NOTES SUR OPENSSL : PARTIE I

par

Ainigmatias Cruptos

Résumé. — Nous commençons avec cette partie I, une suite de présentations de openSSL, et des standards importants, comme la certification X.509, qui y sont utilisés.

1. Généralités

1.1. La place du protocole TLS. — Le protocole **TLS**, *Transport Layer Security*, qui a pris la suite de **SSL**, *Secure Socket Layer*, est un protocole de sécurisation des échanges sur Internet. De ce fait, c'est une construction très complexe, qui doit régler beaucoup de problèmes et en premier point la relation client-serveur dans divers types d'applications.

Le protocole TLS se situe au-dessus de la couche réseau **TCP** et au-dessous des applications. Il peut en principe sécuriser toute application réseau **TCP/IP**.

Il n'est donc pas au même niveau qu'un protocole comme **IPSec** par exemple, qui intervient au niveau de la couche IP pour sécuriser l'acheminement des paquets de données.

Un certain nombre d'autres protocoles agissant au même niveau, se préoccupent aussi de la sécurisation de certains services de l'Internet.

Mots clefs. — cryptographie, protocole cryptographique, SSL,TLS, signature, chiffrement, base64, certificats.

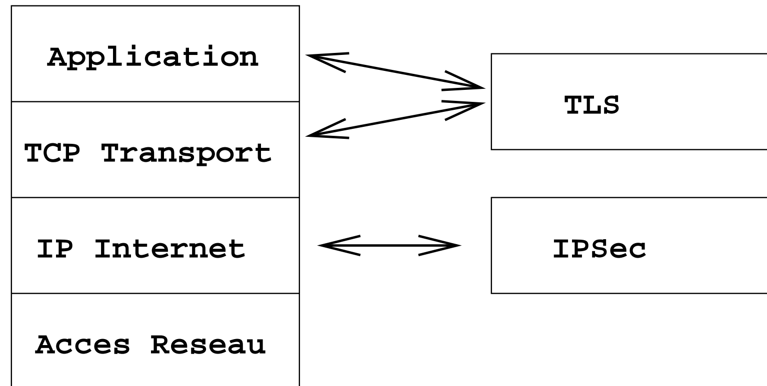

FIGURE 1. Place de TLS dans le modèle TCP/IP

Parmi eux on peut citer **SSH**, *Secure Shell*, spécialisé dans la connexion à distance sur un compte, des protocoles de sécurisation de la messagerie comme **PEM**, *Privacy Enhanced Mail*, **S/Mime**, *Secure Multipurpose Mail Extension* et enfin des concurrents plus directs comme **PGP**, *Pretty Good Privacy* et **SET**, *Secure Electronic Transaction*. Certains d'entre eux sont supportés par TLS, d'autres sont tout à fait indépendants, voire concurrents de TLS. À vrai dire, comme on va le voir tout au long de cette suite d'exposés, la situation est relativement embrouillée. Divers protocoles interviennent à des niveaux à peu près identiques et inter-agissent : certains utilisent quelques parties d'autres, certains utilisent des standards communs etc. Nous essaierons autant que faire se peut d'éclaircir les rapports entre ces protocoles et les divers standards qu'ils utilisent.

1.2. Les implémentations de TLS. — Il existe différentes implémentations de SSL/TLS. On peut citer en particulier : **OpenSSL** (qu'on va regarder plus particulièrement), **GnuTLS**, **NSS**, **YaSSL**.

1.3. Les standards utilisés par TLS. — Le standard TLS lui-même est défini dans la note **RFC 4346** suivie des révisions **RFC 4366**, **RFC 4680**, **RFC 4681**. Ces notes, qui définissent TLS Version1.1, se servent d'autres RFC, pour préciser leurs termes et décrire exactement les diverses parties du protocole. À titre documentaire, nous fournissons ici

tous les RFC appelés par les RFC définissant TLS, dans le corps du texte, à l'exception donc de ceux qui ne sont cités qu'en bibliographie.

