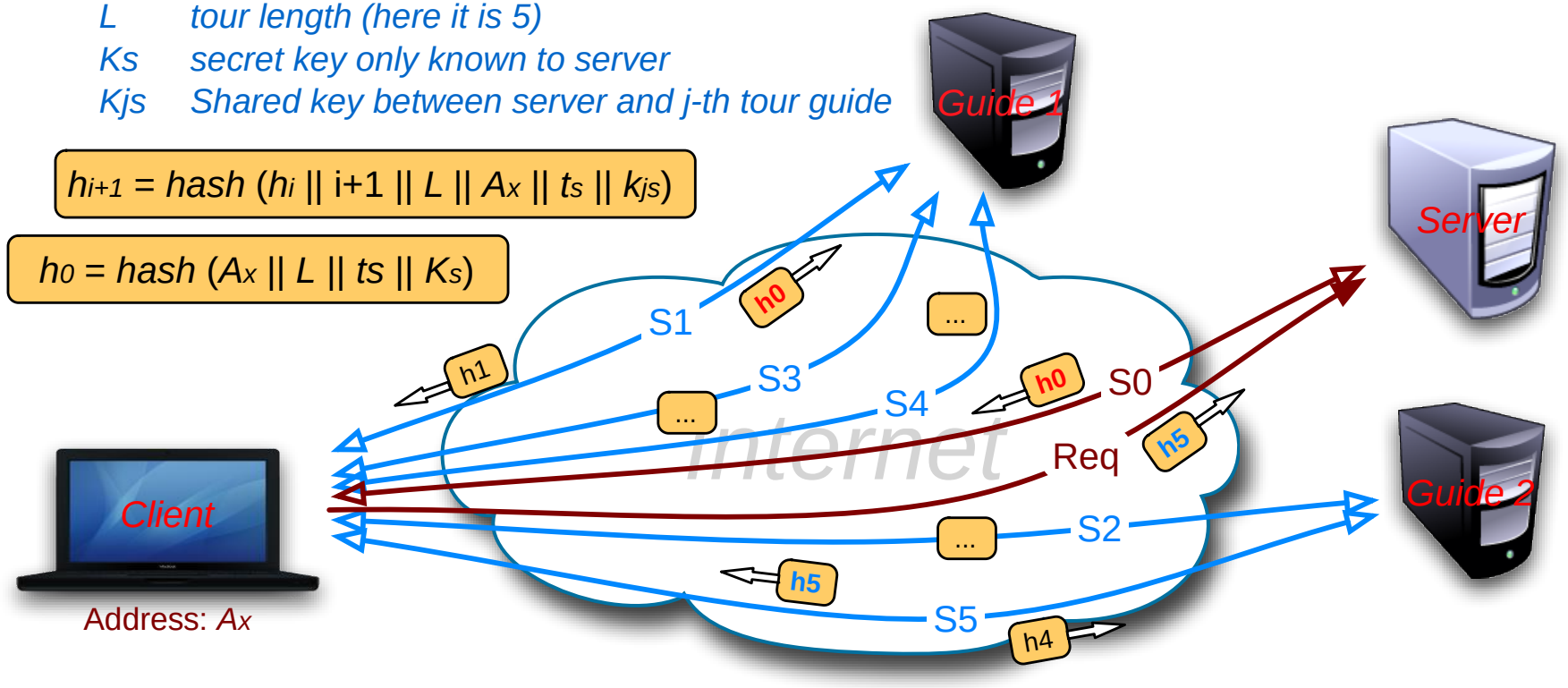
However, they cannot control the overall latency characteristics of routes that traverse segments of the Internet

Rather than asking clients to solve computational puzzles, we can ask them to provide evidence that they have carried out an ordered traversal of some number of nodes

Guided tour puzzles do exactly this!

- Puzzle: An ordered list of tour guides to contact
- Solution: A serial computation carried out by these nodes in order

How do guided tours work?



Guided tour puzzles are efficient to check

- Next tour guide chosen using a single hash operation
- A length n tour is checked using n hash operations

Cheating is hard

- Cannot guess next tour guide without knowing the secret key
- Cannot control the delay characteristics of the Internet

Summary

Today we talked about a variety of real-time issues

Session key security issues include:

- Perfect forward secrecy: Session keys safe even if long term keys compromised
- Forward secrecy: Compromising current key does not compromise previous keys
- Backward secrecy: Compromising current key does not lead to the compromise of future keys

Deniable protocols have completely forgeable transcripts, yet still provide authentication, confidentiality, and integrity protection

DoS attacks impact system availability

Puzzles can be used to mitigate DoS attacks

Transport Layer Security (TLS)

What is TLS and why do we need it?

How does TLS work?

- High-level intuitive explanation
- Packet-level details
- Key derivation
- Session resumption

