



WILLIAM & MARY

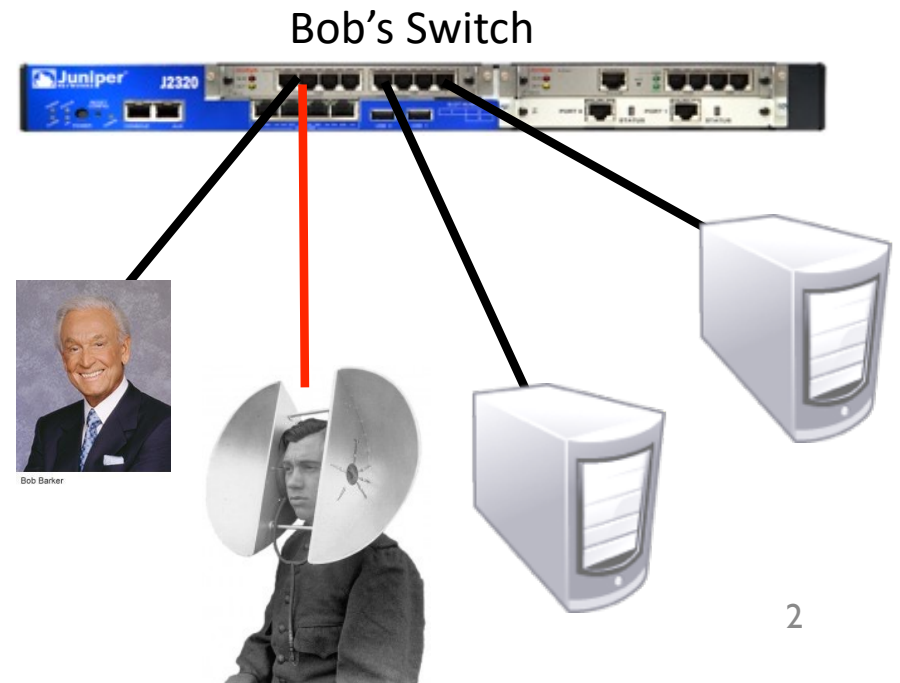
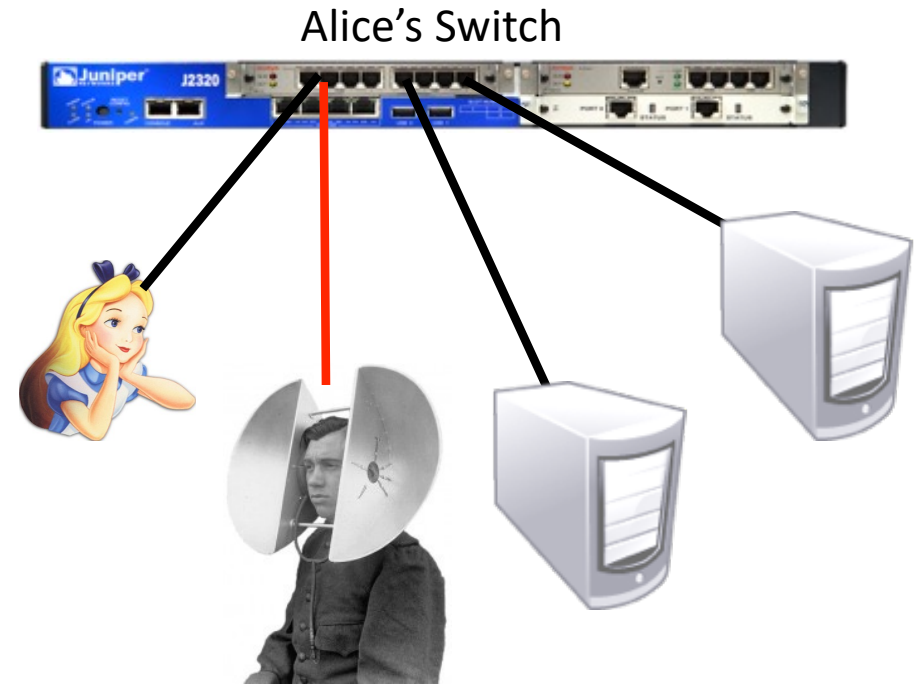
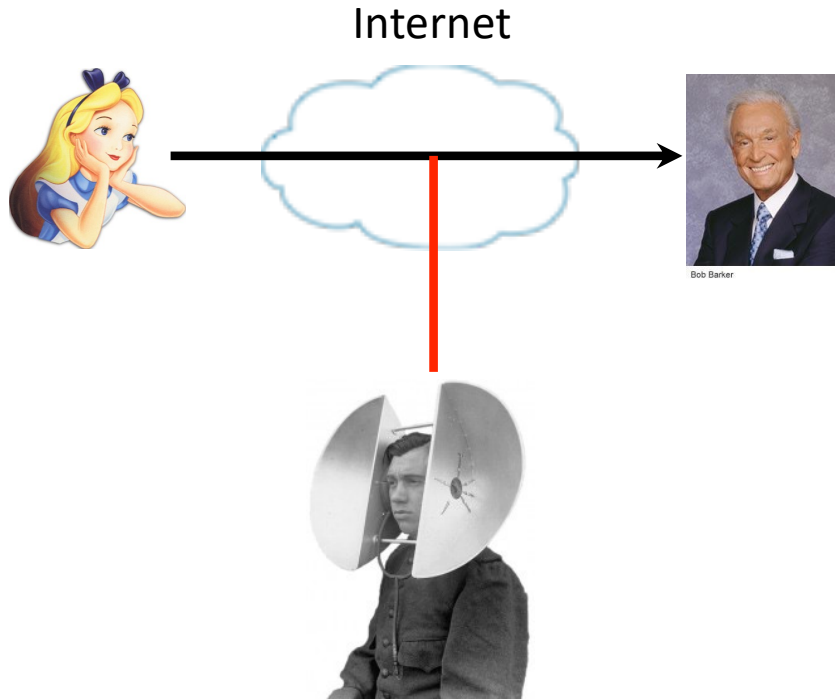
CHARTERED 1693

# CSCI 445: Mobile Application Security

Lecture 6

Prof. Adwait Nadkarni

# Eavesdropping



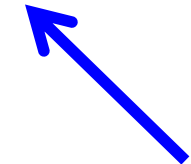
# Why is crypto useful?

```
MacBook-Pro-45:~ adwait$ echo "Security is Fun" | netcat -v localhost 8080  
localhost [127.0.0.1] 8080 (http-alt) open
```

