Chapitre 16 - Sécurisation des communications

Objectifs :

- ▷ Décrire les principes de chiffrement symétrique et asymétrique (avec clef privée/clef publique).
- ▷ Décrire l'échange d'une clef symétrique en utilisant un protocole asymétrique pour sécuriser une communication HTTPS.

1 Introduction

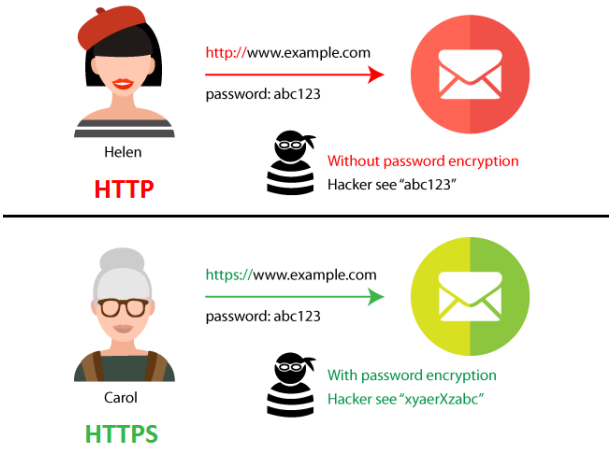
Pour qu'un message entre deux machines dans un réseau informatique ne puisse pas être compréhensible s'il est intercepté, il faut qu'il soit **chiffré**.

A retenir !

Il y a deux manières principales pour **chiffrer un message** :

- ▷ par un **chiffrement symétrique** qui utilise une clé unique, connue seulement de l'émetteur et du récepteur. L'émetteur chiffre le message avec la clé et le récepteur le déchiffre avec la même clé.
- ▷ par un **chiffrement asymétrique** qui utilise un couple de clés, l'une publique connue de tout le monde, l'autre privée connue uniquement par le récepteur. L'émetteur chiffre le message avec la clé publique. Le récepteur le déchiffre avec la clé privée.

Ces deux méthodes de chiffrement sont utilisées lorsqu'un navigateur demande une page web avec le **protocole HTTPS**.



Les caractéristiques que l'on cherche à protéger :

— La **confidentialité** : les données échangées ne peuvent être connues que de l'expéditeur (Anne) et du destinataire (Bob). Pour ce faire, Alice va chiffrer le message et Bob devra le déchiffrer avant de le lire. Toute personne qui ne possède pas le moyen de déchiffrer ce message chiffré se verra dans l'impossibilité de comprendre le contenu du message.

— L'**authentification** : les interlocuteurs sont réellement qui ils prétendent être. Un **certificat numérique** est alors nécessaire, car il permet d'utiliser une signature numérique comme moyen d'authentifier des informations numériques.

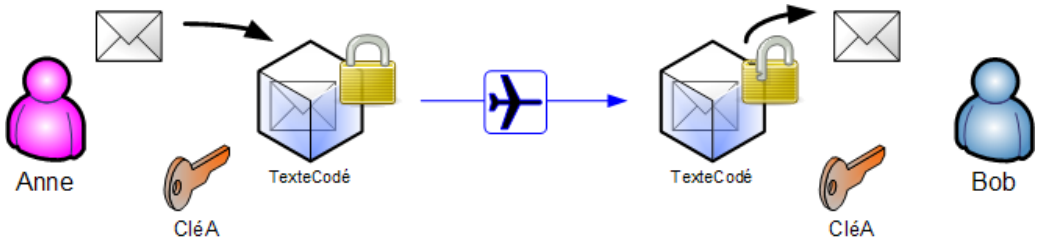
— L'**intégrité** : toute modification (involontaire ou malveillante) est détectée. Dans ce cas, il faut calculer la somme de contrôle (*checksum*, en anglais) ou « empreinte » du message et la comparer avec celle du message original.

2 Le chiffrement symétrique

L'idée de chiffrer des messages (de les rendre illisibles pour des personnes non autorisées) ne date pas du début de l'ère de l'informatique. En effet, dès l'*Antiquité*, on cherchait déjà à sécuriser les communications en chiffrant les messages sensibles. On a des traces de son utilisation par les Égyptiens vers *2000 av. J.-C.*

Nous nous intéresserons ici uniquement aux communications ayant lieu par l'intermédiaire d'un réseau informatique.

Dans un chiffrement symétrique, la clé utilisée par l'expéditeur pour chiffrer le message est la même que celle utilisée par le récepteur pour déchiffrer le message.



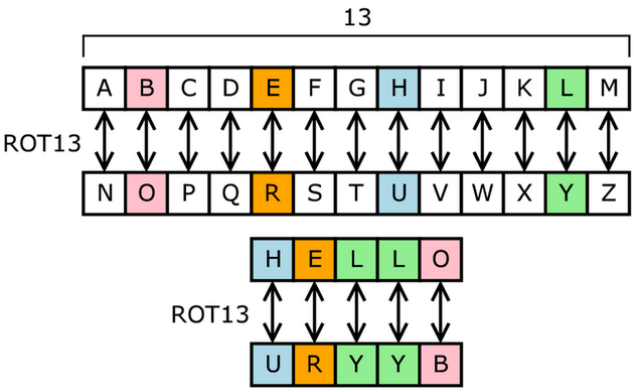
2.1 Le chiffrement de César

Principe : on décale les lettres de n rangs. La clé est tout simplement le **décalage**.

