Annonces

- Les TP des **groupes 2 et 3** des jeudis 1er et 8 mai sont déplacés aux **lundis 28 avril et 5 mai de 10h45 à 12h45** en salle 2031.
- La date limite pour les projets est fixée au jeudi 22 mai soir.
- Les soutenances se dérouleront les 26 et 27 mai 2025.

XII - Échanges TCP sécurisés

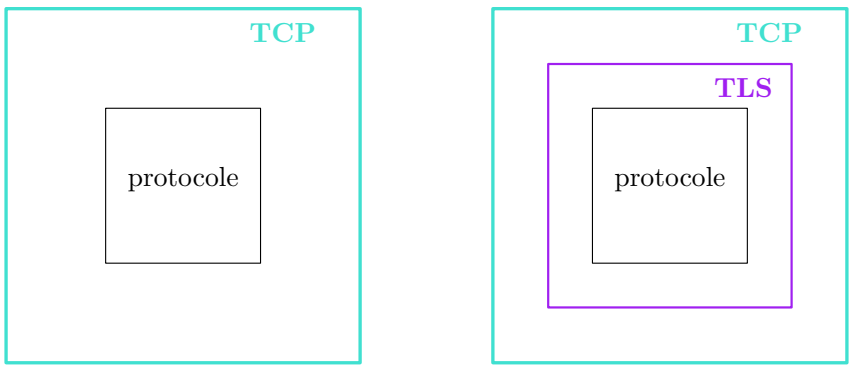
Données sécurisées

Protocole TLS

Protocole sécurisé :

- authentification : hôtes
- cryptage : messages échangés
- intégrité : données non modifiées

$\longrightarrow$ protocole **TLS** (Transport Layer Security) assurant la sécurité des connexions TCP.



Données sécurisées

Cryptage

$$\text{données} \xrightarrow{chiffrement+cle} \text{données cryptées} \xrightarrow{dechiffrement+cle} \text{données}$$

On utilise un algorithme de cryptage (cipher) avec une clé pour chiffrer les données. Il y a deux types de chiffrement :

- le **chiffrement symétrique** : la même clé est utilisée pour le chiffrement et le déchiffrement,
- le **chiffrement asymétrique** : deux clés distinctes sont respectivement utilisées pour le chiffrement et le déchiffrement.

Chiffrement symétrique : utilisé pour crypter les données du protocole

Comment les deux hôtes font pour avoir la même clé ?

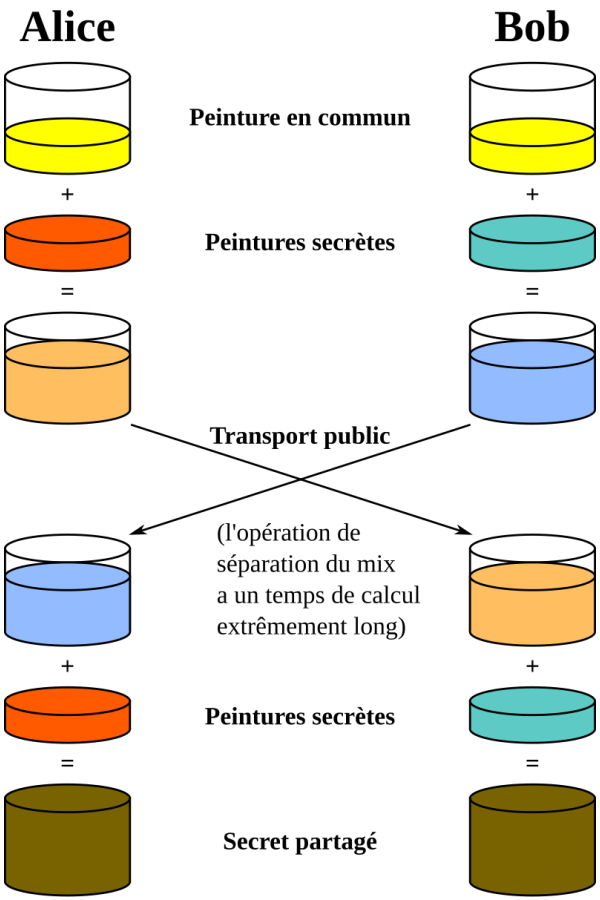
$\longrightarrow$ algorithme de Diffie-Hellman d'échange de clés

