

Transport Layer Security

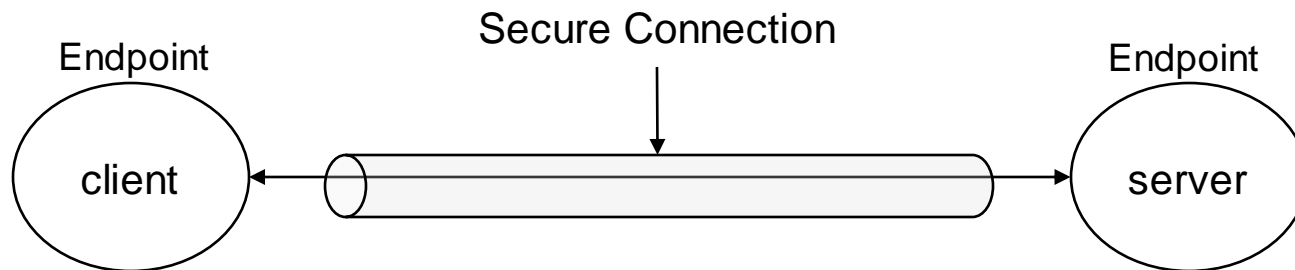
Protocolo criptográfico

Alguma história

- SSL - Secure Sockets Layer
 - Protocolo proprietário da Netscape
 - 1994, v1.0 – não publicado
 - 1994, v2.0 – vulnerabilidades críticas
 - 1995, v3.0 – muito dessiminado, IETF draft
- TLS – Transport Layer Security
 - 1999, IETF RFC 2246
 - Semelhante mas incompatível com SSL v3.0
 - versão mais recente 1.3, <https://tools.ietf.org/html/rfc8446>

Objectivos do protocolo SSL/TLS

- O Transport Layer Security (TLS) é um protocolo que fornece um canal seguro entre dois *endpoints*. O canal seguro tem três propriedades:
 - **Confidencialidade**: ninguém além dos *endpoints* pode ver o conteúdo dos dados transmitidos
 - **Integridade**: podem ser detectadas quaisquer alterações feitas nos dados durante a transmissão
 - **Autenticação**: pelo menos um *endpoint* do canal precisa ser autenticado para que o outro endpoint tenha garantias sobre a quem está ligado.



