

M1 Cyberschool, Université de Rennes

Introduction to security and cryptography: TLS

Aurore Guillevic

`aurore.guillevic@inria.fr`

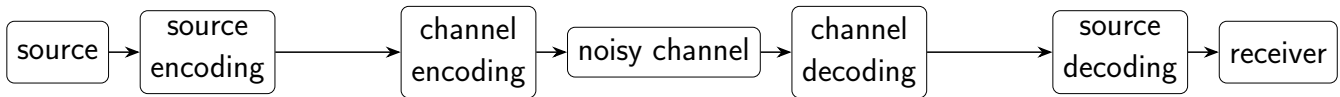
Première partie

Introduction to cryptography

(Slides de Marion Videau)

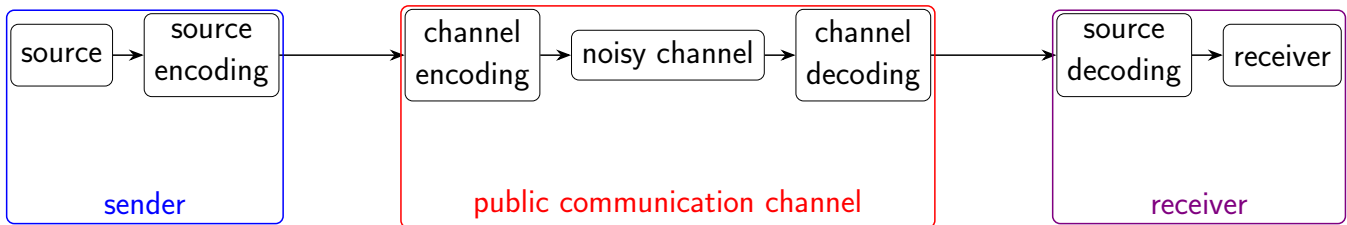
Model of a communication channel

Model of a communication channel, general [Shannon, 1948]



Model of a communication channel

Model of a communication channel, without noise



Model of a communication channel

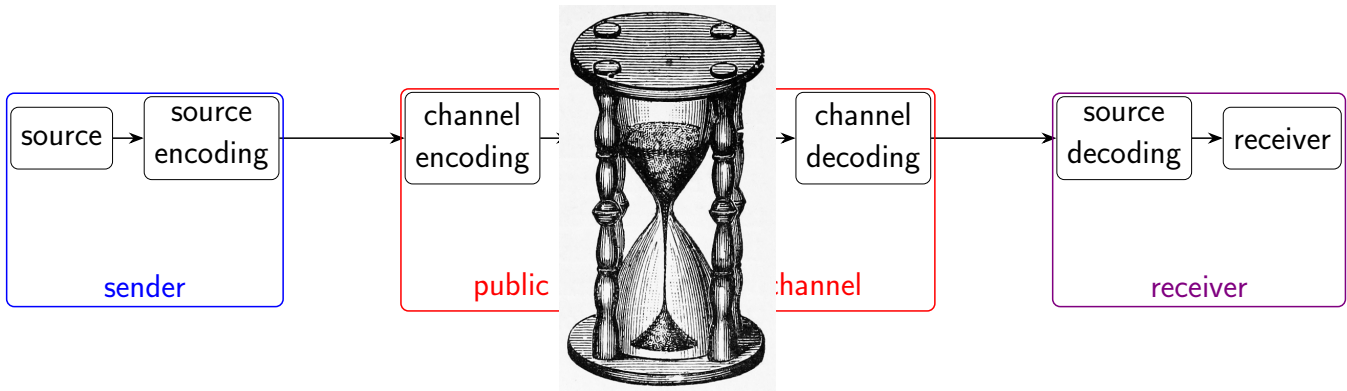
Model of a communication channel, without noise



<https://commons.wikimedia.org/w/index.php?curid=805736>

Model of a communication channel

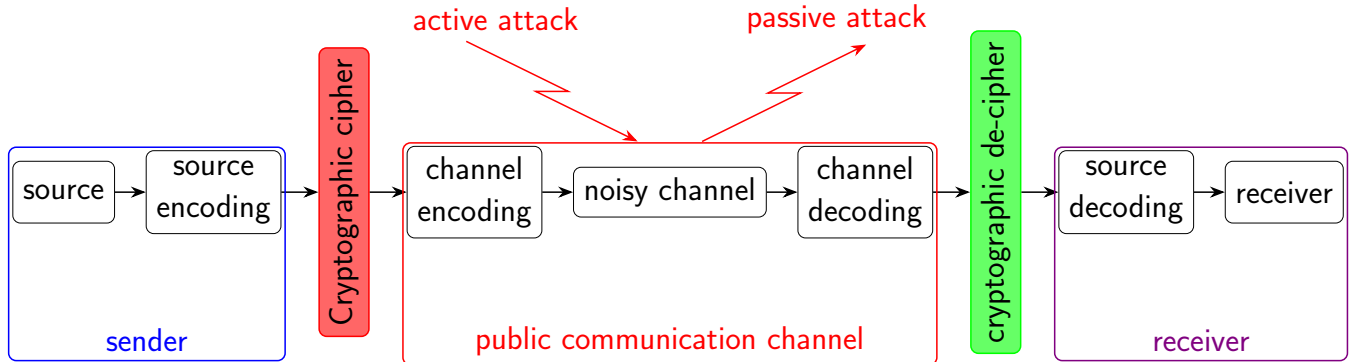
Model of a communication channel, without noise



<https://commons.wikimedia.org/w/index.php?curid=1015977>

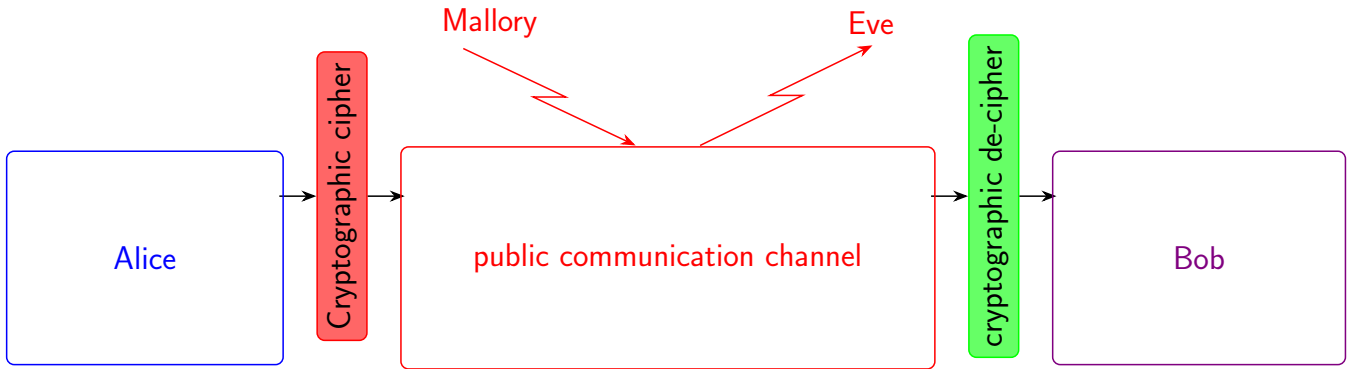
Model of a communication channel

Model of a communication channel, cryptographic



Model of a communication channel

Model of a communication channel, cryptographic



Different threats

An attack can be

- ▶ passive : espionage (spying) / intelligence
- ▶ active :
 - impersonation (identity fraud) (of emitter or receiver)
 - data tampering = modification of message content
 - message repudiation = the sender denies having sent the message
 - message repetition = subsequent replay

 - transmission delay
 - message destruction

Answers thanks to cryptography

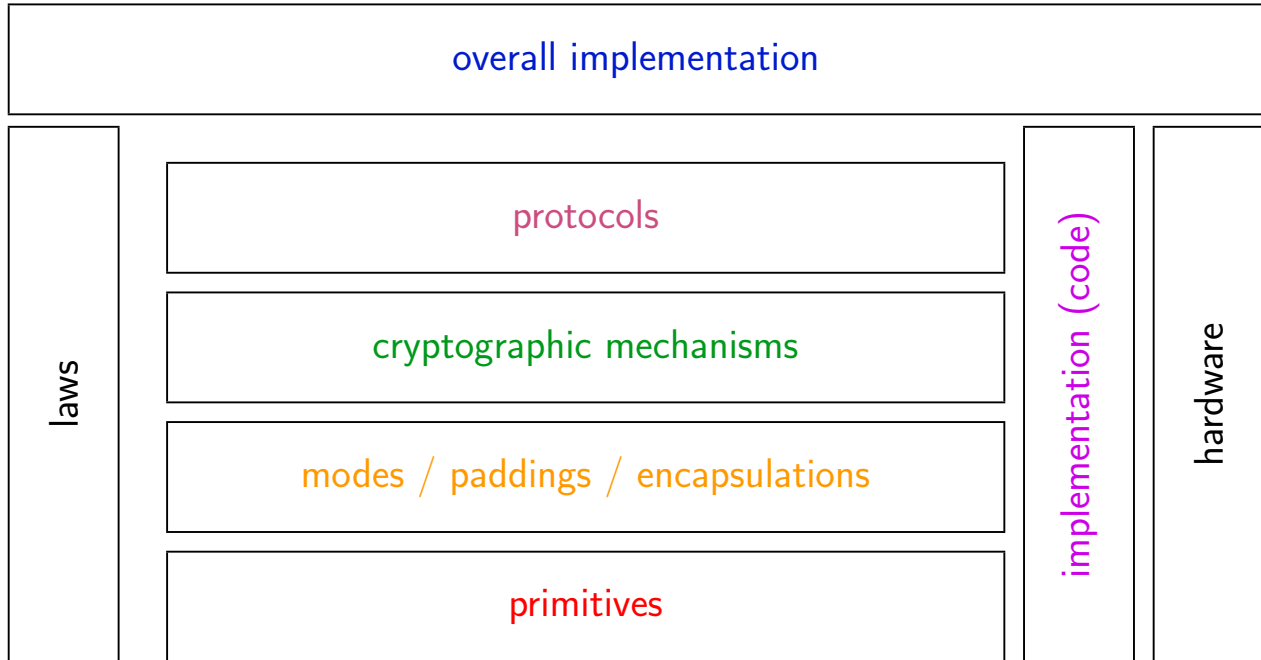
Three fundamental aspects :

