MODERN CRYPTOGRAPHY FOR INFORMATION SECURITY
MSc Computer Security

cryptoing3@gmail.com

January 2019 — Asymmetric Cryptography

Rappel du Plan

Diffie-Hellman (D-H)

Diffie-Hellman (D-H) key exchange is historically the first asymmetric algorithm. It's a cryptographic protocol that allows two parties that have no prior knowledge of each other to jointly establish a *shared secret key* over an insecure communications channel. This key can then be used to encrypt subsequent communications using a symmetric key cipher.
Synonyms of Diffie-Hellman key exchange include :

- Diffie-Hellman key agreement
- Diffie-Hellman key establishment
- Diffie-Hellman key negotiation
- Exponential key exchange

Cryptography for security : remainder

Key Agreement versus Key Transportation ?

- Diffie-Hellman Key Agreement : it is a computation, two or more parties can agree on a key in such a (quite symmetic) way that both influence the outcome (wiki), widley used in SSL/TLS.
- Diffie-Hellman Key Transportation : a non symmetric way, it is a one way negociation !
  - Alice sends to Bernard : $C = E_{RSA_{Bernard}}(K_{Alice})$
  - Bernard can decipher : $D_{RSA_{Bernard}}(C) = K_{Alice}$

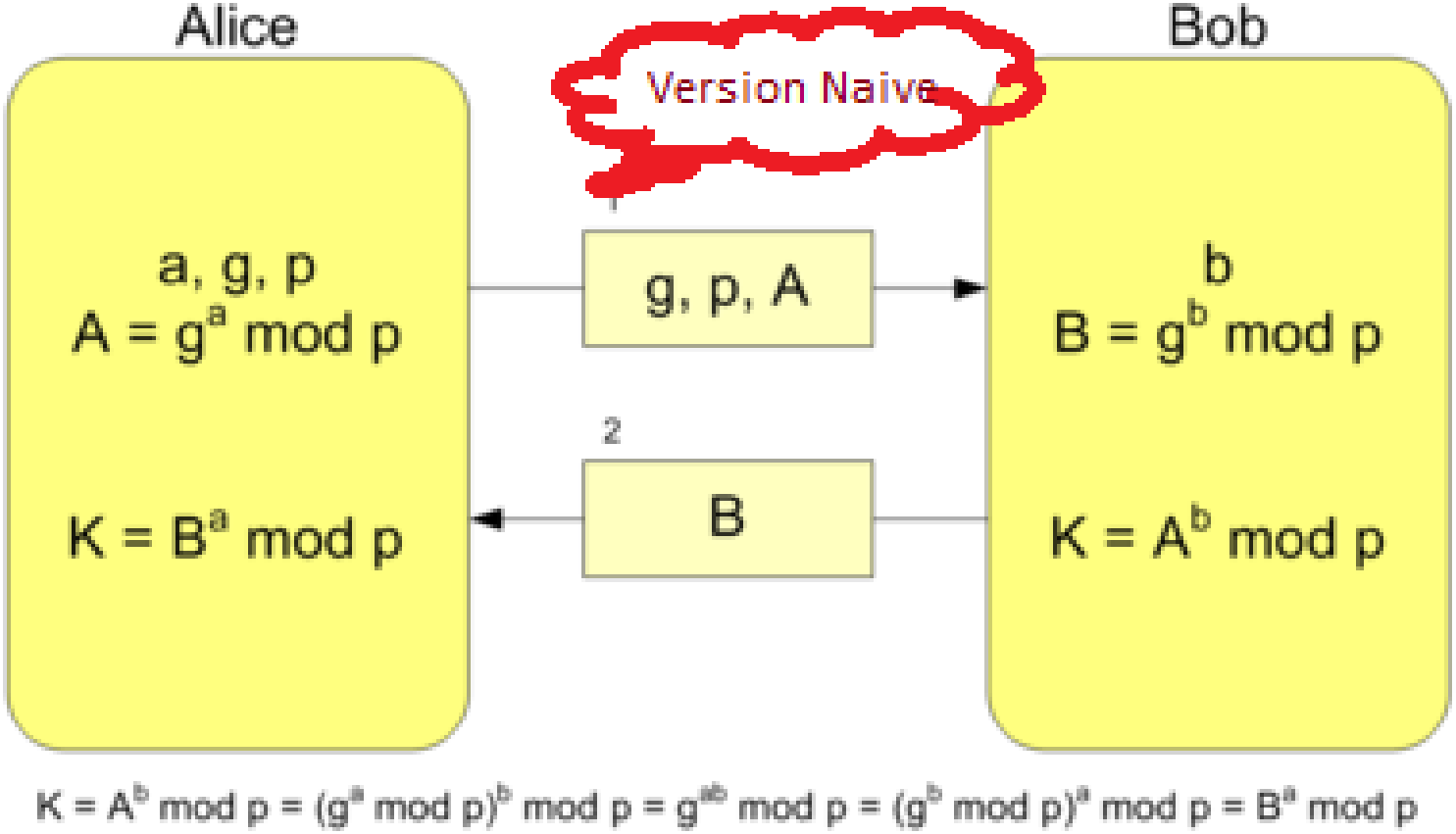
  Can be used in PGP/GnuPG.

Cryptography for security : remainder

Diffie-Hellman (D-H)

The scheme was first published publicly by Whitfield Diffie and Martin Hellman in 1976, although it later emerged that it had been invented a few years earlier within GCHQ, the British signals intelligence agency, by Malcolm J. Williamson but was kept classified. In 2002, Hellman suggested the algorithm be called Diffie-Hellman-Merkle key exchange in recognition of Ralph Merkle's contribution to the invention of public-key cryptography (Hellman, 2002).
Although Diffie-Hellman key agreement itself is an anonymous (non-authenticated) key-agreement protocol, it provides the basis for a variety of authenticated protocols, and is used to provide perfect forward secrecy in Transport Layer Security's (TLS/SSL) ephemeral modes.