PKI considerations when using TLS

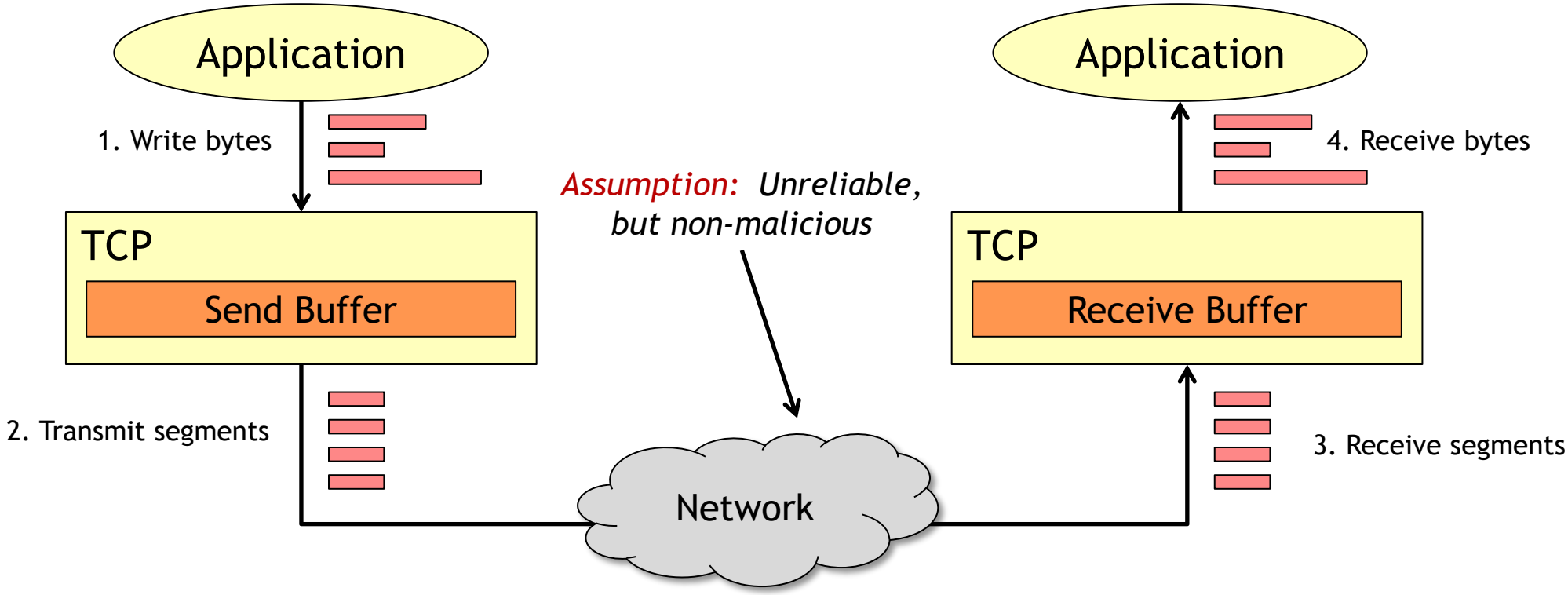
TCP is reliable, but not secure

TCP provides a reliable transport service

- Acknowledgements ensure that data is eventually delivered
- Checksums help protect packet integrity in the event of transient failure

Please note that:

- Active attackers can change the contents of packets
- Checksums are not cryptographic, so they can be forged



SSL and TLS were developed to provide applications with a secure means of utilizing TCP

Goal: Provide a generic means of authentication, confidentiality, and integrity protection to networked applications

That is, SSL/TLS were designed to simplify network security in the same way that Berkeley sockets simplified network programming

Where is SSL/TLS used?

- Web browsers
- Protecting FTP (FTPS)
- POP/IMAP via STARTTLS
- VoIP security
- …

TLS protection in Chrome

Authentication in SSL can be one-way or mutual

- One way: Web browser authenticating
- Mutual: B2B web services transactions

Historical Context

Building a protocol suite for secure networking applications is a great idea. As such, there were many attempts made at this.

Secure Sockets Layer (SSL) was developed by Netscape

- Version 1 was never deployed
- Version 2 appeared in 1995
- Version 3 appeared in 1996

