

# Lecture 24: Security

## Authentication, TLS/SSL

COMP 332, Spring 2024

Victoria Manfredi

W E S L E Y A N  
U N I V E R S I T Y



**Acknowledgements:** materials adapted from Computer Networking: A Top Down Approach 7<sup>th</sup> edition: ©1996-2016, J.F Kurose and K.W. Ross, All Rights Reserved as well as from slides by Abraham Matta at Boston University, and some material from Computer Networks by Tannenbaum and Wetherall.

# Today

## 1. Announcements

- Homework 8 due today at 11:59p (no coding)
- Homework 9 due Wednesday, May 8 at 11:59p (no written)

## 2. Network security

- authentication
- message integrity

## 3. Transport layer security

- overview
- toy TLS
- real TLS

# What is network security?

Goal: enable secure communication over insecure channel

## Confidentiality

- only sender, intended receiver understand message contents
  - sender **encrypts** message
  - receiver **decrypts** message
- use cryptography: public/private key vs symmetric key

## Authentication

- sender, receiver want to **confirm identity** of each other
- use nonce to confirm liveness, Certificate Authority to confirm identity

## Message integrity

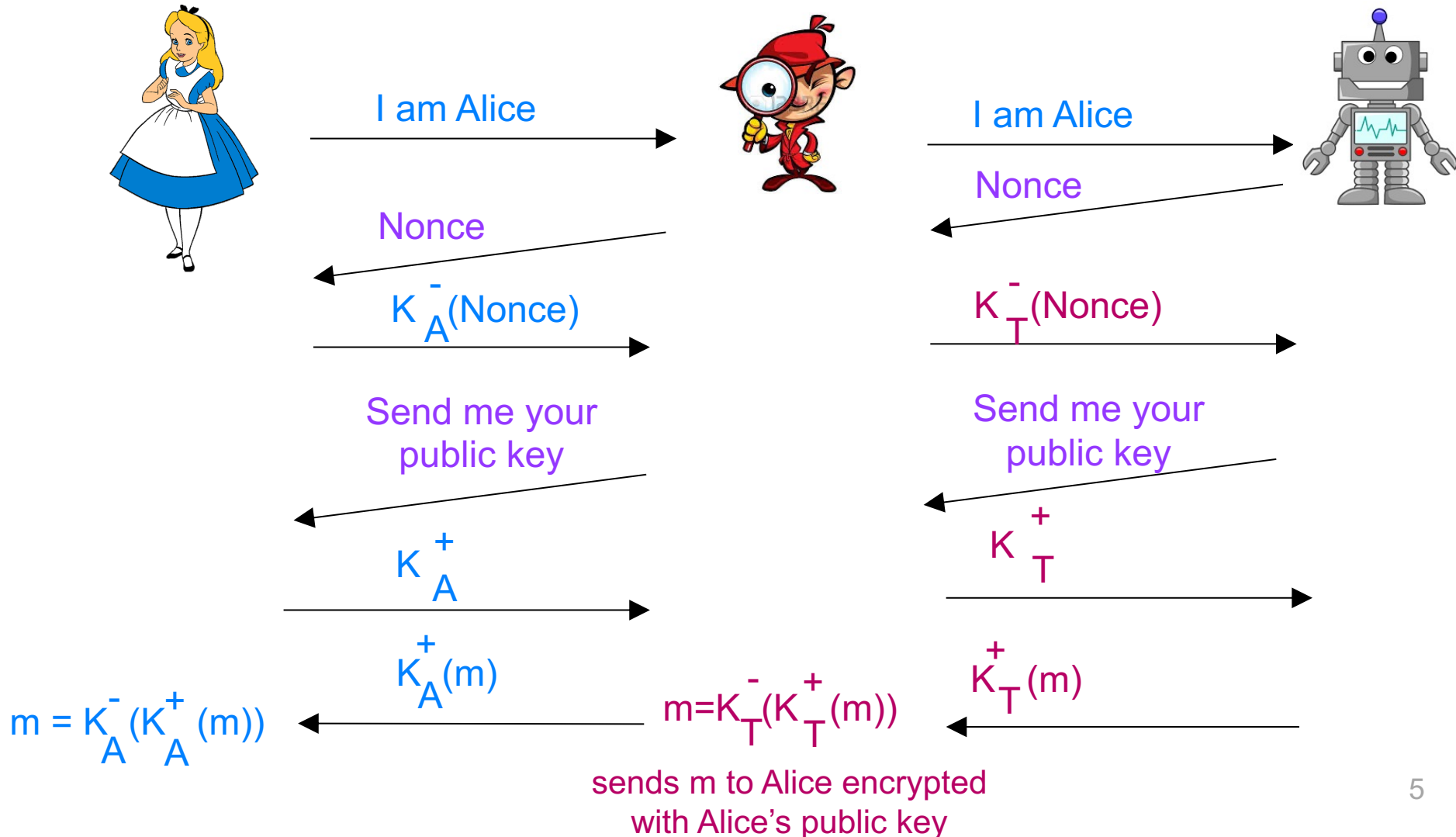
- sender, receiver want to ensure **message not altered** (in transit, or afterwards) without detection

# Network Security

# **AUTHENTICATION**

# Recall: ap5.0 man-in-the-middle attack

Trudy poses as Alice (to Bob) and as Bob (to Alice)



# Distinguishing Alice's vs. Trudy's public key

## Use certification authority (CA)

- binds public key to particular entity
  - e.g., Alice, Bob, website, ...
- 100s of certification authorities