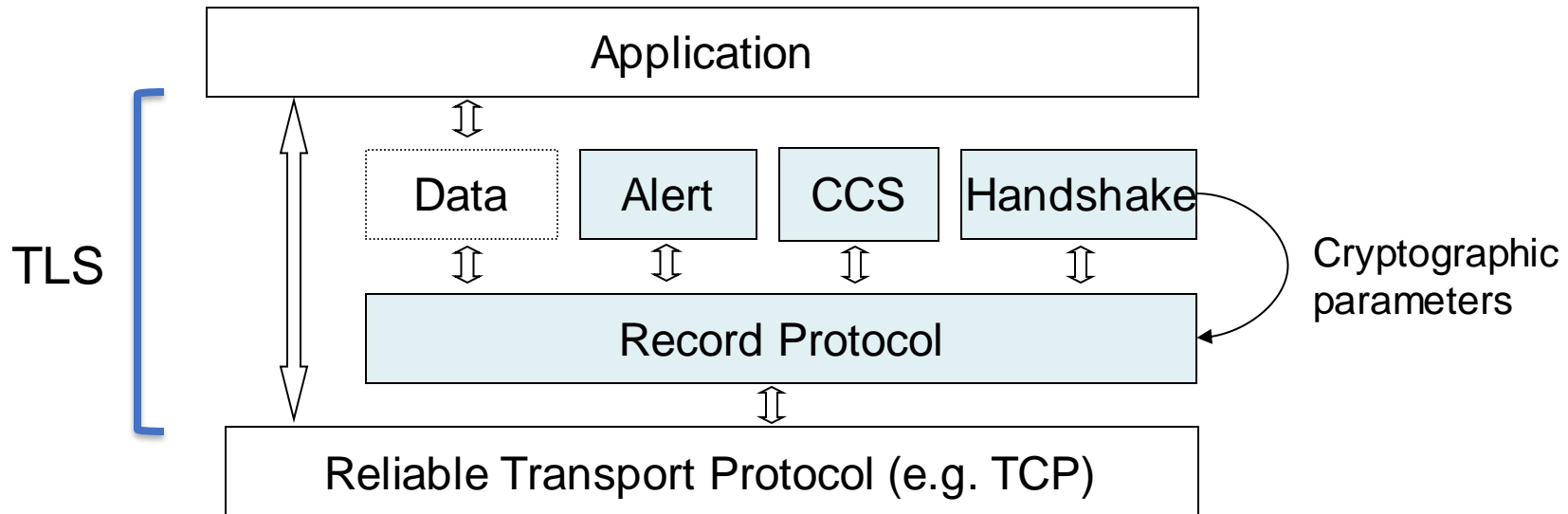
Camada TLS

- TLS fica entre a camada de transporte e aplicação
 - Os dados não protegidos são fornecidos ao TLS pela camada de aplicação
 - O TLS cifra e autentica os dados que envia/recebe da camada de transporte
 - O TLS requer uma camada de transporte fiável (TCP)

Application Layer
TLS Layer
Transport Layer (TCP Protocol)
Network Layer (IP protocol)
Data Link Layer
Physical Layer

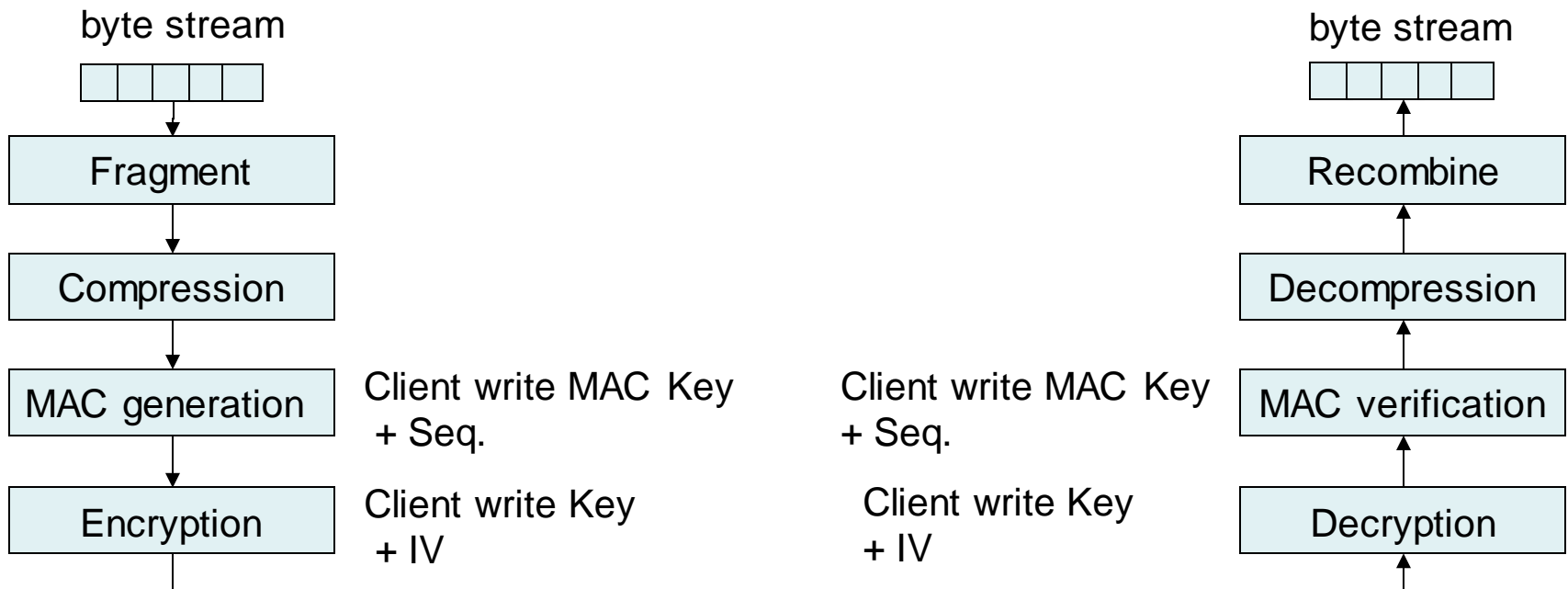
Sub-protocolos

- Dividido em dois subprotocolos principais
- *Record protocol*
 - Requer um protocolo de transporte confiável
- *Handshake protocol*
 - Lida com a criação e gestão de conexão segura, ou seja, o estabelecimento seguro dos parâmetros criptográficos do record protocol