RFC 2119	Key words for use in RFCs to Indicate Requirement Levels
RFC 3447	Public-Key Cryptography Standards (PKCS) #1 : RSA Cryptography Specifications Version 2.1
RFC 2313	PKCS #1 : RSA Encryption Version 1.5
RFC 2434	Guidelines for Writing an IANA Considerations Section in RFCs
RFC 3546	Transport Layer Security (TLS) Extensions
RFC 2104	HMAC : Keyed-Hashing for Message Authentication
RFC 1321	The MD5 Message-Digest Algorithm
RFC 3280	Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile
RFC 3490	Internationalizing Domain Names in Applications (IDNA)

TABLE 1. Les divers RFC appelés par TLS

1.4. Le fonctionnement général de TLS. — Pour bien comprendre le but de TLS, citons le début de la note RFC 4346 : *"The TLS protocol provides communications security over the Internet. The protocol allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery"*.

Le protocole est composé de deux parties principales :

(1) La partie de plus bas niveau, qui se situe juste au dessus de la couche transport (TCP) appelée *TLS Record Protocol*. Ce protocole est en charge du **transport encapsulé** de protocoles de plus haut niveau, de telle sorte que :

(a) La connexion est confidentielle, le chiffrement étant assuré par de la cryptographie symétrique. La clé secrète est une clé de session,

c'est-à-dire qu'elle est reconstruite à chaque connexion. La négociation de la clé secrète est prise en charge par un autre protocole.

(b) La connexion est intègre, un MAC (*Message Authentification Code*), construit à l'aide de fonctions de hachages étant utilisé à cet effet.

En conclusion cette couche doit assurer la **confidentialité** et **l'intégrité**.

(2) La partie de plus haut niveau, *TLS Handshake Protocol* qui doit avoir les propriétés suivantes :

(a) Permettre l'authentification des participants grâce à de la cryptographie à clé publique (signature).

(b) Assurer la négociation de la clé secrète du Record Protocol, et assure la confidentialité de la clé négociée.

(c) Rendre la négociation fiable, c'est-à-dire que toute attaque tentant de modifier les termes de la négociation, doit être détectée.

En conclusion, cette partie assure **l'authentification** des participants ainsi que la **fiabilité et la confidentialité de la négociation** de la clé secrète.

1.5. Les certificats. — Une des parties importantes du Handshake Protocol est **l'authentification**. Si on veut une authentification forte, qui aille jusqu'à **l'identification** (c'est-à-dire non seulement l'authentification d'une clé, mais aussi du couplage d'une clé et d'un individu dont l'identité sociale est reconnue), les clés publiques des participants doivent être contrôlées et signées par une autorité. Ceci se réalise en pratique par l'**émission de certificats**. À vrai dire, il y a deux voies principales pour réaliser ceci. La première définit une structure très hiérarchisée, avec une autorité de certification racine, qui donne naissance à un arbre d'autorités intermédiaires, jusqu'aux utilisateurs qui sont les feuilles de l'arbre ; c'est le principe des certificats **X.509**. La deuxième, dont le modèle est **pgp**, ne définit pas de hiérarchie, mais plutôt une structure de confiance entre les utilisateurs : le certificat d'un utilisateur est signé par tous ses amis, lesquels ont eux mêmes des certificats signés par leurs amis *etc* ; Il est clair qu'aucune de ces deux formes n'est universellement satisfaisante. La politique de certification d'une banque par exemple, pour les échanges

entre employés, et entre employés et clients, ou celle d'une grande administration, ne peut pas ressembler à celle qui préside à des échanges entre individus divers sur Internet.

Bien que la certification X.509 ne soit pas obligatoire dans TLS, elle est citée comme solution probable et implémentée par toutes les réalisations de TLS. Remarquons toutefois que gnuTLS donne le choix entre l'utilisation de certificats X.509 ou de certificats PGP (et de sa version libre GPG).

Nous reviendrons plus en détail dans la suite sur les certificats X.509, qui sont au centre de la question, et nous renverrons à une note sur PGP (ou gnupg) pour les certificats de ce type.

