The Transport Layer Security (TLS) Protocol

Lectures Notes for the “Computer Security” course

Winter 11/12

Pedro Félix (pedrofelix *at* cc.isel.ipl.pt)

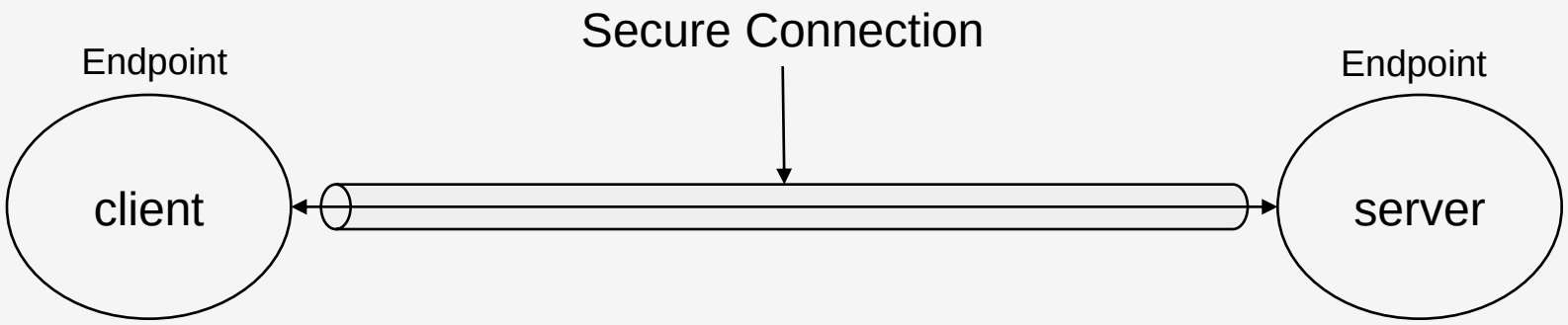
Instituto Superior de Engenharia de Lisboa

Some history

- SSL - Secure Sockets Layer
  - Proprietary specification done by Netscape
  - 1994, v1.0 – not released
  - 1994, v2.0 – critical weaknesses
  - 1995, v3.0 – widely used, IETF draft
- TLS – Transport Layer Security
  - 1999, IETF RFC 2246
  - Very similar but incompatible with SSL v3.0
    - Version value is 3.1
- This presentation
  - General concepts: valid on both SSL v3.0 and TLS
  - Details specific to TLS

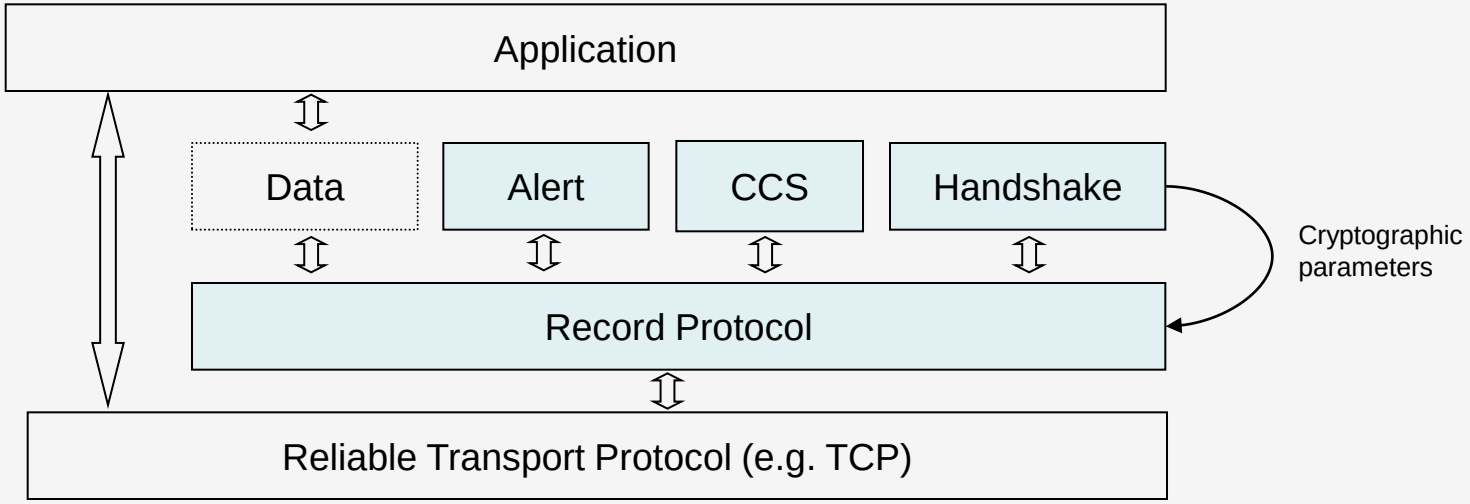
SSL/TLS Goals

- Creation, management and operation of a secure connection
- Creation and management
  - Endpoint authentication
  - Key and parameters establishment
  - Parameter reuse
- Operation
  - Message confidentiality
  - Message authentication



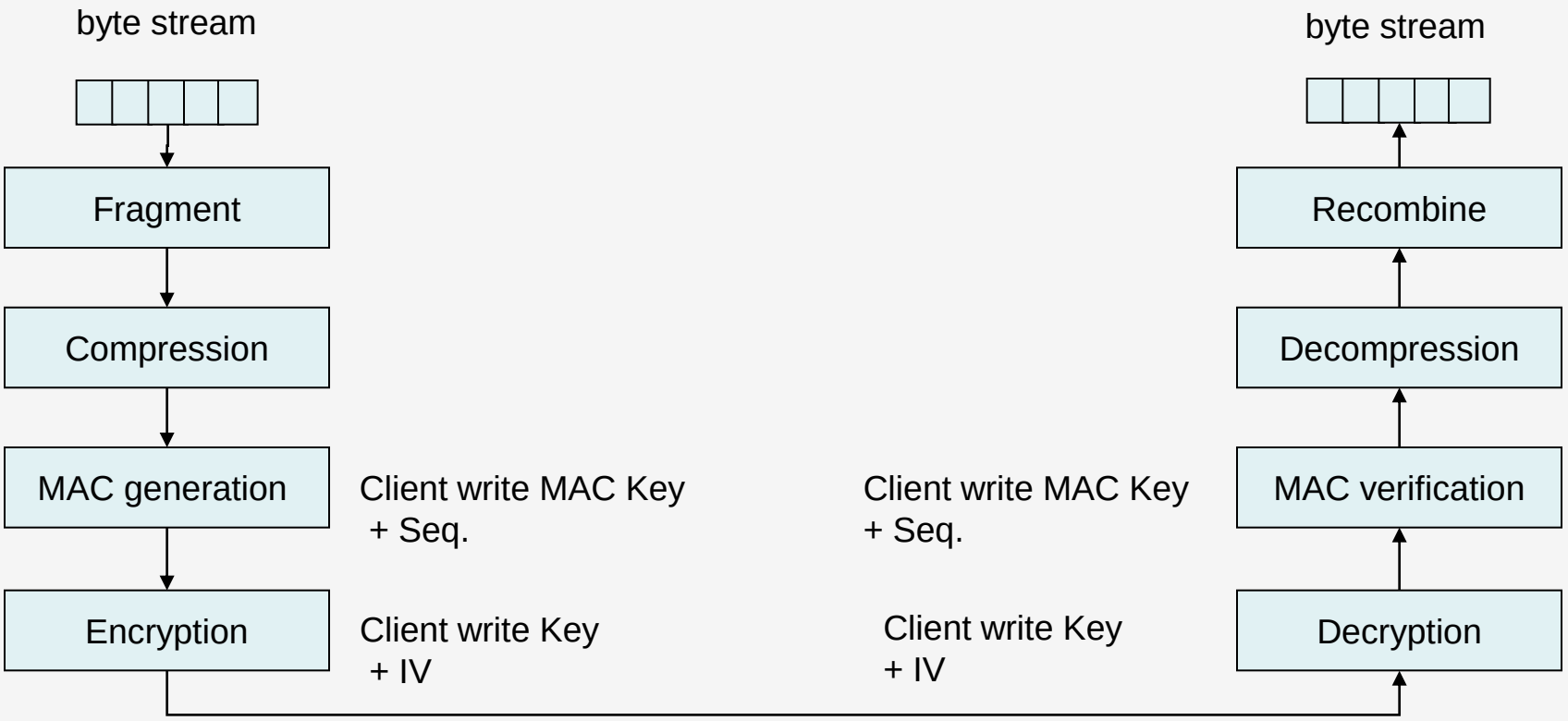
Sub-protocols

- Divided in two major sub-protocols
- Record Protocol
  - Handles data fragmentation, compression, confidentiality and message authentication
  - Requires a reliable transport protocol
- Handshake Protocol
  - Handles the secure connection creation and management, namely the secure establishment of the record protocol cryptographic parameters



