Application Layer – TLS / SSL

Information Security 2019

Johannes Feichtner
johannes.feichtner@iaik.tugraz.at
Outline

- Crypto Crash Course
- TLS Handshake
- Properties
  - Cipher Suites
  - Perfect Forward Secrecy
- Security
  - HSTS
  - Certificate Pinning (HPKP)

TCP / IP Model

- Application
- Transport
- Network
- Link layer
  
  (Ethernet, WLAN, LTE...)
Asymmetric Cryptography: Encryption / Decryption

**Keypair**

- **Private Key** = Really private, only the owner should have it
- **Public Key** = Everyone can have it

- Typically only small data is encrypted with asymmetric keys (performance!)
- Asymmetric schemes often encrypt ( „wrap“ ) symmetric keys

**Diagram Notes**

- **Encrypt (everybody)**
- **Decrypt (only private key owner)**
- **Encrypt (everybody)**
- **Plain data**
- **Encrypted data**
- **Plain data**

**Crypto Crash Course**

IAIK

Graz
Asymmetric Cryptography: Signing / Verification

Signature
- Not applied on complete data block
- Instead, hash calculated over data → signed / verified

Verification: Comparison if hashes match
**Basic idea**

- Server determines DH parameters + generates key pair
- Sends parameters + public key to client
- Client uses DH parameters (of server) + generates key pair
- Client sends public key to server
- Both calculate same secret

**Diffie-Hellman (DH)**
Asymmetric Cryptography: Key Agreement

**User 1**
- Private Key
- Public Key

**User 2**
- Private Key
- Public Key

Exchange public keys

Agree on symmetric key with user 1 private key and user 2 public key

User 1

- Private Key
- Public Key

ECDH / DH

Symmetric key

User 2

- Private Key
- Public Key

ECDH / DH

Symmetric key

Agree on symmetric key with user 2 private key and user 1 public key
X.509 Certificates

Creating an SSL Certificate

1. Certificate Info
2. SHA
3. Fingerprint: d44554eccf0
4. Certificate Authority’s Private Key
5. Signature
6. Signed Certificate

Verifying an SSL Certificate

1. Certificate Info
2. SHA
3. Fingerprint: d44554eccf0
4. Signature
5. Certificate Authority’s Public Key
6. Verified Certificate

Source: https://goo.gl/egFCjg
X.509 Certificates

Validation

Web Browser gets host cert during TLS handshake

1. Verify hostname matches certificate subject
2. Verify signature
Transport Layer Security
TLS Introduction

Basics
- Key protocol for secure communication
  - HTTPS, VPNs, for any secure communication based on certificates
- Designed to operate on TCP (for reliability reasons)
  - Later adapted to support datagram protocols also, e.g. UDP
  → Datagram Transport Layer Security (DTLS), RFC 6347
- Initial development by Netscape in the 90s
  - Named „Secure Sockets Layer“ (SSL)
  - Later standardized by IETF → renamed to TLS
TLS Versions

- **1995**: First public release of proprietary SSL 2.0
  - Critical security flaws briefly afterwards
  - Usage prohibited in 2011 (RFC 6176)

- **1996**: SSL 3.0, RFC 6101, deprecated in June 2015 (RFC 7568)

- **1999**: TLS 1.0, RFC 2246
  - No „dramatic changes“ but no more interoperability between SSL 3.0 & TLS 1.0
  - Includes downgrade option to SSL 3.0 → weakens security!

- **2006**: TLS 1.1, RFC 4346

- **2008**: TLS 1.2, RFC 5246: Removed old ciphers, bugfixes

- **2018**: TLS 1.3, RFC 8446 (Proposed Standard): Drop weak ciphers
All applications running TLS are provided with three essential services:

**Authentication**
Verify identity of client and server

**Data Integrity**
Detect message tampering and forgery, e.g. malicious Man-in-the-middle

**Encryption**
Ensure privacy of exchanged communication

Note: Technically, not all services are required to be used → Can raise risk for security issues!
TLS 1.2 Handshake

= Establish parameters for cryptographically secure data channel

Optional: Only with Client TLS!