Diffie-Hellman (D-H)



$K = A^b \bmod p = (g^a \bmod p)^b \bmod p = g^{ab} \bmod p = (g^b \bmod p)^a \bmod p = B^a \bmod p$

Diffie-Hellman (D-H)

The simplest, and original, implementation of the protocol uses :

1. $\mathbb{Z}_p$: the Multiplicative group of integers modulo p, with p a large prime
2. g : a primitive root mod p. This means $\{1, g, g^2, g^3 \ldots, g^{p-2}\} = \mathbb{Z}_p$
3. **Remark** : $x \in_{\mathcal{R}} \mathbb{Z}_p^*$ means : x is *randomly* chosen in $\mathbb{Z}_p^*$ but generally we don't explain exactly **how** we do it !

$$K_{Alice\&Bob} = g^{a \times b} \bmod p = (g^a)^b \bmod p = (g^b)^a \bmod p$$

Cryptography for security : remainder

Algorithm 1 :

Summary : A and B each send the other one message over an open channel.

Result : shared secret K known to both parties A and B.

Begin

1. **One-time setup** : An appropriate prime p and generator g of $\mathbb{Z}_p^*$ are selected and published.
2. **Protocol actions** : Perform the following steps each time a shared key is required.
   — Al. chooses a random secret $a \in_{\mathcal{R}} \mathbb{Z}_p^*$ and sends B message (1)
   — Bob chooses a random secret $b \in_{\mathcal{R}} \mathbb{Z}_p^*$ and sends B message (2)
   — Bob receives $g^a \bmod p$ and computes the shared key as
   $K = (g^a)^b \bmod p = g^{a.b} \bmod p$
   — Al. receives $g^b \bmod p$ and computes the shared key as
   $K = (g^b)^a \bmod p = g^{a.b} \bmod p$
3. **Protocol messages** : Perform the following steps each time a shared key is required.
   Al. $\rightarrow$ Bob : $A = g^a \bmod p$ (1)
   Bob $\rightarrow$ Al. : $B = g^b \bmod p$ (2)

End.

Security of the Diffie-Hellman

- The protocol is considered *secure* against eavesdroppers if G (the group) and g are chosen properly. (Again : it's supposed thet DLP is *computationally difficult*.)
- In the case where $G = \mathbb{Z}_p^*$ this means p prime *large enough* and $p - 1$ with at least a large factor.
- The eavesdropper ("Eve") must solve the Diffie-Hellman problem (DHP) to obtain g^{ab}.
- The DHP is posed as follows :

  Given an element g and the values of g^x mod p and g^y mod p, what is the value of g^{xy} ?
- DHP is currently considered as computationally difficult.
- An efficient algorithm to solve the DLP would make it easy to compute a or b and solve the DHP, making this and many other public key cryptosystems insecure.

Cryptography for security : remainder

Man in the middle attack (wiki)

In the original description, the DH exchange by itself does not provide authentication of the communicating parties and is thus vulnerable to a man-in-the-middle attack. A person in the middle may establish two distinct DH key exchanges, one with Alice and the other with Bob, effectively masquerading as Alice to Bob, and vice versa, allowing the attacker to decrypt (and read or store) then re-encrypt the messages passed between them. A method to authenticate the communicating parties to each other is generally needed to prevent this type of attack.
A variety of cryptographic authentication solutions incorporate a DH exchange. When Alice and Bob have a public key infrastructure, they may digitally sign the agreed key, or $g^a \bmod p$ and $g^b \bmod p$, as in MQV, STS and the IKE component of the IPsec protocol suite for securing Internet Protocol communications.
When Alice and Bob share a password, they may use a password-authenticated key agreement form of DH.

Rappel du Plan

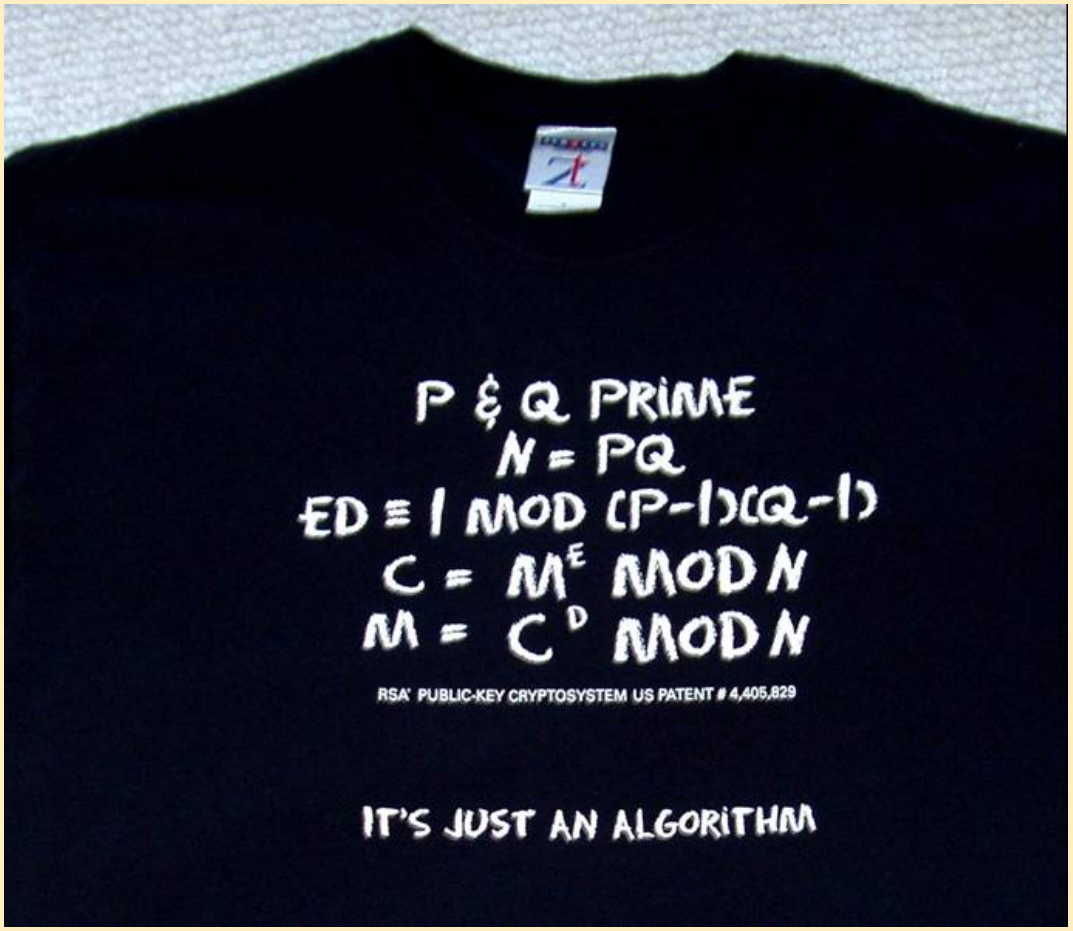
Cryptography for security : remainder

RSA : Rivest Shamir Adleman

- RSA is an algorithm for public-key cryptography. It was the first algorithm known to be suitable for signing as well as encryption, and one of the first great advances in public key cryptography. RSA is widely used in electronic commerce protocols, and is believed to be secure given sufficiently long keys and the use of up-to-date implementations.
- The algorithm was publicly described in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman at MIT.
- *Clifford Cocks, a British mathematician working for the UK intelligence agency GCHQ, described an equivalent system in an internal document in 1973, but given the relatively expensive computers needed to implement it at the time, it was mostly considered a curiosity and, as far as is publicly known, was never deployed. His discovery, however, was not revealed until 1997 due to its top-secret classification.*

