Key Exchange Protocols

Key Exchange Protocols

- Why do we need KE protocols?
  - A way to establish secrets
  - Even if we have pre-established secrets, we want to use refresh "keys" per session

- Remark on (Perfect) Forward Secrecy
  - Protect Encrypted information, even if long-term key (for client and server) is compromised
  - Idea: Generate session keys, and throw away messages used to generate sessions keys…

Key Exchange: NS protocol

- Needham Schroder [1977] – Kerberos

(C)

(S)

Entity Identifier

$$\{Nc, C\}_{Ks} \longrightarrow$$

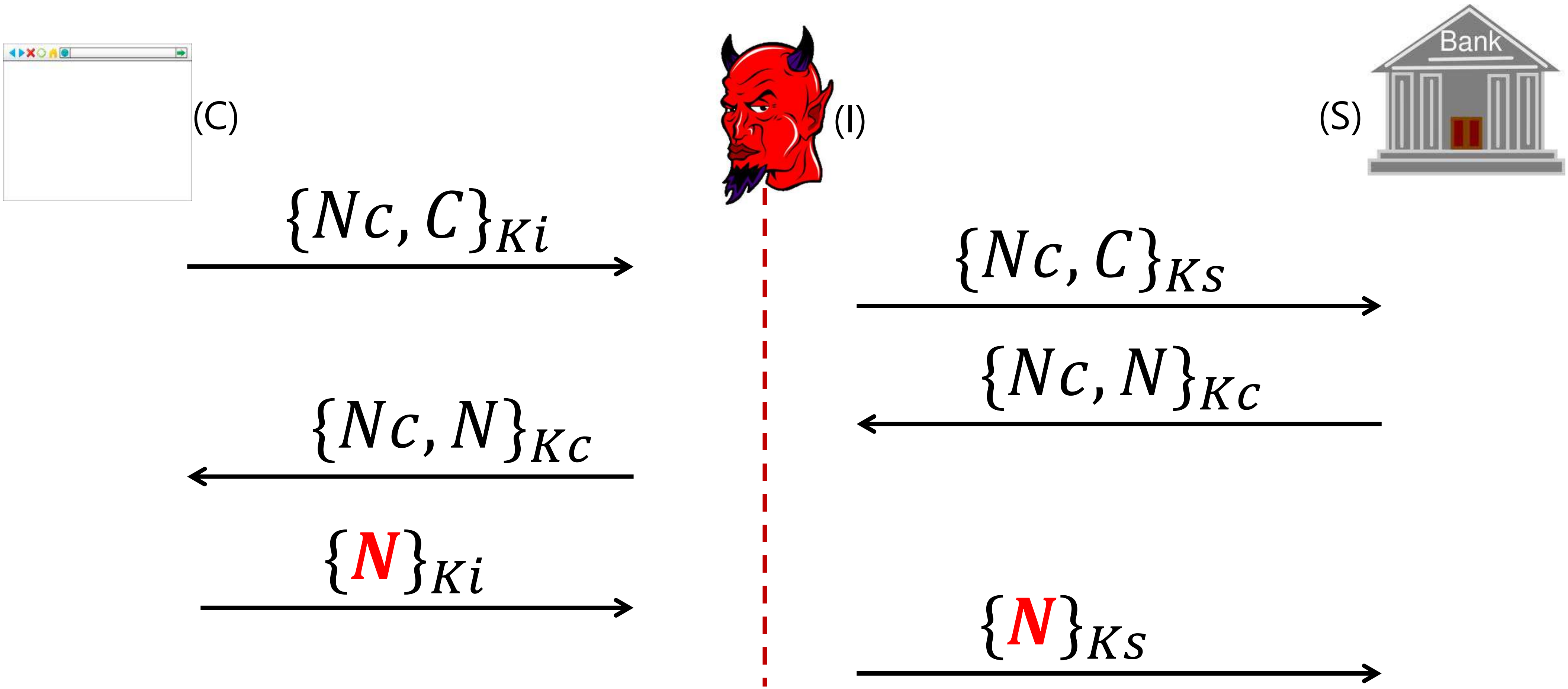
Encryption key

$$\longleftarrow \{Nc, Ns\}_{Kc}$$

$$\{Ns\}_{Ks} \longrightarrow$$

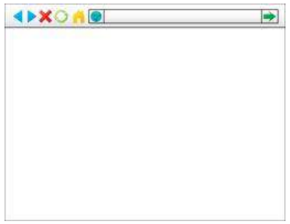
- Ks and Kc are pre-established secrets between (C,S)
- Think about these 2 goals:
  – Is (Nc, Ns) a "good shared key"? [Key Secrecy]
  – Should A believe it's talking to B? [Entity Authentication]
  – Should B be sure if it's talking to A?

Lowe's Attack on NS protocol [1995]



- *Attacker knows **N**.*
- C thinks its talking to I, but *S thinks its talking to C*

Key Exchange:
Diffie Hellman [1976]



Pick fresh a

g^a (mod p) $\longrightarrow$

Pick fresh b

$\longleftarrow$ g^b (mod p)

$\boxed{K = g^{ba} \text{ (mod p)}}$

$\boxed{K = g^{ab} \text{ (mod p)}}$

- Is it secure (authenticated) key-exchange?
- Assuming: ***CDH*** *is hard,* **DLOG** is hard

Diffie Hellman Problem
(Given g, g^a, g^b. Find: g^ab)

Discrete Log Problem
(Given g, g^a. Find: a)

Analysis of DHKE protocol

- Think about these 2 goals:
  - Is K a "good key" to talk with B?   [Key Secrecy]
  - Should A be sure it's talking to B? [Entity Authentication]

a new value of a here can be chosen whenever want to refresh the secret key

Pick fresh a $\xrightarrow{g^a \text{ (mod p)}}$

$\xleftarrow{g^b \text{ (mod p)}}$ Pick fresh b

K = g^{ba} (mod p)

K = g^{ab} (mod p)

Analysis of DHKE protocol

- DH does <u>not</u> achieve entity authentication

Pick fresh a

g^a →

→ g^{r1}

← g^b

Pick fresh b

← g^{r2}

K1 = g^{r2*a} (mod p)

K2 = g^{r1*b} (mod p)

Adding Signatures to DHE: Station-to-Station (STS) Protocol

Pick fresh a

$M1 = g^a$ →

Pick fresh b

K = g^{ab} (mod p)