En ne considérant que les lettres en majuscule et non accentuées, notre alphabet se réduit à 26 lettres :

`A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z`

Par exemple, avec un décalage de 13 : (`A` → `N`) et (`N` → `A`).



On utilise bien la même clé pour chiffrer et déchiffrer, il s'agit donc d'un **chiffrement symétrique**.

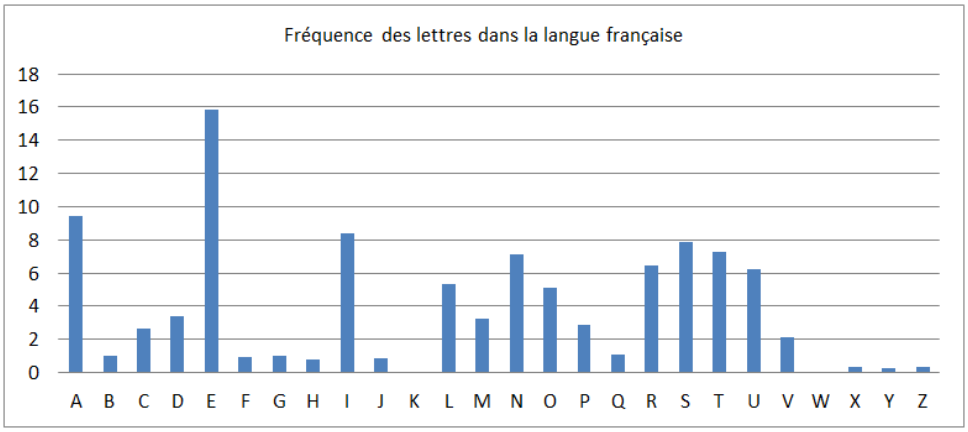
Par exemple, le message `'BONJOUR, COMMENT ALLEZ-VOUS'` se code par : `'OBAWBHE, PBZZRAG NYYRM-IBHF'`

Le déchiffrement nous redonne bien le message d'origine.

Inconvénient :

Cette méthode est intéressante pour vérifier l'**intégrité** et pour les performances mais cela permet à un attaquant de deviner plus facilement la clé.

L'attaque la plus évidente de ce code est d'utiliser la **méthode des fréquences** des caractères.



Par exemple, sur un texte en français suffisamment long, la lettre la plus fréquente (qui revient le plus souvent) du chiffrement correspondra à la lettre `E`. On peut donc retrouver facilement le décalage (la clé).

2.2 Le chiffrement par substitution

Principe : On remplace une lettre par une autre de l'alphabet.

On a donc $26! \approx 4 \times 10^{26}$ clés possibles.

Exemple

Avec la clé suivante :

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
AZERTYUIOPQSDFGHJKLMWXCVBN
```

le message `SUBSTITUTION` devient `LWZLMOMWMOGF`

2.3 Le chiffrement de Vigenère *(XVIème siècle)*

Cette méthode a été mise au point durant la *Renaissance* pour contrer la cryptanalyse par la méthode des fréquences de lettres qui permettait de "casser" les clés de cryptage assez facilement.

On choisit une clé sous la forme d'un mot ou d'une phrase qui donne le décalage à appliquer qui devient alors variable.

Supposons que la clé soit `ABC`, les décalages successifs seront `0`, `1`, `2`, `0`, `1`, `2`, `0`, ..., *etc.*

Exemple

Avec la clé `ABC` :

Le message `SUBSTITUTION` devient `SVDSUKTVVIPP` car :