Cryptography for security : remainder

a "“joke"” : RSA on the shirt/gift september 17th 2000 (end of the patent RSA (1978)) par *RSA Data Security* !



Computing the keys

RSA involves a public and private key. The public key can be known to everyone and is used for encrypting messages. Messages encrypted with the public key can only be decrypted using the private key.
The keys for the RSA algorithm are generated the following way :

1. Choose two different large random prime numbers p and q.
2. Calculate $n = p \times q$; n is used as the modulus for both the public and private keys
3. Calculate the Euler totient : $\varphi(n) = (p-1)(q-1)$.
4. Choose an integer e such that $1 < e < \varphi(n)$, and e is coprime to $\varphi(n)$, i.e. : e and $\varphi(n)$ share no factors other than 1 : $gcd(e, \varphi(n)) = 1$.
5. Compute (with the Extended Euclidean Algorithm) d to satisfy the congruence relation $d\,e \equiv 1 \bmod \varphi(n)$, *i.e.*, for some integer k :

$$d\,e = 1 + k\varphi(n).$$

Cryptography for security : remainder

Remarks

- e is released as the public key exponent.
- d is kept as the private key exponent
- A popular choice for the public exponents is $e = 2^{16} + 1 = 65537$. Some applications choose smaller values such as $e = 3, 5, 17$ or 257 instead. This is done to make encryption and signature verification faster on small devices like smart cards but small public exponents may lead to greater security risks.
- Steps 4 and 5 can be performed with the extended Euclidean algorithm (see Bezout's Algorithm in French).
- The public key consists of the modulus n and the public (or encryption) exponent e.
- The private key consists of the modulus n and the private (or decryption) exponent d which must be kept secret.

Encrypting messages

- Alice transmits her public key (n, e) to Bob and keeps the private key secret.
- Bob then wishes to send message M to Alice.
- He first turns M into a number $m < n$ by using an agreed-upon reversible protocol known as a padding scheme. He then computes the ciphertext c corresponding to :

$$c = m^e \bmod n$$

- This can be done quickly using the method of exponentiation by squaring. Bob then transmits c to Alice.

Decrypting messages

- Alice can recover m from c by using her private key exponent d by the following computation :

$$m = c^d \bmod n.$$

- Given m she can recover the original message M.
- Why ? Because of the Euler theorem :

$$c^d \equiv (m^e)^d \equiv m^{e\,d} \bmod n = m^1 \bmod n = m.$$

$$e\,d \equiv 1 \bmod (p-1)(q-1).$$

Rappel du Plan

Cryptography for security : RSA Resume

Encryption with RSA ?

$$C = M^e \bmod n$$

Decryption with RSA ?

$$M = C^d \bmod n$$

Signature with RSA ?

Signing :

$$S(M) = M^d \bmod n$$

Verifying :

$$S(M)^e = (M^d)^e \bmod n = M^{e\,d} \bmod n = M \bmod n$$

A curiosity

The NIST Special Publication on Computer Security (SP 800-78 Rev 1 of August 2007) does not allow public exponents e smaller than 65537, but does not state a clear reason for this restriction.

Rappel du Plan

Padding schemes

Used in practice, in the real wordld, RSA is generally combined with some *padding scheme*. The goal of the padding scheme is to prevent a number of attacks that potentially work against RSA without padding :

- When encrypting with low encryption exponents (e.g., e = 3) and small values of the message m, (i.e. $m < N^{1/e}$) the result of m^e mod N is strictly less than the modulus n. In this case, ciphertexts can be *easily* decrypted by taking the eth root of the ciphertext over the integers.
- Other attacks : If the same clear text message is sent to e or more recipients in an encrypted way, and the receiver's shares the same exponent e, but different p, q, and n, then it is easy to decrypt the original clear text message via the Chinese remainder theorem. Johan Hastad noticed that this attack is possible even if the cleartexts are not equal, but the attacker knows a linear relation between them. This attack was later improved by Don Coppersmith and is often used for CTF challenges.

Remarks (from *Cryptography* by Wikipedians)

1. Because RSA encryption is a deterministic encryption algorithm — i.e., has no random component — an attacker can successfully launch a chosen plaintext attack against the cryptosystem, by encrypting likely plaintexts under the public key and test if they are equal to the ciphertext.
2. A cryptosystem is called *semantically secure* if an attacker cannot distinguish two encryptions from each other even if the attacker knows (or has chosen) the corresponding plaintexts.
3. As described above, RSA without padding is not semantically secure.

Remarks (from *Cryptography* by Wikipedians)

4. An RSA property : product of two ciphertexts = encryption of the product of the respective plaintexts. That is $m_1^e m_2^e \equiv (m_1 m_2)^e \bmod N$. Because of this *multiplicative property* a chosen-ciphertext attack is possible.
5. An attacker, who wants to know the decryption of $c = m^e \bmod n$ may ask the holder of the secret key to decrypt an unsuspicious-looking ciphertext $c' = (c\,r)^e \bmod N$ for some value r chosen by him/she. So, we have c' is the encryption of $(m\,r) \bmod N$.
5. Hence, if the attacker is successful with the attack, he will learn $(m\,r) \bmod N$ from which he can derive the message m by multiplying $m\,r$ with $r^{-1} \bmod N$ the modular inverse of r modulo N.