$M2 = g^b, \{\boldsymbol{Sig_B(M2, M1)}\}_K$ ←

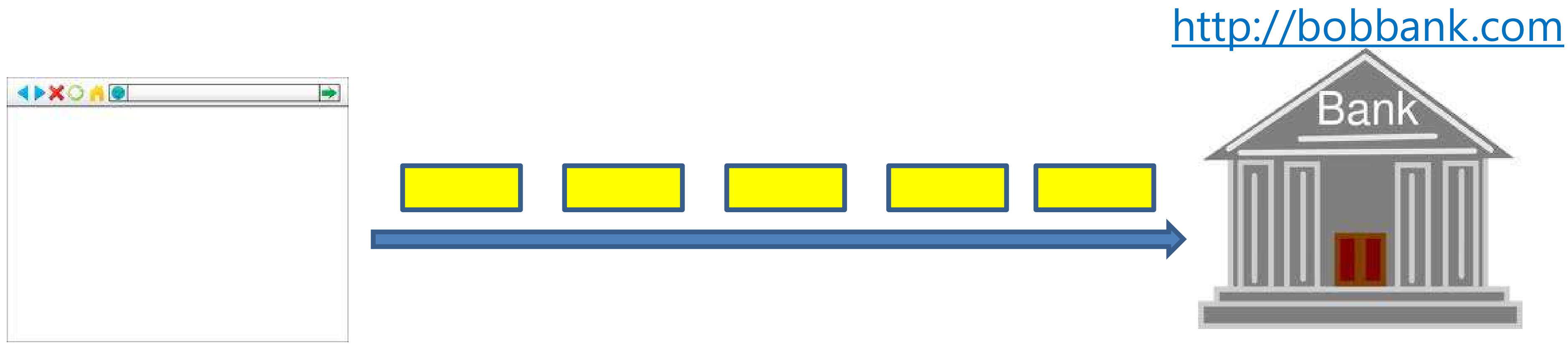
K = g^{ba} (mod p)

$\{\boldsymbol{Sig_A(M1, M2)}\}_K$ →

- Is this protocol secure?
  - Is K a "good key" to talk with B? [Key Secrecy]
  - Should B be sure it's talking to A? [Entity Authentication]
  - Should A be sure it's talking to B? [Entity Authentication]

Security Properties: A Hierarchy of Authentication Specifications [Lowe'97]

We are ready for Secure Channels...

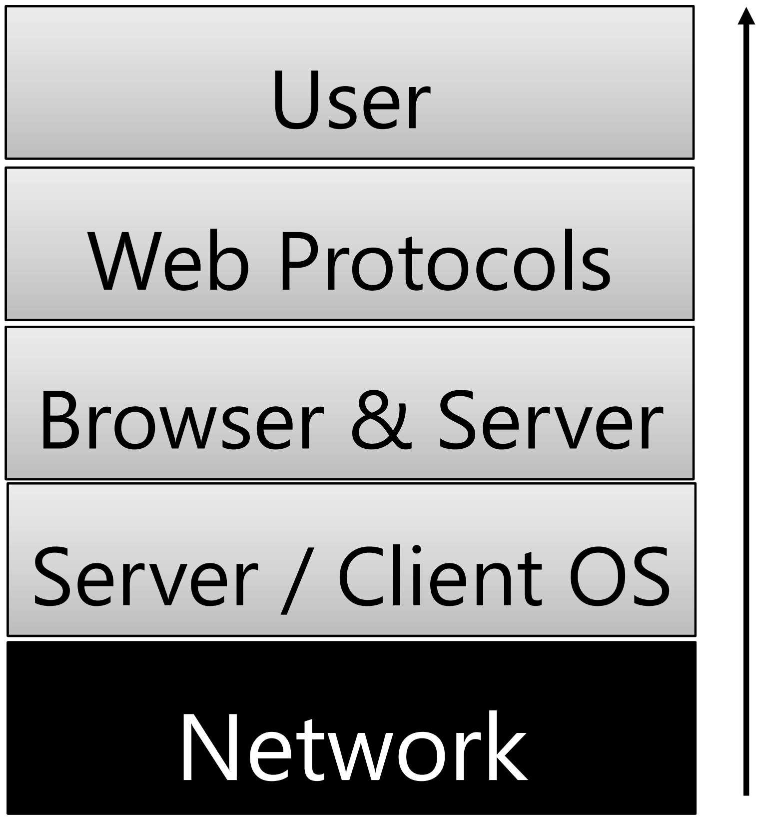


A Secure Channel establishes **between 2 programs**,
a data channel that has:

- **Confidentiality**
- **Integrity**
- **Authentication**

against a computationally-bounded “network attacker”
[Dolev-Yao-1983]