Microsoft developed PCT by tweaking SSL v2

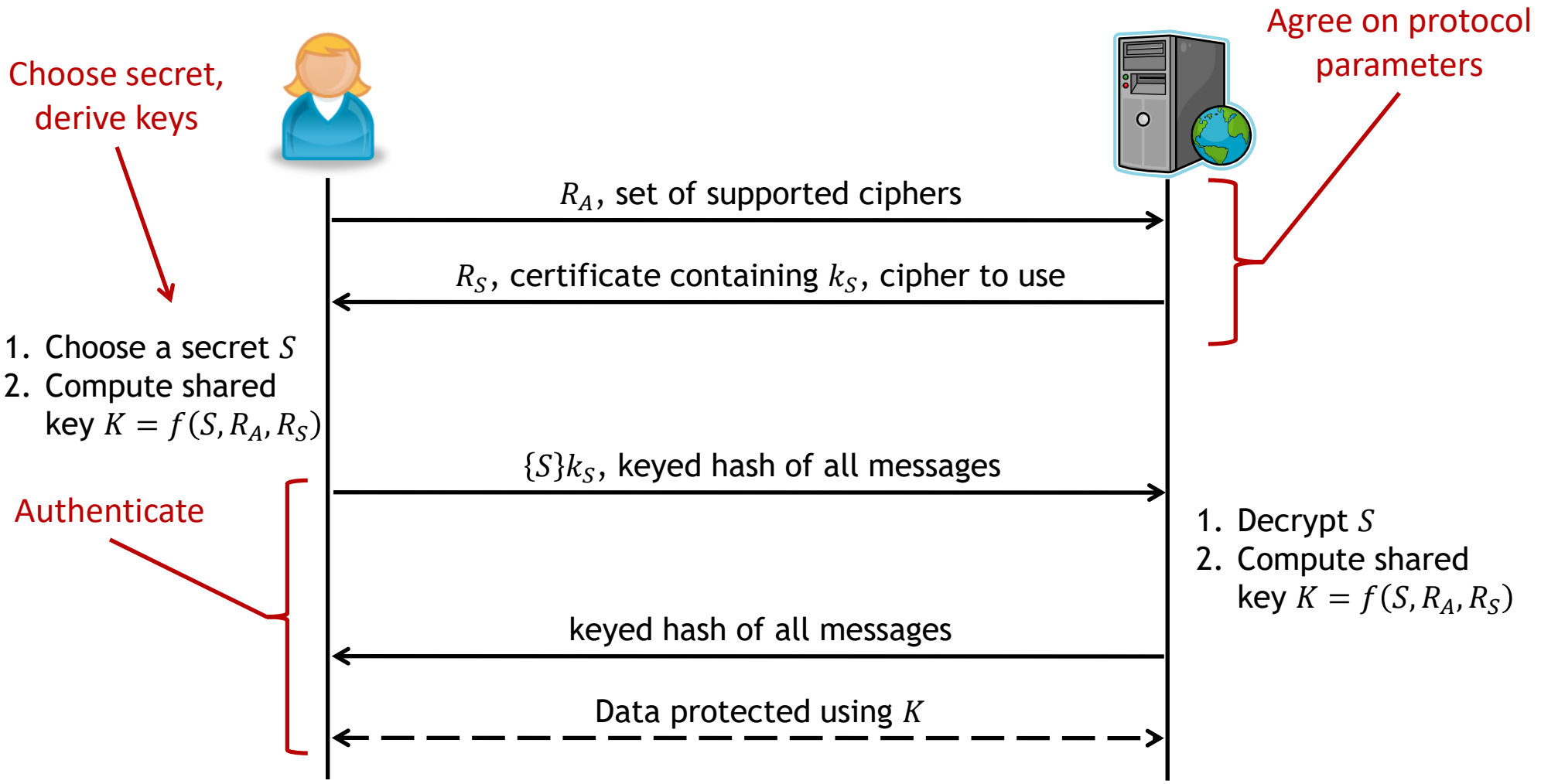
Microsoft

- This was mostly a political move
- Forced Netscape to turn control of SSL over to the IETF

The IETF then developed the Transport Layer Security (TLS) protocol

- TLS 1.0 released in 1999 (RFC 2246)
- Most current version is TLS 1.3, which was released in 2018 (RFC 8446)

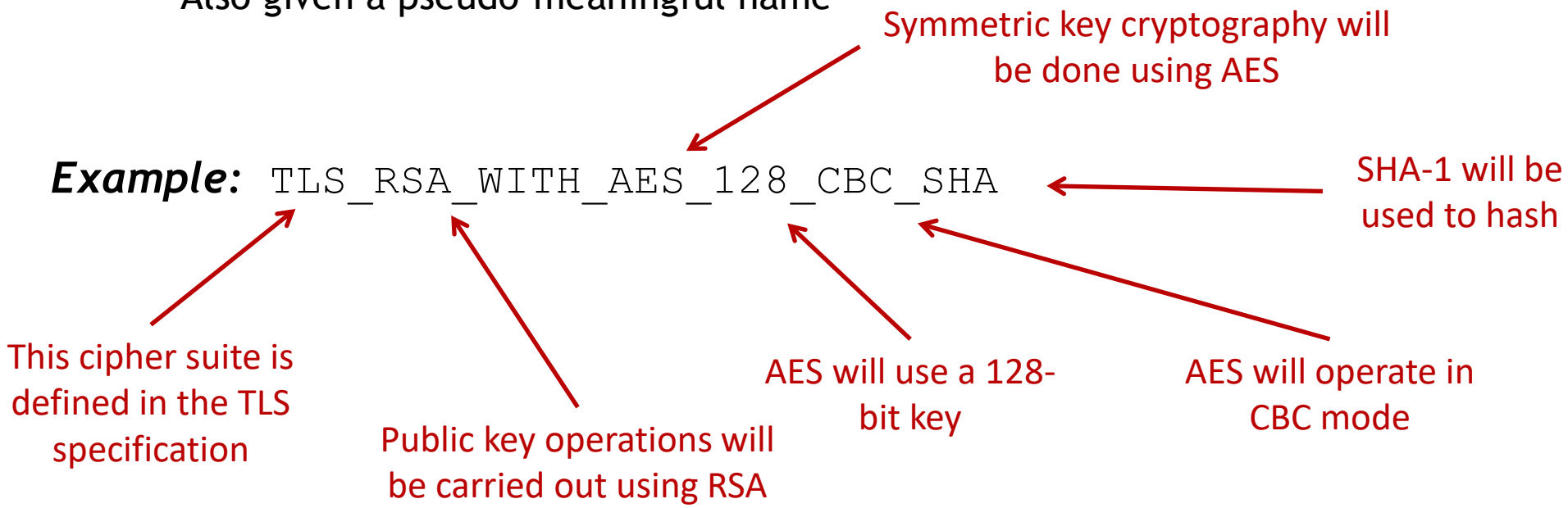
Viewed abstractly, TLS is actually a very easy to understand protocol



What is a cipher suite?

A cipher suite is a complete set of algorithmic parameters specifying exactly how the protocol will run

- Specified using a 16-bit identifier
- Also given a pseudo-meaningful name



In TLS, the client proposes a list of supported cipher specifications, and the server gets to choose which one will be used

Discussion

Why are cipher suites a good idea?

