FACULTEIT INGENIEURSWETENSCHAPPEN

Network and Computer Security

Chapter 3 – Network and communication security

Prof. dr. ir. Eli De Poorter

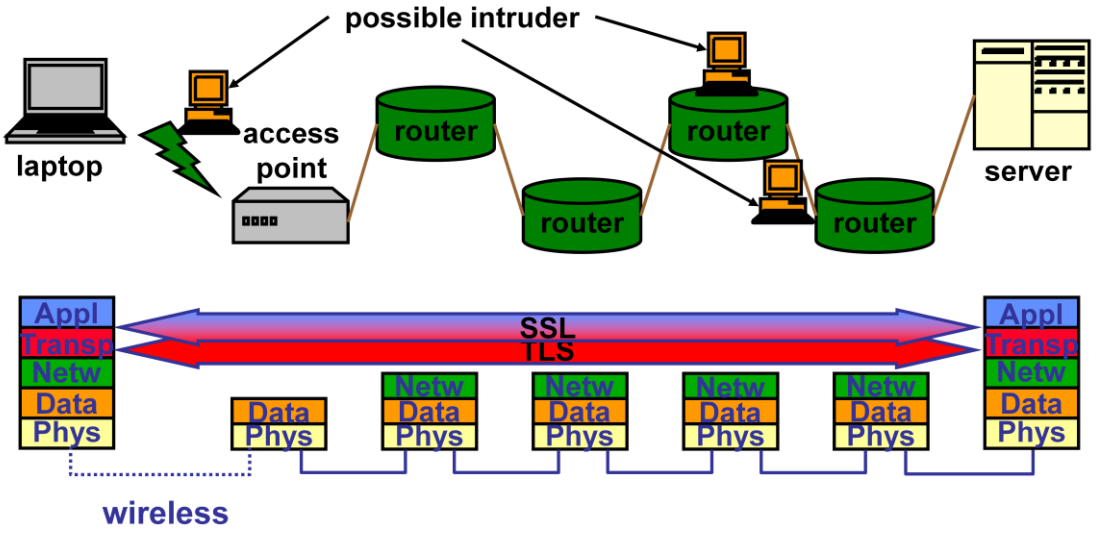
Overview

- Network model
- Secure configuration of devices
  - SSH (Secure Shell)
- Exchanging keys
  - Out of band
  - Diffie-Hellman
  - Asymmetric encryption
  - Trusted third party
    - Key Distribution Centre (KDC)
    - Public Key Infrastructure (PKI)
- Secure networking protocols
  - Transport layer: TLS & SSL
  - Network layer: IPSec & VPN
  - Data link layer: WEB & WPA
- Firewalls
  - Packet filter
  - Circuit-level gateway
  - Application-level gateway (proxy)

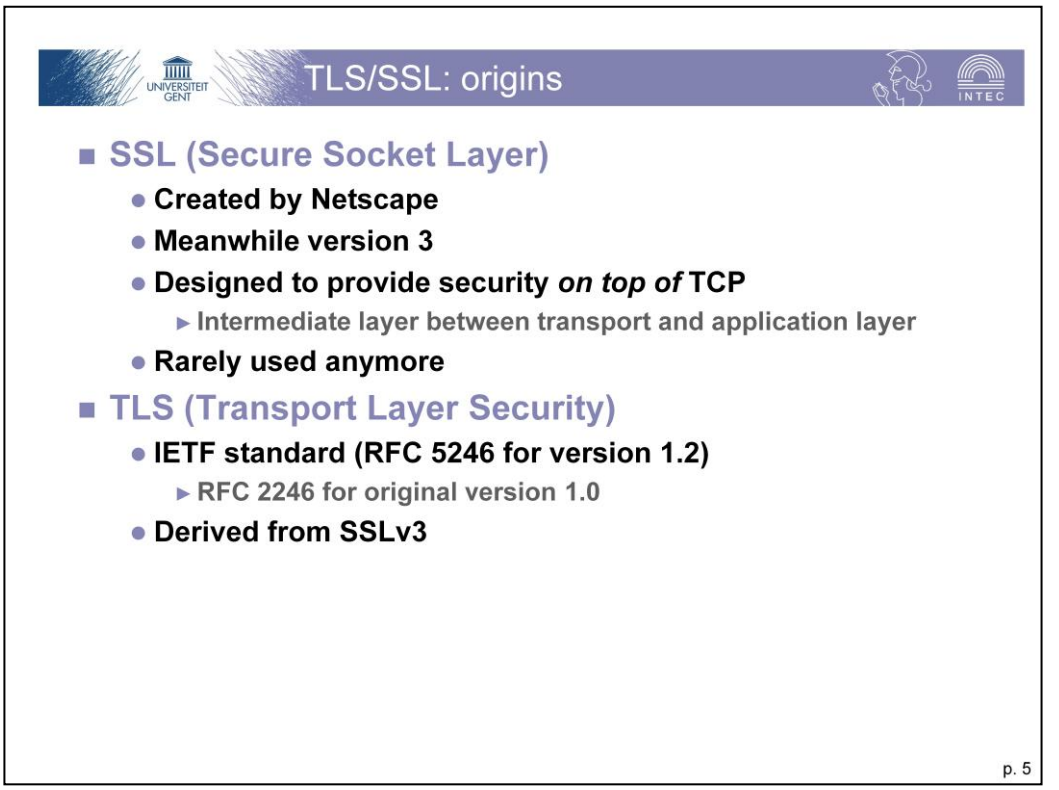
Overview

- Network model
- Secure configuration of devices
- Exchanging keys
- Secure networking protocols
  - **Transport layer: TLS & SSL**
  - Network layer: IPSec & VPN
  - Data link layer: WEB & WPA
- Firewalls

“Transport layer” security

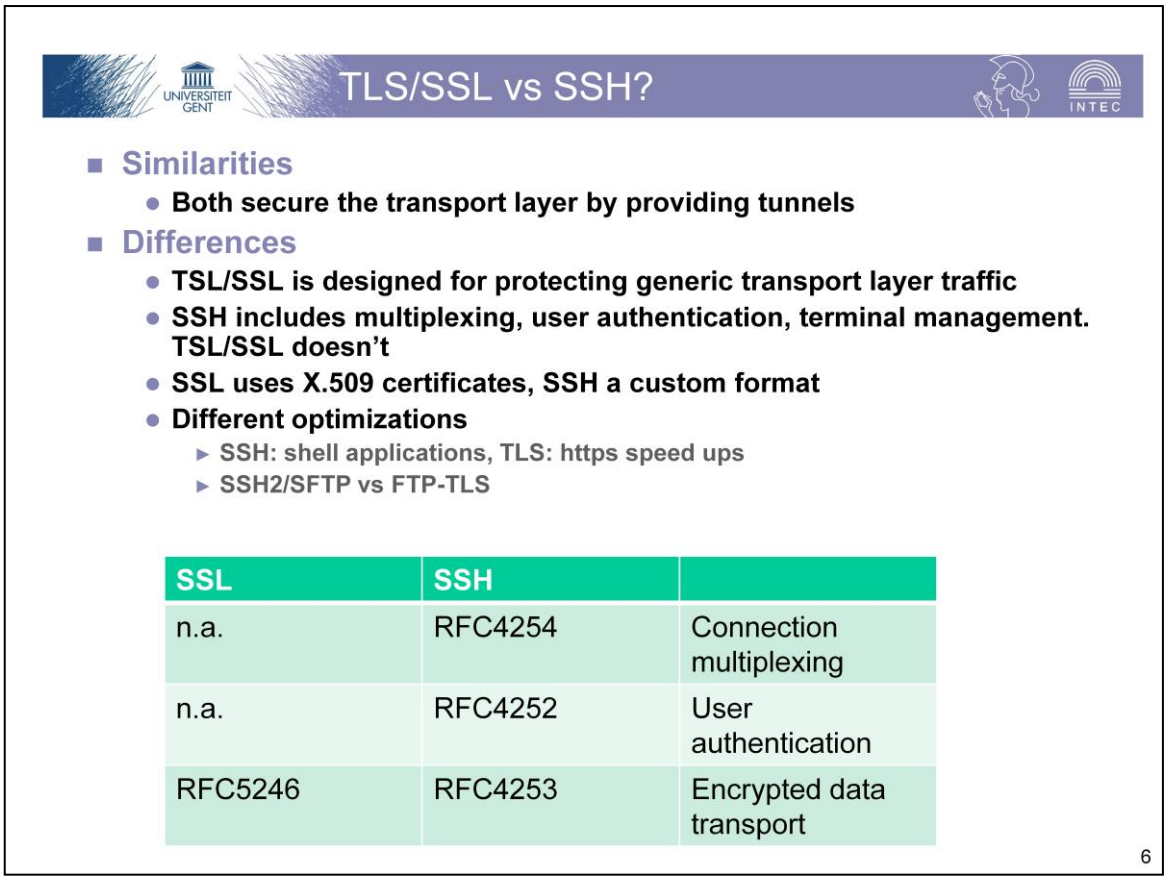


p. 4



The SSL protocol was originally developed at Netscape to enable ecommerce transaction security on the Web, which required encryption to protect customers' personal data, as well as authentication and integrity guarantees to ensure a safe transaction. To achieve this, the SSL protocol was implemented under the application layer, directly on top of TCP, enabling protocols above it (HTTP, email, instant messaging, and many others) to operate unchanged while providing communication security when communicating across the network. When SSL is used correctly, a third-party observer can infer the connection endpoints, type of encryption, as well as the frequency and an approximate amount of data sent, but cannot read or modify any of the actual data.

When the SSL protocol was standardized by the IETF, it was renamed to Transport Layer Security (TLS). Many use the TLS and SSL names interchangeably, but technically, they are different, since each describes a different version of the protocol. TLS only differs in a few points from SSLv3. It generally concerns technical implementations of some algorithms. E.g. the session keys are derived in a different way (using a pseudorandom function PRF), and a different MAC function is used. Different "Alert Codes" are used, with a different categorisation in fatal errors and warnings. We limit ourselves to the discussion of TLS 1.2 since SSL is rarely used anymore. One may still occasionally encounter TLS 1.0 or TLS 1.1 for secure websites, although TLS 1.2 is now supported by any up-to-date version of any common browser (be it IE, Firefox, Chrome, Opera, etc.) (check the configuration parameters of your browser).



SSL is a transport layer method for protecting data transported over a network, whereas SSH is a network application for logging in and sharing data with a remote computer.

TLS has goals and features similar to those of the SSH Transport and User Authentication protocols. It provides a single, full-duplex byte stream to clients, with cryptographically assured privacy and integrity, and optional authentication. It differs from SSH in the following principal ways:

• TLS server authentication is optional: the protocol supports fully anonymous operation, in which neither side is authenticated. Such connections are inherently vulnerable to man-in-the-middle attacks. In SSH-TRANS, server authentication is mandatory, which protects against such attacks. Of course, it is always possible for a client to skip the step of verifying that the public key supplied by the server actually belongs to the entity the client intended to contact (e.g. using the /etc/ssh_known_hosts file). However, SSH-TRANS at least demands going through the motions.

• TLS lacks user authentication because it doesn't need it (TLS just needs to authenticate the two connecting interfaces, which SSH can also do). When using TLS, authentication is done at higher layer such as in the webbrowser.

• In TLS both client and server authentication are done with X.509 public-key certificates whereas SSH has its own format. This makes TLS a bit more cumbersome to use than SSH in practice, since it requires a functioning public-key infrastructure (PKI) to be in place, and certificates are more complicated things to generate and manage than SSH keys. However, a PKI system provides scalable key management, which SSH currently lacks.

• TLS does not provide the range of client authentication options that SSH does; public-key is the only option.

• Since SSL is a transport layer protocol, it lacks connection multiplexing capabilities. TLS does not have the extra features provided by the SSH Connection Protocol (SSH-CONN). SSH-CONN uses the underlying SSH-TRANS connection to provide multiple logical data channels to the application, as well as support for remote program execution, terminal management, tunneled TCP connections, flow control, etc.

•The transport layer protection in SSH is similar in capability to TLS. TLS and SSH both provide the cryptographic elements to build a tunnel for confidential data transport with checked integrity. To this end, they use similar techniques, and may suffer from the same kind of attacks, so they should provide similar security (i.e. good security) assuming they are both properly implemented.

