Séq. 15 - Sécurisation des communications

Objectifs

1. Décrire les principes de chiffrement symétrique (clef partagée).
2. Décrire les principes de chiffrement asymétrique (avec clef privée/clef publique).
3. Décrire l'échange de clé symétrique en utilisant un protocole asymétrique pour sécuriser une communication HTTPS

Cette séquence s'appuie sur :
- http://www.cryptage.org/
- https://pixees.fr/informatiquelycee/n_site/nsi_term_archi_secu.html

1 Introduction

La cryptologie, étymologiquement la « science du secret », ne peut être vraiment considérée comme une science que depuis peu de temps. Elle englobe la cryptographie — l'écriture secrète – et la cryptanalyse – l'analyse de cette dernière.

1.1 Un peu d'histoire

Wikipédia : https://fr.wikipedia.org/wiki/Histoire_de_la_cryptologie

1.2 Vidéo d'introduction

Youtube : scienceetonnante code secret :
https://www.youtube.com/watch?v=8BM9LPDjOw0

2 Le chiffrement symétrique

2.1 Prérequis python : Notion de modulo

En informatique, modulo signifie « reste de la division euclidienne ». Il est utilisé quand on veut boucler et revenir au début d'une chaîne.
Par exemple :
Soit la correspondance : 0-a, 1-b, 2-c, … On s'arrête à 25-z.
Pour certaines raisons, il peut être nécessaire de boucler et avoir 26-a, 27-b, … On vient de boucler à 26.
Pour avoir le caractère correspondant à un nombre, il suffit alors de faire un modulo 26. En effet,

```
>>> 1%26
1
>>> 27%26
1
```

2.2 Prérequis python : Le module string

Voir : https://docs.python.org/fr/3/library/string.html

Les constantes définies dans ce module sont :

- `string.ascii_lowercase` : Les lettres minuscules 'abcdefghijklmnopqrstuvwxyz'.
- `string.ascii_uppercase` : Les lettres majuscules 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'.
- `string.ascii_letters` : La concaténation des constantes `string.ascii_lowercase` et `string.ascii_uppercase`.
- `string.digits` : La chaîne '0123456789'.
- `string.punctuation` : Chaîne de caractères ASCII considérés comme ponctuation dans la configuration de localisation C : !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~.
- `string.whitespace` : Une chaîne comprenant tous les caractères ASCII considérés comme espaces. Sont inclus les caractères espace, tabulations, saut de ligne, retour du chariot, saut de page, et tabulation verticale.
- `string.printable` : Chaîne de caractères ASCII considérés comme affichables. C'est une combinaison de `digits`, `ascii_letters`, `punctuation`, et `whitespace`.

2.3 Prérequis python : ord() et chr()

```
>>> ord('a')
97
>>> chr(97)
'a'
```

A faire vous même 1.

- En 2 lignes de langage python, créez une boucle qui imprime un compteur qui imprime la séquence :

… 1 … 2 … 3 … 4 … 5 … 1 … 2 … 3 … 4 … 5 … 1 … 2 … 3

- Testez les possibilités du module string
- Écrivez un petit programme avec un compteur qui s' incrémente de 1 jusqu' à 200 et qui écrit : `abcd...xyzabcd...xyzabcd...` pour former une chaîne de 200 caractères.

2.4 Codage par substitution : Substitution monoalphabétique

Youtube - scienceetonnante code secret :
https://www.youtube.com/watch?v=8BM9LPDjOw0 de 0 à 2'19

2.4.1 Le code de César

Principe : Pour chaque lettre, on procède à un décalage et donc une lettre se substitue à une autre.
Exemple de décalage de 3 :

CLAIR	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
-> décalage = 3																										
CODE	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C

D'après cette méthode, "VIVE LES MATHS" devient donc "YLYH OHV PDWKV" !

A faire vous même 2.

- En python, écrivez une fonction `numero_lettre` qui donne un numéro d' ordre pour chaque lettre : 1 pour 'A', 2 pour 'B', 3 pour 'C', … 26 pour 'Z'
- Écrivez une fonction `codage_cesar` qui prend 2 arguments (une chaîne de caractères et un nombre pour le décalage) et qui retourne la chaîne codée.
- Implémentez la fonction réciproque : `decodage_cesar`.

