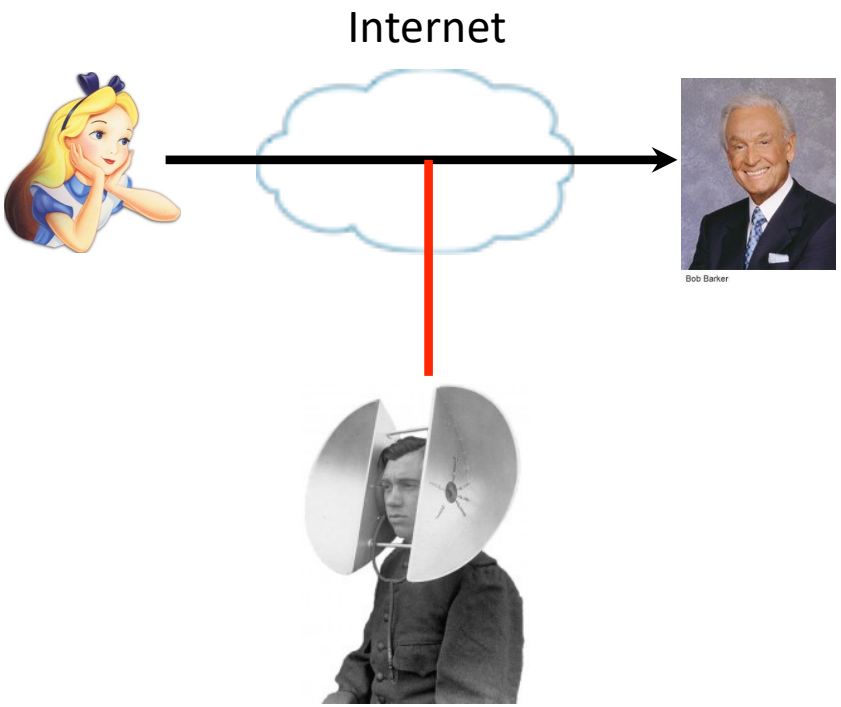
CSCI 445:
Mobile Application Security

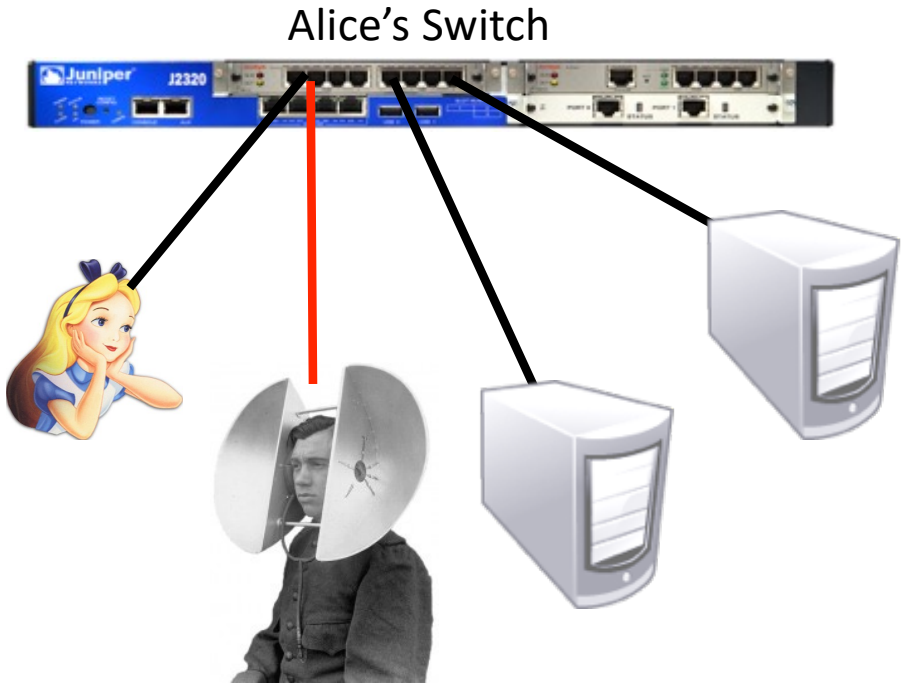
Lecture 6

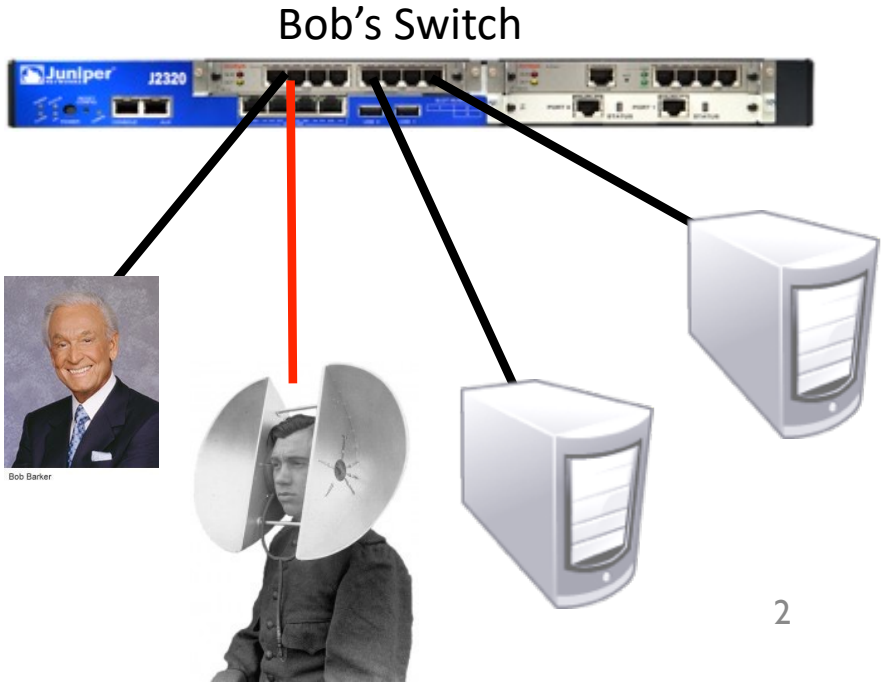
Prof. Adwait Nadkarni

Derived from slides by William Enck and Micah Sherr

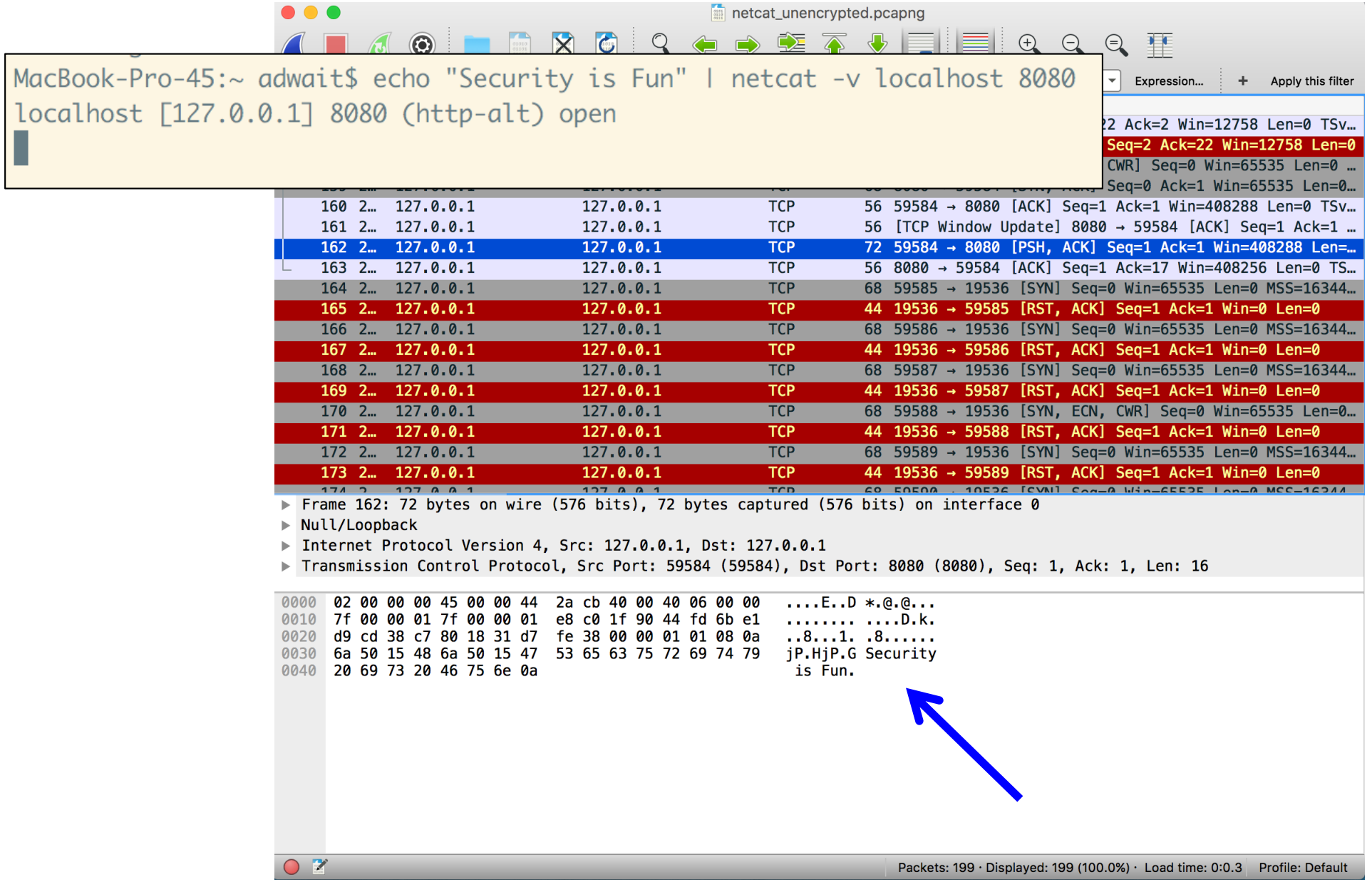
Eavesdropping







Why is crypto useful?



Why is this bad?

- Its just an instant message, right?

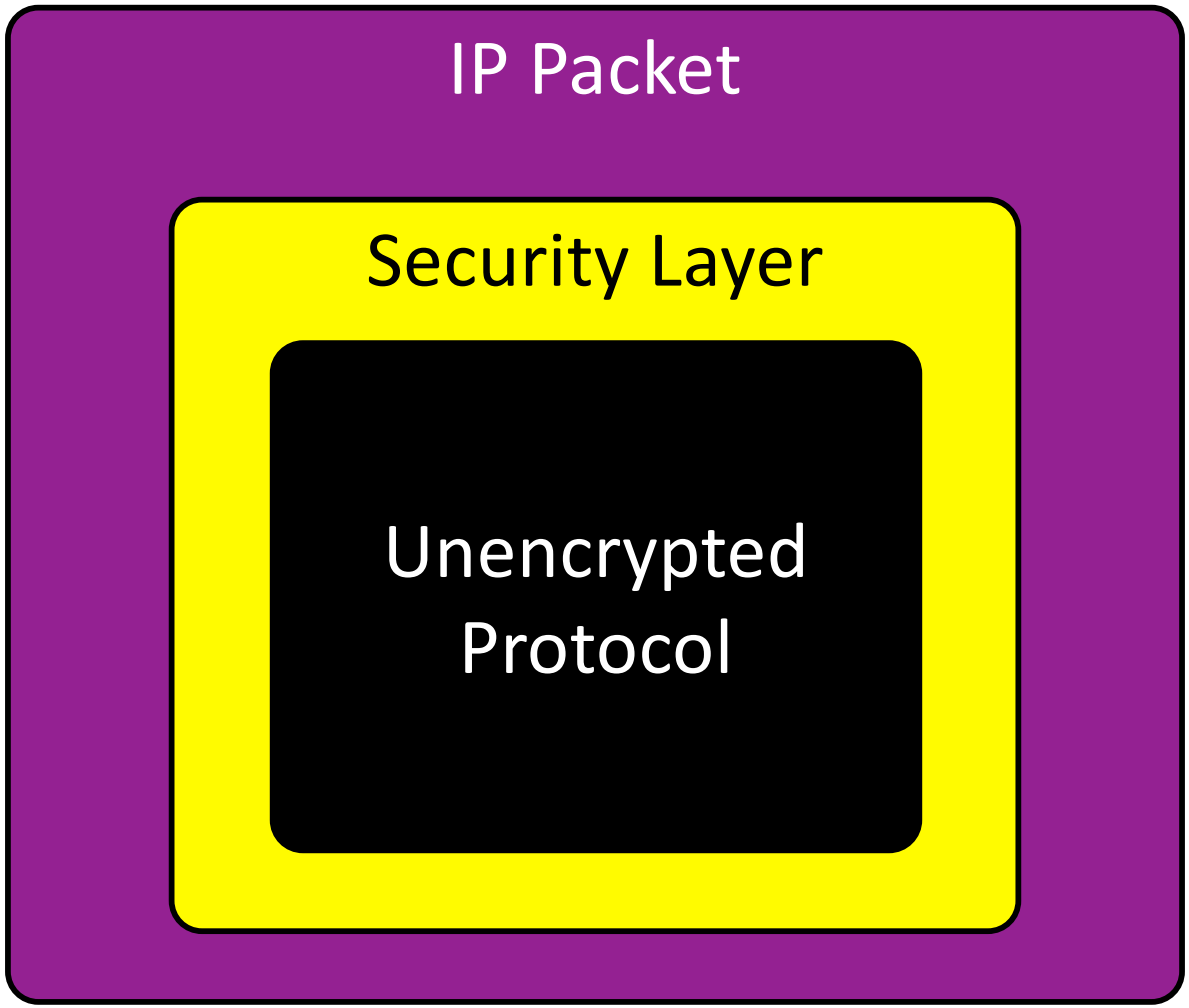
Alice uses the Internet for:

- Email
- Banking
- Online shopping
- Social networking
- …

Let's use that crypto stuff

- Let's build some new protocols
  - HTTP → SecureHTTP
  - POP → POPSecure
  - IMAP → Cry[illegible]
  - SMTP → SMTP[illegible]
  - FTP → FTPS
  - Jabber → SecJabber
  - Telnet → TelCryptNet

Let's build a crypto-wrapper standard instead


IP Packet
Security Layer
Unencrypted
Protocol

What properties should this crypto-wrapper have?

- Confidentiality
- Integrity
- Authenticity
  - Server
  - Client
  - Mutual authentication

SSL / TLS

History

- **Secure Sockets Layer (SSL)** developed by Netscape (remember them?) in 1995
  - Version 1 never released
  - Version 2 incorporated into Netscape Navigator 1.1
  - Microsoft fixes vulnerabilities in SSLv2 and introduces Private Communications Technology (PCT) protocol
  - Netscape overhauls SSLv2, fixing some more security issues, and releases SSLv3
  - IETF takes over and releases **Transport Layer Security (TLS)**, a non-interoperable upgrade to SSLv3
    - current version is TLS version 1.3, RFC 8446 (August 2018)

K.I.S.S.