Conceptually, you could take SSH and replace the tunnel part with the one from SSL. You could also take HTTPS and replace the SSL tunnel with SSH-with-data-transport and a hook to extract the server public key from its certificate. There is no scientific impossibility and, if done properly, security would remain the same. However, there is no widespread set of conventions or existing tools to realize this. So we do not use SSL and SSH for the same things, but that's because of what tools historically came with the implementations of those protocols, not due to a security related difference. And whoever implements or uses SSL or SSH would be well advised to look at what kind of attacks were tried on both protocols.

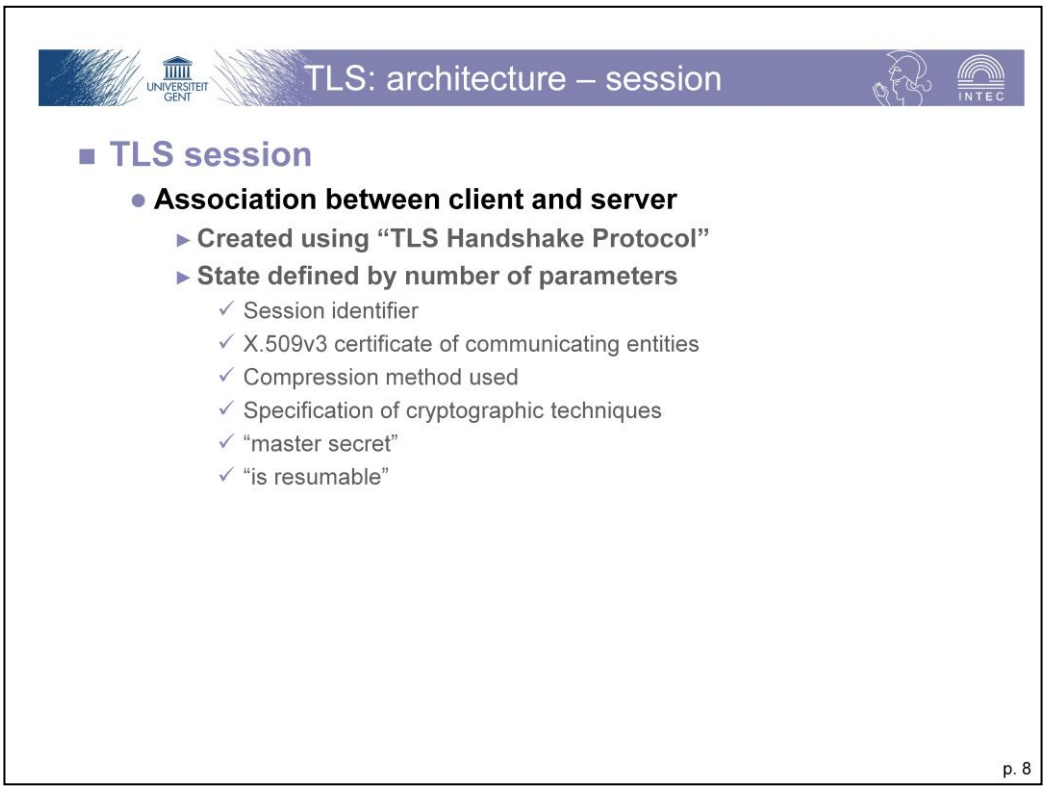
Although SSL/TLS is most well-known for its use in setting up secure http connections (i.e.: HTTPS), other application protocols can also utilize SSL/TLS tunnels. For example, there is an enhancement to standard FTP (as defined in RFC 959), which uses the same FTP commands (and protocol) over secure sockets, i.e. over SSL/TLS. This enhancement is defined in RFC 4217 and is known under the names of FTPS, FTP-TLS, and FTP-over-SSL. This is not the same protocol as the SFTP protocol, which also provides secure file transfer (over SSH tunnels) but is not part of the FTP standard and as such not fully FTP compatible. As such, the two protocols are completely different, not related to each other and not (!) compatible with each other. SSH2/SFTP is more popular in the Unix world while FTP-TLS is more popular in the Windows World.

TLS: architecture – connection

- **TLS connection**
  - **Transport between communicating entities**
    - **Based on peer-to-peer relation between client and server**
    - **Associated with 1 TLS session**
    - **State defined by number of parameters:**
      - ✓ Random numbers for each connection
      - ✓ Keys for symmetric encryption
      - ✓ Initialisation vectors (IV) in CBC-mode
      - ✓ Sequence number for received/sent messages
      - ✓ Key for computation of MAC

p. 7

A connection is a communication channel between a client and a server. Connections are usually short lived and servers are usually configured to timeout a connection if it is left idle for too long.



The difference between a connection and a session is that connection is a live communication channel, and a session is a way of maintaining state on the server side (including a set of negotiated cryptography parameters). You can close a connection, but can keep the session, even store it to disk, and subsequently resume it using another connection, maybe in completely different process, or even after system reboot (of course, stored session should be kept both on the client and on the server). On the other hand, you can renegotiate TLS parameters and create an entirely new session without interrupting connection.

The session identifier is an arbitrary byte sequence, chosen by the server.

It is possible that no certificate is used for one or for both communicating parties.

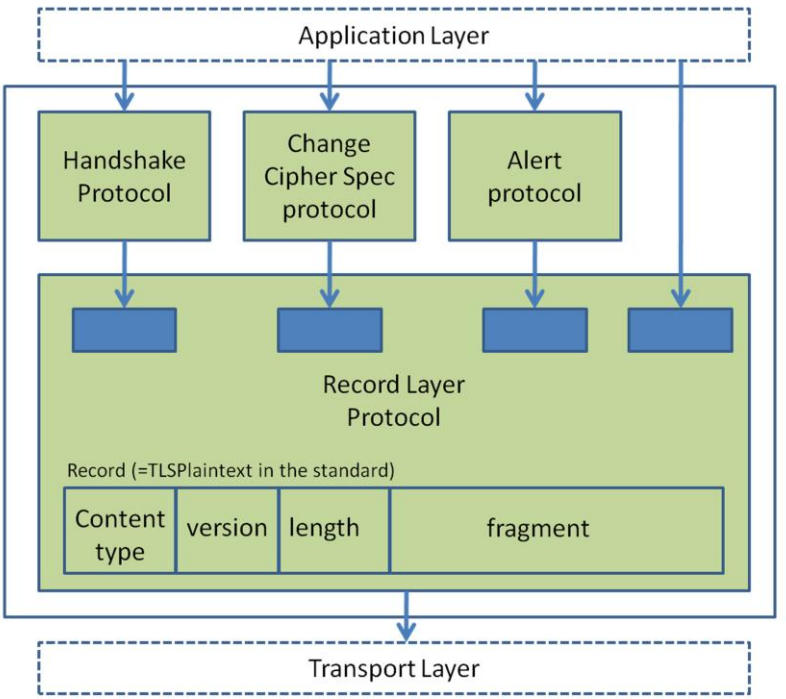
The compression method also is optional.

The "master secret" is secret information (48 bytes) shared by server and client.

The flag "is resumable" indicates whether the session can be used to start new connections.

- **TLS architecture**
  - **Base layer "TLS Record Protocol"**
    - **Security services for protocols in higher layers**
  - **On top of it:**
    - **applications (HTTP, FTP, SMTP, etc.)**
    - **TLS-specific protocols**
      - ✓ "TLS Handshake Protocol"
      - ✓ "TLS Change Cipher Protocol"
      - ✓ "TLS Alert Protocol"

TLS: architecture

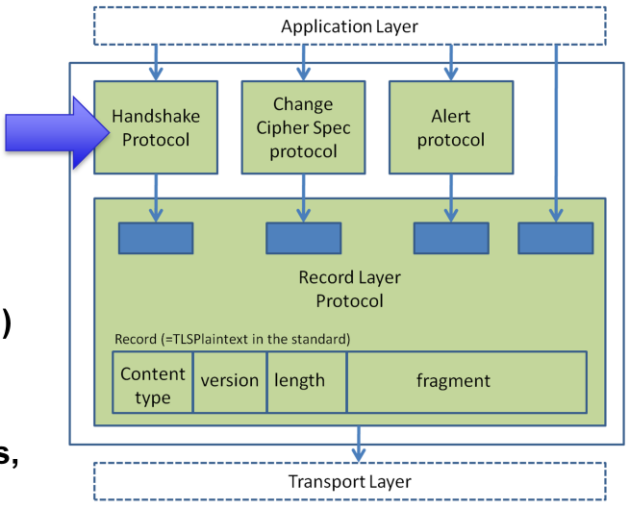


p. 10

The TLS protocol has itself a two layered architecture; the TLS Record layer protocol and the TLS Handshaking protocols. The latter has thee sub-protocols; the Handshake protocol, the Alert protocol and the Change Cipher Spec protocol. On top of these protocols, the application layer resides.

TLS: Handshake protocol

- Allows (reciprocal) authentication of server and client
- Security negotiation
- Message structure:
  - 1 byte for message type (10 types defined)
  - 3 bytes for message length
  - The message contents, with the parameters required for this message type (variable length, empty for some message types)

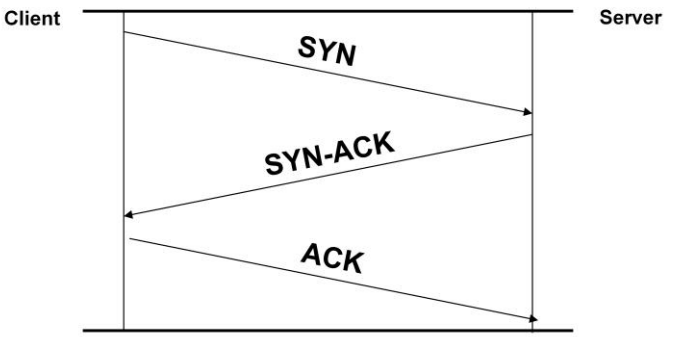


p. 11

Before the client and the server can begin exchanging application data over TLS, the encrypted tunnel must be negotiated: the client and the server must agree on the version of the TLS protocol, choose the ciphersuite, and verify certificates if necessary.

TLS: Handshake protocol

- **Phase 0**
  - **Setting up a TCP connection**

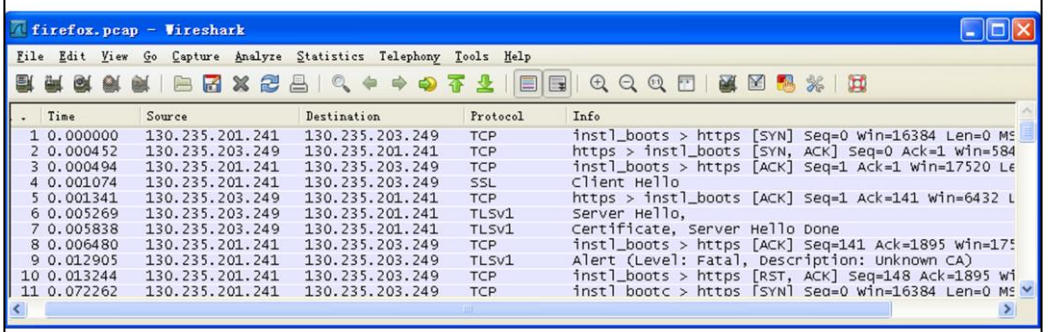


p. 12

The TCP session has to be established before SSL/TLS protocol can start. For setting up a TCP connection, we first complete the TCP three-way handshake.

TLS: Handshake protocol

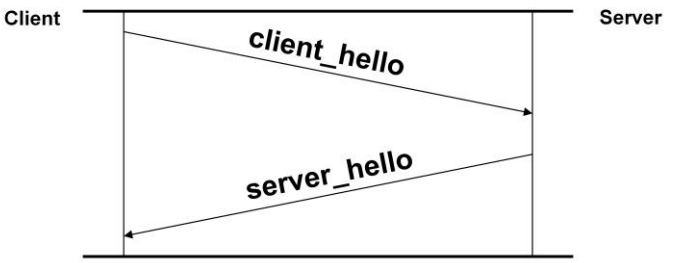
- **Phase 0**
  - **Setting up a TCP connection**



First three packets: setting up a TCP connection

TLS: Handshake protocol

- **Phase 1**
  - **Defining security capabilities**



p. 14

The client sends a message to the server of type "client_hello" with following parameters:

•Version: the highest TLS version supported by the client

•Random: a time stamp (32 bits) and 28 bytes generated by a secure random number generator; to be used as a nonce

•SessionID: non-zero value if the client wants to update the parameters of an existing connection or to create a new connection within an existing session.

