The TLS Protocol

Network Security

Threats on the Web

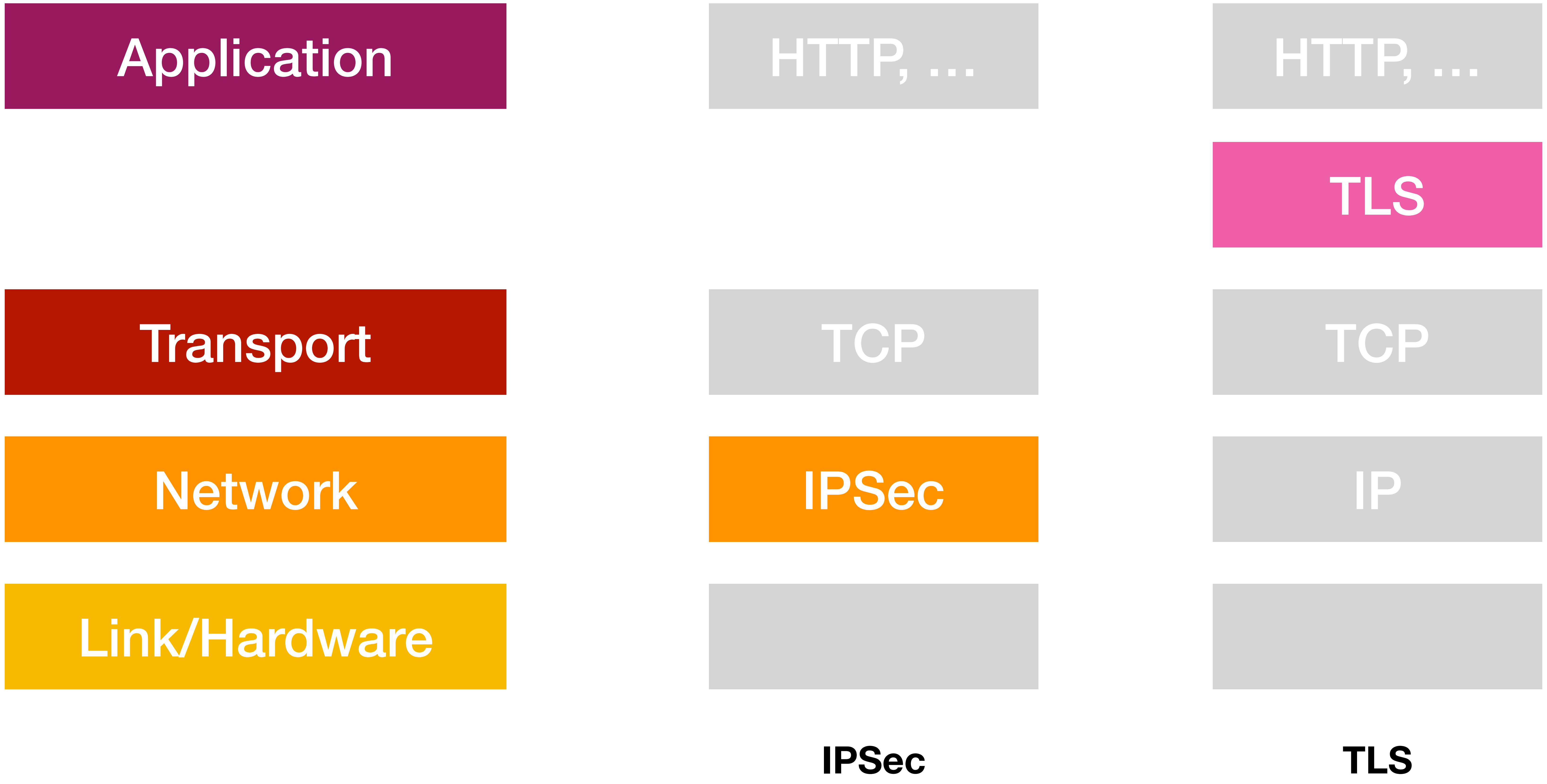
- Locations where threats occur:
  - Web server → **systems security**
  - Web browser → **systems security**
  - Network traffic between browser and server → **network security**
- Threats:
  - CIA(AA)
- Question: how to secure the network traffic?

The Internet Protocol Stack
(Simplified OSI)

Stuff that you write	Application
TCP or UDP	Transport
IP	Network
Ethernet or 802.11	Link/Hardware

Securing Network Traffic

Two solutions:

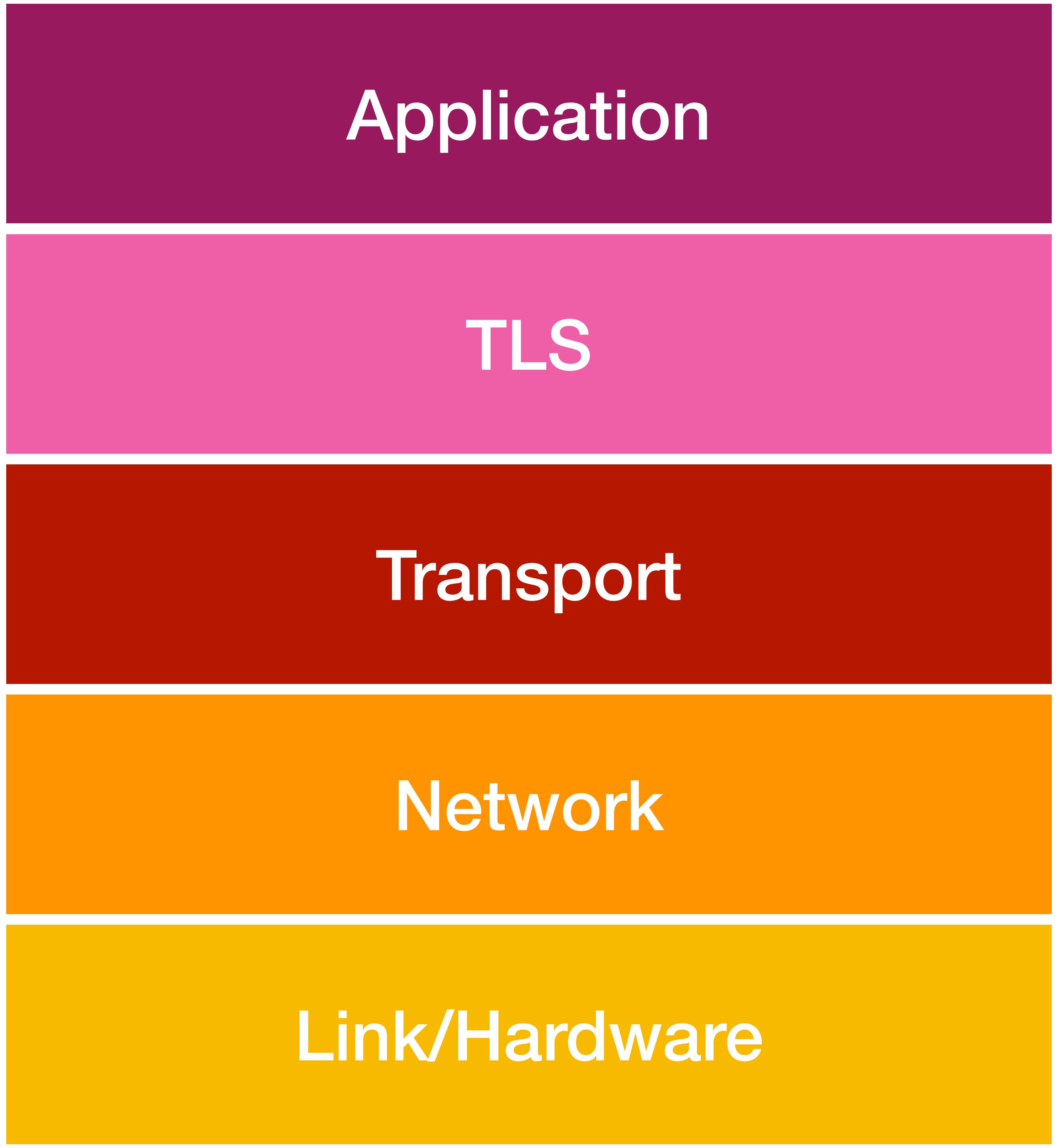


The Internet Protocol Stack with TLS

The TLS layer runs between the Application and Transport layer (Presentation layer).

The encryption is transparent to the Application layer.

Normal TCP and IP protocols etc. can be used at the low layers.



The SSL/TLS Protocol

- Used in many applications (e.g., email, instant messaging, voice over IP)
- Goal: **privacy/confidentiality**, **data integrity**, and **authentication** (at least authenticating the server)
- Most commonly known for **HTTPS** → securing web traffic