The screenshot shows a Wireshark capture of a netcat connection. The terminal window at the top shows the command `echo "Security is Fun" | netcat -v localhost 8080` and the output `localhost [127.0.0.1] 8080 (http-alt) open`. The packet list pane shows several TCP packets, with frame 165 highlighted in red, indicating a RST (Reset) packet. The packet details pane shows the structure of the captured data: Ethernet II, Internet Protocol Version 4, and Transmission Control Protocol. The raw data pane shows the hex and ASCII representation of the captured data, which is the plaintext message "Security is Fun". A blue arrow points to the ASCII text in the raw data pane.

| No. | Time | Source    | Destination | Protocol | Length | Info   |
|-----|------|-----------|-------------|----------|--------|--|
| 160 | 2... | 127.0.0.1 | 127.0.0.1   | TCP      | 56     | 59584 → 8080 [ACK] Seq=1 Ack=1 Win=408288 Len=0 TSv... |
| 161 | 2... | 127.0.0.1 | 127.0.0.1   | TCP      | 56     | [TCP Window Update] 8080 → 59584 [ACK] Seq=1 Ack=1 ... |
| 162 | 2... | 127.0.0.1 | 127.0.0.1   | TCP      | 72     | 59584 → 8080 [PSH, ACK] Seq=1 Ack=1 Win=408288 Len=... |
| 163 | 2... | 127.0.0.1 | 127.0.0.1   | TCP      | 56     | 8080 → 59584 [ACK] Seq=1 Ack=17 Win=408256 Len=0 TS... |
| 164 | 2... | 127.0.0.1 | 127.0.0.1   | TCP      | 68     | 59585 → 19536 [SYN] Seq=0 Win=65535 Len=0 MSS=16344... |
| 165 | 2... | 127.0.0.1 | 127.0.0.1   | TCP      | 44     | 19536 → 59585 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0       |
| 166 | 2... | 127.0.0.1 | 127.0.0.1   | TCP      | 68     | 59586 → 19536 [SYN] Seq=0 Win=65535 Len=0 MSS=16344... |
| 167 | 2... | 127.0.0.1 | 127.0.0.1   | TCP      | 44     | 19536 → 59586 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0       |
| 168 | 2... | 127.0.0.1 | 127.0.0.1   | TCP      | 68     | 59587 → 19536 [SYN] Seq=0 Win=65535 Len=0 MSS=16344... |
| 169 | 2... | 127.0.0.1 | 127.0.0.1   | TCP      | 44     | 19536 → 59587 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0       |
| 170 | 2... | 127.0.0.1 | 127.0.0.1   | TCP      | 68     | 59588 → 19536 [SYN, ECN, CWR] Seq=0 Win=65535 Len=0... |
| 171 | 2... | 127.0.0.1 | 127.0.0.1   | TCP      | 44     | 19536 → 59588 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0       |
| 172 | 2... | 127.0.0.1 | 127.0.0.1   | TCP      | 68     | 59589 → 19536 [SYN] Seq=0 Win=65535 Len=0 MSS=16344... |
| 173 | 2... | 127.0.0.1 | 127.0.0.1   | TCP      | 44     | 19536 → 59589 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0       |

```
0000 02 00 00 00 45 00 00 44 2a cb 40 00 40 06 00 00 ....E..D *.@.@...  
0010 7f 00 00 01 7f 00 00 01 e8 c0 1f 90 44 fd 6b e1 ..... ..D.k.  
0020 d9 cd 38 c7 80 18 31 d7 fe 38 00 00 01 01 08 0a ..8...1. .8.....  
0030 6a 50 15 48 6a 50 15 47 53 65 63 75 72 69 74 79 jP.HjP.G Security  
0040 20 69 73 20 46 75 6e 0a is Fun.
```



# Why is this bad?

- Its just an instant message, right?

*Alice* uses the Internet for:

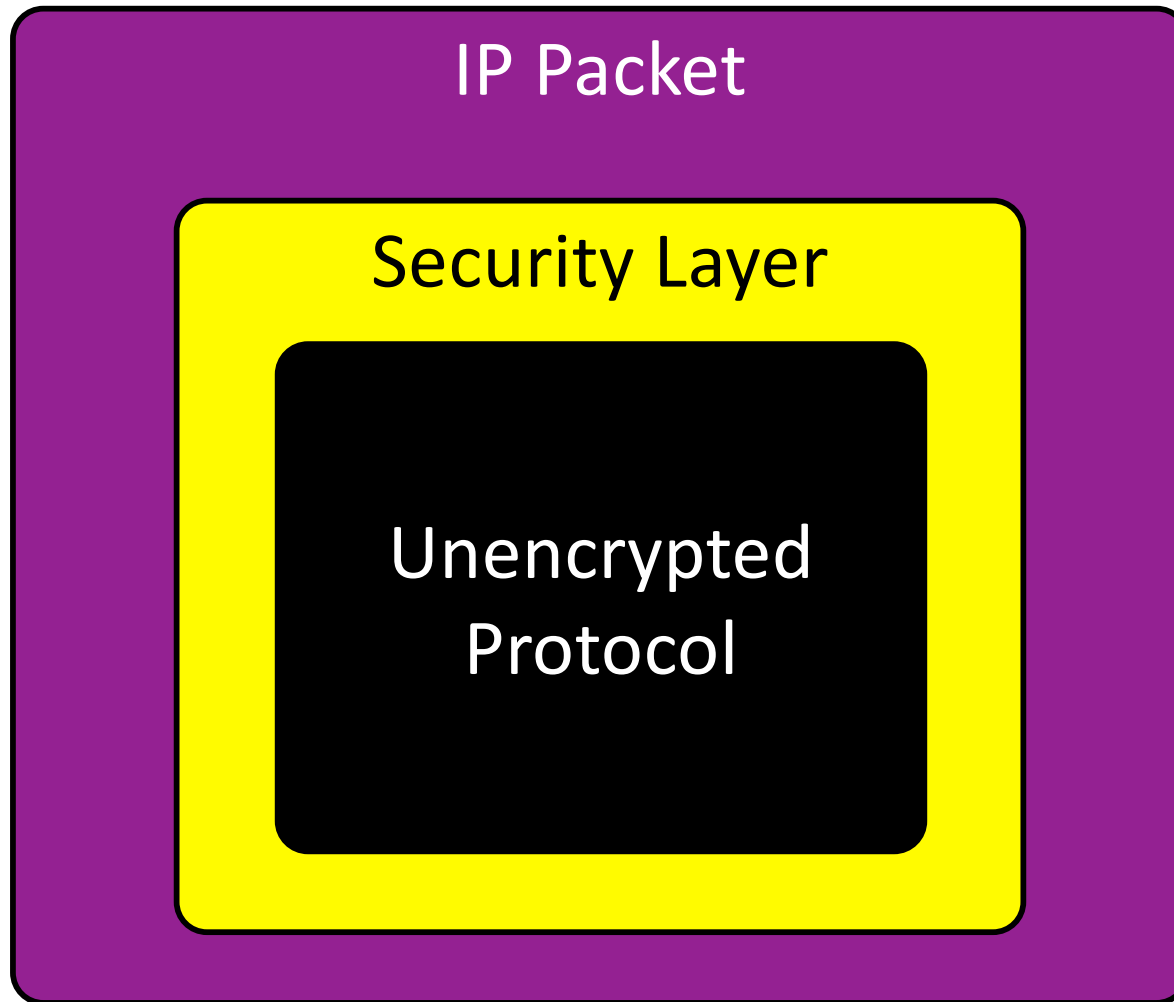
- **Email**
- **Banking**
- Online shopping
- Social networking
- ...