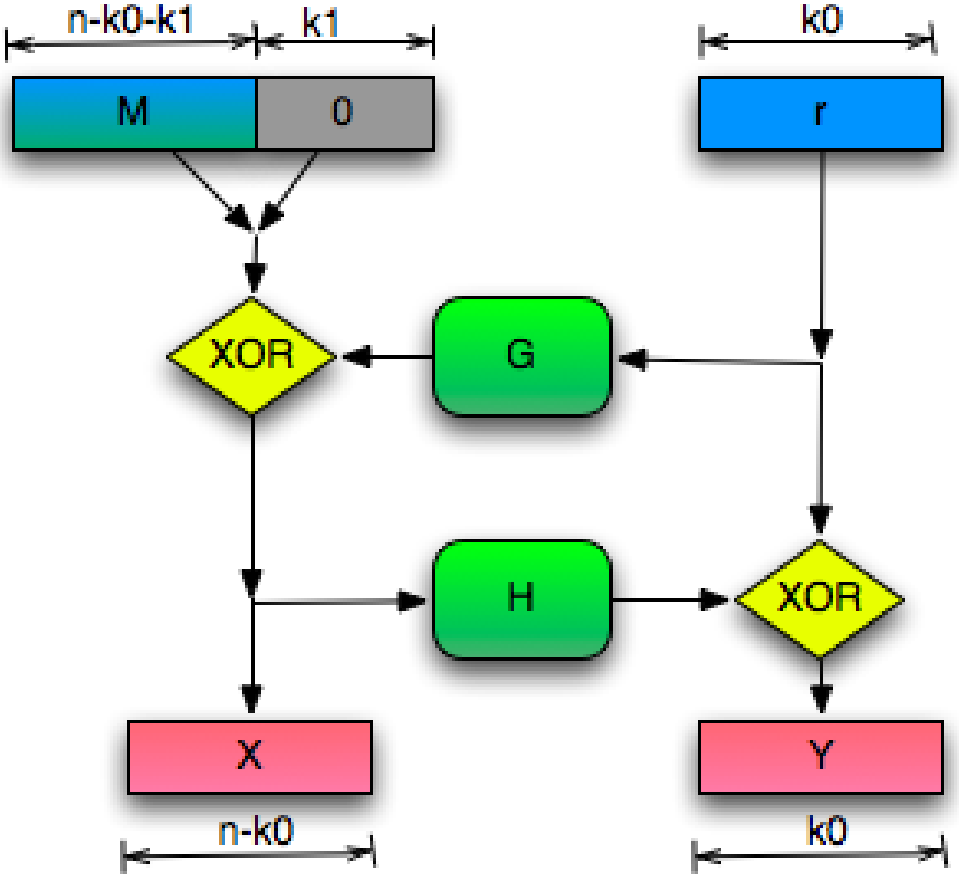
Randomized Padding : why ?

- To avoid these problems, practical RSA implementations typically embed some form of structured, *randomized padding* into the value m before encrypting it.
- This padding ensures that m does not fall into the range of insecure plaintexts, and that a given message, once padded, will encrypt to one of a large number of different possible ciphertexts.

Randomized Padding Scheme (PS) : why ?

- PKCS : designed to securely pad messages prior to RSA encryption. Because these schemes pad the plaintext m with some number of additional bits, the size of the message M encrypted must be somewhat smaller.
- RSA PS must be carefully designed so as to prevent sophisticated attacks which may be facilitated by a predictable message structure.
- Early versions of the PKCS (i.e. PKCS #1 up to version 1.5) used a construction that turned RSA into a semantically secure encryption scheme (version found vulnerable to a practical adaptive chosen ciphertext attack).
- Later versions of the standard include Optimal Asymmetric Encryption Padding (OAEP) prevents these attacks. PKCS also incorporates processing schemes designed to provide additional security for RSA signatures : Probabilistic Signature Scheme for RSA

OAEP ? . . . means : *Optimal Asymmetric Encryption Padding* from http ://scienceblogs.com/goodmath/2009/01/08/cryptograph c-padding-in-rsa/



OAEP ? Let's start with the pieces [OAE]

1. You have a number, n, length of the RSA modulus.
2. You have two integers, k_0 and k_1, parameters of the protocol.
3. You have the original plaintext message, M. By definition, the length of M is $(n - k_0 - k_1)$. (The protocol is responsible for breaking up large messages into sub-messages.)
4. You have a 0-padded version of M, M', which is M followed by k_1 zeros - so M' has length $n - k_0$.
5. You have a random string of bits, r, which has length k0.
6. You have a cryptographic hash function G, which expands a string of k0 bits to a string of $n - k_0$ bits.
7. You have another cryptographic hash function H, which contracts a string of $n - k_0$ bits to a string of k_0 bits.

OAEP ? . . .

The OAEP padding algorithm is illustrated off to the side.

- You first compute $G(r)$, giving you a string of $n - k_0$ bits.
- You now XOR $G(r)$ with M', producing a string of $n - k_0$ bits, which we'll call X.
- You then compute $H(X)$, and XOR that with r, producing a result that we'll call Y.
- You then get the end result of the padding : the concatenation of X and Y : $X|Y$.

OAEP ? . . .

The way to understand this is that you've got some random bits in R, which you're shuffling up (using G and H) and mixing with the bits of the original message. The resulting message is longer, and has a random element mixed into it, which defeats the numerical properties that make some attacks against RSA work. It's easy to compute X and Y given M and r. And given X and Y, if you know G and H it's easy to compute M and r. But given E(X|Y), as an attacker, you're screwed. You can still decrypt the message, and get X and Y, decode them, and get the plaintext. But the process of doing the padding obscures the numerical properties of RSA so that even knowing the padding function, your attacks won't work.

Rappel du Plan

TLS (cf RFC 5246)

1. SSL : Secure Socket Layer
2. SSL : originally developed by Netscape.
3. Netscape's patent buyed in 2001 by l'IETF
4. Now called TLS : *Transport Layer Security*.
5. TLS is a client-server model.
6. TLS 1.3 has now been finalized as of March 21st, 2018.
7. see https ://tools.ietf.org/html/draft-ietf-tls-tls13-28