2.4.2 Autre code de substitution monoalphabétique

Principe : Le code consiste à inverser l'ordre des lettres. Ainsi A devient Z, B devient Y, C devient X, ...
D'après cette méthode, "`BONJOUR`" devient donc "`YLMQLFI`"
Un des avantages c'est que la même méthode sert à coder et à décoder.

A faire vous même 3.

- Écrivez la fonction `codage_inverse` qui prend 1 argument (une chaîne de caractères) et qui retourne la chaîne codée.
- Vérifiez que la même fonction sert aussi bien à coder qu'à décoder.

2.4.3 Chiffrement par substitution

Principe : Créer une table de correspondance au hasard.
Il faut alors transmettre toute la table qui sert à coder et à décoder.

A faire vous même 4.

- Écrivez la fonction `genere_table` qui génère aléatoirement une table de correspondance (ou plutôt un dictionnaire)
- Écrivez une fonction codage qui prend 2 arguments (une chaîne de caractères et une table de codage) et qui retourne la chaîne codée.
- Implémentez la fonction réciproque : `decodage`.

2.4.4 Moyen de casser le code : La brute force

La force brute consiste à tester toute les combinaisons possibles pour trouver la bonne. Si on prend seulement les 26 lettres de l'alphabet, il y aurait 26 ! possibilités.
Il faut beaucoup de temps et des machines puissantes pour y arriver.

A faire vous même 5. Pour les plus rapides

- Voir vidéo : https://www.youtube.com/watch?v=IXjrGUhe7vw
- Ecrivez et testez la fonction de la vidéo
- Dans la liste, remplacez chaque lettres pour des prénoms, des noms, des nombres, des dates de naissance, ...

2.4.5 Moyen de casser le code : La fréquence des lettres

Youtube - scienceetonnante code secret :
https://www.youtube.com/watch?v=8BM9LPDjOw0 de 2'19 à 3'35

- P.274 ex 8

2.5 Variante avec une clé de chiffrement – Substitution polyalphabétique

2.5.1 Principe

Youtube - scienceetonnante code secret :
https://www.youtube.com/watch?v=8BM9LPDjOw0 de 3'35 à 4'34

A faire vous même 6. Pour les plus rapides ou plus motivés

- Écrivez une fonction `codage_cle_chiffrement` qui prend 2 arguments (une chaîne de caractères à décoder et une chaîne de caractères pour le chiffrement) et qui retourne la chaîne codée.
  Exemple :
  ```
  JE TIENS ENFIN LA GAULE
  SC IENCE SCIEN CE SCIEN
  CH CNSQX XQONB OF ZDDQS
  ```
  10 (correspond à J) + 19 (correspond à S) = 29 = 26 + 3 (correspond à C)
- Implémentez la fonction réciproque : `decodage_cle_chiffrement`.

2.5.2 Passage en binaire

Soit 2 individus A et B qui cherchent à s' envoyer des messages par l'intermédiaire d'un réseau informatique. A et B désirent qu'une tierce personne (par exemple P) ne soit pas capable de lire les messages. Pour ce faire, A va chiffrer le message .
Pour chiffrer un message, A va utiliser une suite de caractère que l'on appelle "clé de chiffrement". Dans le cas du chiffrement symétrique, cette clé de chiffrement sera aussi utilisée par B pour déchiffrer le message envoyé par A. Dans ce cas, la clé de chiffrement est identique à la clé de déchiffrement. Concrètement comment cela se passe-t-il ?
Comme nous avons déjà eu l'occasion de le voir en première, toute "donnée informatique" peut être vue comme une suite de zéro et de un. Nous chercherons donc à chiffrer une suite de zéro et de un :
Soit le message "Hello World!" ce qui nous donnera en binaire :

```
010010000110010101101100011011000110111100100000010101110110111101110011001101100
0110010000100001
```