- ▶ **confidentiality** (Who receives the message ?) : information does not reach people not entitled to know it
- ▶ **integrity** : information is not altered, there is no fraud
- ▶ **authentication** (Who sent the message ?) : information comes from its true emitter

More elaborated functionalities :

- **non-repudiation** (signer cannot deny having signed the document),
- **zero-knowledge** (proof, interaction),
- **electronic voting**,
- **secret sharing**,
- **broadcast encryption**,
- *etc.*

Layers of cryptography



Fundamental cryptographic mechanisms

Algorithms providing a fundamental cryptographic functionality

- ▶ Cryptographic integrity check → [hash function](#)
- ▶ Key generation, IV (Init Vector), randomness → [cryptographic \(pseudo-\)random generator](#)
- ▶ Authentication of message origin → [message authentication code \(MAC\)](#), [signature algorithm](#)
- ▶ Confidentiality → [cipher](#)

Simplified classification of fundamental cryptographic mechanisms

Mechanisms	without key		secret key		public key	
Types	hash function	random generator	MAC	cipher (chiffrement)	signature	cipher (chiffrement)
Examples	SHA-384	/dev/random	AES-OMAC	AES-CTR	RSA-PSS	RSA-OAEP
Examples of modes, <i>padding</i> s or encapsulations	chop-MD, sponge, MD-strengthening	Barak-Halevi	HMAC, OMAC, Wegman-Carter	stream (<i>flot</i>), block, auto-synchronous, synchronous, authenticating, dedicated, CBC, CFB, OFB, CTR, CCM, GCM, OCB, XEX	PSS	OAEP
Examples of primitives	compression function	hash function, stream cipher, block cipher, MAC	hash function, stream cipher, block cipher, universal hash function	retroaction function / filtering / initialization, parameterized random permutation (block cipher)	RSA, ElGamal, ECDSA, EdDSA, NTRU, McEliece	RSA, ElGamal, NTRU, McEliece
Additional mechanisms (optional)	random generator (salt)	physical random generator, entropy source	random generator (key)	random generator (key, IV)	random generator (key, salt), hash function	random generator (key, salt), hash function
Examples of standards	FIPS 180-3 (SHS)	NIST SP 800-90	NIST SP 800-38B	FIPS 197 (AES), NIST SP 800-38A	FIPS 186-3 (DSS)	PKCS#1 v2.1

Deuxième partie

Cryptographic protocol example : SSL/TLS

(Slides de Jérémie Detrey)

The SSL/TLS protocol

- ▶ Protocol for **securing transmissions** over the Internet
- ▶ **Client-Server** mode, on top of **TCP** (Transmission Control Protocol)
- ▶ Ensures
 - **Authentication** of the server
 - **Confidentiality** of exchanged data
 - **Integrity** and **Authentication of the origin** of the exchanged data
 - **Authentication** of the client (optional)
- ▶ Enables transparent encapsulation of protocols of the **application layer**
 - HTTP (80) → **HTTPS** (443)
 - IMAP (143) → **IMAPS** (993)
 - POP3 (110) → **POP3S** (995)
 - **STARTTLS** (for IMAP, POP3, SMTP, FTP, etc.) over the **same port**

Some historical facts

- ▶ At the beginning, **SSL** (*Secure Sockets Layer*), developed by Netscape (first browser)
 - **SSL 1.0** (1994) : theoretical protocol, never used
 - **SSL 2.0** (1995-2011) : first version used
 - **SSL 3.0** (1996-..., RFC 6101) : last version, on which TLS will be built

Some historical facts

- ▶ At the beginning, **SSL** (*Secure Sockets Layer*), developed by Netscape (first browser)
 - **SSL 1.0** (1994) : theoretical protocol, never used
 - **SSL 2.0** (1995-2011) : first version used
 - **SSL 3.0** (1996-..., RFC 6101) : last version, on which TLS will be built
- ▶ Then **TLS** (*Transport Layer Security*), developed by IETF (*Internet Engineering Task Force*)
 - **TLS 1.0** (1999-..., RFC 2246) : took over SSL, various improvements until 2002
 - **TLS 1.1** (2006-..., RFC 4346)
 - **TLS 1.2** (2008-..., RFC 5246)
 - **TLS 1.3** (2018-..., RFC 8446) Some differences between 1.2 and 1.3

Some historical facts

- ▶ At the beginning, **SSL** (*Secure Sockets Layer*), developed by Netscape (first browser)
 - **SSL 1.0** (1994) : theoretical protocol, never used
 - **SSL 2.0** (1995-2011) : first version used
 - **SSL 3.0** (1996-..., RFC 6101) : last version, on which TLS will be built
- ▶ Then **TLS** (*Transport Layer Security*), developed by IETF (*Internet Engineering Task Force*)
 - **TLS 1.0** (1999-..., RFC 2246) : took over SSL, various improvements until 2002
 - **TLS 1.1** (2006-..., RFC 4346)
 - **TLS 1.2** (2008-..., RFC 5246)
 - **TLS 1.3** (2018-..., RFC 8446) Some differences between 1.2 and 1.3
- ▶ **Compatibility** of the servers HTTPS (source : **SSL Pulse**, 12/2023)

SSL 2.0	SSL 3.0	TLS 1.0	TLS 1.1	TLS 1.2	TLS 1.3
0,2 %	1,6 %	29,1 %	31,4 %	99,9 %	66,9 %

Cryptographic mechanisms in TLS

- ▶ TLS offers the choice of **several cryptographic mechanisms** for flexibility and to adapt to the **constraints** of each application and system

Cryptographic mechanisms in TLS

- ▶ TLS offers the choice of **several cryptographic mechanisms** for flexibility and to adapt to the **constraints** of each application and system
- ▶ **Authentication** of the server (and the client, optionally) :
 - public key : **RSA**, **DSS**, **ECDSA**
 - secret key or shared password : **PSK** (*Pre-Shared Key*), **SRP** (*Secure Remote Password*)
 - no authentication : **ANON**

Cryptographic mechanisms in TLS

- ▶ TLS offers the choice of **several cryptographic mechanisms** for flexibility and to adapt to the **constraints** of each application and system
- ▶ **Authentication** of the server (and the client, optionally) :
 - public key : **RSA**, **DSS**, **ECDSA**
 - secret key or shared password : **PSK** (*Pre-Shared Key*), **SRP** (*Secure Remote Password*)
 - no authentication : **ANON**
- ▶ **Key exchange** :
 - public key (always the same private key on the server's side) : **RSA**
 - Static Diffie–Hellman (same problem) : **DH**, **ECDH**
 - secret key or shared password : **PSK**, **SRP**
 - Ephemeral Diffie–Hellman (secret keys **for each single connection** ; ensures *forward secrecy*) : **DHE**, **ECDHE**

Cryptographic mechanisms in TLS

► Encryption :

- block cipher : AES-CBC, 3DES-CBC, DES-CBC, etc.
- block cipher with authentication mode : AES-CCM, AES-GCM, etc.
- stream cipher : RC4
- no encryption : NULL

Cryptographic mechanisms in TLS

► Encryption :

- block cipher : AES-CBC, 3DES-CBC, DES-CBC, etc.
- block cipher with authentication mode : AES-CCM, AES-GCM, etc.
- stream cipher : RC4
- no encryption : NULL

► Integrity and authentication of origin of the messages :

- HMAC : HMAC-MD5, HMAC-SHA1, HMAC-SHA256, etc.
- block cipher authentication mode : AEAD

Cryptographic mechanisms in TLS

► Encryption :

- block cipher : AES-CBC, 3DES-CBC, DES-CBC, etc.
- block cipher with authentication mode : AES-CCM, AES-GCM, etc.
- stream cipher : RC4
- no encryption : NULL

► Integrity and authentication of origin of the messages :

- HMAC : HMAC-MD5, HMAC-SHA1, HMAC-SHA256, etc.
- block cipher authentication mode : AEAD

► A combined set of these mechanisms is named a *cipher suite*

- for example : TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

Cryptographic mechanisms in TLS

► Encryption :

- block cipher : AES-CBC, 3DES-CBC, DES-CBC, etc.
- block cipher with authentication mode : AES-CCM, AES-GCM, etc.
- stream cipher : RC4
- no encryption : NULL

► Integrity and authentication of origin of the messages :

- HMAC : HMAC-MD5, HMAC-SHA1, HMAC-SHA256, etc.
- block cipher authentication mode : AEAD

► A combined set of these mechanisms is named a *cipher suite*

- for example : TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

► As a bonus, TLS can support a data compression mode : NULL or DEFLATE

Establishing a SSL/TLS connection

- ▶ Context : a **client** aims at opening a SSL/TLS connection with a **server**
- ▶ Aims :
 - agreeing on a **common *cipher suite***
 - **authenticate** the server
 - exchange **secret keys** for encryption and authentication of the messages
- ▶ ***Handshake*** protocol in 4 steps

Simplified SSL/TLS Handshake

1. Client → Server

- **ClientHello** : highest supported version of the protocol, list of *cipher suites* and supported compression modes

Simplified SSL/TLS Handshake

1. Client → Server

- **ClientHello** : highest supported version of the protocol, list of *cipher suites* and supported compression modes

2. Server → Client

- **ServerHello** : chosen protocol version (usually 1.3 or 1.2), *cipher suite* and compression mode
- **Certificate** (optional) : server public key in a X.509 certificate so that its authenticity can be verified
- **ServerKeyExchange** (optional) : Diffie-Hellman public key for key exchange
- **ServerHelloDone**