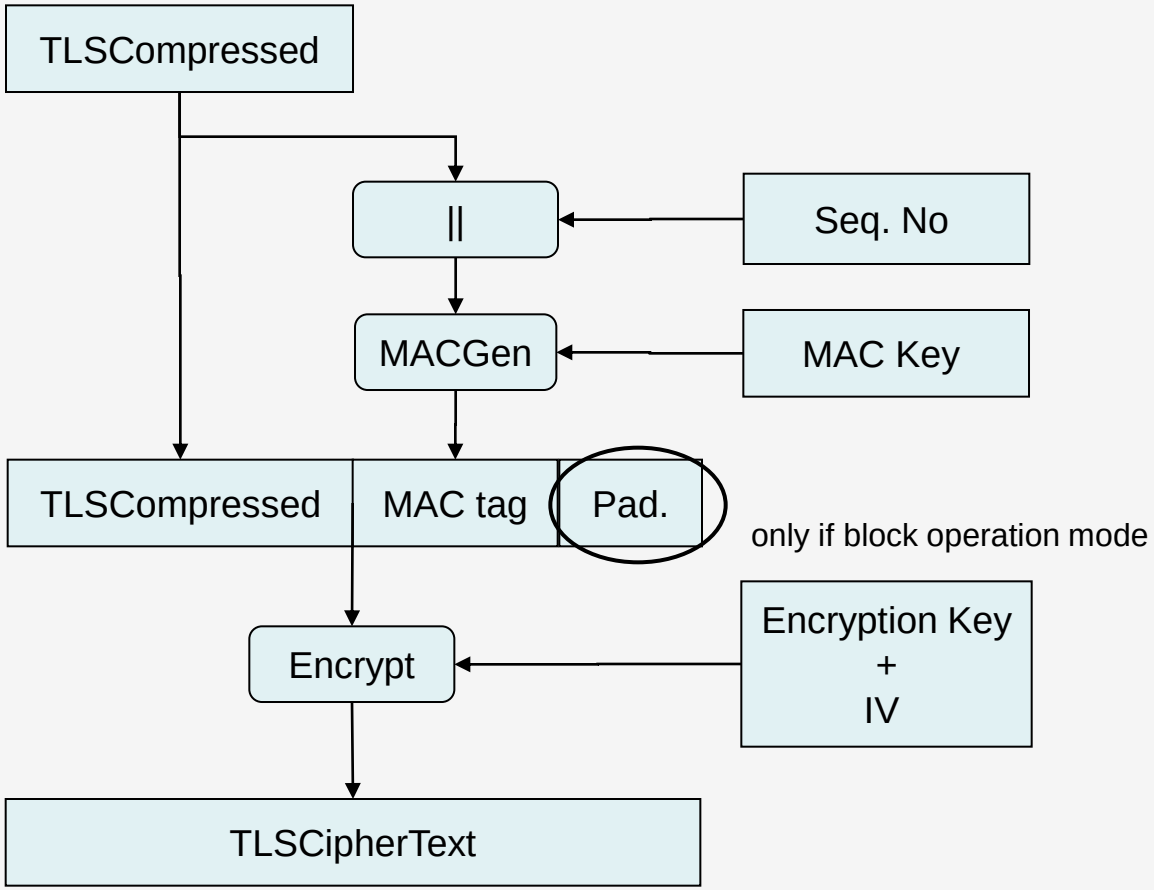
Record Protocol

- Fragment, Compress, Authenticate (MAC) then Encrypt
- Two independent connection directions
  - Separate keys, IVs and sequence number (client write and server write)



Record Protocol: authentication and encryption

- The MAC protects the: seq. no. + packet type + version + payload



Remarks

- Message replays
  - Detected by the sequence number
- Message reflection
  - Separate MAC keys for each direction
- Keystream reuse (stream based symmetric encryption)
  - Separate encryption keys and IVs for each direction
- Traffic analysis
  - Separate encryption keys
  - Variable padding length

Cryptographic schemes

- The cryptographic schemes used depend on the agreed *cipher suite*
- The cipher suite and compression algorithms are negotiated by the handshake protocol
- Examples
  - TLS_NULL_WITH_NULL_NULL
  - TLS_**RSA**_WITH_**3DES_EDE_CBC_SHA**
  - TLS_**RSA**_WITH_**RC4_128_SHA**

- A *cipher suite* defines
  - The hash function used by the HMAC (e.g. SHA)
  - The symmetric encryption scheme (e.g. 3DES_EDE_CBC or RC4_128)
    - Supports both block and stream schemes
  - Key establishment scheme (RSA or DH)

Handshake protocol

- Responsible for
  - Negotiation of the operation parameters
  - Endpoint authentication
  - Secure key establishment

- Endpoint authentication and key establishment
  - Authentication is optional on both ends
  - Supports several cryptographic techniques, namely:
    - Key transport (e.g. RSA)
    - Key agreement (e.g. DH)

  - Typical scenario on the internet (HTTPS)
    - RSA based key transport using X.509 certificates
    - Mandatory server authentication
    - Optional client authentication

Handshake Protocol: sketch

- If based on RSA key transport
  - C $\leftrightarrow$ S: negotiation of the algorithms to be used
  - C $\leftarrow$ S: server certificate
  - C $\rightarrow$ S: random secret encrypted with the server public key
  - C $\leftarrow$ S: proof of possession of the random secret

- If client authentication is required
  - C $\leftarrow$ S: Server requests the client certificate
  - C $\rightarrow$ S: client certificate
  - C $\rightarrow$ S: proof of possession of the private key, done by signing the handshake messages

Handshake Protocol (1): RSA based

ClientHello	C → S: client capabilities
ServerHello	C ← S: parameter definitions
Certificate	C ← S: server certificate (**KeS**)
CertificateRequest(*)	C ← S: Trusted CAs
ServerHelloDone	C ← S: synchronization
Certificate(*)	C → S: client certificate (**KvC**)
ClientKeyExchange	C → S: **Enc(KeS: pre_master_secret)**
CertificateVerify(*)	C → S: **Sign(KsC: handshake_messages)**
ChangeCipherSpec	C → S: record protocol parameters change
Finished	C → S: **{PRF(master_secret, handshake_messages)}**
ChangeCipherSpec	C ← S: record protocol parameters change
Finished	C ← S: **{PRF(master_secret, handshake_messages)}**