```
SUBSTITUTION
ABCABCABCABC
------------
SVDSUKTVVIPP
```

2.4 Le code ASCII

Nous pouvons utiliser le code ASCII de chaque caractère et coder notre message en binaire (par exemple, on peut vérifier que le `H` correspond bien à l'octet `01001000`).

Letter	ASCII Code	Binary	Letter	ASCII Code	Binary
a	097	01100001	A	065	01000001
b	098	01100010	B	066	01000010
c	099	01100011	C	067	01000011
d	100	01100100	D	068	01000100
e	101	01100101	E	069	01000101
f	102	01100110	F	070	01000110
g	103	01100111	G	071	01000111
h	104	01101000	H	072	01001000
i	105	01101001	I	073	01001001
j	106	01101010	J	074	01001010
k	107	01101011	K	075	01001011
l	108	01101100	L	076	01001100
m	109	01101101	M	077	01001101
n	110	01101110	N	078	01001110
o	111	01101111	O	079	01001111
p	112	01110000	P	080	01010000
q	113	01110001	Q	081	01010001
r	114	01110010	R	082	01010010
s	115	01110011	S	083	01010011
t	116	01110100	T	084	01010100
u	117	01110101	U	085	01010101
v	118	01110110	V	086	01010110
w	119	01110111	W	087	01010111
x	120	01111000	X	088	01011000
y	121	01111001	Y	089	01011001
z	122	01111010	Z	090	01011010

Pour effectuer la "conversion" texte vers un code binaire ASCII ou vice-versa, on peut utiliser le site :

https://www.rapidtables.com/convert/number/ascii-hex-bin-dec-converter.html

Par exemple le message `Hello World!` se code par :

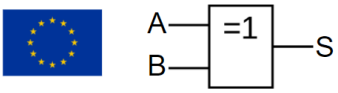
`010010000110010101101100011011000110111100100000010101110110111101110010011011000110010000100001`

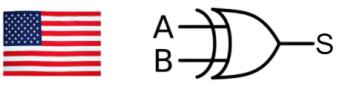
2.5 Le chiffrement de Vernam

Il s'agit d'un système parfaitement sûr, le mathématicien *C.Shannon* a montré que si on respecte bien les trois règles de *Vernam*, ce système est inviolable.

- ▷ La clé doit être une suite de caractères au moins aussi longue que le message à chiffrer.
- ▷ Les caractères composant la clé doivent être choisis de façon totalement aléatoire.
- ▷ Chaque clé, ou masque, ne doit être utilisée qu'une seule fois (d'où le nom de **masque jetable**).

Une méthode particulièrement efficace pour traiter le chiffrement et le déchiffrement de messages à partir de la clé est d'utiliser la fonction logique `XOR`, c'est-à-dire l'opération logique **OU EXCLUSIF** vue en classe de 1ère.





Pour rappel, vous trouverez la table de vérité du `XOR` ci-dessous :

A	B	S
0	0	0
0	1	1
1	0	1
1	1	0

L'une des propriétés les plus intéressantes de cette opération est qu'elle est **symétrique** et **réversible** :

$$(A \oplus B) \oplus B = A$$

`XOR` se note $\oplus$.

Exemple

On note M le message et S la clé secrète.

On obtient le message chiffré C en faisant : $C = M \oplus S$

Le déchiffrement se fait tout simplement en appliquant la même opération $C \oplus S = M$

Malgré sa simplicité, ce système de chiffrement (inventé en 1917 par *Gilbert Vernam*) est très sûr. Il a été utilisé pour le **téléphone rouge** reliant directement le *Kremlin* à la *Maison-Blanche*, les clés transitant alors par valises diplomatiques.

Le système de masque jetable était également employé par les espions soviétiques. Certains masques furent utilisés plus d'une fois (parfois avec des années d'intervalle) ce qui permit aux services de renseignements anglais de déchiffrer certains messages.

2.6 En résumé

Le gros problème avec le chiffrement symétrique, c'est qu'il est indispensable pour A et B de se mettre d'accord à l'avance sur la clé qui sera utilisée lors des échanges et que cette clé reste secrète. Le chiffrement asymétrique permet d'éviter ce problème.

	Avantages	*Inconvénients*
Cryptographie à clefs secrètes / symétriques	+ Rapidité de traitement + Relativement sûr	- Protection et stockage de la clef secrète - Répudiation possible - Pas d'authentification
Cryptographie à clefs publiques / asymétriques	+ Permet la signature + Permet l'authentification + Non répudiation + Permet la transmission sur un canal peu sûr d'une clef de chiffrement	- Protection et stockage de la clef privée - Lenteur de la procédure de déchiffrement

Vidéo présentant un résumé de quelques une des méthodes de chiffrements symétriques.

Faire les exercices 1 à 5.

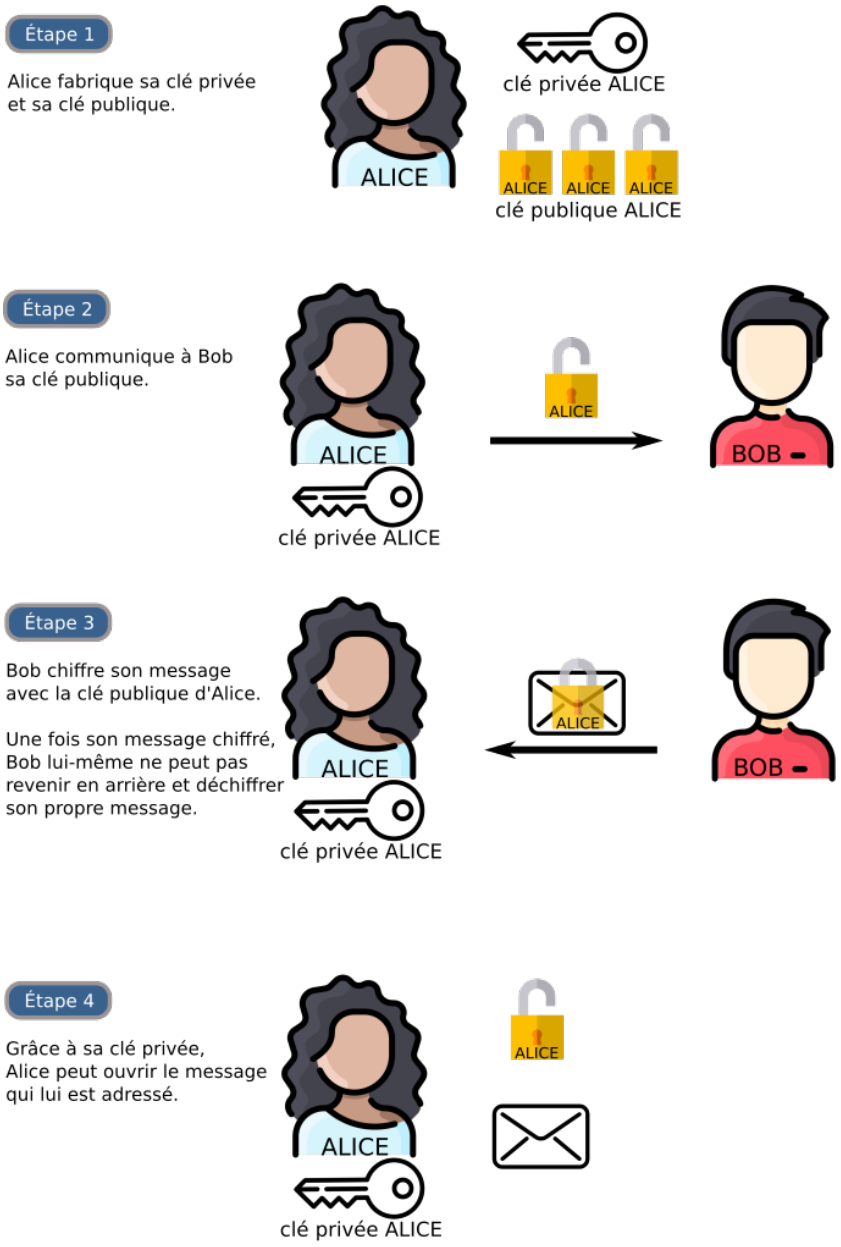
3 Le chiffrement asymétrique

3.1 Principe du chiffrement asymétrique

L'inconvénient essentiel du chiffrement symétrique est le nécessaire **partage d'un secret (la clé) avant l'établissement de la communication sécurisée**.

La cryptographie asymétrique permet de résoudre le problème de l'échange de la **clé secrète**. Elle fut inventée par des chercheurs de l'université de *Stanford*, *Whitfield Diffie* et *Martin Hellman* en 1976, qui reçurent le *prix Turing* 2015 pour cette découverte.

Le principe de base est l'existence d'une **clé publique**, appelée à être distribuée largement, et d'une **clé privée**, qui ne quitte jamais son propriétaire.



L'illustration précédente associe :

- une image de **cadenas** à la **clé publique** (car on s'en sert pour chiffrer les messages)
- une image de **clé** à la **clé privée** (car on s'en sert pour déchiffrer les messages)

3.2 Le rôle interchangeable des clés publiques et privées

Concrètement, la clé privée et la clé publique sont deux nombres aux rôles identiques.

Appelons-les A et B :

— il est impossible de trouver A en fonction de B. Réciproquement, si on connaît A, il est impossible d'en déduire B.