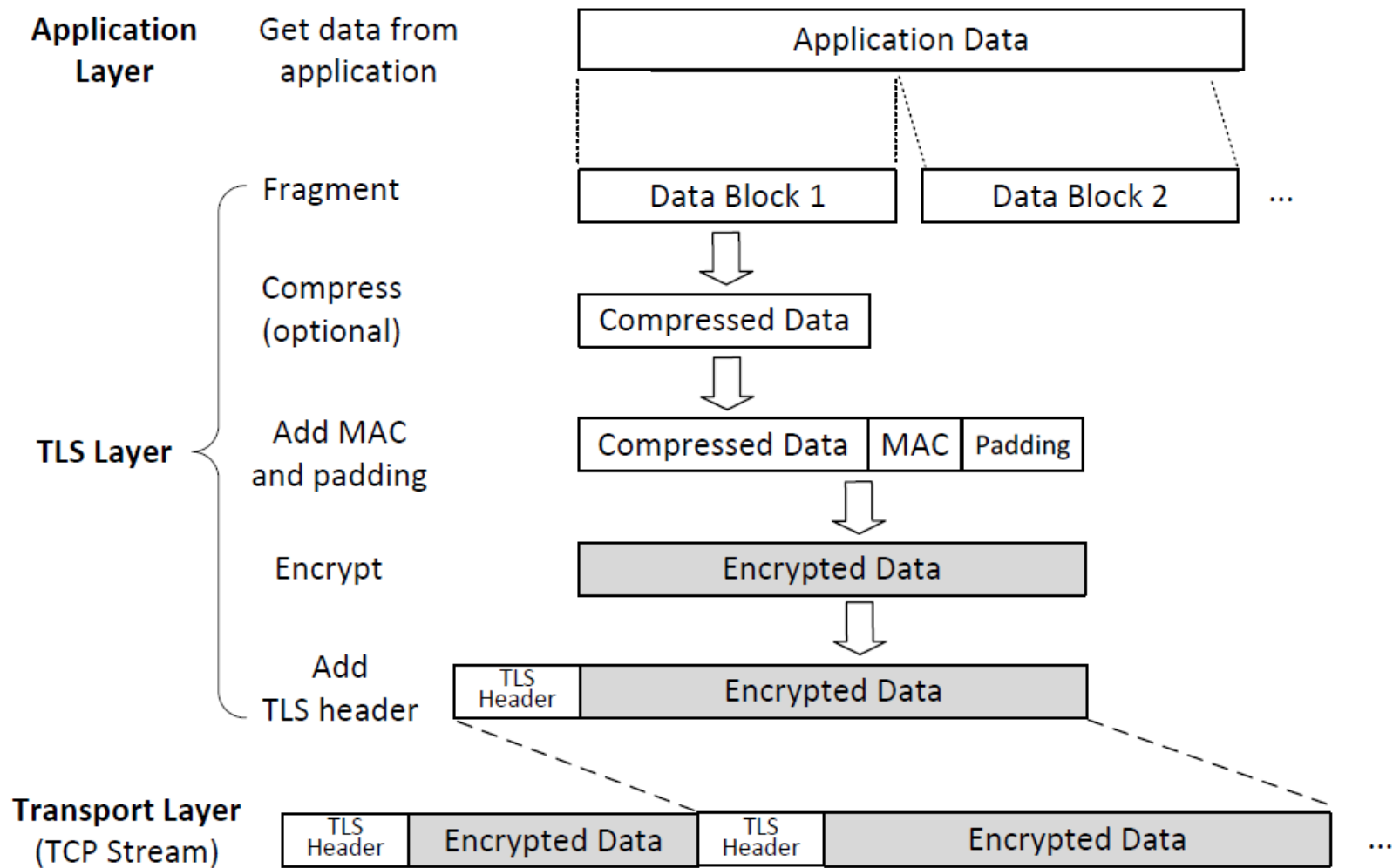


Record Protocol

- Fragmenta, comprime, autentica (MAC) e depois cifra
- A mesma ligação TCP, duas direções independentes de dados
 - Chaves, IVs e número de sequência diferentes (*client write* e *server write*)



