The Transport Layer Security (TLS) Protocol

Lectures Notes for the “Computer Security” course

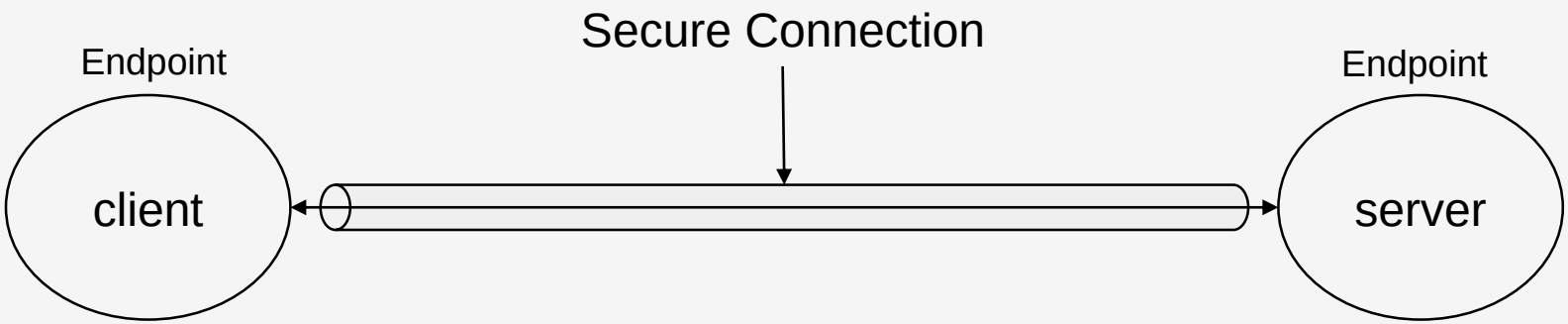
Pedro Félix (pedrofelix *at* cc.isel.ipl.pt)
José Simão (jsimao *at* cc.isel.ipl.pt)
Instituto Superior de Engenharia de Lisboa