— si on chiffre un message avec A, on peut le déchiffrer avec B. Réciproquement, si on chiffre avec B, on peut déchiffrer le message grâce à A.
— on peut donc chiffrer avec une clé publique et déchiffrer avec la clé privée associée (ce qui est fait dans l'exemple précédent). Mais on peut aussi chiffrer avec la clé privée, et déchiffrer avec la clé publique associée.

A et B ont donc des rôles interchangeables (chacun peut être un cadenas, chacun peut être une clé), et ce n'est qu'en connaissant A et B qu'on peut déchiffrer le message.

Pour que cela fonctionne, il faut que la paire clé publique/clé privée ait une propriété particulière.

Soit F_P la fonction de chiffrement utilisée avec la clé publique, et F_S la fonction relative à la clé privée. Une relation particulière relie ces deux fonctions :

$$m = F_S(F_P(m))$$

Enfin pour que ce système fonctionne, il faut que :

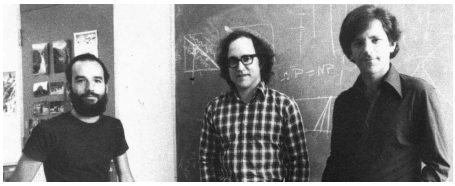
— la fonction F_P soit facile à calculer pour tout le monde,
— la fonction F_S soit facile à calculer uniquement pour le détenteur de la clé privée.

Les algorithmes de chiffrement asymétriques reposent sur des problèmes faciles à résoudre dans un sens et très difficiles à résoudre dans l'autre sens.

Un système qui satisfait ces deux critères est le système de **chiffrement RSA** utilisé pour échanger des données confidentielles sur Internet.

3.3 Le système RSA

Cet algorithme fut inventé en 1977 par *Ronald **R**ivest*, *Adi **S**hamir* et *Leonard **A**dleman* breveté par le MIT en 1983. Le brevet a expiré le 21 septembre 2000 ce qui permet de l'utiliser librement depuis.



Le système de chiffrement asymétrique RSA est le plus couramment utilisé aujourd'hui, notamment pour tout ce qui concerne le **commerce électronique**.

Il consiste en la création, pour chacun des deux participants, de deux clés (publique et privée), liées entre elles de telle sorte que le chiffrement d'un message par la clé publique, puis par la clé privé redonne le message original, et que l'opération inverse est vraie aussi.

RSA se base sur la factorisation des très grands nombres premiers.

Vidéo sur le principe de l'arithmétique modulaire dans le protocole RSA.

3.3.1 Les étapes de l'algorithme RSA

1. Génération des clés :

- Choisissez deux nombres premiers distincts très grands, p et q.
- Calculez leur produit, $n = p \times q$. Ceci est appelé le **module de chiffrement**.
- Calculez la fonction d'Euler de n : $\phi(n) = (p-1) \times (q-1)$.
- Choisissez un exposant de chiffrement e tel que $1 < e < \phi(n)$ et que e soit premier avec $\phi(n)$(c'est-à-dire qu'ils n'ont pas de facteur premier commun autre que 1).
- Calculez l'exposant de déchiffrement, d, tel que $(d \times e)$ % $\phi(n) == 1$. C'est-à-dire, d est l'inverse modulaire de e modulo $\phi(n)$.

2. Chiffrement :

- Convertissez le message en un entier représentatif.
- Chiffrez le message en utilisant la formule : $c = (message)^e$ % n.

3. Déchiffrement :

Utilisez la clé privée *(d, n)* pour déchiffrer le message chiffré en utilisant la formule : $message = (c^d)$ % n.

Ces étapes vous permettent de chiffrer un message avec la clé publique et de le déchiffrer avec la clé privée correspondante. C'est le principe fondamental de l'algorithme RSA.

3.3.2 Exemple de chiffrement avec RSA

Supposons que vous souhaitez utiliser l'algorithme RSA pour chiffrer un message "`HELLO`".

Génération des clés :

- Choisissez deux nombres premiers : $p = 7$ et $q = 11$.
- Calculez le module de chiffrement : $n = p \times q = 7 \times 11 = 77$.
- Calculez la fonction $\phi(n)$: $\phi(77) = (p-1) \times (q-1) = (7-1) \times (11-1) = 6 \times 10 = 60$.
- Choisissez un exposant de chiffrement : $e = 13$.
- Trouvez l'exposant de déchiffrement d tel que $(d * e)$ % $\phi(n) == 1$.

  En utilisant l'algorithme d'Euclide étendu, trouvez que $d = 37$.

Chiffrement :

- Convertissez le message "HELLO" en nombres, par exemple, en utilisant la table *ASCII* : `H = 72`, `E = 69`, `L = 76`, `L = 76`, `O = 79`.
- Chiffrez chaque nombre en utilisant la formule $c = (message^e)$ % n :
- Pour `H` : $c = (72^{13})$ % $77 = 344373768865426431$ % $77 = 48$.
- Pour `E` : $c = (69^{13})$ % $77 = 680369848560$ % $77 = 47$.
- Pour `L` : $c = (76^{13})$ % $77 = 2131167792897337916176$ % $77 = 59$.
- Pour `L` :$c = (76^{13})$ % $77 = 2131167792897337916176$ % $77 = 59$.
- Pour `O` :$c = (79^{13})$ % $77 = 575398451422268922025$ % $77 = 47$.
- Le message chiffré est donc : `48 47 59 59 47`.

Déchiffrement :

Pour déchiffrer chaque nombre chiffré, utilisez la formule : $message = (c^d)$ % n.

- Pour 48 : $message = (48^{37})$ % $77 = 72$ (`H`).
- Pour 47 : $message = (47^{37})$ % $77 = 69$ (`E`).
- Pour 59 : $message = (59^{37})$ % $77 = 76$ (`L`).
- Pour 59 : $message = (59^{37})$ % $77 = 76$ (`L`).
- Pour 47 : $message = (47^{37})$ % $77 = 79$ (`O`).

Le message original "`HELLO`" a été chiffré en "`48 47 59 59 47`" et déchiffré avec succès en "`HELLO`" en utilisant l'alogorithme de chiffrement RSA.

3.3.3 RSA, un système inviolable ?

Le problème du chiffrement asymétrique est que ses **performances (mémoire, vitesse) sont nettement inférieures** au chiffrement symétrique. Le chiffrement RSA a des défauts (notamment une grande consommation des ressources, due à la manipulation de très grands nombres).