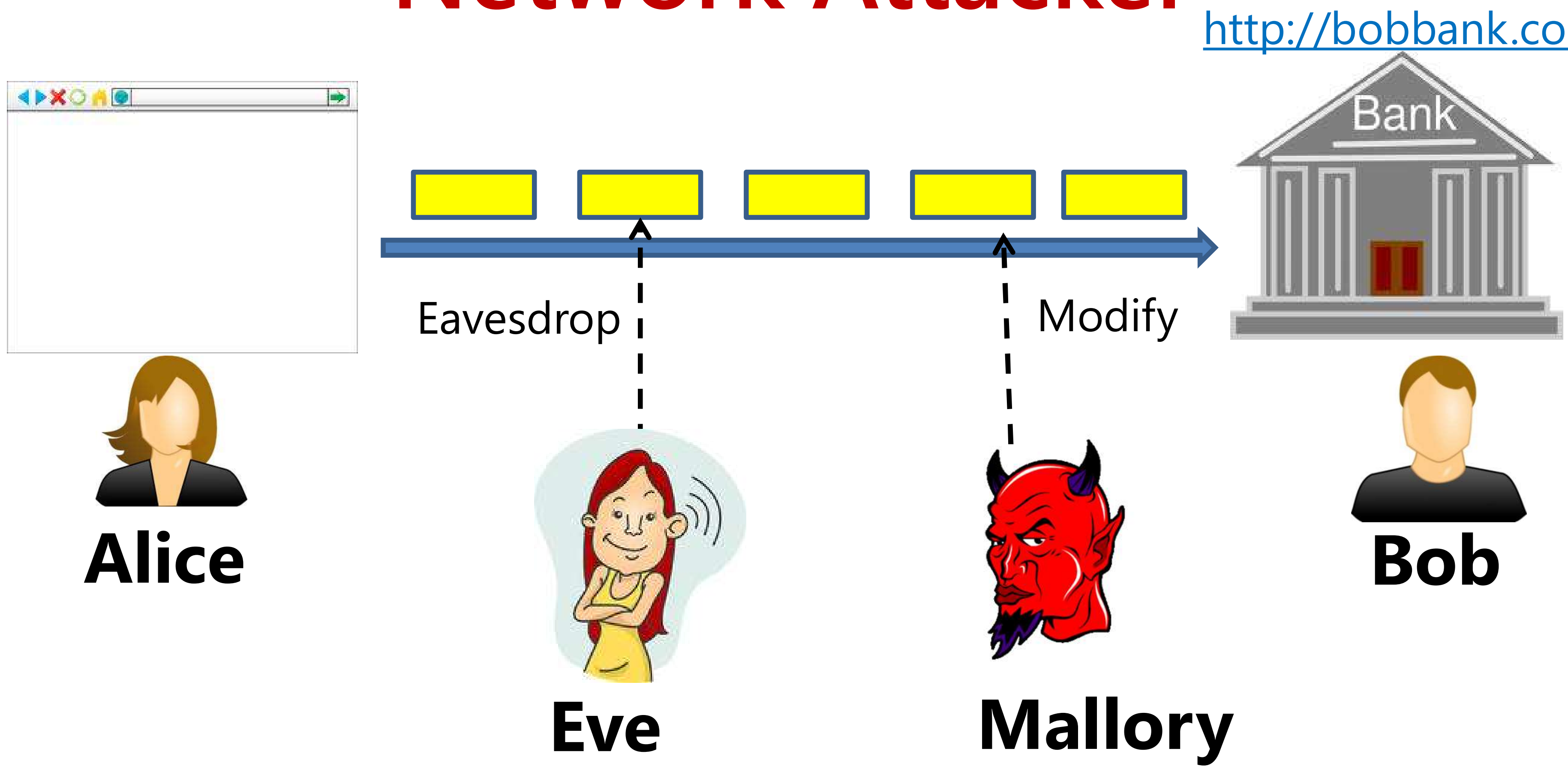
Recap:
Threat Model At Different Layers



Recap:
Story So Far

- Attacks: BGP, TCP/IP, DNS
- Threat Model:
  - Attackers (Eve & Mallory)
  - Assumptions
  - Desired Security Property:
    - The "CIA" of secure channels
- Solution: Secure Channels
- Building Blocks: Crypto primitives

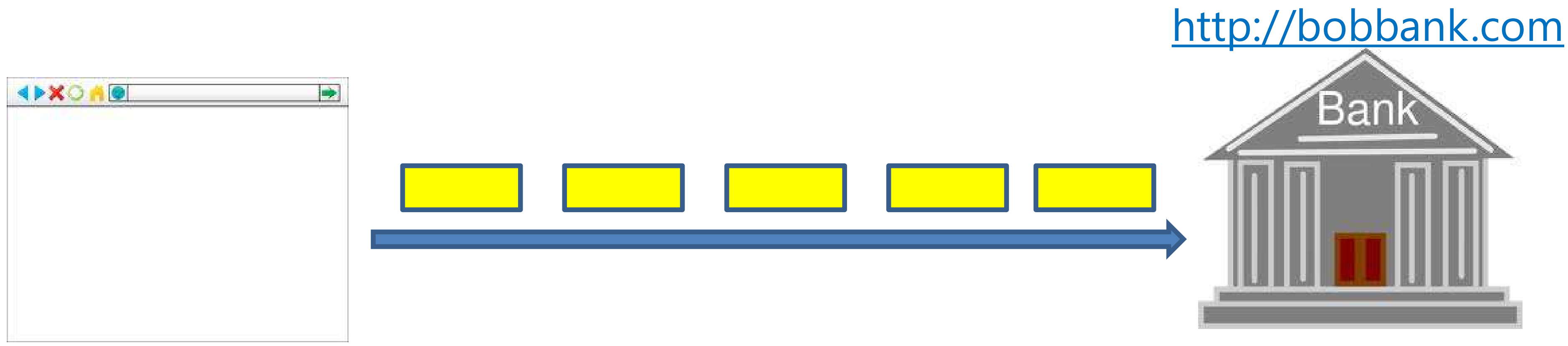
Definition: Network Attacker



Intercept ALL Traffic between Alice and Bob!

- Eve is assumed to only eavesdrop on traffic
- Mallory can listen and tamper with traffic

Definition:
A Secure Channel



A Secure Channel is a data communication protocol established **between 2 programs** which preserves data:

- **C**onfidentiality
- **I**ntegrity
- **A**uthentication

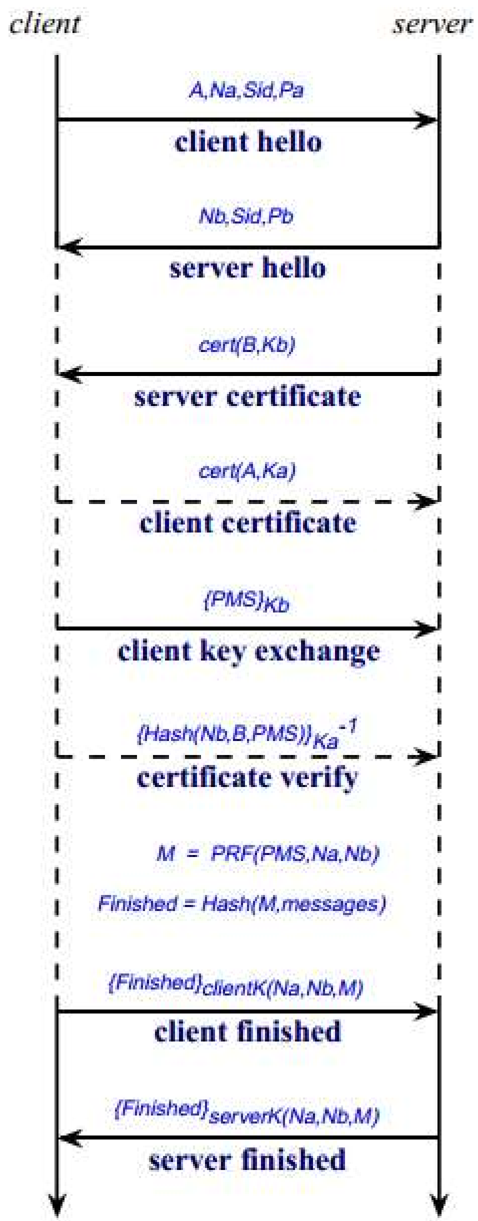
against a computationally-bounded "network attacker"
[Dolev-Yao-1983]

* Note that availability is not a goal. So, denial-of-service attacks are permitted by the threat model.
* Integrity is also referred to as "authenticity" sometimes. Not to be confused with "authentication"

Secure Channels:
The Case of HTTPS (SSL/TLS)

Secure Channel on the Internet: SSL + HTTP = HTTPS

- Used in HTTPS, SMTP, fax, …
- Originated in Netscape SSL 2.0 [1993]
- Revised name to TLS
- Many versions:
  – TLS 1.2 [2008]
  – TLS 1.3 [2018]