(*) optional

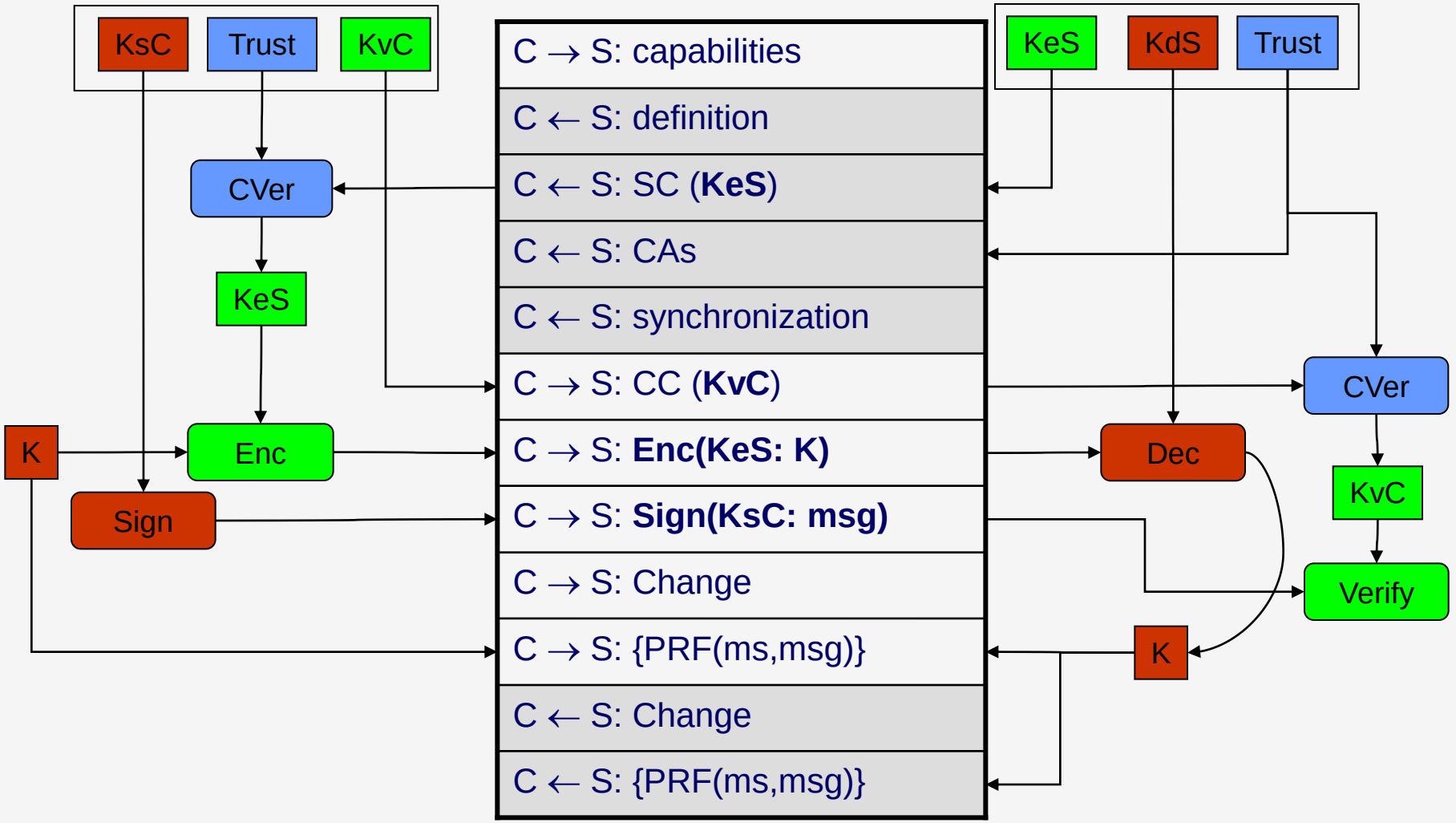
Handshake Protocol (2): RSA-based

- **ClientHello** – supported cipher suites and compression algorithms + proposed session identifier + client random value (!)
- **ServerHello** – Chosen cipher suite and compression algorithm + chosen session identifier + server random value (!)
- **Certificate** – certificate with the server public key (this key should support the agreed key establishment algorithm)
- **CertificateRequest** – client authentication request, containing:
  - List of the public key types supported by the server
  - List of the trust anchors trusted by the server
- **ServerHelloDone** – end of response

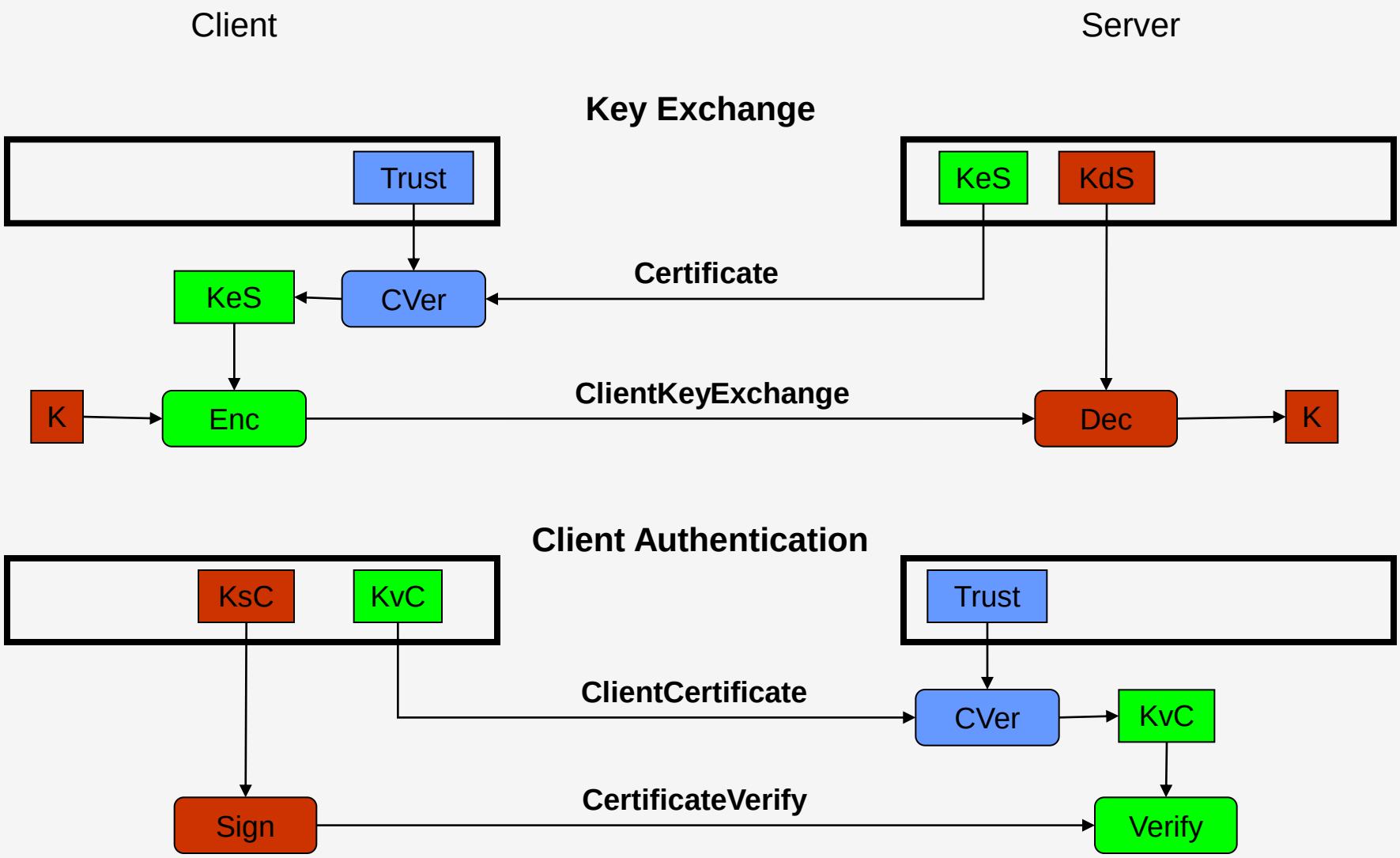
Handshake Protocol (3): RSA based

- **Certificate** – client certificate
- **ClientKeyExchange** – establishment of the premaster secret.
  - RSA - premaster secret is encrypted with the server public key
- **CertificateVerify** – Private key proof of possession
  - Done via the signature of all the previous protocol messages
- ***ChangeCipherSpec*** – Signalization that the next message is going to use the negotiated keys and settings
- **Finished** – Signalization of the end of the handshake.
  - Includes the PRF of all the messages, using the master secret
- All handshake messages (except **ClientKeyExchange**) are sent unprotected by the record protocol (null cipher suite)