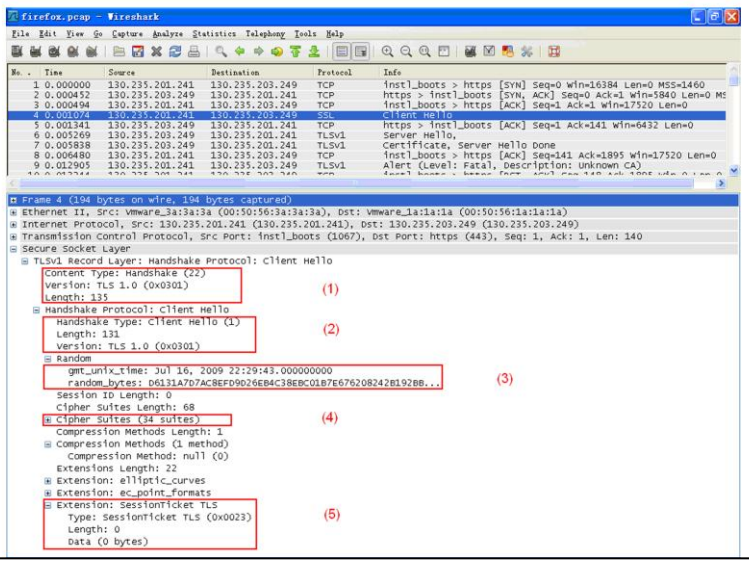
•CipherSuite: a list of combinations of cryptographic algorithms supported by the client (combinations in order of preference); such a combination describes the key exchange method (RSA, DH (fixed, ephemeral, or anonymous), etc.), the encryption specification (algorithm, hash function, keys, etc.), the MAC function, and (starting with version 1.2) the PRF (pseudorandom function) used.

•Compression Method: a list of compression techniques supported by the client.

The server responds with a "server_hello" message, with a similar structure as the "client_hello" message. The chosen version of TLS is the lowest of 1) the version requested by the client 2) the highest version supported by the server. The "CipherSuite" contains the combination proposed by the client, which was chosen by the server (similar for the "Compression Method"). Because of the numbering used (cf. Major and Minor version fields in the TLS record header) TLS 1.0 is a higher version than SSLv3.

TLS: Handshake protocol

- **Phase 1**
  - **Defining security capabilities**



4th packet: ClientHello (client to server)

(1) Record Layer Protocol Type and Version
Here the value 22 represents the Handshake protocol contained in this Record Layer and the client uses TLS version1.0.

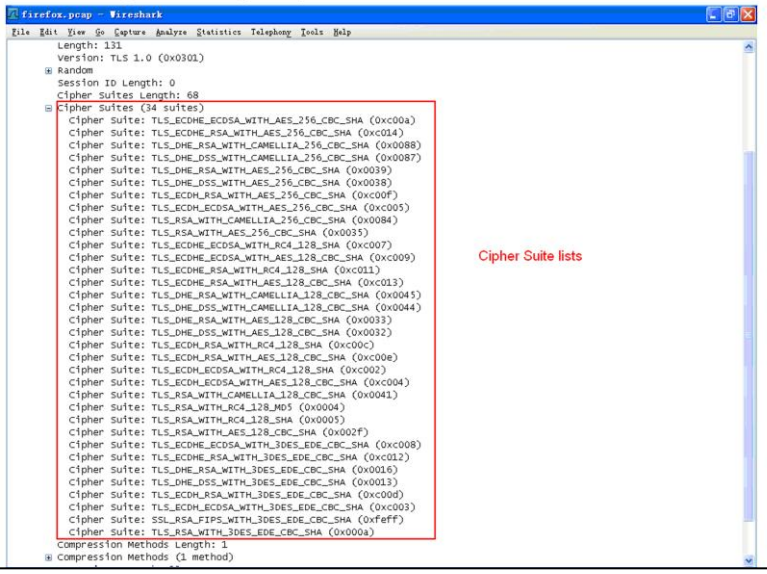
(2) Handshake Message type
Here the code 1 identifies the ClientHello message

(3) client side Random
The Random number generated by Client here is a parameter for the key calculation between client and server.

(4) Cipher suites
There are 34 suites supported. The details of this field are shown in the next slide.

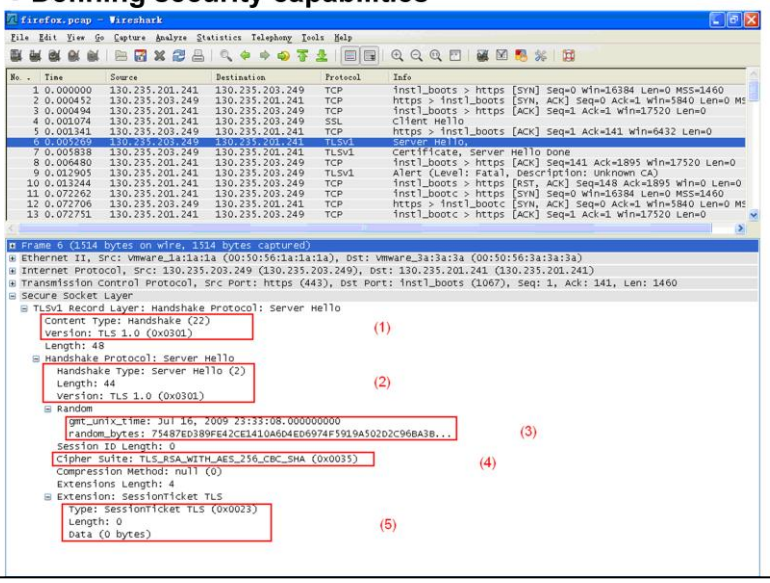
(5) SessionTicket
SessionTicket is a TLS extension included in the ClientHello message, which defines a way to resume a TLS session without requiring session-specific states at the TLS server. Here the extension is empty since the client connect to the server for the first time.

- **Phase 1**
  - **Defining security capabilities**



TLS: Handshake protocol

- **Phase 1**
  - **Defining security capabilities**



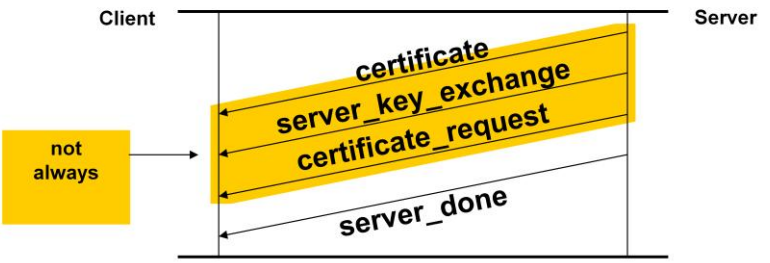
6th packet: ServerHello (server to client)

(1) Record Layer Protocol Type and Version
The server indicate its use of TLS version 1.0.
(2) Handshake Message type
The ServerHello is the respons to the client request message "ClientHello".
(3) server side Random
The random number generated by the server is used in the key calculation together with the one from the client side.
(4) selected Cipher Suite
The server checks its support Cipher Suites and select one matched to the Cipher Suites brought up by client in the ClientHello message. Here the TLS_RSA_WITH_AES_256_CBC_SHA is selected, ie: the key exchange uses RSA and the encryption and MAC algorithms are AES and SHA respectively.
(5) SessionTicket
The server here sends an empty SessionTicket extension to indicate that it will send a new session ticket later in a NewSessionTicket handshake message before the ChangeCipherSpec message finishes.

For the wireshark traces of the remaining phases, we refer to http://ipseclab.eit.lth.se/tiki-index.php?page=3.+SSL+and+TLS

TLS: Handshake protocol

- **Phase 2**
  - **Server authentication and key exchange**



p. 18

If authentication is required (which is true for most key exchange procedures) the server sends a "certificate" message containing a X.509 certificate (or a chain of such certificates).

After this message may follow a "server_key_exchange" message if necessary (e.g. not for fixed DH, or if RSA is used for the key exchange, where the server key may be used for encryption). This message will generally contain a digital signature over the parameters of the server and over the nonces that were used in phase 1 (to avoid replay attacks).

If client authentication is also required, the server may request a client certificate using a "certificate_request" message, having as parameters the certificate type (for the asymmetric algorithm and the desired key usage) and a list of acceptable certification authorities (impossible for anonymous DH).

Finally, this phase is closed with a "server_done" message, without contents.

TLS: Handshake protocol

- **Phase 3**
  - **Client authentication and key exchange**

p. 19

The client verifies the validity of the certificate he has received from the server and he verifies whether the parameters he received from the server are acceptable to him.

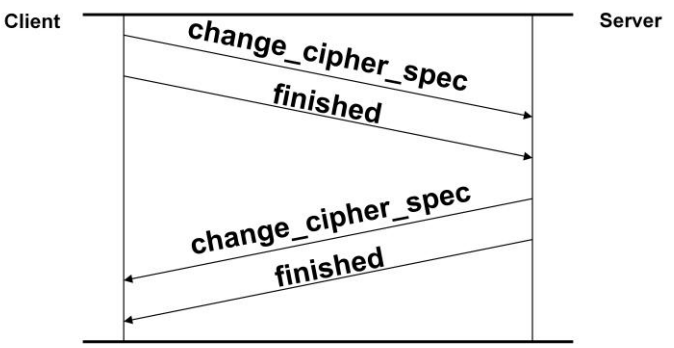
If the server has requested a certificate, the client will send a "certificate" message to the server, containing a valid certificate. If he can't do this, he sends back an empty certificate list.

The client then sends a "client_key_exchange" message, with the information needed to generate a session key (48 bytes "pre-master secret" for RSA; public parameters for ephemeral or anonymous DH; nothing for fixed DH).

Finally, the client may send a "certificate_verify" message (if a certificate had been sent before and if he has the capability to sign a message). This message is the digital signature of the hash value of information from the previous messages of the handshake, including the "Master Secret", from which the session key will later be derived). This avoids the abuse of someone else's certificate by a malicious client, who could send a "certificate" message, but not this "certificate_verify" message.

TLS: Handshake protocol

- **Phase 4**
  - **Finishing handshake**



p. 20

The client now sends a "change_cipher_spec" message (using the Change Cipher Spec protocol) to the server and copies the temporary "CipherSpec" (as has been agreed upon in this handshake) to the present value of "CipherSpec" (which will be used in this session).

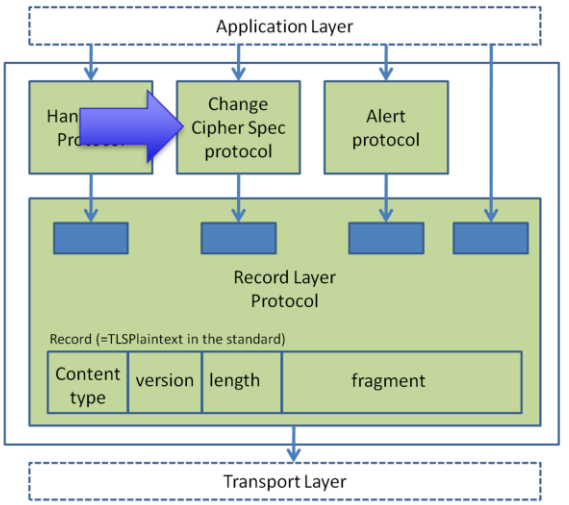
He then sends a "finished" message (using the new security parameters), signifying that the handshake is over for him. This verifies that the key exchange and the authentication have been successful. It contains hash values (using MD5 and SHA-1 in TLS1.0 and 1.1, using SHA-256 (or the hash function agreed upon in the ciphersuite) from TLS 1.2) of data with among other things the "Master Secret" and all the previous messages from the handshake.

The server then responds with his "change_cipher_spec" message (using the Change Cipher Spec protocol) and his "finished" message.

After this client and server are able to communicate with each other and to exchange data using the SSL Record Protocol.