Mais le choix d'une clé publique de grande taille (actuellement 1024 ou 2048 bits) le rend pour l'instant **inviolable**.

Actuellement, il n'existe pas d'algorithme efficace pour factoriser un nombre ayant plusieurs centaines de chiffres.

Deux évènements pourraient faire s'écrouler la sécurité du RSA :

- la découverte d'un **algorithme efficace de factorisation**, capable de tourner sur les ordinateurs actuels. Cette annonce est régulièrement faite, et tout aussi régulièrement contredite par la communauté scientifique. (cf. https://www.schneier.com/blog/archives/2021/03/no-rsa-is-not-broken.html)
- l'**avènement d'ordinateurs quantiques**, dont la vitesse d'exécution permettrait une factorisation rapide. Il est à noter que l'algorithme de factorisation destiné à tourner sur un ordinateur quantique existe déjà : l'algorithme de Schor.

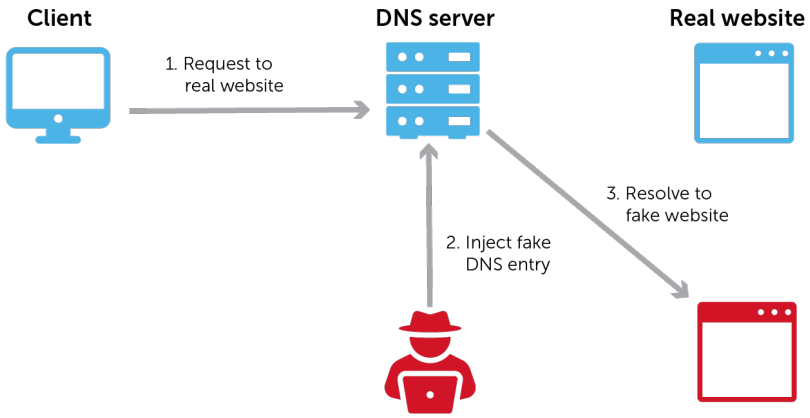
4 Le protocole HTTPS

Nous allons maintenant voir une utilisation concrète de ces chiffrements symétriques et asymétriques : le protocole HTTPS.

Avant de parler du protocole HTTPS, petit retour sur le protocole HTTP : un client effectue une requête HTTP vers un serveur, le serveur va alors répondre à cette requête (par exemple en envoyant une page HTML au client).

Le **protocole HTTP** pose 2 problèmes en termes de sécurité informatique :

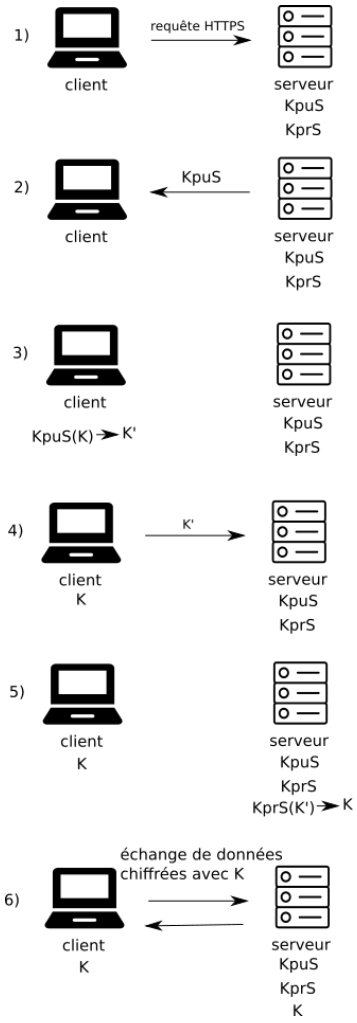
1. Un individu qui intercepterait les données transitant entre le client et le serveur pourrait les lire sans aucun problème (ce qui serait problématique notamment avec un site de e-commerce au moment où le client envoie des données bancaires)
2. Grâce à une technique qui ne sera pas détaillée ici (le *DNS spoofing*), un serveur "pirate" peut se faire passer pour un site sur lequel vous avez l'habitude de vous rendre en toute confiance. Comme illustré sur le schéma ci-dessous, un pirate pourrait récupérer vos identifiants bancaires en créant un faux site identique au site de votre banque.



HTTPS est donc la version sécurisée de HTTP, il s'appuie sur le protocole TLS (*Transport Layer Security*) anciennement connu sous le nom de SSL (*Secure Sockets Layer*).

Il permet d'éviter les 2 problèmes évoqués en garantissant la **confidentialité** et l'**authentification**.

4.1 La confidentialité dans HTTPS



HTTPS fournit une couche de chiffrement des données circulant entre le client et le serveur.

Les communications vont être chiffrées grâce à une clé symétrique (pour des raisons de débit). Problème : comment échanger cette clé entre le client et le serveur ?

En utilisant une paire clé publique / clé privée !

Voici le déroulement des opérations :

1 et 2. Le client effectue une requête HTTPS vers le serveur, en retour le serveur envoie sa clé publique (KpuS) au client.

3. Le client "fabrique" une clé K (qui sera utilisé pour chiffrer les futurs échanges), chiffre cette clé K avec KpuS et envoie la version chiffrée de la clé K au serveur.

4 et 5. Le serveur reçoit la version chiffrée de la clé K et la déchiffre en utilisant sa clé privée (KprS). À partir de ce moment-là, le client et le serveur sont en possession de la clé K.

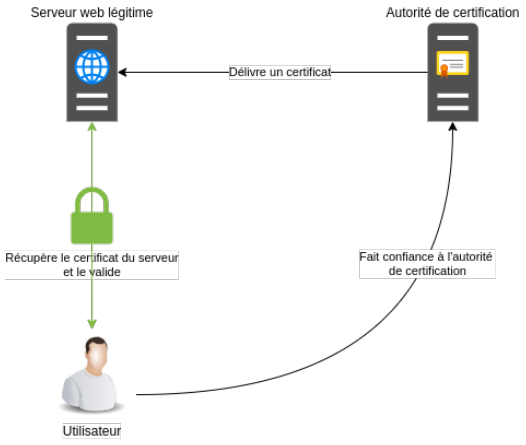
6. Le client et le serveur commencent à échanger des données en les chiffrant et en les déchiffrant à l'aide de la clé K (chiffrement symétrique).

Ce processus se répète à chaque fois qu'un nouveau client effectue une requête HTTPS vers le serveur.

4.2 L'authentification dans HTTPS

L'autre aspect de la sécurité du protocole HTTPS est de s'assurer que le serveur justifie de son "identité".

Le problème vient de la transmission de la clé publique : si celle-ci n'est pas sécurisée, un attaquant peut se positionner entre l'entreprise et son audience en diffusant de fausses clés publiques (par le biais d'un faux site web par exemple) puis intercepter toutes les communications, lui permettant de se faire passer pour l'entreprise.



Les **certificats** résolvent ce problème grâce à la signature d'une autorité de confiance.

Un certificat électronique est un fichier contenant :

- la clé publique de l'entreprise ;
- des informations d'identification, par exemple : nom, localisation, adresse électronique de l'entreprise ;
- une **signature sécurisée** d'une autorité de confiance garantissant l'exactitude des informations du certificat.

Le serveur enverra donc le **certificat authentifié** par l'autorité.

En cas d'absence de certificat (ou d'envoi de certificat non conforme), le client stoppe immédiatement les échanges avec le serveur. Ce mécanisme rend toute attaque du type *DNS Spoofing* impossible sur HTTPS.

Vidéo sur le chiffrement SSL/TLS : https://youtu.be/7W7WPMX7arI?si=zvsZdg0D7_P5N-Ji

5 Exercices

Exercice 1 : *Chiffrement de Vernam*

On souhaite transmettre le message `NSI` de façon privée. Le code *ASCII* correspondant au message M est `01001110 01010011 01001001`. On crée une clés secrète S aléatoirement que l'on transmet au destinataire : `11001001 11001101 01100011`.

1. Ecrire le message chiffré `C` en faisant : $C = M \oplus S$
2. Vérifier qu'on déchiffre par la même opération : $C \oplus S = M$

Exercice 2 : *Module* `string`

Pour réaliser les exercices ci-dessous, on aura besoin d'utiliser le module `string` de Python.

1. Dans un fichier Python, copier le code ci-dessous :