2. OpenSSL

OpenSSL réalise une implémentation du protocole SSL/TLS. Ce faisant, OpenSSL met en place une boîte à outils très complète, qui va des briques de base d'un système cryptographique, comme par exemple la réalisation de primitives de hachage (SHA1, SHA256, *etc.*), de primitives de chiffrement asymétrique (RSA-OAEP), de primitives de signature (RSA,DSS), de primitives de chiffrement symétriques (AES128 en mode CBC *etc*), jusqu'à des protocoles complexes (mise en place d'une autorité de certification, protocole de requête et de signature des certificats, interface avec le fonctionnement des serveurs).

OpenSSL offre donc d'une part **une bibliothèque** de procédures en **C**, apte à permettre le développement d'applications sécurisées par le protocole SSL/TLS, d'autre part un accès à des fonctionnalités de sécurité au niveau de la **ligne de commande**. C'est cette dernière possibilité que nous regarderons ici, en partant des fonctions les plus simples, jusqu'aux plus complexes.

3. Opérations de base depuis la ligne de commande

3.1. Partons d'une simple biclé RSA. — Nous allons dans un répertoire d'essai créer une biclé RSA de 1024 bits (en travail réel il vaut

mieux 2048 bits), dont la clé privée sera protégée en la chiffrant à partir d'une **passphrase** et du standard **AES128**.

```
$ openssl genrsa -aes128 -out ma_bicle.pem 1024
```

La réponse à cette commande est le dialogue :

```
Generating RSA private key, 1024 bit long modulus
......++++++
........++++++
e is 65537 (0x10001)
Enter pass phrase for ma_bicle.pem:helloworld
Verifying - Enter pass phrase for ma_bicle.pem:helloworld
```

Bien entendu, j'ai fait apparaître la passphrase (trop courte) « helloworld » que j'ai utilisée pour permettre de faire des essais sur cette biclé. En réalité, la passphrase entrée au clavier n'apparaît pas.

Le fichier obtenu, que j'ai appelé *ma_bicle.pem*, est au format défini par PEM (**RFC 1421, 1422, 1423, 1424**). C'est une structure dont la syntaxe est définie en ASN1 par :

```
RSAPrivateKey ::= SEQUENCE {
    version           Version,
    modulus           INTEGER,  -- n
    publicExponent    INTEGER,  -- e
    privateExponent   INTEGER,  -- d
    prime1            INTEGER,  -- p
    prime2            INTEGER,  -- q
    exponent1         INTEGER,  -- d mod (p-1)
    exponent2         INTEGER,  -- d mod (q-1)
    coefficient       INTEGER,  -- (inverse of q) mod p
    otherPrimeInfos   OtherPrimeInfos OPTIONAL
}
```

sérialisée en suite d'octets grâce à DER, *Distinguished Encoding Rules* (ITU-T X.690 ou ISO/IEC 8825-1), pour ensuite être encodée en utilisant *Base64*, et enfin encapsulée avec un en-tête. Ce fichier est donc un fichier ASCII que voici :

```
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: AES-128-CBC,7DB04F39ECA809167647F4DA5FCF9862

wSiR/0DPHH2u3QWusITsQZBcciIJhpcseWFB3vbsZONi4/coY8ZNJ6zV7ioFzKRj
4vnHNrz1EGmOXOLgEzoTCo3HpKp1JNmdfli/mlSKIUJq/PDnFtFVBfJvbs6mXJCv
N4gJbdPFq7GMDBo1BcttYvk6B43ok5Df0lP5F7hDU+ugm1uguLe0XYFf4nZiUKAu
ZmzXTvL2JLczRW3jF4K81vczU5DLTqEqQVa9iXvznQG+0SrqiC4z4EjdBT+ewCBZ
8lfCp58G9O8Kr7WqrziodAZYG4YlhvexZNEEN0h/BrTUm0Z8m8MeUW4wgM02o1S1
LdKEZswI5gYVuy6QTkNGm4zB4hqqxtxWWwJqaWExx01SRazQzGZyCkHQiazB6Ehp
NApUYs1860WVI377I0F3R+Ij8kw5W9N8pAz7QA/WHi4FU1Fy5BDdwMjxeFS5zC4F
m+vBCp/6myBIqbI2VKh9IuprU19mP3v46DAJpjWGc4ovkIVOVqLHPy5StodnjjGm
14l+os3FlgcW2wdyvwEGQ8WVVLcdri/EaKmzSAID0Z2Ll4Wwc7YDe4uf4XLCrKgr
GxugLTJX/+IV0ZG113g8oYqSZGJQgifg+WFdjrF2TSjWGKpndYS8hYueBmiNYbiu
dSvr2ea6kJErnACCrrkCvaGBgbEe377vfx5a1Ueb8r6fS9iYAMT0mPdR1R++osXp
syNGC3B22h+On3oax4Pif+++4tj9kf5pP8O066s7/5ONli/s/omgXPMcUcEJZ7mJ
0WqZkyBsvssmWFWmjU47IMytqgTPG4FGnsI1T/qYQ94=
-----END RSA PRIVATE KEY-----
```