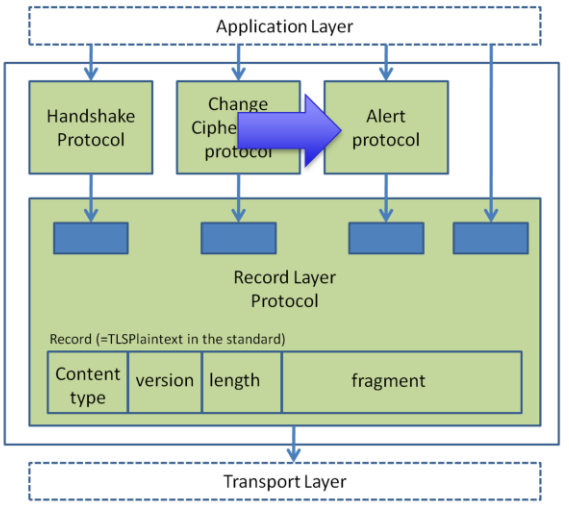
TLS: Change Cipher Spec Protocol

- **Only 1 possible message: 1 byte with value 1**
  - **Causes pending state to be copied to present state**
  - **Needed when updating encryption techniques of present connection**



TLS: Alert protocol

- **Messages for errors and warnings**
  - **1 byte for the level: "fatal" or "warning"**
    - **In case of "fatal" alert:**
      - ✓ Connection terminated
      - ✓ No new connections for this session
  - **1 byte for problem description**



p. 22

Possible warnings:

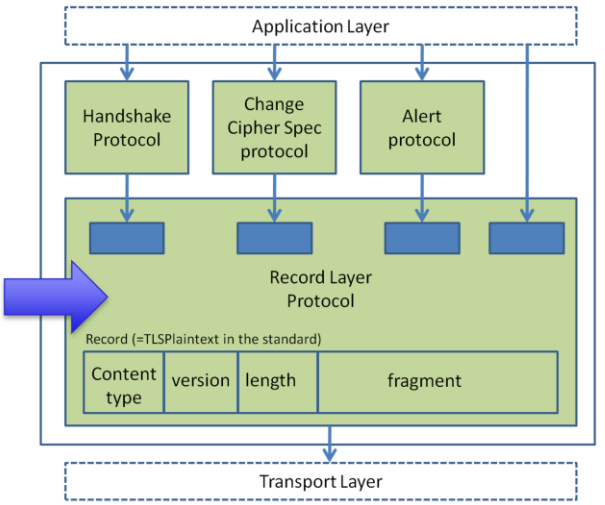
- problems with certificates
- connection termination by 1 of both communicating entities

Possible problems which can occur and are fatal:

- reception of an invalid message
- reception of an invalid MAC
- problems when decompressing
- error during handshake procedure (no acceptable security parameters were found)
- illegal parameters in a field of the handshake procedure

TLS: TLS Record Protocol

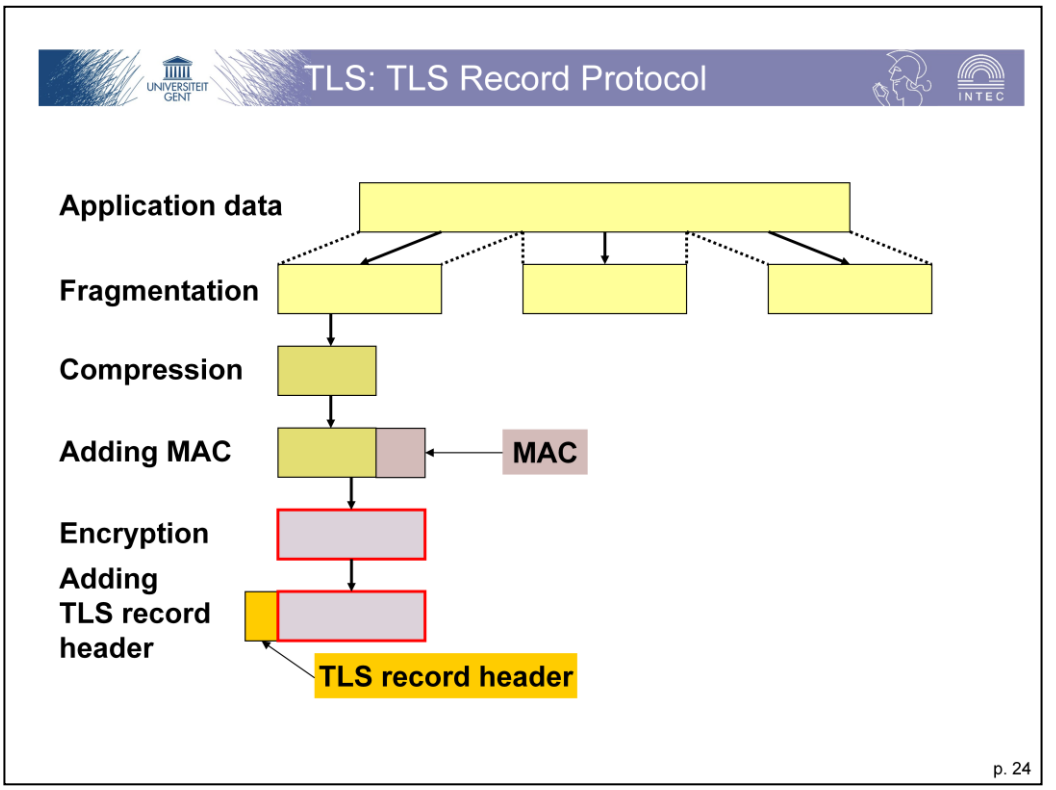
- **Base layer of TLS**
  - **Fragments data**
  - **Optionally compresses data**
  - **Provides confidentiality and authentication for TLS connections**
    - **Uses 2 secret keys**
      - ✓ 1 for confidentiality
      - ✓ 1 for authentication using MAC
    - **Both created by TLS Handshake Protocol**



p. 23

The TLS Record Protocol is a layered protocol which processes the data to be transmitted, fragments the data into manageable blocks of at most 2^14 bytes, optionally compresses the data, applies a MAC for integrity protection, encrypts, and transmits the result. Received data is handled in the reversed order: its is decrypted, verified, decompressed, reassembled, and then delivered to higher-level clients.

Note that transport-level TLS compression is not content aware and will end up attempting to recompress already compressed data (images, video, etc.). In practice, most browsers disable support for TLS compression. For these browsers, instead of relying on TLS compression, make sure your server is configured to compress all text-based assets and that you are using an optimal compression format for all other media types, such as images, video, and audio.



A typical workflow for delivering application data is as follows:

- The application data is divided into fragments of maximum 2^{14} (=16384) bytes or 16 KB.
- Application data is optionally compressed. The compression is optional (default is no compression), must be reversible and mustn't increase the data length by more than 1024 bytes.
- Message authentication code (MAC) or HMAC is added for data integrity. The MAC is negotiated during the TLS handshake (see later). The MAC is calculated over the (possibly compressed) data fragment. The MAC is calculated over the following fields.
  - MAC(K_{MAC}, Seq#||Type||Length||Data)
    - MAC: MAC function used (generally HMAC)
    - K_{MAC}: shared secret key for MAC computation
    - Seq#: sequence number for message (against replay attacks)
    - Type: protocol for fragment processing (cf. header)
    - Length: length of (uncompressed) data fragment
    - Data: the data fragment
- Data is encrypted using the symmetric encryption algorithm agreed upon (IDEA, DES, 3DES, RC4-40, RC4-128, AES-128, AES-256, etc.). If block encryption is used, it may be necessary to use the required padding after adding the MAC before encrypting the data.
- Finally, a TLS record header is added (see next slide)

Compared to SSH, both provide authenticated encryption, but in two different ways.

- SSH uses the so-called Encrypt-and-MAC, that is the ciphered message is juxtaposed to a message authentication code (MAC) of the clear message to add integrity. This is not proven to be always fully secure (even if in practical cases it should be enough).
- SSL uses MAC-then-Encrypt: a MAC is juxtaposed to the clear text, then they are both encrypted. This is not the best either, as with some block cipher modes parts of the MAC can be guessable and reveal something on the cipher. This led to vulnerabilities in TLS 1.0 (BEAST attack).

TLS: TLS Record Protocol

- **TLS Record Header**
  - **Content Type**
    - 8 bits, determines application layer protocol for data processing
  - **Major and Minor version**
    - 2 times 8 bits (for TLS 1.2: values 3 and 3)
  - **Compressed Length**
    - Length of the data

p. 25

The TLS Record protocol is also responsible for identifying different types of messages (handshake, alert, or data) via the "Content Type" field that is part of the TLS record header that is added to each outgoing packet. Only 4 categories are taken into account for the field “Content Type”: the 3 TLS specific protocols (Change Cipher Spec, Alert, and Handshake Protocols) and all other application protocols (e.g. HTTP, FTP, SMTP, etc.). The values of Major and Minor version indicate that TLS can be seen as an extension of SSL (SSLv3 had values 3 and 0, TLS 1.0 had values 3 and 1, etc.).

TLS: shared key

- Derived from "pre_master_secret" using PRF (pseudorandom function)
  - Generated by client and sent to server using RSA encryption
  - Or exchanged using DH key exchange
  - 384 bits
- Dependent on values of "nonces" from first phase of handshake
  - Reuse of keys impossible

Roger Buffle Jr. supplies his father with yet another computer password.

The importance of a good random generator

TLS: shared key

- **Master Secret**
  - **Pseudorandom function PRF**
    - **PRF(secret, label, seed) = P_hash (secret, label || seed)**
      - ✓ P_hash(secret, seed) = HMAC_hash(secret, A(1) || seed) || HMAC_hash(secret, A(2) || seed) || HMAC_hash(secret, A(3) || seed) || …
        - » A(0) = seed
        - » A(i) = HMAC_hash(secret, A(i-1))
      - ✓ hash: chosen hash function
        - » MD5 or SHA-1 (for TLS 1.0 and 1.1)
        - » SHA-256 by default (TLS 1.2)

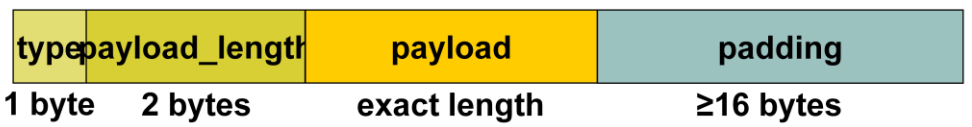
p. 27

Provided only for your information: the method for calculating the shared key. In TLS 1.2 the PRF may depend on the cipher suite. The version presented here is the default version (and in practice more or less the only PRF used). From this session dependent shared secret value the required keys and other parameters are derived: client write MAC key, server write MAC key,client write key, server write key,client write IV, server write IV

TLS: Heartbeat protocol

- **Heartbeat protocol**
  - **New protocol (2012) on top of TLS Record Protocol**
    - **Can be sent at each moment**
      - ✓ Except during handshake
  - **Main goal**
    - **Liveliness check**
      - ✓ Especially for DTLS (no session management)
      - ✓ But also for TLS
        - » Useful when no data is sent for a long period of time

- **Heartbeat protocol**
  - **Two types of messages**
    - **HeartbeatRequest**
      - ✓ Payload length must correspond to given length
        - » Otherwise: discard message
    - **HeartbeatResponse**
      - ✓ Payload must be exact copy of payload from HeartbeatRequest
        - » Otherwise: tacitly discard message



TLS: Heartbeat protocol

- **Heartbleed bug**
  - **Nickname of CVE-2014-0160**
  - **Nothing wrong with protocol**
  - **But something was wrong with its implementation in OpenSSL versions 1.0.1 up to 1.0.1f**
    - **Incorrectly dealing with wrong HeartbeatRequests**
    - **Reveals contents of remaining server memory (or causes server to crash)**
      - ✓ Could be server private keys
      - ✓ Could be cached passwords

A few links

- **Heartbeat protocol:**
  - https://tools.ietf.org/html/rfc6520
- **Heartbleed bug:**
  - http://heartbleed.com/
  - http://www.klocwork.com/blog/software-security/saving-you-from-heartbleed/ (with the faulty code)
  - https://www.schneier.com/blog/archives/2014/04/heartbleed.html
  - https://blog.wireshark.org/2014/04/heartbleed-traffic/?utm_source=rss&utm_medium=rss&utm_campaign=heartbleed-traffic
  - http://www.riverbed.com/blogs/Retroactively-detecting-a-prior-Heartbleed-exploitation-from-stored-packets-using-a-BPF-expression.html

- **Performance comparison with SSH**
  - **TLS/SSL handshake**
    - **Fewer message exchanges**
      - ✓ More efficient
    - **Entirely authenticated**
      - ✓ Only partial authentication with SSH
    - **More straightforward cipher suite selection**
  - **No encryption of packet length in TLS/SSL**
    - **Making decryption easier**