ClientHello → ServerHello
Certificate → ServerKeyExchange
CertificateRequest → Certificate
ServerHelloDone
ClientKeyExchange → ClientKeyExchange
CertificateVerify → CertificateVerify
ChangeCipherSpec → ChangeCipherSpec
Finished → Finished

Application Data → Application Data

Full handshake scenario!
**Client:** ClientHello

With TCP connection setup on **port 443**, clients initiate the TLS negotiation.

**Message contains**

- Highest supported TLS version
- Random number (for key exchange)
- Session ID
  - If existing session should be resumed
  - Kind of „keep-alive“ across requests
- Suggested cipher suites
- Supported compression methods
- Extensions
**Server: ServerHello**

Response to ClientHello if server finds common set of algorithms

**Message contains**

- Chosen TLS version
- Random number (for key exchange)
- Session ID
  - If supported / enabled by server
- Chosen cipher suite
  - No list, only the selected one
- Chosen compression method
- Common extensions

If no match on TLS version and cipher suite → Handshake abort with error, e.g.
Firefox: "SSL_ERROR_NO_CYPHER_OVERLAP"
Chrome: "ERR_SSL_VERSION_OR_CIPHER_MISMATCH"
Server: Certificate

Server sends X.509v3 certificate chain

- Server’s certificate has to be the first certificate
- Each following (intermediate) certificate must certify the preceding one
- Root certificates can be excluded
  - Browsers need to know them anyway
Server: ServerKeyExchange

- Carry additional data needed for key exchange
  - Only sent when required for specified protocol
  - Our example: Parameters for ECDH

- Often this information is already within the certificate, e.g. if key exchange is RSA
Server: CertificateRequest

1. ClientHello
2. ServerHello
3. Certificate
   - CertificateRequest
   - ServerKeyExchange
   - ClientKeyExchange
   - CertificateVerify
   - ChangeCipherSpec
   - Finished
4. ChangeCipherSpec
   - Finished

- Request client authentication and tell client expected public key

Only with Client TLS!
Server: ServerHelloDone

TLSv1.2 Record Layer: Handshake Protocol: Server Hello Done
Content Type: Handshake (22)
Version: TLS 1.2 (0x0303)
Length: 4

Handshake Protocol: Server Hello Done
Handshake Type: Server Hello Done (14)
Length: 0

Signal that server has sent all handshake messages
**Client: Certificate**

1. **ClientHello**

2. **ServerHello**
   - Certificate
   - ServerKeyExchange
   - CertificateRequest
   - ServerHelloDone

3. **Certificate**
   - ClientKeyExchange
   - CertificateVerify
   - ChangeCipherSpec
   - Finished

4. **Application Data**

   - Empty if no certificate requested by server

**Only with Client TLS!**
**Client:** ClientKeyExchange

*Carries client’s contribution (= preMaster secret) to key exchange*

- Content depends on used cipher
  - If RSA is used, an RSA-encrypted secret is transferred
  - If Diffie Hellman (DH) is used, only the parameters are sent → enables both parties to agree on same preMaster secret
  - If *ephemeral* Diffie Hellman (DHE) is used, message contains client’s DH public key

---

**Secure Sockets Layer**

TLv1.2 Record Layer: Handshake Protocol: Client Key Exchange
- Content Type: Handshake (22)
- Version: TLS 1.2 (0x0303)
- Length: 70

Handshake Protocol: Client Key Exchange
- Handshake Type: Client Key Exchange (16)
- Length: 65

EC Diffie-Hellman Client Params
- Pubkey Length: 65
- Pubkey: 0440a27d25db5e4e3cc49a61356f8eef85f9d825fd04254...
**Client:** ClientKeyExchange

**Example:** RSA is used for key exchange

**Step 1**
- Client generates „PreMaster secret“ (48 random bytes)
- PreMaster secret encrypted with public key of server certificate
- Server decrypts PreMaster secret with private RSA key