Paulson JSC'98

A Very High-Level of TLS

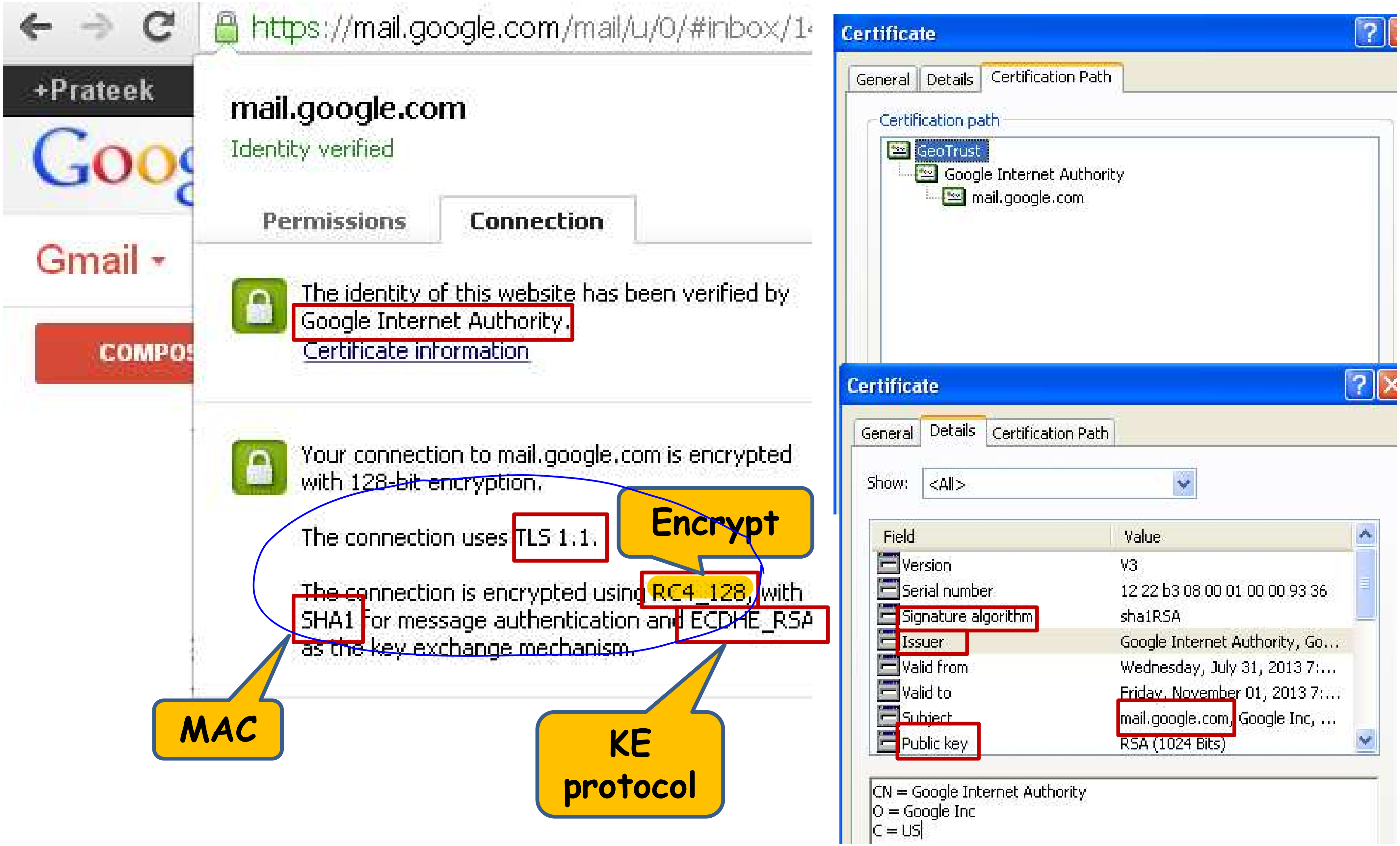
Negotiation Phase

Key Exchange (RSA, DHE, ...)

Symmetric Key Encrypted Session

Re-negotiation

HTTPS



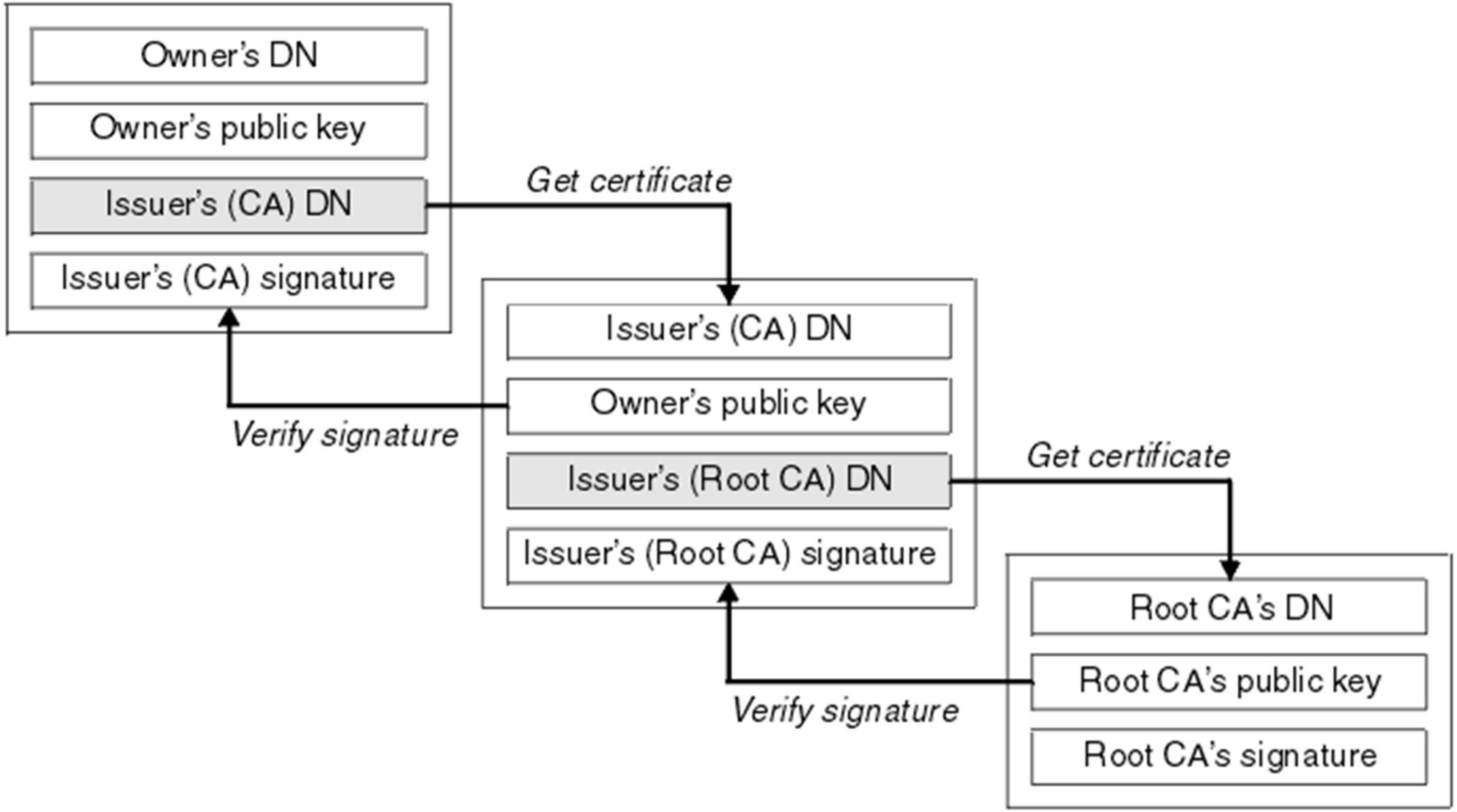
https://mail.google.com/mail/u/0/#inbox/1
+Prateek
Gmail
COMPOS
mail.google.com
Identity verified
Permissions
Connection
The identity of this website has been verified by Google Internet Authority.
Certificate information
Your connection to mail.google.com is encrypted with 128-bit encryption.
The connection uses TLS 1.1.
The connection is encrypted using RC4_128, with SHA1 for message authentication and ECDHE_RSA as the key exchange mechanism.
Encrypt
MAC
KE protocol
Certificate
General
Details
Certification Path
Certification path
GeoTrust
Google Internet Authority
mail.google.com
Certificate
General
Details
Certification Path
Show: <All>
Field
Value
Version
V3
Serial number
12 22 b3 08 00 01 00 00 93 36
Signature algorithm
sha1RSA
Issuer
Google Internet Authority, Go...
Valid from
Wednesday, July 31, 2013 7:...
Valid to
Friday, November 01, 2013 7:...
Subject
mail.google.com, Google Inc, ...
Public key
RSA (1024 Bits)
CN = Google Internet Authority
O = Google Inc
C = US