TLS transmits data one record at a time

TLS defines four specific message types

- Handshake records contain connection establishment/setup information
- The ChangeCipherSpec record is a flag used to indicate when cryptographic changes will go into effect
- Alert records contain error messages or other notifications
- Application_Data records are used to transport protected data

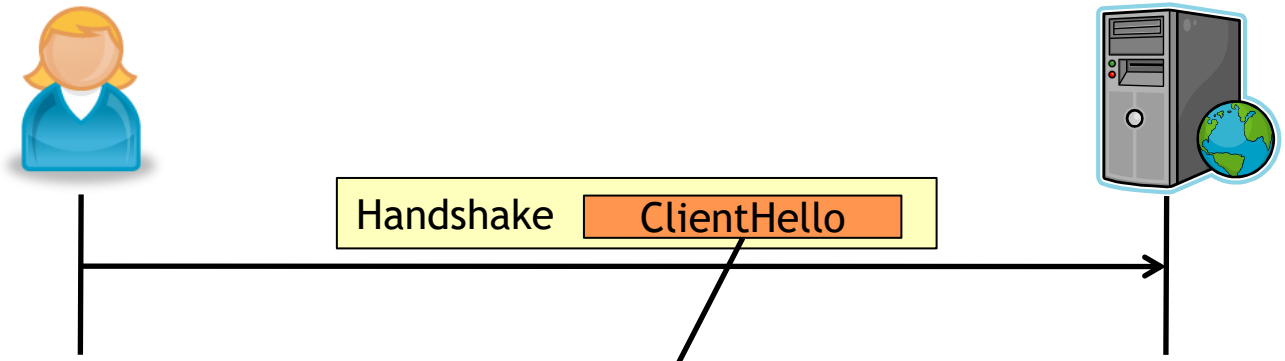
Note that a single record may contain multiple messages inside of it

Each record is delineated using a 5 byte record header

Bytes	***Content***
1	Record type code
2	Version number
2	Length

To illustrate this record/message sending process, let's look into the details of the protocol…

Handshake: Message 1

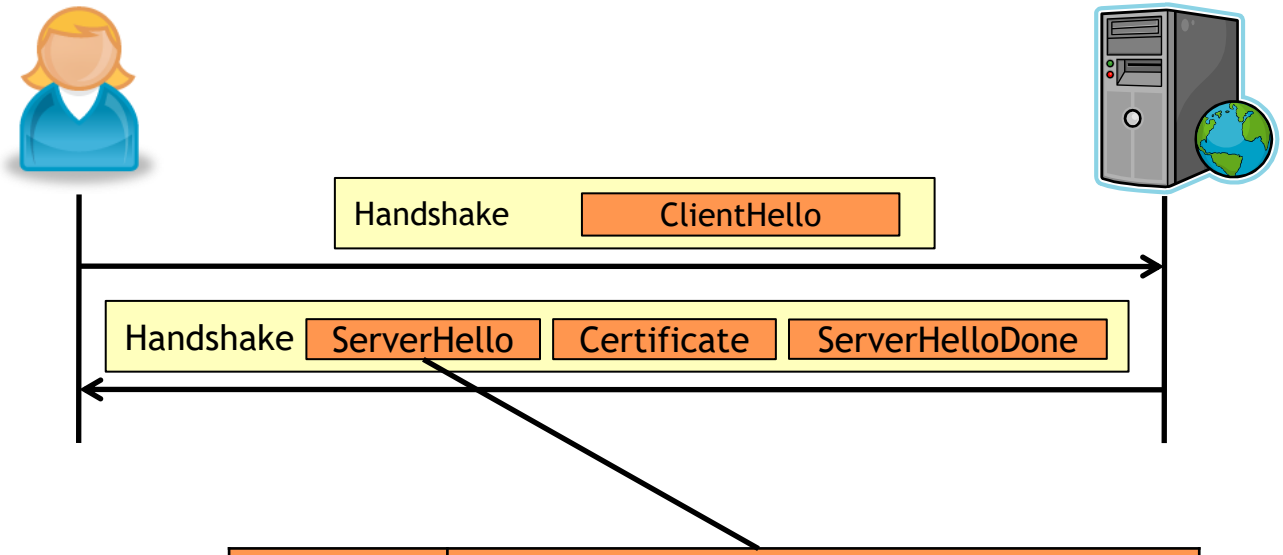


Bytes	*Content*
1	Message type: ClientHello
3	Length
2	Version number
32	R_A
1	Length of session ID
var	Session ID
2	Length of cipher suite list
var	List of supported cipher suites
1	Length of compression mode list
var	List of supported compression modes

Used to resume sessions. More on this later.

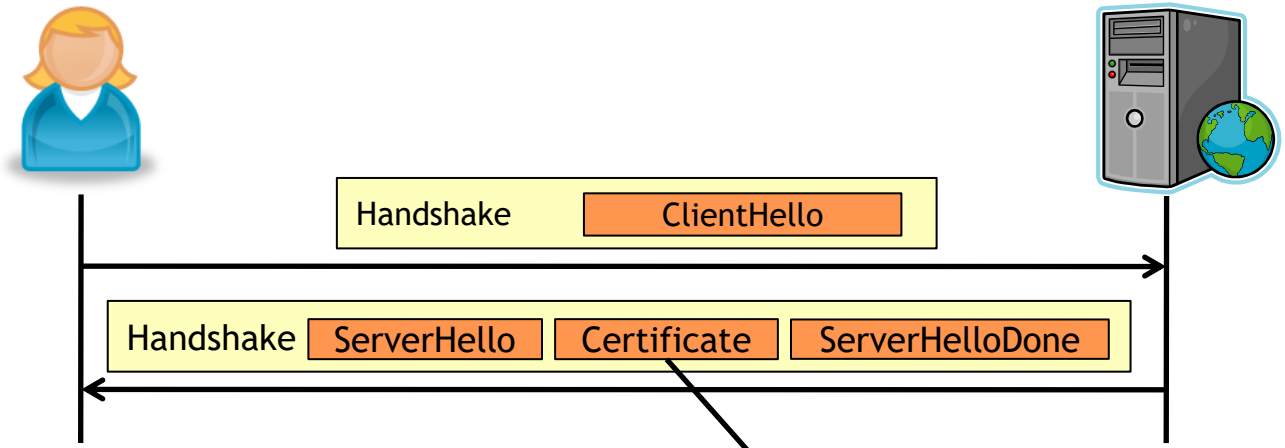
For efficiency reasons, TLS compresses the data that it transmits.