**Step 2**
- Master secret (= session key) is derived by server and client

```
masterSecret = PRF(preMasterSecret, „master secret“, ClientHello.random + ServerHello.random)[0..47]
```

PRF = Pseudo-Random Function
Client: ClientKeyExchange – Security

RSA

- Simpler than others but with a fundamental weakness
  - PreMaster secret encrypted with server’s public key
  - Anyone with access to private key can recover preMaster secret
  - Using preMaster secret → master secret recomputable

Diffie Hellman

- Security depends on quality of chosen parameters
  - If server sends weak or insecure parameters → compromise security of session

- Solution is to use standardized domain parameters of varying strength
Client: CertificateVerify

Client proves possession of private key for sent client certificate

Certificate
ClientHello
ChangeCipherSpec
Finished

ClientKeyExchange
CertificateVerify
Application Data

Certificate
ServerKeyExchange
CertificateRequest
ServerHelloDone

ServerHello
Application Data

ChangeCipherSpec
Finished

Only with Client TLS!
Client & Server: ChangeCipherSpec

Signal that one party has all needed parameters, has generated encryption keys and is switching to encryption

- TLSv1.2 Record Layer: Change Cipher Spec Protocol: Change Cipher Spec
  Content Type: Change Cipher Spec (20)
  Version: TLS 1.2 (0x0303)
  Length: 1
  Change Cipher Spec Message

Sent by client and server as soon as they are ready...
Client & Server: Finished

Signal that handshake is complete

- Purpose is to verify integrity of entire handshake
  - Content is already encrypted

- Message contains hash of all handshake messages
  
  \[
  \text{verify\_data} = \text{PRF(masterSecret, finishedLabel, hash(handshakeMessages))}
  \]
  
  - Integrity of Finished message itself is guaranteed by negotiated MAC algorithm
  - Both parties decrypt message → check hash values

\[\text{TLSv1 Record Layer: Handshake Protocol: Encrypted Handshake Message}\]
Content Type: Handshake (22)
Version: TLS 1.0 (0x0301)
Length: 36
Handshake Protocol: Encrypted Handshake Message
TLS Handshake Summary

1. Client starts handshake, sends parameters to Server
2. Server chooses common connection parameters
3. Server sends his certificate chain
4. If needed for key exchange → Server sends needed parameters to client
5. Server informs client that everything is done

6. Client sends parameters for key exchange to Server
7. Client switches to encrypted communication and informs Server about this
8. Client sends checksum (MAC) of all sent and received handshake messages to Server

9. Server switches to encrypted communication and informs client about this
10. Server also sends MAC of handshake messages
## TLS Record

<table>
<thead>
<tr>
<th>Byte</th>
<th>+0</th>
<th>+1</th>
<th>+2</th>
<th>+3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Content type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1..4</td>
<td>Version</td>
<td>Length</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5..n</td>
<td>Payload</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n..m</td>
<td>MAC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m..p</td>
<td>Padding (block ciphers only)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: [http://goo.gl/7zig7b](http://goo.gl/7zig7b)

---

### Typical workflow

- Record protocol receives application data
- Received data is divided into blocks (max. 16 KB per record)
- Add message authentication code (MAC)
- Data is encrypted using negotiated masterSecret
TLS Properties
Overview

Cryptographic aspects of TLS are fully configurable by cipher suites.

→ Define exactly how security will be implemented

Defines the following attributes

- Key exchange: RSA, DH, DHE, ECDH, ECDHE
- Authentication: RSA, DSA, DSS, ECDSA
- Hash function for MAC: MD5, SHA-1, SHA-256, SHA-512
- Encryption algorithm & key size: none, RC4, (3)DES, AES, ...