Simplified SSL/TLS Handshake

3. Client → Server

- **ClientKeyExchange** : either a secret (*PreMasterKey*) encrypted with the server public key, or a Diffie-Hellman public key for the key exchange
- **ChangeCipherSpec** (ends the *handshake* on the client side)
- **Finished** : encrypted and authenticated message containing a MAC of the previous messages of the *handshake*

Simplified SSL/TLS Handshake

3. Client → Server

- **ClientKeyExchange** : either a secret (*PreMasterKey*) encrypted with the server public key, or a Diffie-Hellman public key for the key exchange
- **ChangeCipherSpec** (ends the *handshake* on the client side)
- **Finished** : encrypted and authenticated message containing a MAC of the previous messages of the *handshake*

4. Server → Client (if the client's **Finished** is valid)

- **ChangeCipherSpec** (ends the *handshake* on the server side)
- **Finished** : encrypted and authenticated message containing a MAC of the previous messages of the *handshake*

Simplified SSL/TLS Handshake

3. Client → Server

- **ClientKeyExchange** : either a secret (*PreMasterKey*) encrypted with the server public key, or a Diffie-Hellman public key for the key exchange
- **ChangeCipherSpec** (ends the *handshake* on the client side)
- **Finished** : encrypted and authenticated message containing a MAC of the previous messages of the *handshake*

4. Server → Client (if the client's **Finished** is valid)

- **ChangeCipherSpec** (ends the *handshake* on the server side)
- **Finished** : encrypted and authenticated message containing a MAC of the previous messages of the *handshake*

5. If the **Finished** of the server is also valid, the connection is established

Server Authentication

- ▶ In the previous *handshake*, the server sends *itself* its *public key* to *authenticate its messages*
→ Isn't it a problem ?
- ▶ How to *check the authenticity of the server public key* ?

Server Authentication

- ▶ In the previous *handshake*, the server sends *itself* its *public key* to *authenticate its messages*
→ Isn't it a problem ?
- ▶ How to *check the authenticity of the server public key* ?
 - Public key *signed by the server* ?

Server Authentication

- ▶ In the previous *handshake*, the server sends *itself* its *public key* to *authenticate its messages*
 - Isn't it a problem ?
- ▶ How to *check the authenticity of the server public key* ?
 - Public key *signed by the server* ?
 - The verification requires to trust the public key...

Server Authentication

- ▶ In the previous *handshake*, the server sends *itself* its *public key* to *authenticate its messages*
 - Isn't it a problem ?
- ▶ How to *check the authenticity of the server public key* ?
 - Public key *signed by the server* ?
 - The verification requires to trust the public key...
 - Public key *signed by a third party* (named *Certification Authority*, or CA) ?

Server Authentication

- ▶ In the previous *handshake*, the server sends *itself* its *public key* to *authenticate its messages*
 - Isn't it a problem ?
- ▶ How to *check the authenticity of the server public key* ?
 - Public key *signed by the server* ?
 - The verification requires to trust the public key...
 - Public key *signed by a third party* (named *Certification Authority*, or CA) ?
 - OK, but how to verify this signature ?

Server Authentication

- ▶ In the previous *handshake*, the server sends *itself* its *public key* to *authenticate its messages*
 - Isn't it a problem ?
- ▶ How to *check the authenticity of the server public key* ?
 - Public key *signed by the server* ?
 - The verification requires to trust the public key...
 - Public key *signed by a third party* (named *Certification Authority*, or CA) ?
 - OK, but how to verify this signature ?
 - Public key *signed par a CA*, and *CA public key* ?

Server Authentication

- ▶ In the previous *handshake*, the server sends *itself* its *public key* to *authenticate its messages*
 - Isn't it a problem ?
- ▶ How to *check the authenticity of the server public key* ?
 - Public key *signed by the server* ?
 - The verification requires to trust the public key...
 - Public key *signed by a third party* (named *Certification Authority*, or CA) ?
 - OK, but how to verify this signature ?
 - Public key *signed par a CA*, and *CA public key* ?
 - How to verify the *authenticity of the CA public key* ? The problem just moved to another place...

Server Authentication

- ▶ In the previous *handshake*, the server sends *itself* its *public key* to *authenticate its messages*
→ Isn't it a problem ?
- ▶ How to *check the authenticity of the server public key* ?
 - Public key *signed by the server* ?
→ The verification requires to trust the public key...
 - Public key *signed by a third party* (named *Certification Authority*, or CA) ?
→ OK, but how to verify this signature ?
 - Public key *signed par a CA*, and *CA public key* ?
→ How to verify the *authenticity of the CA public key* ? The problem just moved to another place...
- ▶ It is a difficult and complex problem, which requires to set up a *Public Key Infrastructure* (or PKI)

Public Key Infrastructure

- ▶ Enables to answer the question :

How to trust a public key ?

- ▶ Certificate : signature of the public key by a trusted third party

Public Key Infrastructure

- ▶ Enables to answer the question :

How to trust a public key ?

- ▶ Certificate : signature of the public key by a trusted third party
- ▶ Possible infrastructures :
 - hierarchical : certification authorities (SSL/TLS and X.509, EMV)
 - decentralized : trust network (PGP, GnuPG)

Certification Authorities

- ▶ Several levels :
 - Root CA : can certify the public keys of other CA
 - Intermediate CA : in general, cannot certify other CA
→ certify the server public keys

Certification Authorities

- ▶ Several levels :
 - Root CA : can certify the public keys of other CA
 - Intermediate CA : in general, cannot certify other CA
→ certify the server public keys
- ▶ Certification chain :

root CA $\xrightarrow{\text{signs}}$ CA 1 $\xrightarrow{\text{signs}}$ CA 2 $\xrightarrow{\text{signs}}$ Server

Certification Authorities

- ▶ Several levels :
 - Root CA : can certify the public keys of other CA
 - Intermediate CA : in general, cannot certify other CA
→ certify the server public keys
- ▶ Certification chain :

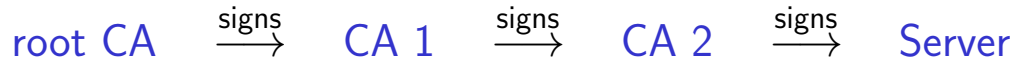
root CA $\xrightarrow{\text{signs}}$ CA 1 $\xrightarrow{\text{signs}}$ CA 2 $\xrightarrow{\text{signs}}$ Server

- ▶ Authentication : the server sends 3 certificates
 - its own public key, signed by CA 2
 - the public key of CA 2, signed by CA 1
 - the public key of CA 1, signed by root CA

Certification Authorities

- ▶ Several levels :
 - Root CA : can certify the public keys of other CA
 - Intermediate CA : in general, cannot certify other CA
→ certify the server public keys

- ▶ Certification chain :



- ▶ Authentication : the server sends 3 certificates
 - its own public key, signed by CA 2
 - the public key of CA 2, signed by CA 1
 - the public key of CA 1, signed by root CA
- ▶ Verification : if the client knows (and trusts) the root CA public key, it can verify the validity of all the certificates

Certification Authorities

- ▶ Always the same problems :
 - how to obtain the public key of root CA ?
 - How much trust to place in them ?

Certification Authorities

- ▶ Always the same problems :
 - how to obtain the **public key of root CA** ?
 - **How much** trust to place in them ?
- ▶ List of the **trusted root CA** maintained by **browsers**
 - at present, **80+ root CA** in Firefox
 - but how to **trust the browser** that we download ?!
 - **integrity and authenticity check** of the downloaded code ?
 - **chicken-and-egg problem**...

Certification Authorities

- ▶ Always the same problems :
 - how to obtain the **public key of root CA** ?
 - **How much** trust to place in them ?
- ▶ List of the **trusted root CA** maintained by **browsers**
 - at present, **80+ root CA** in Firefox
 - but how to **trust the browser** that we download ?!
 - **integrity and authenticity check** of the downloaded code ?
 - **chicken-and-egg problem**...
- ▶ **Pay attention to the CA security** (root and intermediate) !
 - if the CA **private key** is compromised : **issue of false certificates** for example **DigiNotar** in 2011 : 500 false certificates, including ***.google.com**
 - regular **security audits**
 - **revocation mechanism** of the compromised certificates : CRL (*Certificate Revocation List*) or OCSP (*Online Certificate Status Protocol*)

Vulnerabilities in SSL/TLS

- ▶ Vulnerabilities are continuously found in SSL/TLS
 - example : DROWN, revealed on March 1st, 2016

Vulnerabilities in SSL/TLS

- ▶ Vulnerabilities are continuously found in SSL/TLS
 - example : DROWN, revealed on March 1st, 2016
- ▶ Patches exist, but the servers must be updated

Vulnerabilities in SSL/TLS

- ▶ Vulnerabilities are continuously found in SSL/TLS
 - example : DROWN, revealed on March 1st, 2016
- ▶ Patches exist, but the servers must be updated
 - insecure renegotiation (2009, MITM) : 1,8 + 0,6 % vulnerable servers

Vulnerabilities in SSL/TLS

- ▶ Vulnerabilities are continuously found in SSL/TLS
 - example : DROWN, revealed on March 1st, 2016
- ▶ Patches exist, but the servers must be updated
 - insecure renegotiation (2009, MITM) : 1,8 + 0,6 % vulnerable servers
 - BEAST (*Browser Exploit Against SSL/TLS*, 2011, violation of the original purpose/constraint of cookies) : 91,5 %

Vulnerabilities in SSL/TLS

- ▶ Vulnerabilities are continuously found in SSL/TLS
 - example : DROWN, revealed on March 1st, 2016
- ▶ Patches exist, but the servers must be updated
 - insecure renegotiation (2009, MITM) : 1,8 + 0,6 % vulnerable servers
 - BEAST (*Browser Exploit Against SSL/TLS*, 2011, violation of the original purpose/constraint of cookies) : 91,5 %
 - CRIME (*Compression Ratio Info-leak Made Easy*, 2012) : 3,2 %