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

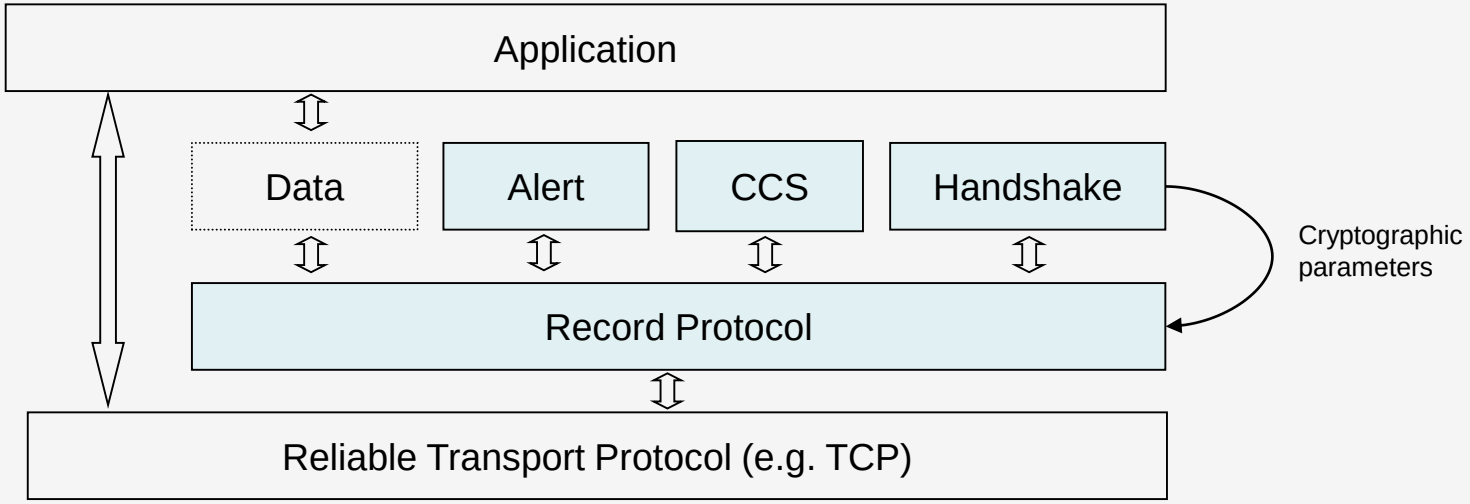
SSL/TLS Goals

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



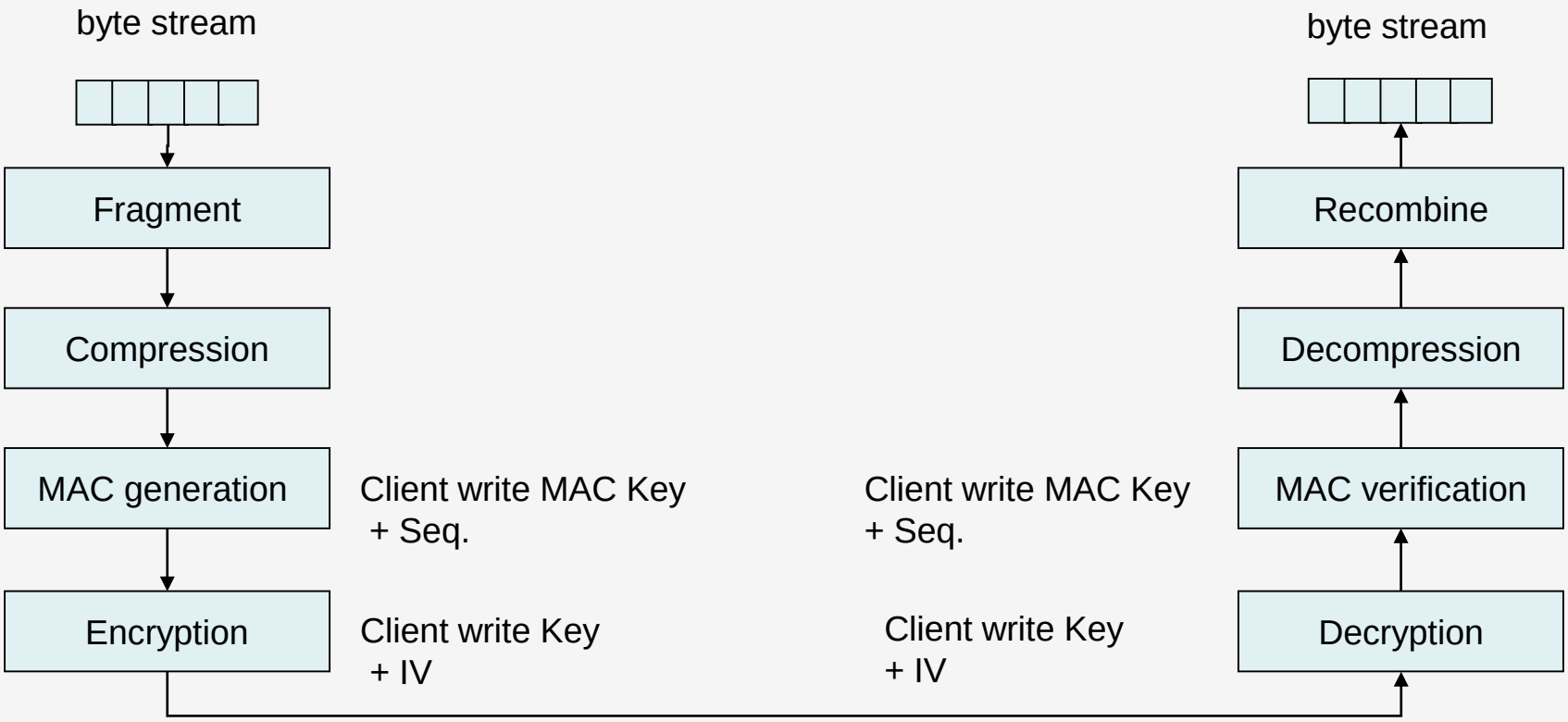
Sub-protocols