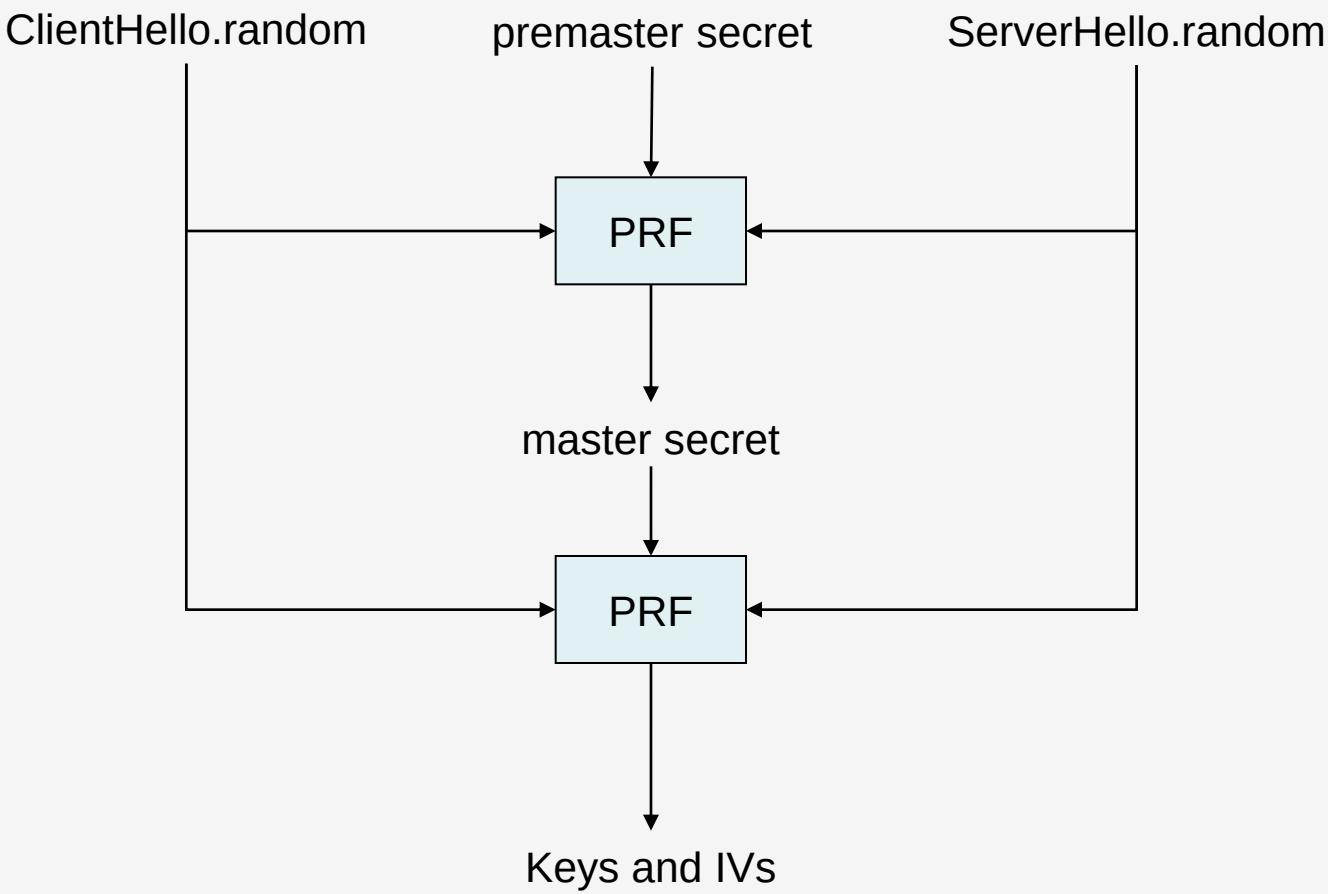
Yet another one



A schematic view



Key derivation



Handshake tampering and replay

- Handshake tampering is detected using the **Finished** message
  - The **Finished** message ensures both endpoints that the messages received are the same

- Handshake message replay
  - **ClientHello** and **ServerHello** contains random values (*nounces*), different for each handshake
  - Implies that the **Finished** message is different for each handshake

Endpoint authentication

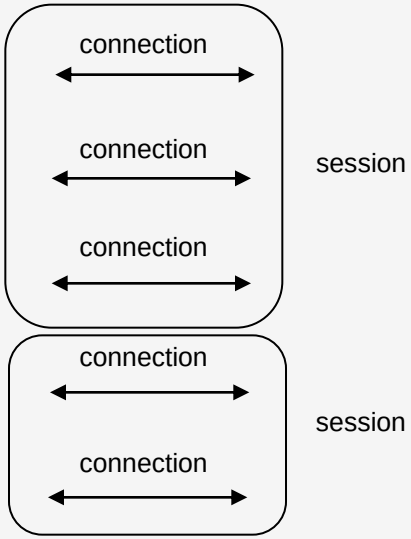
- The endpoint authentication uses different techniques for the client and the server

- Client authentication
  - Client certificate binds an identity (X500/DNS) to a public key
  - **CertificateVerify** proves the possession of the associated private key - signature of the handshake messages

- Server authentication
  - Server certificate binds an identity (X500/DNS) to a public key
  - **Finished** message proves that the server knows the master secret, implying that he was able to decrypt the premaster secret, implying that he has the associated private key

Connections and Sessions

- Connection
  - transport protocol connection
  - E.g. TCP connection
- Session
  - Association between the client and the server, containing the cryptographic parameters negotiated by the handshake protocol
- Sharing of a session between multiple connections
  - Due to the computational cost of the handshake protocol
  - E.g.: browser with multiple connections to the same server
- Different connections with the same session use different keys
  - Session defines a master secret
  - Each connection derives its key based on this master secret



Session and connection state

- Session
  - Session identifier
  - Peer certificates – identity of the endpoints
  - Compression method and cipher suite
  - Master secret – seed for the key derivation process
  - *Is resumable* – boolean indicating if the session can be used on several connections
- Connection state
  - Sequence numbers
  - MAC keys
  - Encryption Keys
  - Encryption Initialization Vectors (IV)
- The keys and IVs are created based on the master secret
- However, different keys are create for each connection, even if based in the same session

Session reuse

- Each handshake is associated with a session
- The **ClientHello** has ID
  - New one
  - Of another session
- The goal is to establish a new connection using the same session
  - Minimize the connection establishment overhead

ClientHello	C → S: Session ID of a previous session
ServerHello	C ← S: ID of the new session
ChangeCipherSpec	C ← S: Change Cipher Spec
Finished	C ← S: **{PRF(ms,handshake_messages)}**
ChangeCipherSpec	C → S: Change Cipher Spec
Finished	C → S: **{PRF(ms,handshake_messages)}**

Handshake request

- The **HelloRequest** message can be sent by the server at any time
- It's goal is to force a new handshake
- The client responds to this message with a **ClientHello** message

HelloRequest	C ← S: Request for a new *handshake*
ClientHello	C → S: ...
ServerHello	C ← S: ...
...	...

Attacks

- RSA timing attacks
  - D. Boneh, D. Brumley, *Remote timing attacks are practical*, 2th Usenix Security Symposium
- PKCS #1 v1.5 attacks
  - D. Bleichenbacher, *A chosen ciphertext attack against protocols based on the RSA encryption standard RSA PKCS #1*, Crypto'98
  - V. Klima, O. Pokorny and T. Rosa, *Attacking RSA-based Sessions in SSL/TLS*, CHES'03
- CBC/Authenticate-Then-Encrypt
  - S. Vaudenay, *Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS*, EuroCrypt'02

HTTPS protocol

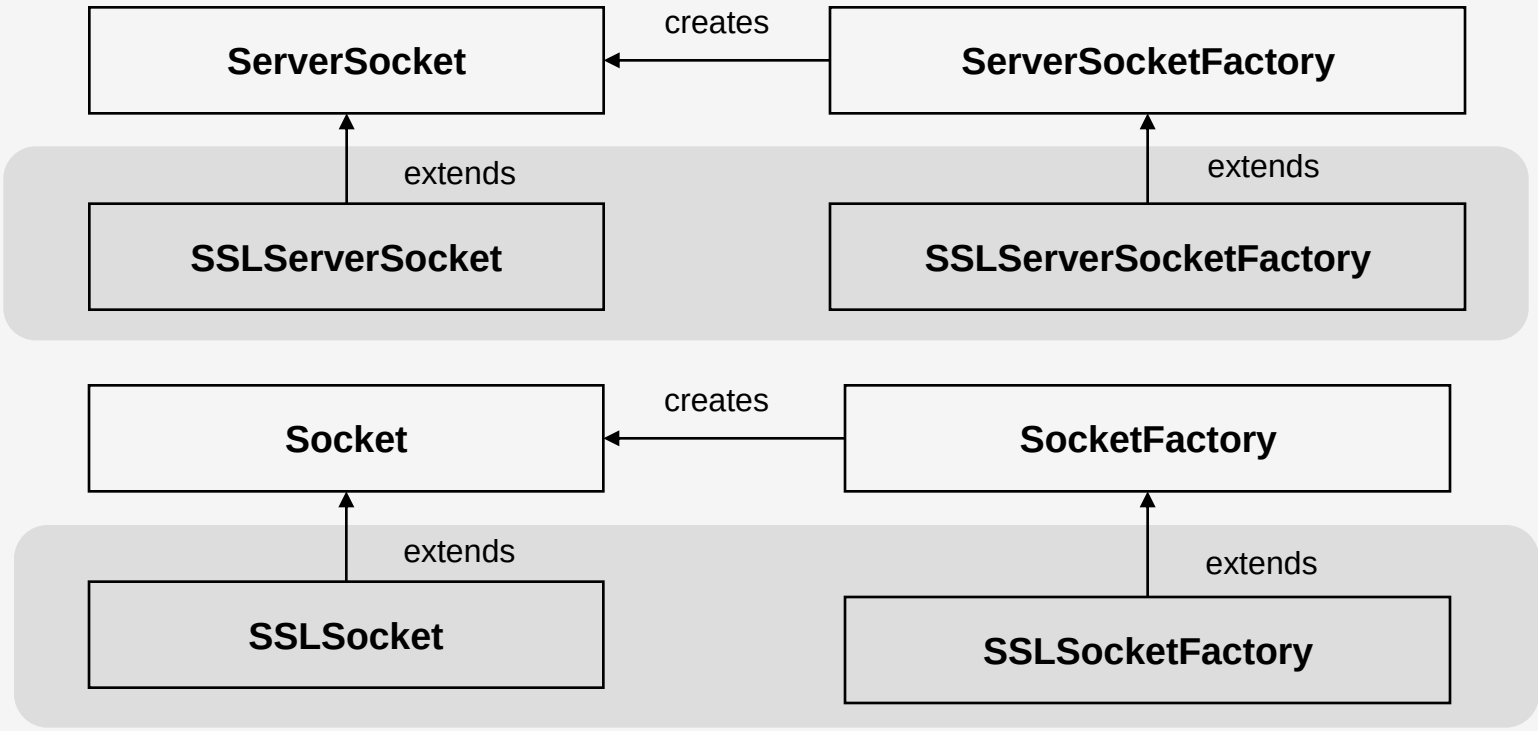
- HTTP over TLS
- Default port: 443
- Check between the URI and the certificate
  - **subjectAltName** extension of type **dNSName** (if present)
  - the (most specific) **Common Name** field in the **Subject** field
- Proxies
  - CONNECT method requests the proxy to create a TCP tunnel to the endpoint