# Let's use that crypto stuff

- Let's build some new protocols
  - HTTP → SecureHTTP
  - POP → POPSecure
  - IMAP → CryIMAP
  - SMTP → SMTPS
  - FTP → FTPS
  - Jabber → SecJabber
  - Telnet → TelCryptNet

Let's build a  
crypto-wrapper  
standard instead





# 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.

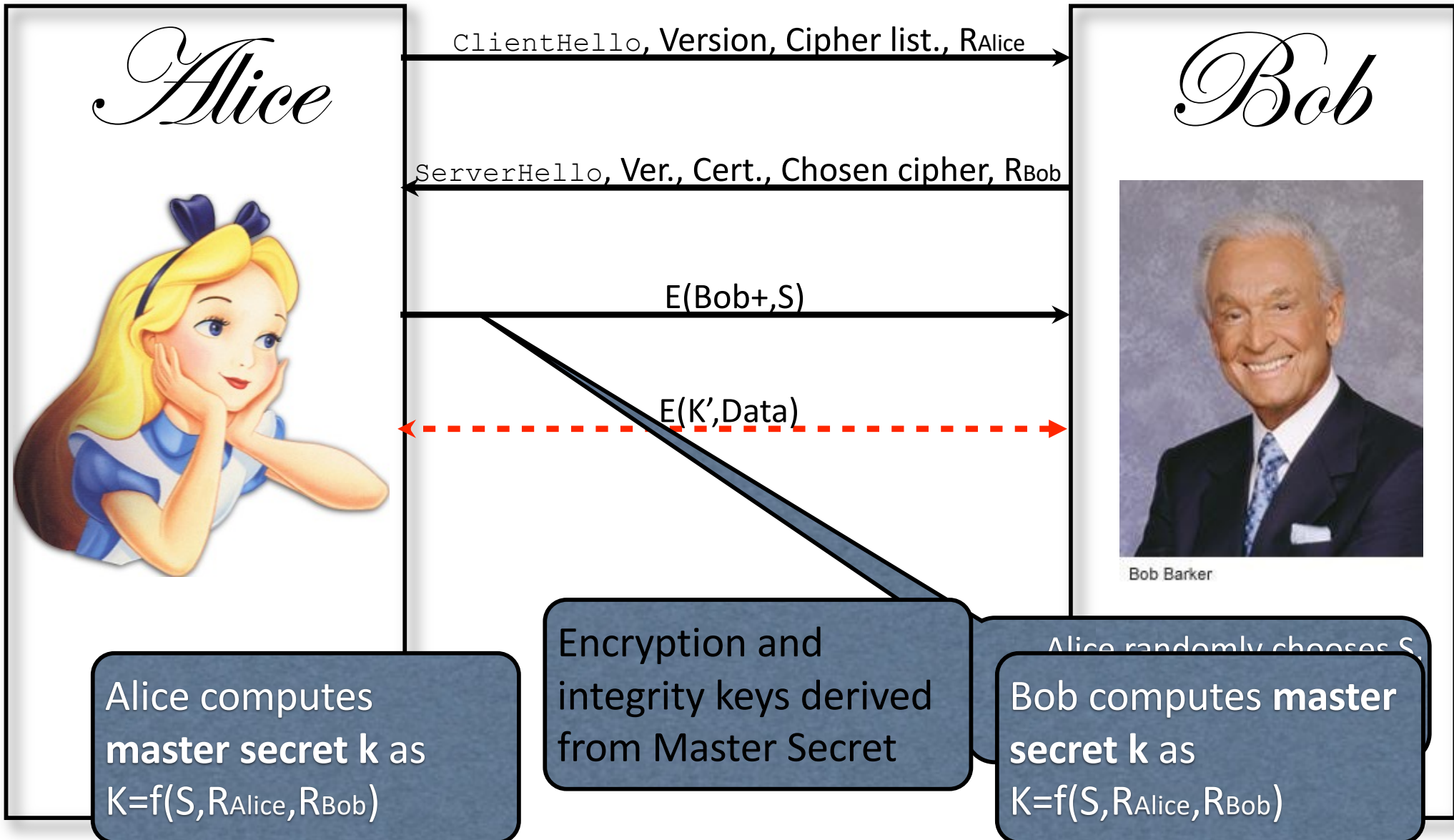
- Application-layer protocol
- Operates over TCP -- **WHY?**