Transport Secure Layer : objectives

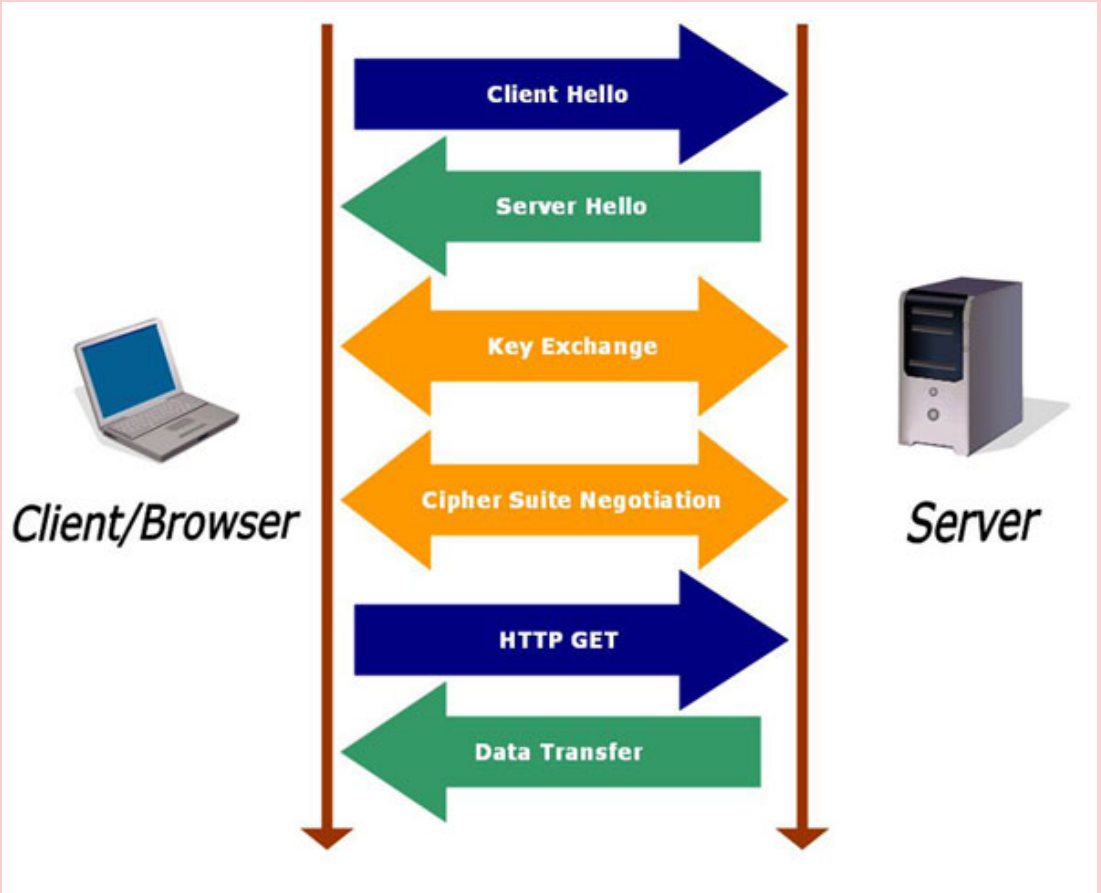
1. Authentification : Who is on the other side ?
   - Server Authentication — required [*via* PKC and possibly with a X.509 certificate]
   - Client authentication — optional
2. Confidentiality of all data exhanged : *via* symmetric cryptography
3. Integrity : messages integrity *via* hash functions
4. Spontaneity : any Client contacts Server (possibly for the first time) without problem
5. Transparency : You do not want to know about security, so you do not have to modify http, you just use it.

SSL (cf RFC 6101) : The SSL protocol provides connection ...

... security that has three basic properties :

1. The connection is **private**. Encryption is used after an initial handshake to define a secret key. Symmetric cryptography is used for data encryption (e.g., DES [obsolete], 3DES , AES, FORTEZZA, RC4).
2. The peer's identity can be **authenticated** using asymmetric, or public key, cryptography (e.g., RSA , DSS).
3. The connection is **reliable**. Message transport includes a message integrity check using a keyed Message Authentication Code (MAC) [RFC2104]. Secure hash functions (e.g., SHA, MD5) are used for MAC computations.

Image ! (cf ssl.trustwave.com)



Rappel du Plan

CipherSuites

1. Suites used to choose algorithms :
   - Key Exchange
   - Authentication
   - Encryption
   - MAC : Message Authentication Code [MAC is a special digest, which incorporate a key into the computation of the digest. The MAC value is dependent on both the message and the key.]
2. Construction : Kx-Auth-Enc-MAC
3. Examples :
   - DHE-RSA-AES128-SHA
   - RSA-RC4-MD5 : RSA is used for Key Exchange and for Server Authentication.

Key Exchange Algorithms

- RSA : The client chooses the secret and encrypts it with the Server RSA PK. THis PK has to be authenticated in a certificate (PKCS#1 v1.5 standard).
- DH_DSS & DH_RSA : fixed Diffie-Hellman with long-term parameters.
- DHE_DSS & DHE_RSA : *Ephemeral* Diffie-Hellman with random parameters (Client & Server).
- DH_Anon : DH_Anon_EXPORT (Ex : TLS_DH_anon_WITH_AES_128_CBC_SHA)/ Particular case where parameters are not authenticated. Of course, vulnerable to MIM.
- ECDHE : Elliptic Curve Diffie-Hellman Exchange.

TLS/SSL Session in Details : Client_Hello

Handshaking - Ciphersuit Negotiation [A d, Vau06]

Client sends a plaintext Client_Hello message and suggests some cryptographic parameters (collectively called ciphersuit) to be used for their communication session. The Client_Hello message also contains a 32-byte random number denoted as client_random. For example,
Client_Hello :

1. Protocol Version : TLSv1 if you can, else SSLv3.
2. Key Exchange : RSA if you can, else Diffe-Hellman.
3. Secret Key Cipher Method : 3DES if you can, else DES.
4. Message Digest : SHA-1 if you can, else MD5.
5. Data Compression Method : PKZip if you can, else gzip.
6. Client Random Number : 32 bytes.

Handshaking - Ciphersuit Negotiation

The stronger method (in terms of security) shall precede the weaker one, e.g. RSA (1024-bit) precedes DH, 3DES precedes DES, SHA-1 (160-bit) precedes MD5 (128-bit). Server responds with a plaintext Server_Hello to state the ciphersuit of choice (server decides on the ciphersuit). The message also contains a 32-bytes random number denoted as server_random. For example, Server_Hello : :

1. Protocol Version : TLSv1.
2. Key Exchange : RSA.
3. Secret Key Cipher Method : DES.
4. Message Digest : SHA-1.
5. Data Compression Method : PKZip.
6. Server Random Number : 32 bytes.