À la lecture du *header* de ce fichier, on apprend qu'il s'agit d'une clé privée RSA, c'est-à-dire en fait une biclé RSA complète, que cette biclé est protégée en ayant été chiffrée par AES 128 bits, utilisé en mode **CBC**, la valeur initiale IV étant en hexadécimal : $7DB04F39ECA809167647F4DA5FCF9862$. Si on veut pouvoir comprendre le reste du contenu de ce fichier, on peut en extraire les informations et les afficher en format texte compréhensible grâce à la commande :

```
$ openssl rsa -text -in ma_bicle.pem -out ma_bicle.txt
```

La réponse est :

```
Enter pass phrase for ma_bicle.pem:
writing RSA key
```

On remarque que la passphrase (helloworld) est demandée, de manière à pouvoir déchiffrer la clé privée. On obtient alors le fichier texte *ma_bicle.txt* sous la forme suivante :

```
Private-Key: (1024 bit)
modulus:
    00:ab:a1:cd:4d:69:5d:9a:fa:99:38:2a:c0:72:71:
    28:83:d3:b8:ed:03:00:76:3a:72:37:4a:ef:43:35:
    dd:f6:a3:8a:b8:a4:a8:39:2f:78:d7:7a:b4:43:6e:
    9f:08:a7:39:dd:3d:92:67:4c:3e:fb:c1:76:00:d8:
    02:a6:f6:8c:4c:c9:7c:f5:e0:c4:9d:25:d9:bc:18:
    07:68:39:63:da:9f:62:75:26:e2:5a:c1:88:dd:17:
    4f:4c:3b:d4:27:74:11:64:56:21:6c:58:7f:c3:e7:
    2f:46:8f:3f:63:14:7a:f4:48:ae:a7:de:eb:54:fe:
    ec:09:76:29:01:db:4c:a6:13
publicExponent: 65537 (0x10001)
privateExponent:
    39:f8:94:d7:a8:d7:2d:19:a7:d7:08:d9:a7:ce:00:
    d9:46:12:18:3c:03:53:eb:b9:d8:63:3f:1d:7c:7c:
    54:6a:38:d8:d5:04:dd:0b:e3:cd:24:6c:ee:b9:d6:
    8d:9d:ae:35:c4:2e:47:25:c1:c0:57:3f:fc:58:f9:
    cc:5b:4a:57:b7:a6:cb:8a:f7:27:86:84:47:7d:0c:
    1a:0e:58:39:fc:23:f0:c7:c5:d4:d8:67:d3:3a:f4:
    aa:87:5a:0c:e9:b1:85:be:4f:96:98:88:ea:da:d7:
    67:e9:a3:27:d5:3a:5d:ad:26:14:ac:bf:47:71:af:
    37:e8:72:a3:71:4a:28:c1
prime1:
    00:d9:06:6c:28:7c:c4:d3:2b:c2:17:3d:01:d1:a1:
    b5:77:29:f1:2f:e5:84:98:11:6e:58:a1:68:91:5a:
    a4:1e:7c:f7:b2:2f:df:76:23:7b:f6:13:ee:06:b3:
    b8:24:90:64:f5:54:f5:24:f0:5d:57:5d:86:e9:31:
    e4:13:1d:26:9f
prime2:
    00:ca:74:77:32:18:d1:0a:78:86:0f:bc:b7:b3:81:
    49:18:b3:99:f0:fd:6e:48:97:66:f0:eb:12:cd:db:
    41:9a:3f:38:8d:dc:c4:f3:04:ff:15:fe:e9:20:e3:
```

```
        8b:32:df:e2:f7:e5:ef:92:3e:75:31:0e:2a:f0:64:
        cc:81:40:50:0d
    exponent1:
        36:27:1d:1b:e9:2e:2f:c7:2e:72:1f:fd:f1:32:19:
        96:b7:77:80:4d:14:0d:e1:e3:97:e8:06:b0:a6:5e:
        67:61:25:69:67:fa:a9:7e:e7:32:9a:fc:7c:dd:a1:
        68:36:43:8a:d7:fd:27:8f:76:ab:13:22:53:d3:e8:
        26:40:d2:df
    exponent2:
        68:39:bb:1f:6c:4d:39:d1:c0:5a:9b:b2:0a:d4:75:
        18:25:66:0b:fd:bc:67:dc:a1:df:47:75:bf:ca:af:
        a4:44:05:ce:6a:a8:6d:df:d5:9e:b2:43:bc:6d:c8:
        3c:a1:ac:0c:29:30:c3:9e:29:e0:de:45:56:ca:dd:
        c9:70:9d:85
    coefficient:
        0b:34:3e:39:ce:e9:00:aa:0e:d5:0a:89:71:35:77:
        d9:c9:ad:d6:d1:06:b6:d7:60:1d:fb:6b:61:ea:d2:
        a5:26:b2:90:c2:80:4d:9f:eb:f6:3d:06:b0:46:98:
        41:39:69:b9:2a:5e:f3:72:3b:39:84:f1:f5:31:66:
        af:20:cd:01
    -----BEGIN RSA PRIVATE KEY-----
    MIICWwIBAAKBgQCroc1NaV2a+pk4KsBycSiD07jtAwB2OnI3Su9DNd32o4q4pKg5
    L3jXerRDbp8IpzndPZJnTD77wXYA2AKm9oxMyXz14MSdJdm8GAdoOWPan2J1JuJa
    wYjdF09MO9QndBFkViFsWH/D5y9Gjz9jFHr0SK6n3utU/uwJdikB20ymEwIDAQAB
    AoGAOfiU16jXLRmn1wjZp84A2UYSGDwDU+u52GM/HXx8VGo42NUE3QvjzSRs7rnW
    jZ2uNcQuRyXBwFc//Fj5zFtKV7emy4r3J4aER30MGg5YOfwj8MfF1Nhn0zr0qoda
    DOmxhb5PlpiI6trXZ+mjJ9U6Xa0mFKy/R3GvN+hyo3FKKMECQQDZBmwofMTTK8IX
    PQHRobV3KfEv5YSYEW5YoWiRWqQefPeyL992I3v2E+4Gs7gkkGT1VPUk8F1XXYbp
    MeQTHSafAkEAynR3MhjRCniGD7y3s4FJGLOZ8P1uSJdm8OsSzdtBmj84jdzE8wT/
    Ff7pIOOLMt/i9+Xvkj51MQ4q8GTMgUBQDQJANicdG+kuL8cuch/98TIZlrd3gE0U
    DeHjl+gGsKZeZ2ElaWf6qX7nMpr8fN2haDZDitf9J492qxMiU9PoJkDS3wJAaDm7
    H2xNOdHAWpuyCtR1GCVmC/28Z9yh30d1v8qvpEQFzmqobd/VnrJDvG3IPKGsDCkw
    w54p4N5FVsrdyXCdhQJACzQ+Oc7pAKoO1QqJcTV32cmt1tEGttdgHftrYerSpSay
    kMKATZ/r9j0GsEaYQTlpuSpe83I7OYTx9TFmryDNAQ==
    -----END RSA PRIVATE KEY-----
```