→ Ensure TLS principles: Authenticity, Integrity, Confidentiality

Key exchange is a requirement for integrity and confidentiality

Note: RSA can be used for key exchange and authentication!
Cipher Suites

Name construction

Different notations

- IANA: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- OpenSSL: ECDHE-RSA-AES128-GCM-SHA256
- PolarSSL: TLS-ECDHE-RSA-WITH-AES-128-GCM-SHA256

→ [SSL|TLS], [Key Exchange], [Authentication], [Bulk cipher], [MAC]
Cipher Suites

Key Exchange

- ECDH/ECDHE is similar to DH/DHE but faster!
- ECDH keys with elliptic curves instead of DH parameters
- Table of equivalent key lengths:

<table>
<thead>
<tr>
<th>Symmetrisch</th>
<th>RSA / DH</th>
<th>ECDH</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>1024</td>
<td>160</td>
</tr>
<tr>
<td>112</td>
<td>2048</td>
<td>224</td>
</tr>
<tr>
<td>128</td>
<td>3072</td>
<td>256</td>
</tr>
<tr>
<td>192</td>
<td>7680</td>
<td>384</td>
</tr>
<tr>
<td>256</td>
<td>15360</td>
<td>512</td>
</tr>
</tbody>
</table>

DH = Diffie Hellman
DHE = Diffie Hellman Ephemeral
ECDH = Elliptic Curve Diffie Hellman
ECDHE = Elliptic Curve Diffie Hellman Ephemeral

Note

- ECDH/ECDHE is similar to DH/DHE but faster!
- ECDH keys with elliptic curves instead of DH parameters
- Table of equivalent key lengths:

## Cipher Suites

**openssl ciphers -v**

<table>
<thead>
<tr>
<th>Cipher Suite</th>
<th>Version</th>
<th>Key Exchange</th>
<th>Authentication</th>
<th>Encryption</th>
<th>Mac</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_AES_256_GCM_SHA384</td>
<td>TLSv1.3</td>
<td>Kx=any</td>
<td>Au=any</td>
<td>Enc=AESGCM(256)</td>
<td>Mac=AEAD</td>
</tr>
<tr>
<td>TLS_CHACHA20_POLY1305_SHA256</td>
<td>TLSv1.3</td>
<td>Kx=any</td>
<td>Au=any</td>
<td>Enc=CHACHA20/POLY1305(256)</td>
<td></td>
</tr>
<tr>
<td>TLS_AES_128_GCM_SHA256</td>
<td>TLSv1.3</td>
<td>Kx=any</td>
<td>Au=any</td>
<td>Enc=AESGCM(128)</td>
<td>Mac=AEAD</td>
</tr>
<tr>
<td>ECDHE-RSA-AES256-GCM-SHA384</td>
<td>TLSv1.2</td>
<td>Kx=ECDH</td>
<td>Au=RSA</td>
<td>Enc=AESGCM(256)</td>
<td>Mac=AEAD</td>
</tr>
<tr>
<td>ECDHE-ECDSA-AES256-GCM-SHA384</td>
<td>TLSv1.2</td>
<td>Kx=ECDH</td>
<td>Au=ECDSA</td>
<td>Enc=AESGCM(256)</td>
<td>Mac=AEAD</td>
</tr>
<tr>
<td>ECDHE-RSA-AES256-SHA384</td>
<td>TLSv1.2</td>
<td>Kx=ECDH</td>
<td>Au=RSA</td>
<td>Enc=AES(256)</td>
<td>Mac=SHA384</td>
</tr>
<tr>
<td>ECDHE-ECDSA-AES256-SHA384</td>
<td>TLSv1.2</td>
<td>Kx=ECDH</td>
<td>Au=ECDSA</td>
<td>Enc=AES(256)</td>
<td>Mac=SHA384</td>
</tr>
<tr>
<td>ECDHE-RSA-AES256-SHA</td>
<td>SSLv3</td>
<td>Kx=ECDH</td>
<td>Au=RSA</td>
<td>Enc=AES(256)</td>
<td>Mac=SHA1</td>
</tr>
<tr>
<td>ECDHE-ECDSA-AES256-SHA</td>
<td>SSLv3</td>
<td>Kx=ECDH</td>
<td>Au=ECDSA</td>
<td>Enc=AES(256)</td>
<td>Mac=SHA1</td>
</tr>
<tr>
<td>SRP-DSS-AES-256-CBC-SHA</td>
<td>SSLv3</td>
<td>Kx=SRP</td>
<td>Au=DSS</td>
<td>Enc=AES(256)</td>
<td>Mac=SHA1</td>
</tr>
<tr>
<td>SRP-RSA-AES-256-CBC-SHA</td>
<td>SSLv3</td>
<td>Kx=SRP</td>
<td>Au=RSA</td>
<td>Enc=AES(256)</td>
<td>Mac=SHA1</td>
</tr>
<tr>
<td>SRP-AES-256-CBC-SHA</td>
<td>SSLv3</td>
<td>Kx=SRP</td>
<td>Au=SRP</td>
<td>Enc=AES(256)</td>
<td>Mac=SHA1</td>
</tr>
<tr>
<td>DHE-DSS-AES256-GCM-SHA384</td>
<td>TLSv1.2</td>
<td>Kx=DH</td>
<td>Au=DSS</td>
<td>Enc=AESGCM(256)</td>
<td>Mac=AEAD</td>
</tr>
</tbody>
</table>