# 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

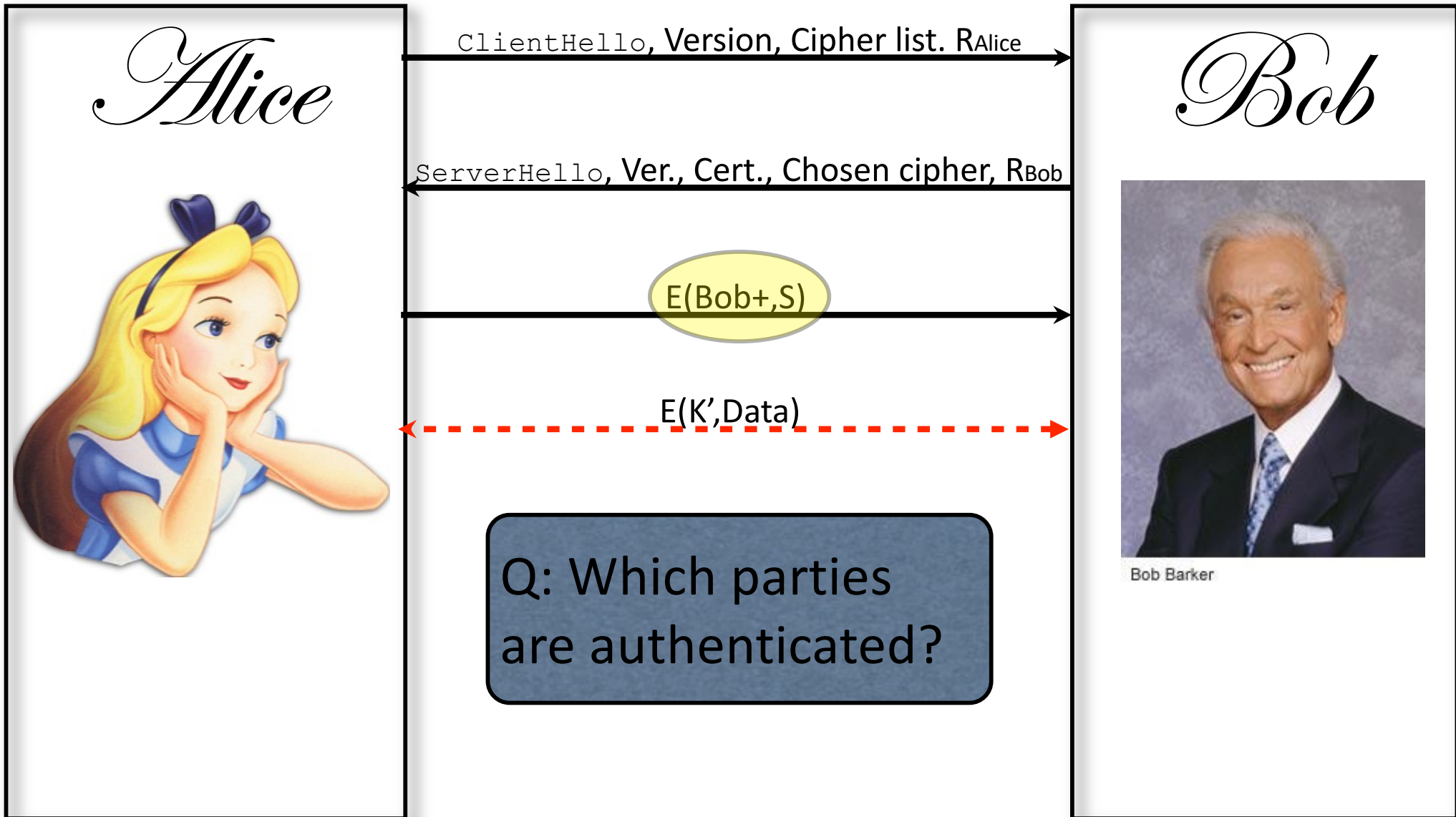
# SSLv2 Handshake



# Cryptographic Parameters

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

# Authentication



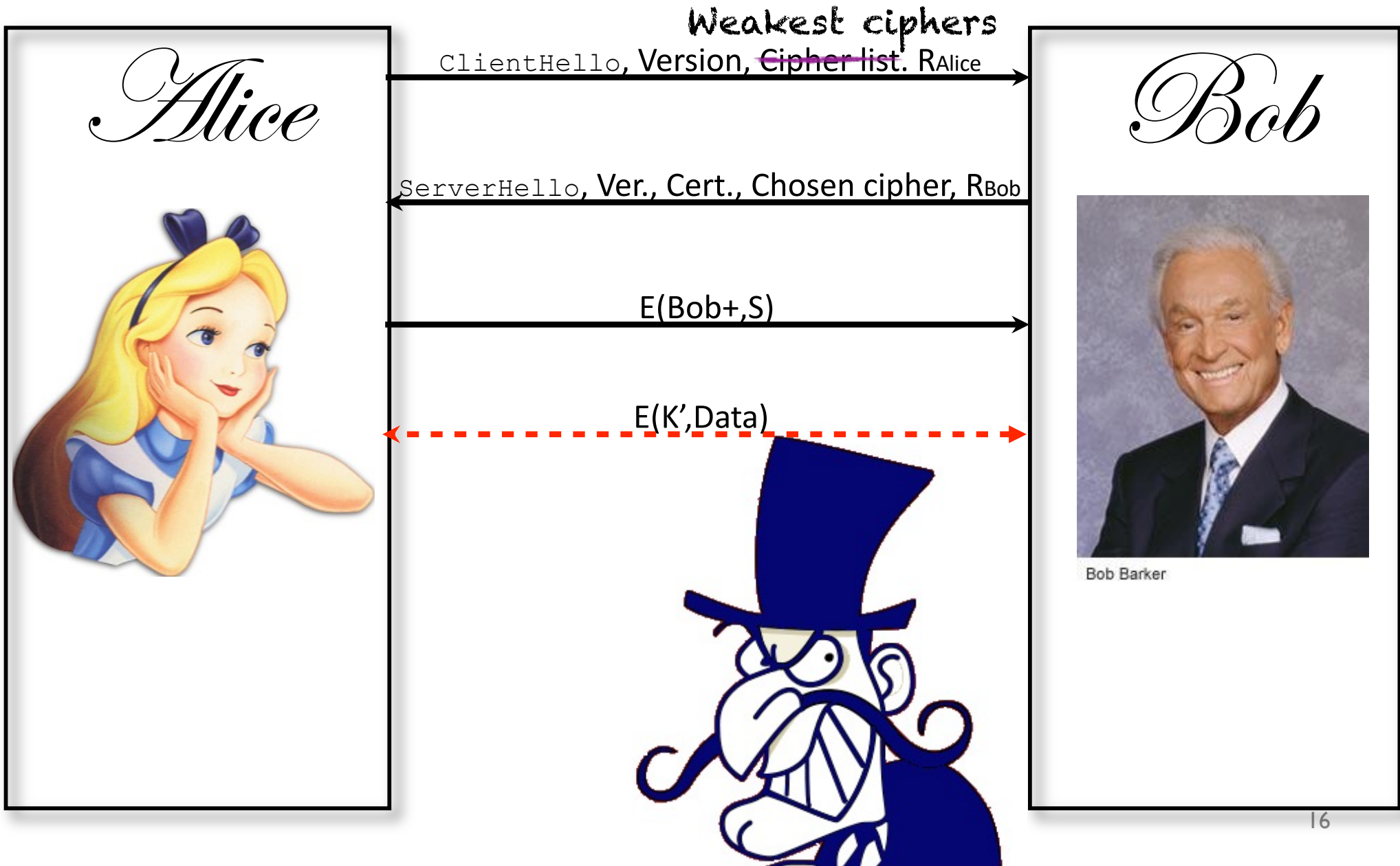
# Cipher Suites

- 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
```

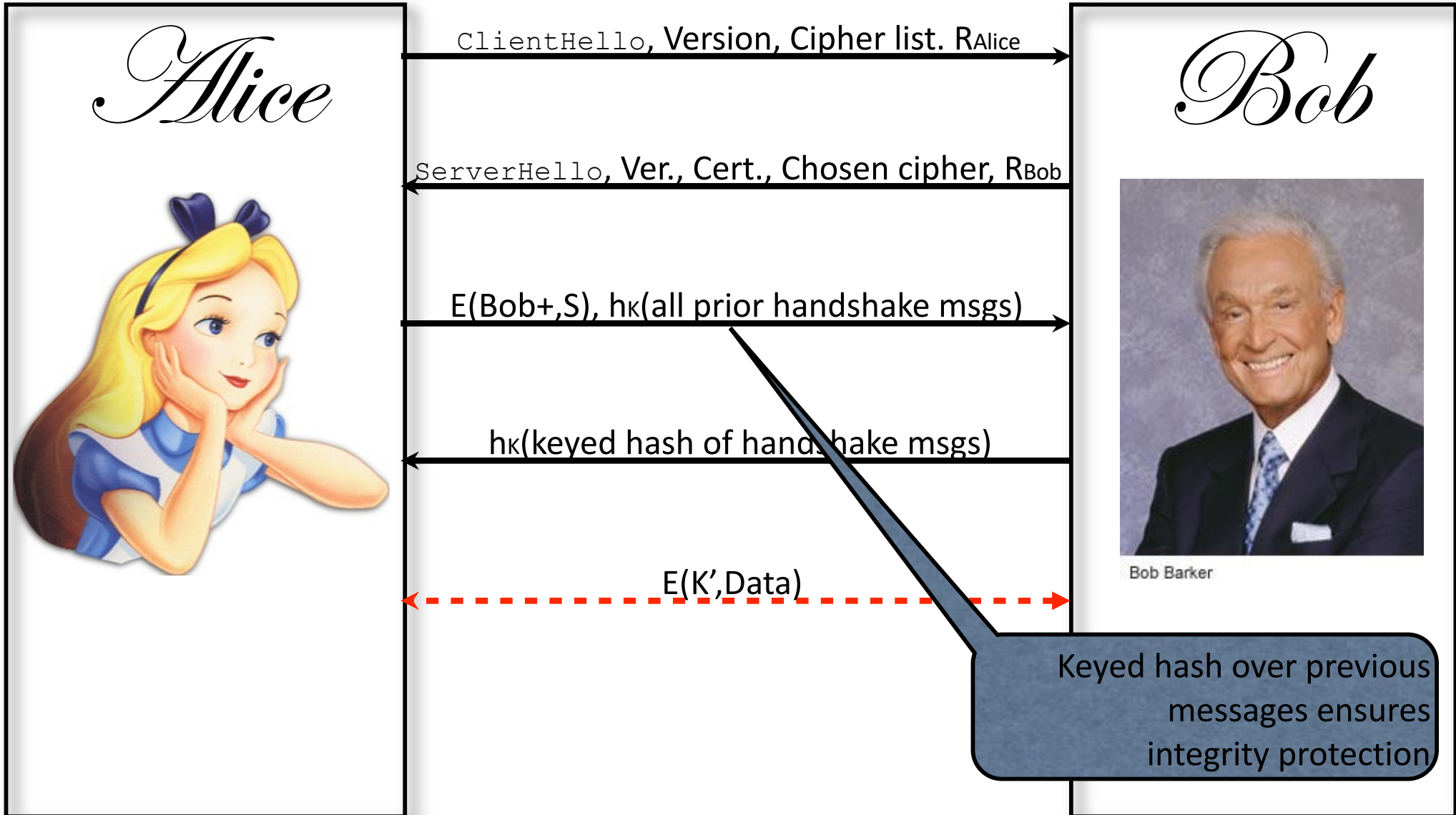


# SSLv2 Problems

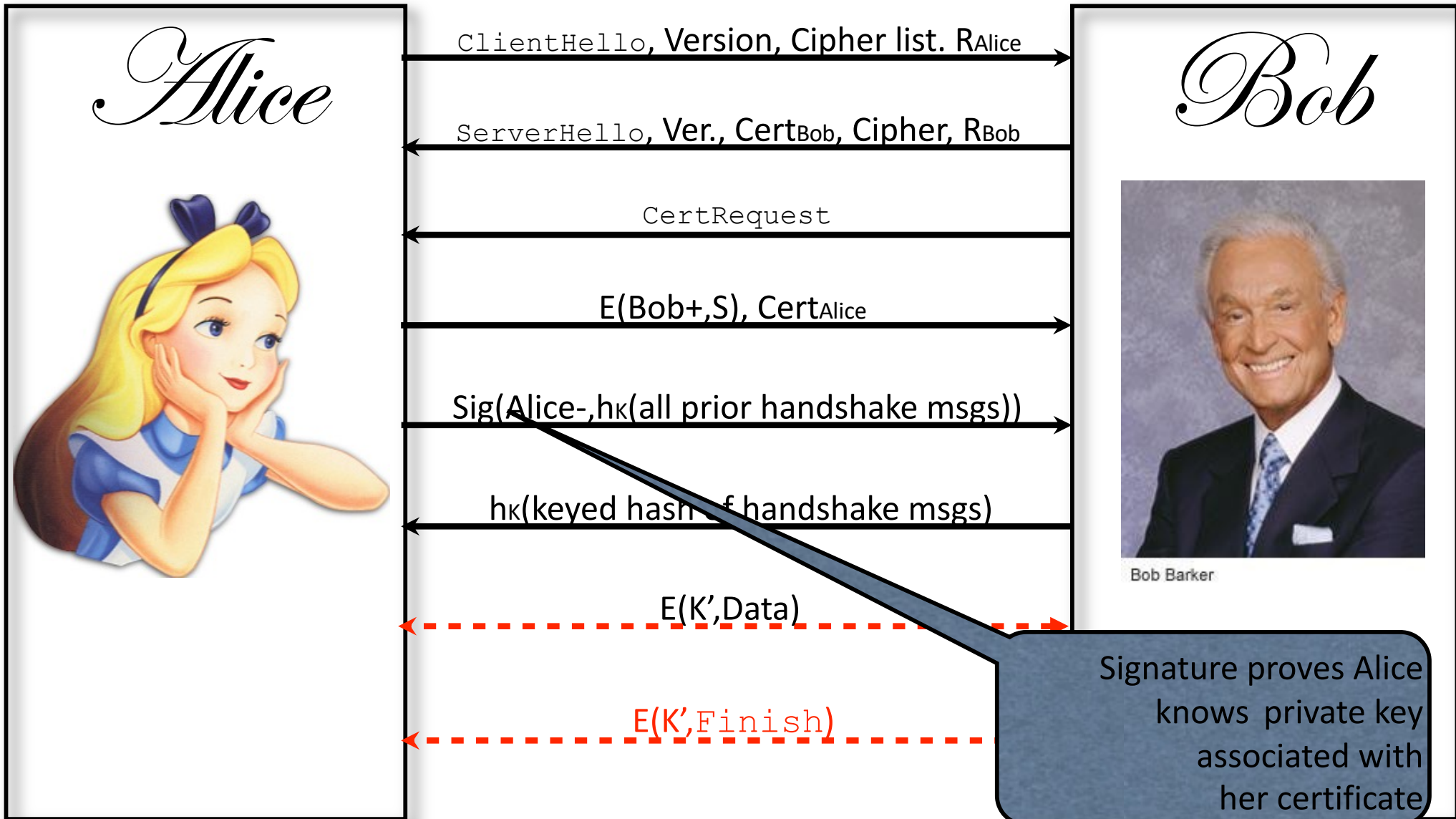