Chain Of Trusted Certificates

- Root CAs (e.g. GeoTrust)
  - Can designate Intermediate CA
    - E.g. Google Internet Authority
    - Restricted to signing certs for its subdomains
- Where does the browser start trusting?
  - Root CA's certificates are baked in your browser
  - ~50
- Who are the root CAs for the web?
  - Symantec (GeoTrust) – 38%
  - Comodo – 20%
  - GoDaddy – 13%, GlobalSign – 10%

Chain Of Trusted Certificates

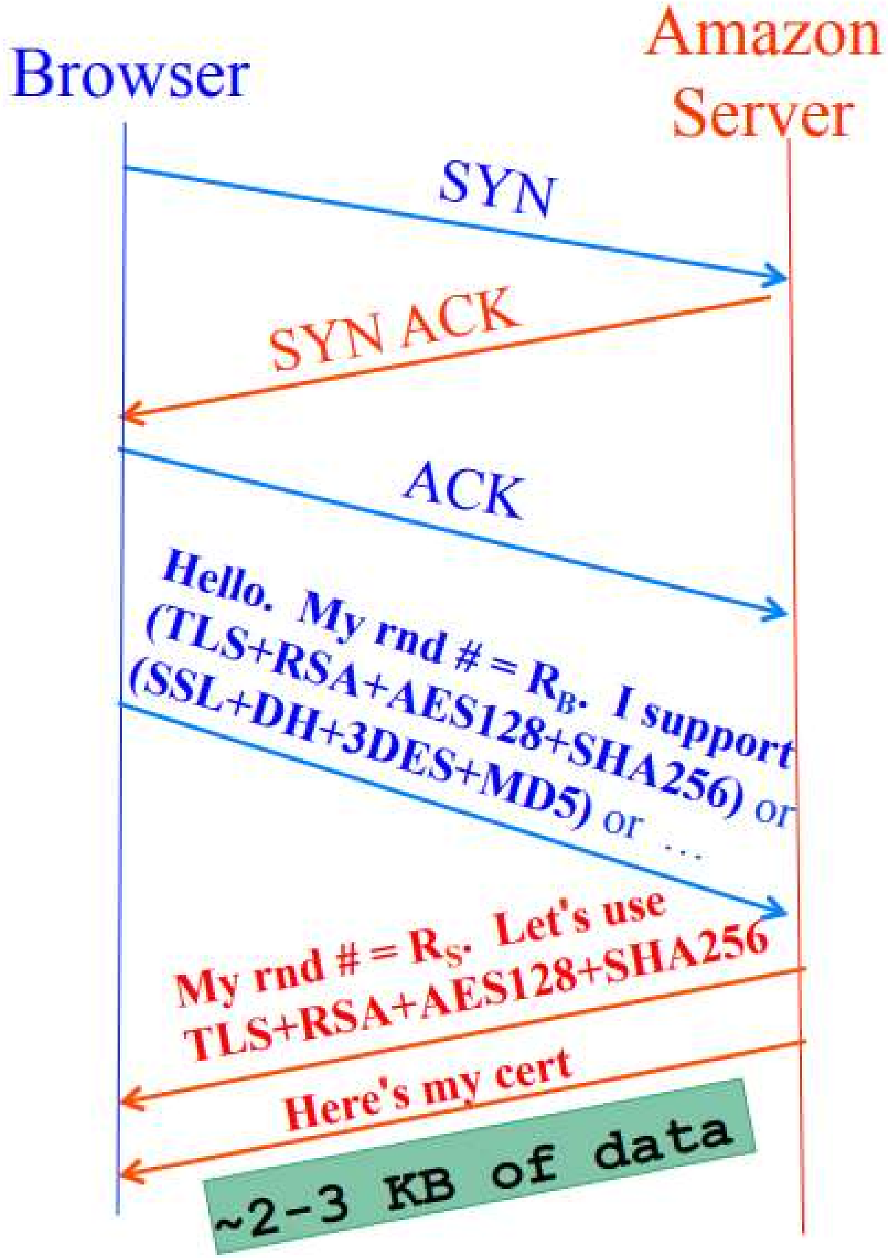
- How Does the Chain Get Verified?