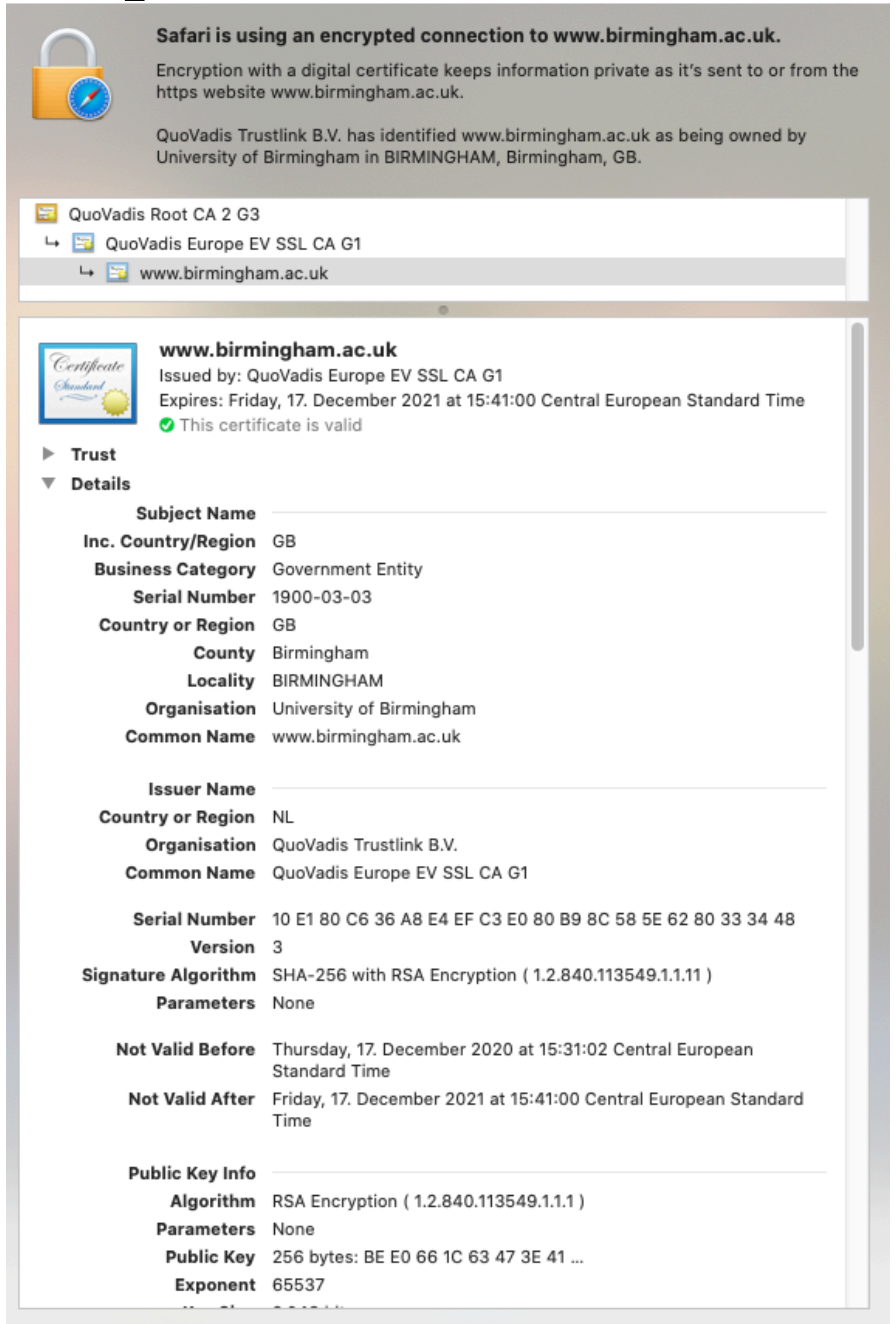
The SSL/TLS Protocol

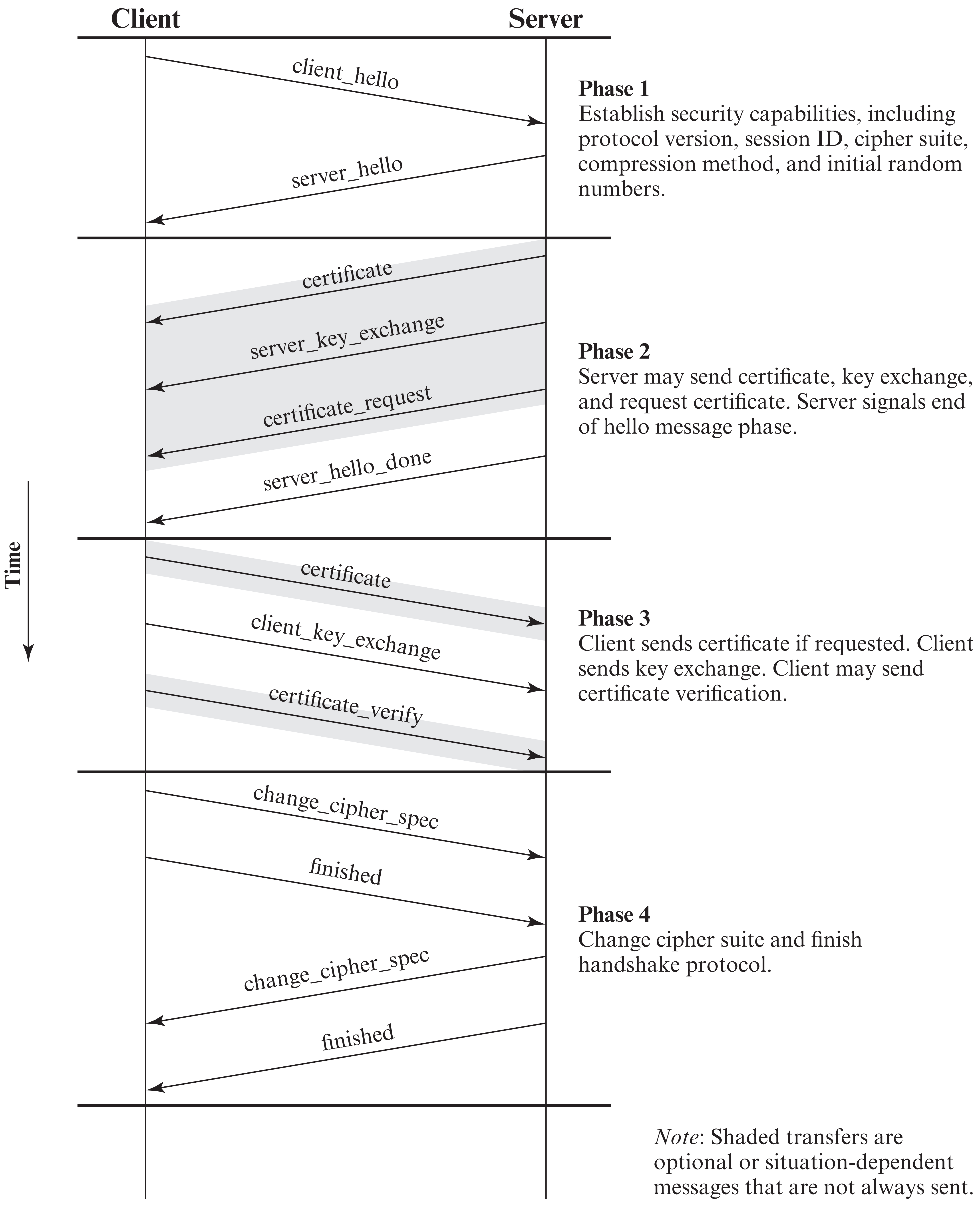
- The Secure Sockets Layer (SSL) protocol was proposed in 1995 and is the predecessor of the Transport Layer Security (TLS) protocol (first proposed in 1999).
- It provides encrypted socket communication and authentication, based on public keys.
- It may use a range of ciphers (RSA, AES, DES, DH,…)
  - These are negotiated at the start of the protocol (during handshake).
- Data integrity is ensured by an HMAC.

X.509 Standard for Certificates

- X.509 certificates contain a subject, subject’s public key, issuer name, etc.
- Certificates bind a public key to an entity.
- The issuer signs the hash of all the data.
- To check a certificate, all the data is hashed and checked against the issuers public key.
- If one has the issuer’s public key and trusts the issuer, one can be sure of the subject’s public key.
- Used for **authentication** in TLS.

Example Certificate





TLS handshake

- **Phase 1:** Establish parameters to use
- **Phase 2:** Server authenticates itself to client
- **Phase 3:** Client may authenticate identity to server
- **Phase 4:** Changing to agreed ciphers

Source: Cryptography and Network Security

(Deprecated) RSA Key Exchange

1. $C \rightarrow S : N_C$
2. $S \rightarrow C : N_S, Cert_S$
3. $C \rightarrow S : E_S(K_{\text{seed}}), \{Hash_1\}_{K_{CS}}$
4. $S \rightarrow C : \{Hash_2\}_{K_{CS}}$

$$Hash_1 = \#(N_C, N_S, E_S(K_{\text{seed}}))$$

$$Hash_2 = \#(N_C, N_S, E_S(K_{\text{seed}}), \{Hash_1\}_{K_{CS}})$$

K_{CS} is a session key based on $N_C, N_S, K_{\text{seed}}$

Assuming only server is authenticating via certificate

The RSA key exchange is not permitted anymore in TLS 1.3

Why? A compromised RSA key will compromise all handshakes.

All previous messages are hashed and then encrypted with K_{CS} for integrity.

TLS-DHE

A variant uses Diffie-Hellman for *forward secrecy* (DHE = ephemeral Diffie Hellman)

i.e., if someone gets the server's key later, they can't go back and break a recording of the traffic.

1. $C \to S : N_C$
2. $S \to C : N_S, g^x, Cert_S, Sign_S(\#(N_C, N_S, g^x))$
3. $C \to S : g^y, \{\#(\text{All previous messages})\}_{K_{CS}}$
4. $S \to C : \{\#(\text{All previous messages})\}_{K_{CS}}$

K_{CS} is a session key based on N_C, N_S, g^{xy}.

Cipher Suites

Cipher Suites with encryption and authentication:

```
SSL_RSA_WITH_3DES_EDE_CBC_SHA
SSL_RSA_WITH_DES_CBC_SHA
SSL_RSA_WITH_RC4_128_MD5
SSL_RSA_WITH_RC4_128_SHA
TLS_DHE_DSS_WITH_AES_128_CBC_SHA
TLS_DHE_DSS_WITH_AES_256_CBC_SHA
TLS_DHE_RSA_WITH_AES_128_CBC_SHA
TLS_DHE_RSA_WITH_AES_256_CBC_SHA
TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5
TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA
...
```

Cipher Suites with just authentication:

```
SSL_RSA_WITH_NULL_MD5
SSL_RSA_WITH_NULL_SHA
```

Cipher Suites with just encryption:

```
SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA
SSL_DH_anon_EXPORT_WITH_RC4_40_MD5
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
SSL_DH_anon_WITH_DES_CBC_SHA
SSL_DH_anon_WITH_RC4_128_MD5
TLS_DH_anon_WITH_AES_128_CBC_SHA
TLS_DH_anon_WITH_AES_256_CBC_SHA
```

Cipher Suites

Cipher Suites with encryption and authentication:

```
SSL_RSA_WITH_3DES_EDE_CBC
SSL_RSA_WITH_DES_CBC_SHA
SSL_RSA_WITH_RC4_128_MD5
SSL_RSA_WITH_RC4_128_SHA
TLS_DHE_DSS_WITH_AES_128_
TLS_DHE_DSS_WITH_AES_256_
TLS_DHE_RSA_WITH_AES_128_CBC_SHA
TLS_DHE_RSA_WITH_AES_256_CBC_SHA
TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5
TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA
...
```