Vulnerabilities in SSL/TLS

- ▶ Vulnerabilities are continuously found in SSL/TLS
 - example : DROWN, revealed on March 1st, 2016
- ▶ Patches exist, but the servers must be updated
 - insecure renegotiation (2009, MITM) : 1,8 + 0,6 % vulnerable servers
 - BEAST (*Browser Exploit Against SSL/TLS*, 2011, violation of the original purpose/constraint of cookies) : 91,5 %
 - CRIME (*Compression Ratio Info-leak Made Easy*, 2012) : 3,2 %
 - protocol downgrade (force using older version SSL 3.0) : 28,0 %

Vulnerabilities in SSL/TLS

- ▶ **Vulnerabilities** are continuously found in SSL/TLS
 - example : **DROWN**, revealed on March 1st, 2016
- ▶ Patches exist, but the servers **must be updated**
 - **insecure renegotiation** (2009, MITM) : **1,8 + 0,6 %** vulnerable servers
 - **BEAST** (*Browser Exploit Against SSL/TLS*, 2011, violation of the original purpose/constraint of cookies) : **91,5 %**
 - **CRIME** (*Compression Ratio Info-leak Made Easy*, 2012) : **3,2 %**
 - **protocol downgrade** (force using older version SSL 3.0) : **28,0 %**
 - **attacks on RC4** (2013) : **8,5 + 34,8 %**

Vulnerabilities in SSL/TLS

- ▶ Vulnerabilities are continuously found in SSL/TLS
 - example : DROWN, revealed on March 1st, 2016
- ▶ Patches exist, but the servers must be updated
 - insecure renegotiation (2009, MITM) : 1,8 + 0,6 % vulnerable servers
 - BEAST (*Browser Exploit Against SSL/TLS*, 2011, violation of the original purpose/constraint of cookies) : 91,5 %
 - CRIME (*Compression Ratio Info-leak Made Easy*, 2012) : 3,2 %
 - protocol downgrade (force using older version SSL 3.0) : 28,0 %
 - attacks on RC4 (2013) : 8,5 + 34,8 %
 - POODLE on TLS (2014) : 3,0 %

Vulnerabilities in SSL/TLS

- ▶ Vulnerabilities are continuously found in SSL/TLS
 - example : DROWN, revealed on March 1st, 2016
- ▶ Patches exist, but the servers must be updated
 - insecure renegotiation (2009, MITM) : 1,8 + 0,6 % vulnerable servers
 - BEAST (*Browser Exploit Against SSL/TLS*, 2011, violation of the original purpose/constraint of cookies) : 91,5 %
 - CRIME (*Compression Ratio Info-leak Made Easy*, 2012) : 3,2 %
 - protocol downgrade (force using older version SSL 3.0) : 28,0 %
 - attacks on RC4 (2013) : 8,5 + 34,8 %
 - POODLE on TLS (2014) : 3,0 %
 - Heartbleed (2014, arbitrary memory access (on the processor/cache) in OpenSSL) : 0,3 %

Vulnerabilities in SSL/TLS

- ▶ **Vulnerabilities** are continuously found in SSL/TLS
 - example : **DROWN**, revealed on March 1st, 2016
- ▶ Patches exist, but the servers **must be updated**
 - **insecure renegotiation** (2009, MITM) : **1,8** + **0,6 %** vulnerable servers
 - **BEAST** (*Browser Exploit Against SSL/TLS*, 2011, violation of the original purpose/constraint of cookies) : **91,5 %**
 - **CRIME** (*Compression Ratio Info-leak Made Easy*, 2012) : **3,2 %**
 - **protocol downgrade** (force using older version SSL 3.0) : **28,0 %**
 - **attacks on RC4** (2013) : **8,5** + **34,8 %**
 - **POODLE on TLS** (2014) : **3,0 %**
 - **Heartbleed** (2014, arbitrary memory access (on the processor/cache) in OpenSSL) : **0,3 %**
 - **no forward secrecy** : **21,2** + **29,4 %**

Vulnerabilities in SSL/TLS

- ▶ Vulnerabilities are continuously found in SSL/TLS
 - example : DROWN, revealed on March 1st, 2016
- ▶ Patches exist, but the servers must be updated
 - insecure renegotiation (2009, MITM) : 1,8 + 0,6 % vulnerable servers
 - BEAST (*Browser Exploit Against SSL/TLS*, 2011, violation of the original purpose/constraint of cookies) : 91,5 %
 - CRIME (*Compression Ratio Info-leak Made Easy*, 2012) : 3,2 %
 - protocol downgrade (force using older version SSL 3.0) : 28,0 %
 - attacks on RC4 (2013) : 8,5 + 34,8 %
 - POODLE on TLS (2014) : 3,0 %
 - Heartbleed (2014, arbitrary memory access (on the processor/cache) in OpenSSL) : 0,3 %
 - no forward secrecy : 21,2 + 29,4 %
 - Too small keys : 0,1 % (public key), 6,9 + 25,2 % (key exchange)

Vulnerabilities in SSL/TLS

- ▶ **Vulnerabilities** are continuously found in SSL/TLS
 - example : **DROWN**, revealed on March 1st, 2016
- ▶ Patches exist, but the servers **must be updated**
 - **insecure renegotiation** (2009, MITM) : **1,8** + **0,6 %** vulnerable servers
 - **BEAST** (*Browser Exploit Against SSL/TLS*, 2011, violation of the original purpose/constraint of cookies) : **91,5 %**
 - **CRIME** (*Compression Ratio Info-leak Made Easy*, 2012) : **3,2 %**
 - **protocol downgrade** (force using older version SSL 3.0) : **28,0 %**
 - **attacks on RC4** (2013) : **8,5** + **34,8 %**
 - **POODLE on TLS** (2014) : **3,0 %**
 - **Heartbleed** (2014, arbitrary memory access (on the processor/cache) in OpenSSL) : **0,3 %**
 - **no forward secrecy** : **21,2** + **29,4 %**
 - **Too small keys** : **0,1 %** (public key), **6,9** + **25,2 %** (key exchange)
 - etc.

Troisième partie

TLS 1.3

(Slides Alfred Menezes, cryptography101.ca)

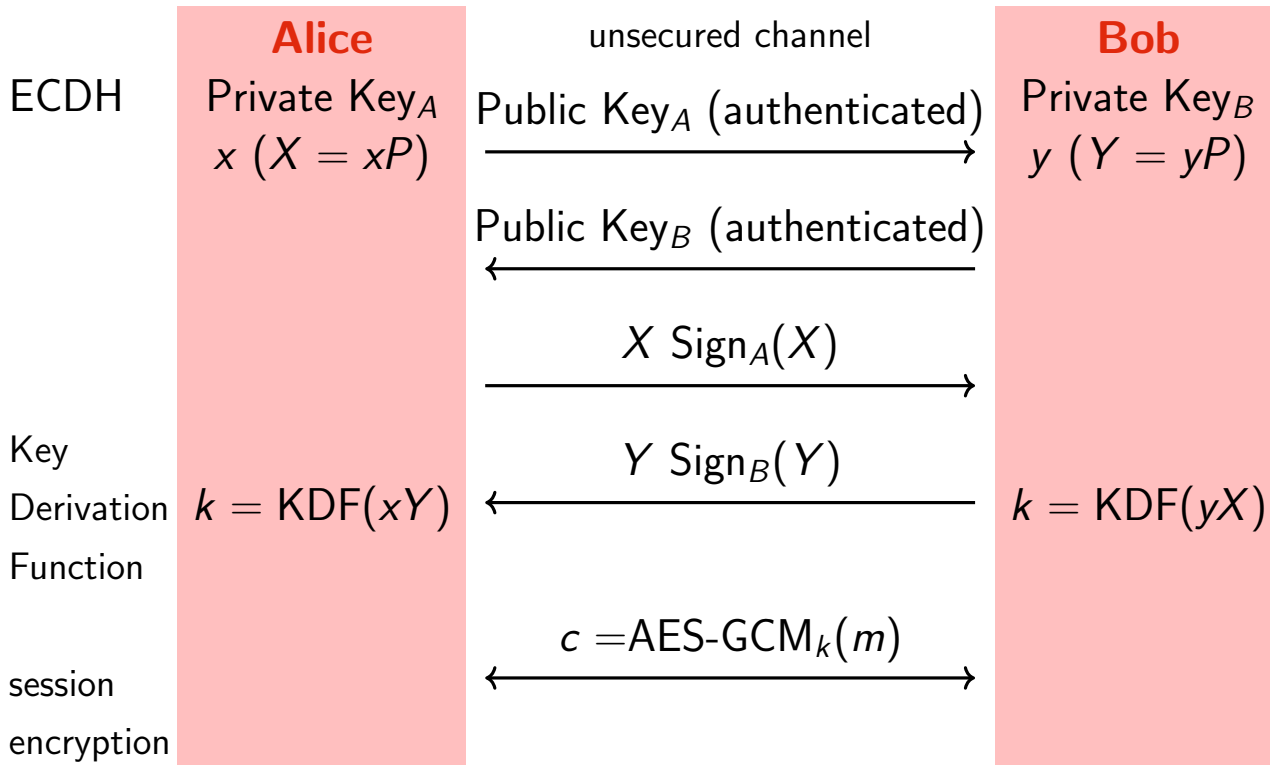
Last TLS update : TLS 1.3

Course materials : Lectures (videos) of Alfred Menezes, University of Waterloo, ON, Canada ($\approx 100\text{km}$ from Toronto)

<https://cryptography101.ca> Section Real-World Deployments

1. TLS 1.3
2. Handshake protocol
3. Record protocol
4. Forward secrecy
5. Public key management
6. CA breaches

Set-up a secure channel from an unsecured one



Transport Layer Security (TLS)

- ▶ **SSL** (Secure Sockets Layer) was designed in 1995 by Netscape (1994–2008, US, acquired by AOL in 1999).
- ▶ **TLS** is an IETF (Internet Engineering Task Force) version of SSL.
- ▶ **SSL/TLS** is used by web browsers to protect web traffic.
- ▶ There are many versions of SSL/TLS :
 - SSL 2.0 (1995)
 - SSL 3.0 (1996) [almost identical to SSL 2.0]
 - TLS 1.0 (1999)
 - TLS 1.1 (2006)
 - **TLS 1.2** (2008)
 - **TLS 1.3** (2018)
- ▶ Google removed SSL 3.0 support from Chrome only in 2014.
- ▶ Amazon AWS began **requiring** TLS 1.2+ in June 2023.