- 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



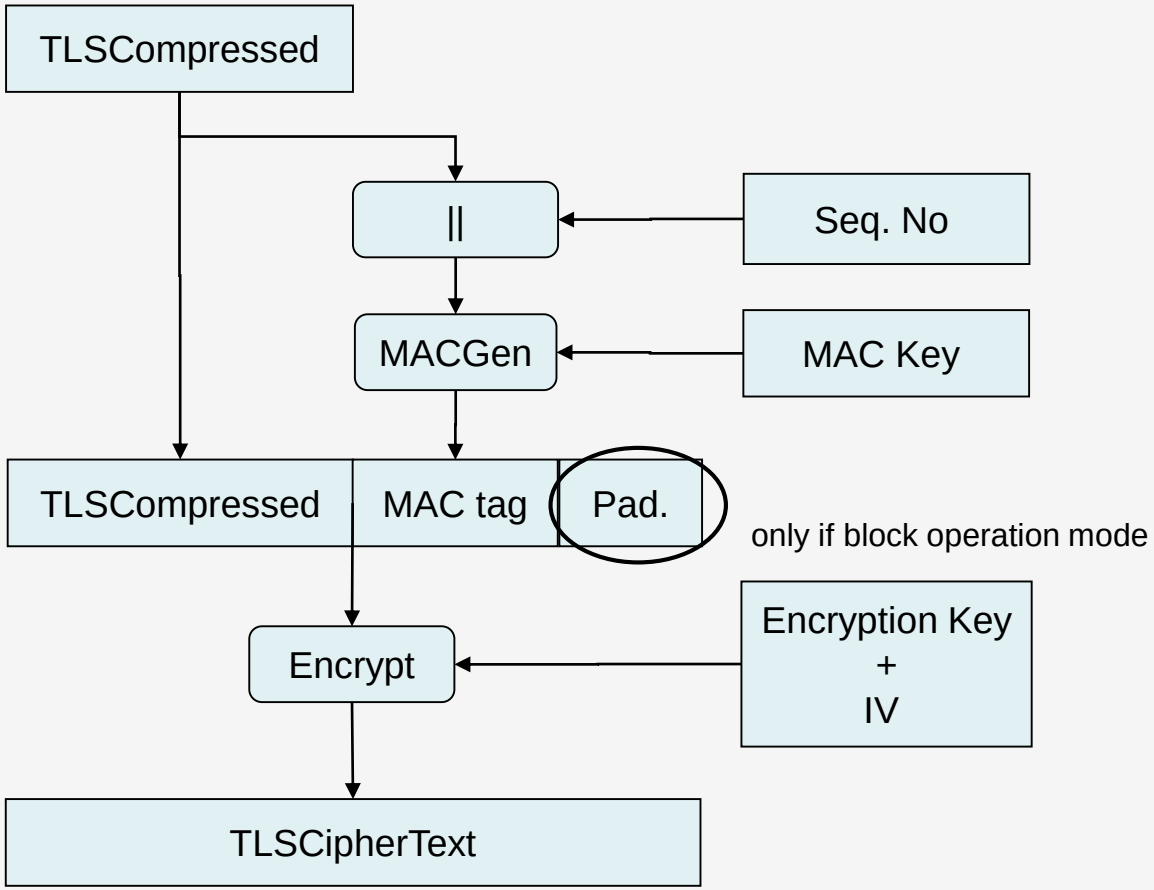
Record Protocol

- 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

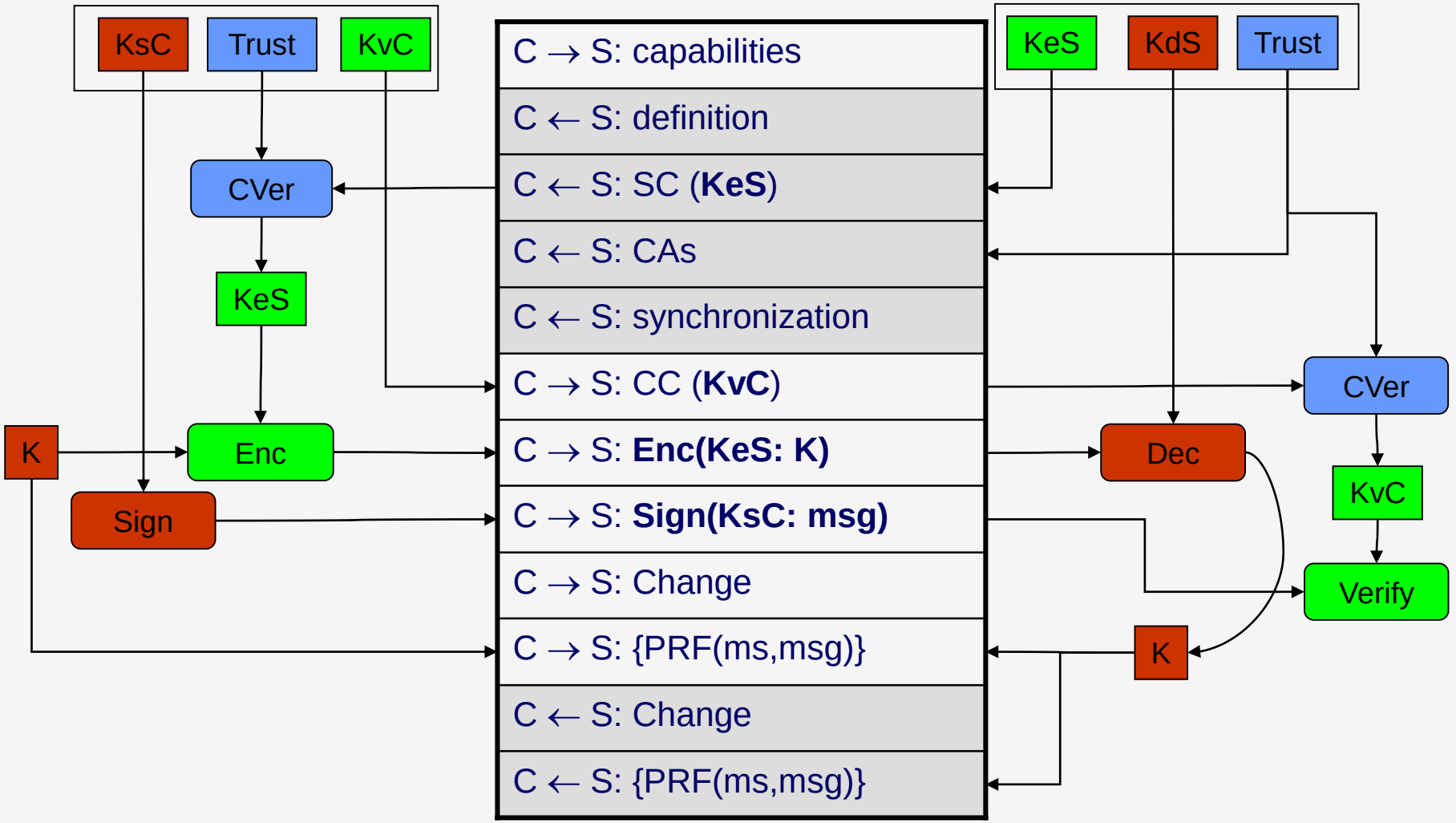