In TLS 1.0–1.2 [edit] *Source: Wikipedia*

Algorithms supported in TLS 1.0–1.2 cipher suites

Key exchange/agreement	Authentication	Block/stream ciphers	Message authentication
RSA	RSA	RC4	Hash-based MD5
Diffie–Hellman	DSA	Triple DES	SHA hash function
ECDH	ECDSA	AES	
SRP		IDEA	
PSK		DES	
		Camellia	
		ChaCha20	

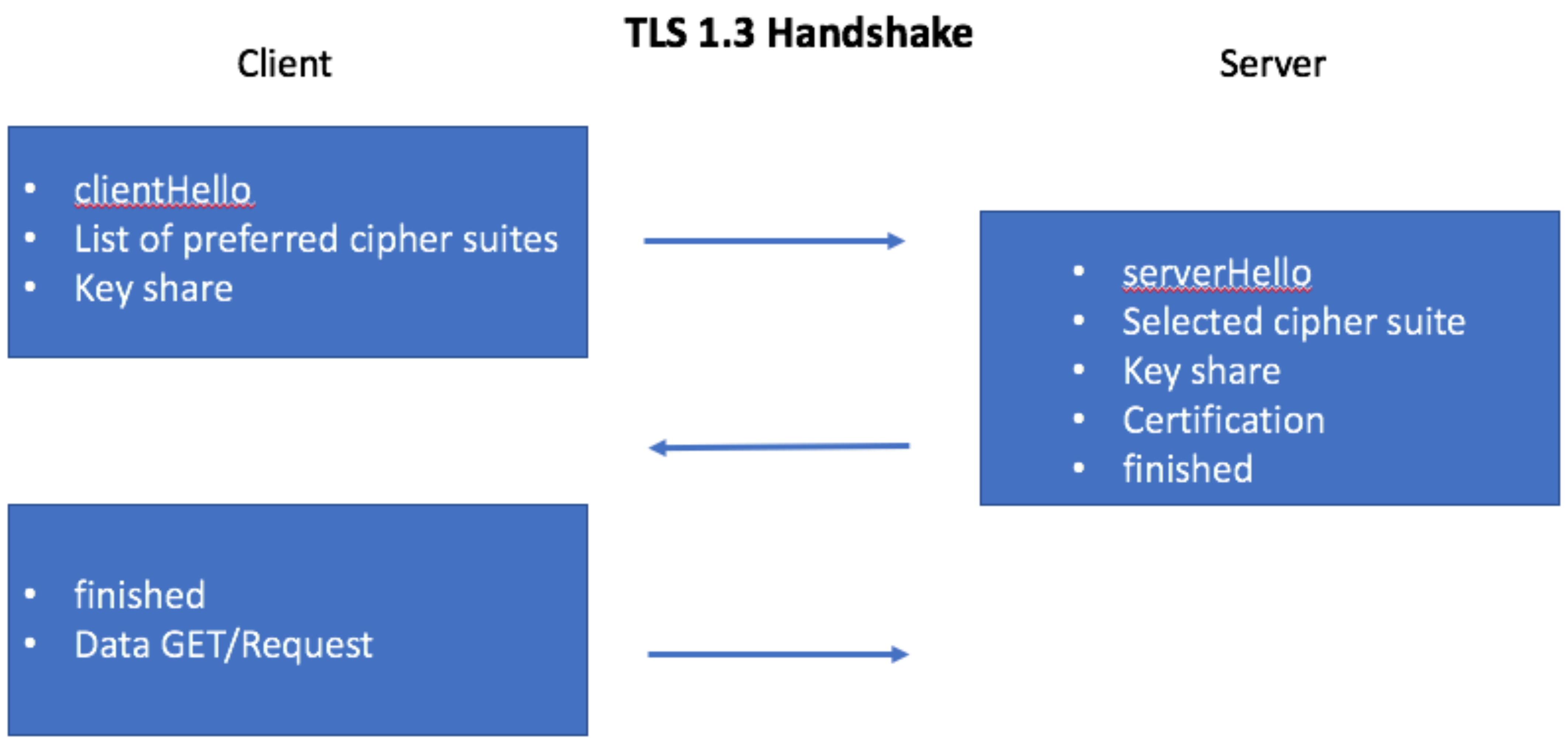
Cipher Suites with just authentication:

```
H_NULL_MD5
H_NULL_SHA
```

tes with just
:

```
SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA
SSL_DH_anon_EXPORT_WITH_RC4_40_MD5
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
SSL_DH_anon_WITH_DES_CBC_SHA
SSL_DH_anon_WITH_RC4_128_MD5
TLS_DH_anon_WITH_AES_128_CBC_SHA
TLS_DH_anon_WITH_AES_256_CBC_SHA
```

Cipher Suites Handshake



https://en.wikipedia.org/wiki/Cipher_suite

TLS Demo

- Websites
- Wireshark
- Using TLS in Java

Weaknesses in TLS

- Configuration weaknesses:
    - Cipher downgrading
    - Self-signed certificates
- Direct attack against implementations:
    - Apple’s goto fail bug
    - LogJam attack
    - HeartBleed

Cipher downgrading attack

Client supports:

```
TLS_DHE_DSS_WITH_AES_256_CBC_SHA
TLS_DHE_DSS_WITH_AES_128_CBC_SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA
SSL_RSA_WITH_DES_CBC_SHA
```

Server supports:

```
TLS_DHE_DSS_WITH_AES_128_CBC_SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA
SSL_RSA_WITH_DES_CBC_SHA
TLS_DHE_RSA_WITH_AES_256_CBC_SHA
TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5
TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA
```

Ciphers are listed in the order of preference.
What cipher will be used?

BUT...

Cipher
downgrading attack

Client supports:

~~TLS_DHE_DSS_WITH_AES_256_CBC_SHA~~
~~TLS_DHE_DSS_WITH_AES_128_CBC_SHA~~
~~SSL_RSA_WITH_3DES_EDE_CBC_SHA~~
SSL_RSA_WITH_DES_CBC_SHA

Server supports:

~~TLS_DHE_DSS_WITH_AES_128_CBC_SHA~~
~~SSL_RSA_WITH_3DES_EDE_CBC_SHA~~
SSL_RSA_WITH_DES_CBC_SHA
TLS_DHE_RSA_WITH_AES_256_CBC_SHA
TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5
TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA

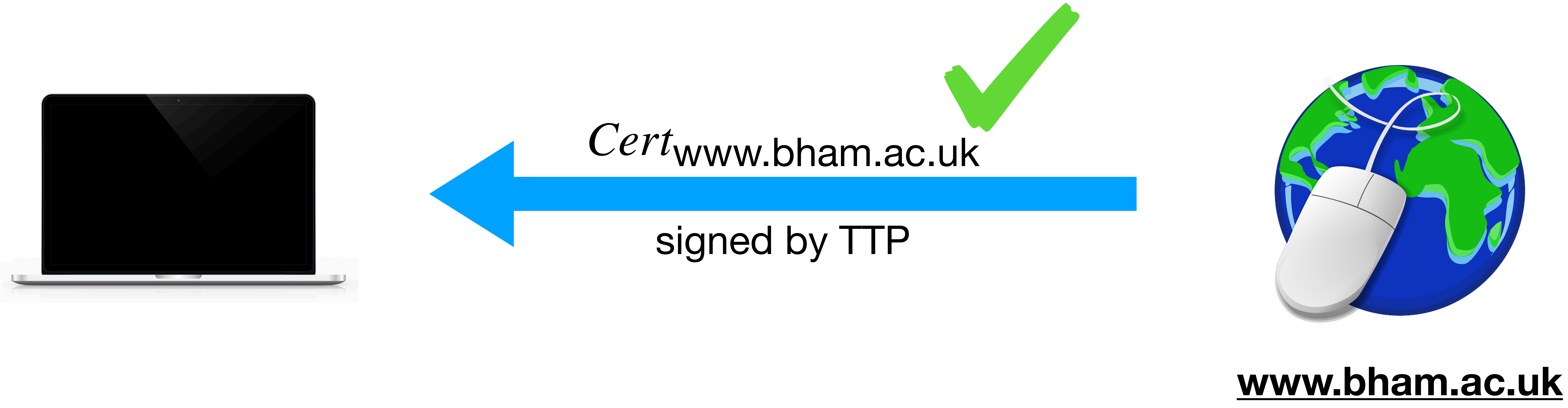
The cipher suite messages are not authenticated!

An attacker that owns the network can remove strong ciphers.

If both client and server support a weak cipher, then an attack can force its use.

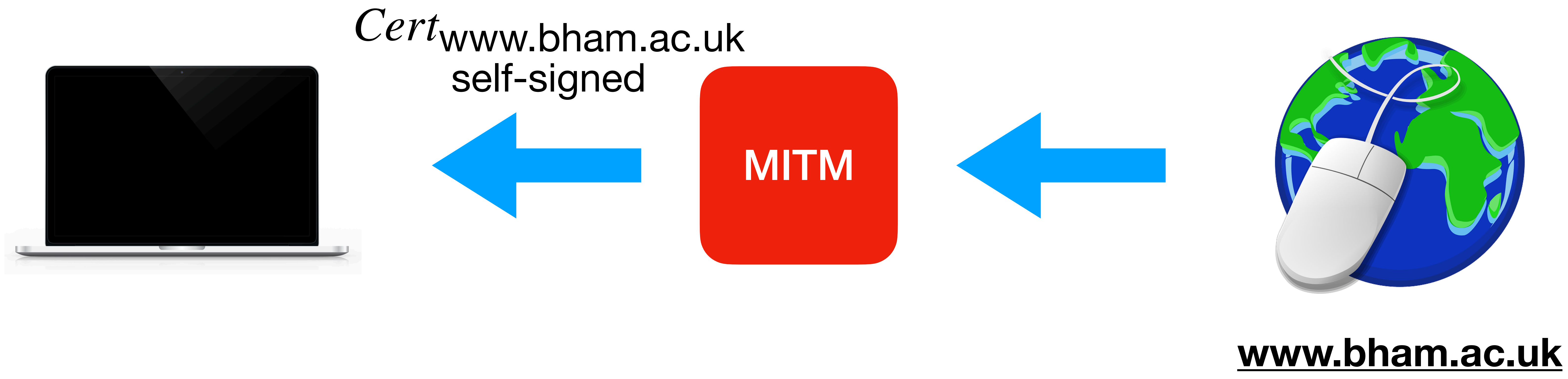
Self-signed Certificates

- Maintaining a set of certificates is hard (especially on apps and IoT devices).
- It’s much easier just to accept any certificate (or certificates that sign themselves).
- What’s the problem?



Self-signed Certificates

- Maintaining a set of certificates is hard (especially on apps and IoT devices).
- It’s much easier just to accept any certificate (or certificates that sign themselves).
- What’s the problem?



Self-signed Certificates

- Maintaining a set of certificates is hard
  (especially on apps and IoT devices).

- It’s much easier just to accept any certificate
  (or certificates that sign themselves).

- If the client accepts the self-signed certificates, then it’s easy to machine-in-the-middle.

- This has been shown to happen a lot in devices and code that use TLS!

Apple's Implementation of TLS

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
    OSStatus        err;
    SSLBuffer       hashOut, hashCtx, clientRandom, serverRandom;
    uint8_t         hashes[SSL_SHA1_DIGEST_LEN + SSL_MD5_DIGEST_LEN];
    SSLBuffer       signedHashes;
    uint8_t            *dataToSign;
     size_t            dataToSignLen;

     signedHashes.data = 0;
    hashCtx.data = 0;

    clientRandom.data = ctx->clientRandom;
    clientRandom.length = SSL_CLIENT_SRVR_RAND_SIZE;
    serverRandom.data = ctx->serverRandom;
    serverRandom.length = SSL_CLIENT_SRVR_RAND_SIZE;


     if(isRsa) {
          /* skip this if signing with DSA */
          dataToSign = hashes;
          dataToSignLen = SSL_SHA1_DIGEST_LEN + SSL_MD5_DIGEST_LEN;
          hashOut.data = hashes;
          hashOut.length = SSL_MD5_DIGEST_LEN;

          if ((err = ReadyHash(&SSLHashMD5, &hashCtx)) != 0)
              goto fail;
          if ((err = SSLHashMD5.update(&hashCtx, &clientRandom)) != 0)
              goto fail;
          if ((err = SSLHashMD5.update(&hashCtx, &serverRandom)) != 0)
              goto fail;
          if ((err = SSLHashMD5.update(&hashCtx, &signedParams)) != 0)
              goto fail;
          if ((err = SSLHashMD5.final(&hashCtx, &hashOut)) != 0)
              goto fail;
     }
     else {
          /* DSA, ECDSA - just use the SHA1 hash */
          dataToSign = &hashes[SSL_MD5_DIGEST_LEN];
          dataToSignLen = SSL_SHA1_DIGEST_LEN;
     }
```

```
     hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
    hashOut.length = SSL_SHA1_DIGEST_LEN;
    if ((err = SSLFreeBuffer(&hashCtx)) != 0)
        goto fail;

    if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;

     err = sslRawVerify(ctx,
                        ctx->peerPubKey,
                        dataToSign,                /* plaintext */
                        dataToSignLen,             /* plaintext length */
                        signature,
                        signatureLen);
     if(err) {
         sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
                     "returned %d\n", (int)err);
         goto fail;
     }

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;

}
```

This vulnerability was found and fixed in 2014.

http://opensource.apple.com/source/Security/Security-55471/libsecurity_ssl/lib/sslKeyExchange.c

Apple's TLS-DHE

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
  goto fail;
  goto fail;
... other checks ...
fail:
  ... buffer frees (cleanups) ...
  return err;
```

1. $C \rightarrow S : N_C$

2. $S \rightarrow C : N_S, g^x, Cert_S, Sign_S(\#(N_C, N_S, g^x))$

3. $C \rightarrow S : g^y, \{\#(\text{All previous messages})\}_{K_{CS}}$

4. $S \rightarrow C : \{\#(\text{All previous messages})\}_{K_{CS}}$

K_{CS} is a session key based on N_C, N_S, g^{xy}.