(...)  

For complete list, see [http://goo.gl/Jg5wUp](http://goo.gl/Jg5wUp)
Cipher Suites in the Browser

Which are offered by your client?

- Depends on used library
  - Internet Explorer (Edge): Cryptography Service Provider (CSP)
  - Mozilla Firefox: Network Security Services (NSS)
  - Google Chrome: NSS with own adaptions
  - Apple Safari: SecureTransport
  - Android: AndroidOpenSSL and BouncyCastle (modified)

→ Modern browsers prefer AES-GCM and AES-CBC
Find out your preferences at https://www.howsmyssl.com
Cipher Suites in the Browser

This page is secure (valid HTTPS).

- **Certificate - valid and trusted**
  The connection to this site is using a valid, trusted server certificate issued by DigiCert SHA2 High Assurance Server CA.

- **Connection - secure connection settings**
  The connection to this site is encrypted and authenticated using TLS 1.3, X25519, and AES_128_GCM.

- **Resources - all served securely**
  All resources on this page are served securely.
(Perfect) Forward Secrecy

Compromise of long-term keys should not compromise past session keys

Without Forward Secrecy

- Security of all connections depend on server’s private key
- If broken or stolen → previous communication can be decrypted

Why is this possible?

- During the handshake, the client creates a preMaster secret
- Encrypted using the server’s public (RSA) key it is sent to the server
  - Server uses his private key to decrypt it → calculate common masterSecret

→ If you have the private key, you can decrypt past and future data!!
Without PFS

Key Exchange via RSA (no PFS)

The client generates a session key, encrypts it via RSA, and sends it to the server.

- Generates session key $K_{Sess}$
- (RSA Encryption) Encrypts $K_{Sess}$ with the public long term key from the server $K_{Pub}$ and sends it to server

Communication encrypted with symmetric cipher using $K_{Sess}$

- (RSA Decryption) Decrypts $K_{Sess}$ with its private key $K_{Priv}$

Complete communication is stored by a third party

If the third party has access to $K_{Priv}$ some day, it can subsequently decrypt all communication since it can reproduce all session keys.

Source: [http://goo.gl/q1FfGS](http://goo.gl/q1FfGS)
(Perfect) Forward Secrecy

With Forward Secrecy

- Server generates a *short-living („ephemeral“) Diffie-Hellman keypair*
  - DHE = Diffie-Hellman *Ephemeral*
  - ECDHE = Elliptic Curve Diffie-Hellman *Ephemeral*

- Server signs the public key of this DH pair with the private key of the server’s certificate
  - Can be RSA or ECDSA depending on the certificate

- Client receives the signed public DH key, checks if signature is verifiable using public key of the previously received server’s certificate

→ Instead of „Key transport“ (RSA), forward secrecy works with „Key agreement“!
With PFS

Key Agreement via DH (with PFS)

Client

- (Diffie-Hellman) Generates random value a and computes A
- Sends A to the server
- Computes K_Sess from input of itself (a) and the server (B).