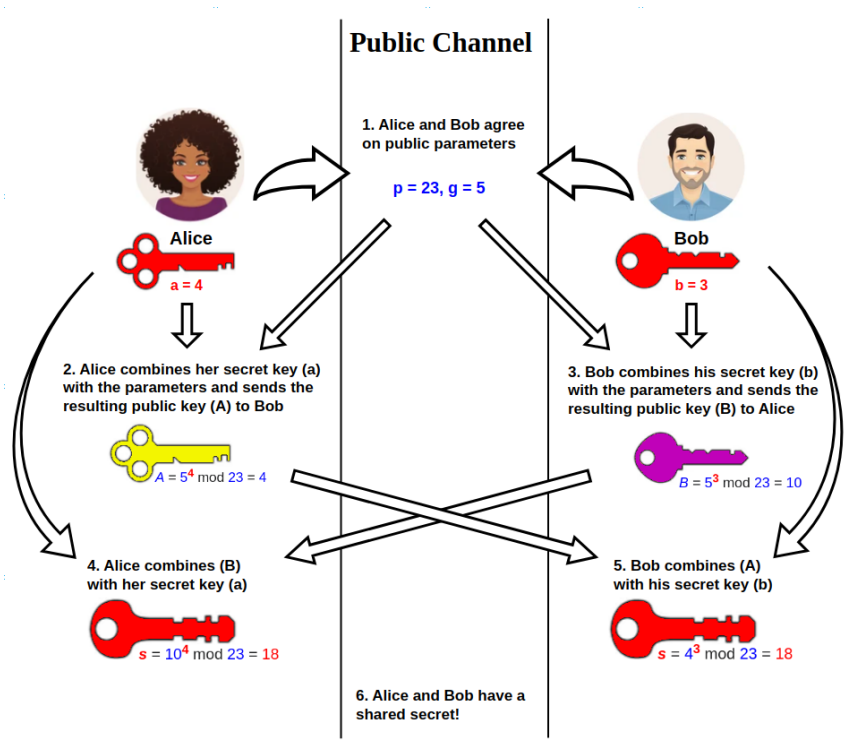
atout : protège contre la récupération par un autre hôte de la clé.

problème : ne protège pas contre l'attaque du « *man-in-the-middle* ».

Données sécurisées

Cryptage





sources : https ://commons.wikimedia.org

Chiffrement asymétrique : utilisé pour authentifier un hôte par le biais d'une signature digitale

données $\xrightarrow{chiffrement+cle1}$ données signées $\xrightarrow{dechiffrement+cle2}$ données vérifiées