TLS 1.3

- ▶ **TLS 1.3**, approved in August 2018, was a major overhaul of TLS 1.2.
- ▶ Some of the changes made in TLS 1.3 were :
 - Removed RC4, Triple-DES, CBC-mode
 - Removed MAC-then-encrypt
 - Removed MD5 and SHA-1
 - Removed RSA key transport
 - Mandates AES-GCM (and optionally ChaCha20 + Poly1305)
 - All public-key exchanges use DH or ECDH
 - Elliptic curves include P-256, Curve25519, and P-384

TLS components

The main components of TLS are :

1. **Handshake protocol** : Allows the server to authenticate itself to the client and negotiate cryptographic keys.
2. **Record layer protocol** : Used to encrypt and authenticate all transmitted data for the remainder of the session.

TLS 1.3 handshake protocol (simplified)

1. **Phase 1** : The client sends to the server a list of supported protocol versions and supported algorithms, a client nonce, and a one-time ECDH public key $X = xP$.
2. **Phase 2** : The server selects a **one-time ECDH public key** $Y = yP$, computes the master secret $k = \text{KDF}(yX)$, and derives four keys $k_{sh}, k_{ch}, k_{sr}, k_{cr}$ from k .

The server sends to the client a server nonce, Y , the server certificate chain, the server's signature on the handshake transcript up to this point, and the **server-finished MAC** (using key k_{sh}) on the handshake transcript up to this point.

TLS 1.3 handshake protocol (simplified)

1. **Phase 3** : The client uses the certificate chain to verify the authenticity of the server's public key, and uses the public key to verify the server's signature on the handshake transcript. The client then computes the master secret $k = \text{KDF}(xY)$, derives four keys $k_{sh}, k_{ch}, k_{sr}, k_{cr}$ from k , verifies the server-finished MAC (using key k_{sh}), and sends to the server the **client-finished MAC** (using key k_{ch}) on the handshake transcript up to this point.
2. **Phase 4** : The server verifies the client-finished MAC (using key k_{ch}).

TLS 1.3 record layer protocol

Data exchanged for the remainder of the session is encrypted and authenticated using AES-GCM or ChaCha20-Poly1305.

- ▶ The client uses key k_{cr} to encrypt/authenticate its data.
- ▶ The server uses key k_{sr} to encrypt/authenticate its data.

RSA key transport or ECDH ?

RSA key transport (used in TLS 1.2) : A session key k is selected by the client and encrypted with the server's RSA public key ; call the resulting ciphertext c . The session key k is then used to encrypt and authenticate all data exchanged for the remainder of the session (e.g. using AES-GCM).

- ▶ **Drawback** : **Forward secrecy** is not provided.
- ▶ That is, suppose that an eavesdropper saves a copy of c and the encrypted data.
If, at a future point in time, the eavesdropper is able to break into the server and learn its RSA private key, then the eavesdropper is able to decrypt c thereby obtaining k ; she can then decrypt the data that was encrypted with k .
 - Alternatively, a law enforcement agent could demand that the server hand over its RSA private key.
- ▶ **In contrast, forward secrecy can be provided if ECDH is used to establish k .**

ECDH and forward secrecy

When a client (Alice) visits a server (Bob) :

1. Alice selects $x \in_R [1, n - 1]$ and sends $X = xP$ to Bob.
2. Bob selects $y \in_R [1, n - 1]$, signs $Y = yP$ with its RSA signing key, and sends Y and the signature to Alice.
3. Alice verifies the signature using Bob's RSA public key.
4. Both Alice and Bob compute the master secret $k = \text{KDF}(xyP)$.
5. Alice deletes x and Bob deletes y .
6. Alice and Bob use (keys derived from) k to authenticate/encrypt data with AES-GCM.
7. At the end of the session, Alice and Bob delete k (and all derived session keys)

Note : Forward secrecy is provided.

Google started using ECDH for forward secrecy in 2011 ; see

tinyurl.com/GoogleECDH.

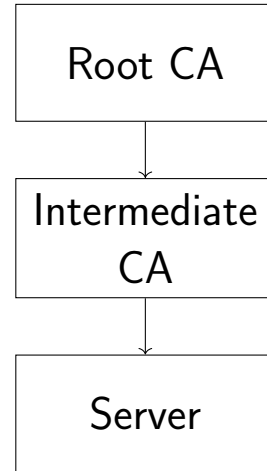
Performance

Timings using Amazon's cryptography library (AWS-libcrypto 2022) running on a c5.2xlarge EC2 instance :

Algorithm	# Operations per second
ECDH P-256	17,625
RSA 2048 enc/verify	53,960
RSA 2048 dec/sign	1,741
AES128-GCM (16 kbytes plaintext)	331,031

The TLS PKI

- ▶ **Root CA** public keys are pre-installed in browsers.
- ▶ Root CAs certify public keys of **intermediate CAs**.
- ▶ **Web servers** get their public keys certified by Intermediate CAs (perhaps for a fee).
 - **GlobalSign** : Web server certificate business. See globalsign.com.
 - **Let's Encrypt** : Non-profit CA that provides free web certificates (360+ million). Automatic certificate issuance; relies on domain validation. See letsencrypt.org.
- ▶ **Clients** (users) can obtain their own certificates.
 - However, most users do not have their own certificates.
 - If clients do not have certificates, then authentication is only one-way, i.e., the server authenticates itself to the client.



Example of a TLS certificate : <https://bbc.com>

Subject Name	
Country	GB
State/Province	London
Locality	London
Organization	BRITISH BROADCASTING CORPORATION
Common Name	www.bbc.com

Issuer name	
Country	BE
Organization	GlobalSign nv-sa
Common Name	GlobalSign RSA OV SSL CA 2018

Validity	
Not Before	Wed, 26 Jun 2024 08 :32 :02 GMT
Not After	Sat, 19 Jul 2025 06 :26 :04 GMT

Miscellaneous	
Serial Number	62:C3:48:11:A4:08:3C:C5:62:DF:7E:91
Signature Algorithm	SHA-256 with RSA Encryption
Version	3 RSA PKCS #1 v1.5 signature

Example of a TLS certificate : <https://bbc.com>

Public Key Info

Algorithm RSA RSA PKCS #1 v1.5 signature

Key Size 2048

Exponent 65537 $e = 2^{16} + 1$

Modulus 9D:8F:4A:99:78:7C:01:C6:73:03:D5:54:41:39:9E:E2:BC:69:D2:D0:75:56:C1:FF:
19:15:C2:66:72:65:70:0D:49:73:F1:64:BF:EA:8E:9C:F7:7A:B0:4A:13:09:8E:65:
1A:A4:54:5C:CD:F4:17:AD:65:59:B1:E9:81:9A:6C:41:37:52:C9:30:E8:9C:8B:C2:
6D:43:EC:E8:D2:6F:28:AE:97:AD:00:60:F8:E2:7D:1E:21:B7:B5:70:A5:4E:99:8A:
2E:C3:68:3C:BB:B2:D6:C6:BA:D2:D0:CB:37:AF:55:FB:92:D5:06:83:13:51:13:D2:
9E:94:5B:87:FC:07:7C:F0:5A:B7:E1:5E:EA:A6:0D:F5:1A:7B:CF:99:60:24:5B:69:
B3:E7:84:08:C9:29:58:3C:F4:9F:6F:19:F2:64:DB:CA:3C:C0:2D:12:CA:AE:35:AA:
DF:1F:6F:22:BF:92:1F:86:35:3E:3E:24:3E:8A:AD:EC:6A:FF:B6:91:1B:2E:A9:62:
AA:26:65:F2:DB:7E:B4:64:7A:89:6F:A1:F5:11:D8:44:8B:3E:68:C9:86:3B:46:C8:
3D:04:65:13:06:38:88:82:97:DC:5D:E2:C5:63:C2:55:62:AD:2C:7B:42:9D:D3:52:
28:BC:CD:5E:E9:40:A3:94:E6:69:F2:81:14:62:39:47

Fingerprints

SHA-256 4A:F5:61:F0:09:06:C5:89:3C:CF:D1:5C:87:C3:27:1A:33:32:98:4D:4C:D1:8F:3A:
72:4B:40:3B:A5:C1:11:AD

SHA-1 58:A0:49:56:FD:02:47:49:94:43:2C:76:ED:C0:7D:A1:F4:1C:94:38

Visiting <https://bbc.com>

Alice : Client (a person's web browser)

Bob : BBC's server.

Key pairs :

1. **BBC's** RSA public key (n_B, e_B) private key d_B . BBC's public key is certified by the intermediate CA GlobalSign-ICA (full name : "GlobalSign RSA OV SSL CA 2018").
2. **GlobalSign-ICA's** RSA public key (n_I, e_I) , private key d_I .
GlobalSign-ICA's public key is certified by the root CA GlobalSign.
3. **GlobalSign's** RSA public key (n_G, e_G) , private key d_G . GlobalSign's RSA public key is certified by itself (i.e., it is **self-signed**), and is preinstalled in all browsers.