Handshake Protocol (2): RSA-based

- **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

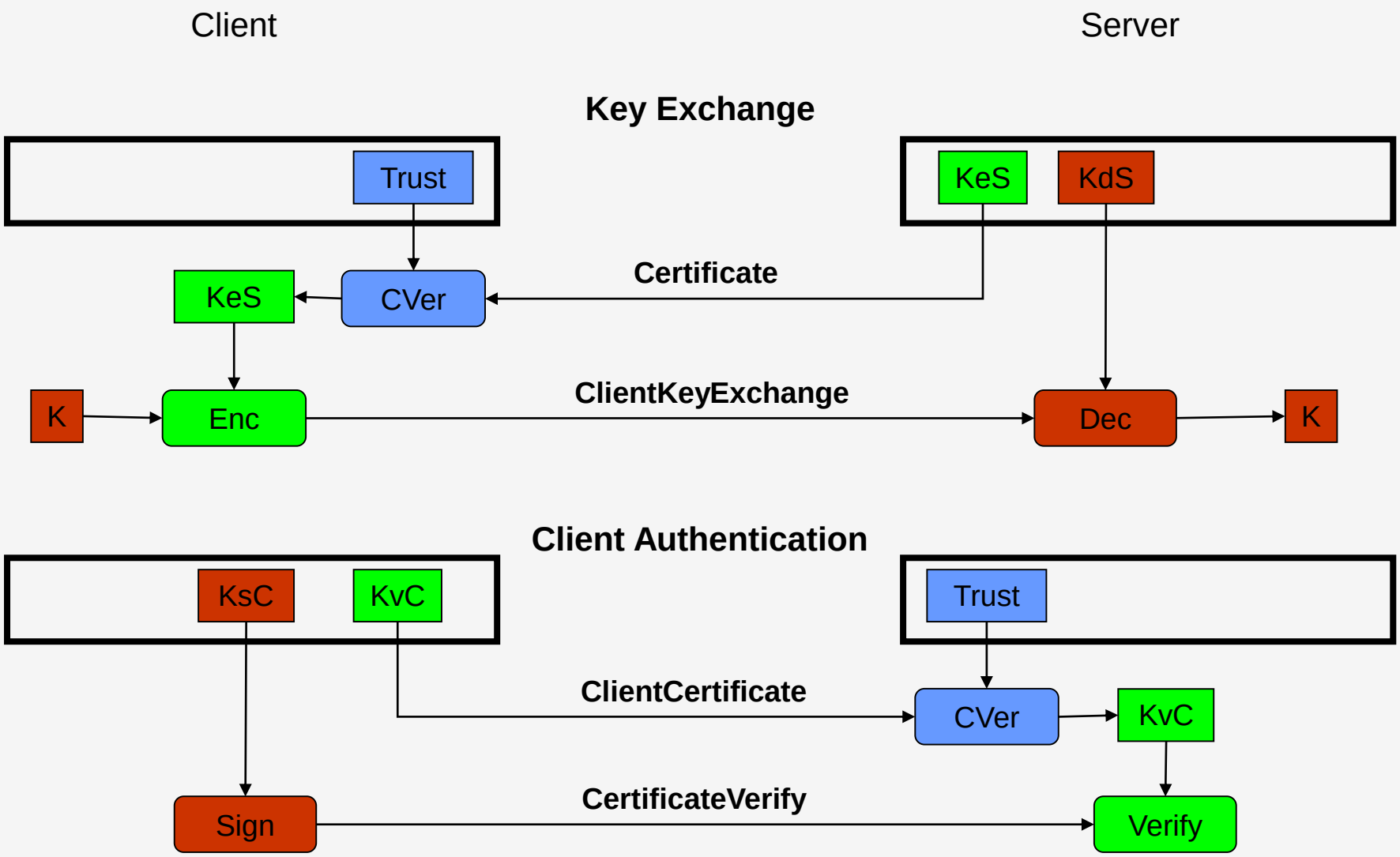
Handshake Protocol (3): RSA based

- **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)