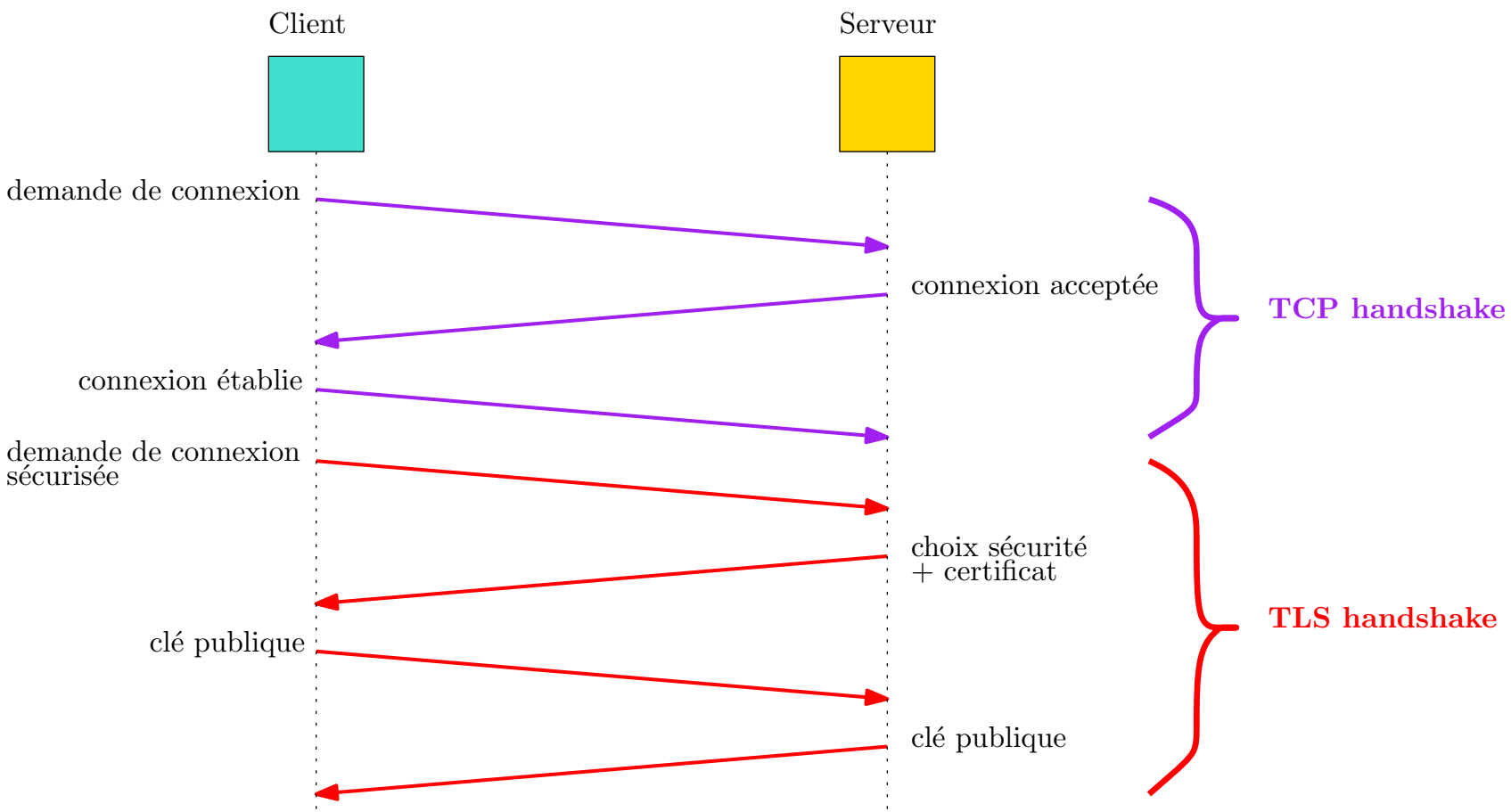
atout : protège contre l'attaque du « *man-in-the-middle* ».

problème : inefficace sur de grosses données.

1. Après que la connexion TCP est établie :
   - le client initie un « TLS handshake » avec spécifications : versions SSL/TLS, algorithmes de chiffrement disponibles, ...
   - le serveur répond en choisissant la version commune de SSL/TLS la plus récente et un agorithme de chiffrement parmis ceux proposés. S'il n'y a pas d'algorithme de chiffrement commun, la connexion TLS échoue.
2. Le serveur s'authentifie auprès du client en envoyant un certificat (signature digitale) : certificat signé par une autorité ou certificat signé par l'hôte lui-même.
3. Le serveur et le client procèdent à l'échange de clés.