Handshaking - Key Exchange

The server sends its digital certificate to the client, which is supposedly signed by a root CA. The client uses the root CA's public key to verify the server's certificate (trusted root-CAs' public key are pre-installed inside the browser). It then retrieves the server's public key from the server's certificate. (If the server's certificate is signed by a sub-CA, the client has to build a digital certificate chain, leading to a trusted root CA, to verify the server's certificate.)
The server can optionally request for the client's certificate to authenticate the client. In practice, server usually does not authenticate the client. This is because :

1. Server authenticates client by checking the credit card in an e-commerce transaction.
2. Most clients do not have a digital certificate.
3. Authentication via digital certificate takes time and the server may lose an impatient client.

Handshaking - Key Exchange :

HKE : Next step is to establish the Session Key

1. The client generates a 48-byte (384-bit) random number called pre_master_secret, encrypts it using the verified server's public key and sends it to the server.
2. Server decrypts the pre_master_secret using its own private key. Eavesdroppers cannot decrypt the pre_master_secret, as they do not possess the server's private key.
3. Client and server then independently and simultaneously create the session key, based on the pre_master_secret, client_random and server_random. Notice that both the server and client contribute to the session key, through the inclusion of the random number exchange in the hello messages. Eavesdroppers can intercept client_random and server_random as they are sent in plaintext, but cannot decrypt the pre_master_secret.
4. In a SSL/TLS session, the session key consists of 6 secret keys (to thwart crypto-analysis). 3 secret keys are used for client-to-server messages, and the other 3 secret keys are used for server-to-client messages.

Handshaking - Key Exchange : . . .

4 . . . Among the 3 secret keys, one is used for encryption (e.g., DES secret key), one is used for message integrity (e.g., HMAC) and one is used for cipher initialization. (Cipher initialization uses a random plaintext called Initial Vector (IV) to prime the cipher pump.)

5 Client and server use the pre_master_secret (48-byte random number created by the client and exchange securely), client_random, server_random, and a pseudo-random function (PRF) to generate a master_secret. They can use the master_secret, client_random, server_random, and the pseudo-random function (PRF) to generate all the 6 shared secret keys. Once the secret keys are generated, the pre_master_secret is no longer needed and should be deleted.

6 From this point onwards, all the exchanges are encrypted using the session key.

7 The client sends Finished handshake message using their newly created session key. Server responds with a Finished handshake message.

Message Exchange

Client and server can use the agreed-upon session key (consists of 6 secret keys) for secure exchange of messages.

Sending messages :

1. The sender compresses the message using the agreed-upon compression method (e.g., PKZip, gzip).
2. The sender hashes the compressed data and the secret HMAC key to make an HMAC, to assure message integrity.
3. The sender encrypts the compressed data and HMAC using encryption/decryption secret key, to assure message confidentiality.

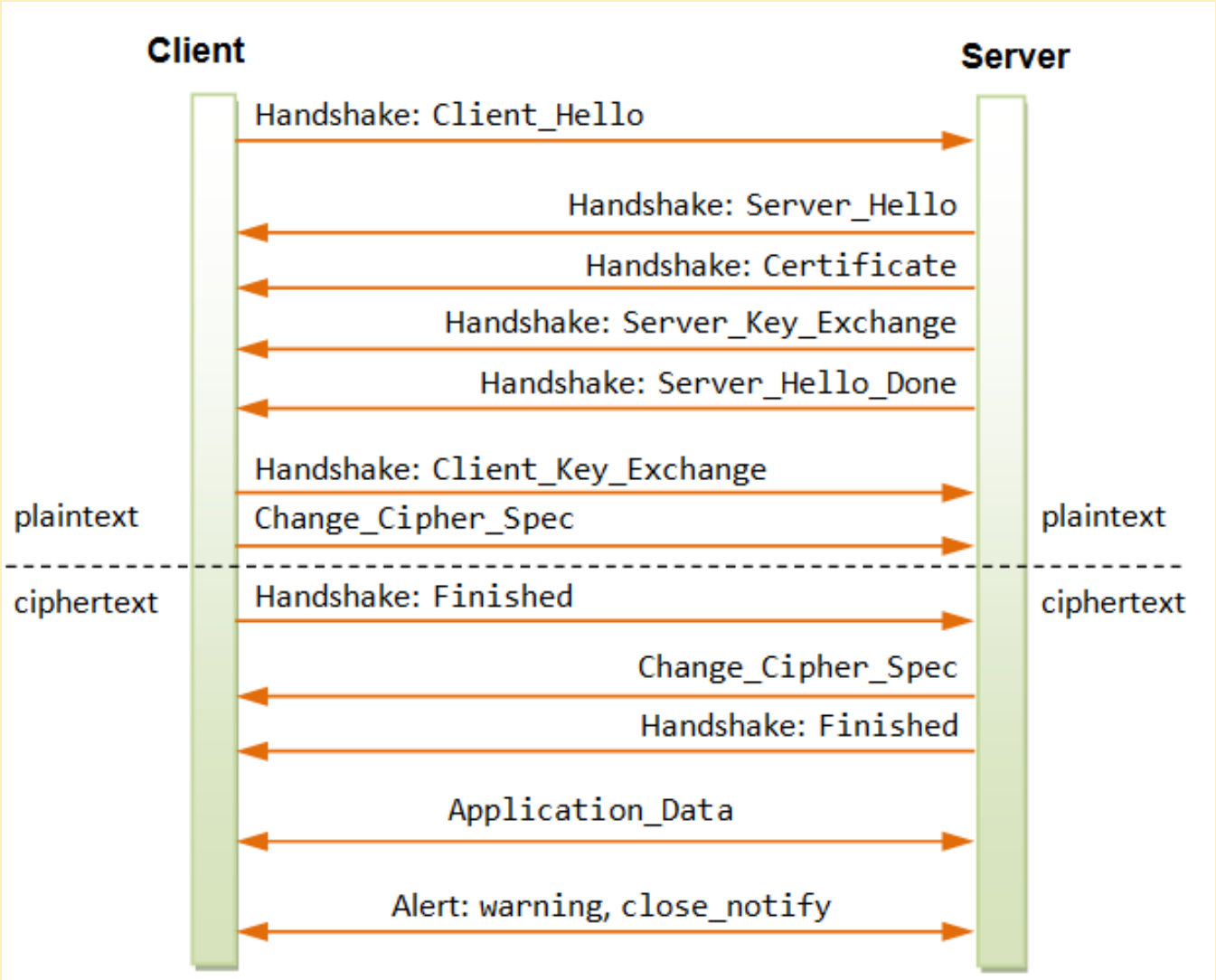
Message Exchange

Retrieve messages :

1. The receiver decrypts the ciphertext using the encryption/decryption secret key to retrieve the compressed data and HMAC.
2. The receiver hashes the compressed data to independently produce the HMAC. It then verifies the generated HMAC with the HMAC contained in the message to assure message integrity.
3. The receiver un-compresses the data using the agreed-upon compression method to recover the plaintext.