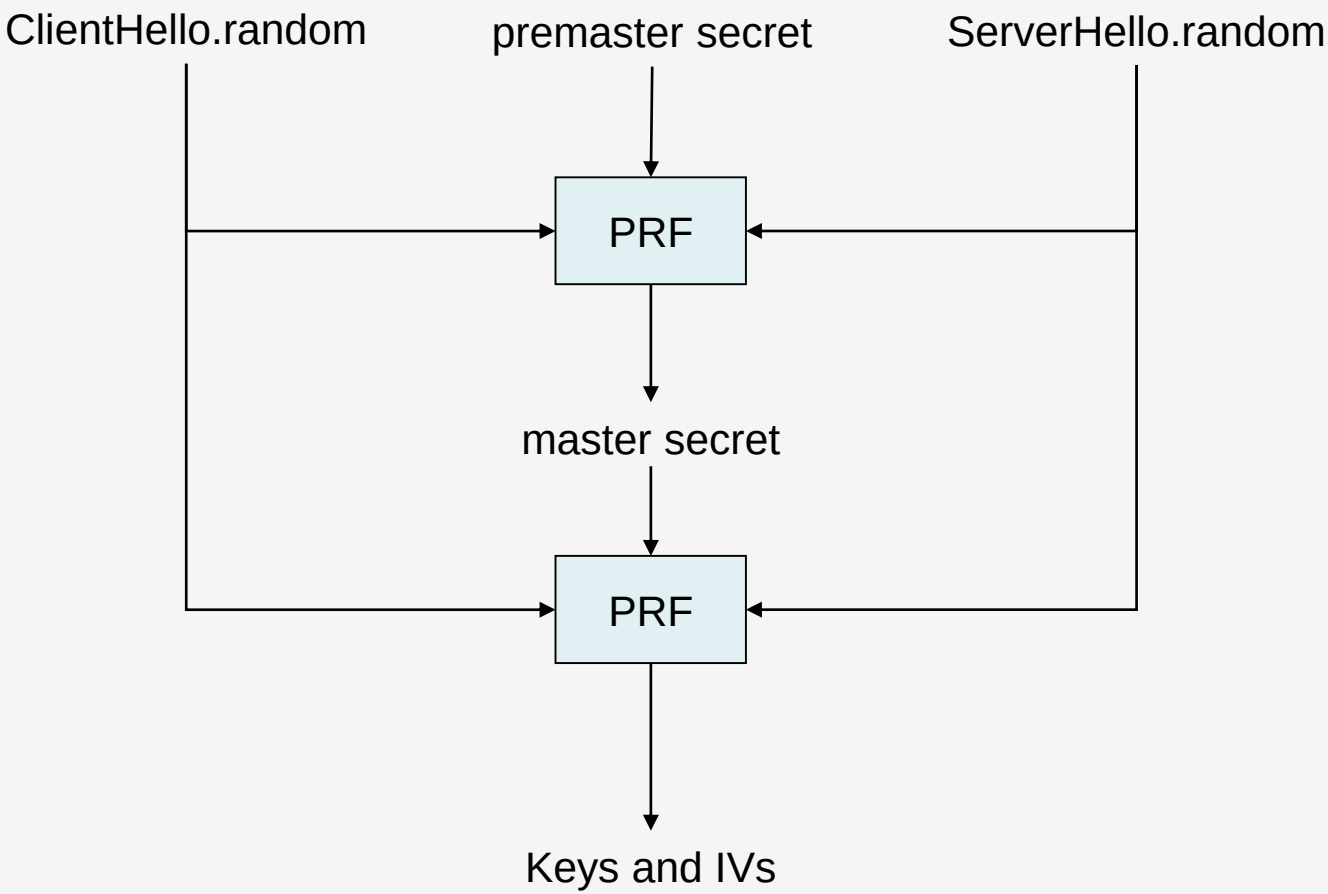
Yet another one



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

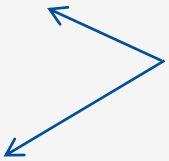
- The endpoint authentication uses different techniques for the client and the server

- 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

Master Secret based on Diffie-Hellman

1. Client and Server agree about parameters p e g
2. Server:
   - Chooses a
   - Computes $Y=g^a \bmod p$
   - Sends Y, $Sign(K_sB)(Y)$
3. Client:
   - Chooses b
   - Computes and sends $X = g^b \bmod p$
   - Computes $Y^b \bmod p = g^{ab} \bmod p = Z$
4. Server:
   - Computes $X^a \bmod p = g^{ab} \bmod p = Z$



Pre master secret

- The atacker knows p, g and sees $g^a \bmod p$, $g^b \bmod p$
- An efficient algorithm to determine, in general, a e b, is still to be found (if there is any)

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

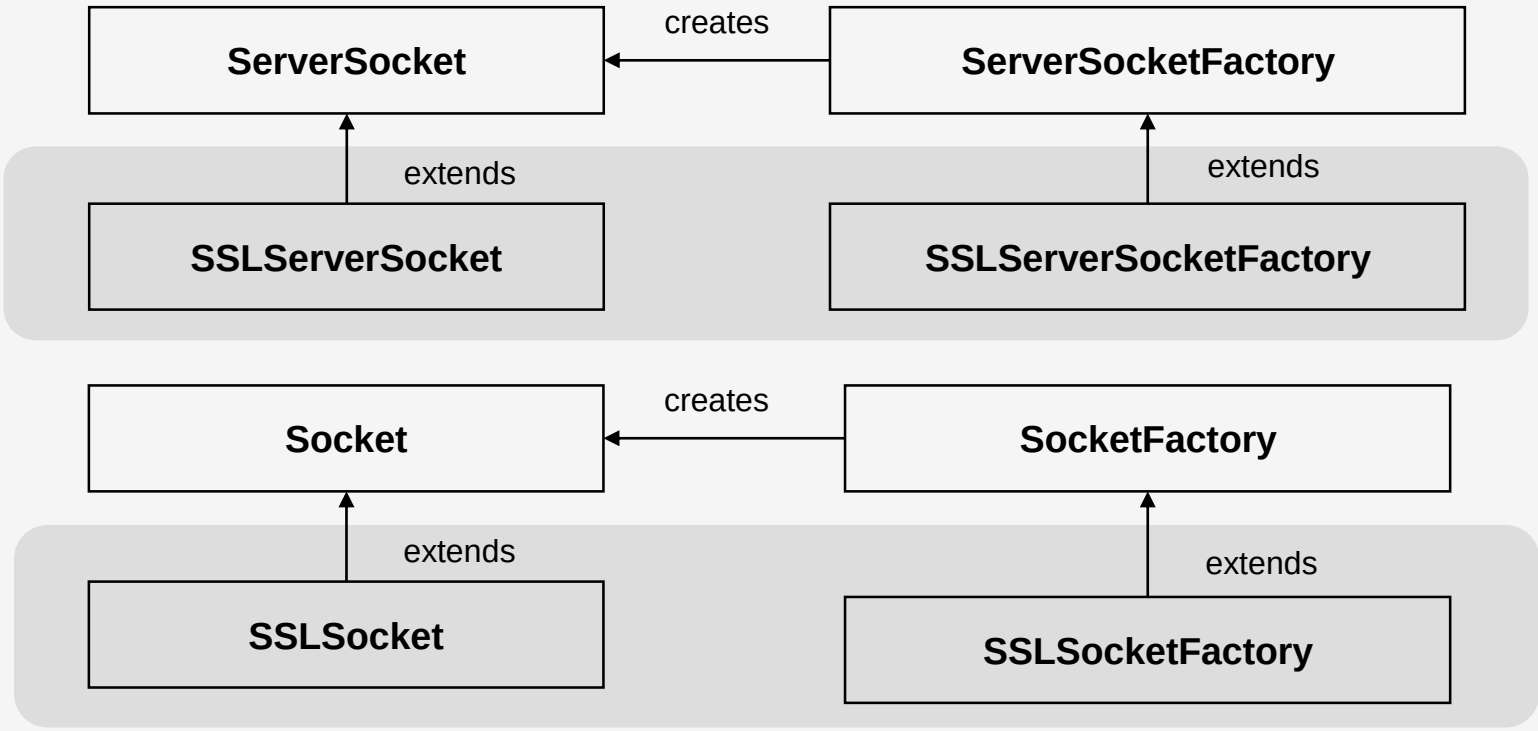
- HTTP over TLS
- Default port: 443
- Check between the URI and the certificate
  - **subjectAltName** extension of type **dNSName** (if present)
  - the (most specific) **Common Name** field in the **Subject** field
- Proxies
  - CONNECT method requests the proxy to create a TCP tunnel to the endpoint