```
1 from string import ascii_uppercase as alphabet
2
3 print(alphabet)
4 alphabet[0], alphabet[25]
5
6 alphabet.index("A")
7 alphabet.index("Z")
8
9 "Message".upper()
```

2. Que produisent les lignes 3 et 4 ?
3. Que produisent les lignes 6 et 7 ?
4. Que permet de faire la ligne 9 ?
5. Trouver la commande qui permet de mettre n'importe quel texte en minuscules.

Exercice 3 : *Chiffrement de César*

Créer deux fonctions `chiff_cesar` et `dechiff_cesar` pour réaliser respectivement le chiffrement et déchiffrement avec en argument le message et la clé.

Ces deux fonctions ont deux paramètres :

— `message` : une chaîne de caractères à (dé)coder.
— `n` : un entier qui donne le décalage.

Par exemple, la fonction `chiff_cesar` doit convertir le message en majuscule, et décaler toutes les lettres de `n` grâce au codage César, et laisser tous les autres caractères inchangés (ponctuation).

Voici une série de tests à passer.

```
1  assert cesar('message', 0) == "MESSAGE"
2  assert cesar('message', 1) == 'NFTTBHF'
3  assert cesar('message', -1) == 'LDRRZFD'
4  assert cesar('message', 26) == 'MESSAGE'
5  assert cesar('message', 53) == 'NFTTBHF'
6  assert cesar('message', -27) == 'LDRRZFD'
7
8  eluard = "Sur mes cahiers d ecolier, sur mon pupitre et les arbres, sur le
       sable, sur la neige, j ecris ton nom"
9
10 assert cesar(eluard, 13) == 'FHE ZRF PNUVREF Q EPBYVRE, FHE ZBA CHCVGER RG
       YRF NEOERF, FHE YR FNOYR, FHE YN ARVTR, WEPEVF GBA ABZ'
11
```