Données sécurisées

Protocole TLS - TLS handshake



Pour compiler un programme utilisant OpenSSL, il faut ajouter les options de compilation `-lssl` et `-lcrypto`.

Un programme utilisant la bibliothèque *OpenSSL* pour mettre en place le protocole TLS, doit commencer par son initialisation :

```
SSL_library_init();                    // initialisation de OpenSSL
```

Il faut alors créer le contexte OpenSSL.

on appelle la fonction `SSL_CTX_new()`, retournant un objet `SSL_CTX`.

```
SSL_CTX *ctx = SSL_CTX_new(TLS_client_method());
if (!ctx) return 1;
```

`TLS_client_method()` indique que nous voulons négocier la version commune la plus récente de SSL/TLS.

Il faut alors établir la connexion TCP comme vous savez le faire qui crée une socket `sock`.

Données sécurisées

client SSL

On peut alors initier une connexion TLS.

On commence par créer un objet SSL à partir du contexte créé précédemment :

```
SSL *ssl = SSL_new(ctx);
if (!ssl) return 1;
```

Puis on initialise le nom du domaine du serveur auquel le client veut se connecter :

```
if (!SSL_set_tlsext_host_name(ssl, argv[1])) return 1;
```

Cette étape est nécessaire si l'hôte distant héberge plusieurs serveurs afin qu'il choisisse le bon certificat pour établir la connexion TLS.

Enfin le client demande une connexion SSL/TLS :

```
SSL_set_fd(ssl, sock); //lier la socket à l'objet SSL
if (SSL_connect(ssl) == -1) return 1;
```

Données sécurisées

client SSL

Une fois la connexion TLS établie, on communique avec le serveur en utilisant les fonctions `SSL_read()` et `SSL_write()` qui s'utilisent comme `read` et `write`.

```
sprintf(buf, "Hello le monde");
int ecrit = 0;
while(ecrit < strlen(buf))
        ecrit += SSL_write(ssl, buf+ecrit, strlen(buf)-ecrit);

int total = 0;
while(total < taille){
        int lu = SSL_read(ssl, buf+total, taille-total);
        if (lu == 0) {
                printf("\nConnexion interrompue.\n");
                break;
        }
        total += lu;
}
```

Pour Terminer la connexion, on invoque `SSL_shutdown()` qui ferme la connexion sécurisée en écriture avant d'invoquer `close`. Puis on libère la mémoire.

```
SSL_shutdown(ssl);
close(sock);
SSL_free(ssl);
SSL_CTX_free(ctx);
```

Après l'appel à `SSL_shutdown()` :

- la connexion TCP sous-jacente n'est pas fermée,
- on peut continuer à lire avec `SSL_read()` des données envoyées par le pair distant si celui-ci ne fait pas appel à `SSL_shutdown()`,
- l'appel à `SSL_read()` par le pair distant échoue en retournant la valeur `0`.

Après l'initialisation de la bibliothèque OpenSSL (voir slide 10), on crée un contexte SSL pour le serveur :

```
SSL_CTX *ctx = SSL_CTX_new(TLS_server_method());
if (!ctx) return 1;
```

Et on lui demande d'utiliser notre certificat (ici `serv.cert`) et clé privée (ici `serv.key`) pour les connexions sécurisées :

```
if (!SSL_CTX_use_certificate_file(ctx, "serv.crt" , SSL_FILETYPE_PEM) ||
    !SSL_CTX_use_PrivateKey_file(ctx, "serv.key", SSL_FILETYPE_PEM))
        return 1;
```

Ici `serv.cert` et `serv.key` doivent être au format PEM.