La dernière partie est la clé privée déchiffrée au format PEM, c'est-à-dire en fait la biclé complète (le module, les divers exposants publics et privés etc.) que nous avions affiché chiffrée précédemment. Si on ne tient pas à afficher cette partie, il suffit de rajouter l'option « -noout ».

On remarque que l'exposant public e est $2^{16}+1$ (4[e]nombre de Fermat, qui est un nombre premier et qui est un exposant commode. Par l'algorithme *square and multiply*, $a^e \mod n$ ne demande que 17 multiplications modulo n.

On pourrait générer un fichier au format PEM qui ne contient que la partie publique de la biclé avec la commande :

```
$ openssl rsa -in ma_bicle.pem -pubout -out ma_bicle_partpub.pem
```

qui donne la réponse suivante :

```
Enter pass phrase for ma_bicle.pem:
writing RSA key
```

La sortie est le fichier PEM *ma_bicle_partpub.pem* :

-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCroc1NaV2a+pk4KsBycSiD07jt
AwB2OnI3Su9DNd32o4q4pKg5L3jXerRDbp8IpzndPZJnTD77wXYA2AKm9oxMyXz1
4MSdJdm8GAdoOWPan2J1JuJawYjdF09MO9QndBFkViFsWH/D5y9Gjz9jFHr0SK6n
3utU/uwJdikB20ymEwIDAQAB
-----END PUBLIC KEY-----

qu'on peut voir sous forme compréhensible texte avec la commande :

```
$ openssl rsa -pubin -in ma_bicle_partpub.pem -text -noout
-out ma_bicle_partpub.txt
```

dont la sortie est le fichier texte *ma_bicle_partpub.txt* suivant :

```
Modulus (1024 bit):
    00:ab:a1:cd:4d:69:5d:9a:fa:99:38:2a:c0:72:71:
    28:83:d3:b8:ed:03:00:76:3a:72:37:4a:ef:43:35:
    dd:f6:a3:8a:b8:a4:a8:39:2f:78:d7:7a:b4:43:6e:
```

```
    9f:08:a7:39:dd:3d:92:67:4c:3e:fb:c1:76:00:d8:
    02:a6:f6:8c:4c:c9:7c:f5:e0:c4:9d:25:d9:bc:18:
    07:68:39:63:da:9f:62:75:26:e2:5a:c1:88:dd:17:
    4f:4c:3b:d4:27:74:11:64:56:21:6c:58:7f:c3:e7:
    2f:46:8f:3f:63:14:7a:f4:48:ae:a7:de:eb:54:fe:
    ec:09:76:29:01:db:4c:a6:13
Exponent: 65537 (0x10001)
```

dont on ne s'étonnera pas qu'il ne contienne que le module et l'exposant public, et qui correspond tout à fait à la description ASN1 :

```
RSAPublicKey ::= SEQUENCE {
    modulus           INTEGER,  -- n
    publicExponent    INTEGER   -- e
}
```

3.2. Construire un nombre occasionnel au hasard. — La construction de nombres aléatoires est importante en cryptographie : construction de clés de session ou de nombres servant à initialiser des modes de chiffrement, ou encore servant à rendre un chiffrement aléatoire par un encodage aléatoire des données (voir OAEP par exemple) ou enfin permettant de construire un masque à la volée pour du chiffrement à flot. D'un autre côté, les façons de réaliser un tirage aléatoire peuvent utiliser un phénomène physique ou, de manière opposée, utiliser une suite pseudoaléatoire.