- Application-layer protocol
- Operates over TCP **-- WHY?**

Overview

- Alice (client) initiates conversation with Bob (server)
- Bob sends Alice his certificate
- Alice verifies certificate
- Alice picks a random number S and sends it to Bob, encrypted with Bob's public key
- Both parties derive key material from S
- Client and server exchange encrypted and integrity-protected data

SSLv2 Handshake

Alice

Bob

`ClientHello`, Version, Cipher list., R_{Alice}

`ServerHello`, Ver., Cert., Chosen cipher, R_{Bob}

E(Bob+,S)

E(K',Data)

Bob Barker

Alice computes **master secret k** as $K=f(S,R_{Alice},R_{Bob})$

Encryption and integrity keys derived from Master Secret

Alice randomly chooses S.

Bob computes **master secret k** as $K=f(S,R_{Alice},R_{Bob})$

Cryptographic Parameters

- Generated from
  - the master secret K
  - Rc
  - Rs
- *Six values* to be generated
  - client authentication and encryption keys
  - server authentication and encryption keys
  - client encryption IV
  - server encryption IV
- Generator functions: $k_i = g_i(K,Rc,Rs)$

Authentication

Alice

Bob

`ClientHello`, Version, Cipher list. R_{Alice}

`ServerHello`, Ver., Cert., Chosen cipher, R_{Bob}

E(Bob+,S)

E(K',Data)

Q: Which parties are authenticated?