N.B. nous avons simplement utilisé le code ASCII de chaque caractère (par exemple, on peut vérifier que le H correspond bien à l'octet 01001000). Pour effectuer la "conversion" texte vers code binaire ASCII ou vis versa, vous pouvez utiliser le site https://www.rapidtables.com/convert/number/ascii-to-binary.html
Choisissons maintenant un mot (ou une phrase) qui nous servira de clé de chiffrement, prenons pour exemple le mot "toto". "toto" nous donne en binaire :

```
01110100011011110111010001101111
```

Pour chiffrer le message nous allons effectuer un XOR bit à bit. Pour rappel, vous trouverez la table de vérité du XOR ci-dessous :
Table de vérité "XOR" :

E1	E2	S
0	0	0
0	1	1
1	0	1
1	1	0

Comme la clé est plus courte que le message, il faut "reproduire" la clé vers la droite autant de fois que nécessaire (si la taille du message n'est pas un multiple de la taille de la clé, on peut reproduire seulement quelques bits de la clé pour la fin du message):

```
⊕ 010010000110010101101100011011000110111100100000010101110110111101110011001101100
  011101000110111101110100011011110111010001101111011100100110110001100100001000001
  ---------------------------------------------------------------------------------
  001111000000101000011000000000011000110110100111100100011000000000000011000000011
  000100000100111
```

Le signe + dans un cercle symbolise le XOR
Après ce XOR on obtient donc la suite de bits suivante :

```
0011110000001010000110000000001100011011010011111001000110000000000000110000000011
0001000001001110
```

Soit la chaîne de caractères suivante (si on cherche à afficher le message chiffré avec un éditeur de texte) : O#N
Maintenant ce message est prêt pour être envoyé à son destinataire B. Si P intercepte le message est cherche à le lire avec un éditeur de texte, il obtiendra la suite de caractère O#N
B a maintenant reçu le message chiffré, il possède la clé (toto), il va donc pouvoir déchiffrer le message en appliquant un XOR entre le message chiffré et la clé (on applique exactement la même méthode que ci-dessus).

```
00111100000010100001100000000011 00011011010011110010001100000000 00000110000000110001000001001110
01110100011011110111010001101111 01110100011011110111010001101111 01110100011011110111010001101111
--------------------------------------------------------------------------------------------------
01001000011001010110110001101100 01101111001000000101011101101111 01110010011011000110010000100001
```

On trouve le code binaire suivant :

```
0100100001100101011011000110110001101111001000000101011101101111011100100110110011
00110010000100001
```

Vous pouvez remarquer que nous avons bien retrouvé le code binaire d'origine. Si vous ne voulez pas vous embêter à vérifier bit par bit, vous pouvez utiliser ce site qui vous permettra de repasser du code binaire ASCII au texte.
On retrouve bien le message d'origine : Hello World!, B a pu lire le message envoyé par A alors que pour P, malgré le fait qu'il a pu intercepter le message, il n'a pas pu prendre connaissance de son contenu sans la clé.

A faire vous même 7.

- En python, écrivez la fonction `codage_binaire` qui prend 2 arguments (deux chaînes de caractères) et qui retourne une autre chaîne de caractères.
- Vérifiez que la même fonction sert aussi bien à coder qu'à décoder.

Aides python :

```
>>> 0b100
4
>>> 0b100^0b111
3
>>> 4^7
3
>>> bin(0b100^0b111)
'0b11'
>>> f"{9:b}"
'1001'
>>>  f"{9:09b}"
'000001001'
```

Voir cours P. 264

2.6 Codage par substitution des blocs

A faire vous même 8. Pour les costauds

- P. 271 ex 7 (principe intéressant mais difficile à coder)

2.7 Conclusion

Malgré toutes ses évolutions et ses mises en oeuvre, la cryptographie à clé secrète est toujours entravée par un défaut : la condition sine qua non de son succès est et restera le secret de sa clé (principe de Kerckhoffs). Bien qu' ayant pu au fil du temps réduire sa taille, les cryptographes ont toujours été confrontés au problème de la transmission de cette clé... Mais le progrès ne s'arrête jamais ! Si le problème est de conserver le secret de la clé, pourquoi ne pas le contourner... en inventant un système qui la rend *publique* ?