Apple's
TLS-DHE

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
  goto fail;
  goto fail;
... other checks ...
fail:
  ... buffer frees (cleanups) ...
  return err;
```

1. $C \rightarrow S : N_C$

2. $S \rightarrow C : N_S, g^x, Cert_S, True$

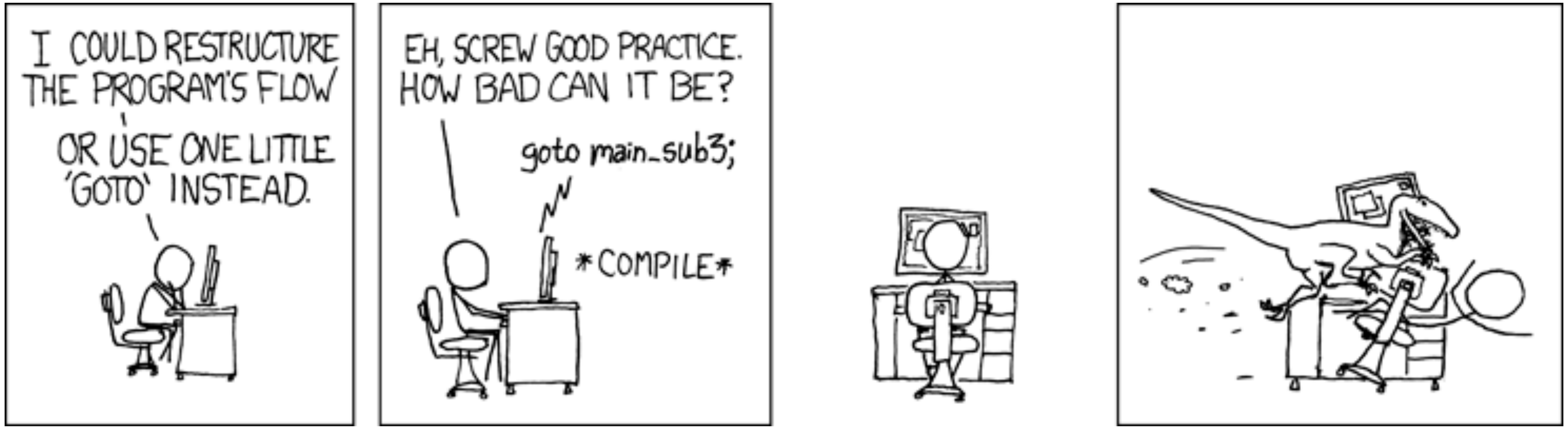
3. $C \rightarrow S : g^y, \{\#(\text{All previous messages})\}_{K_{CS}}$

4. $S \rightarrow C : \{\#(\text{All previous messages})\}_{K_{CS}}$

K_{CS} is a session key based on N_C, N_S, g^{xy}.

Major issues

- iOS fixed days before macOS!
- Why didn't tests pick this up?
- Compiler should have warned of unreachable code.
- Bad programming style: no brackets, goto:



http://xkcd.com/292/

Cipher Suites

- What if one side supports a weak cipher suite but the other does not?
- Generally considered safe.
- Browser developers removed all weak ciphers, some remained in servers.
- This depends on different cipher suites being incompatible, e.g.:
  `SSL_RSA_WITH_DES_CBC_SHA` and
  `TLS_DHE_DSS_WITH_AES_256_CBC_SHA`

LogJam

- The Snowden leaks revealed that the NSA regularly MITMed TLS.
- How could they be doing this? Someone had missed something.
- In 2015, LogJam was discovered: https://weakdh.org/imperfect-forward-secrecy-ccs15.pdf
- A weak Diffie Hellman key is compatible with a strong Diffie Hellman key!

Diffie-Hellman

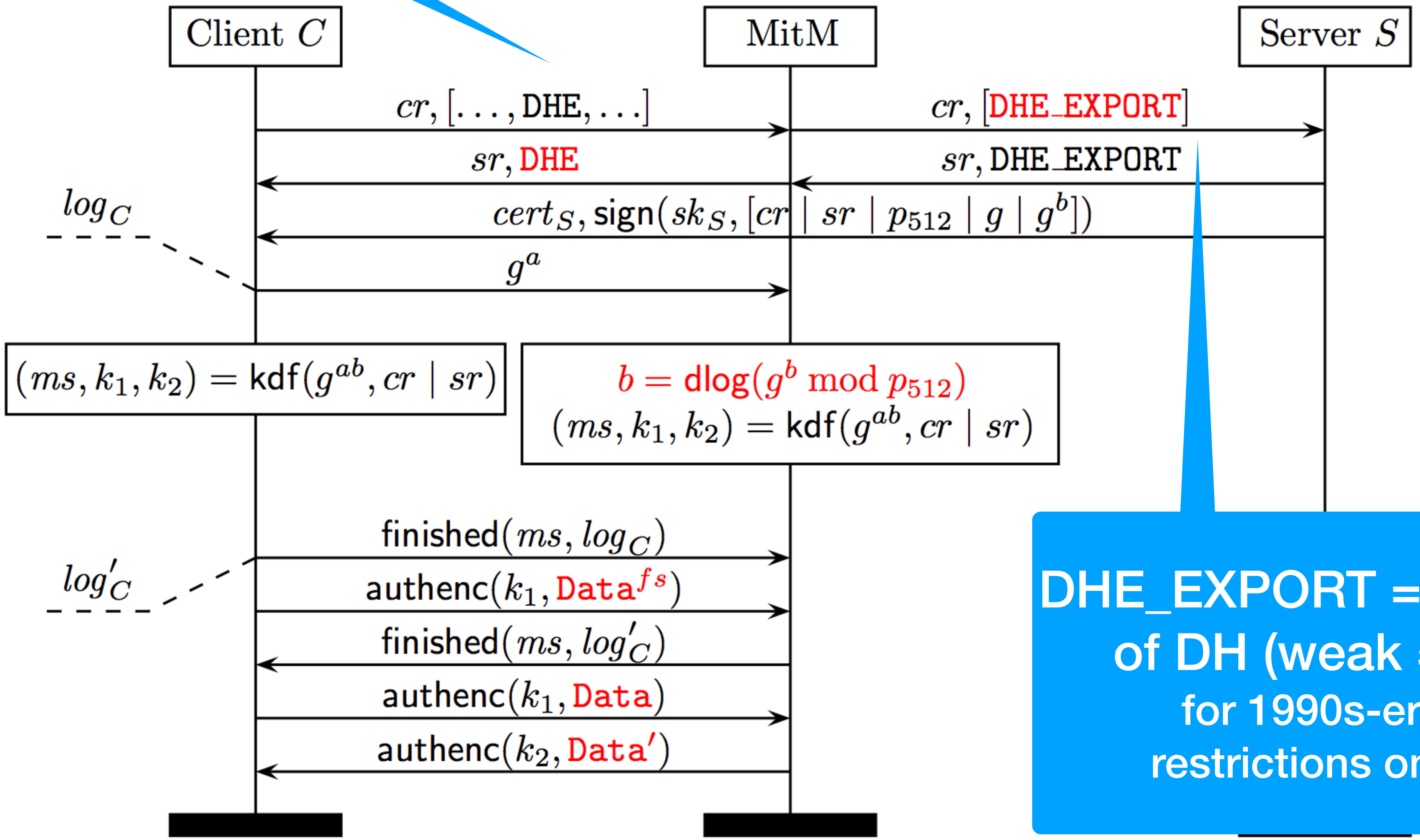
- Alice and Bob pick random numbers r_A and r_B and find "$t_A = g^{r_A} \mod p$" and "$t_B = g^{r_B} \mod p$"

- The protocol just exchanges these numbers:
    1. $A \rightarrow B : t_A$
    2. $B \rightarrow A : t_B$

- Alice calculates "${t_B}^{r_A} \mod p$" and Bob "${t_A}^{r_B} \mod p$", receiving the key: $K = g^{r_A r_B} \mod p$

Attack Diagram from paper

DHE = ephemeral DH (strong prime)



DHE_EXPORT = "export version" of DH (weak 512-bit prime)