Java Security Sockets Extension

- Java framework and implementation of the SSL and TLS protocols
- Provides SSL/TLS extensions of the regular (java.net) **Socket** and **ServerSocket** classes
- What can be done with it?
  - Secure socket creation and usage
  - Peer authentication
  - Custom certificate validation and trust anchor selection
  - Custom key selection
  - Session management
- Based on the same design criteria of the JCA (Java Cryptography Architecture)

Sockets and socket factories

- The regular sockets use factory-based instance creation
  - **ServerSocket** and **Socket** instances are created by **ServerSocketFactory** and **SocketFactory** instances
- The JSSE has specializations for the SSL/TLS protocols



Java Cryptography Architecture (JCA)

- Cryptographic framework for the Java platform
- Design principles
  - Algorithm independence and extensibility
  - Implementation independence and interoperability

- Architecture based on:
  - Cryptographic Service Providers (CSPs)
    - Packages implementing one or more cryptographic services
  - Engine Classes
    - Abstract definition (abstract class) of a cryptographic service
    - Instance creation via factory methods (static method **getInstance**)
  - Specification Classes
    - Normalized and transparent representations of cryptographic objects, such as keys and other parameters

Engine class examples

- **Signature** class – digital signature
- **Cipher** class – symmetric and asymmetric encryption
- **Mac** class – message authentication codes

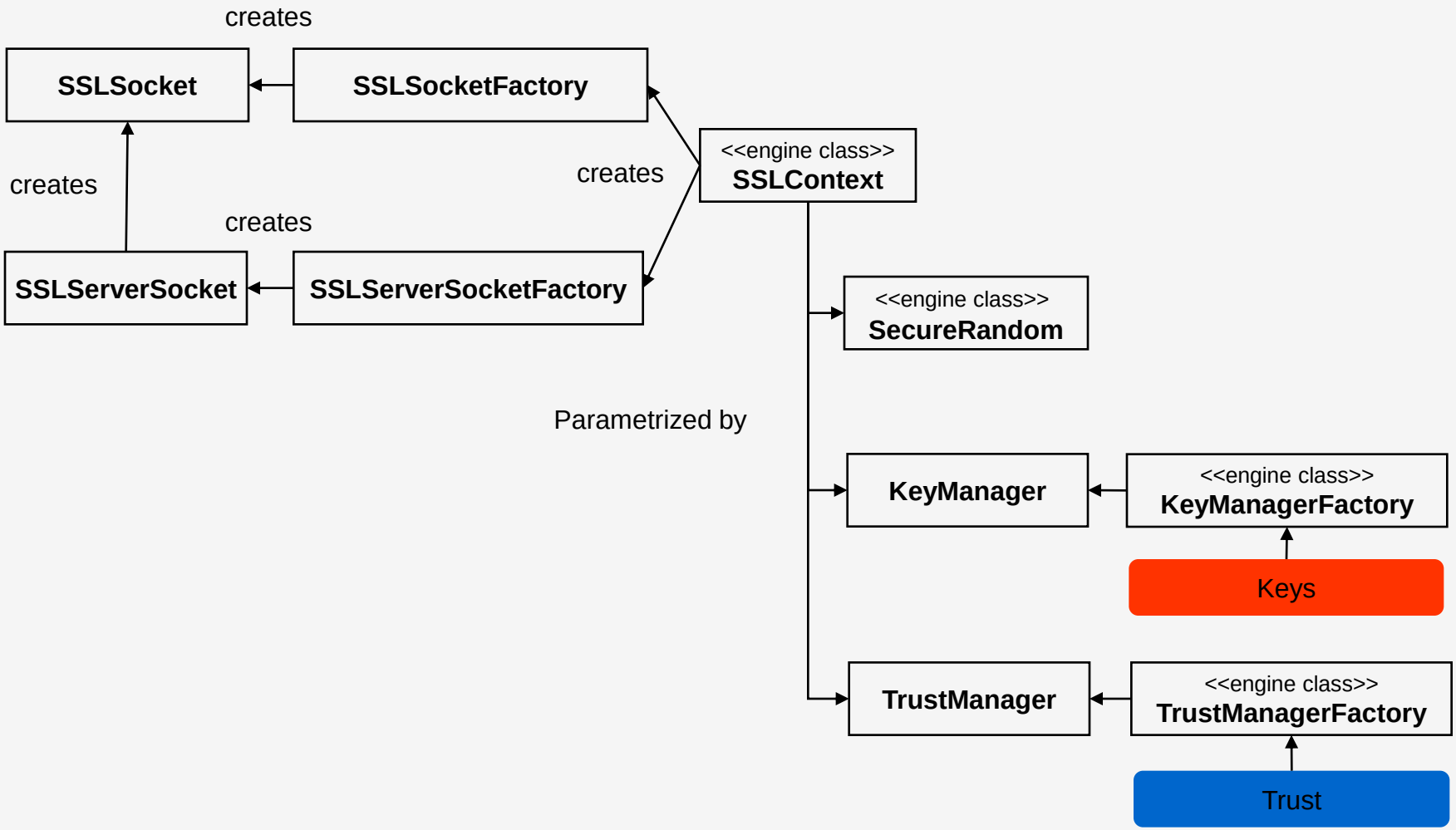
- **KeyStore** class - storage for cryptographic keys and certificates
  - Entries stored: private keys, secret keys and trusted certificates
  - Each entry is identified by a alias
  - Abstract, various implementations: JKS (Sun), PKCS# 12

- **CertPathValidator** class – validation of certificate chains

Functionality

- **SSLSocketFactory** and **SSLServerSocketFactory**
  - Obtain the default and supported cipher suites
  - Create socket instances
- **SSLSocket** and **SSLServerSocket**:
  - Initialize the handshake and receive notifications of its completion
  - Define the enabled protocols (SSL v3.0, TLS v1.0) and enabled cipher suites
  - Accept/require client authentication
  - Obtain the negotiated session
- **SSLSession**
  - Obtain the negotiated cipher suite
  - Get the authenticated peer identity and certificate chain