TLS computational overhead

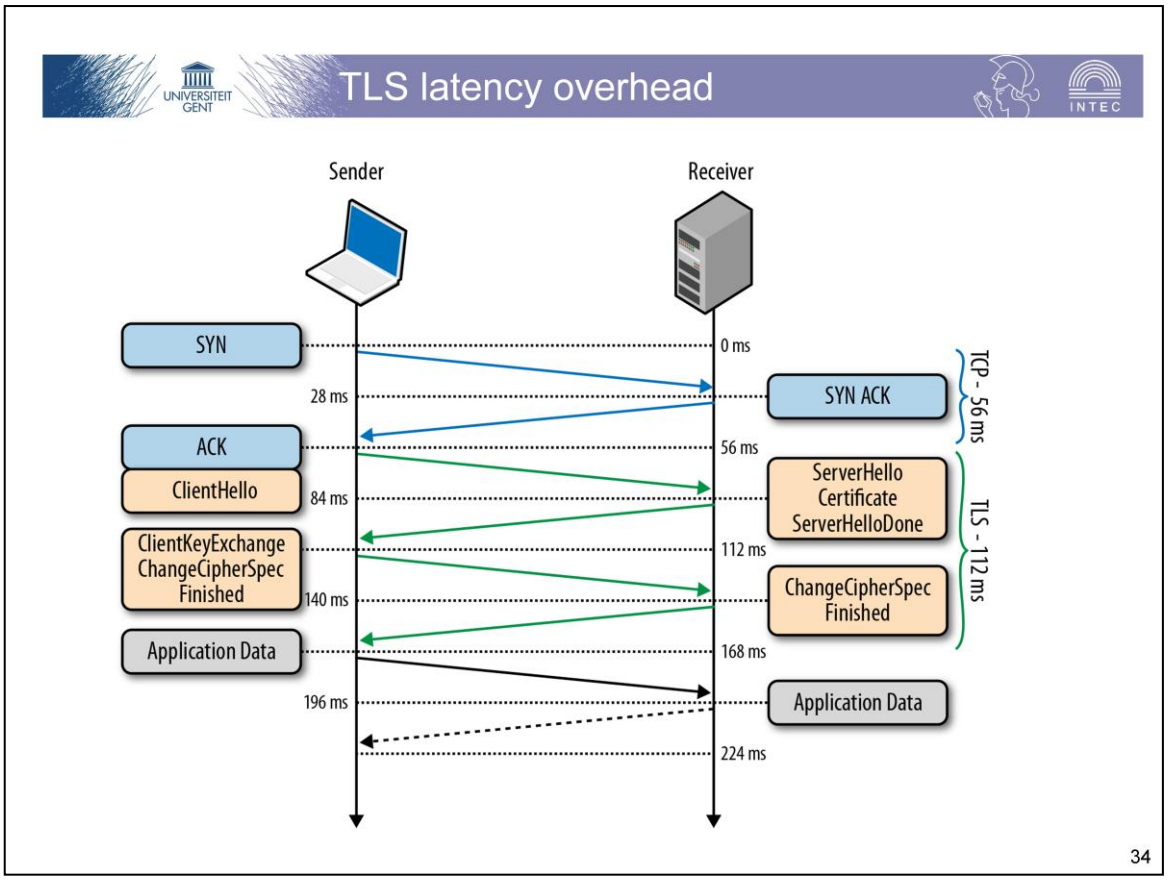
- **Overhead?**
  - **Gmail switched to use only HTTPS**
    - *On our production frontend machines, SSL/TLS accounts for less than 1% of the CPU load, less than 10 KB of memory per connection and less than 2% of network overhead.* - Adam Langley (Google), 2010
  - **Facebook uses commodity hardware**
    - *We have deployed TLS at a large scale using both hardware and software load balancers. We have found that modern software-based TLS implementations running on commodity CPUs are fast enough to handle heavy HTTPS traffic load without needing to resort to dedicated cryptographic hardware - Doug Beaver (Facebook)*
  - **Twitter uses best encryption whenever possible**
    - *Elliptic Curve Diffie-Hellman (ECDHE) is only a little more expensive than RSA for an equivalent security level… In practical deployment, we found that enabling and prioritizing ECDHE cipher suites actually caused negligible increase in CPU usage. HTTP keepalives and session resumption mean that most requests do not require a full handshake, so handshake operations do not dominate our CPU usage. We find 75% of Twitter's client requests are sent over connections established using ECDHE. The remaining 25% consists mostly of older clients that don't yet support the ECDHE cipher suites. - Jacob Hoffman-Andrews (Twitter)*

33

Due to the layered architecture of the network protocols, running an application over TLS is no different from communicating directly over TCP. As such, there are no, or at most minimal, application modifications that you will need to make to deliver it over TLS. However, establishing and maintaining an encrypted channel introduces additional computational costs for both peers. Specifically, first there is the asymmetric (public key) encryption used during the TLS handshake. Then, once a shared secret is established, it is used as a symmetric key to encrypt all TLS records.

As we noted earlier, public key cryptography is more computationally expensive when compared with symmetric key cryptography, and in the early days of the Web often required additional hardware to perform "SSL offloading." The good news is this is no longer the case. Modern hardware has made great improvements to help minimize these costs, and what once required additional hardware can now be done directly on the CPU. Large organizations such as Facebook, Twitter, and Google, which offer TLS to hundreds of millions of users, perform all the necessary TLS negotiation and computation in software and on commodity hardware.

Nevertheless, techniques such as "TLS Session Resumption" (see next slide) are still important optimizations, which will help you decrease the computational costs and latency of public key cryptography performed during the TLS handshake. There is no reason to spend CPU cycles on work that you don't need to do.



Compared to traditional TCP traffic (e.g. HTTP requests), the use of TLS introduces overhead. Before the client and the server can begin exchanging application data over TLS, the encrypted tunnel must be negotiated: the client and the server must agree on the version of the TLS protocol, choose the ciphersuite, and verify certificates if necessary. Unfortunately, each of these steps requires new packet roundtrips between the client and the server, which adds startup latency to all TLS connections.

0 ms – 56 ms: TLS runs over a reliable transport (TCP), which means that we must first complete the TCP three-way handshake, which takes one full roundtrip.

56 ms – 84 ms: With the TCP connection in place, the client sends a number of specifications in plain text, such as the version of the TLS protocol it is running, the list of supported ciphersuites, and other TLS options it may want to use.

84 ms – 112 ms: The server picks the TLS protocol version for further communication, decides on a ciphersuite from the list provided by the client, attaches its certificate, and sends the response back to the client. Optionally, the server can also send a request for the client's certificate and parameters for other TLS extensions.

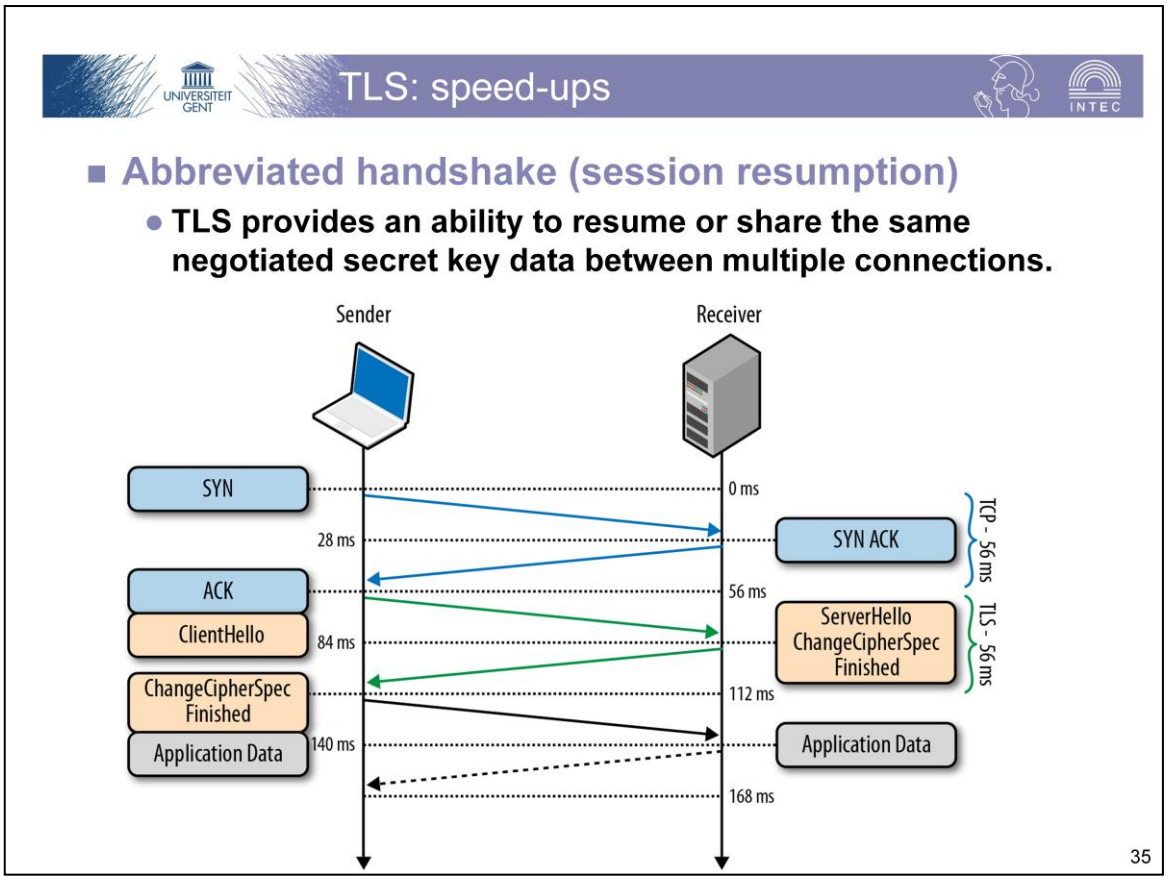
112 ms – 140 ms: Assuming both sides are able to negotiate a common version and cipher, and the client is happy with the certificate provided by the server, the client initiates either the RSA or the Diffie-Hellman key exchange, which is used to establish the symmetric key for the ensuing session.

140 ms – 168 ms: The server processes the key exchange parameters sent by the client, checks message integrity by verifying the MAC, and returns an encrypted "Finished" message back to the client.

168 ms – 224 ms: The client decrypts the message with the negotiated symmetric key, verifies the MAC, and if all is well, then the tunnel is established and application data can now be sent.

(These results assumes a 28 millisecond one-way "light in fiber" delay between New York and London)

It is important to realize that every TLS connection will require up to two extra roundtrips on top of the TCP handshake—that's a long time to wait before any application data can be exchanged! If not managed carefully, delivering application data over TLS can add hundreds, if not thousands of milliseconds of network latency.



New TLS connections require two roundtrips for a "full handshake" and CPU resources to verify and compute the parameters for the ensuing session. However, the good news is that we don't have to repeat the "full handshake" in every case.

Abbreviated handshake (using a session ID)

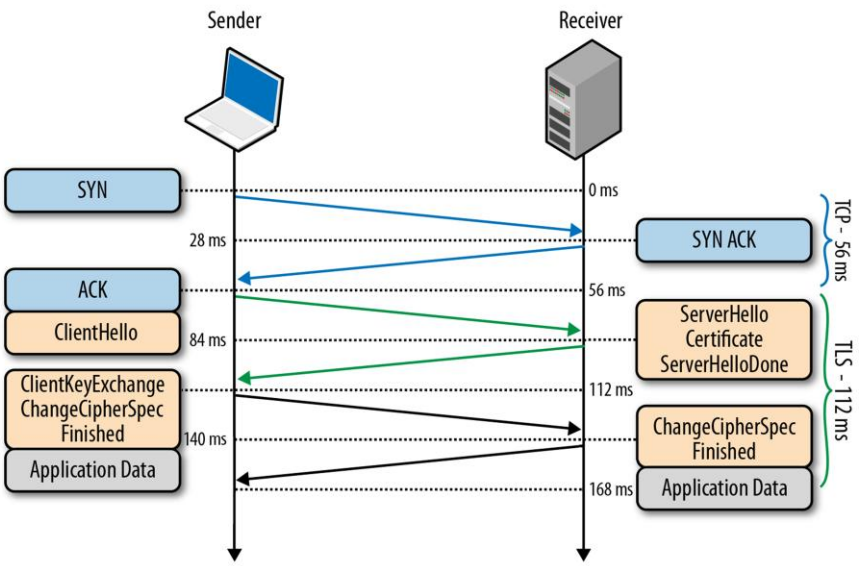
If the client has previously communicated with the server, an "abbreviated handshake" can be used, which requires one roundtrip and allows the client and server to reduce the CPU overhead by reusing the previously negotiated parameters for the secure session. To this end, session identifiers are used. The first Session Identifiers (RFC 5246) resumption mechanism was introduced in SSL 2.0, which allowed the server to create and send a 32-byte session identifier as part of its "ServerHello" message during the full TLS negotiation from the previous slide. Internally, the server could then maintain a cache of session IDs and the negotiated session parameters for each peer. In turn, the client could then also store the session ID information and include the ID in the "ClientHello" message for a subsequent session, which serves as an indication to the server that the client still remembers the negotiated cipher suite and keys from previous handshake and is able to reuse them. Assuming both the client and the server are able to find the shared session ID parameters in their respective caches, then an abbreviated handshake can take place. Otherwise, a full new session negotiation is required, which will generate a new session ID. Leveraging session identifiers allows us to remove a full roundtrip, as well as the overhead of public key cryptography, which is used to negotiate the shared secret key. This allows a secure connection to be established quickly and with no loss of security, since we are reusing the previously negotiated session data. In practice, most web applications attempt to establish multiple connections to the same host to fetch resources in parallel, which makes session resumption a must-have optimization to reduce latency and computational costs for both sides. Most modern browsers intentionally wait for the first TLS connection to complete before opening new connections to the same server: subsequent TLS connections can reuse the SSL session parameters to avoid the costly handshake. However, one of the practical limitations of the Session Identifiers mechanism is the requirement for the server to create and maintain a session cache for every client. This results in several problems on the server, which may see tens of thousands or even millions of unique connections every day: consumed memory for every open TLS connection, a requirement for session ID cache and eviction policies, and nontrivial deployment challenges for popular sites with many servers, which should, ideally, use a shared TLS session cache for best performance. None of the preceding problems are impossible to solve, and many high-traffic sites are using session identifiers successfully today. But for any multiserver deployment, session identifiers will require some careful thinking and systems architecture to ensure a well operating session cache.