Il faut bien voir que le problème de construire une clé secrète au hasard, clé qui va servir pour une session dont la durée est relativement longue, disons plusieurs minutes, est tout à fait différent de la construction d'un masque servant à chiffrer à flot un flux de données. Dans le premier cas on doit construire un **nombre occasionnel**, dans le deuxième cas, on doit disposer d'une suite dont les termes doivent être calculés très rapidement.

Il est aussi indispensable de prendre en compte le contexte matériel dans lequel se situe le problème. Doit on travailler en hardware par exemple pour un circuit dédié, ou au contraire développer une application dans un langage de programmation de haut niveau destinée à s'exécuter

sur des machines généralistes. Bien entendu, la réponse à cette question influe fortement sur les solutions retenues.

Nous allons décrire ici une commande très simple, sous le système d'exploitation Linux, qui permet de construire un nombre occasionnel en utilisant le périphérique */dev/random*. La lecture de */dev/random* renvoie une suite imprévisible d'octets, pourvu qu'on laisse le temps à l'accumulateur d'aléas, de construire une reserve suffisante d'octets calculés en fonction d'un certain nombre de paramètres physiques de l'ordinateurs (souris, clavier, horloge). L'idée est de bien utiliser cette quantité d'aléas en faisant subir un hachage à la suite d'octets retournée. On peut à la place de */dev/random* utiliser la version non bloquante */dev/urandom* avec le risque d'une quantité d'aléas moindre. La note **RFC 1750** (*Randomness Recommendations for Security*) précise et donne des indications sur les notions de sécurité concernant les nombres aléatoires.

```
$ head -c 128 /dev/random | openssl dgst -sha256
   -binary > seed.bin
```

On peut voir le résultat avec od :

```
$ od -t x seed.bin
0000000 6026882e a19a8e6b feaabdc3 8bc4eef9
0000020 c441e65d 213adeaf bb383900 f7eca25f
```

on voit les 32 octets (256 bits qui ont été renvoyés).

Si on veut que le fichier renvoyé soit en base64 :

```
$ head -c 128 /dev/random | openssl dgst -sha256
  -binary | openssl enc -base64 > seed.b64
```

Dans ces commandes, on a commencé par extraire 128 octets (c'est-à-dire 1024 bits) du périphérique /dev/random par « head -c 128 /dev/random », puis la commande « openssl dgst » est utilisée avec les options « -sha256 -binary ». pour effectuer un hachage, ici avec **sha256**, la sortie étant binaire (256 bits).

Si on rajoute dans le *pipe* la commande « openssl enc » avec l'option « -base64 », la sortie binaire précédente est encodée en base64 pour fournir une sortie ASCII.

3.3. Chiffrer et déchiffrer. — La commande « openssl enc » permet de chiffrer un texte clair, de déchiffrer un texte chiffré, d'encoder un fichier en base64.

3.3.1. Chiffrer un fichier. — Le chiffrement d'un fichier s'effectue par la commande :

```
$ openssl enc -e -in message.clair
  -out message.crypt -aes-128-cbc
```

qui a pour réponse :

```
enter aes-128-cbc encryption password:
Verifying - enter aes-128-cbc encryption password:
```

On remarquera que le fichier chiffré est un peu plus long que le fichier clair, en raison du découpage en blocs et du rajout de la valeur initiale du mode cbc. Si le texte clair contient $16k + l$ octets (où $0 \leq l < 16$) alors le texte chiffré contient $16(k + 2)$ octets (la complétion du bloc a lieu y compris si $l = 0$). La clé secrète quant-à elle est construite à partir du mot de passe fourni.

3.3.2. Déchiffrer un fichier chiffré. — Pour déchiffrer il suffit alors d'utiliser la commande :

```
$ openssl enc -d -in message.crypt  -out message_orig.clair
  -aes-128-cbc
```

avec pour réponse :

```
enter aes-128-cbc decryption password:
```

Remarquons qu'il faut de nouveau préciser le chiffrement utilisé. À ce niveau assez bas du système, le mode de chiffrement n'est pas sauvegardé avec le texte chiffré. Le fichier message_orig.clair doit être identique au fichier message.clair.