3 Le chiffrement asymétrique – Algorithme RSA

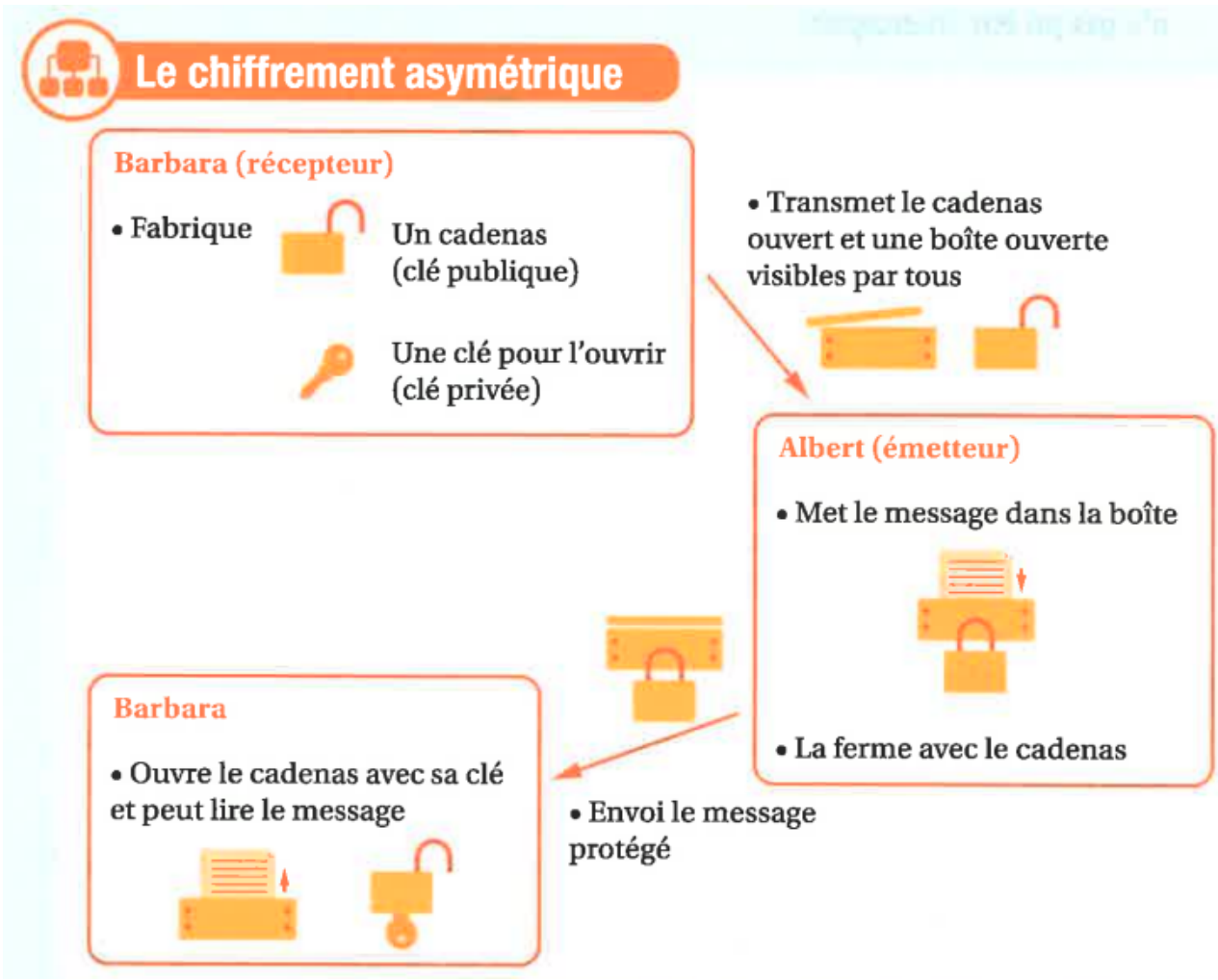
3.1 Principe

- Youtube - scienceetonnante code secret :
  https://www.youtube.com/watch?v=8BM9LPDjOw0 de 4'34 à la fin
- Voir livre P. 264