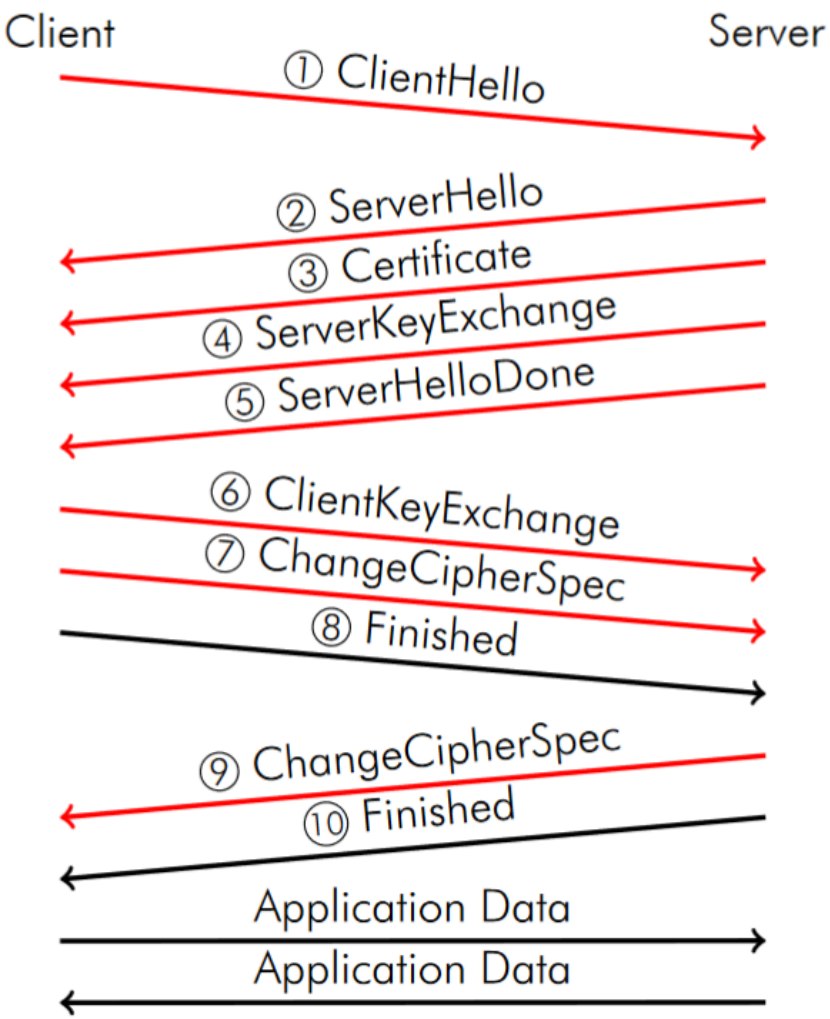
TLS/SSL : a picture

The following diagram shows the sequence of the SSL messages for a typical client/server session.



TLS/SSL : a picture

A picture from ANSSI : Generic initiation of a TLS session [?]



TLS/SSL : a resume

A resume from ANSSI : Generic initiation of a TLS session

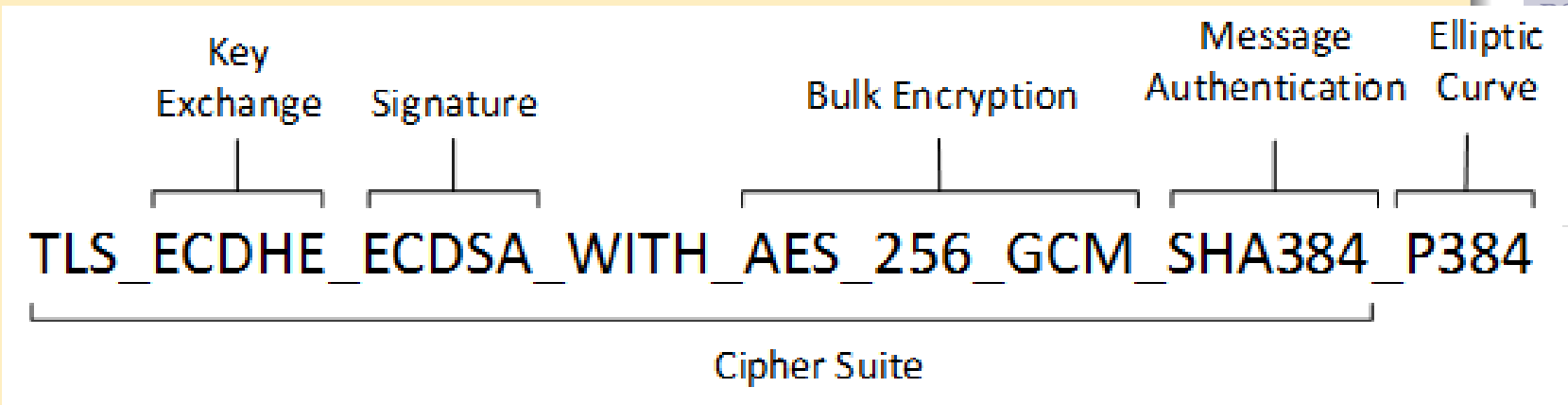
1] The client initiates a query by sending a message of type ClientHello, containing in particular the cipher suites that it supports;
2] The server responds with a ServerHello which contains the retained suite;
3] The server sends a Certificate message which in particular contains its public key within a digital certificate;
4] The server transmits in a ServerKeyExchange an ephemeral value that it signs using the private key associated with the preceding public key;
5] The server manifests that it is on hold with a ServerHelloDone

A resume from ANSSI : Generic initiation of a TLS session

6] After verification of the certificate and authentication of the preceding value, the client in turn chooses an ephemeral value that it encrypts using the public key of the certificate and then transmits in a ClientKeyExchange;
7] The server signals it activates the suite fully with a ChangeCipherSpec;
8] The client sends a Finished, the first message protected according to the cipher suite with the secrets coming from the exchange of ephemeral keys;
9] The server signals the activation of the same suite with a ChangeCipherSpec;
10] The server sends in turn a Finished, its first secure message.