Données sécurisées

format PEM

fichier PEM (Private Enhanced Mail) : fichier texte contenant un ou plusieurs éléments en codage ASCII Base64, chacun avec des en-têtes et pieds de page en texte brut. Les éléments peuvent être un certificat, une chaîne de certificats incluant la clé publique, la clé privée, la clé privée et des certificats, ...

exemple :

```
-----BEGIN PUBLIC KEY-----
MCowBQYDK2VwAyEA7GRgvpY71p4B8mRjFp9Fh6+fzFm8/rLBTVOCgO1XMzc=
-----END PUBLIC KEY-----
```

Données sécurisées

serveur SSL

On met en place le serveur comme vous savez le faire avec socket serveur `sock` et socket client `sockclient`. Puis on établit ensuite la connexion sécurisée.
On crée pour cela un objet `SSL` à partir du contexte et de la socket client, puis on établit une connexion TLS :

```
SSL *ssl = SSL_new(ctx);
if (!ctx) return 1;

SSL_set_fd(ssl, sockclient);
if (SSL_accept(ssl) <= 0) return 1
```

Une fois la connexion TLS établie, on communique avec le client en utilisant les fonctions `SSL_read()` et `SSL_write()`.

Pour créer une clé privée (ici clé `ed25519`) au format PEM, on utilise la commande `openssl` :

```
$ openssl genpkey -algorithm ed25519 -out serv.key
```

et pour engendrer un certificat `serv.cert` signé par soi-même au format PEM :

```
$ openssl req -new -x509 -days 365 -key serv.key -out serv.crt
```

On peut aussi créer les deux en même temps avec clé rsa :

```
$ openssl req -x509 -newkey rsa:2048 -nodes -sha256 -keyout serv.key
-out serv.crt -days 365
```

Données sécurisées

gérer les erreurs

Pour obtenir les informations sur les erreurs produites lors des appels aux fonctions `SSL_...`, on peut appeler la fonction
`void ERR_print_errors_fp(FILE *fp)`

```
int nb = SSL_read(ssl, buf, SIZE_MESS);
if (nb == 0) {
 printf("\nConnexion interrompue\n");
 exit(1);
}
if(nb < 0) {
 fprintf(stderr, "erreur SSL_read\n");
 ERR_print_errors_fp(stderr);
 exit(1);
}
```

XIII - Signature électronique

La signature électronique permet d'authentifier la provenance d'un document (ou message).

Elle est basée sur la cryptographie asymétrique. Elle utilise deux algorithmes :

- le hachage pour obtenir une empreinte du message,
- le chiffrement/déchiffrement asymétrique de l'empreinte.

Pour le chiffrement/déchiffrement asymétrique il faut disposer d'une paire de clés privée, publique `(privK, pubK)`.

Signature électronique

signature

Si Alice veut signer un document ou message `mess` qu'elle souhaite transmettre à Bob :

1. Alice doit avoir/engendrer une paire de clés privée, publique `(privK, pubK)` et choisir une fonction de hachage `H` (avec l'algorithme `md5` ou `sha` par exemple),

2. elle transmet à Bob, sa clé publique `pubK` et la fonction de hachage `H`,

3. elle chiffre le message `mess` à l'aide de la fonction de hachage `H` et de sa clé privée `privK` :
   - elle extrait une empreinte `checksum` du message `mess` avec la fonction de hachage `H`,
   - elle chiffre l'empreinte `checksum` avec la clé privée `privK` $\Rightarrow$ elle obtient la signature `sig` du message `mess`,

4. elle transmet à Bob le message et sa signature `(mess, sig)`.

Signature électronique

authentification

À la réception du message et de la signature `(mess, sig)`, pour authentifier le message (vérifier que le message a été signé par Alice), Bob procède de la façon suivante :

1. il extrait l'empreinte `checksum` du message `mess`,
2. il déchiffre la signature `sig` avec la clé publique `pubK` $\Rightarrow$ il obtient une empreinte `checksum-bis`,
3. il compare `checksum` et `checksum-bis`. Le message est authentifié si les empreintes sont identiques.

Signature électronique

engendrer une paire de clés

Les clés doivent être stockées au format *PEM*.
On peut engendrer des clés au format PEM avec la bibliothèque OpenSSL.