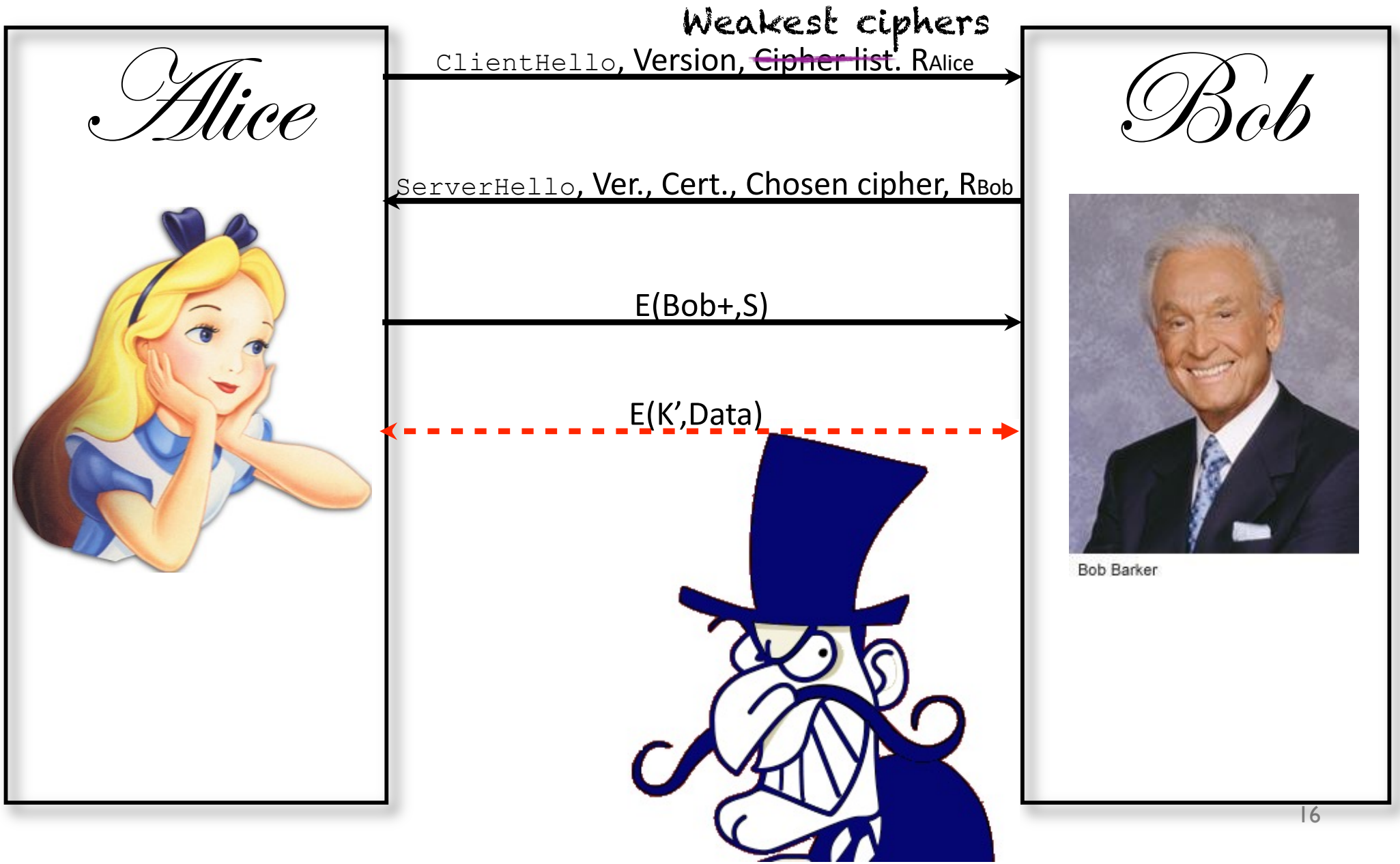
Bob Barker

Cipher Suites

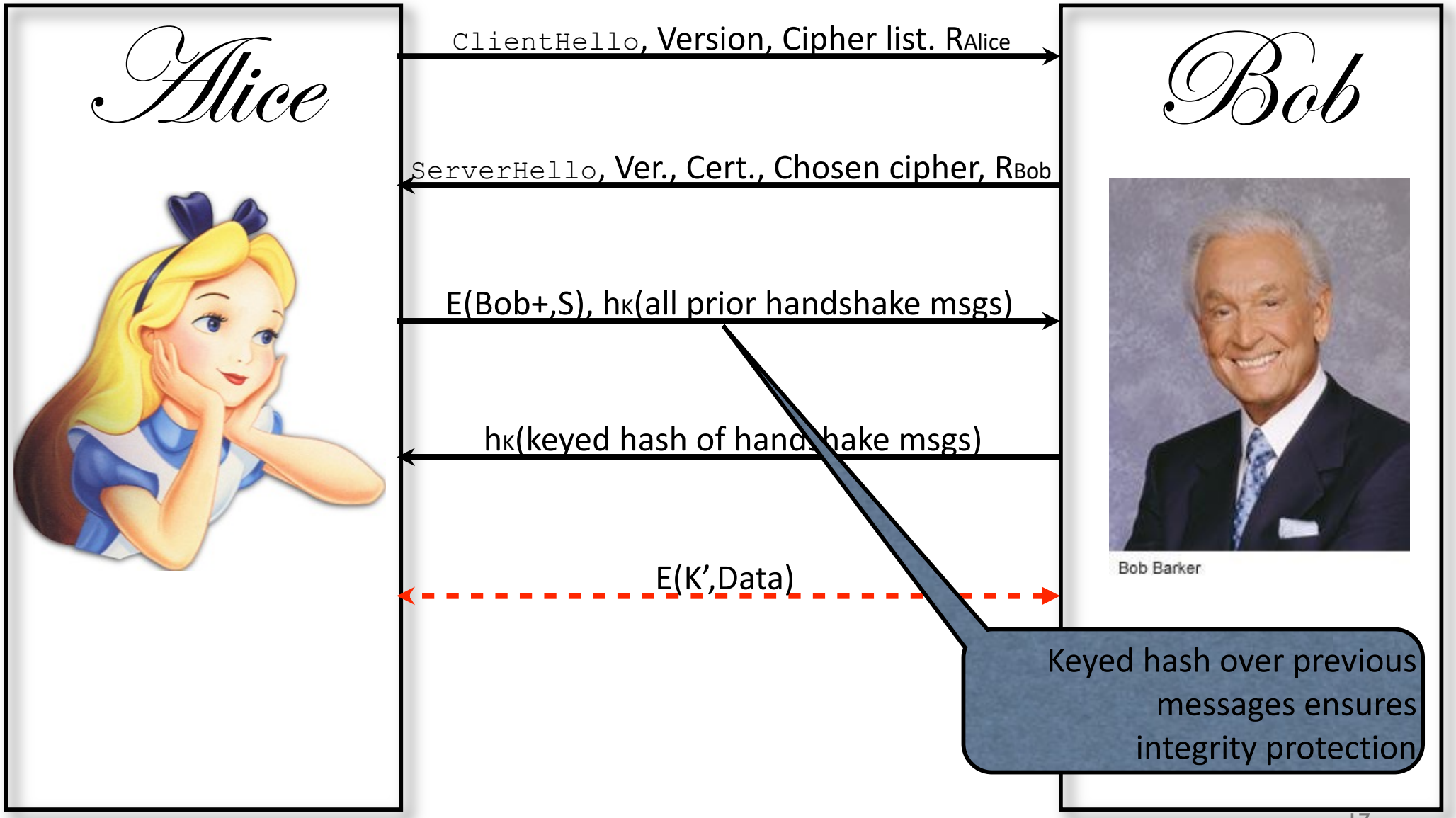
- Includes encryption algorithm, key length, block mode, and integrity checksum algorithm
- ~90 defined cipher suites
- Alice gives Bob a list of supported cipher suites; Bob makes final choice

```
% openssl ciphers -v
ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH     Au=RSA  Enc=AESGCM(256) Mac=AEAD
ECDHE-ECDSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH     Au=ECDSA Enc=AESGCM(256) Mac=AEAD
ECDHE-RSA-AES256-SHA384 TLSv1.2 Kx=ECDH     Au=RSA  Enc=AES(256)  Mac=SHA384
ECDHE-ECDSA-AES256-SHA384 TLSv1.2 Kx=ECDH     Au=ECDSA Enc=AES(256)  Mac=SHA384
ECDHE-RSA-AES256-SHA    SSLv3 Kx=ECDH     Au=RSA  Enc=AES(256)  Mac=SHA1
ECDHE-ECDSA-AES256-SHA  SSLv3 Kx=ECDH     Au=ECDSA Enc=AES(256)  Mac=SHA1
SRP-DSS-AES-256-CBC-SHA SSLv3 Kx=SRP      Au=DSS  Enc=AES(256)  Mac=SHA1
SRP-RSA-AES-256-CBC-SHA SSLv3 Kx=SRP      Au=RSA  Enc=AES(256)  Mac=SHA1
SRP-AES-256-CBC-SHA     SSLv3 Kx=SRP      Au=SRP  Enc=AES(256)  Mac=SHA1
DHE-DSS-AES256-GCM-SHA384 TLSv1.2 Kx=DH       Au=DSS  Enc=AESGCM(256) Mac=AEAD
DHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=DH       Au=RSA  Enc=AESGCM(256) Mac=AEAD
DHE-RSA-AES256-SHA256   TLSv1.2 Kx=DH       Au=RSA  Enc=AES(256)  Mac=SHA256
DHE-DSS-AES256-SHA256   TLSv1.2 Kx=DH       Au=DSS  Enc=AES(256)  Mac=SHA256
DHE-RSA-AES256-SHA      SSLv3 Kx=DH       Au=RSA  Enc=AES(256)  Mac=SHA1
DHE-DSS-AES256-SHA      SSLv3 Kx=DH       Au=DSS  Enc=AES(256)  Mac=SHA1
DHE-RSA-CAMELLIA256-SHA SSLv3 Kx=DH       Au=RSA  Enc=Camellia(256) Mac=SHA1
DHE-DSS-CAMELLIA256-SHA SSLv3 Kx=DH       Au=DSS  Enc=Camellia(256) Mac=SHA1
ECDH-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH/RSA Au=ECDH Enc=AESGCM(256) Mac=AEAD
ECDH-ECDSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH/ECDSA Au=ECDH Enc=AESGCM(256) Mac=AEAD
ECDH-RSA-AES256-SHA384  TLSv1.2 Kx=ECDH/RSA Au=ECDH Enc=AES(256)  Mac=SHA384
ECDH-ECDSA-AES256-SHA384 TLSv1.2 Kx=ECDH/ECDSA Au=ECDH Enc=AES(256)  Mac=SHA384
ECDH-RSA-AES256-SHA     SSLv3 Kx=ECDH/RSA Au=ECDH Enc=AES(256)  Mac=SHA1
ECDH-ECDSA-AES256-SHA   SSLv3 Kx=ECDH/ECDSA Au=ECDH Enc=AES(256)  Mac=SHA1
AES256-GCM-SHA384       TLSv1.2 Kx=RSA      Au=RSA  Enc=AESGCM(256) Mac=AEAD
AES256-SHA256           TLSv1.2 Kx=RSA      Au=RSA  Enc=AES(256)  Mac=SHA256
AES256-SHA              SSLv3 Kx=RSA      Au=RSA  Enc=AES(256)  Mac=SHA1
CAMELLIA256-SHA         SSLv3 Kx=RSA      Au=RSA  Enc=Camellia(256) Mac=SHA1
PSK-AES256-CBC-SHA      SSLv3 Kx=PSK      Au=PSK  Enc=AES(256)  Mac=SHA1
ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH     Au=RSA  Enc=AESGCM(128) Mac=AEAD
ECDHE-ECDSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH     Au=ECDSA Enc=AESGCM(128) Mac=AEAD
ECDHE-RSA-AES128-SHA256 TLSv1.2 Kx=ECDH     Au=RSA  Enc=AES(128)  Mac=SHA256
ECDHE-ECDSA-AES128-SHA256 TLSv1.2 Kx=ECDH     Au=ECDSA Enc=AES(128)  Mac=SHA256
```