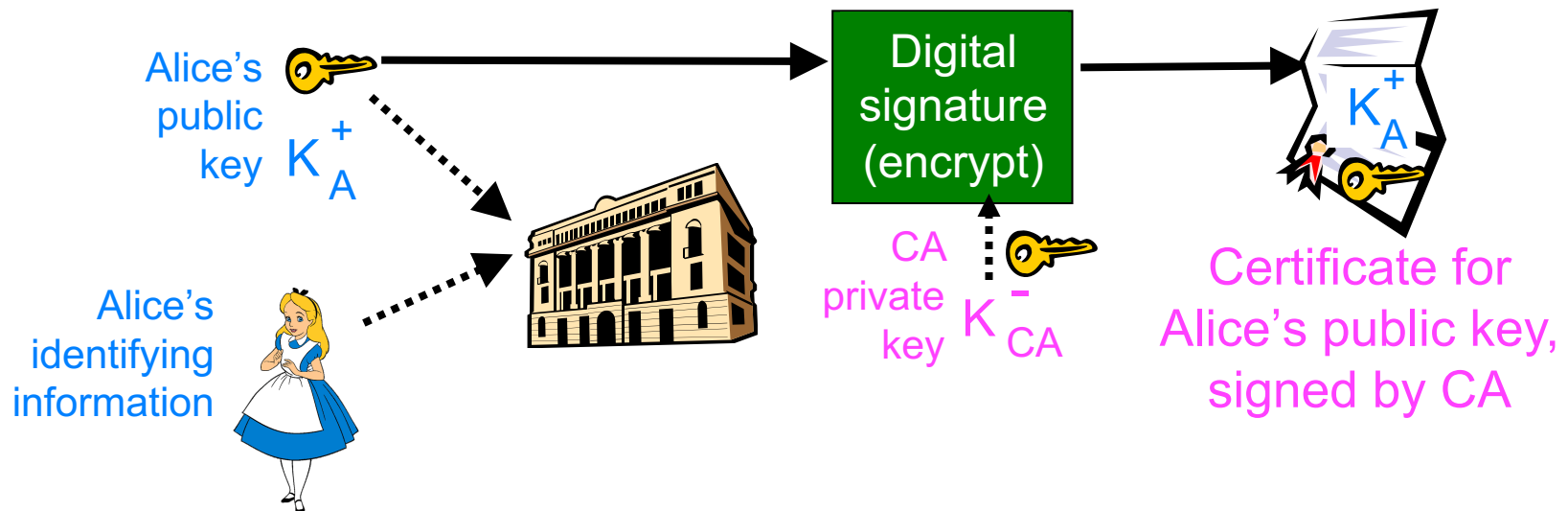
## Aside

- CAs are critical but potentially weak link ...

# How certification authorities work

## Alice registers her public key with CA

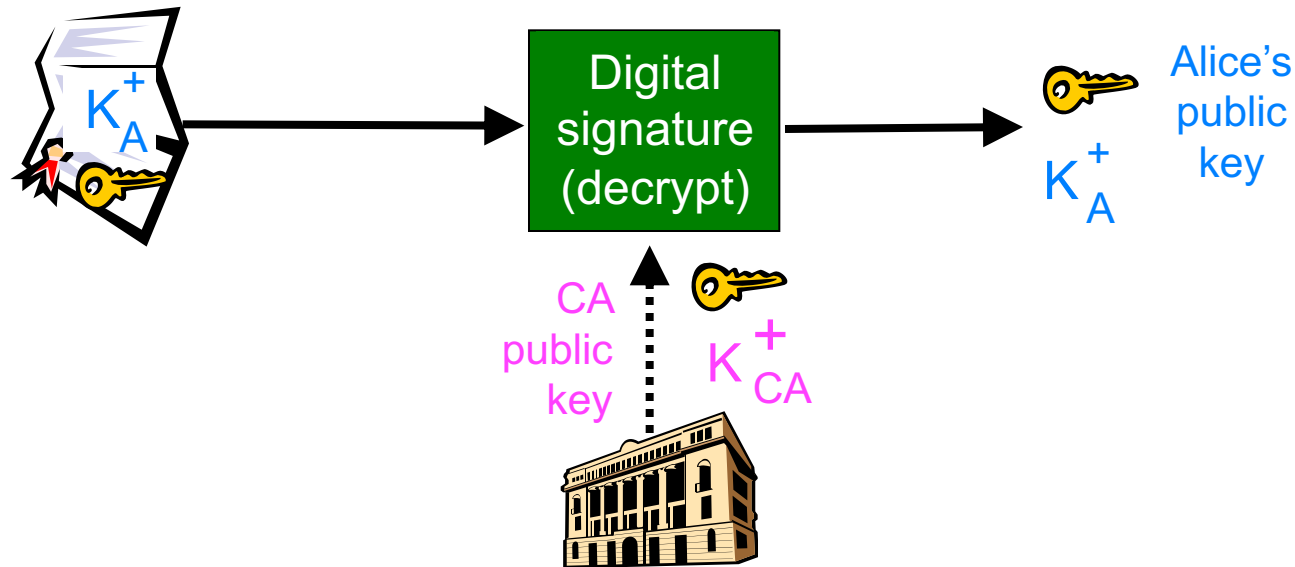
- Alice provides **proof of identity** to CA
- CA creates **certificate** binding Alice to its public key
  - certificate digitally signed by CA



# Certification authorities

## When Bob wants Alice's public key

1. gets Alice's certificate from Alice or elsewhere
2. applies CA's public key to Alice's certificate
3. gets Alice's public key






# Example

VeriSign Class 3 Public Primary Certification Authority - G5  
↳ Symantec Class 3 EV SSL CA - G3  
↳ www.bankofamerica.com

---

**www.bankofamerica.com**  
Issued by: Symantec Class 3 EV SSL CA - G3  
Expires: Thursday, July 26, 2018 at 7:59:59 PM Eastern Daylight Time  
✔ This certificate is valid

▼ **Details**

|                            |   |
|----------------------------|---|
| <b>Subject Name</b>        |   |
| <b>Inc. Country</b>        | US  |
| <b>Inc. State/Province</b> | Delaware  |
| <b>Business Category</b>   | Private Organization  |
| <b>Serial Number</b>       | 2927442   |
| <b>Country</b>             | US  |
| <b>Postal Code</b>         | 60603   |
| <b>State/Province</b>      | Illinois  |
| <b>Locality</b>            | Chicago   |
| <b>Street Address</b>      | 135 S La Salle St   |
| <b>Organization</b>        | Bank of America Corporation                                 |
| <b>Organizational Unit</b> | eComm Network Infrastructure                                |
| <b>Common Name</b>         | www.bankofamerica.com                                       |
| <b>Issuer Name</b>         |   |
| <b>Country</b>             | US  |
| <b>Organization</b>        | Symantec Corporation  |
| <b>Organizational Unit</b> | Symantec Trust Network                                      |
| <b>Common Name</b>         | Symantec Class 3 EV SSL CA - G3                             |
| <b>Serial Number</b>       | 4E 49 91 F1 B7 6A 9D 8D 16 23 5F 38 81 DD F5 E1             |
| <b>Version</b>             | 3   |
| <b>Signature Algorithm</b> | SHA-256 with RSA Encryption ( 1.2.840.113549.1.1.11 )       |
| <b>Parameters</b>          | none  |
| <b>Not Valid Before</b>    | Monday, July 24, 2017 at 8:00:00 PM Eastern Daylight Time   |
| <b>Not Valid After</b>     | Thursday, July 26, 2018 at 7:59:59 PM Eastern Daylight Time |
| <b>Public Key Info</b>     |   |
| <b>Algorithm</b>           | RSA Encryption ( 1.2.840.113549.1.1.1 )                     |