Detalhe do *record protocol*



Notas sobre o *record protocol*

- Repetições de mensagens
 - Detectado pelo número de sequência
- Reflexão da mensagem
 - Chaves MAC separadas para cada direção
- Reutilização de *keystream* (criptografia simétrica baseada em *streams*)
 - Chaves de criptografia e IVs separados para cada direção
- Análise de tráfego
 - Chaves de criptografia separadas

Esquemas criptográficos

- Os esquemas criptográficos usados dependem do *cipher suite* acordado
- O *cipher suite* e os algoritmos de compressão são negociados pelo protocolo de handshake
- Examplos
 - TLS_NULL_WITH_NULL_NULL
 - TLS_RSA_WITH_3DES_EDE_CBC_SHA
 - TLS_RSA_WITH_RC4_128_SHA
- Um *cipher suite* define
 - A função de hash usada pelo HMAC (e.g. SHA)
 - O equemas simétrico (e.g. 3DES_EDE_CBC or RC4_128)
 - Suporta modos de bloco ou *stream*
 - Esquema de estabelecimento de chaves (RSA or DH)

Handshake protocol

- Responsável por
 - Negociação dos parâmetros de operação
 - Autenticação dos *endpoints*
 - Estabelecimento de chave segura
- Autenticação de *endpoint* e estabelecimento de chave
 - A autenticação é opcional em ambas as extremidades
 - Suporta várias técnicas criptográficas:
 - Transporte de chave (por exemplo, RSA)
 - Acordo de chave (por exemplo, DH)
- Cenário típico na internet (HTTPS)
 - Transporte de chave baseado em RSA usando certificados X.509
 - Autenticação de servidor obrigatória
 - Autenticação de cliente opcional

Handshake Protocol: resumo

- Quando é usado RSA para transporte de chave
 - $C \leftrightarrow S$: negociação dos algoritmos a serem usados
 - $C \leftarrow S$: certificado de servidor
 - $C \rightarrow S$: segredo aleatório cifrado com a chave pública do servidor
 - $C \leftarrow S$: prova de posse do segredo aleatório
- Se for necessário autenticação de cliente
 - $C \leftarrow S$: O servidor solicita o certificado de cliente
 - $C \rightarrow S$: certificado de cliente
 - $C \rightarrow S$: prova de posse da chave privada, assinando as mensagens anteriores

Handshake Protocol (1): RSA based

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

Protocolo HTTPS

- HTTP sobre TLS
- Porta por omissão: 443
- Verificar entre o URI e o certificado
 - extensão **subjectAltName** do tipo **dNSName** (se existir)
 - o campo **Common Name** no campo **Subject field**