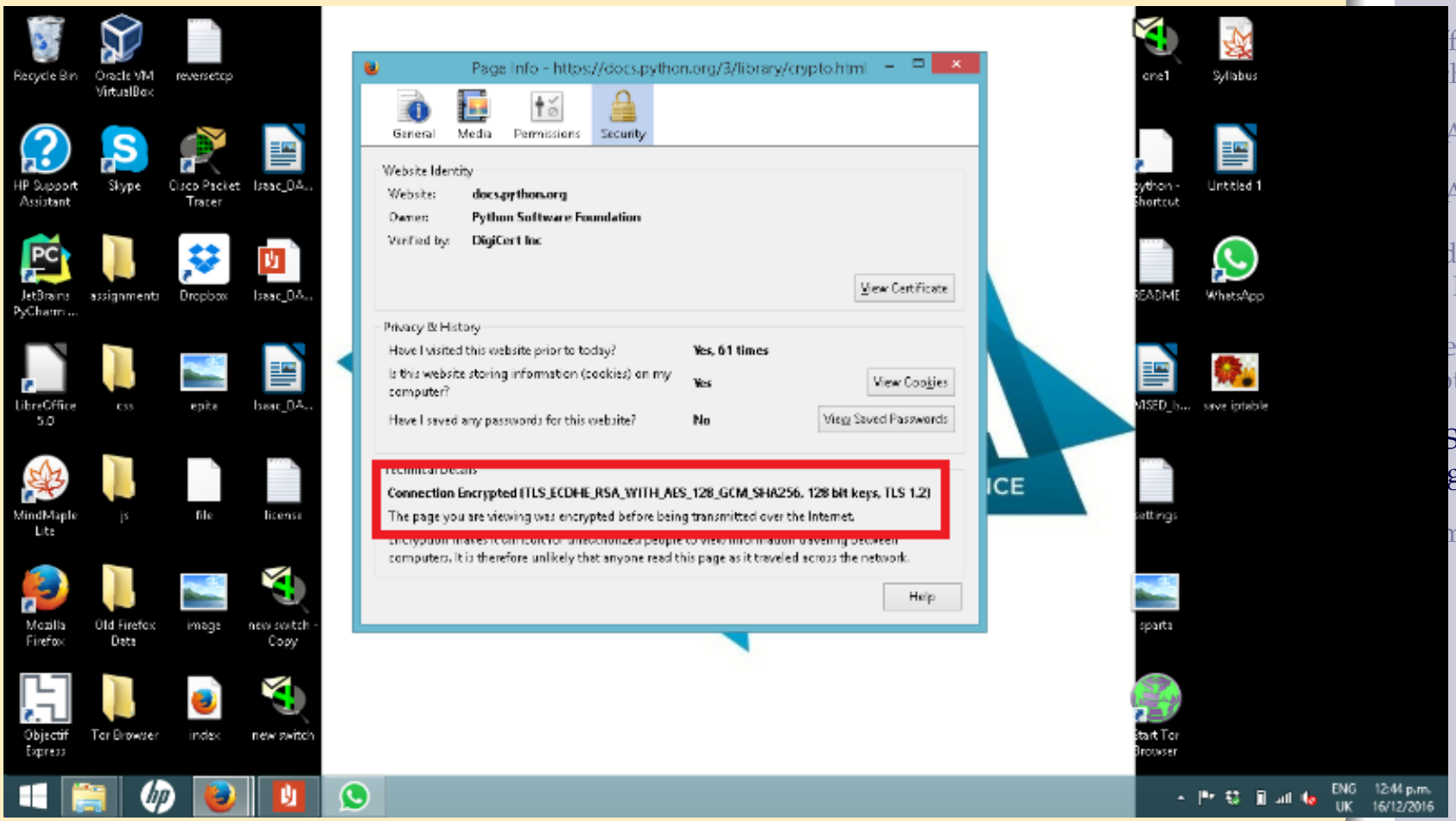
TLS/SSL : a picture

A picture from Microsoft [TLS]



TLS/SSL : a picture

A picture from a student



Rappel du Plan

TLS/SSL : weaknesses ?

Some interesting links

- Timing attack on RSA/OpenSSL : https ://ssrg.nicta.com.au/projects/TS/cachebleed//
- The Drown Attack : https ://www.drownattack.com/
- The ANSSI's work (in French) : see
- http ://www.ssi.gouv.fr/uploads/IMG/pdf/SSL_TLS_etat_des_lieux_e

Timing attack on RSA/OpenSSL

https ://ssrg.nicta.com.au/projects/TS/cachebleed

- CacheBleed : A Timing Attack on OpenSSL Constant Time RSA
- Authors : Yuval Yarom, Daniel Genkin & Nadia Heninger

Timing attack on RSA/OpenSSL

https ://ssrg.nicta.com.au/projects/TS/cachebleed

CacheBleed is a side-channel attack that exploits information leaks through cache-bank conflicts in Intel processors. By detecting cache-bank conflicts via minute timing variations, we are able to recover information about victim processes running on the same machine. Our attack is able to recover both 2048-bit and 4096-bit RSA secret keys from OpenSSL 1.0.2f running on Intel Sandy Bridge processors after observing only 16,000 secret-key operations (decryption, signatures). This is despite the fact that OpenSSL's RSA implementation was carefully designed to be constant time in order to protect against cache-based (and other) side-channel attacks.

Timing attack on RSA/OpenSSL

https ://ssrg.nicta.com.au/projects/TS/cachebleed

While the possibility of an attack based on cache-bank conflicts has long been speculated, this is the first practical demonstration of such an attack. Intel's technical documentation describes cache-bank conflicts as early as 2004. However, these were not widely thought to be exploitable, and as a consequence common cryptographic software developers have not implemented countermeasures to this attack.

- A description of SSL/TLS.
  https ://www3.ntu.edu.sg/home/ehchua/programming/webprogramm
- OAEP : RFC 2437.
  https ://tools.ietf.org/html/rfc2437.
- https ://msdn.microsoft.com/en-us/library/windows/desktop/aa374757(v=vs.85).aspx.
- S. Vaudenay.
  Introduction to classical cryptography.
  Springer, 2006.