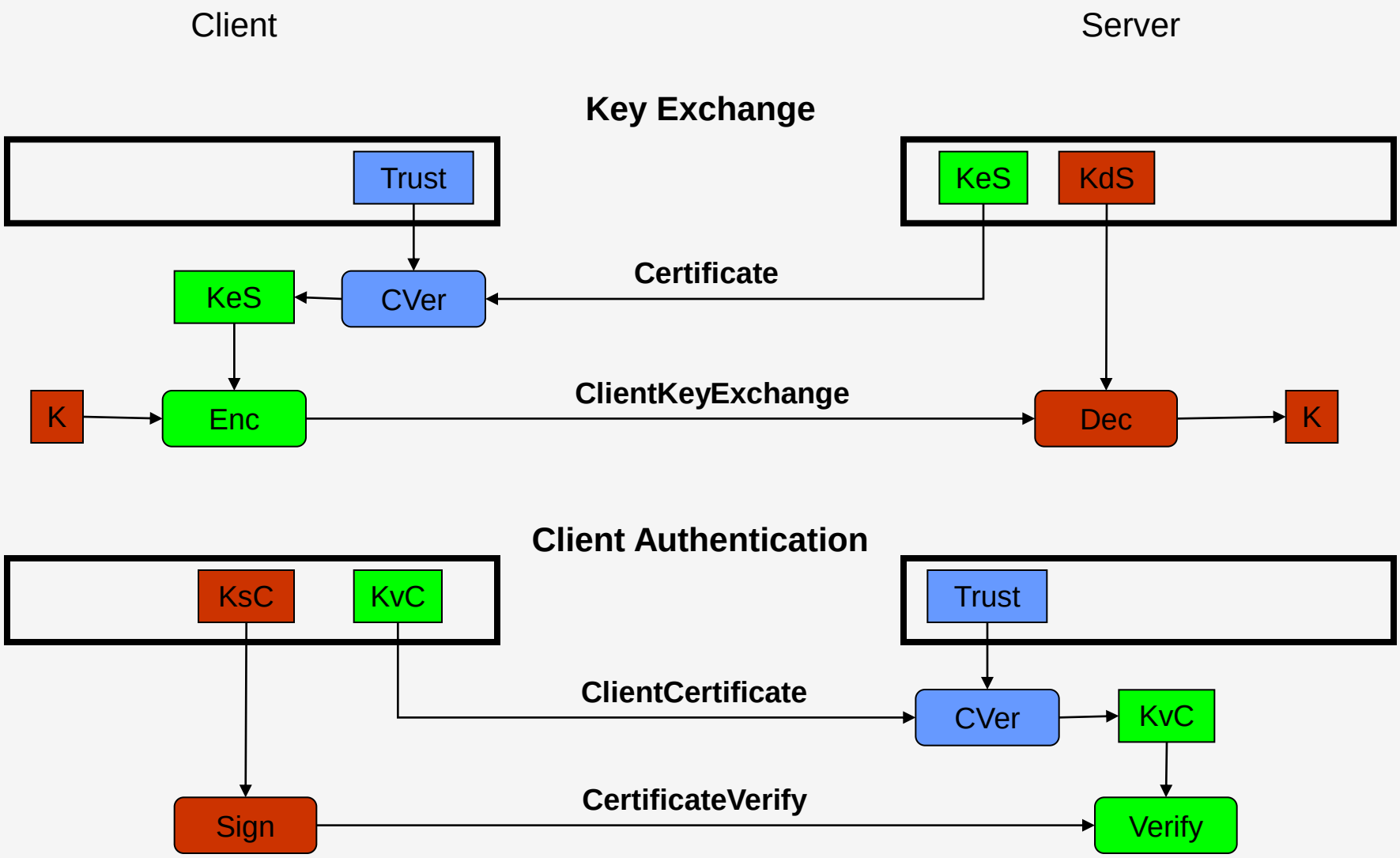
Factory based architecture



Socket Factory creation

- The creation of **SSLSocketFactory** and **SSLServerSocketFactory** is done by **SSLContext** instances
- Implicit used via the static methods **getDefault** from **SSLSocketFactory** and **SSLServerSocketFactory**
- **SSLContext** instance creation using the **getInstance** static method, initialized with
  - Randomness source – **SecureRandom** class
  - Key manager – **KeyManager** class
  - Trust manager – **TrustManager** class

Remember this?



Key and trust managers

- Trust Manager - determines whether the remote authentication credentials (and thus the connection) should be
  - Construction and verification of certificate chains
  - Determination of the trust anchors

- Key Manager - determines which authentication credentials to send to the remote host
  - Choose the identity to be used (alias string), given a list of accepted trust anchors
  - Get the private key associated with an alias
  - Get the certificate chain associated with an alias

X509TrustManager e X509KeyManager

- X509TrustManager
  - void checkClientTrusted(X509Certificate[] chain, String authType)
  - void checkServerTrusted(X509Certificate[] chain, String authType)
  - X509Certificate[] getAcceptedIssuers()

- X509KeyManager
  - String chooseClientAlias(String[] keyType, Principal[] issuers, Socket socket)
  - String chooseServerAlias(String keyType, Principal[] issuers, Socket socket)
  - X509Certificate[] getCertificateChain(String alias)
  - String[] getClientAliases(String keyType, Principal[] issuers)
  - PrivateKey getPrivateKey(String alias)
  - String[] getServerAliases(String keyType, Principal[] issuers)

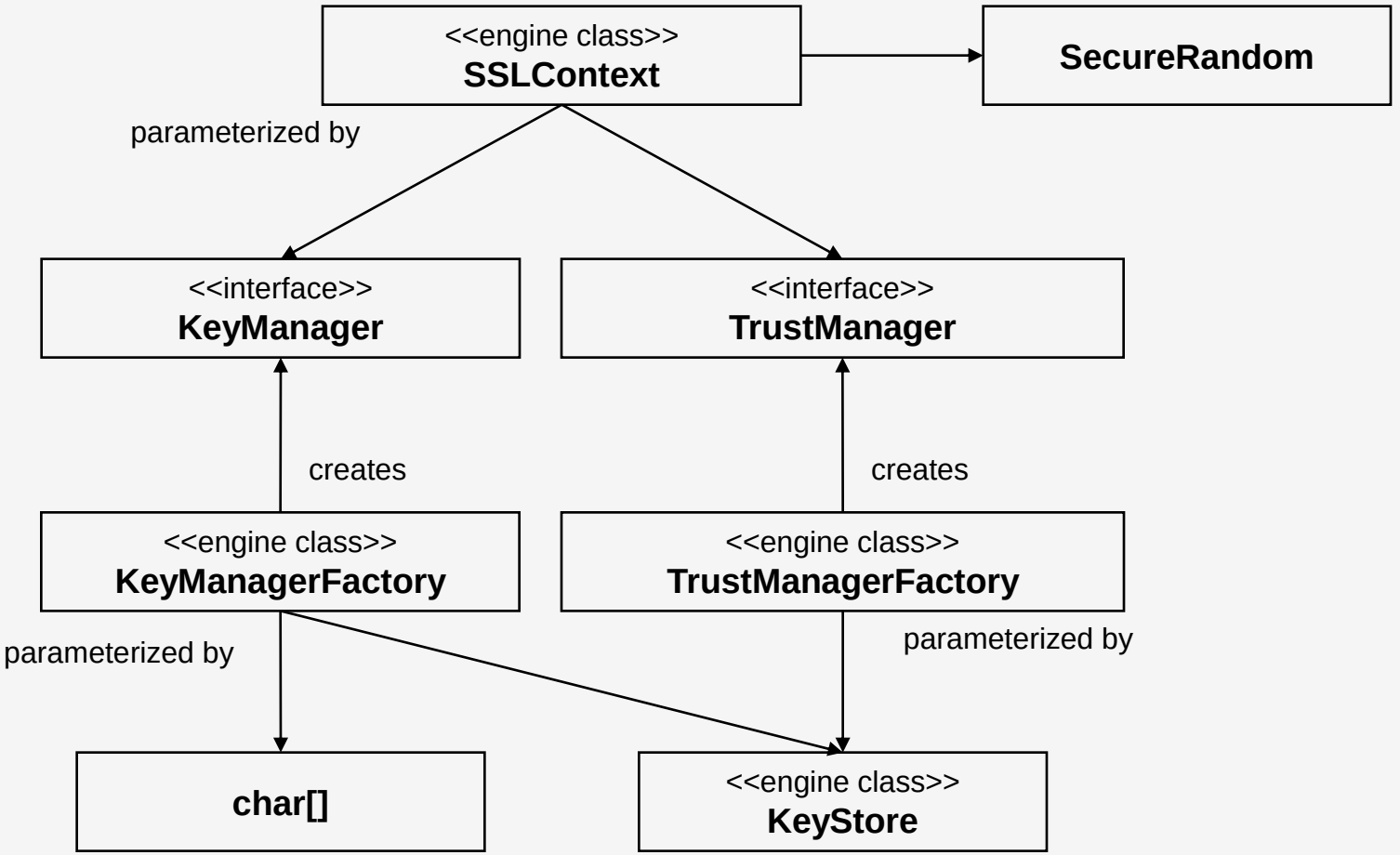
Manager Factories

- Creation of **KeyManager** and **TrustManager** instances using **KeyManagerFactory** e **TrustManagerFactory** (engine classes) instances

- KeyManagerFactory
  - static KeyManagerFactory getInstance(String algorithm)
  - void init(KeyStore ks, char[] password)
  - KeyManager[] getKeyManagers()

- TrustManager
  - static TrustManagerFactory getInstance(String algorithm)
  - void init(KeyStore ks)
  - TrustManager[] getTrustManagers()