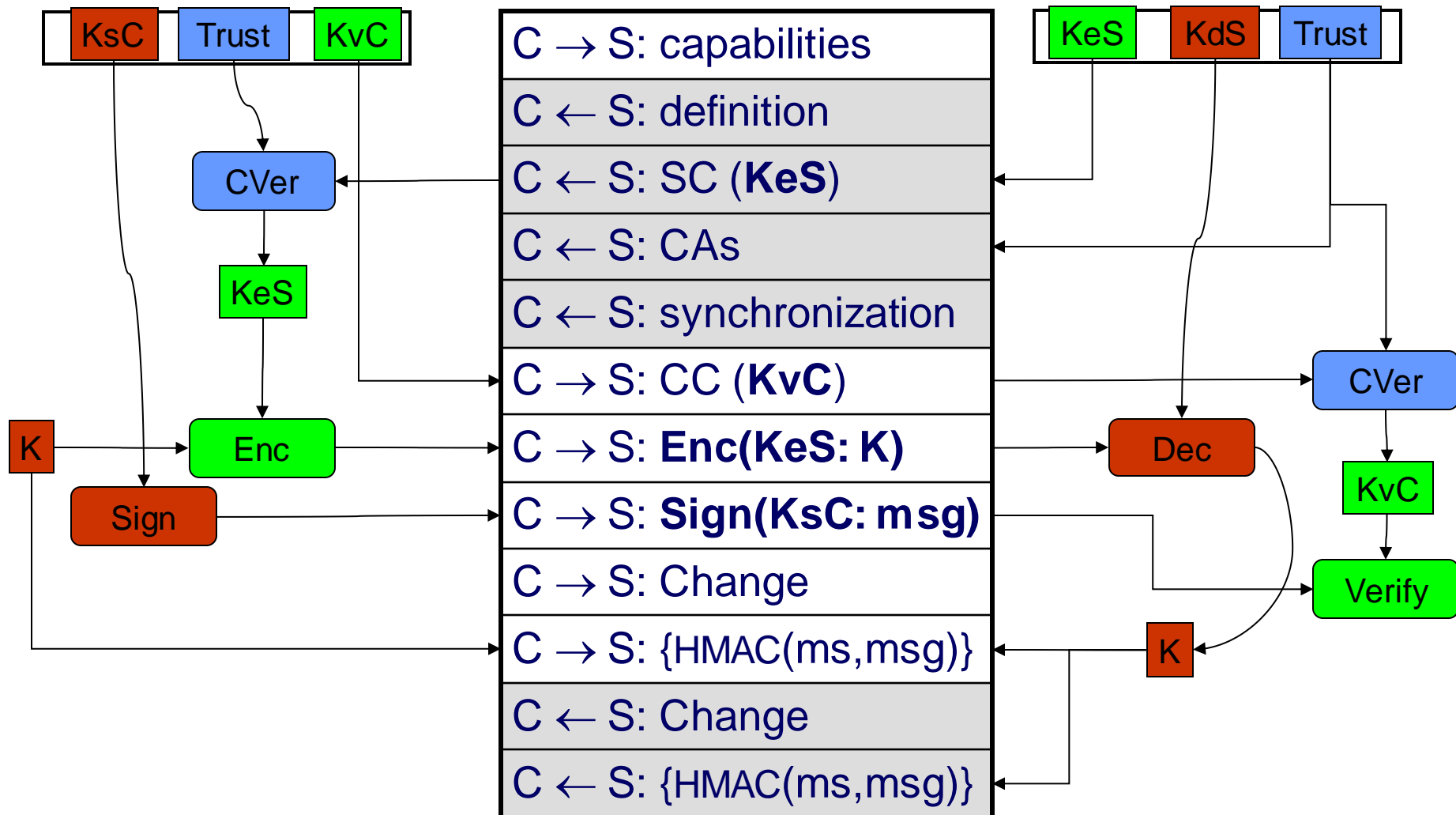
Demonstração

- HTTPS
- Repositório com raízes de confiança
- Ataques

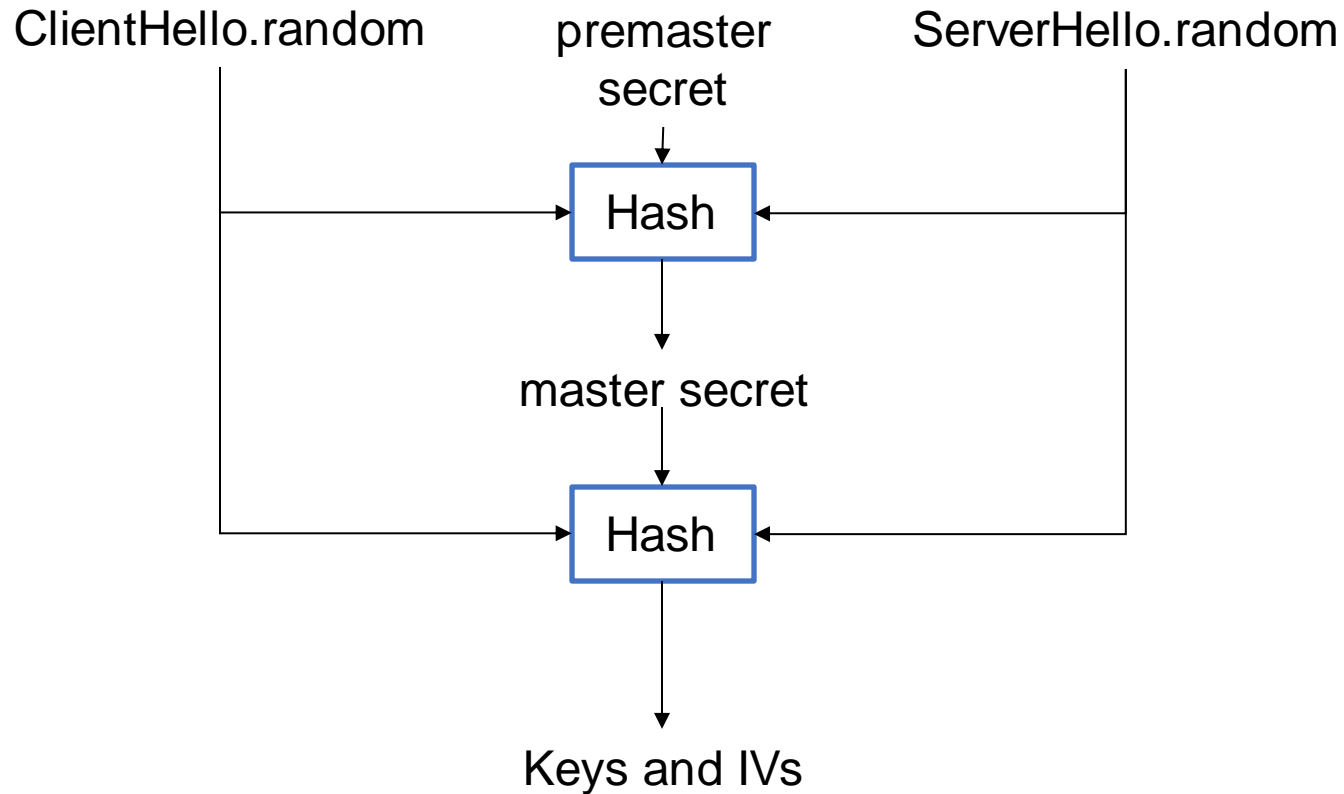
Outra vista

cliente

servidor



Derivação de chaves



Alteração e repetição de mensagens de handshake

- Alteração de mensagens de *handshake* é detetado com a mensagem **Finished**
 - A mensagem Finished garante que ambos os endpoints recebem a mesma mensagem
- Repetição de mensagens de Handshake
 - **ClientHello** and **ServerHello** contém valores aleatórios, diferentes para cada handshake
 - Implica que a mensagem **Finished** é diferente para cada *handshake*

Perfect forward secrecy

- A troca de chaves com RSA implica que o browser usa a chave pública do servidor para cifrar o *pre master secret*
 - O servidor decifra o *pre master secret* usando a chave privada
- Este processo é seguro e garante confidencialidade do *pre master secret*
- O que acontece se a chave privada for comprometida?
 - O *pre master secret* dos handshakes seguintes e dos anteriores (guardados pelo atacante) podem ser decifrados
- *Perfect forward secrecy* é a propriedade do handshake que garante que, se a chave privada for comprometida, não é possível decifrar *master secret* anteriores (e consequentemente não é possível decifrar mensagens do record protocol)

Master Secret baseado em *Diffie-Hellman*

- Cliente e servidor escolhem parâmetros p e g
- Servidor:
 - Escolhe a e calcula $Y = g^a \bmod p$
 - Envia Y , $Sign(KsB)(Y)$
- Cliente:
 - Escolhe b calcula e envia $X = g^b \bmod p$
 - Computes $Y^b \bmod p = g^{ab} \bmod p = \mathbf{Z}$
- Servidor:
 - Calcula $X^a \bmod p = g^{ab} \bmod p = \mathbf{Z}$
- O atacante conhece p , g and vê $g^a \bmod p$, $g^b \bmod p$
- É computacionalmente difícil determinar a e b