Visiting <https://bbc.com> (Handshake 1)

When Alice visits bbc.com, BBC sends Alice the following :

1. **BBC's certificate** which contains BBC's name, RSA public key (n_B, e_B) , etc., and GlobalSign-ICA's RSA signature s_1 on this data.
2. **GlobalSign-ICA's certificate** which contains GlobalSign-ICA's name, RSA public key (n_I, e_I) , etc., and GlobalSign's RSA signature s_2 on this data.
3. **GlobalSign's certificate** which contains GlobalSign's name, RSA public key (n_G, e_G) , etc., and GlobalSign's RSA signature s_3 on this data.
4. A randomly selected point $Y = yP$ on the elliptic curve Curve25519, and BBC's RSA signature s_4 on this point.

Visiting <https://bbc.com> (Handshake 2)

Upon receiving the three certificates and (Y, s_4) , Alice does :

1. **Verify** GlobalSign's signature s_3 using GlobalSign's public key (n_G, e_G) (which is embedded in Alice's browser).
2. **Verify** GlobalSign's signature s_2 using GlobalSign's public key (n_G, e_G) , thereby authenticating GlobalSign-ICA's public key (n_I, e_I) .
3. **Verify** GlobalSign-ICA's signature s_1 using GlobalSign-ICA's public key, thereby authenticating BBC's public key (n_B, e_B) .
4. **Verify** BBC signature s_4 using BBC's public key, thereby authenticating Y .
5. Compute the **master secret** $k = \text{KDF}(xY)$, where (x, X) is the elliptic-curve key pair she had selected at the beginning of the protocol run.

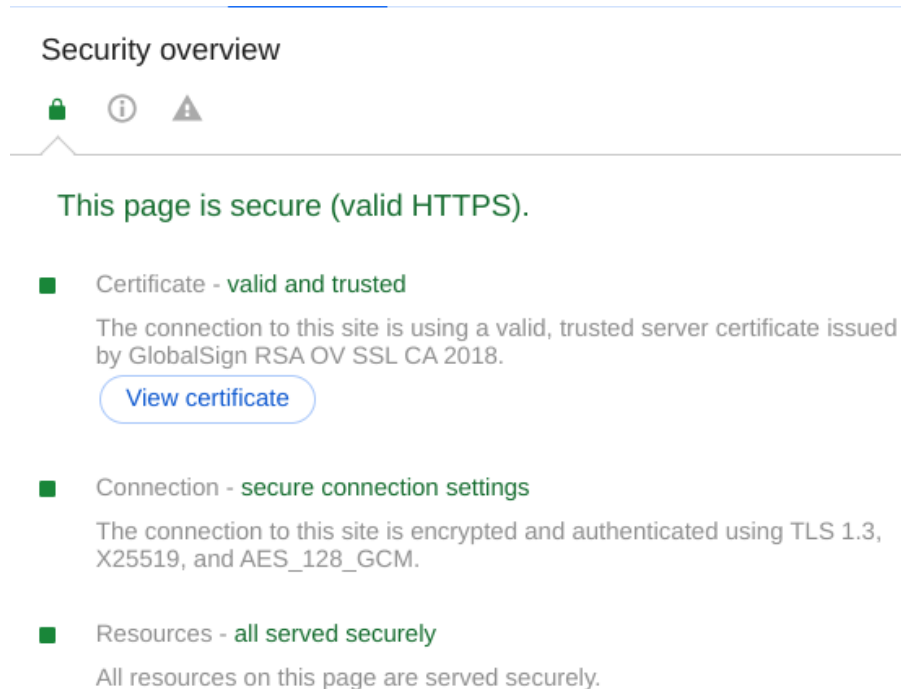
Visiting <https://bbc.com> (Handshake 3)

BBC receives X and computes the **master secret** $k = \text{KDF}(yX)$.

Alice and BBC now share the master secret key k , from which session keys are derived to encrypt and authenticate (with **AES-GCM**) all data exchanged for the remainder of the session.

Secure connection

Chrome browser : : → More tools → Developer tools → Security



GlobalSign

Alice and BBC are relying on GlobalSign to do its job well, which includes :

- ▶ Carefully verifying the identify of the certificate holder (BBC's name and url) and not issue certificates to imposters.
- ▶ Carefully guard its private keys d_G and d_I from disclosure.
- ▶ Carefully guard misuse of its private keys d_G and d_I by company employees.

To engender confidence and trust in its certification practice, Global publishes its **Certification Practices Statement** (CPS ; see <https://tinyurl.com/GlobalSignCPS>) and has its security practices audited by external parties

Can all CAs be trusted ?

Even though BBC might have full confidence in GlobalSign, there remains the possibility of another Root CA or intermediate CA issuing fraudulent certificates for BBC's domain.

- ▶ There are several hundred root CA public keys in a browser, and thousands of Intermediate CAs. Dangers include :
 - Mistakenly-issued, maliciously-issued, maliciously-requested certificates.
- ▶ **CAB** (CA/Browser Forum) : Voluntary consortium of CAs and browser vendors that prepares and maintains industry guidelines for certificate issuance and management. See cabforum.org.
- ▶ **Common CA Database** (managed by Mozilla) : a repository of information about CAs whose root and intermediate certificates are included in browsers. See ccadb.org.

DigiNotar

- ▶ DigiNotar was a Netherlands-based CA, whose root CA public key was embedded in all browsers. DigiNotar issued TLS certificates as well as certificates for government agencies in the Netherlands.
- ▶ In July 2011, DigiNotar issued several hundred fraudulent certificates for domains including `aol.com`, `microsoft.com`, and `*.google.com`.
- ▶ Shortly after, $\approx 300,000$ gmail users in Iran were redirected to websites that looked like gmail's site.
 1. The fraudulent web site would appear to be valid and secured to the gmail users (and the users would establish a secret key with the fraudulent web site using TLS).
 2. Presumably, the fraudulent web site would then capture the gmail userids and passwords of the users. See tinyurl.com/SlateDigiNotar.
- ▶ It is suspected that the DigiNotar attack was mounted by an individual or organization in Iran, but this has not been proven.

Other CA breaches

- ▶ In 2014, an intermediate CA NIC (National Informatics Centre, India) issued unauthorized certificates for several Google domains. NIC was trusted by the Indian Controller of Certifying Authorities. See tinyurl.com/NICGoogle.
- ▶ In 2015, Google discovered that the root CA CNNIC (China Internet Network Information Center) had issued an intermediate CA certificate to an Egyptian company MCS Holdings, which in turn issued fraudulent certificates for several Google domains. See tinyurl.com/GoogleCNNIC.
- ▶ In 2016, it was discovered that the Chinese root CA WoSign had issued fraudulent certificates for several domains including GitHub and Alibaba. See tinyurl.com/GoogleWoSign.

Dealing with CA breaches

- ▶ In response to the CA breaches, DigiNotar, NIC, CNNIC and WoSign's root CA keys were removed from the list of trusted root CA keys in browsers.
- ▶ Root CA's had to meet certain requirements to qualify for **Extended Validation** (EV) status.
- ▶ Google's **Certificate Transparency** :
 - A certificate **logging mechanism** to allow anyone to check which certificates a CA has issued, when, and for which domains.
 - **Auditors** monitor Certificate Logs to watch for malicious behaviour.
 - **Domain name owners** monitor the logs to check for certificate issued for their domains.
 - Development and deployment is ongoing.
 - See certificate.transparency.dev.

Quatrième partie

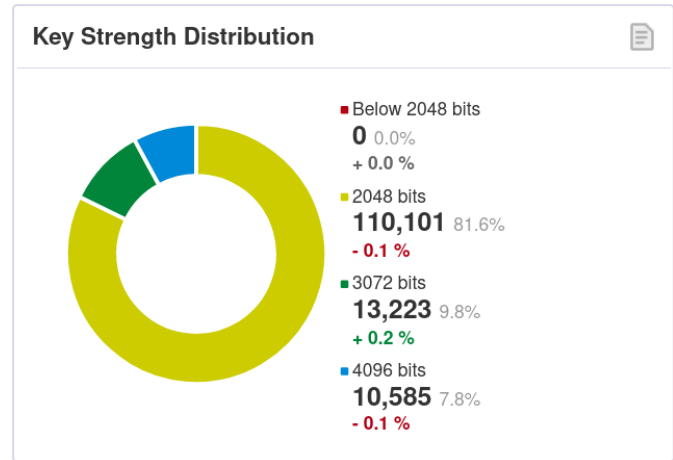
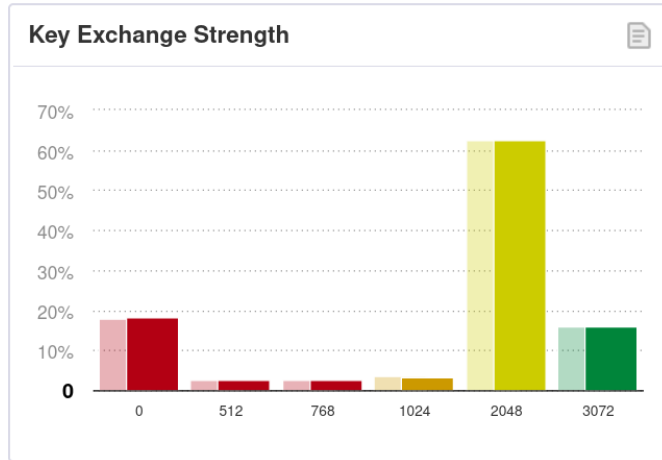
Weak Diffie–Hellman and the LogJam attack

Examples

<https://www.lemonde.fr/>, [https](https://www.lemonde.fr/), security information →
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256, 128 bits, TLS 1.2



<https://www.ssllabs.com/ssl-pulse/>



Weak Diffie–Hellman and the Logjam attack (2015)

<https://weakdh.org/>

- ▶ inspired by the **FREAK** attack
- ▶ Active TLS MITM (Malicious Intruder in The Middle) downgrade attack
- ▶ Force use of DHE_EXPORT cipher suite (Diffie–Hellman Ephemeral key-exchange) with 512-bit prime p
- ▶ precomputation of a huge database of the discrete logarithms of a **factor basis** so as to get a **targeted individual discrete log** in live

TLS 1.2 Handshake reference and tutorial :

<https://datatracker.ietf.org/doc/html/rfc5246>

<https://tlseminar.github.io/first-few-milliseconds/>

What makes the attack possible ?