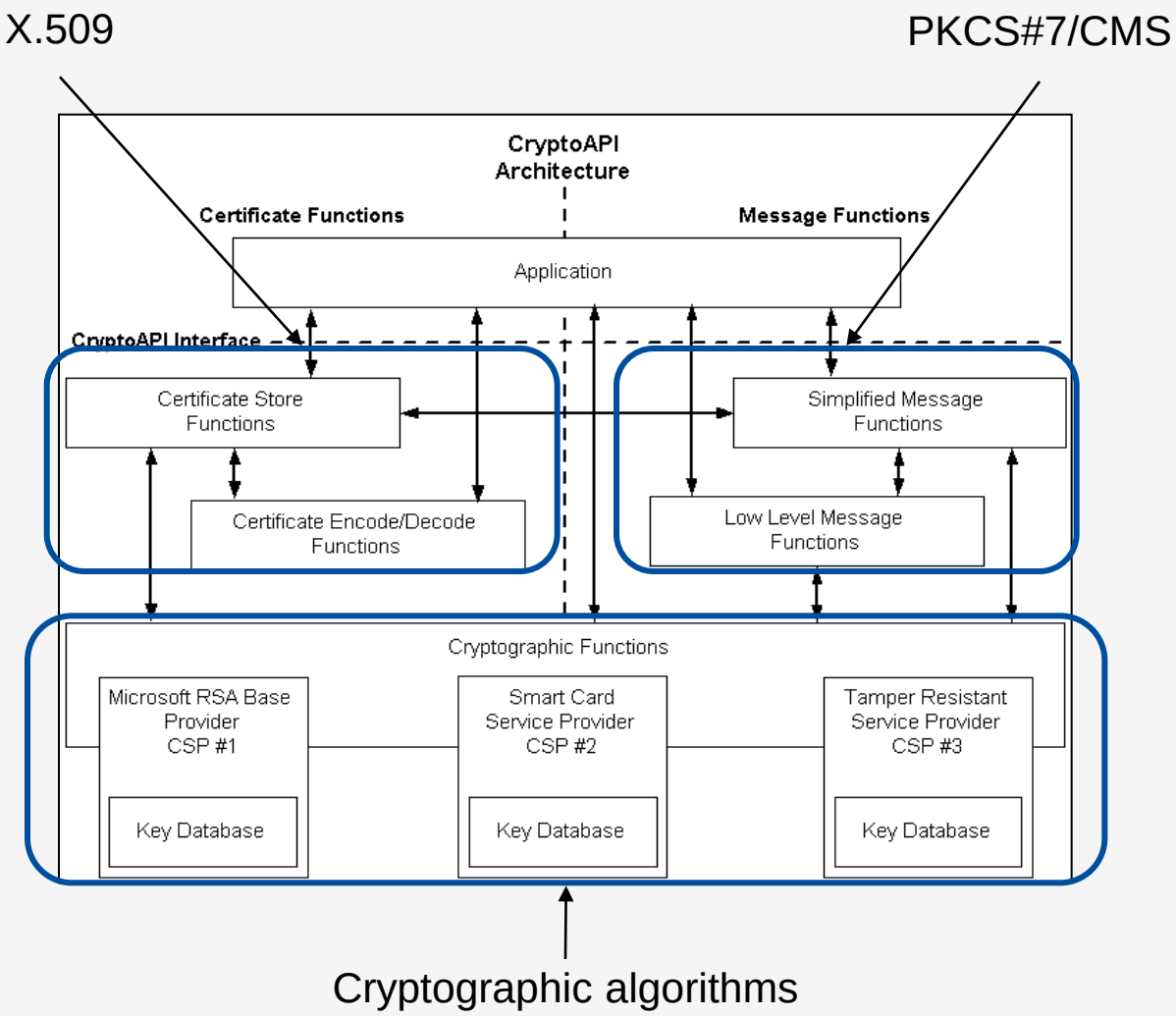
The big picture



Cryptography on Windows platforms

- 1996 – CryptoAPI (C language interface)
  - Symmetric encryption schemes, MAC schemes, hash functions, digital signature schemes, key establishment and encryption
  - X.509 certificates
  - PKCS 7/ CMS cryptographic messages
- ? - SSPI – Security Support Provider Interface (C language interface)
  - Authentication and key establishment protocols (ex. *Kerberos*, SSL)
- 2001 – CAPICOM (COM)
  - Subset of CryptoAPI exported as COM components
- 2002 – System.Security.Cryptography (.NET Framework Class Library)
- 2005 – Addition of functionality to System.Security
  - X509 certificates
  - PKCS#7/CMS messages
  - XML signature and encryption

CryptoAPI: architecture



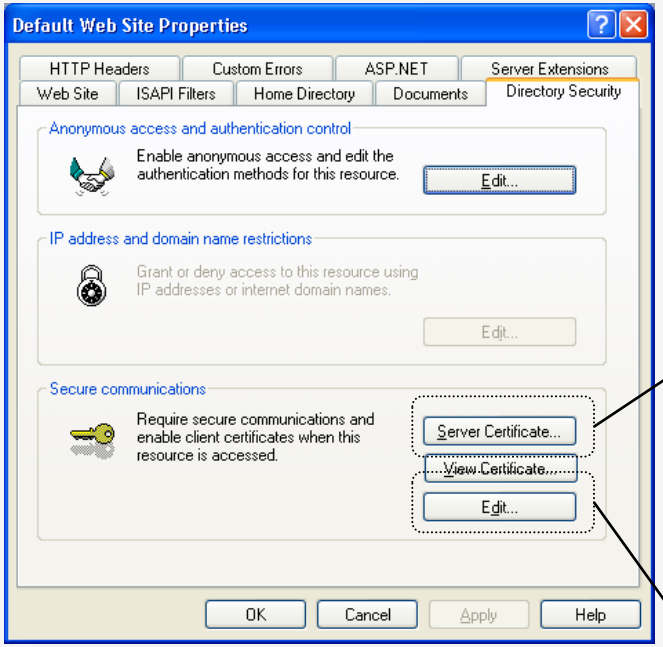
Certificate stores

- Physical stores: files, registry,…
- Logical stores – collection of physical stores
- System stores – logical stores associated with
  - Current user
  - Local computer
  - A service
- All the stores are identified by a string: “MY”, “CA”, “ROOT”
- CryptoAPI has tools for the creation and management of certificate stores
- There is also a Microsoft Management Console snap-in for this purpose
- Contents of the certificate stores
  - Trusted root certificates (trust anchors)
  - Personal certificates (associated private key is store on the CryptoAPI)
  - Other certificates (e.g. intermediate certificates)

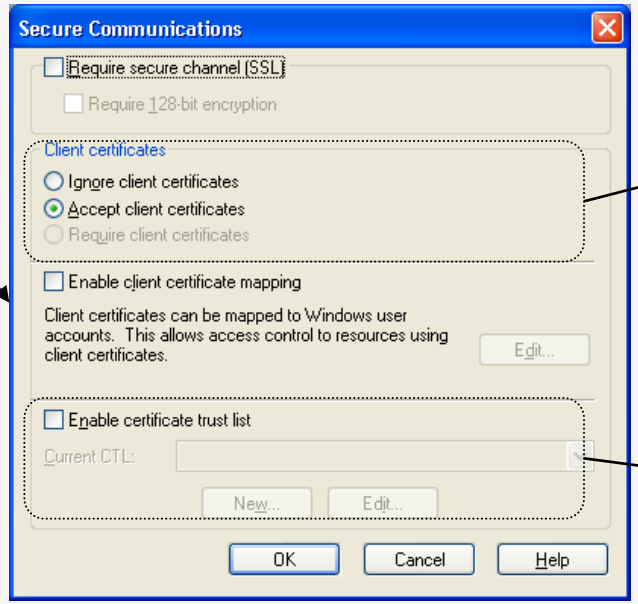
SSL/TLS on Windows: examples

- Configuring HTTPS
  - Server – IIS
  - Client – IE, Firefox
- Programming HTTPS clients and servers
  - ASP.NET server page
  - Client with and without authentication
  - Certificate policies
- Web services using HTTPS transport
  - Client
  - Service

Configuring IIS (server)