Pre master secret

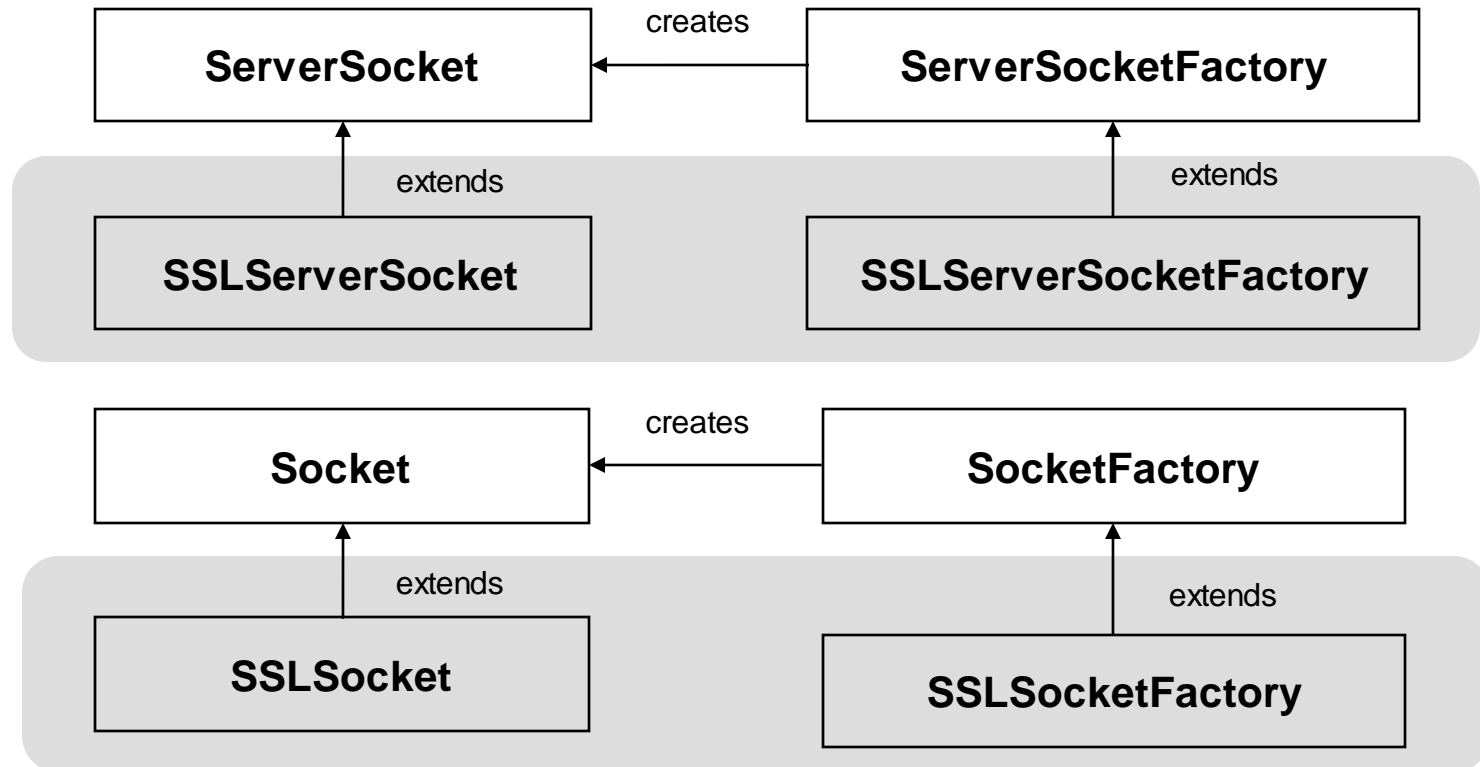
Ataques a autoridades de certificação

- A validação do certificado do servidor depende de uma raiz de confiança
- As raízes de confiança são certificados emitidos por autoridades de certificação
- A autoridade de certificação Diginotar foi comprometida em 2011 e o atacante conseguiu emitir um certificado para serviços Google
- Um ataque de *man-in-the-middle* substituiu o certificado do Gmail pelo novo certificado, conseguindo ver mensagens sem quebrar o TLS