Handshake: Message 2



Bytes	*Content*
1	Message type: ServerHello
3	Length
2	Version number
32	R_S
1	Length of session ID
var	Session ID
2	Chosen cipher suite
1	Chosen compression method

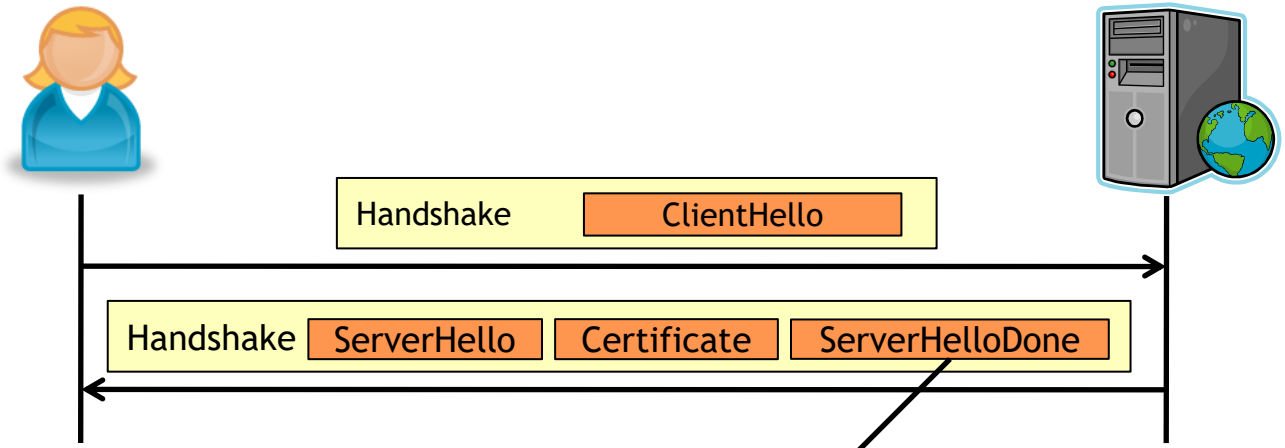
Handshake: Message 2



Bytes	*Content*
1	Message type: Certificate
3	Length
3	(Redundant) length
3	Length of first certificate
var	First certificate
var	More (length, certificate) pairs

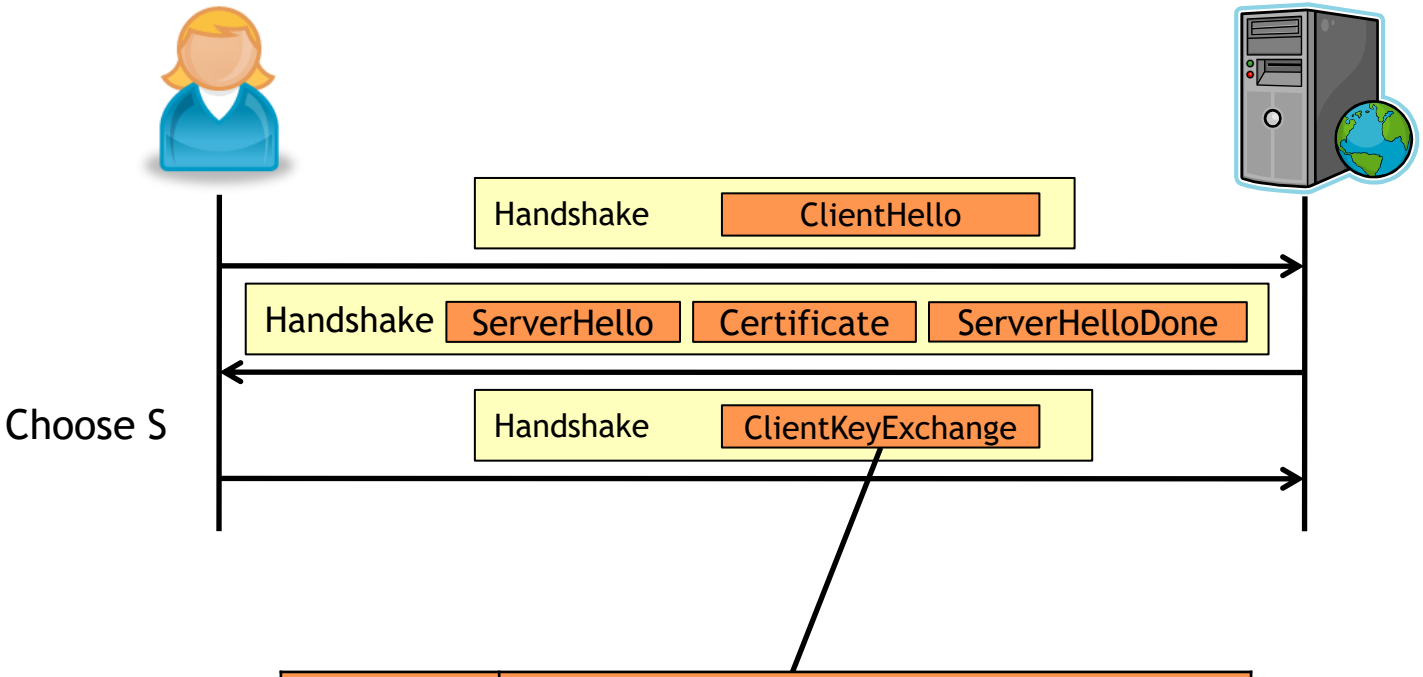
Why?