```
12 # verification du decodage
13 assert cesar(cesar(eluard,13),-13) == eluard.upper()
```

Exercice 4 : *Chiffrement par substitution*

Implémenter le chiffrement par substitution avec une fonction `substitution` qui prend deux paramètres :

- `message` : une chaîne de caractères à coder ou décoder.
- `clé` : une chaîne de caractères qui donne les correspondances des caractères.

On pourra vérifier son fonctionnement sur l'exemple du cours :

```
1 assert substitution("SUBSTITUTION", "AZERTYUIOPQSDFGHJKLMWXCVBN") ==
    "LWZLMOMWMOGF"
```

Comment devra être effectué le décodage à partir du message codé et de la clé ?

Exercice 5 : *OU Exclusif*

`XOR` est une opération de logique bit à bit qui renvoie 0 si les deux bits sont égaux et 1 sinon.

Une particularité de cette opération est :

`Si (b1 xor b2 == b3) alors (b1 xor b3 == b2) et (b2 xor b3 == b1)`

Ce qui permet un chiffrement symétrique.

Par exemple, le mot `bonjour` dont le code *ASCII* est donné par ce programme :

```
m = "bonjour"
for c in m:
    print(ord(c),end=' ')
```

On obtient : `98 111 110 106 111 117 114`

Le mot '`nsi`' va nous servir de clé : ('`nsi`' → `110 115 105`)

La méthode consiste alors à aligner le mot et la clé en la répétant autant de fois que nécessaire et d'effectuer un `XOR` entre les codes *ASCII* :

b	o	n	j	o	u	r
98	111	110	106	111	117	114
n	s	i	n	s	i	n
110	115	105	110	115	105	110

En pratiquant un `XOR` lettre par lettre entre les nombres obtenus (après une écriture en binaire...), on obtient les codes *ASCII* suivant : `12 28 7 4 28 28 28` (que l'on pourrait transformer en caractères mais ce n'est pas très utile...)

Pour décoder le message, il suffit alors de recommencer l'opération avec les codes *ASCII* du message chiffré en utilisant la même clé : (`12 xor 110 = 98` donc '`b`') etc.

En Python, l'opérateur `^` permet d'effectuer un `XOR` directement sur deux entiers.

Le programme suivant chiffre un message en utilisant cette méthode :

```
message = "Bonjour, vive la NSI!"
cle = "mystere"

def chiffre(message,key):
    c = []
    n = len(message)
    lk = len(key)
    j = 0

    for i in range(n):
        c.append(ord(message[i]) ^ ord(key[j]))
        j = (j+1)%lk

    return c

print(chiffre(message,cle))
```

Ce qui donne en console :