Sockets TLS em Java

Sockets e fábricas de sockets

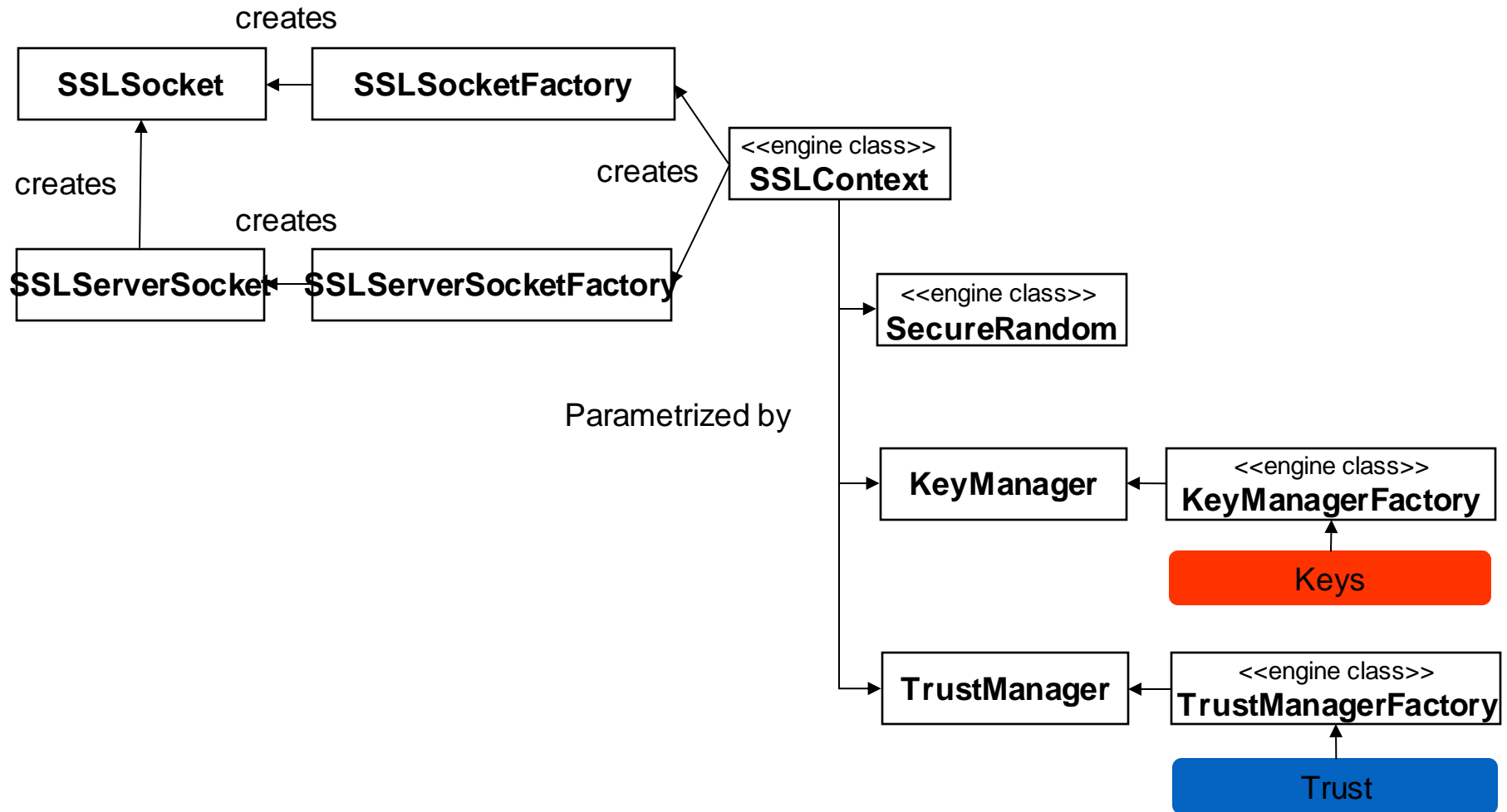
- Os sockets regulares podem ser instanciados através de classes fábrica
- Instâncias **ServerSocket** e **Socket** são criadas por **ServerSocketFactory** e **SocketFactory** instances



Funcionalidade

- **SSLSocketFactory** and **SSLServerSocketFactory**
 - Obtenção de cipher suites suportados por missão
 - Criação de instâncias de sockets
- **SSLSocket** and **SSLServerSocket**:
 - Inicia o handshake e recebe notificações da sua conclusão
 - 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

Arquitetura baseada em fábricas de objectos



Raizes de confiança por omissão em Java

```
public class SSLDemo {  
    public static void main(String[] args) throws IOException {  
        SSLSocketFactory sslFactory =  
            HttpURLConnection.getDefaultSSLSocketFactory();  
        SSLSocket client = (SSLSocket)  
            sslFactory.createSocket("docs.oracle.com", 443);  
        client.startHandshake();  
        SSLSession session = client.getSession();  
        System.out.println(session.getCipherSuite());  
        System.out.println(session.getPeerCertificates()[0]);  
        client.close();  
    }  
}
```

Usa raízes de confiança por omissão em <java_home>\jre\lib\security\cacerts