Server

- (Diffie-Hellman) Generates random value b and computes B
- Sends B to the client
- Computes K_Sess from input of itself (b) and the client (A).

Communication encrypted with symmetric cipher using K_Sess

Complete communication is stored by a third party

Source: [http://goo.gl/q1FfGS](http://goo.gl/q1FfGS)

Note: This graphic misses the key signing part!
(Perfect) Forward Secrecy

Security

- For every new session, client & server generate new Diffie-Hellman parameters
  - If compromised somehow → attacker could only read this particular session

- Attacking the session key
  - If parameters are securely chosen, brute-force should not be possible
    - E.g. use 2048-bit or stronger Diffie-Hellman groups with „safe“ primes

- Attacking the server’s private key
  - With PFS, only used to sign ephemeral public DH keys sent to the client
  - If broken or leaked → would not compromise past sessions

- Hacking the server: Attacker only gets current session keys & key for signatures
(Perfect) Forward Secrecy

How to get Forward Secrecy?

- Server needs at least TLS 1.2 + offer PFS supporting cipher suite
- Important: Only key exchange with DHE or ECDHE offers forward secrecy!
  - Cipher suite, e.g. DHE-RSA-AES128-SHA or ECDHE-ECDSA-AES128-SHA

Test servers

- [https://www.ssllabs.com/ssltest/](https://www.ssllabs.com/ssltest/)
- [https://testssl.sh](https://testssl.sh)
- [https://github.com/nabla-c0d3/sslyze](https://github.com/nabla-c0d3/sslyze)

- Examples on how not to configure servers: [https://badssl.com](https://badssl.com)
  - Small DH groups, weak ciphers, etc.
# (Perfect) Forward Secrecy

SSL/TLS Server Test for online.tugraz.at

**Server information**

- **Hostname**: online.tugraz.at
- **IP address**: 129.27.2.210
- **Port**: 443
- **Server type**: Apache

<table>
<thead>
<tr>
<th>Cipher suite</th>
<th>Key exchange</th>
<th>MAC</th>
<th>Cipher</th>
<th>Key length</th>
<th>Forward Secrecy</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_DHE_RSA_WITH_AES_256_GCM_SHA384</td>
<td>DHE RSA 2048 bits</td>
<td>SHA384</td>
<td>AES/GCM/NoPadding</td>
<td>256 bits</td>
<td>Yes</td>
</tr>
<tr>
<td>TLS_DHE_RSA_WITH_AES_256_CBC_SHA256</td>
<td>DHE RSA 2048 bits</td>
<td>SHA256</td>
<td>AES/CBC/NoPadding</td>
<td>256 bits</td>
<td>Yes</td>
</tr>
<tr>
<td>TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384</td>
<td>ECDHE RSA 256 bits</td>
<td>SHA384</td>
<td>AES/GCM/NoPadding</td>
<td>256 bits</td>
<td>Yes</td>
</tr>
<tr>
<td>TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384</td>
<td>ECDHE RSA 256 bits</td>
<td>SHA384</td>
<td>AES/CBC/NoPadding</td>
<td>256 bits</td>
<td>Yes</td>
</tr>
<tr>
<td>TLS_DHE_RSA_WITH_AES_128_GCM_SHA256</td>
<td>DHE RSA 2048 bits</td>
<td>SHA256</td>
<td>AES/GCM/NoPadding</td>
<td>128 bits</td>
<td>Yes</td>
</tr>
<tr>
<td>TLS_DHE_RSA_WITH_AES_128_CBC_SHA256</td>
<td>DHE RSA 2048 bits</td>
<td>SHA256</td>
<td>AES/CBC/NoPadding</td>
<td>128 bits</td>
<td>Yes</td>
</tr>
<tr>
<td>TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256</td>
<td>ECDHE RSA 256 bits</td>
<td>SHA256</td>
<td>AES/GCM/NoPadding</td>
<td>128 bits</td>
<td>Yes</td>
</tr>
</tbody>
</table>
TLS Security
Overview