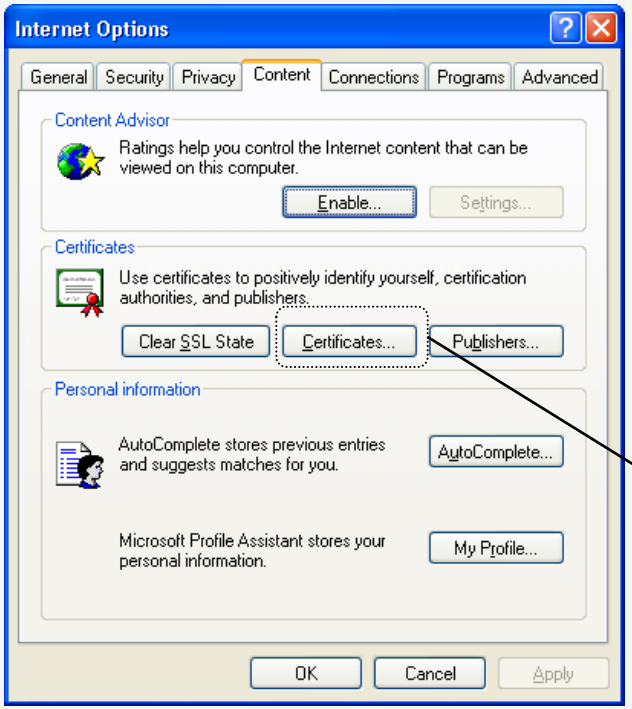
Certificate (public key) + private key



Client Authentication

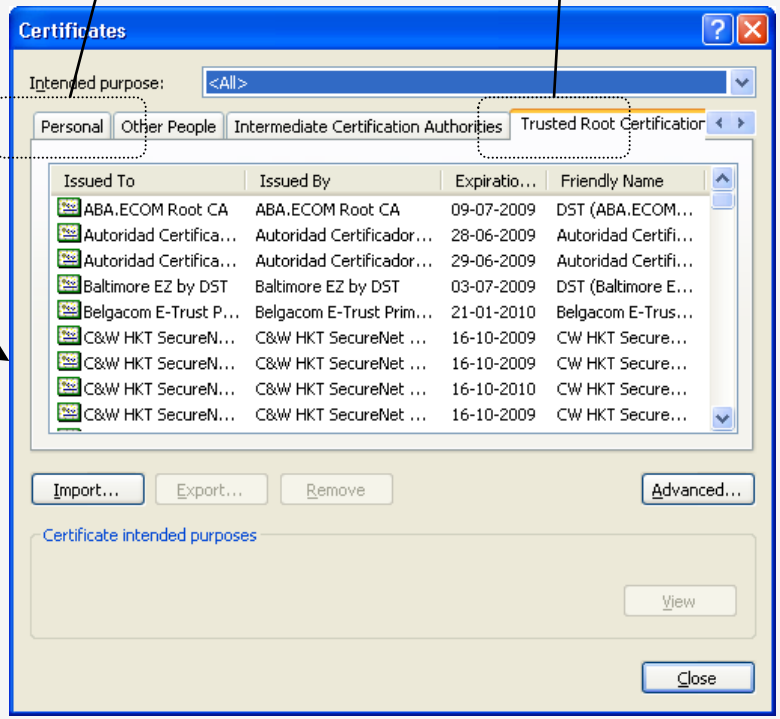
Trust anchors

Configuring IE (client)

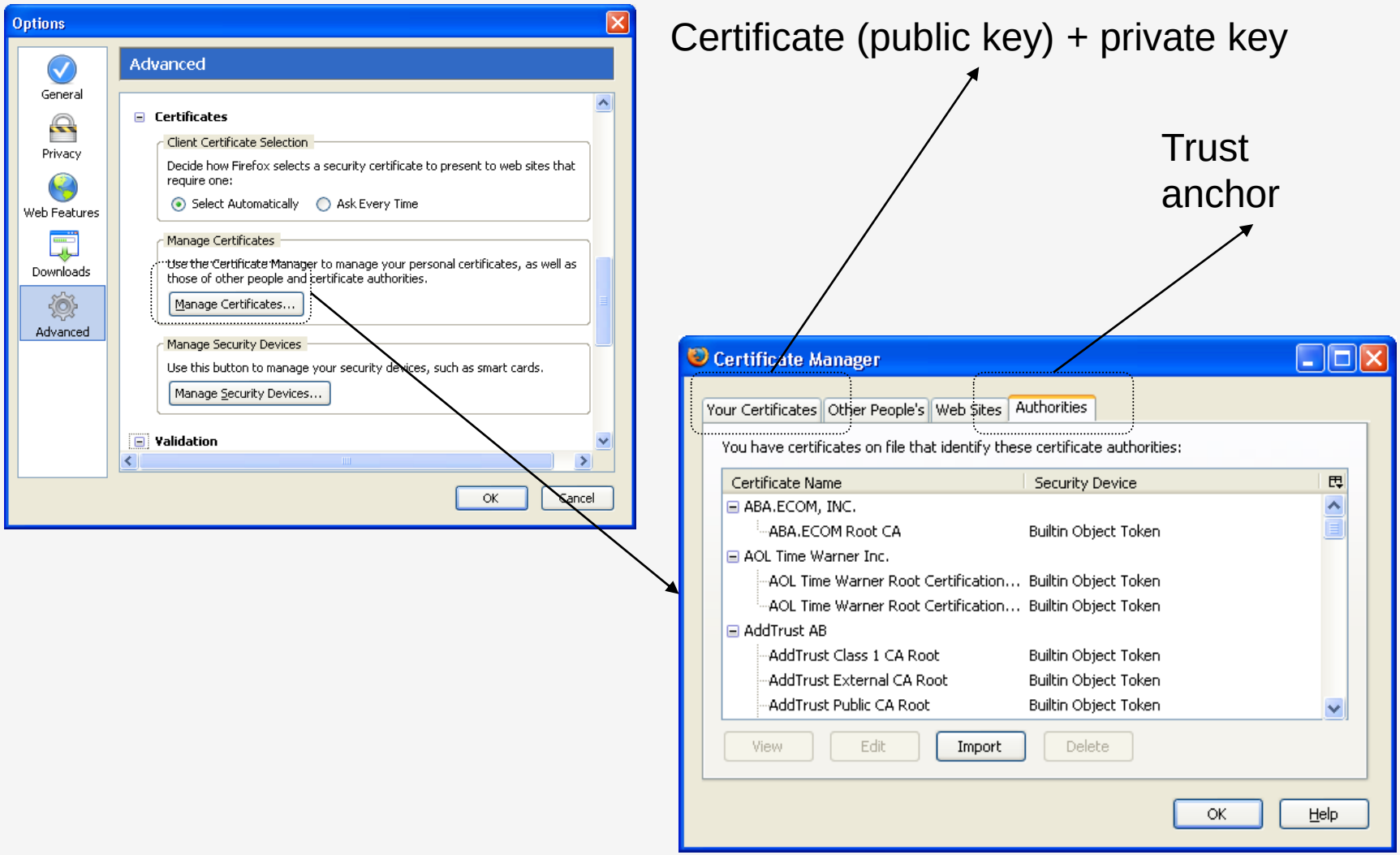


Certificate (public key) + private key

Trust anchors



Configuring Firefox (client)



Server: ASP.NET page using HTTPS

```
public class CertEcho : System.Web.UI.Page{
    protected TextBox message;
    protected override void OnLoad(EventArgs ea){
        // Get the client certificate
        HttpClientCertificate clientCert = Context.Request.ClientCertificate;

        // check if the certificate exists
        if(clientCert.IsPresent == false)                    {
            message.Text = "No client authentication";
        }else{
            // show the certificate information
            message.Text = "Client authentication using certificate:" + "\n";
            message.Text += "Issuer : " + clientCert.Issuer + "\n";
            message.Text += "Subject : " + clientCert.Subject + "\n";
        }
    }
}
```

Client: HTTP request

- HTTP request

```
static void GetHttp(string url)
{
    HttpWebRequest req = WebRequest.Create(url) as HttpWebRequest;
    HttpWebResponse res = req.GetResponse() as HttpWebResponse;
    ShowResponseContent(res);
}
```

Client: HTTPS request

- HTTPS request
- Additions
  - Define the *trust* policy

```
static void GetHttpSecure(string url)
{
    ServicePointManager.CertificatePolicy = new MyCertificatePolicy1();
    ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls;
    HttpWebRequest req = WebRequest.Create(url) as HttpWebRequest;
    HttpWebResponse res = req.GetResponse() as HttpWebResponse;
    ServicePoint sp = req.ServicePoint;
    Console.WriteLine("Certificate : "+sp.Certificate.GetName());
    ShowResponseContent(res);
}
```

ICertificatePolicy interface

- The **ICertificatePolicy** interface defines the certificate verification policy

```
public class MyCertificatePolicy1 : ICertificatePolicy
{
    public bool CheckValidationResult(ServicePoint sp,
        X509Certificate cert, WebRequest request, int problem)
    {
        ...
        return ValidationResult;
    }
}
```

Client : HTTPS request with client authentication

- HTTPS request with client authentication
- Additions
  - Define the client certificate **and associated private key**

```
static void GetHttpSecureWithClientAuth(string url){
    ServicePointManager.CertificatePolicy = new MyCertificatePolicy1();
    HttpWebRequest req = WebRequest.Create(url) as HttpWebRequest;
    req.ClientCertificates.Add(
            X509Certificate.CreateFromCertFile("c:/.../pedrofelix.cer")
    );
    HttpWebResponse res = req.GetResponse() as HttpWebResponse;
    ServicePoint sp = req.ServicePoint;
    Console.WriteLine("Certificate : "+sp.Certificate.GetName());
    Console.WriteLine("Client Certificate : "+ sp.ClientCertificate.GetName() );
    ShowResponseContent(res);
}
```

ClientCertificates

- The ClientCertificates property returns the certificate collection that can be used to authenticate the client
  - The client must have a private key associated with the certificate
  - The chosen certificate has a root trusted by the CA

- The **ServicePoint.ClientCertificate** returns the used certificate

Usage on Web Services (ASP.NET)

- **Service**
  - The client certificate is available on the request context

```
public class certecho : System.Web.Services.WebService{
    [WebMethod]
    public string CertEcho()    {
        // Obter certificado de cliente
        HttpClientCertificate clientCert = Context.Request.ClientCertificate;
        ...
```

- **Client**
  - Add the client certificate to the proxy certificate collection

```
ServicePointManager.CertificatePolicy = new TLSClient.MyCertificatePolicy1();
proxy.ClientCertificates.Add(
    X509Certificate.CreateFromCertFile("c:/.../pedrofelix.cer")
);
```