for 1990s-era U.S. export restrictions on cryptography

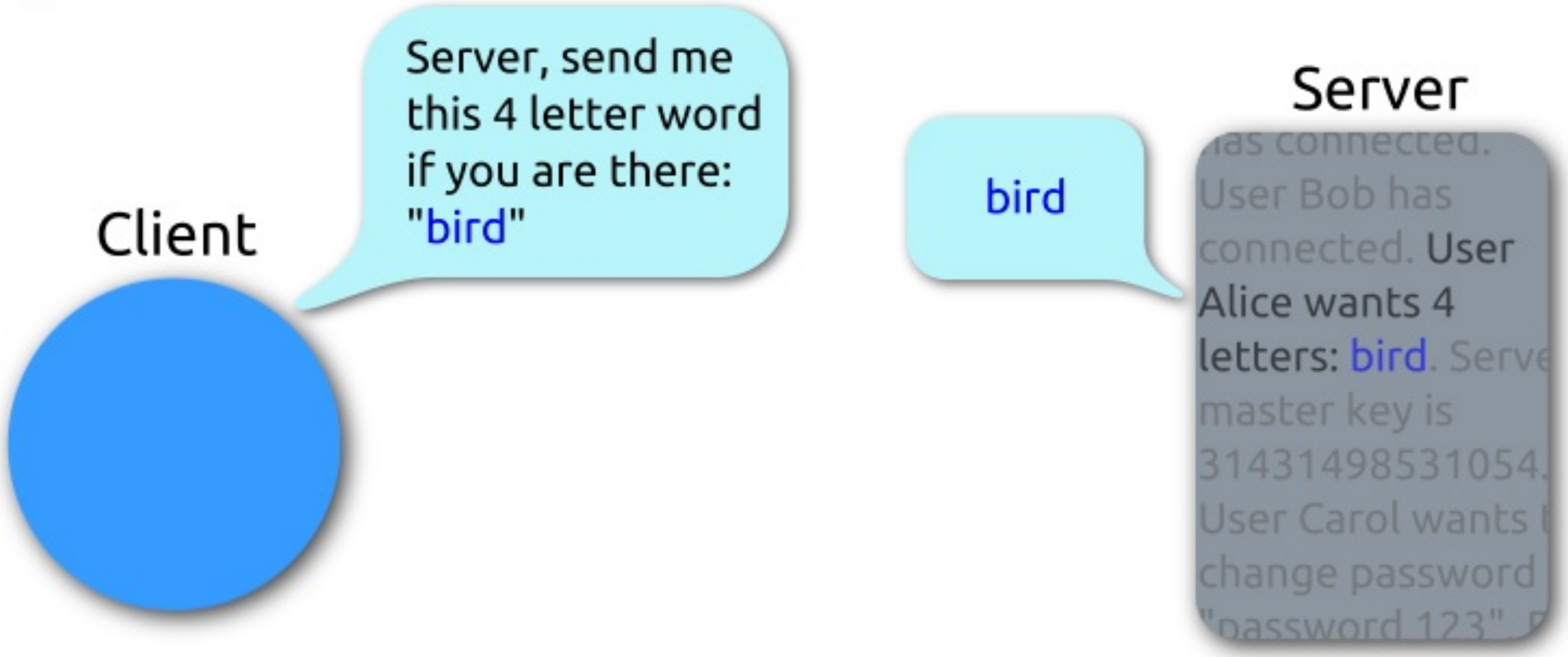
HeartBleed

- A programming error in OpenSSL

- Introduced in 2012, made public in 2014.
- Rumours it was being exploited.

- TLS client can request a “heart beat” from the server to make sure the connection is still open.
- This memory could contain the server’s key.

Heartbeat – Normal usage



From Wikipedia

BREACH

- The BREACH attack was discovered in 2013, one year after a similar attack named CRIME.
- BREACH (Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext) is not actually targeting TLS or any feature of TLS but **HTTP compression** when used with HTTPS.
- It requires
  - Confidential data to be included in the page
  - User data to be reflected in the page
  - Attacker, who can trigger user requests and observe encrypted traffic

BREACH

https://foobar.org/?input=canary

Input: canary

Secret: foobar123

- **Main insights:**
  - If the user input matches the secret, compression is most effective
  - And the size of the encrypted response leaks information about the compression

User input	Compressed response size
canary	100
foo	96
foobar123	92

Padding Oracles

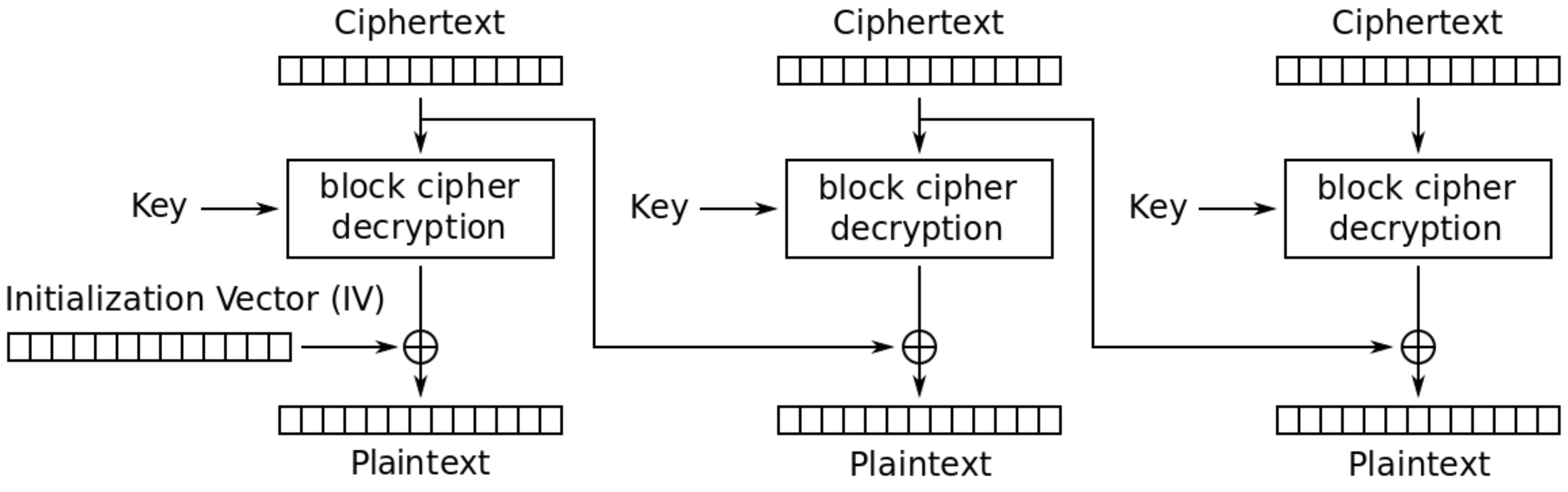
- Earlier TLS versions were vulnerable against Padding Oracle Attacks.
- A variation called *Lucky Thirteen* attack was published in 2013 (and AES-CBC is still vulnerable to this).

- PKCS#7 Padding:
  - 1 byte padding: append $0x01$
  - 2 bytes padding: append $0x0202$
  - 3 bytes padding: append $0x030303$
  - …

Padding Oracles

- How a padding oracle against CBC decryption works:

- CBC decryption: $C_0 = IV, P_i = D_K(C_i) \oplus C_{i-1}$
- Given C_1, C_2, change last byte of C_1 yielding C_1'
- Send C_1', C_2 to server and observe response