Handshake: Message 2



Bytes	*Content*
1	Message type: ServerHelloDone
3	Length = 0

Handshake: Message 3



Bytes	*Content*
1	Message type: ClientKeyExchange
3	Length = 0
2	(Redundant) length
var	Encrypted pre-master secret $\{S\}k_S$

How are keys computed?

How are session keys computed?

RFC 5246 defines a pseudorandom function that is used to expand the master secret into an randomized key stream:

```
PRF(secret, seed) =
    HMAC(secret, A(1) || seed) ||
    HMAC(secret, A(2) || seed) ||
    HMAC(secret, A(3) || seed) || ...
```

The sequence A is defined as follows:

- $A(0) = \text{seed}$
- $A(i) = \text{HMAC}\big(\text{secret}, A(i-1)\big)$

Note: The version of HMAC that is used depends on the cipher suite!

- E.g., `TLS_RSA_WITH_AES_128_CBC_SHA` uses HMAC-SHA-1

Key derivation, continued

First, the master secret is computed:

- `master_secret = PRF(pre_master_secret, "master secret" || R_A || R_S)[0..47]`

Next, a stream of random bytes is generated from the master secret

- `key_block = PRF(master_secret, "key expansion" || R_S || R_A)`

Keys are taken from the above block of key material in the following order

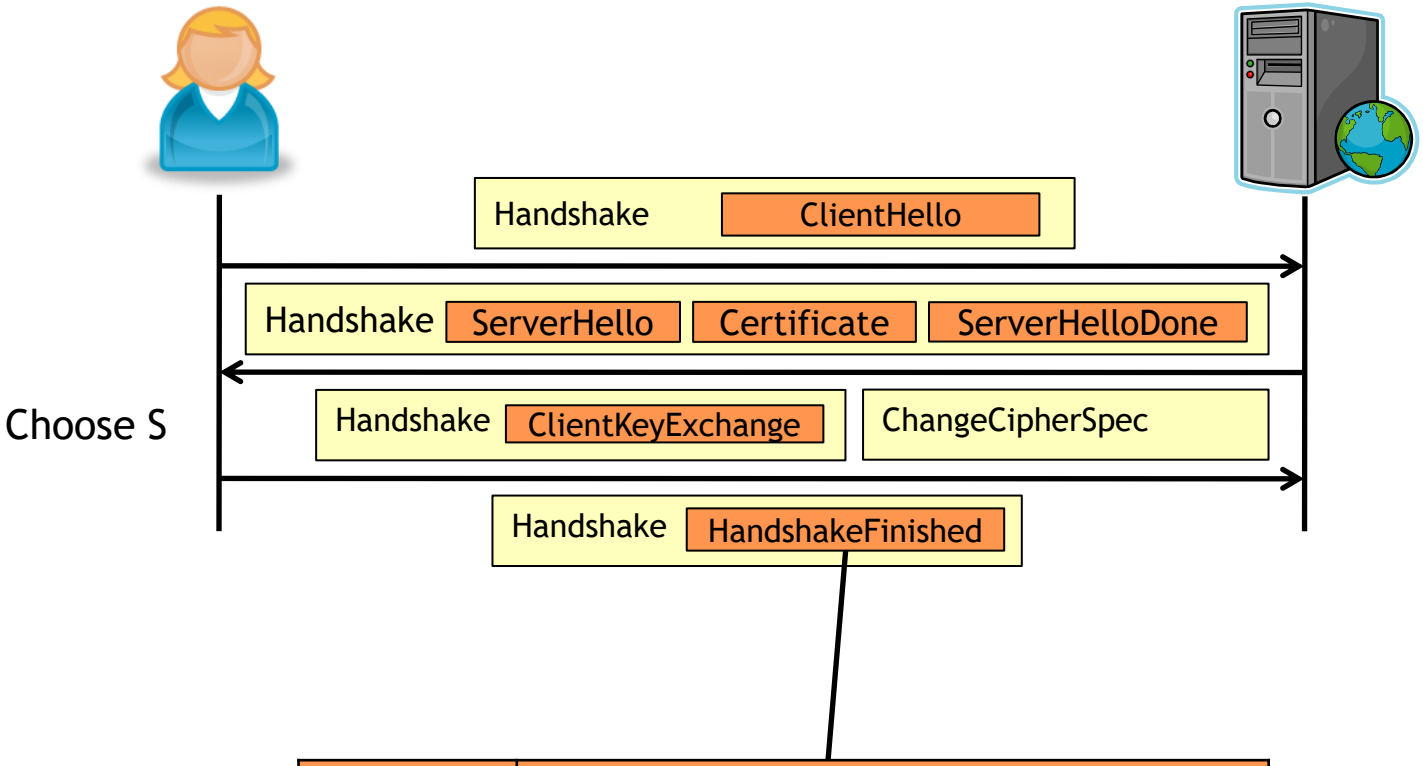
- Client → server MAC key
- Server → client MAC key
- Client → server symmetric encryption key
- Server → client symmetric encryption key
- Client → server IV
- Server → client IV

This process is called key expansion. Note that the amount of key material generated depends on the cipher suite chosen. (Why?)