To address this concern for server-side deployment of TLS session caches, the "Session Ticket" (RFC 5077) replacement mechanism was introduced, which removes the requirement for the server to keep per-client session state. Instead, if the client indicated that it supports Session Tickets, in the last exchange of the full TLS handshake, the server can include a New Session Ticket record, which includes all of the session data encrypted with a secret key known only by the server. This session ticket is then stored by the client and can be included in the SessionTicket extension within the ClientHello message of a subsequent session. Thus, all session data is stored only on the client, but the ticket is still safe because it is encrypted with a key known only by the server.

The session identifiers and session ticket mechanisms are respectively commonly referred to as *session caching* and *stateless resumption* mechanisms. The main improvement of stateless resumption is the removal of the server-side session cache, which simplifies deployment by requiring that the client provide the session ticket on every new connection to the server—that is, until the ticket has expired. In practice, deploying session tickets across a set of load-balanced servers also requires some careful thinking and systems architecture: all servers must be initialized with the same session key, and an additional mechanism may be needed to periodically rotate the shared key across all servers.

TLS: speed-ups

- **False start**
  - **Don't wait for handshake**



36

Session resumption does not help in cases where the visitor is communicating with the server for the first time, or if the previous session has expired, to this end the false start optimization is defined. False Start is an optional TLS protocol extension that allows the client and server to start transmitting encrypted application data when the handshake is only partially complete—i.e. once ChangeCipherSpec and Finished messages are sent, but without waiting for the other side to do the same. This optimization reduces new handshake overhead to one roundtrip. False Start does not modify the TLS handshake protocol, rather it only affects the protocol timing of when the application data can be sent. Intuitively, once the client has sent the ClientKeyExchange record, it already knows the encryption key and can begin transmitting application data—the rest of the handshake is spent confirming that nobody has tampered with the handshake records, and can be done in parallel. As a result, False Start allows us to keep the TLS handshake at one roundtrip regardless of whether we are performing a full or abbreviated handshake.

Because False Start is only modifying the timing of the handshake protocol, it does not require any updates to the TLS protocol itself and can be implemented unilaterally—i.e. the client can simply begin transmitting encrypted application data sooner. Well, that's the theory. In practice, even though TLS False Start should be backwards compatible with all existing TLS clients and servers, enabling it by default for all TLS connections proved to be problematic due to some poorly implemented servers. As a result, all modern browsers are capable of using TLS False Start, but will only do so when certain conditions are met by the server:

• Chrome and Firefox require an NPN/ALPN protocol advertisement to be present in the server handshake, and that the cipher suite chosen by the server enables forward secrecy.

• Safari requires that the cipher suite chosen by the server enables forward secrecy.

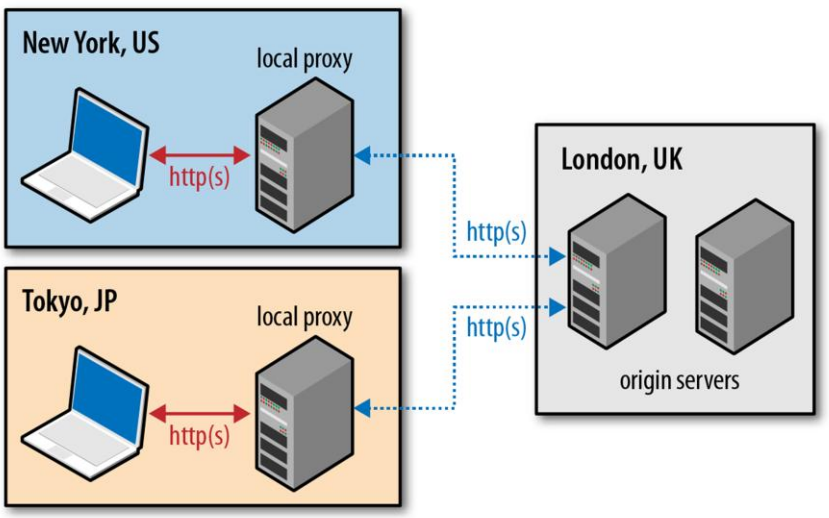
• Internet Explorer uses a combination of a blacklist of known sites that break when TLS False Start is enabled, and a timeout to repeat the handshake if the TLS False Start handshake failed.

To enable TLS False Start across all browsers the server should advertise a list of supported protocols via the NPN/ALPN extension—e.g. "http/1, http/1.1"—and be configured to support and prefer cipher suites that enable forward secrecy.

For best results, both optimizations should be used together to provide a single roundtrip handshake for new and returning visitors, plus computational savings for sessions that can be resumed based on previously negotiated session parameters.

TLS: speed-ups

- **Early termination (proxies)**
  - **Place servers closer to the client**
  - **Minimize latency costs**



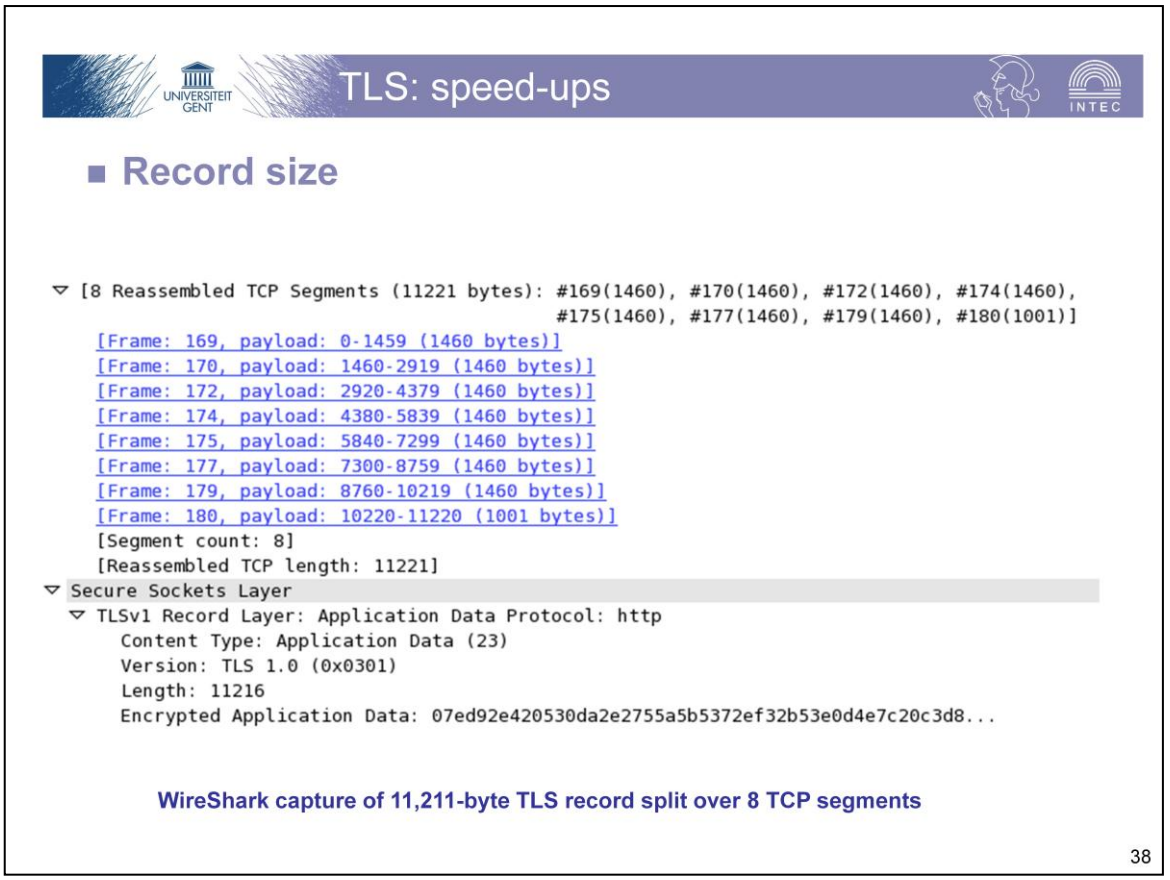
37

The extensibility and the success of HTTP created a vibrant ecosystem of various proxies and intermediaries on the Web: cache servers, security gateways, web accelerators, content filters, and many others. In some cases we are aware of their presence (explicit proxies), and in others they are completely transparent to the end user. Unfortunately, the very success and the presence of these servers has created a small problem for anyone who tries to deviate from the HTTP protocol in any way: some proxy servers may simply relay HTTP extensions or alternative wire formats they cannot interpret, others may continue to blindly apply their logic even when they shouldn't, and some, such as security appliances, may infer malicious traffic where there is none.

In other words, in practice, deviating from the well-defined semantics of HTTP on port 80 will often lead to unreliable deployments: some clients will have no problems, while others may fail with unpredictable behaviors—e.g., the same client may see different connectivity behaviors as it migrates between different networks. Due to these behaviors, new protocols and extensions to HTTP, such as WebSocket, SPDY, and others, often rely on establishing an HTTPS tunnel to bypass the intermediate proxies and provide a reliable deployment model: the encrypted tunnel obfuscates the data from all intermediaries. This solves the immediate problem, but it does have a real downside of not being able to leverage the intermediaries, many of which provide useful services: authentication, caching, security scanning, and so on.

It is worth noting that the use of proxies also has disadvantages. With HTTPS, once the packet is received by a device in the destination network (Web Server, Border Router, load balancer, etc...)it is un-encrypted and spends the rest of its journey in plain text. Many would argue that this is not a big deal since the traffic is internal at this time, but if the payload contains sensitive data, it is being stored un-encrypted in the log files of every network device it passes through until it gets to its final destination (in contrast, with SSH, typically, the destination device is specified and the transmission is encrypted until it reaches this device). There are ways of re-encrypting the HTTPS data but these are extra steps that most forget to take when implementing an HTTPS solution in their environment.

Early termination is a simple technique of placing your servers closer to the user to minimize the latency cost of each roundtrip between the client and the server. A nearby server can also terminate the TLS session, which means that the TCP and TLS handshake roundtrips are much quicker and the total connection setup latency is greatly reduced. In turn, the same nearby server can then establish a pool of long-lived, secure connections to the origin servers and proxy all incoming requests and responses to and from the origin servers. In a nutshell, moving the server closer to the client accelerates TCP and TLS handshakes.