MITM

- No signature of the cipher suite chosen (DHE vs DHE_EXPORT)
- The attacker can intercept the communication
- the attacker convinces the server that the browser wants DHE_EXPORT
- the attacker answers back DHE and fools the browser with the server's DHE_EXPORT 512-bit prime p
- the attack works because the browser does not check the key sizes (512 bits) of the server's parameters
- real-time discrete log computation to hack the MACs in the Finished messages

Diffie-Hellman key exchange

Alice

Bob

Diffie-Hellman key exchange

Alice

Bob

$(\mathbf{G}, \cdot), g, r = \# \mathbf{G}$ public parameters $(\mathbf{G}, \cdot), g, r = \# \mathbf{G}$

Diffie-Hellman key exchange

Alice

Bob

$(\mathbf{G}, \cdot), g, r = \#\mathbf{G}$ public parameters $(\mathbf{G}, \cdot), g, r = \#\mathbf{G}$

secret key $sk_A = a \leftarrow (\mathbb{Z}/r\mathbb{Z})^*$

public value $PK_A = g^a$

secret key $sk_B = b \leftarrow (\mathbb{Z}/r\mathbb{Z})^*$

public value $PK_B = g^b$

Diffie-Hellman key exchange

Alice

Bob

$(\mathbf{G}, \cdot), g, r = \#\mathbf{G}$ public parameters $(\mathbf{G}, \cdot), g, r = \#\mathbf{G}$

secret key $\text{sk}_A = a \leftarrow (\mathbb{Z}/r\mathbb{Z})^*$

public value $\text{PK}_A = g^a$

secret key $\text{sk}_B = b \leftarrow (\mathbb{Z}/r\mathbb{Z})^*$

public value $\text{PK}_B = g^b$



Diffie-Hellman key exchange

Alice

Bob

$(\mathbf{G}, \cdot), g, r = \#\mathbf{G}$ public parameters $(\mathbf{G}, \cdot), g, r = \#\mathbf{G}$

secret key $sk_A = a \leftarrow (\mathbb{Z}/r\mathbb{Z})^*$

public value $PK_A = g^a$

secret key $sk_B = b \leftarrow (\mathbb{Z}/r\mathbb{Z})^*$

public value $PK_B = g^b$

$\xleftrightarrow{PK_B}$
 $\xleftarrow{PK_A}$

gets Bob's public key PK_B

$sk = PK_B^a = g^{ab}$

gets Alice's public key PK_A

$sk = PK_A^b = g^{ab}$

Diffie-Hellman key exchange

Alice

Bob

$(\mathbf{G}, \cdot), g, r = \#\mathbf{G}$ public parameters $(\mathbf{G}, \cdot), g, r = \#\mathbf{G}$

secret key $sk_A = a \leftarrow (\mathbb{Z}/r\mathbb{Z})^*$

public value $PK_A = g^a$

secret key $sk_B = b \leftarrow (\mathbb{Z}/r\mathbb{Z})^*$

public value $PK_B = g^b$



gets Bob's public key PK_B

$$sk = PK_B^a = g^{ab}$$

gets Alice's public key PK_A

$$sk = PK_A^b = g^{ab}$$

it works because $(g^a)^b = (g^b)^a = g^{ab}$

Regular Diffie–Hellman Ephemeral key-exchange in TLS 1.2

Client

Server

ClientHello = {supported cipher suites}, client random nonce *cr*



Regular Diffie–Hellman Ephemeral key-exchange in TLS 1.2

Client

Server

ClientHello = {supported cipher suites}, client random nonce cr



ServerHello = chosen cipher suite, server random nonce sr



Regular Diffie–Hellman Ephemeral key-exchange in TLS 1.2

Client

Server

ClientHello = {supported cipher suites}, client random nonce cr



ServerHello = chosen cipher suite, server random nonce sr



Certificate = Server public RSA key S , CA signatures chain



Regular Diffie–Hellman Ephemeral key-exchange in TLS 1.2

Client

Server

ClientHello = {supported cipher suites}, client random nonce cr



ServerHello = chosen cipher suite, server random nonce sr



Certificate = Server public RSA key S , CA signatures chain



ServerKeyExchange = $p, g, g^b, \text{Sign}_{\text{RSA key } S}(cr, sr, p, g, g^b)$



Regular Diffie–Hellman Ephemeral key-exchange in TLS 1.2

Client

Server

ClientHello = {supported cipher suites}, client random nonce cr



ServerHello = chosen cipher suite, server random nonce sr



Certificate = Server public RSA key S , CA signatures chain



ServerKeyExchange = $p, g, g^b, \text{Sign}_{\text{RSA key } S}(cr, sr, p, g, g^b)$



ClientKeyExchange = g^a



Regular Diffie–Hellman Ephemeral key-exchange in TLS 1.2

Client

Server

ClientHello = {supported cipher suites}, client random nonce cr

→

ServerHello = chosen cipher suite, server random nonce sr

←

Certificate = Server public RSA key S , CA signatures chain

←

ServerKeyExchange = $p, g, g^b, \text{Sign}_{\text{RSA key } S}(cr, sr, p, g, g^b)$

←

ClientKeyExchange = g^a

→

$\text{KDF}(g^{ab}, cr, sr)$

→ k_{mc}, k_{ms}, k_e

g^{ab} = pre-master secret

key derivation function

$\text{KDF}(g^{ab}, cr, sr)$

→ k_{mc}, k_{ms}, k_e

Regular Diffie–Hellman Ephemeral key-exchange in TLS 1.2

Client

Server

ClientHello = {supported cipher suites}, client random nonce cr

→

ServerHello = chosen cipher suite, server random nonce sr

←

Certificate = Server public RSA key S , CA signatures chain

←

ServerKeyExchange = $p, g, g^b, \text{Sign}_{\text{RSA key } S}(cr, sr, p, g, g^b)$

←

ClientKeyExchange = g^a

→

$\text{KDF}(g^{ab}, cr, sr)$ g^{ab} = pre-master secret $\text{KDF}(g^{ab}, cr, sr)$

→ k_{mc}, k_{ms}, k_e key derivation function → k_{mc}, k_{ms}, k_e

MAC = **M**essage **A**uthentication **C**ode

ClientFinished = $\text{MAC}_{k_{mc}}(\text{Client's view of handshake})$

→

Regular Diffie–Hellman Ephemeral key-exchange in TLS 1.2

Client

Server

ClientHello = {supported cipher suites}, client random nonce cr

→

ServerHello = chosen cipher suite, server random nonce sr

←

Certificate = Server public RSA key S , CA signatures chain

←

ServerKeyExchange = $p, g, g^b, \text{Sign}_{\text{RSA key } S}(cr, sr, p, g, g^b)$

←

ClientKeyExchange = g^a

→

$\text{KDF}(g^{ab}, cr, sr)$ g^{ab} = pre-master secret $\text{KDF}(g^{ab}, cr, sr)$

→ k_{mc}, k_{ms}, k_e key derivation function → k_{mc}, k_{ms}, k_e

MAC = **M**essage **A**uthentication **C**ode

ClientFinished = $\text{MAC}_{k_{mc}}(\text{Client's view of handshake})$

→

ServerFinished = $\text{MAC}_{k_{mc}}(\text{Server's view of handshake})$

←

Regular Diffie–Hellman Ephemeral key-exchange in TLS 1.2

Client

Server

ClientHello = {supported cipher suites}, client random nonce cr

→

ServerHello = chosen cipher suite, server random nonce sr

←

Certificate = Server public RSA key S , CA signatures chain

←

ServerKeyExchange = $p, g, g^b, \text{Sign}_{\text{RSA key } S}(cr, sr, p, g, g^b)$

←

ClientKeyExchange = g^a

→

$\text{KDF}(g^{ab}, cr, sr)$ g^{ab} = pre-master secret $\text{KDF}(g^{ab}, cr, sr)$