Problem
Attacks often based on downgrades HTTPS → HTTP ("SSLStrip")

- Variant A
  - Web page offers HTTP and HTTPS version
  - Attacker injects HTTP links to force user to use weak HTTP communication

- Variant B
  - Web page offers HTTPS only
  - Attacker uses proxy server (Man-in-the-middle) and translates to HTTP communication

Solution? HTTP Strict Transport Security (HSTS)
HSTS

= Tell browser that all connections to a domain are HTTPS only

→ Specified via HTTP header that can only be sent during valid HTTPS request

```
Strict-Transport-Security: max-age=10886400; includeSubDomains
```

Browser remembers (for specified max-age period) that it should only request HTTPS resources for this site (and optionally subdomains)

→ Effectively prevents „SSL Stripping“ attacks!
**HSTS**

*But: What if an attacker has control over the initial HTTPS requests?*

**Scenario**
- Attacker would strip HSTS headers
- Browsers would not know HSTS should be active

**Solution**
- Browsers ship with „preloaded“ HSTS lists → Sites that *always* require HTTPS
- Add „preload“ header and add domain here: [https://hstspreload.appspot.com](https://hstspreload.appspot.com)

```plaintext
Strict-Transport-Security: max-age=10886400; includeSubDomains; preload
```
Man-in-the-Middle

Problem
You are not presented the „correct“ certificate for a domain

- Variant A
  - Attacker malevolently exchanges certificate with self-generated one
  - Client connects and attacker redirects data transfer

- Variant B
  - Certificate Authority (CA) is compromised
  - Attacker generates trusted certificate and exchanges it

Solution? HTTP Public Key Pinning (HPKP)
Certificate Pinning (HPKP)

Problem
Our browsers trust ~130 CAs ("Trust Store")

How is trust established?
1. Browser compares DNS hostname with subject name in certificate
2. Upon match, check if certificate issued by trusted CA
Certificate Pinning

Scenario

Usually the certificate chain for google.com looks as follows:

GlobalSign Root CA - R2
  - GTS CA 101
  - google.com

Now:

- Assume „TÜRKTRUST Elektronik Sunucu Sertifikası Hizmetleri“ issues a certificate for google.com
- A webserver for google.com is setup, DNS entries are rewritten to point at that server and the user is forwarded there → would he notice?

TÜRKTRUST Elektronik Sunucu Sertifikası Hizmetleri
  - e-islem.kktcmerkezbankasi.org
  - google.com

No, he would not!
https://goo.gl/QVfHYV
Certificate Pinning

Another scenario
1. Attacker has access to trusted CA, issues certificates for arbitrary hostnames
2. Attacker performs MITM attack using previously generated certificate
   → Attacker could replace any TLS certificate, browser would still trust it

Remedy?
- Remember hash values („pins“) of public keys associated with certificates
- If PIN changes (= certificate changes), drop connection even if certificate would be trustworthy and DNS name matches with cert’s subject name
- PINs either stored in browser (or mobile app) or sent via HTTP header
Certificate Pinning

Public-Key-Pins:

```plaintext
pin-sha256="GRAH5Ex+kB4cCQi5gMU82urf+6kEgbVtzfCSkw55AGk=";
pin-sha256="lERGk61FITjzyKHcJ89xpc6aDwtRk0PAU0jdnUqzW2s=";
max-age=15768000; includeSubDomains
```

How to generate PINs?

- Get SHA-256 hash value of public key of server certificate
- Base-64 encoding of hash and inserting into header

Advantages

- Defeats MITM attacks
- PIN can also be stored in browser

Disadvantages

- “Trust-on-first-use“ mechanism (like HSTS)
- Many things can go wrong while setup
- You **must** have >= 2 PINs
Outlook

- **24.01.2020**
  - TLS Vulnerabilities & Attacks
  - DNS Security

- **31.01.2020**
  - Lecture Exam