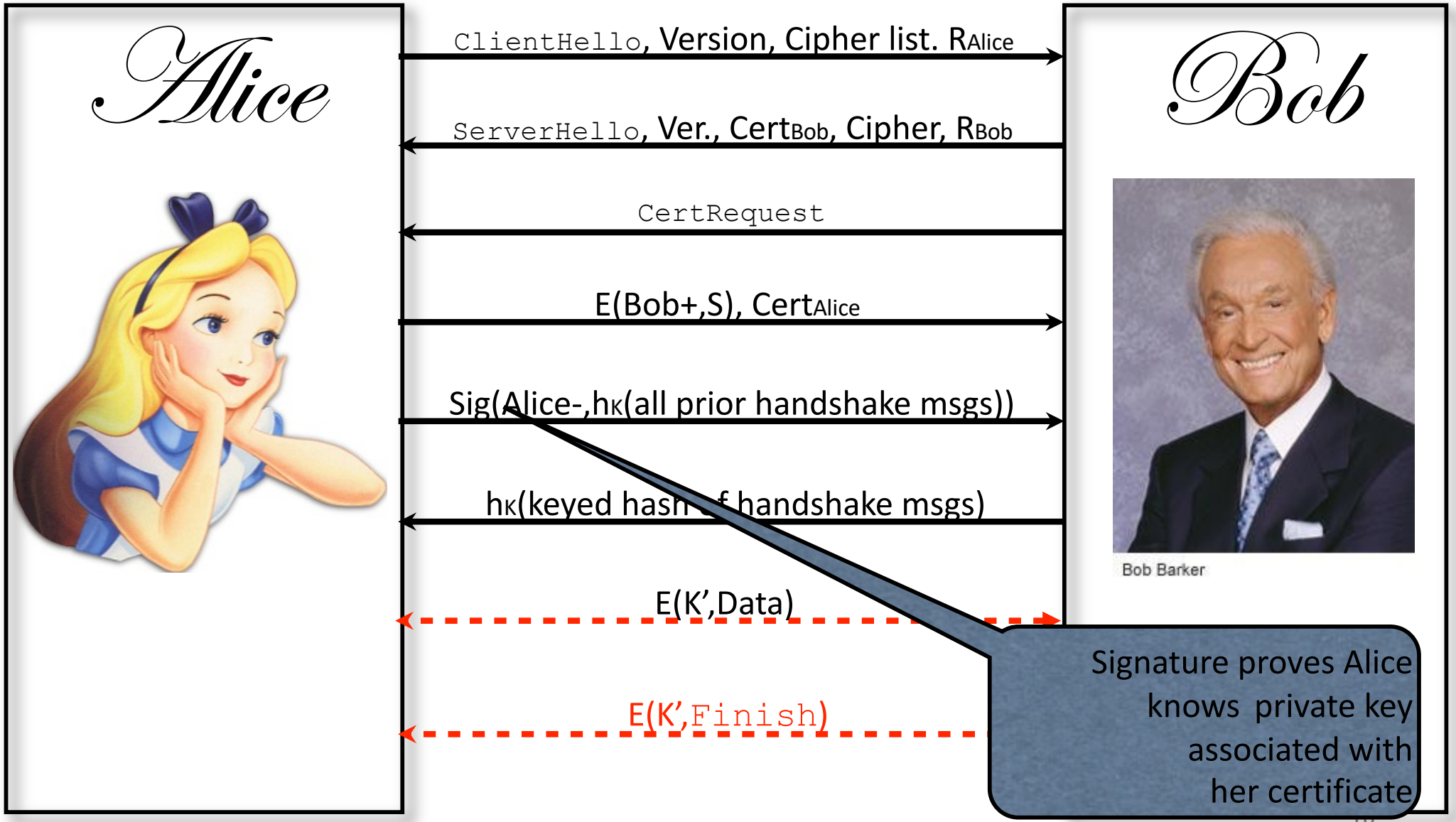
SSLv2 Problems



SSLv3 Fixes

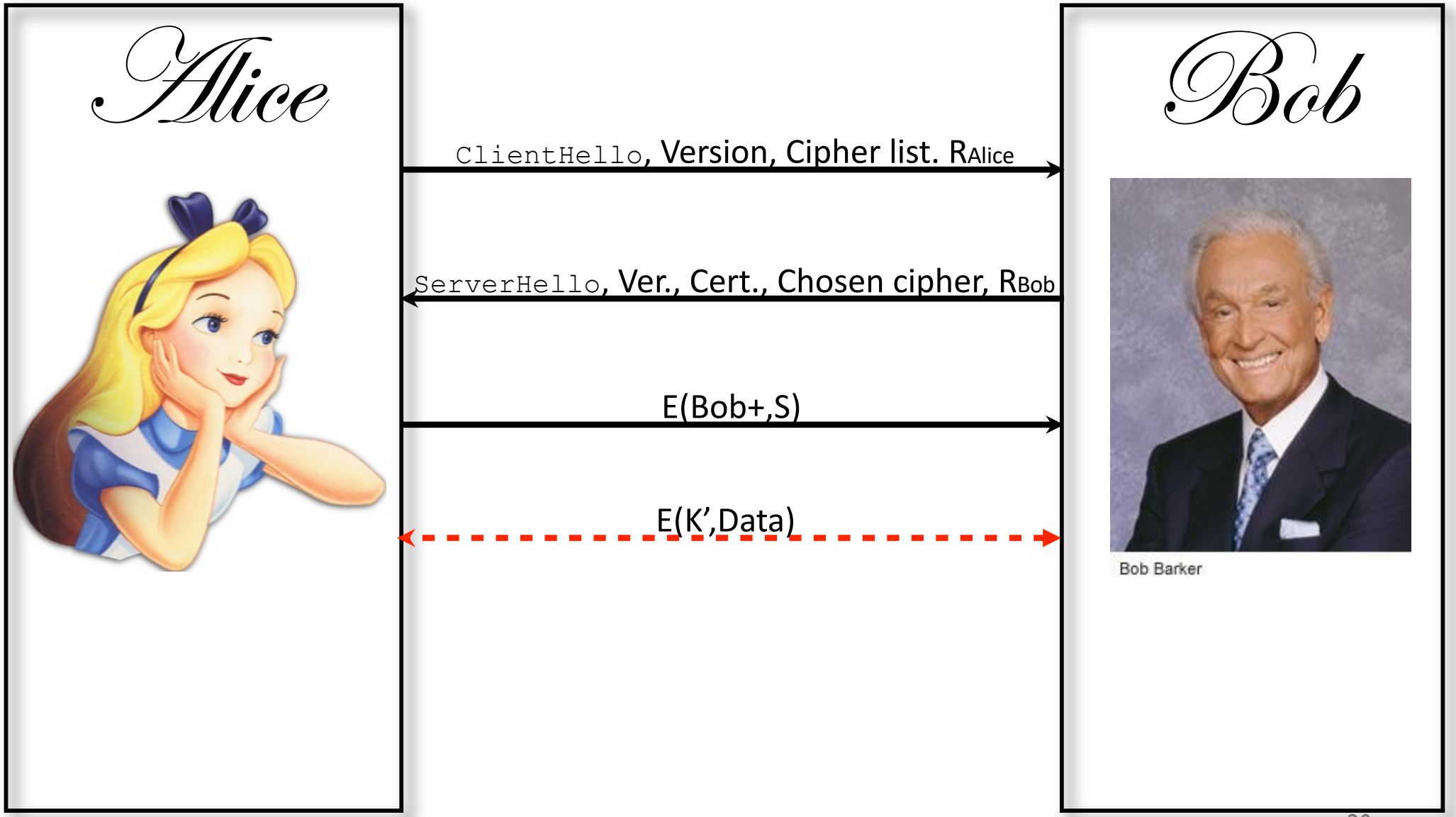

Alice
Bob
ClientHello, Version, Cipher list. RAlice
ServerHello, Ver., Cert., Chosen cipher, RBob
E(Bob+,S), hK(all prior handshake msgs)
hK(keyed hash of handshake msgs)
E(K',Data)
Bob Barker
Keyed hash over previous messages ensures integrity protection