Ainsi, un système cryptographie à clé publique est en fait basé sur **deux clés** :

- Une clé publique, pouvant être distribuée librement, *c'est le cadenas ouvert*
- Une clé secrète, connue uniquement du receveur, *c'est le cadenas fermé*

C'est la raison pour laquelle on parle de chiffrement asymétrique.
En résumé, on dispose d'une fonction P sur les entiers, qui possède un inverse S. On suppose qu'on peut fabriquer un tel couple (P,S), mais que connaissant uniquement P, il est impossible (ou au moins très difficile) de retrouver S. Autrement dit, il faut **déterminer mathématiquement des fonctions difficilement inversibles, ou "à sens unique".**



3.1.1 Le protocole de Diffie et Hellman

Parallèlement à leur principe de cryptographie à clé publique, Diffie et Hellman ont proposé un protocole d'échanges de clés totalement sécurisé, basé sur des fonctions difficiles à inverser.

(1) Alice et Bob se mettent d'accord publiquement sur un très grand nombre premier "p" et sur un nombre "n" inférieur à "p".

(2) Alice engendre une clé secrète "a" et Bob une clé secrète "b".

(3) Alice calcule l'élément public k_a et Bob l'élément public k_b :

$$k_a = n^a \bmod p$$

$$k_b = n^b \bmod p$$

(4) Alice transmet sa clé publique k_a à Bob, et Bob transmet sa clé publique k_b à Alice.

(5) Alice et Bob profitent ensuite de la commutativité de la fonction exponentielle pour établir leur secret commun :

$$K_{Alice} = (k_b)^a = (n^b)^a \bmod p$$

$$K_{Bob} = (k_a)^b = (n^a)^b \bmod p$$

=> $K_{Alice} = K_{Bob} = n^{ab} \bmod p$

3.1.2 Sécurité du système

A priori, il n'y a pas moyen, à partir des informations transmises publiquement (p, n, n^a, n^b), de trouver n^{ab} sans calculer un logarithme modulo p, ou faire un quelconque calcul d'une complexité exagérée.

Ainsi, la sécurité du système est dite *calculatoire* et repose sur deux hypothèses :

- L' adversaire dispose d'une puissance de calcul limitée
- Avec cette contrainte de puissance et un temps limité, il n'est pas possible d' inverser la fonction exponentielle, ni de trouver n^{ab} à partir de p, n, n^a, n^b .

Remarque 1 : malgré tout cela, en 2001, des experts français ont réussi à inverser la fonction exponentielle modulaire pour un nombre p de 120 chiffres ! La sécurité d'un système dépend donc des progrès constants dans le domaine de la complexité algorithmique.

3.1.3 Les limites du système

Le schéma de Diffie-Hellman, bien qu' astucieux, reste un schéma de principe et souffre d'un inconvénient majeur : il n' assure pas les services de sécurité classiques que sont l' authentification mutuelle des deux intervenants, le contrôle de l'intégrité de la clé et l'anti-rejeu (vérifier qu'une information déjà transmise ne l'est pas à nouveau). L' ennemi peut donc facilement usurper l'identité d' Alice, en remplaçant son élément public par le sien.

3.2 Le RSA

3.2.1 Le principe

Le premier système à clé publique solide à avoir été inventé, et le plus utilisé actuellement, est le système RSA. Publié en 1977 par Ron **R**ivest, Adi **S**hamir et Leonard **A**dleman de l'Institut de technologie du Massachusetts (MIT), le RSA est fondé sur la difficulté de factoriser des grands nombres, et la fonction à sens unique utilisée est une fonction "puissance".