**Network Security**

**MESSAGE INTEGRITY**

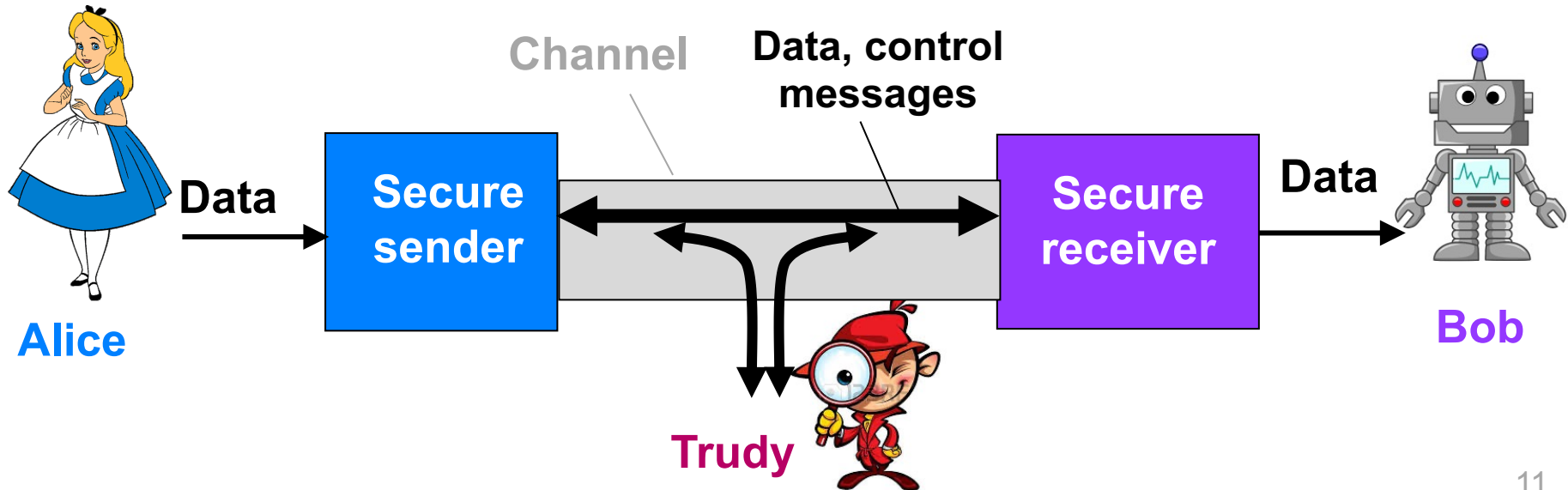
# Message integrity

Alice and Bob must be able to detect whether msg changed

1. verify msg originated from Alice
2. verify msg not tampered with on way to Bob

## Solution

- **digital signatures**: cryptographic technique like hand-written signature



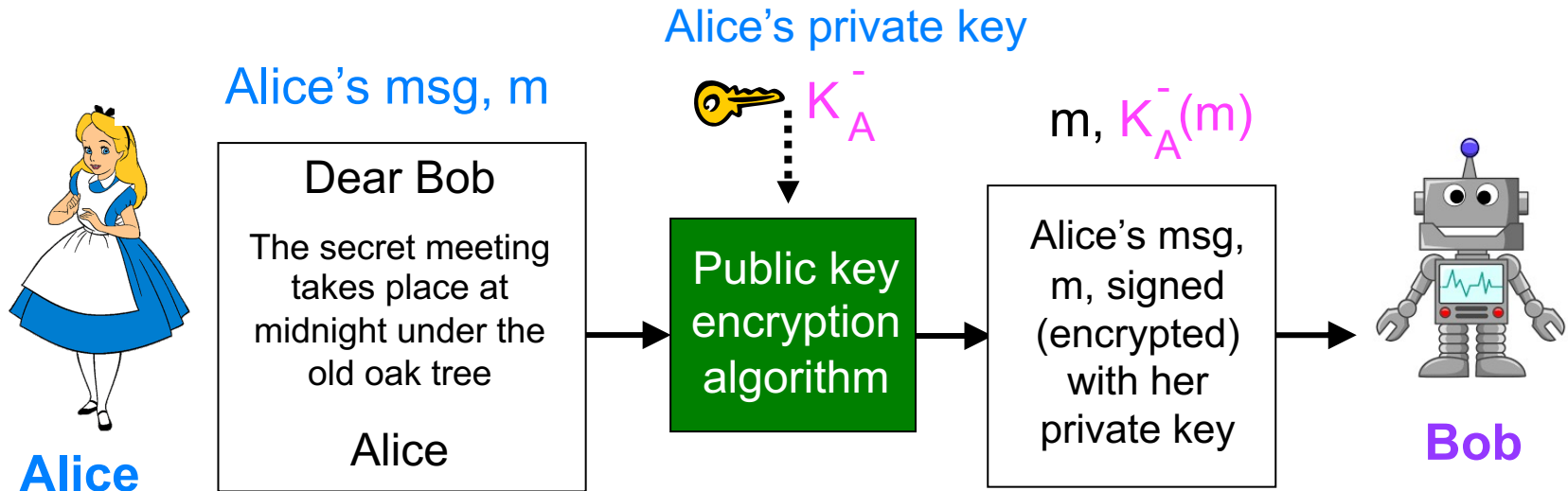
# Simple digital signature for message, m

## Sender (Alice)

- encrypts msg m with her **private key**  $K_A^-$  to create signed message,  $K_A^-(m)$
- proves she is owner/creator

## Recipient (Bob)

- applies Alice's **public key**  $K_A^+$  to  $K_A^-(m)$
- if  $K_A^+(K_A^-(m)) = m$  whoever signed m was Alice or has Alice's private key



# Problem for digital signatures

## Public key cryptography is expensive

- more expensive the longer the message is
- Why?