Discussion

In TLS, the client provides the pre master secret, S, to the server. Why are R_A and R_S included when computing the master secret from the pre master secret?

Finishing the Handshake



Bytes	*Content*
1	Message type: HandshakeFinished
3	Length of digest
var	Keyed digest of message exchange

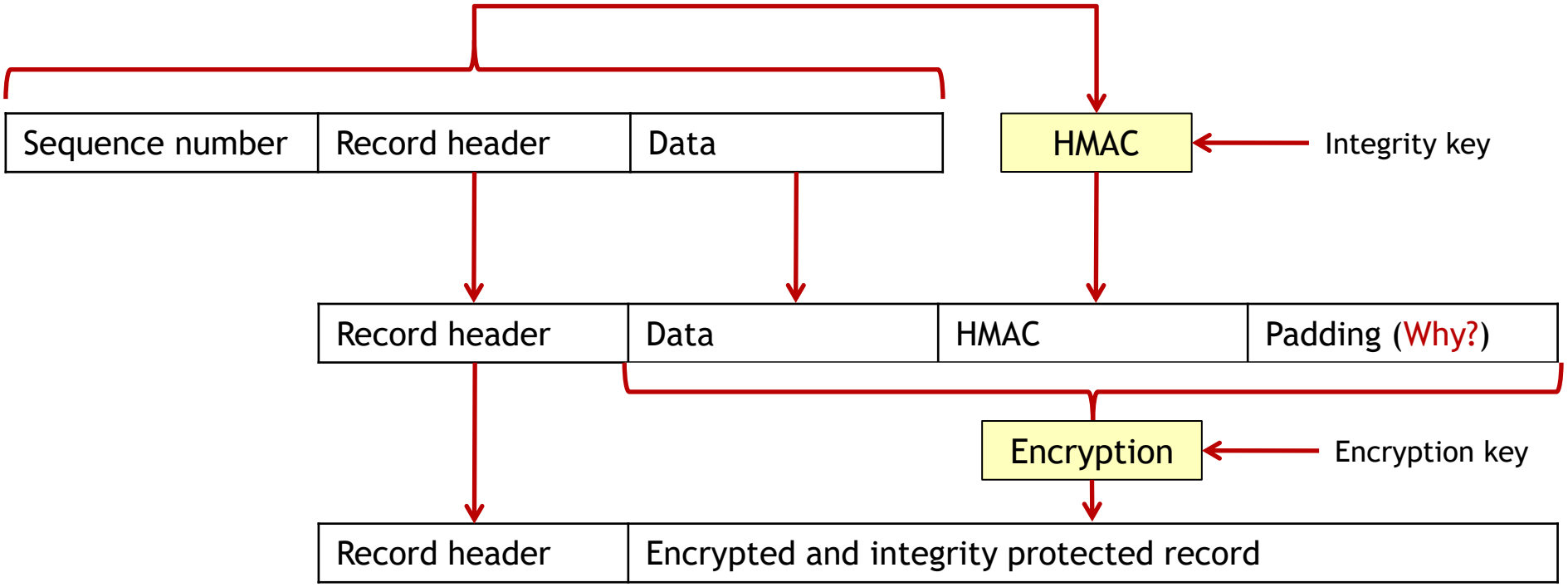
All records sent after ChangeCipherSpec are encrypted and integrity protected

All of this protection is afforded by the algorithms identified in the cipher suite chosen by the server

Example: `TLS_RSA_WITH_AES_128_CBC_SHA`

- Encryption provided using 128 bit AES in CBC mode
- Integrity protection provided by HMAC-SHA-1

Note: Data is protected one record at a time



Protocol summary

At a high level, this protocol proceeds in four phases

- Setup and parameter negotiation
- Key exchange/derivation
- Authentication
- Data transmission

For security reasons, both parties participate in (almost) all phases

- Setup: Client proposes, server chooses
- Key derivation: Randomness contributed by both parties
- Authentication: Usually, just the server is authenticated (Why?)
- Data transmission: Both parties can encrypt and integrity check

The low level details are not much more complicated than that!

This handshake procedure is fairly heavyweight

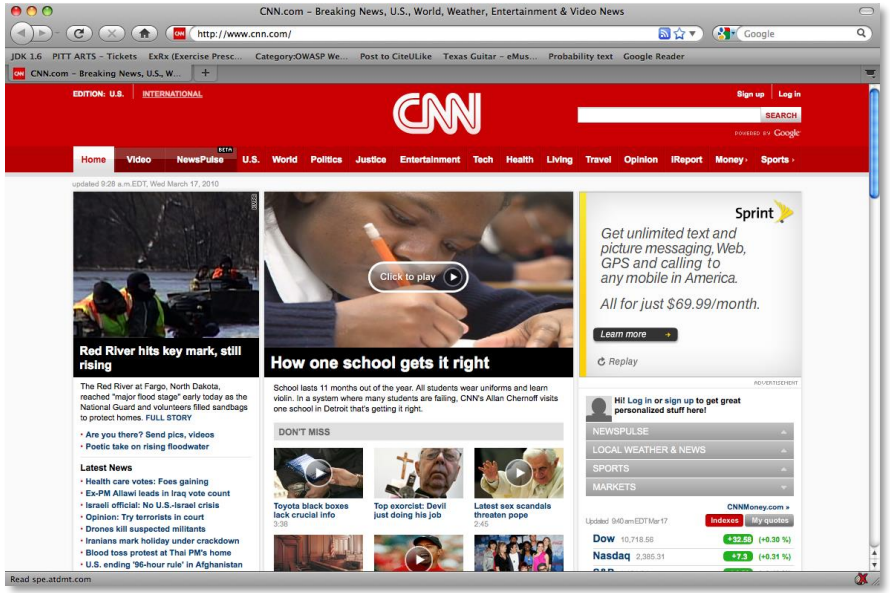
Public key cryptography is used by both parties

- Alice encrypts her pre-master secret using the server's public key
- The server decrypts this pre-master secret

So what! Aren't connections long lived?