- Response is only whether padding was correct (P_2' ends with $0x01$)
- If padding is correct, the last byte of $D_K(C_2) \oplus C_1'$ is $0x01$ → last byte of $D_K(C_2)$ is $C_1' \oplus 0x01$; if not try out all 256 values
- Then continue with second-to-last byte and $0x02$ padding
  → for 128-bit blocks: max $16 \cdot 255 = 4080$ tries for the whole block



Cipher Block Chaining (CBC) mode decryption

TLS 1.3

- Newest standard, ratified August 2018
- Removes obsolete (cryptographic) protocols and features, among others:
    - MD5, RC4
    - Compression
    - Unauthenticated encryption
    - Static key exchanges (RSA and fixed DH;
      RSA because compromise of the private key would compromise all handshakes)
    - Renegotiation
- Simplified “1 round trip time” handshake → efficiency gain
- Forward secrecy mandatory
- Intercepting TLS connections now only possible as active attacker performing MITM attack

Cranor et al.’s Crying Wolf: “An Empirical Study of SSL Warning Effectiveness”

People that ignored warnings:

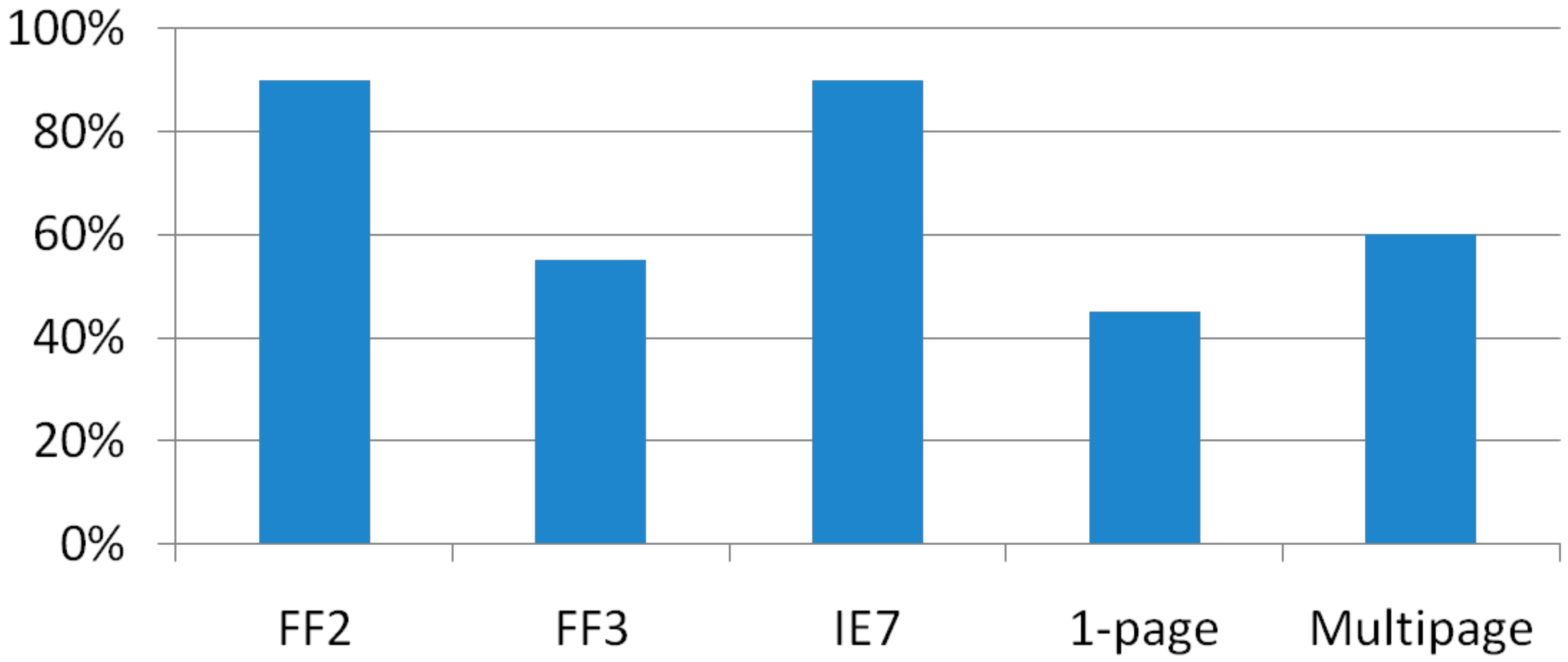


Image courtesy of Johnathan Nightingale

Image courtesy of Johnathan Nightingale

Checking servers

- There are many insecure TLS servers on the internet.
- The most common problems are support for weak ciphers and old unpatched code.
- SSL labs provide a useful testing tool:
  - https://www.ssllabs.com/ssltest/index.html

Summary

- What does TLS?
  - It provides **confidentiality**, **data integrity** and (partial) **authentication** on top of the transport layer
- How does TLS achieve this?
  - Encryption (several ciphers possible)
  - Message Authentication Codes (HMACs in particular)
  - Or Authenticated Encryption (integrating the authentication into the encryption)
  - Certificates
- Moreover, we have seen several attacks on TLS or associated applications!