```
>>>
[47,22,29,30,135,7,23,65,89,16,27,133,31,0,3,13,83,21,132,30,0,23,84,5,27,
157,1,69,82]
```

Chaque nombre peut être converti en caractère (le contenu complet du message est chiffré, y compris les caractères de ponctuation et d'espace).

Chaque nombre peut être converti en caractère (le contenu complet du message est chiffré, y compris les caractères de ponctuation et d'espace).

1. Modifier la fonction `chiffre` pour qu'elle renvoie le message chiffré (avec les caractères).
2. Écrire une fonction `dechiffre` qui prend en paramètres le message chiffré et la clé et qui renvoie le message déchiffré.
3. Valider les tests unitaires suivants :

   ```
   assert dechiffre(chiffre("Hello World!","key1"),"key1") == "Hello World!"
   ```

   ```
   assert dechiffre(chiffre("Hello World!","key1"),"key2") != "Hello World!"
   ```
4. Vérifier que les méthodes soient documentées (*docstring*).

Exercice 6 : *Module sympy*

1. Consulter la documentation de librairie `sympy`.
2. Décoder la phrase `RYTVJKGCLJWRTZCVRMVTLEDFULCVHLZWRZKKFLKRMFKIVGCRTV`, sachant qu'elle a été chiffrée par décalage (*shift* en anglais...)

Exercice 7 : *Chiffrage affine*

Principe du chiffrage affine :

— Chaque lettre est codée par son rang, en commençant à 0 (`A` : 0, `B` : 1, ..., `Z` : 25)
— On applique à chaque rang la transformation affine : $f(x) = (ax + b)\ \%\ 26$

où a et b sont deux nombres entiers. Attention, a doit être premier avec 26 !

1. Implémenter la fonction `affine(msg,a,b)`
2. Comparer vos résultats avec ceux obtenus par la fonction `encipher_affine()` de `sympy`.
3. Décodez la phrase `UCGXLODCMOXPMFMSRJCFQOGTCRSUSXC`, sachant qu'elle contient le mot `TRAVAIL` et que et a et b sont inférieurs à 20.

Aide : L'instruction `gcd` du module `math` permet de calculer le PGCD de deux nombres.

Exercice 8 : *Chiffrement RSA*

Alice veut écrire à Bob.

Soit le couple de nombre premiers (p, q) avec $p = 15$ et $q = 13$.

1. Calculer n et $\phi(n)$.
2. Justifier que $(9, 65)$ ne peut pas être une clé publique.
3. Vérifier que $(11, 65)$ est une clé publique. C'est la clé publique de Bob.
4. Vérifier que 35 est un inverse de 11 modulo 48.
5. En déduire la clé privée de Bob.
6. Chiffrer le nombre secret d'Alice 17 avec la clé publique de Bob. C'est ce nombre qu'Alice envoie à Bob.
7. Déchiffrer le nombre reçu par Bob.

Exercice 9 : *Le chiffrement asymétrique*

Le schéma ci-dessous rappelle le principe du chiffrement asymétrique :

Nous allons illustrer ce principe en utilisant le module `rsa.py` fourni.

Ce module contient la classe `rsa` dont nous utiliserons 3 méthodes :

- `rsa_get_public_key()`
- `rsa_encode_text()`
- `rsa_decode_text()`

1. Ouvrir le fichier `rsa.py` et lire la description (*Docstring*) de ces méthodes.
2. Écrire un programme qui réalise les opérations suivantes :

   - Anne partage sa clef publique.
   - Anne reçoit le message "`test`" de Bob
   - Anne envoie le message en clair `"Hello World!"`.
   - Bob et Dylan lisent le message d'Anne.
   - Bob envoie le message `"OK"` chiffré **uniquement pour Anne** et l'envoie à Anne et Dylan.
   - Anne et Dylan lisent le message de Bob.

   Résultat attendu :

   - Anne reçoit : `test`
   - Bob reçoit : `Hello World!`
   - Dylan reçoit : `Hello World!`
   - Dylan reçoit : ... (ce message sera différent à chaque test)
   - Anne reçoit : `OK`

3. Tester l'échange de messages chiffrés entre Dylan et Bob.

Exercice 10 : *Certificats*

1. À l'aide de *Chrome*, accéder au site de l'IFS : https://www.ifs.edu.sg
2. Afficher son certificat en cliquant sur ... puis sur *Connection is secure* et enfin sur *Certificate is valid*.
3. Dans *Details*, rechercher et noter les informations suivantes :
   - la date de validité
   - l'autorité de certification
   - les informations sur le site
   - l'algorithme de chiffrement de la communication
   - la clé publique

Exercice 11 : *Chiffrons une image*

Le code suivant permet de créer et afficher une image de dimension 240 x 180 en niveaux de gris (valeurs entre 0 et 255) ; cette image est composée de bandes blanches (valeur 255) et noires (valeur 0) verticales.

```
from PIL import Image
import numpy as np

    def matrice(n,p):
        #Initialise m comme une matrice de 0 de taille n x p

        m=[[0 for j in range(p)] for i in range(n)]

        for i in range(n):
            #n est la hauteur de l'image
            for j in range(p):
                #p est la largeur de l'image
                #valeurs differentes dans chaque bande de 20 pixels
                if (j // 20) % 2 == 0:
                    m[i][j]=255
                else:
                    m[i][j]=0
        return np.array(m)

img=matrice(240,180)

print("Image originale")
Image.fromarray(img).show()
```

1. Créer une fonction `generer_cle(n,p)` qui génère une matrice de taille 240 x 180 remplie de valeurs aléatoires (vous pouvez utiliser la fonction `randint(0,255)`). Afficher cette clé.
2. Créer une fonction `chiffre` qui prend en argument l'image originale et la clé, et fait un *OU EXCLUSIF* entre chaque élément (pixel) de l'image et de la clé. Afficher l'image chiffrée.
3. On souhaite maintenant déchiffrer l'image, que suggérez-vous ?
4. Réaliser le déchiffrement et afficher l'image finale pour vérifier qu'elle est identique à l'image originale.