Example: Visiting http://www.cnn.com

Visiting this single web page triggers over 130 separate HTTP connections!

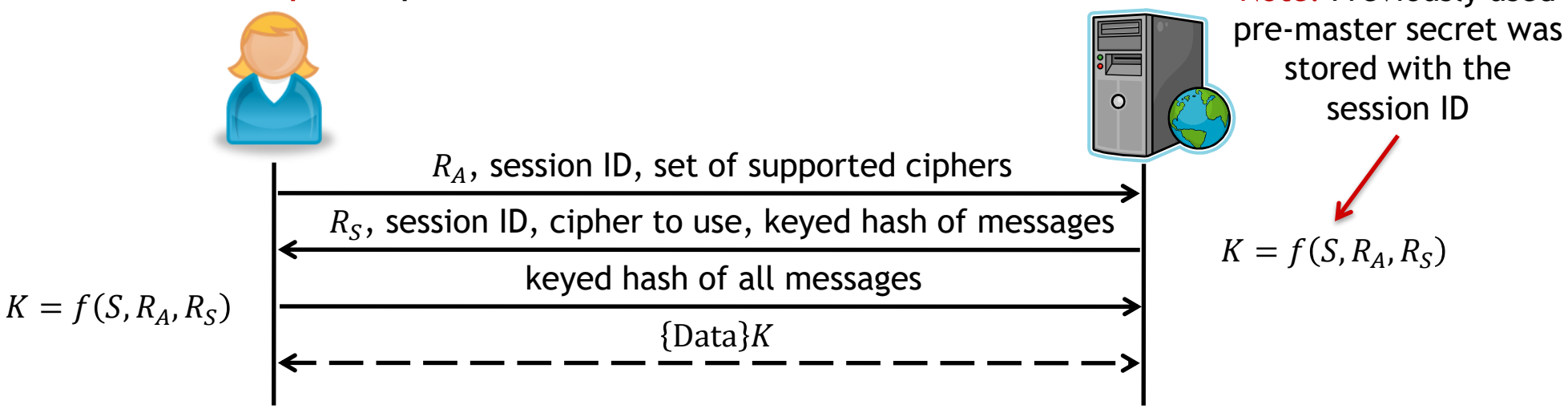


This is less than optimal.
Can we do better?

Sessions vs. Connections

In TLS, sessions are assumed to be long-lived entities that may involve many smaller connections

Connections can be spawned from an existing session using a streamlined session resumption protocol:

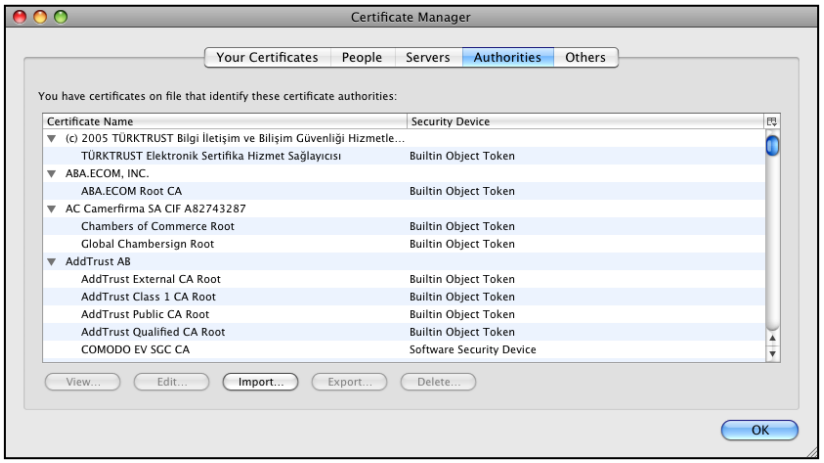


In this model, a single session could be set up with CNN and connections can be spawned as needed to retrieve content

Using HTTP/2, this concept can be taken even farther: server can respond with data for queries before they're even requested

Servers authenticate themselves using certificates. How do we authenticate the certificates?

Most TLS deployments use an oligarchy PKI model



That is, as long as the server presents a certificate chain that uses one of our trusted roots, we're happy

What about naming?

- Servers are usually known by their DNS names
- X.509 is not set up for DNS naming
- Usually the CN field of the X.509 certificate contains the DNS name

Example: C = US, ST = Washington, L = Seattle, O = Amazon.com Inc., CN = www.amazon.com

Why is this safe?

Summary of TLS

Although TCP provides a reliable data transfer protocol, it is not secure

- TCP can recover from bit-flips and dropped packets
- But malicious adversaries can alter data undetected

TLS provides cryptographic confidentiality and integrity protection for data sent over TCP

The security afforded by TLS is defined by using cipher suites

- Developers to easily incorporate new algorithms
- Security professionals can tune the level of security offered
- Breaking a cipher does not break the protocol