3.2.2 L'algorithme de chiffrement

Départ :

- Il est facile de fabriquer de grands nombres premiers p et q (environ 100 chiffres)
- Étant donné un nombre entier n = pq, il est très difficile de retrouver les facteurs p et q

(1) Création des clés

- La clé secrète : 2 grands nombres premiers p et q
- La clé publique : n = pq ; un entier e premier avec (p-1)(q-1)

(2) Chiffrement : le chiffrement d'un message M en un message codé C se fait suivant la transformation suivante :

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

(3) Déchiffrement : il s'agit de calculer la fonction réciproque

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

tel que $e.d = 1 \bmod [(p-1)(q-1)]$

3.2.3 La signature électronique

Après la confidentialité de la transmission d'un message subsiste un problème : son authenticité. Alice voudrait bien envoyer un message M à Bob de telle façon que celui-ci soit sûr qu'elle est réellement l' émettrice du message, et qu'un intrus ne tente pas de venir semer la confusion.
Le système RSA fournit une solution à ce problème :
Rappelons les données :

- Alice seule détient la clé secrète d et diffuse la clé publique (n,e)
- Alice va se servir de la clé publique pour chiffrer le message M

(1) Alice accompagne son message chiffré de sa **signature**, qui correspond à :

$$M^d$$

(2) Bob va donc voir si l'égalité $(M^d)^e \bmod n = M$ est vérifiée. Si c'est le cas, Alice est bien l' émettrice du message.

3.2.4 Sécurité du système : primalité, factorisation

Signalons enfin que le réel problème du RSA (et des autres systèmes à clé publique) n'est pas la sécurité, mais la lenteur. Tous les algorithmes à clé publique sont 100 à 1000 fois plus lents que les algorithmes à clé secrète, quelle que soit leur implémentation (logicielle ou matérielle) !

3.2.5 Exemple : chiffrer BONJOUR

1) Alice crée ses clés :

- La clé secrète : **p = 53 , q = 97** (Note : en réalité, p et q devraient comporter plus de 100 chiffres !)
- La clé publique : **e = 7 (premier avec 52*96), n = 53*97 = 5141**

2) Alice diffuse sa clé publique (par exemple, dans un annuaire).
3) Bob ayant trouvé le couple **(n,e)**, il sait qu'il doit l'utiliser pour chiffrer son message. Il va tout d'abord remplacer chaque lettre du mot BONJOUR par le nombre correspondant à sa position dans l'alphabet (codé sur 2 chiffres) :
B = 02, O = 15, N = 14, J = 10, U = 21, R = 18

$$BONJOUR = 02\,15\,14\,10\,15\,21\,18$$

4) Ensuite, Bob découpe son message chiffré en blocs de même longueur représentant chacun un nombre plus petit que **n**. Cette opération est essentielle, car si on ne faisait pas des blocs assez longs (par exemple, si on laissait des blocs de 2 chiffres), on retomberait sur un simple chiffre de substitution que l'on pourrait attaquer par **l'analyse des fréquences.**

$$BONJOUR = 002\,151\,410\,152\,118$$

5) Bob chiffre chacun des blocs que l'on note **B** par la transformation $C = B^e \bmod n$ (où C est le bloc chiffré) :

$$C1 = 2^7 \bmod 5141 = 128$$
$$C2 = 151^7 \bmod 5141 = 800$$
$$C3 = 410^7 \bmod 5141 = 3761$$
$$C4 = 152^7 \bmod 5141 = 660$$
$$C5 = 118^7 \bmod 5141 = 204$$

On obtient donc le message chiffré C : 128 800 3761 660 204.