SSL/TLS with Server and Client Authentication

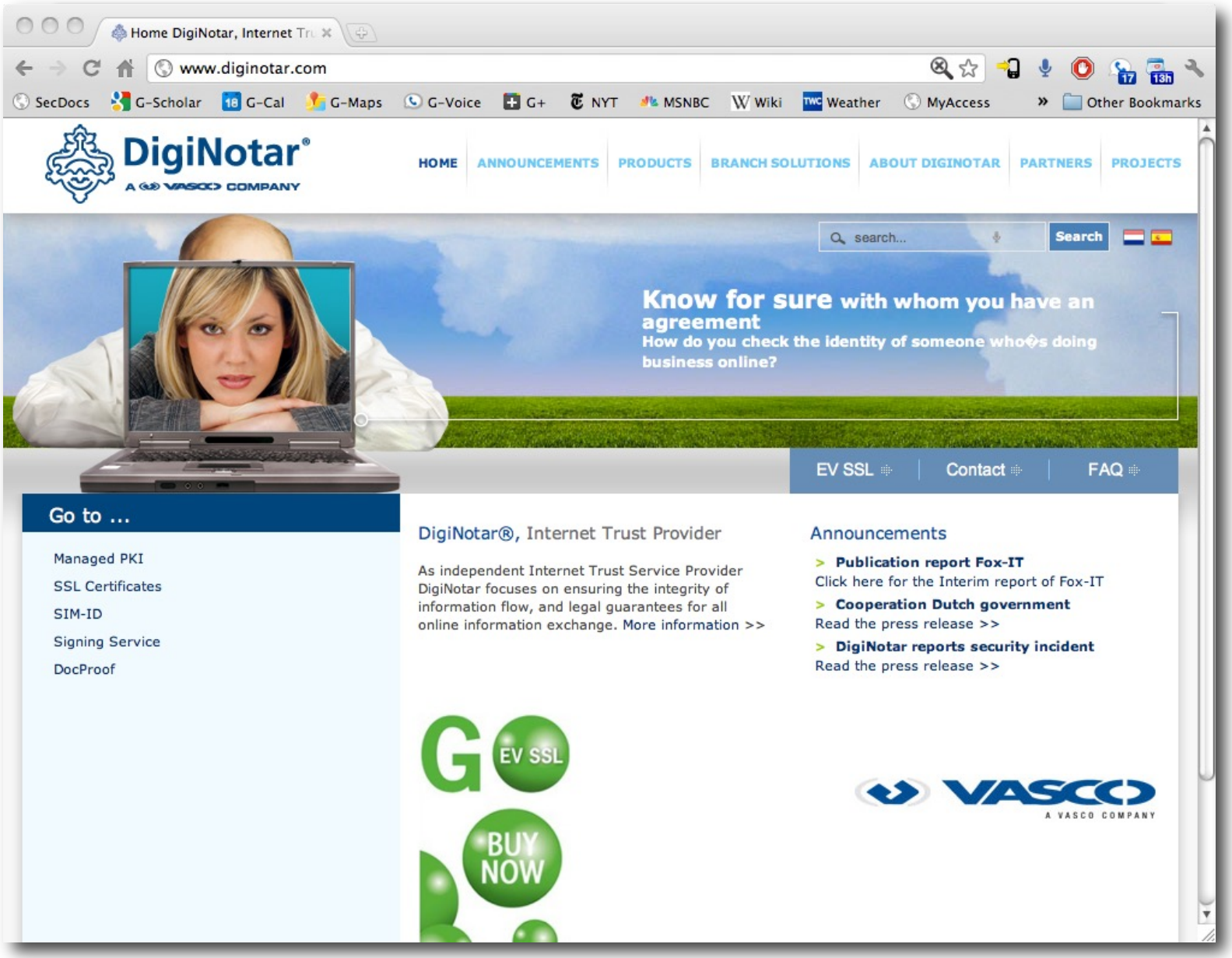


Problems with TLS/SSL

If Bob's cert isn't verified, how do you know you're actually talking to Bob?