En lignes de commandes :

- engendrer une clé RSA 2048 bits privée :
  `openssl genrsa -out priv-rsa.pem 2048`
- engendrer la clé publique associée :
  `openssl pkey -in priv-rsa.pem -pubout -out pub-rsa.pem`
- engendrer une clé ED2519 privée :
  `openssl genpkey -algorithm ed25519 -out priv-ed.pem`
- engendrer la clé publique associée :
  `openssl pkey -in priv-ed.pem -pubout -out pub-ed.pem`

Signature électronique

engendrer une paire de clés

On peut engendrer des clés au format PEM avec la bibliothèque OpenSSL.

En utilisant des fonctions C :

- engendrer une paire de clés RSA 2048 bits privée, publique :
  `EVP_PKEY *EVP_RSA_gen(unsigned int bits)`

- engendrer une paire de clés ED2519 privée, publique :
  `int EVP_PKEY_generate(EVP_PKEY_CTX *ctx, EVP_PKEY **ppkey)`

Signature électronique

engendrer une paire de clés RSA

```
EVP_PKEY *pkey = EVP_RSA_gen(2048);
if(pkey == NULL) exit(1);
```

Et pour sauvegarder les clés dans des fichiers au format PEM, pour la clé publique :

```
FILE *fd_pub = fopen("pub_rsa.pem", "w+");
int ret = PEM_write_PUBKEY(fd_pub, pkey);
fclose(fd_pub);
if(ret != 1) exit(1);
```

et pour la clé privée :

```
FILE *fd_priv = fopen(private_key_name, "w+");
ret = PEM_write_PrivateKey(fd_priv, pkey, NULL, NULL, 0, NULL, NULL);
fclose(fd_priv);
if(ret != 1) exit(1);
```

Signature électronique

engendrer une paire de clés ED25519

On commence par créer un contexte pour la clé :

```
EVP_PKEY_CTX *pctx = EVP_PKEY_CTX_new_id(EVP_PKEY_ED25519, NULL);
if(pctx == NULL) exit(1);
```

- Le 1er argument est une macro pour désigner le format des clés que l'on veut engendrer. Ici on veut engendrer des clés au format ED25519, on utilise donc la macro `EVP_PKEY_ED25519` (pour engendrer des clés RSA 2048, on peut passer le type `EVP_PKEY_RSA`).
- Le 2ème argument est à `NULL` si on veut utiliser la bibliothèque OpenSSL pour les opérations cryptographiques.
- `EVP_PKEY_CTX_new_id` alloue la place pour `pctx`.

Ensuite, on initialise le contexte avec l'algorithme de génération des clés :

```
if(EVP_PKEY_keygen_init(pctx) <= 0) exit(1);
```

Puis, on engendre la paire de clés :

```
EVP_PKEY *pkey = NULL;
if(EVP_PKEY_generate(pctx, &pkey) <= 0) exit(1);
```

On initialise `pkey` à `NULL` car dans ce cas, `EVP_PKEY_generate` alloue la place pour `pkey`.

On peut alors sauvegarder les clés dans des fichiers au format PEM avec exactement le même code que celui donné pour les clés RSA.

Si on a plus besoin de `pctx` ou de `pkey`, on libère la mémoire avec les appels :

```
EVP_PKEY_free(pkey);
EVP_PKEY_CTX_free(pctx);
```

Signature électronique

conversion clé PEM vers clé EVP

Les clés manipulées dans OpenSSL sont de type `EVP_PKEY`.
Pour convertir une clé au format PEM en une clé de type `EVP_PKEY` :

pour une clé privée

```
EVP_PKEY *pkey = NULL;
FILE *fp = fopen(privkey, "r");
if(fp == NULL) exit(1);

PEM_read_PrivateKey(fp, &pkey, NULL, NULL);
if(pkey == NULL) exit(1);

fclose(fp);
```

pour une clé publique

```
EVP_PKEY *pkey = NULL;
FILE *fp = fopen(privkey, "r");
if(fp == NULL) exit(1);

PEM_read_PUBKEY(fp, &pkey, NULL, NULL);
if(pkey == NULL) exit(1);

fclose(fp);
```

On initialise `pkey` à `NULL` afin que `PEM_read_PrivateKey` et `PEM_read_PUBKEY` alloue la place pour `pkey`.