## Solution

- sign digital “fingerprint” of msg rather than msg itself  
Message digest

# Message digest

Desired features are what hash function gives

- fixed-length
- easy-to-compute
- 2 msgs unlikely to have same digest

Apply hash function  $H$  to  $m$



Hash function properties

- many-to-1 function
- produces fixed-size msg digest,  $H(m)$
- given message digest  $H(m)$ , computationally infeasible to find  $m'$  such that  $H(m) = H(m')$

# Some hash function standards

## MD5 hash function (RFC 1321)

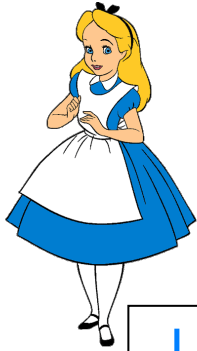
- computes 128-bit message digest in 4-step process.
- “cryptographically broken and unsuitable for further use”
  - CMU Software engineering Institute

## SHA-1

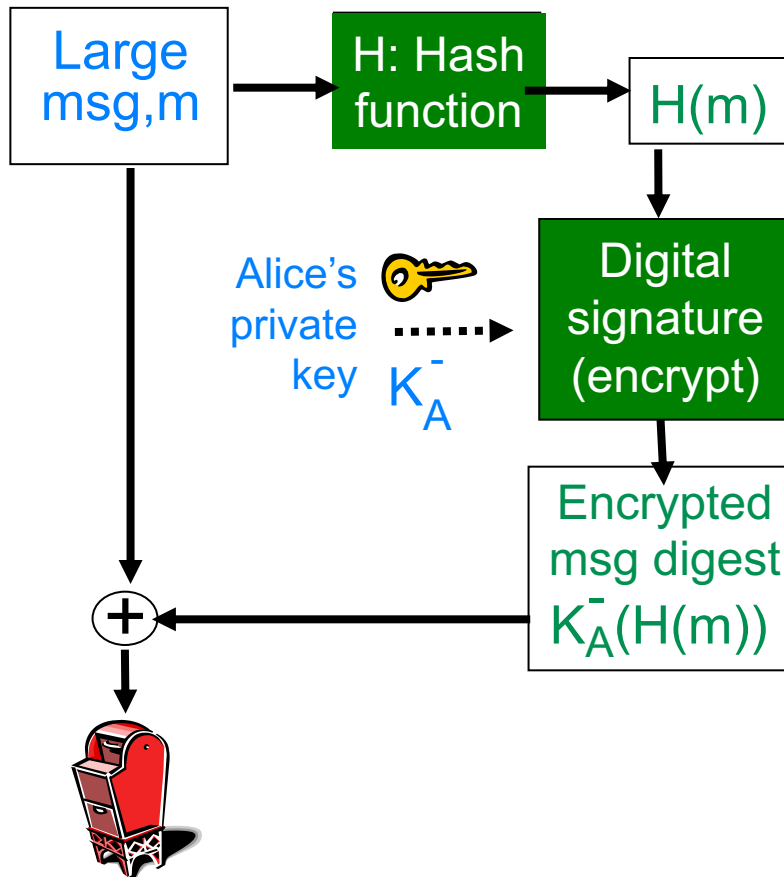
- 160-bit message digest
- many vulnerabilities, browsers will no longer use/accept

## SHA-2, SHA-3

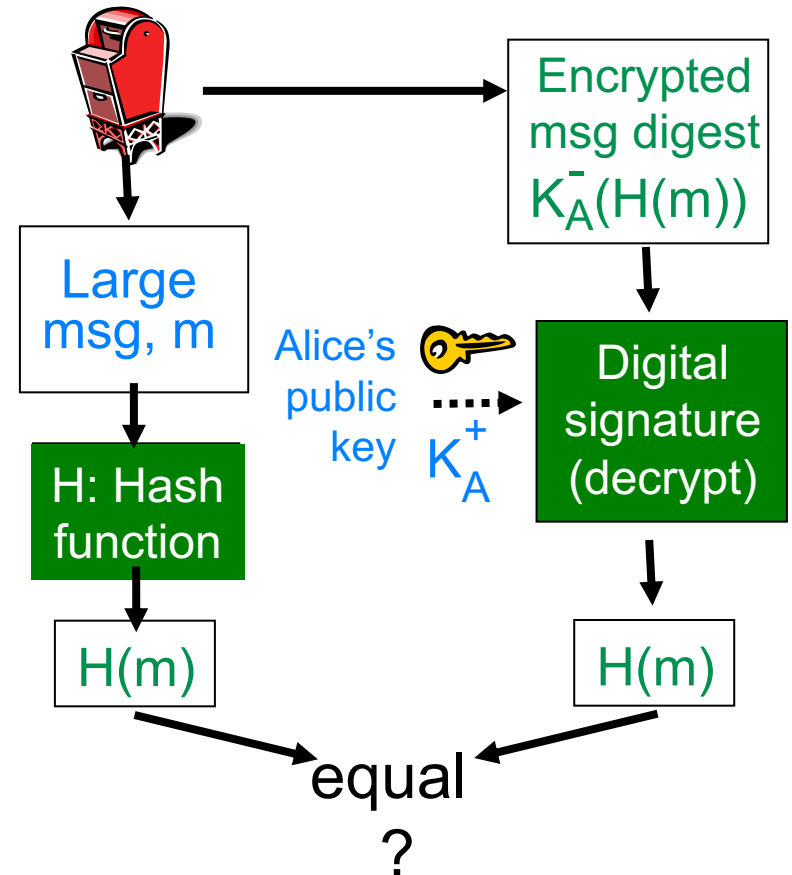
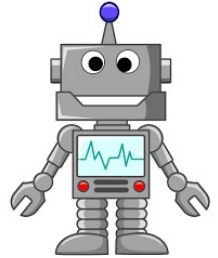
# Use signed message digest as digital signature



Alice sends digitally signed message



Bob verifies signature, integrity of digitally signed msg





# Transport Layer Security

## **OVERVIEW**

# TLS aka SSL

## Secures data at and above transport layer

- provides confidentiality, integrity, authentication
- **SSL**: Secure Sockets Layer, predecessor to TLS
- **TLS**: Transport Layer Security

## Available to all TCP applications

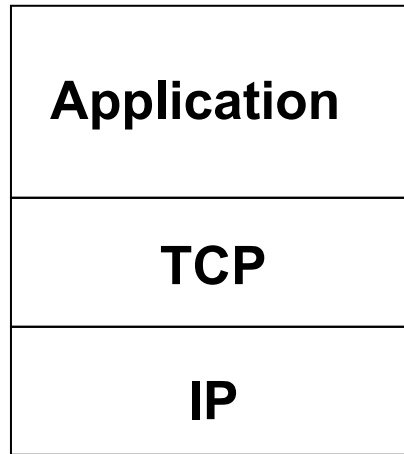
- first setup TCP connection, then run TLS as application

## Widely deployed

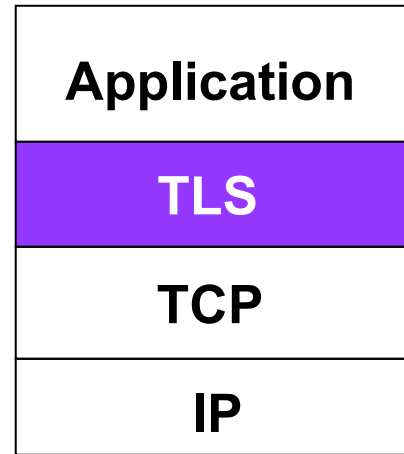
- supported by almost all browsers, web servers
- billions \$/year over SSL
- HTTP + SSL = HTTPS

# Where TLS sits in Internet stack

TLS provides application programming interface to apps



Normal application



Application with TLS

Very likely your operating system using open source library

- <https://www.openssl.org/>
- <https://firefox-source-docs.mozilla.org/security/nss/index.html>

# TLS goals

Send byte streams & interactive data

- why?