HTTPS (SSL/TLS):
Details of the Protocol

HTTPS Connection (TLS / SSL)

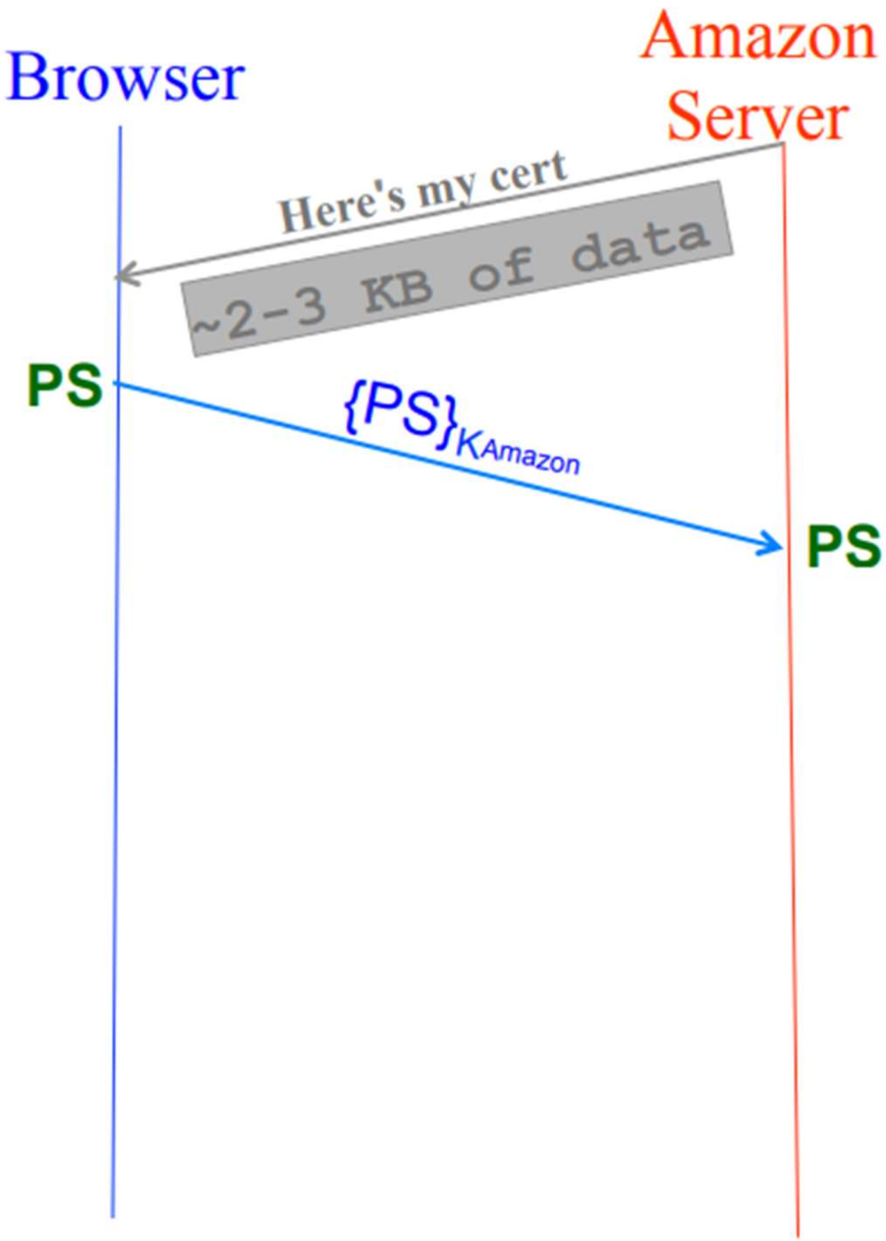
- Browser (client) connects via TCP to Amazon's **`HTTPS`** server
- Client picks 256-bit random number R_B, sends over list of crypto protocols it supports
- Server picks 256-bit random number R_S, selects *cipher suite* to use for this session
- Server sends over its certificate
- (all of this is in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



Slide Courtesy (thanks!): Vern Paxson

HTTPS Connection (TLS / SSL)

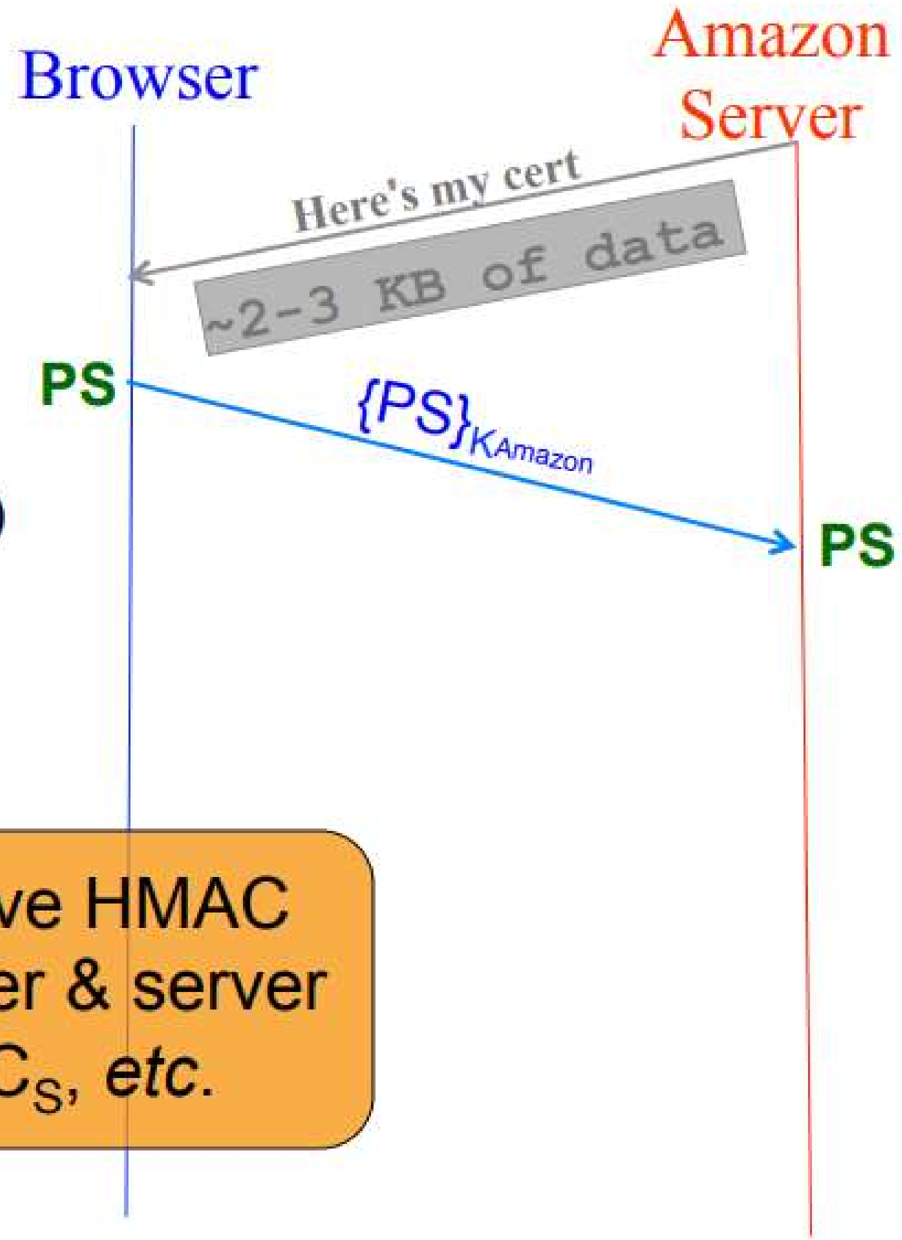
- For RSA, browser constructs long (368 bits) “Premaster Secret” **PS**
- Browser sends PS encrypted using Amazon's public RSA key K_{Amazon}
- Using PS, R_B, and R_S, browser & server derive symm. *cipher keys* (C_B, C_S) & MAC *integrity keys* (I_B, I_S)
  - One pair to use in each direction