# SSLv3 Fixes

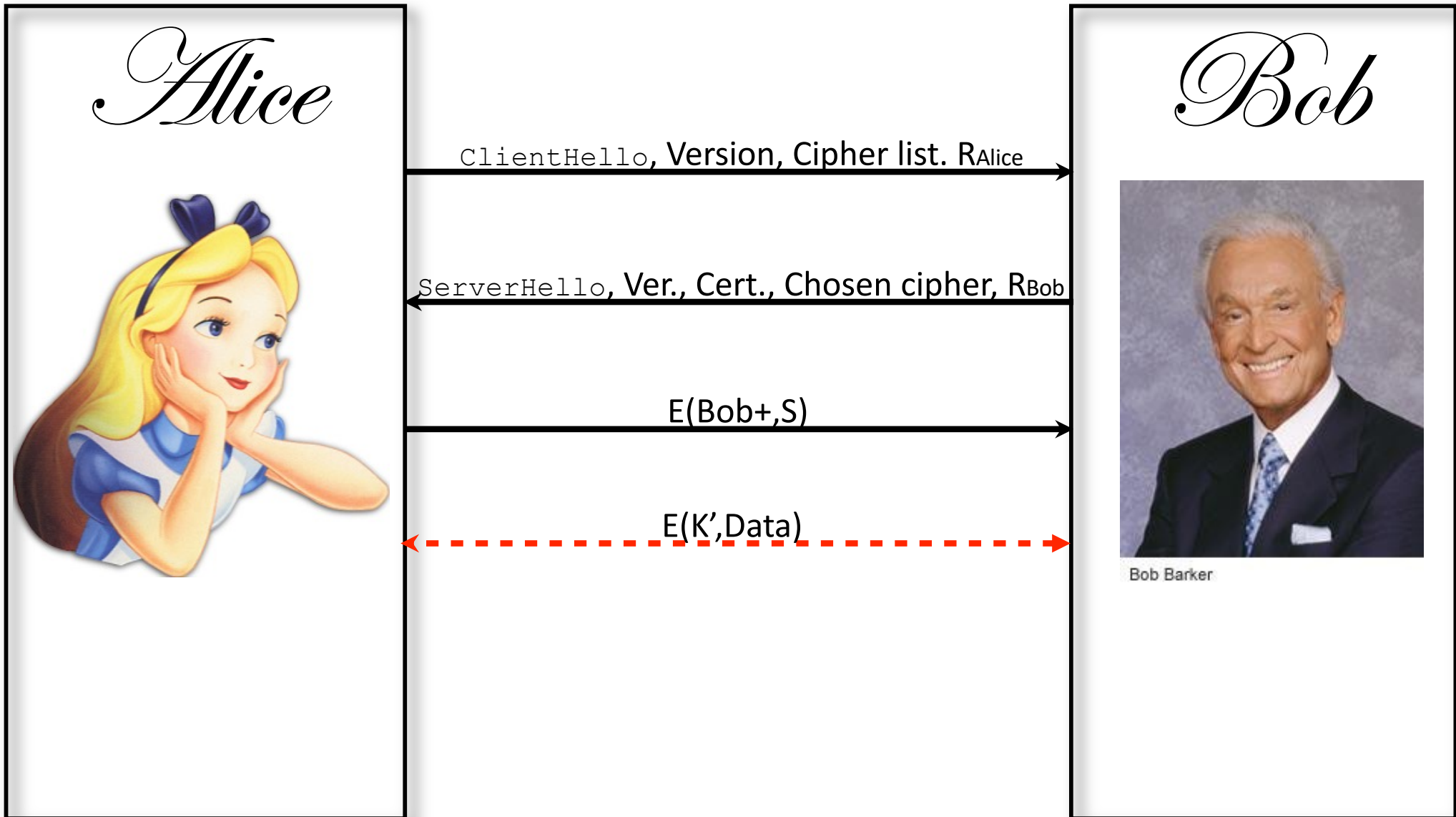


# SSL/TLS with Server and Client Authentication



# Problems with TLS/SSL

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



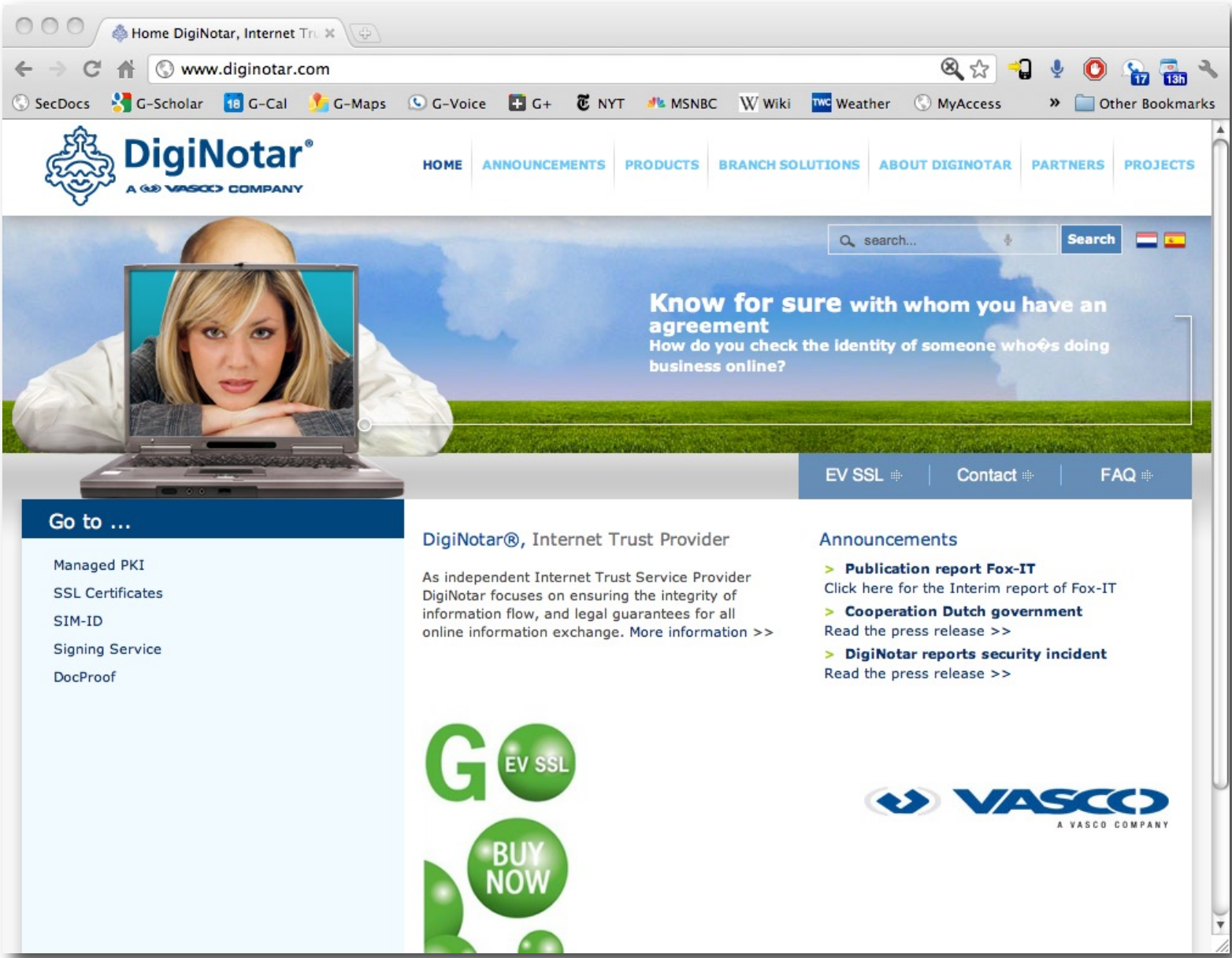
# Solution: Use a PKI

The screenshot shows the DigiNotar website homepage in a browser window. The browser's address bar displays 'www.diginotar.com'. The website features a navigation menu with links for HOME, ANNOUNCEMENTS, PRODUCTS, BRANCH SOLUTIONS, ABOUT