Signature électronique

signer un message

Pour signer un message `msg` :

on commence pas créer un « message digest context » :

```
EVP_MD_CTX *mdctx;
if((mdctx = EVP_MD_CTX_create()) == NULL) exit(1);
```

Puis, on initialise ce contexte avec la clé privée :

```
if(EVP_DigestSignInit(mdctx, NULL, NULL, NULL, pkey) != 1) exit(1);
```

Puis, on initialise ce contexte avec la clé privée :

```
if(EVP_DigestSignInit(mdctx, NULL, NULL, NULL, pkey) != 1) exit(1);
```

- le 2ème paramètre (`EVP_PKEY_CTX **pctx`) est à `NULL` si on ne veut pas récupérer le contexte de la clé EVP,
- Le 3ème paramètre (`const EVP_MD *type`) attend un pointeur de fonction correpondant à la fonction de hachage à utiliser. Pour une clé au format :
  - RSA, on peut préciser la fonction de hachage, si on ne veut pas celle par défaut. Par exemple, `EVP_sha256()` ou `EVP_sha512()`,
  - ED2519, la fonction de hachage est SHA-512. On ne précise donc pas la fonction de hachage et on met l'argument à `NULL`.
- le 4ème paramètre (`ENGINE *e`) est à `NULL` si on veut utiliser la bibliothèque OpenSSL pour les opérations cryptographiques.

Signature électronique

signer un message

On va maintenant utiliser la fonction

```
int EVP_DigestSign(EVP_MD_CTX *ctx, unsigned char *sig,
size_t *siglen, const unsigned char *tbs, size_t tbslen);
```

pour la signature, qui :

- si `sig` n'est pas `NULL`, signe les données `tbs` de longueur `tbslen` et met le résultat dans `sig` de longueur `siglen`,
- sinon, la taille maximale nécessaire à la signature des données `tbs` est mise dans `siglen`.

Signature électronique

signer un message

On récupère tout d'abord la longueur `slen` de la signature, pour allouer la mémoire pour la signature, en faisant une première fois appel à `EVP_DigestSign` :

```
size_t slen;
unsigned char *sig;
if(EVP_DigestSign(mdctx, NULL, &slen, msg,  strlen(msg)) != 1) exit(1);
if(!(sig = malloc(sizeof(*sig) * (slen)))) exit(1);
```

On peut maintenant signer le message `msg` :

```
if(EVP_DigestSign(mdctx, sig, &slen, msg, msg_len) != 1) exit(1);
```

On libère les variables allouées lorsqu'on ne les utilise plus :

```
EVP_MD_CTX_free(mdctx);
free(sig);
```

Signature électronique

authentifier un message

Pour authentifier un message `msg` :

on commence pas créer un « message digest context » et l'initialiser avec la clé publique :

```
EVP_MD_CTX *mdctx;
if((mdctx = EVP_MD_CTX_create()) == NULL) exit(1);
if(EVP_DigestVerifyInit(mdctx, NULL, NULL, NULL, pubkey) != 1) exit(1);
```

On peut alors faire la vérification :

```
if(EVP_DigestVerify(mdctx, sig, slen, msg, strlen(msg)) == 1)
  printf("authentification réussie\n");
else
  printf("échec de l'authentification\n");
```

où `sig` est la signature avec sa longueur `slen` et `msg` est le message d'origine (avant hachage et signature).