All application data delivered via TLS is transported within a record protocol . The maximum size of each record is 16 KB, and depending on the chosen cipher, each record will add anywhere from 20 to 40 bytes of overhead for the header, MAC, and optional padding. If the record then fits into a single TCP packet, then we also have to add the IP and TCP overhead: 20-byte header for IP, and 20-byte header for TCP with no options. As a result, there is potential for 60 to 100 bytes of overhead for each record. For a typical maximum transmission unit (MTU) size of 1,500 bytes on the wire, this packet structure translates to a minimum of 6% of framing overhead. The smaller the record, the higher the framing overhead. However, simply increasing the size of the record to its maximum size (16 KB) is not necessarily a good idea! If the record spans multiple TCP packets, then the TLS layer must wait for all the TCP packets to arrive before it can decrypt the data. If any of those TCP packets get lost, reordered, or throttled due to congestion control, then the individual fragments of the TLS record will have to be buffered before they can be decoded, resulting in additional latency. In practice, these delays can create significant bottlenecks for the browser, which prefers to consume data byte by byte and as soon as possible.

Small records incur overhead, large records incur latency, and there is no one value for the "optimal" record size. Instead, for web applications, which are consumed by the browser, the best strategy is to dynamically adjust the record size based on the state of the TCP connection:

• When the connection is new and TCP congestion window is low, or when the connection has been idle for some time (e.g slow-start restart), each TCP packet should carry exactly one TLS record, and the TLS record should occupy the full maximum segment size (MSS) allocated by TCP.

• When the connection congestion window is large and if we are transferring a large stream (e.g. streaming video), the size of the TLS record can be increased to span multiple TCP packets (up to 16KB) to reduce framing and CPU overhead on the client and server.

Using a dynamic strategy delivers the best performance for interactive traffic: small record size eliminates unnecessary buffering latency and improves the *time-to-first-{HTML byte, …, video frame}*, and a larger record size optimizes throughput by minimizing the overhead of TLS for long-lived streams.

To determine the optimal record size for each state let's start with the initial case of a new or idle TCP connection where we want to avoid TLS records from spanning multiple TCP packets:

• Allocate 20 bytes for IPv4 framing overhead and 40 bytes for IPv6.

• Allocate 20 bytes for TCP framing overhead.

• Allocate 40 bytes for TCP options overhead (timestamps, SACKs).

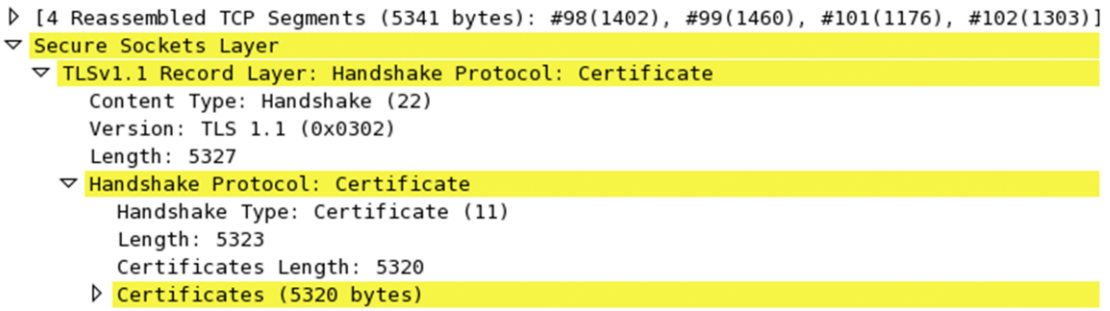
Assuming a common 1,500-byte starting MTU, this leaves 1,420 bytes for a TLS record delivered over IPv4, and 1,400 bytes for IPv6. To be future-proof, use the IPv6 size, which leaves us with 1,400 bytes for each TLS record payload—adjust as needed if your MTU is lower.

Next, the decision as to when the record size should be increased and reset if the connection has been idle, can be set based on pre-configured thresholds: increase record size to up to 16 KB after X KB of data have been transferred, and reset the record size after Y milliseconds of idle time. Typically, configuring the TLS record size is not something we can control at the application layer. Instead, this is a setting and perhaps even a compile-time constant or flag on the TLS server.

As of early 2014, Google's servers use small TLS records that fit into a single TCP segment for the first ~1 MB of data, increase record size to 16 KB after that to optimize throughput, and then reset record size back to a single segment after ~1 second of inactivity—lather, rinse, repeat. Similarly, if your servers are handling a large number of TLS connections, then minimizing memory usage per connection can be a vital optimization. By default, popular libraries such as OpenSSL will allocate up to 50 KB of memory per connection, but as with the record size, it may be worth checking the documentation or the source code for how to adjust this value. Google's servers reduce their OpenSSL buffers down to about 5 KB.

TLS: speed-ups

- **Certificate chain**



WireShark capture of a 5,323-byte TLS certificate chain

39

Another speed-up optimization that is targeted mainly towards HTTPS traffic is the inclusion of the certificate chains. Verifying the chain of trust requires that the browser traverse the chain, starting from the site certificate, and recursively verifying the certificate of the parent until it reaches a trusted root. To find the parent certificates, the child certificate will usually contain the URL for the parent.

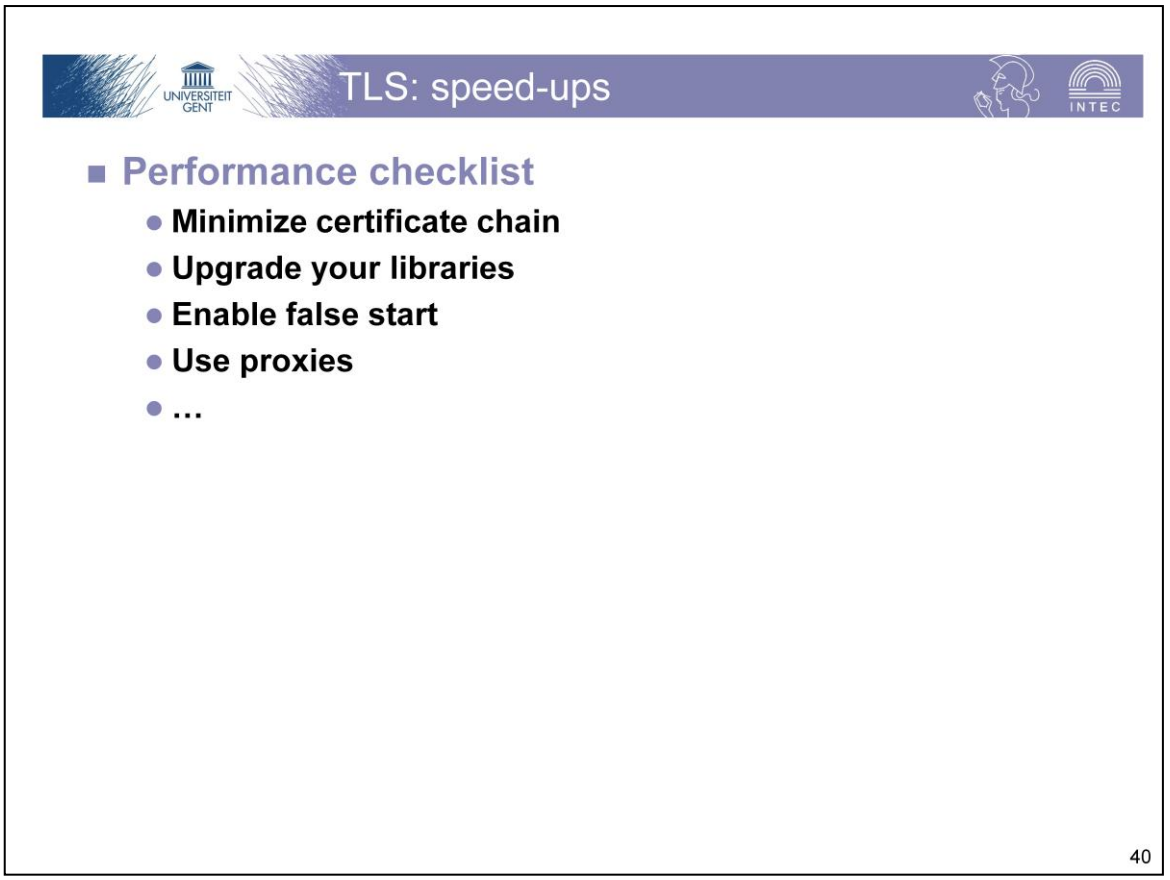
Hence, the first optimization you should make is to verify that the server does not forget to include all the intermediate certificates when the handshake is performed. If you forget, many browsers will still work, but they will instead be forced to pause the verification and fetch the intermediate certificate on their own, verify it, and then continue. This will most likely require a new DNS lookup, TCP connection, and an HTTP GET request, adding hundreds of milliseconds to your handshake. Conversely, it is important to make sure you do not include unnecessary certificates in your chain. Or, more generally, you should aim to minimize the size of your certificate chain. Recall that server certificates are sent during the TLS handshake, which is likely running over a new TCP connection that is in the early stages of its slow-start algorithm. If the certificate chain exceeds TCP's initial congestion window, then we will inadvertently add yet another roundtrip to the handshake: certificate length will overflow the congestion window and cause the server to stop and wait for a client ACK before proceeding.

The certificate chain shown in the slide above is over 5 KB in size, which will overflow the initial congestion window size of older servers and force another roundtrip of delay into the handshake. One possible solution is to increase the initial congestion window. In addition, you should investigate if it is possible to reduce the size of the sent certificates:

• Minimize the number of intermediate CAs. Ideally, your sent certificate chain should contain exactly two certificates: your site and the CA's intermediary certificate; use this as a criteria in the selection of your CA. The third certificate, which is the CA root, should already be in the browser's trusted root and hence should not be sent.

• It is not uncommon for many sites to include the root certificate of their CA in the chain, which is entirely unnecessary: if your browser does not already have the certificate in its trust store, then it won't be trusted, and including the root certificate won't change that.

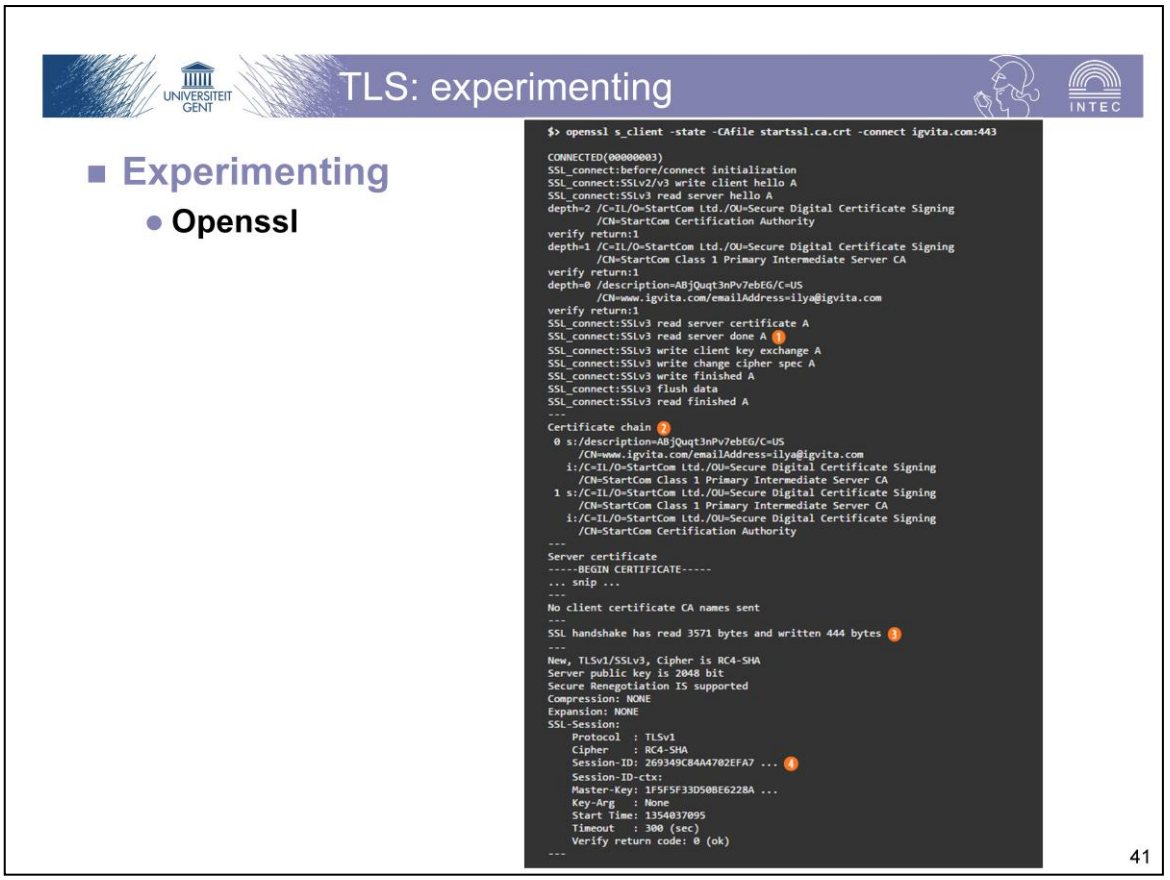
• A carefully managed certificate chain can be as low as 2 or 3 KB in size, while providing all the necessary information to the browser to avoid unnecessary roundtrips or out-of-band requests for the certificates themselves. Optimizing your TLS handshake mitigates a critical performance bottleneck, since every new TLS connection is subject to its overhead.