DIGINOTAR, PARTNERS, and PROJECTS. A search bar is located in the top right corner. The main banner area contains a woman looking at a laptop and the text: 'Know for sure with whom you have an agreement. How do you check the identity of someone who's doing business online?'. Below the banner, there are links for EV SSL, Contact, and FAQ. A 'Go to ...' sidebar lists services: Managed PKI, SSL Certificates, SIM-ID, Signing Service, and DocProof. The main content area includes the heading 'DigiNotar®, Internet Trust Provider' and a paragraph: 'As independent Internet Trust Service Provider DigiNotar focuses on ensuring the integrity of information flow, and legal guarantees for all online information exchange. More information >>'. There are also 'Announcements' with links to 'Publication report Fox-IT', 'Cooperation Dutch government', and 'DigiNotar reports security incident'. At the bottom, there are 'GO EV SSL' and 'BUY NOW' buttons, and the VASCO logo with the text 'A VASCO COMPANY'.

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



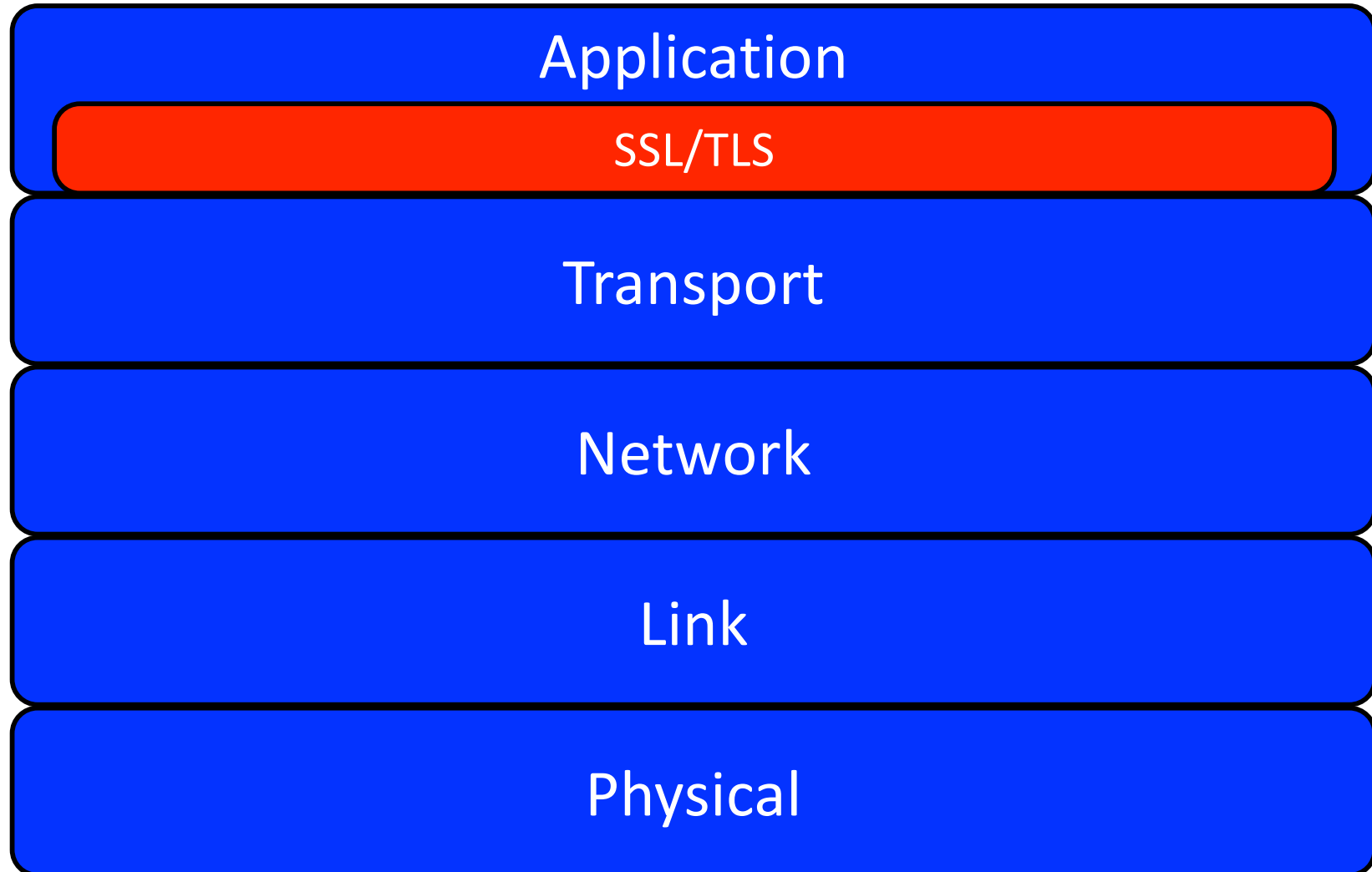
# Recall: The DigiNotar Incident



# SSL/TLS in the Real World



# Network Stack, revisited

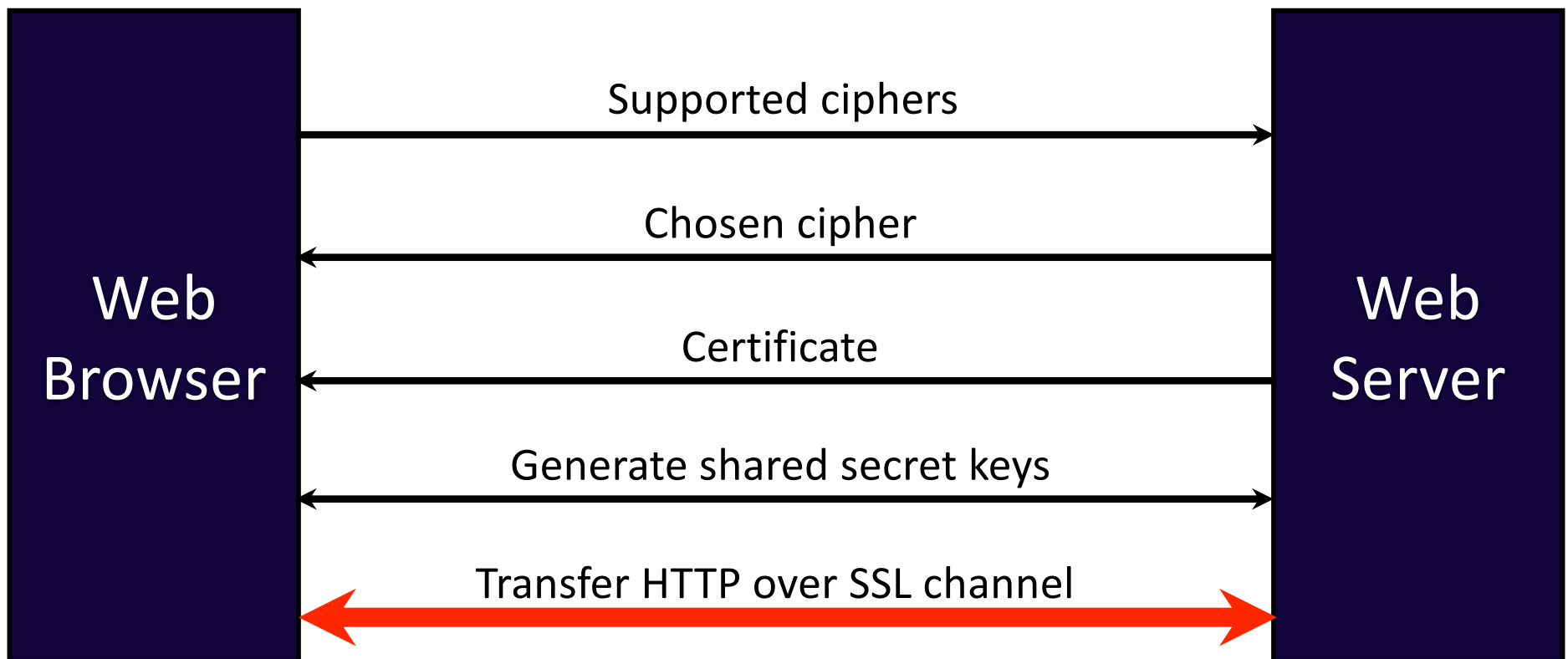


# SSL/TLS in the Real World

- 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 certificates
- 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