Want set of secret keys for entire connection

- why?

Want certificate exchange as part of protocol handshake phase

- why?

# Transport Layer Security

## **TOY TLS**

# A simple secure channel

## Handshake

- Alice and Bob use their certificates, private keys to authenticate each other and exchange shared secret

## Key derivation

- Alice and Bob use shared secret to derive set of keys

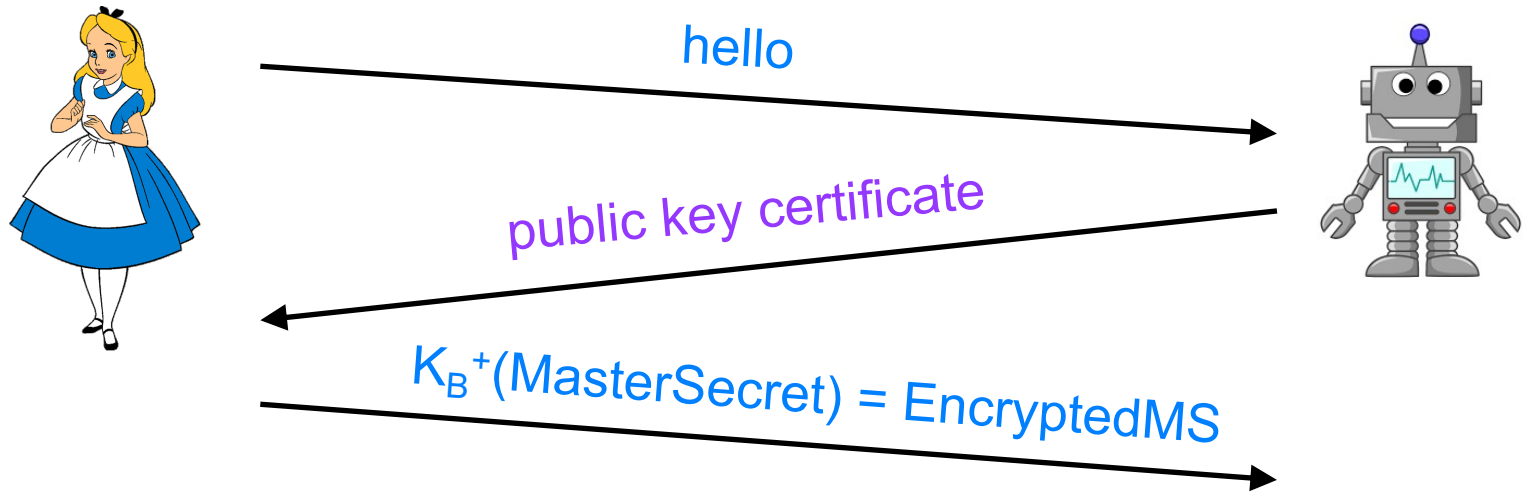
## Data transfer

- data to be transferred is broken up into series of records

## Connection closure

- special messages to securely close connection

# A simple handshake



## Derive keys from master secret

- use key derivation function (KDF)
  - takes master secret and additional random data and creates keys

# Key derivation

## Don't use same key for more than one cryptographic operation

- keys for message authentication code (MAC): like hash
- keys for encryption

## Encryption keys

- $K_c$  = encryption key for data sent from client to server
- $K_s$  = encryption key for data sent from server to client

## MAC keys

- $M_c$  = MAC key for data sent from client to server
- $M_s$  = MAC key for data sent from server to client



# Data records

## Why not encrypt data in constant stream as we write it to TCP?

- where to put MAC?
  - if at end, no message integrity until all data processed
- e.g., instant messaging
  - how can we do integrity check over all bytes sent before displaying?

## Solution: break stream in series of records

- each record carries MAC
- receiver can act on each record as it arrives



# More attacks

What if attacker replays or re-orders records?

- Solution: put sequence # into MAC (no seq # field)
- $MAC = MAC(M_x, \text{sequence} || \text{data})$

What if attacker replays all records?

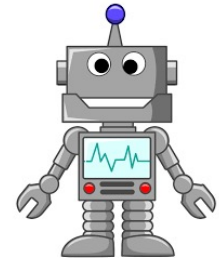
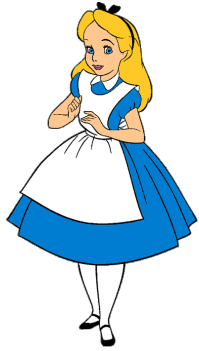
- Solution: use nonce

What if attacker forges TCP connection close?

- Solution: have record types, with one type for closure
  - type 0 for data
  - type 1 for closure
- $MAC = MAC(M_x, \text{sequence} || \text{type} || \text{data})$