3.3.3. Utilisation de l'encodage base64. — Si on désire que le texte chiffré soit encodé en base64 on lancer la commande :

```
$ openssl enc -e -in essai.txt  -out essai.crypt
  -aes-128-cbc -base64
```

qui renvoit le dialogue :

```
enter aes-128-cbc encryption password:
```

```
Verifying - enter aes-128-cbc encryption password:
```

La commande pour déchiffrer est alors :

```
$ openssl enc -d -in essai.crypt  -out essai_orig.clair
  -aes-128-cbc -base64
```

qui ouvre le dialogue :

```
enter aes-128-cbc decryption password:
```

Remarquons que l'option d'encodage en base64 peut être utilisée sans chiffrement ou déchiffrement : c'est alors une simple transformation d'un fichier binaire en sa forme base64 et réciproquement :

```
$ openssl enc -e -in essai.bin  -out essai.b64 -base64
```

pour la transformation en base64 et

```
$ openssl enc -d -in essai.b64  -out essai_orig.bin -base64
```

pour retrouver le fichier original.

3.4. Signer. — Nous allons signer avec RSA un message qui se trouve dans le fichier message.txt. Pour cela avant toute chose il faut disposer de :

(1) une biclé RSA pour celui qui doit signer. On la créera dans le fichier ma_bicle.pem ainsi que décrit dans un paragraphe antérieur :

```
$ openssl genrsa -aes128 -out ma_bicle.pem 1024
```

(2) la clé publique de cette biclé pour ceux qui doivent vérifier. On l'extraira ainsi qu'indiqué dans le paragraphe sur les biclés RSA dans un fichier ma_bicle_partpub.pem :

```
$ openssl rsa -in ma_bicle.pem -pubout
  -out ma_bicle_partpub.pem
```

Le propriétaire de la clé privée va signer un haché (ici avec sha256) du fichier message.txt :

```
$ openssl dgst -sha256 -sign ma_bicle.pem
  -out message.txt.sgn message.txt
```

La signature se trouve dans le fichier message.txt.sgn.

Un correspondant qui dispose :

(1) de la clé publique de celui qui a signé

(2) du fichier message.txt
(3) du fichier de signature message.txt.sgn
(4) du système de hachage utilisé

peut vérifier la signature :

```
$ openssl dgst -sha256 -verify ma_bicle_partpub.pem
  -signature message.txt.sgn message.txt
```

La réponse est OK si la signature est bonne. Là encore, à ce niveau assez bas d'utilisation, l'information n'est pas toute disponible dans les fichiers créés. Par exemple le système de hachage utilisé doit être précisé à la vérification.

4. La suite au prochain numéro

Nous avons vu une première approche de OpenSSL. Nous avons montré l'utilisation de quelques commandes **isolées** d'assez bas niveau. Dans un deuxième article ultérieur nous verrons une utilisation **organisée** de OpenSSL, expliquant le fonctionnement général d'un système à base de certificats X.509. Nous verrons comment créer et gérer une autorité de certification, comment faire des demandes de certificats signés par l'autorité. Comment utiliser ces certificats dans certaines applications comme Mozilla par exemple.

5. Documents

Outre les RFC cités dans le texte, voici quelques documents que le lecteur pourrait consulter :

(1) la fichecrypto_302.pdf qui se trouve en téléchargement sur le site http ://www.acrypta.fr qui décrit plus en détail la forme de la biclé RSA dans openSSL.

(2) le texte gnupg_armure.pdf qui se trouve aussi en téléchargement sur le site http ://www.acrypta.fr et qui précise en particulier le fonctionnement de base64.

(3) Le OpenSSL Command-Line HOWTO de Paul Heinlein sur la page http ://www.madboa.com/geek/openssl/ qui constitue une très bonne

description très bien faite des commandes openSSL. Ce HOWTO m'a bien servi.

---

7 juin 2008

A. Cruptos, Association ACrypTA. • *E-mail :* acrypta@acrypta.fr