Slide Courtesy (thanks!): Vern Paxson

HTTPS Connection (TLS / SSL)

- For RSA, browser constructs long (368 bits) "Premaster Secret" **PS**
- Browser sends PS encrypted using Amazon's public RSA key K_{Amazon}
- Using PS, R_B, and R_S, browser & server derive symm. *cipher keys* (C_B, C_S) & MAC *integrity keys* (I_B, I_S)
  - One pair to use in each direction

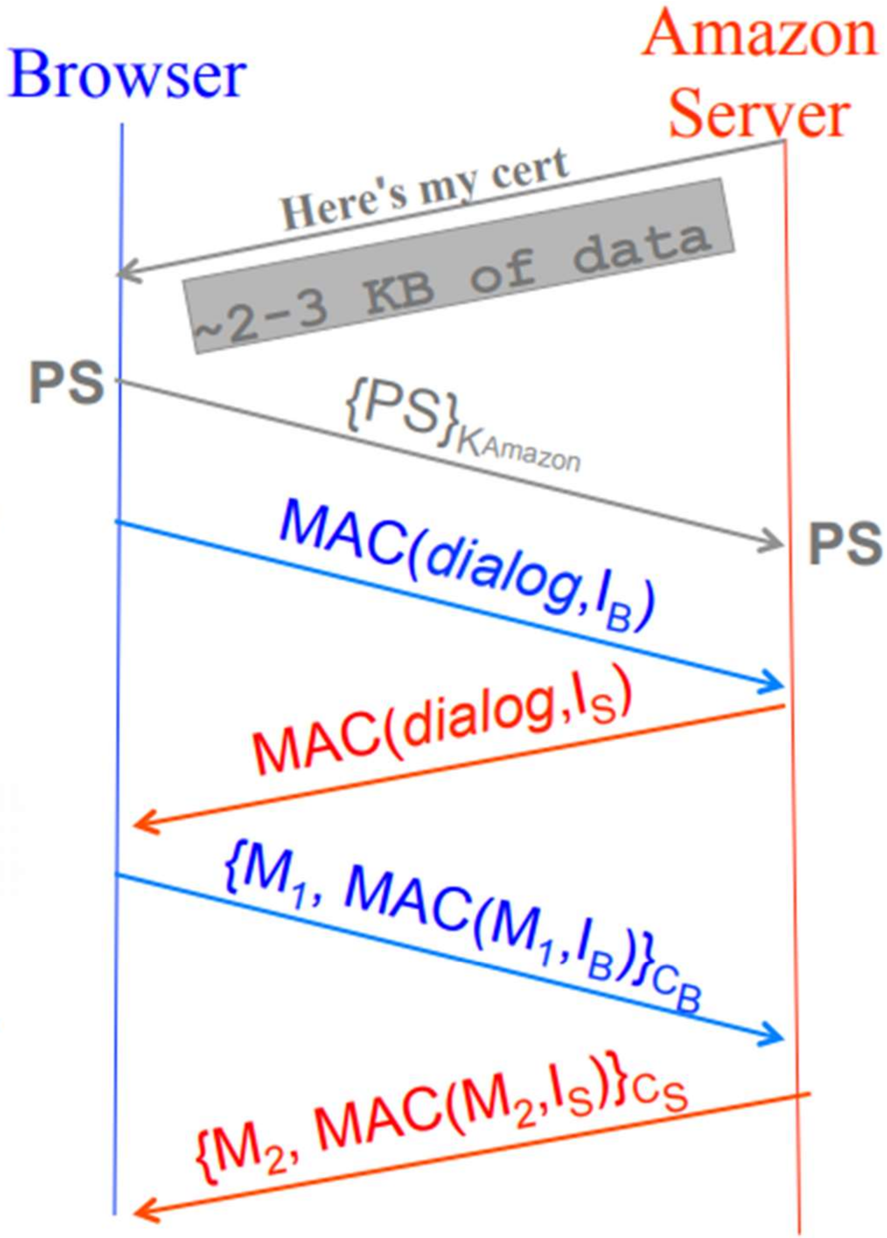


PS is used as the key for iterative HMAC invocations on R_B || R_S. Browser & server use the output to generate C_B, C_S, *etc.*

Slide Courtesy (thanks!): Vern Paxson

HTTPS Connection (TLS / SSL)

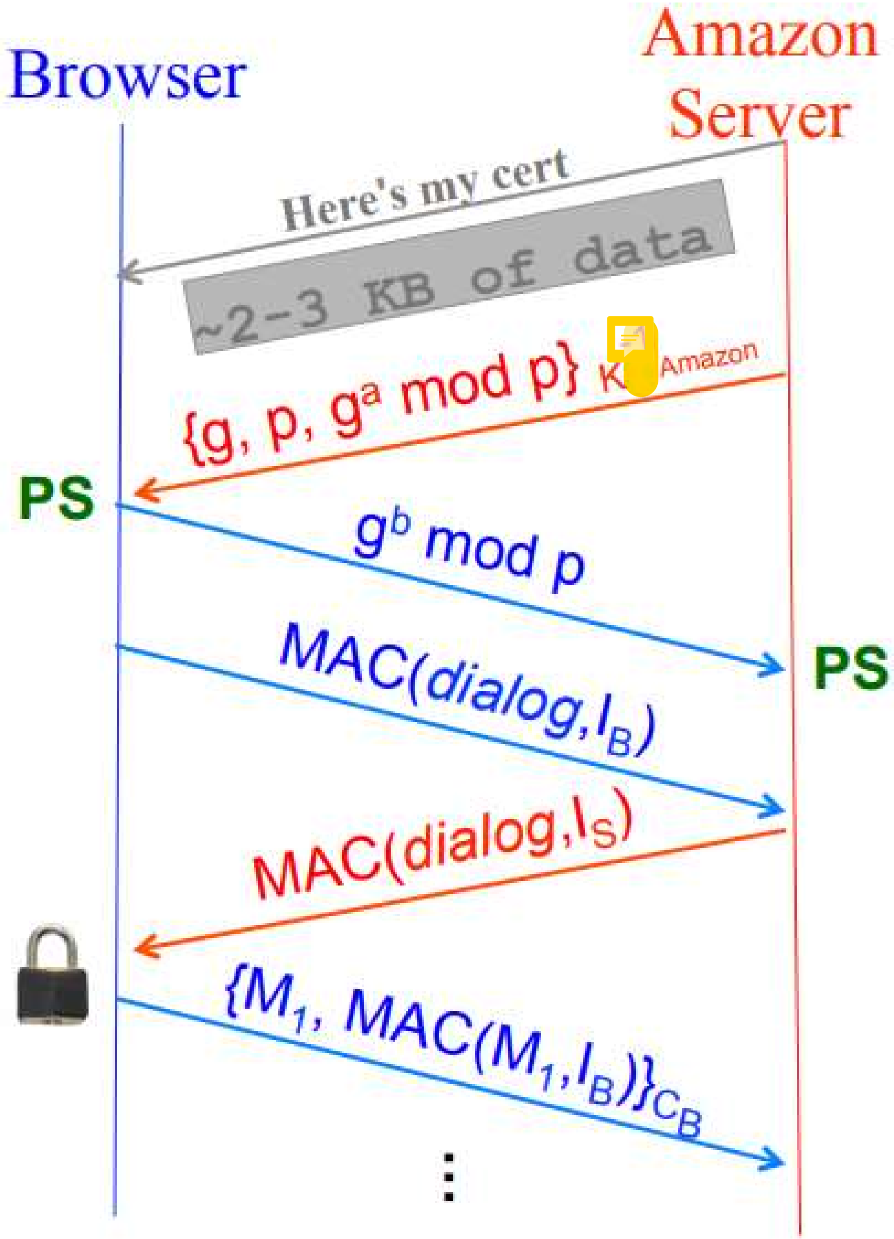
- For RSA, browser constructs long (368 bits) "Premaster Secret" **PS**
- Browser sends PS encrypted using Amazon's public RSA key K_{Amazon}
- Using PS, R_B, and R_S, browser & server derive symm. *cipher keys* (C_B, C_S) & MAC *integrity keys* (I_B, I_S)
  - One pair to use in each direction
- Browser & server exchange MACs computed over entire dialog so far
- If good MAC, Browser displays 🔒
- All subsequent communication encrypted w/ symmetric cipher (e.g., AES128) cipher keys, MACs
  - Messages also numbered to thwart replay attacks