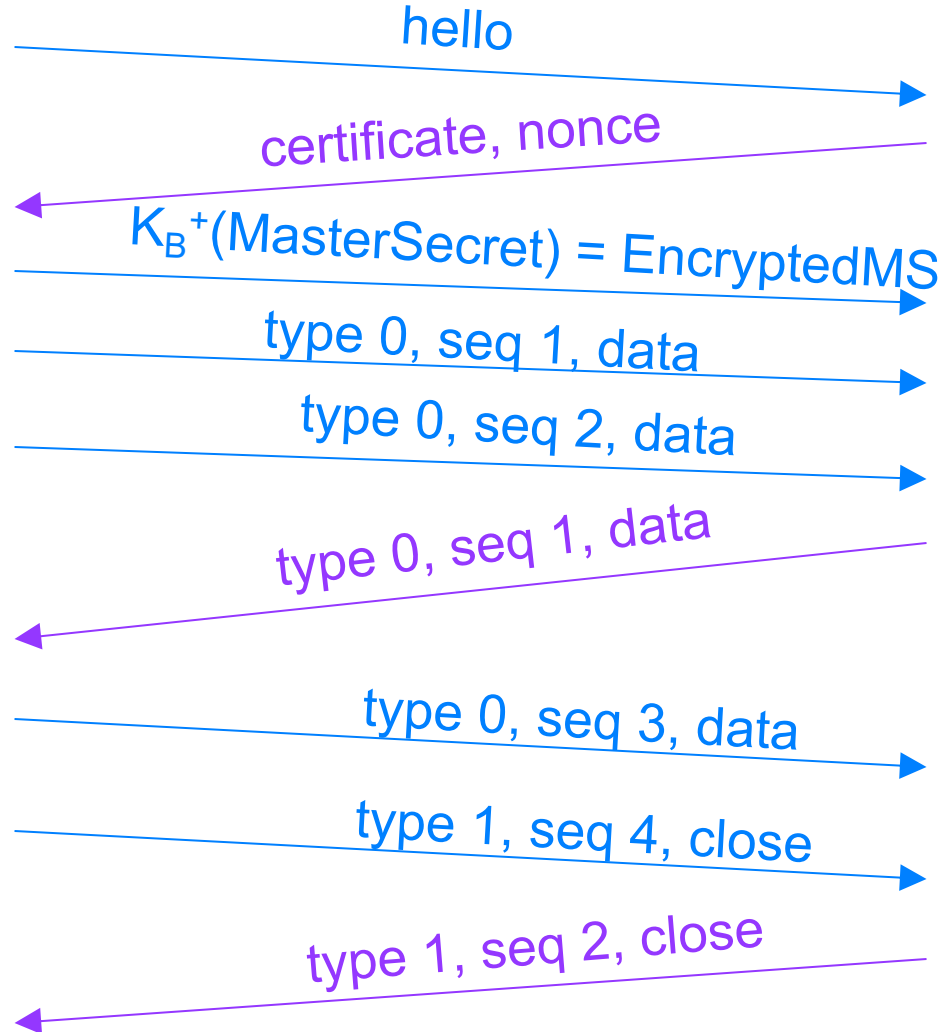


# Summary



bob.com

Encrypted



# Transport Layer Security

## **REAL TLS**

# Toy TLS is incomplete

How long are fields? Which encryption protocols? How do client and server negotiate encryption algorithms?

## TLS Handshake

### – confidentiality

- client and server negotiate encryption algorithms before data transfer
  - i.e., negotiate ciphersuite
- derive keys used in data exchange

### – integrity

- check if handshake tampered with based on hash of handshake msgs

### – authentication

- using public key and server's certificate
- optional client authentication

# TLS cipher suite

Negotiation: client, server agree on cipher suite

- client offers choice server picks one

TLS\_RSA\_WITH\_3DES\_EDE\_CBC\_SHA

Key exchange  
algorithm: public-  
key

Symmetric encryption  
algorithm: block cipher to  
encrypt msg stream

MAC  
algorithm

Which ciphersuites are supported depends on TLS version

- TLS 1.2 supports many cipher suites
- TLS 1.3 supports many fewer cipher suites

# Cipher suites

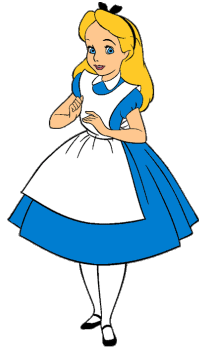
- ▼ TLSv1 Record Layer: Handshake Protocol: Client Hello
  - Content Type: Handshake (22)
  - Version: TLS 1.0 (0x0301)
  - Length: 144
- ▼ Handshake Protocol: Client Hello
  - Handshake Type: Client Hello (1)
  - Length: 140
  - Version: TLS 1.0 (0x0301)
  - ▶ Random: 5ae5dac626d5483a3ea908c593979d44170f3e628f26688d...
  - Session ID Length: 32
  - Session ID: e84d0000076240b35c57828829153be712af150acb327e17...
  - Cipher Suites Length: 32
  - ▼ Cipher Suites (16 suites)
    - Cipher Suite: TLS\_EMPTY\_RENEGOTIATION\_INFO\_SCSV (0x00ff)
    - Cipher Suite: TLS\_ECDHE\_ECDSA\_WITH\_AES\_256\_CBC\_SHA384 (0xc024)
    - Cipher Suite: TLS\_ECDHE\_ECDSA\_WITH\_AES\_128\_CBC\_SHA256 (0xc023)
    - Cipher Suite: TLS\_ECDHE\_ECDSA\_WITH\_AES\_256\_CBC\_SHA (0xc00a)
    - Cipher Suite: TLS\_ECDHE\_ECDSA\_WITH\_AES\_128\_CBC\_SHA (0xc009)
    - Cipher Suite: TLS\_ECDHE\_ECDSA\_WITH\_3DES\_EDE\_CBC\_SHA (0xc008)
    - Cipher Suite: TLS\_ECDHE\_RSA\_WITH\_AES\_256\_CBC\_SHA384 (0xc028)
    - Cipher Suite: TLS\_ECDHE\_RSA\_WITH\_AES\_128\_CBC\_SHA256 (0xc027)
    - Cipher Suite: TLS\_ECDHE\_RSA\_WITH\_AES\_256\_CBC\_SHA (0xc014)
    - Cipher Suite: TLS\_ECDHE\_RSA\_WITH\_AES\_128\_CBC\_SHA (0xc013)
    - Cipher Suite: TLS\_ECDHE\_RSA\_WITH\_3DES\_EDE\_CBC\_SHA (0xc012)
    - Cipher Suite: TLS\_RSA\_WITH\_AES\_256\_CBC\_SHA256 (0x003d)
    - Cipher Suite: TLS\_RSA\_WITH\_AES\_128\_CBC\_SHA256 (0x003c)
    - Cipher Suite: TLS\_RSA\_WITH\_AES\_256\_CBC\_SHA (0x0035)
    - Cipher Suite: TLS\_RSA\_WITH\_AES\_128\_CBC\_SHA (0x002f)
    - Cipher Suite: TLS\_RSA\_WITH\_3DES\_EDE\_CBC\_SHA (0x000a)