→ k_{mc}, k_{ms}, k_e key derivation function → k_{mc}, k_{ms}, k_e

MAC = **M**essage **A**uthentication **C**ode

ClientFinished = $\text{MAC}_{k_{mc}}(\text{Client's view of handshake})$

→

ServerFinished = $\text{MAC}_{k_{mc}}(\text{Server's view of handshake})$

←

Verify Server's MAC

Verify Client's MAC

MITM DHE attack

Client

Attacker

Server

MITM DHE attack

Client

Attacker

Server

ClientHello = { ...DHE... }, *cr*




MITM DHE attack

Client

Attacker

Server

ClientHello = {...DHE...}, *cr*



[DHE_EXPORT], *cr*



MITM DHE attack

Client

Attacker

Server

ClientHello = {...DHE...}, cr



[DHE_EXPORT], cr



[DHE_EXPORT], sr

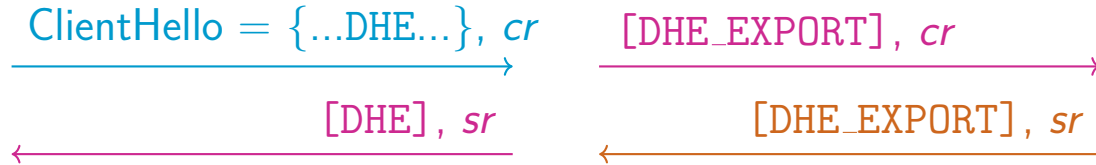


MITM DHE attack

Client

Attacker

Server

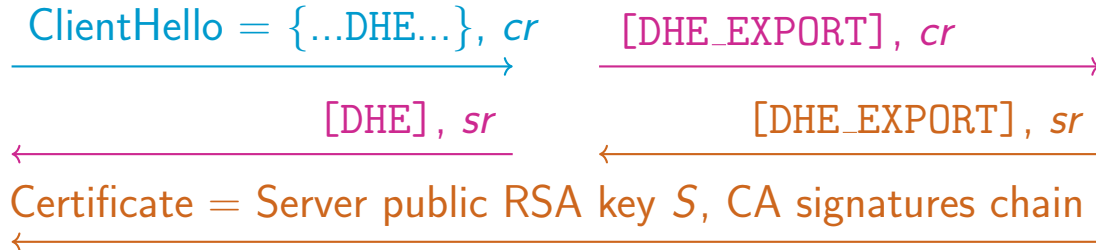


MITM DHE attack

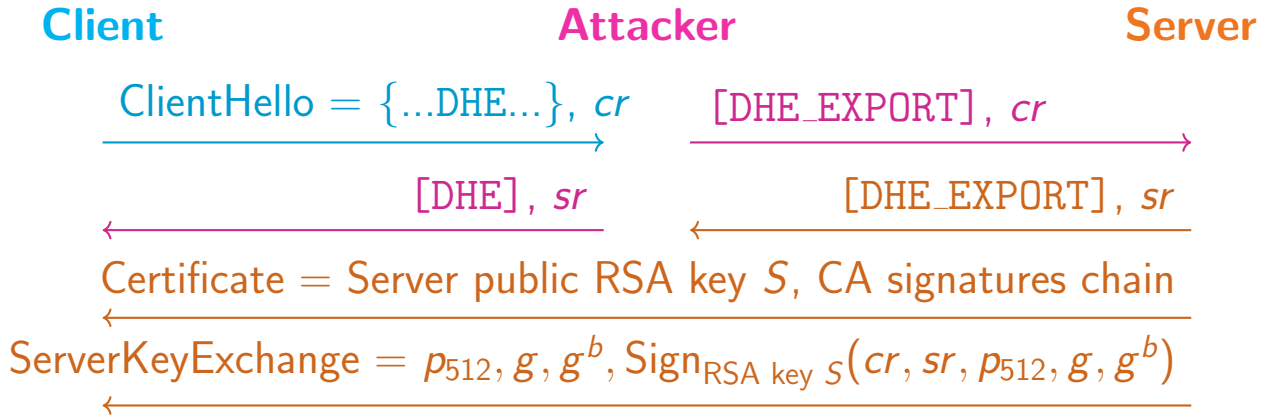
Client

Attacker

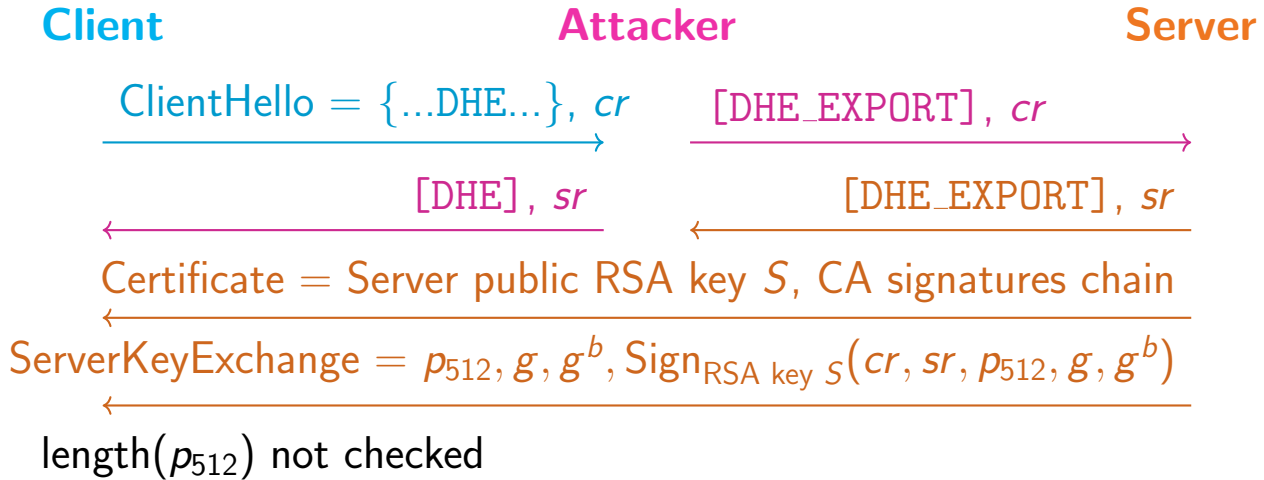
Server



MITM DHE attack



MITM DHE attack



MITM DHE attack

Client

Attacker

Server

ClientHello = $\{\dots\text{DHE}\dots\}$, cr

[DHE_EXPORT], cr

[DHE], sr

[DHE_EXPORT], sr

Certificate = Server public RSA key S , CA signatures chain

ServerKeyExchange = $p_{512}, g, g^b, \text{Sign}_{\text{RSA key } S}(cr, sr, p_{512}, g, g^b)$

length(p_{512}) not checked

ClientKeyExchange = g^a

MITM DHE attack

Client

Attacker

Server

ClientHello = {...DHE...}, cr

[DHE_EXPORT], cr

[DHE], sr

[DHE_EXPORT], sr

Certificate = Server public RSA key S , CA signatures chain

ServerKeyExchange = $p_{512}, g, g^b, \text{Sign}_{\text{RSA key } S}(cr, sr, p_{512}, g, g^b)$

length(p_{512}) not checked

ClientKeyExchange = g^a

KDF(g^{ab}, cr, sr)

→ k_{mc}, k_{ms}, k_e

MITM DHE attack

Client

Attacker

Server

ClientHello = {...DHE...}, cr

[DHE_EXPORT], cr

[DHE], sr

[DHE_EXPORT], sr

Certificate = Server public RSA key S , CA signatures chain

ServerKeyExchange = $p_{512}, g, g^b, \text{Sign}_{\text{RSA key } S}(cr, sr, p_{512}, g, g^b)$

length(p_{512}) not checked

ClientKeyExchange = g^a

$\text{KDF}(g^{ab}, cr, sr)$

$\rightarrow k_{mc}, k_{ms}, k_e$

$\text{MAC}_{k_{mc}}(\text{DHE...})$

MITM DHE attack

Client

Attacker

Server

ClientHello = {...DHE...}, cr

[DHE_EXPORT], cr

[DHE], sr

[DHE_EXPORT], sr

Certificate = Server public RSA key S , CA signatures chain

ServerKeyExchange = $p_{512}, g, g^b, \text{Sign}_{\text{RSA key } S}(cr, sr, p_{512}, g, g^b)$

length(p_{512}) not checked

ClientKeyExchange = g^a

$\text{KDF}(g^{ab}, cr, sr)$

$\rightarrow k_{mc}, k_{ms}, k_e$

$\text{MAC}_{k_{mc}}(\text{DHE...})$

How to hack the Server's $\text{MAC}_{k_{mc}}$

MITM DHE attack

Client

Attacker

Server

ClientHello = {...DHE...}, cr

[DHE_EXPORT], cr

[DHE], sr

[DHE_EXPORT], sr

Certificate = Server public RSA key S , CA signatures chain

ServerKeyExchange = $p_{512}, g, g^b, \text{Sign}_{\text{RSA key } S}(cr, sr, p_{512}, g, g^b)$

length(p_{512}) not checked

ClientKeyExchange = g^a

$\text{KDF}(g^{ab}, cr, sr)$

$\rightarrow k_{mc}, k_{ms}, k_e$

$\text{MAC}_{k_{mc}}(\text{DHE...})$

How to hack the Server's $\text{MAC}_{k_{mc}}$

Keep Client waiting for ServerFinished

MITM DHE attack

Client

Attacker

Server

ClientHello = {...DHE...}, cr

[DHE_EXPORT], cr

[DHE], sr

[DHE_EXPORT], sr

Certificate = Server public RSA key S , CA signatures chain

ServerKeyExchange = $p_{512}, g, g^b, \text{Sign}_{\text{RSA key } S}(cr, sr, p_{512}, g, g^b)$

length(p_{512}) not checked

ClientKeyExchange = g^a

$\text{KDF}(g^{ab}, cr, sr)$

$\rightarrow k_{mc}, k_{ms}, k_e$

$\text{MAC}_{k_{mc}}(\text{DHE...})$

How to hack the Server's $\text{MAC}_{k_{mc}}$

Keep Client waiting for ServerFinished

Solve $\log_g(g^b) = b$, get g^{ab} ,

$\text{KDF}(g^{ab}, cr, sr) \rightarrow k_{mc}, k_{ms}, k_e$

MITM DHE attack

Client

Attacker

Server

ClientHello = {...DHE...}, cr

[DHE_EXPORT], cr

[DHE], sr

[DHE_EXPORT], sr

Certificate = Server public RSA key S , CA signatures chain

ServerKeyExchange = $p_{512}, g, g^b, \text{Sign}_{\text{RSA key } S}(cr, sr, p_{512}, g, g^b)$

length(p_{512}) not checked

ClientKeyExchange = g^a

$\text{KDF}(g^{ab}, cr, sr)$

$\rightarrow k_{mc}, k_{ms}, k_e$

$\text{MAC}_{k_{mc}}(\text{DHE...})$

$\text{MAC}_{k_{mc}}(\text{DHE...})$

How to hack the Server's $\text{MAC}_{k_{mc}}$

Keep Client waiting for ServerFinished

Solve $\log_g(g^b) = b$, get g^{ab} ,

$\text{KDF}(g^{ab}, cr, sr) \rightarrow k_{mc}, k_{ms}, k_e$

MITM DHE attack