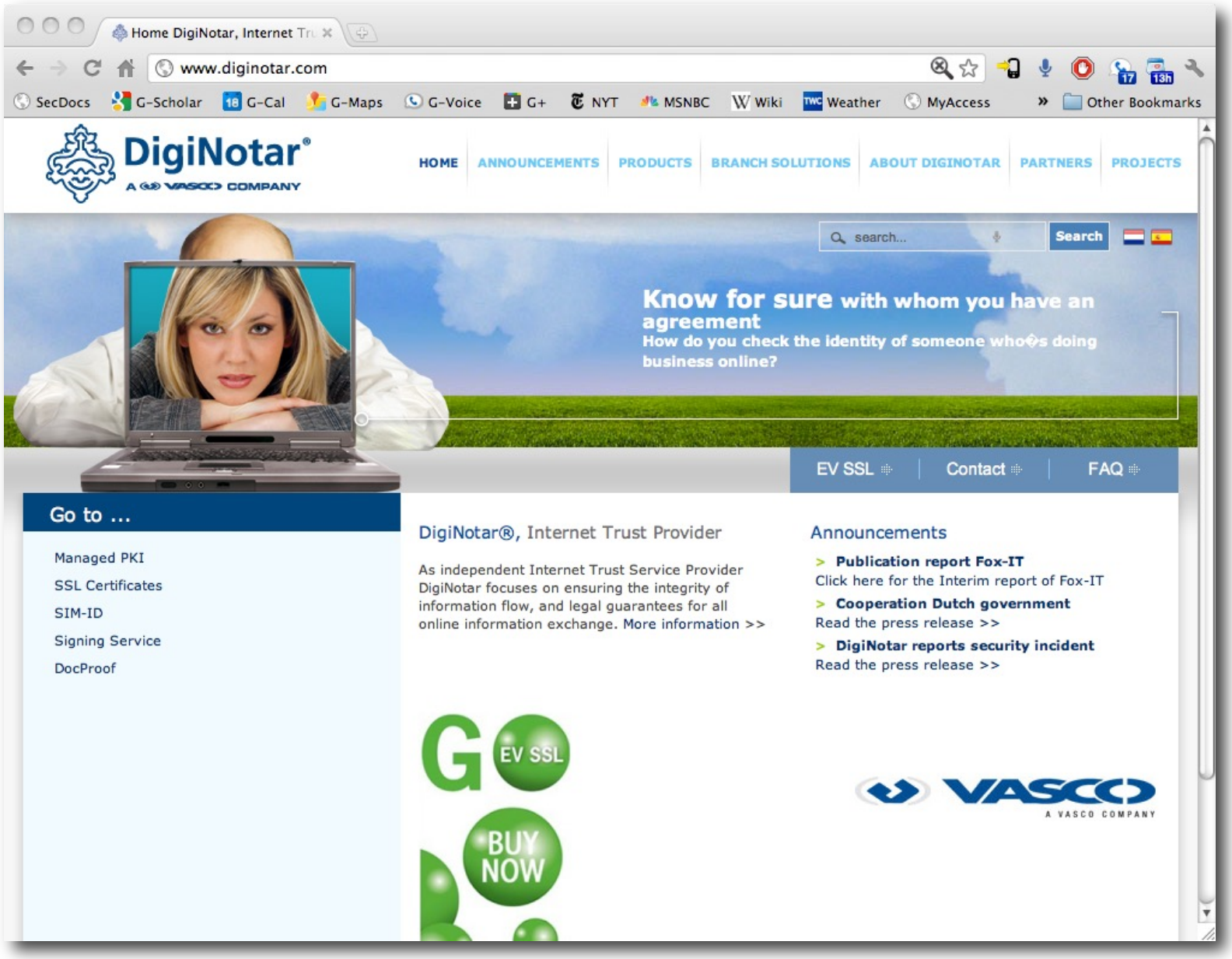


Solution: Use a PKI



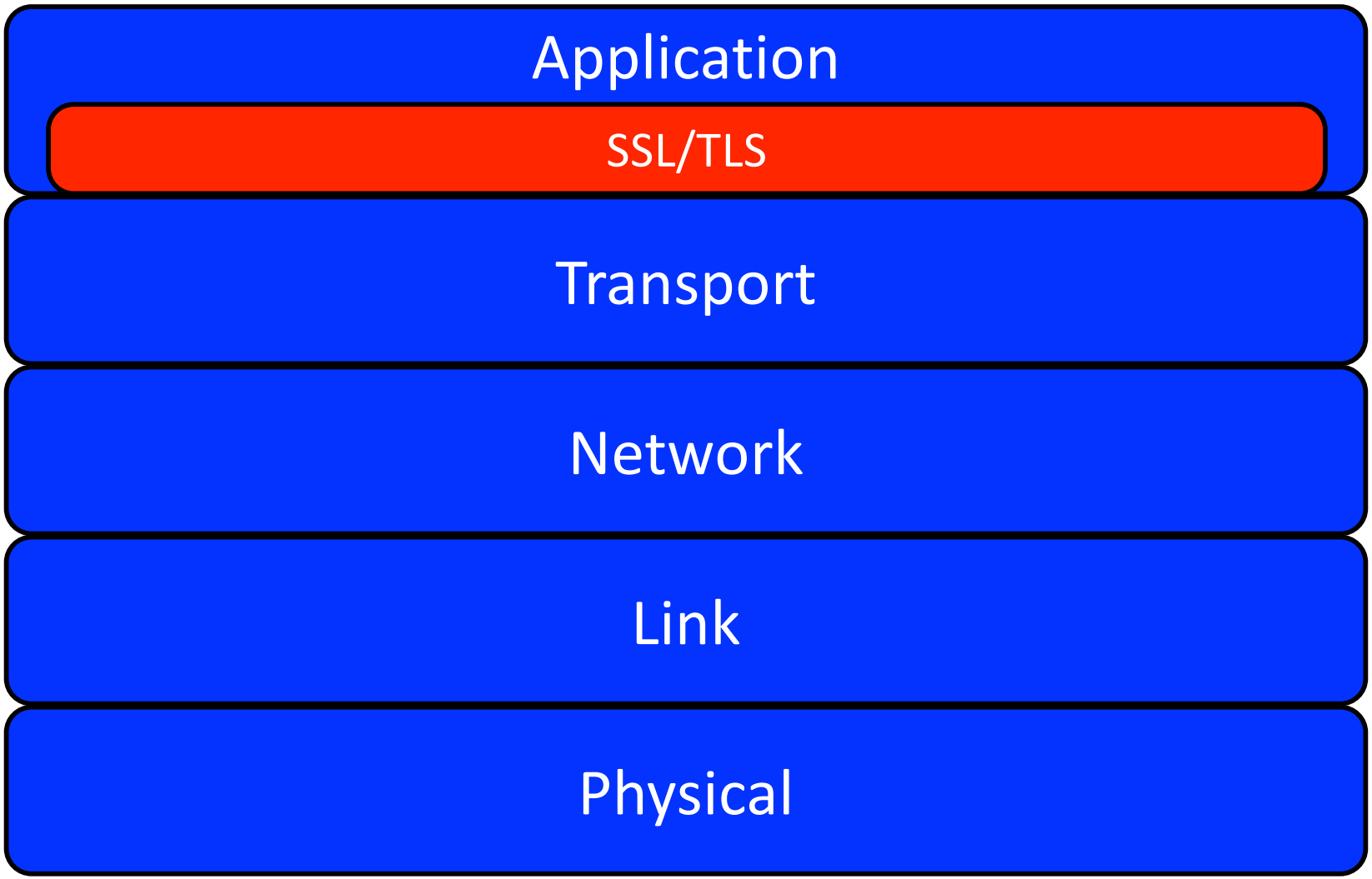
- Any CA may sign any certificate
- Browser weighs all root CAs equally
- *Q: Is this problematic?*

Recall: The DigiNotar Incident



SSL/TLS in the Real World

Network Stack, revisited

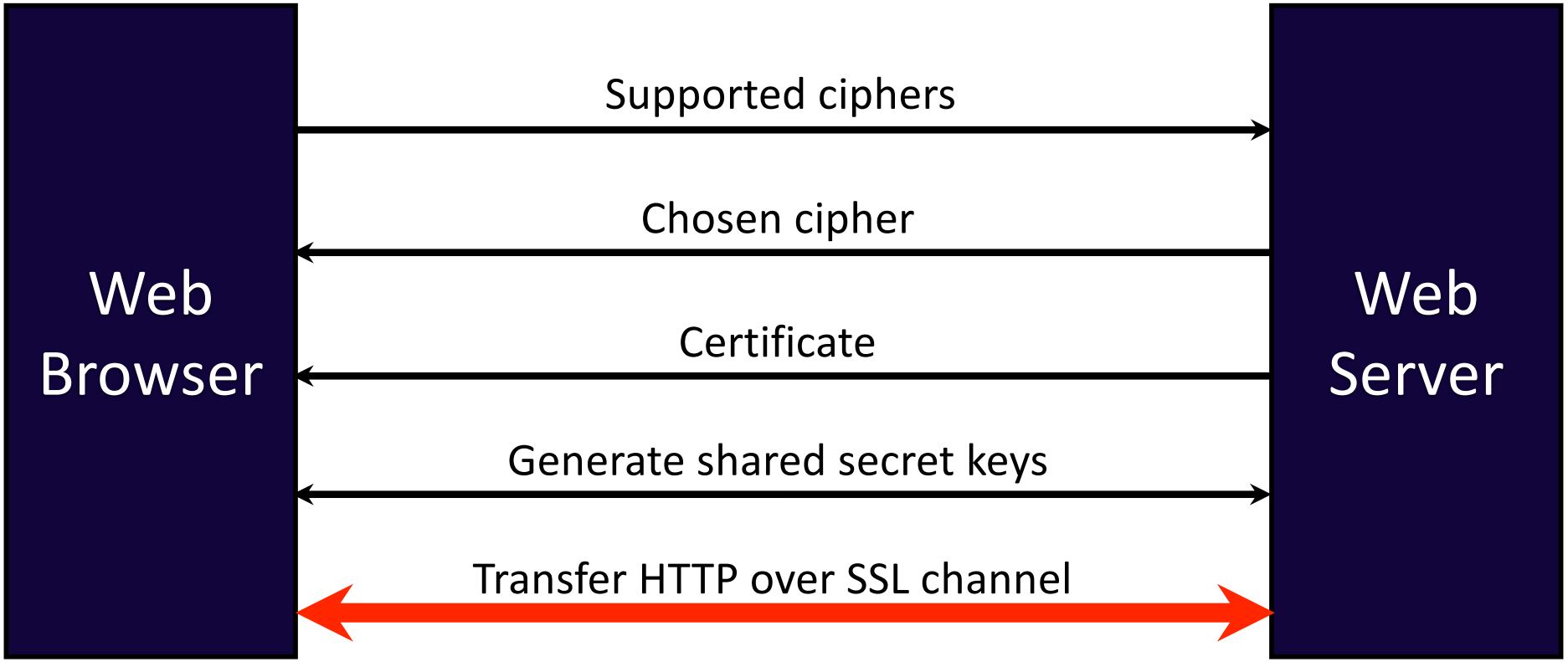


SSL/TLS in the Real World

- All (modern) browsers support SSLv3, TLS 1.2
  - Most major browsers support TLS 1.3
- Client authentication very rare **-- WHY?**
- Implementations:
  - HTTP (80) → HTTPS (443)
  - POP (110) → POP3S (995)
  - IMAP (143) → IMAPS (993)
  - SMTP (25) → SMTP with SSL (465)
  - FTP (20,21)→ FTPS (989,990)
  - Telnet (23) → Telnets (992)