# TLS Client Hello

- ▶ Frame 50: 203 bytes on wire (1624 bits), 203 bytes captured (1624 bits) on interface 0
- ▶ Ethernet II, Src: Apple\_73:43:26 (78:4f:43:73:43:26), Dst: JuniperN\_1e:18:01 (3c:8a:b0:1e:18:01)
- ▶ Internet Protocol Version 4, Src: vmanfredismbp2.wireless.wesleyan.edu (129.133.187.174), Dst: 129.133.187.174
- ▶ Transmission Control Protocol, Src Port: 63173, Dst Port: 443, Seq: 41885059, Ack: 3555367379, Len: 144
- ▼ Secure Sockets Layer
  - ▼ TLSv1 Record Layer: Handshake Protocol: Client Hello
    - Content Type: Handshake (22)
    - Version: TLS 1.0 (0x0301)
    - Length: 144
  - ▼ Handshake Protocol: Client Hello
    - Handshake Type: Client Hello (1)
    - Length: 140
    - Version: TLS 1.0 (0x0301)
    - ▶ Random: 5ae5dac626d5483a3ea908c593979d44170f3e628f26688d...
    - Session ID Length: 32
    - Session ID: e84d0000076240b35c57828829153be712af150acb327e17...
    - Cipher Suites Length: 32
    - ▶ Cipher Suites (16 suites)
    - Compression Methods Length: 1
    - ▶ Compression Methods (1 method)
    - Extensions Length: 35
    - ▶ Extension: supported\_groups (len=8)
    - ▶ Extension: ec\_point\_formats (len=2)
    - ▶ Extension: status\_request (len=5)
    - ▶ Extension: signed\_certificate\_timestamp (len=0)
    - ▶ Extension: extended\_master\_secret (len=0)



# TLS handshake



**Alice**

**1. Client hello →**

client nonce, ciphersuites

**3. Verifies certificate**

generates premaster secret

**4. Premaster secret →**

encrypted with Bob's public key  
from certificate

**6. Generate symmetric keys**

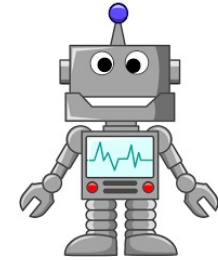
client nonce, server nonce,  
premaster, ciphersuite

**8. Client hello done →**

MAC of all handshake msgs  
encrypted with client symmetric key

**7. Encrypted data →**

**Bob**



**← 2. Server hello**

server nonce, chosen  
ciphersuite, RSA certificate

**5. Generate symmetric keys**

client nonce, server nonce,  
premaster, ciphersuite

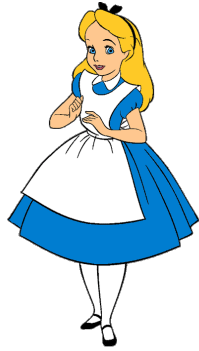
**← 7. Server hello done**

MAC of all handshake msgs  
encrypted with server session keys

**← 8. Encrypted data**

**Protect handshake from tampering**

# Why 2 random nonces?

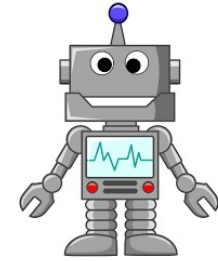


Alice

1. Client hello →

client nonce, ciphersuites

Bob



← 2. Server hello

server nonce, chosen  
ciphersuite, RSA certificate

Suppose Trudy sniffs all messages between Alice & Bob

- next day, Trudy sets up TCP connection with Bob
  - replays sequence of records
  - Bob (Amazon) thinks Alice made two separate orders for same thing

## Solution

- Bob sends different random nonce for each connection
  - causes encryption keys to be different on the 2 days
  - Trudy's messages will fail Bob's integrity check

# Key derivation

## Client nonce, server nonce, pre-master secret

- input into pseudo random-number generator to get master secret

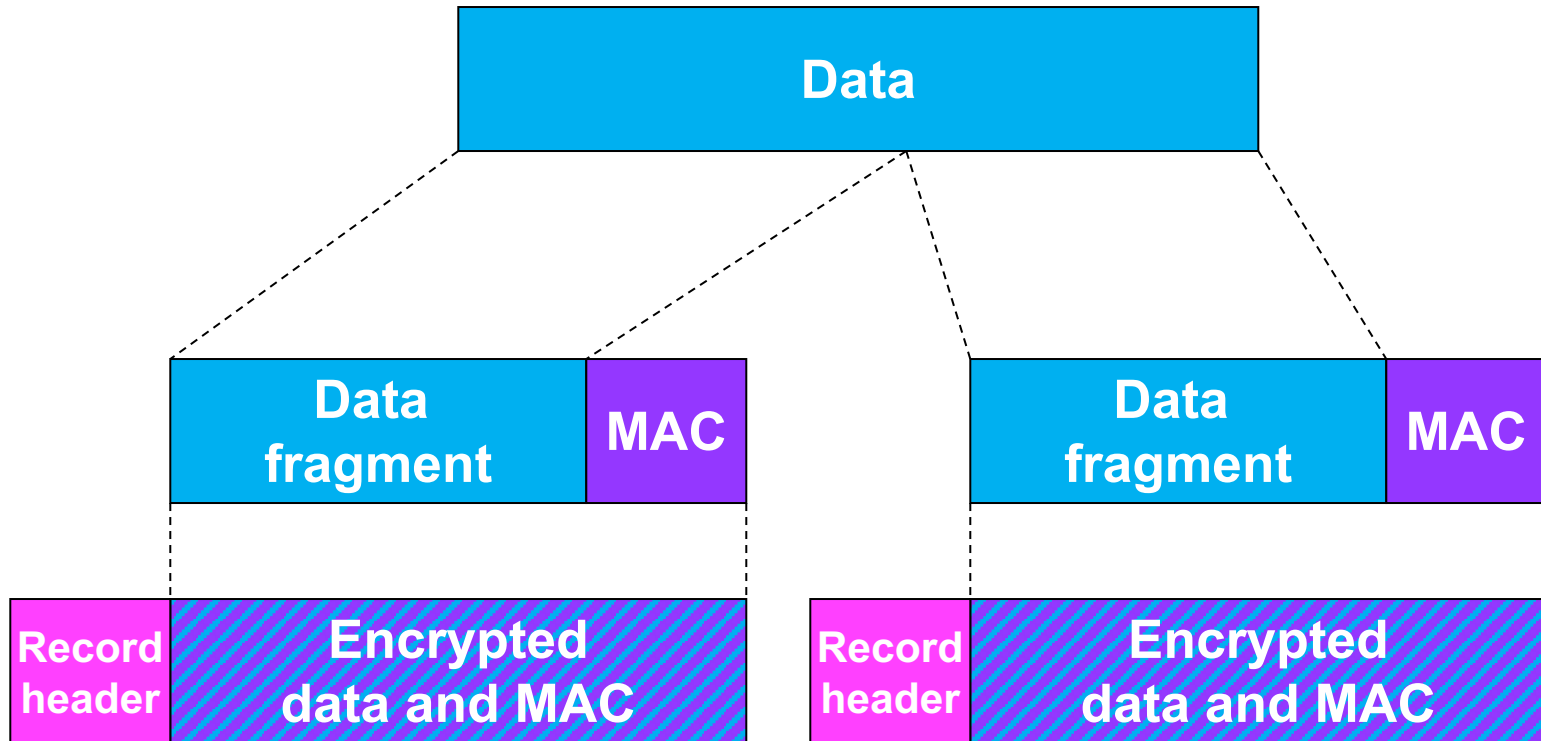
## Master secret, new nonces

- input into another random-number generator to get key block

## Key block sliced and diced

- client MAC key
- server MAC key
- client encryption key
- server encryption key
- client initialization vector (IV)
- server initialization vector (IV)