Slide Courtesy (thanks!): Vern Paxson

HTTPS Connection (TLS / SSL): Alternative via Diffie-Hellman KE

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



Slide Courtesy (thanks!): Vern Paxson

HTTPS (SSL/TLS):
Security Analysis

Security Analysis of HTTPS

- We will assume that:
  - Cryptographic primitives are secure
  - Interacting end points are uncompromised
  - A “Perfect” Protocol achieves its stated / defined properties both in design and implementation
  - Attacker can do anything within its defined power

Important Principles:

1. **State threat model, else there’s no security argument!**
2. **Assumptions can fail, but that’s not a flawed argument**
3. **Choose reasonable assumptions in your threat model**

Assumptions in the threat model

- User is using a secure channel
- Crypto primitives are secure
- TLS protocol design is secure
- TLS protocol implementation is secure
- Certificate issuers are uncompromised
- Users check browser UI correctly
- Alice & Bob’s secrets are secure
- Entities are authenticated correctly

Question (I)

- You visit https://gmail.com on a WiFi network with no cert errors. Can you safely assume that your email will reach Gmail safe from **ALL** network attackers?
  – Assume that HTTPS is a "perfect" secure channel

- **IMP:** A **valid security argument** should accept the threat model. You aren't allowed to:
    - treat assumptions of the threat model as false
    - make additional assumptions just to arrive at a conclusion

Question (I) is Important!

- You visit https://gmail.com on a WiFi network with no cert errors. Can you safely assume that your email will reach Gmail safe from ALL network attackers?

- Stick to the threat model – assume the assumptions hold and then check:
  (A) Defeats DNS Cache Poisoning?
  (B) Defeats BGP Route Hijacking?
  (C) Defeats TCP / IP attacks?

Yes, it does!

Question (II)

- You visit http://evil.com

- Should you switch to https://evil.com ?

Source: http://xkcd.com/1247/

Question (III)

- You are installing an OS update from http://microsoft.com
- Is there a problem here? Fix?

SSL / TLS Protocol (Simplified)
CS3235 Lecture Notes

The basic TLS/SSL handshake protocol is presented below. Note that we use $\{X\}_{K^{-1}}$ to denote an encrypted message X with private key K^{-1} and $\{X\}_K$ is the encrypted message with the public key K. In the Diffie-Hellman (DH) below we denote g as the group generator over a multiplicative group of integers modulo a prime p; values a and b are secrets. Assume that the browser, the CA and the CA keys are not compromised.

1. Client $\xrightarrow{\text{ClientHello}}$ Server: Client sends a **ClientHello** message containing random number R_b and supported cipher suites $Cipher_c$. The ciphersuite will select the key exchange protocol (e.g. RSA, Diffie-Hellman (DH)), the encryption algorithm for the data and the MAC algorithm.

2. Server $\xrightarrow{\text{ServerHello}}$ Client: Server sends a **ServerHello** message random number R_s, supported cipher suites $Cipher_s$.

3. Server $\xrightarrow{\text{Certificate}}$ Client: Server sends its **Certificate** containing the server's public key K_s signed with K_{auth}^{-1} where $auth$ is some signing authority (typically a CA) and ${K_{auth}}^{-1}$ is the authority's private key. If the certificate is invalid, client aborts.

4. Server $\xrightarrow{\text{ServerKeyExchange}}$ Client:
   If DH is selected, the server sends **ServerKeyExchange** message $\{g, p, g^a \bmod p\}_{K_s^{-1}}$

5. Server $\xrightarrow{\text{ServerHelloDone}}$ Client: Server signals end of handshake negotiation

6. Client $\xrightarrow{\text{ClientKeyExchange}}$ Server:
   If DH is selected, client sends $g^b \bmod p$.
   If RSA is selected, client sends $\{PS\}_{K_s}$, where PS is called PreMasterSecret and is a random value generated by the client.
   After the **ClientKeyExchange** the client and server will derive a shared key: $K_{sess} \leftarrow$ derived from DH messages 4, 6 via a standard Diffie Hellman Key Exchange procedure, or $K_{sess} = PS$ if RSA is selected. Then the client and server derive the cipher keys CK_c, CK_s and integrity keys I_c, I_s deterministically using function `f()`:
   $(CK_c, CK_s, I_c, I_s) \leftarrow f(R_b, R_s, K_{sess})$

7. Client $\xrightarrow{\text{ChangeCipherSpec}}$ Server: Client sends MAC(msg, I_c)

8. Server $\xrightarrow{\text{ChangeCipherSpec}}$ Client: Server sends MAC(msg, I_s)

After the last step all data takes the form:

- Client $\rightarrow$ Server: $\{M1, \text{MAC}(M1, I_c)\}_{CK_c}$
- Server $\rightarrow$ Client: $\{M2, \text{MAC}(M2, I_s)\}_{CK_s}$