SSL/TLS and the Web

- HTTPS: Tunnel HTTP over SSL/TLS
- Add golden lock symbol 



The verifier matters

- SSL is an *application layer protocol*
  - Software developers must use it correctly

- Pre-Smartphone World
  - Small set of applications that use SSL (E.g., Web Browser)
  - Lots of attention to those apps

- Smartphone World
  - Possibly *millions of applications that use SSL*
  - Many apps do not verify certificates correctly – **Implications?**
  - Developers change default configuration – **WHY?**

SSL Verification in Apps

- Even popular apps are vulnerable to incorrect SSL use
  - Banking
  - Document storage
  - Social Networks (Facebook, before *Firesheep*)
  - ...
- Common mistakes: Generally, in HTTPS use.
  1. Not using SSL
  2. Mixed SSL use
  3. Accepting all certificates
  4. Accepting all hostnames (i.e., regardless of the CN)
  5. Trusting all CAs

Not using SSL

- What happens when you don't use SSL? E.g., `http://www.mybank.com/loggedin?sessionid=11`
  - If I can *guess*, *infer*, or *steal* the session ID, game over
- Are there any use cases where not using SSL would be okay?
  - It depends. However, unless confidentiality and authenticity are *never* going to be important to the app, use SSL!

Lesson 1: Always use SSL (i.e., mostly HTTPS)

Mixed SSL use

- Mixed use of HTTP and HTTPS on the same site.
- **Use case 1:** Login page is not HTTPS, but the login form is submitted to a HTTPS page.
  - MiTM can *replace HTTPS links with HTTP* (i.e., SSL Stripping)
- **Use case 2:** Login page is HTTPS, but the rest of the website may be HTTP
  - *Unencrypted cookies/session IDs!* (e.g., Firesheep)

Lesson 2: Use HTTPS throughout

Certificate Validation

- Apps can override the *TrustManager* interface

69

```
SSLContext sslContext = SSLContext.getInstance("SSL");

// set up a TrustManager that trusts everything
sslContext.init(null, new TrustManager[] { new X509TrustManager() {
            public X509Certificate[] getAcceptedIssuers() {
                    System.out.println("getAcceptedIssuers =============");
                    return null;
            }

            public void checkServerTrusted(X509Certificate[] certs,
                            String authType) {
                    System.out.println("checkServerTrusted =============");
            }
} }, new SecureRandom());
```

https://stackoverflow.com/questions/2703161/how-to-ignore-ssl-certificate-errors-in-apache-httpclient-4-0

- What is wrong with this example? It accepts all server certificates!

Lesson 3: Always validate the server's certificate

Using self-signed certificates

- *The right way:* Certificate Pinning
  - i.e., hardcode your self-signed certificate.

```
CertificateFactory cf = CertificateFactory.getInstance("X.509");
// From https://www.washington.edu/itconnect/security/ca/load-der.crt
InputStream caInput = new BufferedInputStream(new FileInputStream("load-der.crt"));
Certificate ca;
try {
    ca = cf.generateCertificate(caInput);
    System.out.println("ca=" + ((X509Certificate) ca).getSubjectDN());
} finally {
    caInput.close();
}
```

Step 1: Read in your certificate

Step 2: Create custom TrustManager

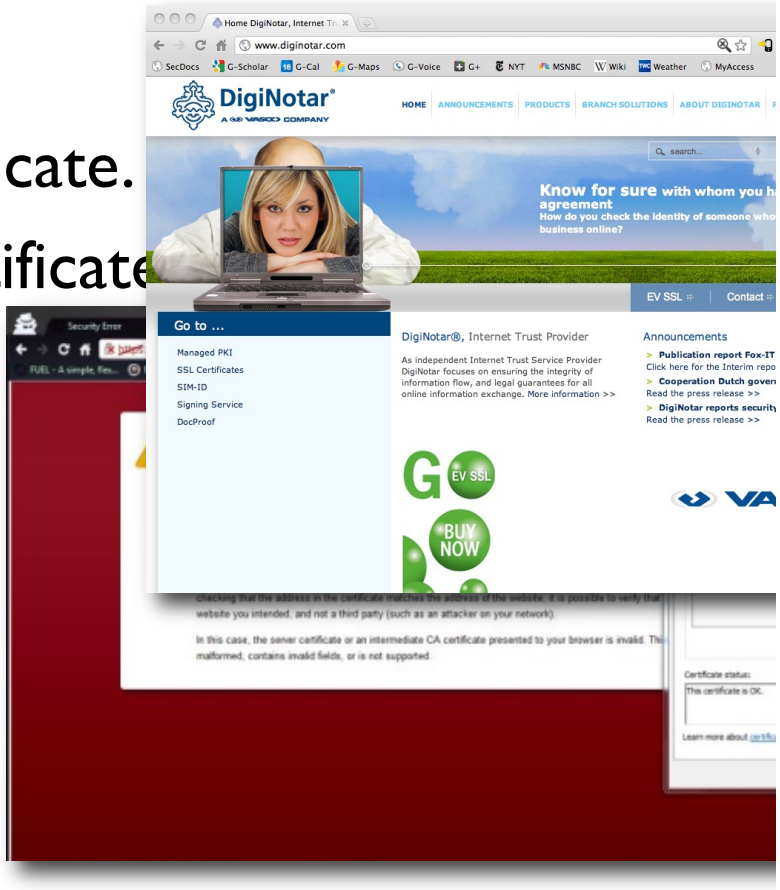
```
// Create a KeyStore containing our trusted CAs
String keyStoreType = KeyStore.getDefaultType();
KeyStore keyStore = KeyStore.getInstance(keyStoreType);
keyStore.load(null, null);
keyStore.setCertificateEntry("ca", ca);

// Create a TrustManager that trusts the CAs in our KeyStore
String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm();
TrustManagerFactory tmf = TrustManagerFactory.getInstance(tmfAlgorithm);
tmf.init(keyStore);

// Create an SSLContext that uses our TrustManager
SSLContext context = SSLContext.getInstance("TLS");
context.init(null, tmf.getTrustManagers(), null);
```

Using self-signed certificates

- *The right way:* Certificate Pinning
  - i.e., hardcode your self-signed certificate.
  - Allows *secure* use of self-signed certificate
- Variation:
  - Pinning own CA certificate
  - Gives you more flexibility.
- How to change the certificate?
  - App updates!
- Don't have to trust 100s of Root CAs!



Lesson 4: Certificate pinning, if done correctly, is more secure than *default SSL use*.

Hostname Verification

- Back to basics: What does a certificate provide?
  - Binding between a *public key* and *identity*

```
HostnameVerifier hostnameVerifier = org.apache.http.conn.ssl.SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER;

DefaultHttpClient client = new DefaultHttpClient();

SchemeRegistry registry = new SchemeRegistry();
SSLSocketFactory socketFactory = SSLSocketFactory.getSocketFactory();
socketFactory.setHostnameVerifier((X509HostnameVerifier) hostnameVerifier);
registry.register(new Scheme("https", socketFactory, 443));
SingleClientConnManager mgr = new SingleClientConnManager(client.getParams(), registry);
DefaultHttpClient httpClient = new DefaultHttpClient(mgr, client.getParams());

// Set verifier
HttpsURLConnection.setDefaultHostnameVerifier(hostnameVerifier);
```

https://stackoverflow.com/questions/2012497/accepting-a-certificate-for-https-on-android?lq=1

- Any certificate issued by any trusted CA will be accepted!
  - i.e., HostName= google.com, but cert has CN=foogle.com? ✓

Lesson 5: Never override the HostNameVerifier

The End