# SSL record protocol



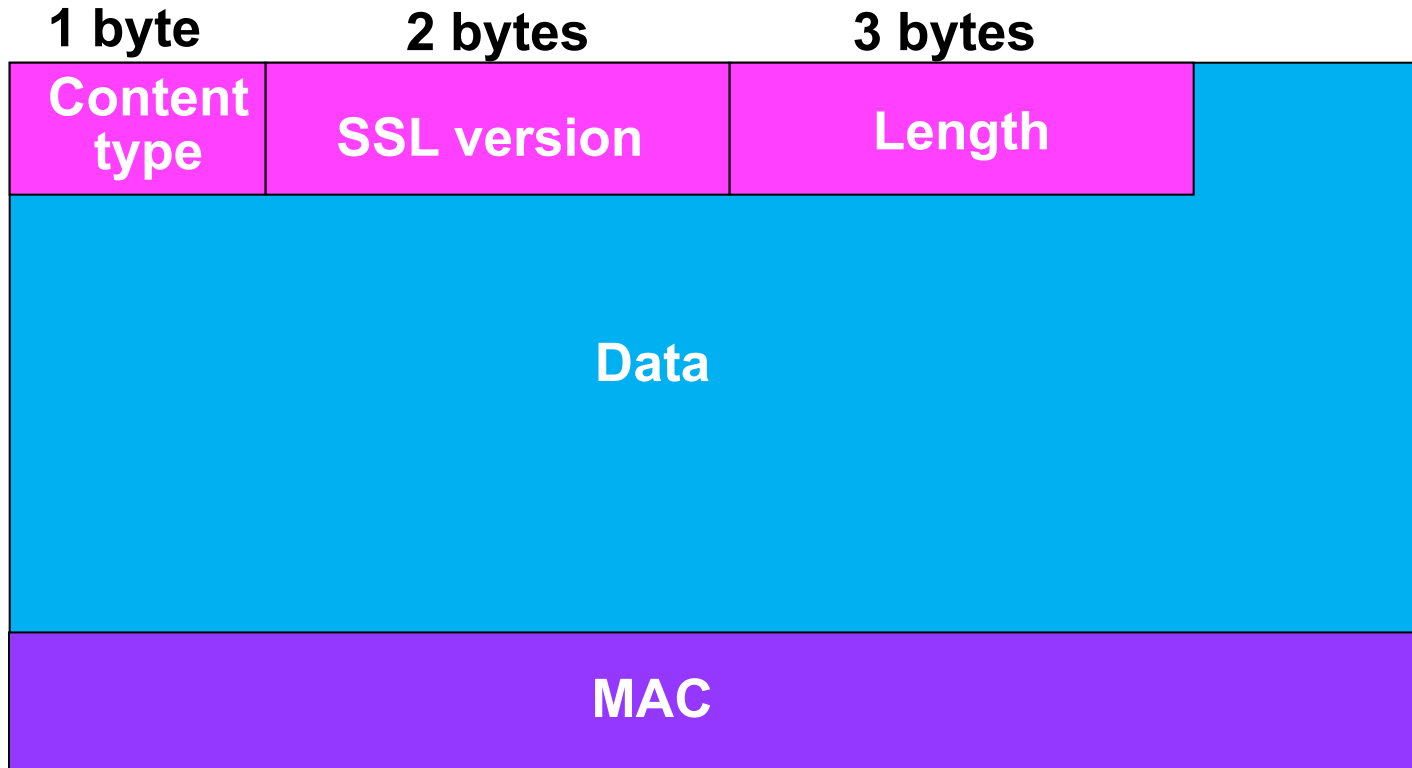
**Record header:** content type; version; length

**MAC:** includes sequence number, MAC key  $M_x$

**Fragment:** each SSL fragment  $2^{14}$  bytes (~16 Kbytes)

**These records are pushed into TCP socket**

# SSL record format



Data and MAC encrypted (symmetric algorithm)

# Wireshark

Look at TLS traffic and openssl s\_client traffic

# Openssl s\_client

```
> echo -e "GET / HTTP/1.1\r\nHost: www.wesleyan.edu\r\n\r\n" | openssl s_client -ign_eof -connect www.wesleyan.edu:443
CONNECTED(00000003)
depth=3 C = SE, O = AddTrust AB, OU = AddTrust External TTP Network, CN = AddTrust External CA Root
verify return:1
depth=2 C = US, ST = New Jersey, L = Jersey City, O = The USERTRUST Network, CN = USERTrust RSA Certification Authority
verify return:1
depth=1 C = US, ST = MI, L = Ann Arbor, O = Internet2, OU = InCommon, CN = InCommon RSA Server CA
verify return:1
depth=0 C = US, postalCode = 06457, ST = CT, L = Middletown, street = 237 High Street, O = Wesleyan University, OU = ITS, CN = www.wesleyan.edu
verify return:1
---
Certificate chain
 0 s:/C=US/postalCode=06457/ST=CT/L=Middletown/street=237 High Street/O=Wesleyan University/OU=ITS/CN=www.wesleyan.edu
   i:/C=US/ST=MI/L=Ann Arbor/O=Internet2/OU=InCommon/CN=InCommon RSA Server CA
 1 s:/C=SE/O=AddTrust AB/OU=AddTrust External TTP Network/CN=AddTrust External CA Root
   i:/C=SE/O=AddTrust AB/OU=AddTrust External TTP Network/CN=AddTrust External CA Root
 2 s:/C=US/ST=New Jersey/L=Jersey City/O=The USERTRUST Network/CN=USERTrust RSA Certification Authority
   i:/C=SE/O=AddTrust AB/OU=AddTrust External TTP Network/CN=AddTrust External CA Root
 3 s:/C=US/ST=MI/L=Ann Arbor/O=Internet2/OU=InCommon/CN=InCommon RSA Server CA
   i:/C=US/ST=New Jersey/L=Jersey City/O=The USERTRUST Network/CN=USERTrust RSA Certification Authority
---
Server certificate
-----BEGIN CERTIFICATE-----
MIITVTCGCCD2eAwTRALC1LD7nn01zDSPTePKviuQwDOY1KeZThycNAQELR0Aw
```