As an application developer, you are shielded from virtually all the complexity of TLS. Short of ensuring that you do not mix HTTP and HTTPS content on your pages, your application will run transparently on both. However, the performance of your entire application will be affected by the underlying configuration of your server. The good news is it is never too late to make these optimizations, and once in place, they will pay high dividends for every new connection to your servers.

A short list of optimizations to keep in mind (from http://chimera.labs.oreilly.com/books/1230000000545/ch04.html)

- Upgrade TLS libraries to latest release, and (re)build servers against them.
- Enable and configure session caching and stateless resumption.
- Monitor your session caching hit rates and adjust configuration accordingly.
- Configure forward secrecy ciphers to enable TLS False Start.
- Terminate TLS sessions closer to the user to minimize roundtrip latencies.
- Use dynamic TLS record sizing to optimize latency and throughput.
- Ensure that your certificate chain does not overflow the initial congestion window.
- Remove unnecessary certificates from your chain; minimize the depth.
- Configure OCSP stapling on your server.
- Disable TLS compression on your server.
- Configure SNI support on your server.
- Append HTTP Strict Transport Security header.
- Get best performance from TCP; see "Optimizing for TCP".



To verify and test your configuration, you should familiarize yourself with the openssl command-line interface, which will help you inspect the entire handshake and configuration of your server locally.

You can observe the following 4 steps when using the openssl command.

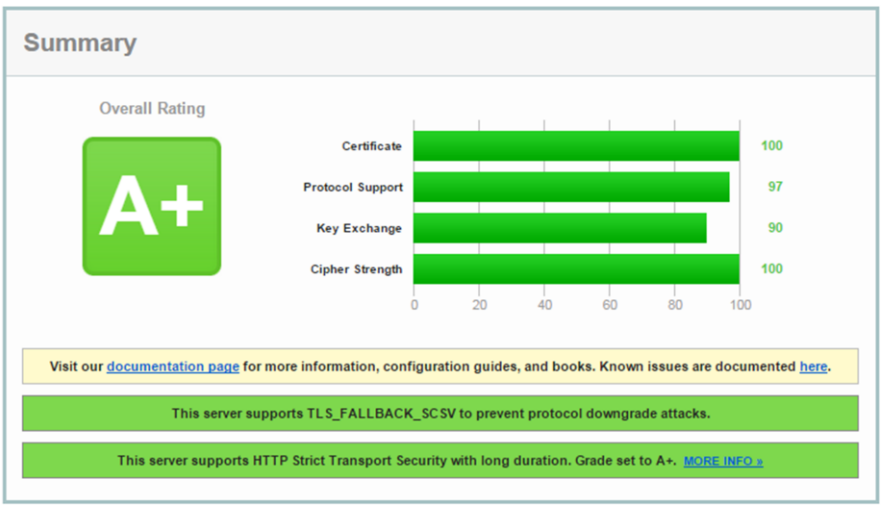
- Client completed verification of received certificate chain.
- Received certificate chain (two certificates).
- Size of received certificate chain.
- Issued session identifier for stateful TLS resume.

In the preceding example, we connect to igvita.com on the default TLS port (443), and perform the TLS handshake. Because the s_client makes no assumptions about known root certificates, we manually specify the path to the root certificate of StartSSL Certificate Authority—this is important. Your browser already has StartSSL's root certificate and is thus able to verify the chain, but s_client makes no such assumptions. Try omitting the root certificate, and you will see a verification error in the log.

Inspecting the certificate chain shows that the server sent two certificates, which added up to 3,571 bytes, which is very close to the three- to four-segment initial TCP congestion window size. We should be careful not to overflow it or raise the cwnd size on the server. Finally, we can inspect the negotiated SSL session variables—chosen protocol, cipher, key—and we can also see that the server issued a session identifier for the current session, which may be resumed in the future.

TLS: experimenting

- **Experimenting**
  - **Free online scans**
    - **Example:** https://www.ssllabs.com/ssltest/



Additionally, you can use an online service, such as the https://www.ssllabs.com/ssltest/ to scan your public server for common configuration and security flaws. It is well worth going to this site to evaluate the performance of a number of TLS connections.

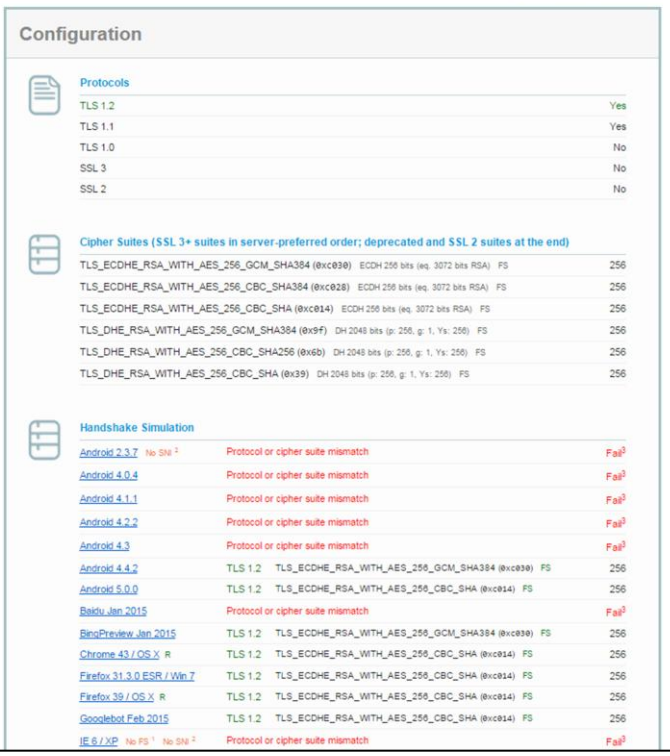
TLS: experimenting

- **Experimenting**
  - **Free online scans**
  - **Example:** https://www.ssllabs.com/ssltest/

43

TLS: experimenting

- **Experimenting**
  - **Free online scans**
  - **Example:** https://www.ssllabs.com/ssltest/



Configuration

Protocols

TLS 1.2	Yes
TLS 1.1	Yes
TLS 1.0	No
SSL 3	No
SSL 2	No

Cipher Suites (SSL 3+ suites in server-preferred order; deprecated and SSL 2 suites at the end)

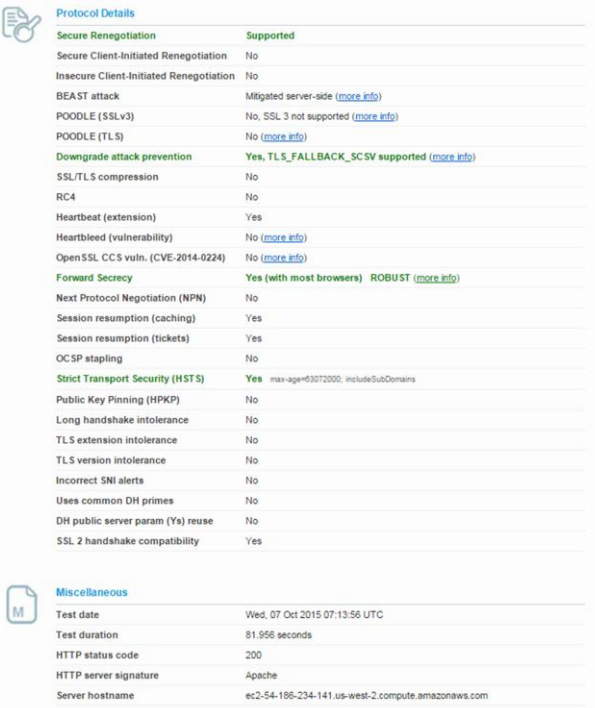
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030) ECDH 256 bits (eq. 3072 bits RSA) FS	256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (0xc028) ECDH 256 bits (eq. 3072 bits RSA) FS	256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014) ECDH 256 bits (eq. 3072 bits RSA) FS	256
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384 (0x9f) DH 2048 bits (p: 256, g: 1, Ys: 256) FS	256
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256 (0x6b) DH 2048 bits (p: 256, g: 1, Ys: 256) FS	256
TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x39) DH 2048 bits (p: 256, g: 1, Ys: 256) FS	256

Handshake Simulation

Android 2.3.7 No SNI [2]	Protocol or cipher suite mismatch		Fail[3]
Android 4.0.4	Protocol or cipher suite mismatch		Fail[3]
Android 4.1.1	Protocol or cipher suite mismatch		Fail[3]
Android 4.2.2	Protocol or cipher suite mismatch		Fail[3]
Android 4.3	Protocol or cipher suite mismatch		Fail[3]
Android 4.4.2	TLS 1.2	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030) FS	256
Android 5.0.0	TLS 1.2	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014) FS	256
Baidu Jan 2015	Protocol or cipher suite mismatch		Fail[3]
BingPreview Jan 2015	TLS 1.2	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030) FS	256
Chrome 43 / OS X R	TLS 1.2	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014) FS	256
Firefox 31.3.0 ESR / Win 7	TLS 1.2	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014) FS	256
Firefox 39 / OS X R	TLS 1.2	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014) FS	256
Googlebot Feb 2015	TLS 1.2	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014) FS	256
IE 6 / XP No FS [1] No SNI [2]	Protocol or cipher suite mismatch		Fail[3]

TLS: experimenting

- **Experimenting**
  - **Free online scans**
  - **Example:** https://www.ssllabs.com/ssltest/



Protocol Details

Secure Renegotiation	**Supported**
Secure Client-Initiated Renegotiation	No
Insecure Client-Initiated Renegotiation	No
BEAST attack	Mitigated server-side (more info)
POODLE (SSLv3)	No, SSL 3 not supported (more info)
POODLE (TLS)	No (more info)
Downgrade attack prevention	**Yes, TLS_FALLBACK_SCSV supported** (more info)
SSL/TLS compression	No
RC4	No
Heartbeat (extension)	Yes
Heartbleed (vulnerability)	No (more info)
OpenSSL CCS vuln. (CVE-2014-0224)	No (more info)
Forward Secrecy	**Yes (with most browsers) ROBUST** (more info)
Next Protocol Negotiation (NPN)	No
Session resumption (caching)	Yes
Session resumption (tickets)	Yes
OCSP stapling	No
Strict Transport Security (HSTS)	**Yes** max-age=63072000; includeSubDomains
Public Key Pinning (HPKP)	No
Long handshake intolerance	No
TLS extension intolerance	No
TLS version intolerance	No
Incorrect SNI alerts	No
Uses common DH primes	No
DH public server param (Ys) reuse	No
SSL 2 handshake compatibility	Yes

Miscellaneous

Test date	Wed, 07 Oct 2015 07:13:56 UTC
Test duration	81.956 seconds
HTTP status code	200
HTTP server signature	Apache
Server hostname	ec2-54-186-234-141.us-west-2.compute.amazonaws.com

Overview

- Network model
- Secure configuration of devices
- Exchanging keys
- Secure networking protocols
  - Transport layer: TLS & SSL
  - **Network layer: IPSec & VPN**
  - Data link layer: WEB & WPA
- Firewalls

References

- This work contains content adapted from, amongst others, the following sources (in no particular order)
  - **Books**
    - William Stallings, "Cryptography and Network Security, principles and practices", 6th (international) edition, Prentice Hall, 2010;
    - Matt Bishop, "Computer Security: Art and Science", Addison Wesley, Pearson Education, 2003, ISBN-13: 978-0-201-44099-7
  - **Lecture slides:**
    - "Informatiebeveiliging", Universiteit Gent, Eric Laermans & Thom Dhaene
    - Stallings, 2014, Lecture slides by Lawrie Brown
  - **Wikipedia**
    - Note page descriptions
  - **Websites**
    - http://chimera.labs.oreilly.com/books/1230000000545/ch04.html
    - http://ipseclab.eit.lth.se/tiki-index.php?page=3.+SSL+and+TLS
    - http://www.ciscopress.com/articles/index.asp?st=42081

47