A faire vous même 9. Pas vraiment debuggé

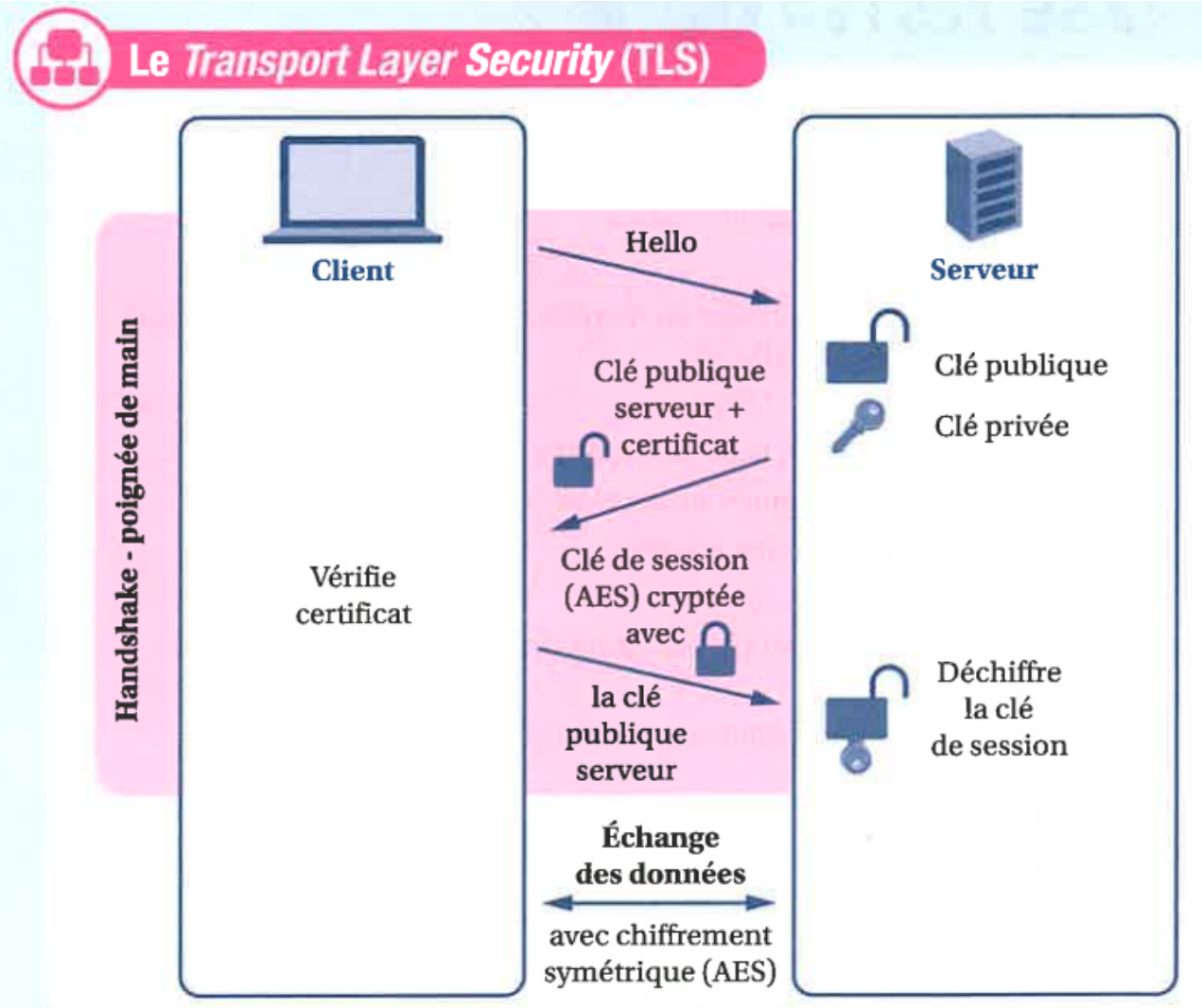
- Créez une fonction `car_to_fig` qui convertit une chaîne de caractères en une suite de chiffres comme décrit ci-dessus.

- Créez une fonction `chiffrement` qui convertit cette suite de chiffres en une autre suite de chiffres codée grâce à la clé publique
- Créez une fonction `dechiffrement` qui convertit cette suite de chiffres en une autre suite de chiffres décodée grâce à la clé privée
- Améliorez cette fonction en retrouvant le code initial.

Objectif : Décrire l'échange d'une clef symétrique en utilisant un protocole asymétrique pour sécuriser une communication HTTPS.

4 Le protocole https

Voir cours P. 265



P. 274 ex 8 – Objectif BAC

https://www.youtube.com/watch?v=S9pLqm8g4Bs