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

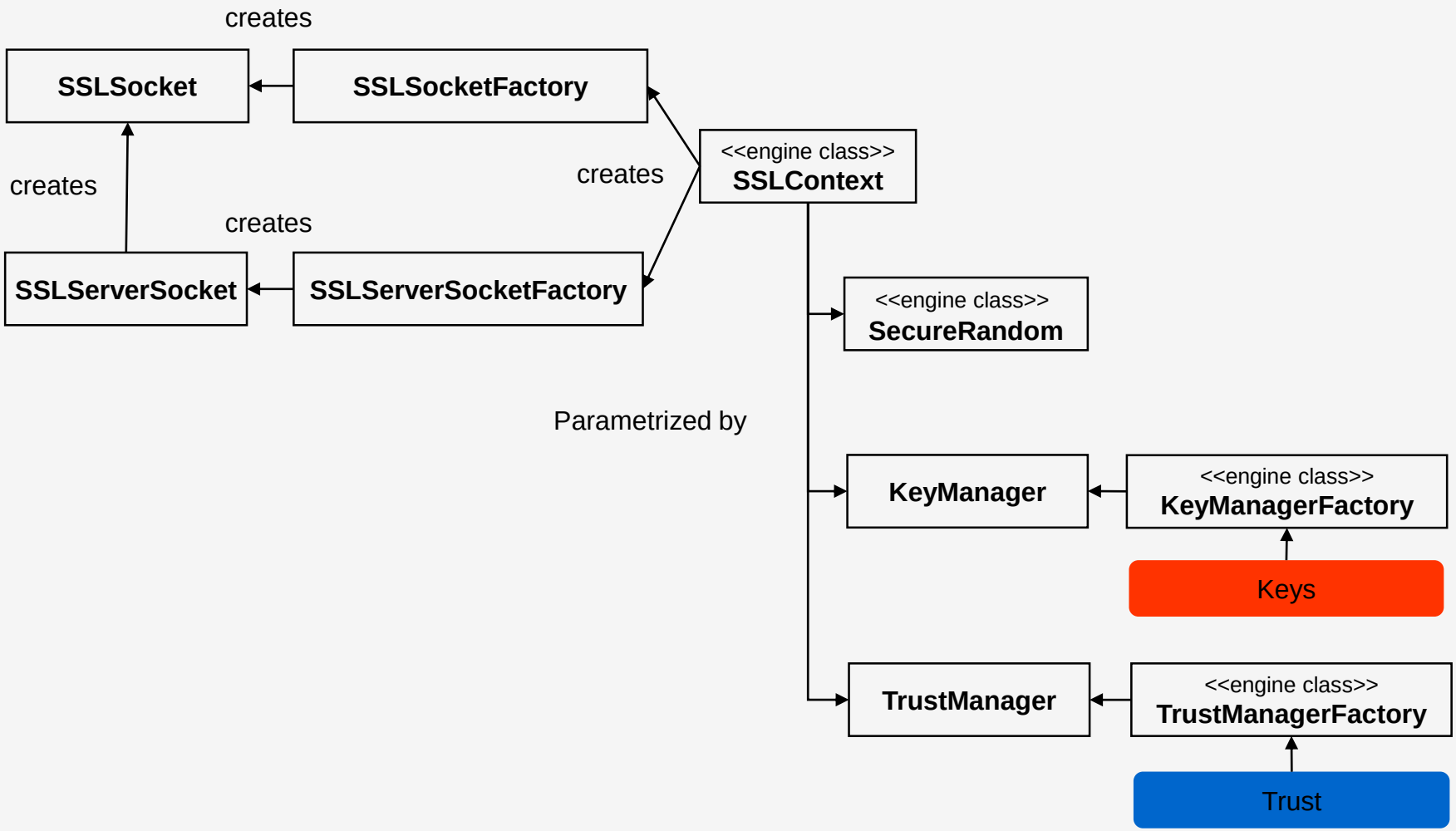
- The regular sockets use factory-based instance creation
  - **ServerSocket** and **Socket** instances are created by **ServerSocketFactory** and **SocketFactory** instances
- The JSSE has specializations for the SSL/TLS protocols



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

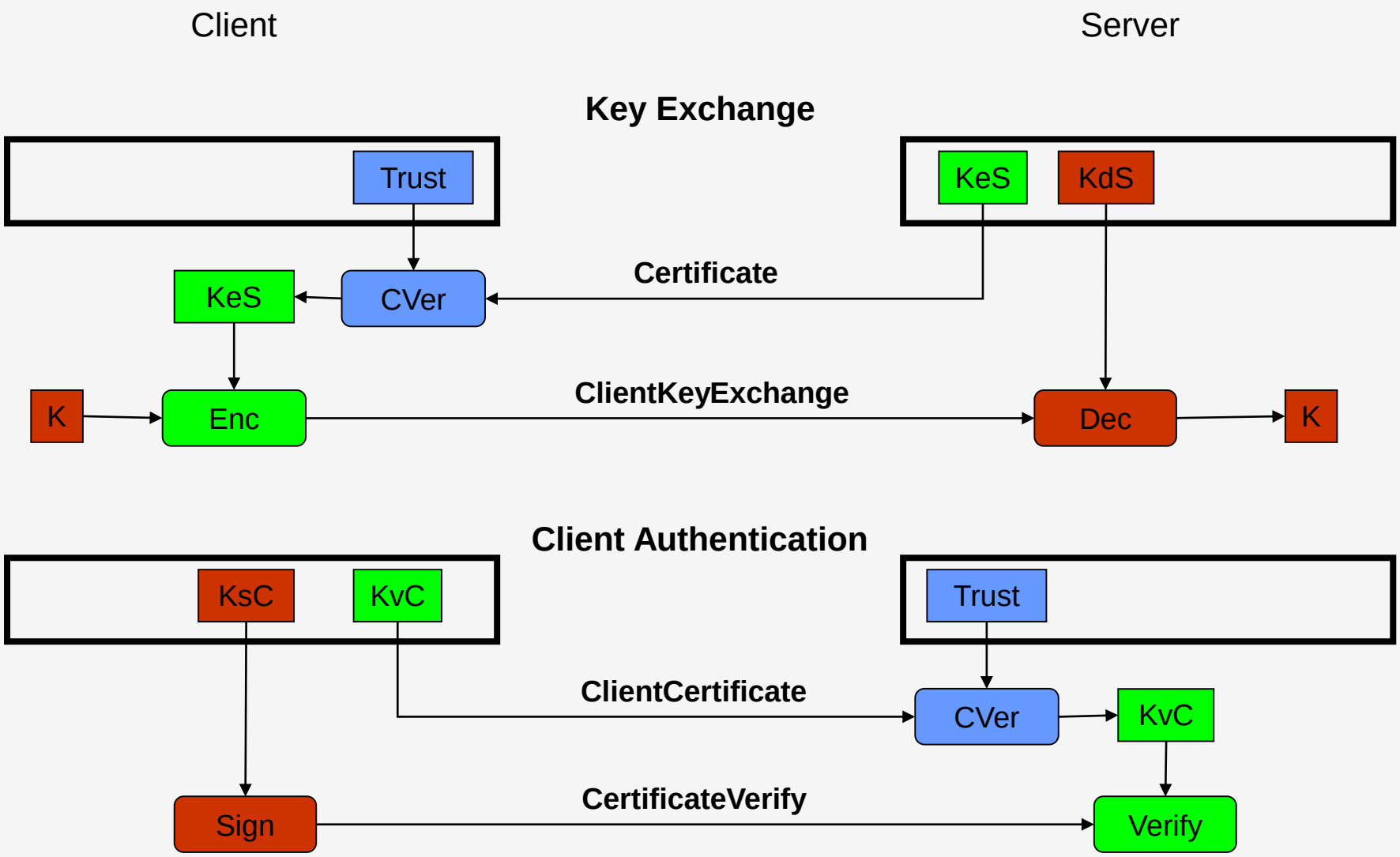
Factory based architecture



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