ECOLE ROYALE MILITAIRE

156e Promotion Polytechnique

Lieutenant-Général Baron de GREEF

Année académique 2005 – 2006

3ème épreuve

Signatures électroniques dans les applications INTERNET

Par le

Sous-lieutenant élève
Rodolphe CARDON DE LICHTBUER

Mémoire introduit pour l'obtention du diplôme du grade d'ingénieur civil polytechnicien

Sous la direction de : Major Mees
Chargé de cours militaire

Bruxelles, 2006

Avant propos

La dématérialisation des documents signés offre beaucoup d'avantages : réduction de la consommation de papier et d'encre, diminution du coût d'envoi, livraison presque instantanée du courrier, signature quasi infalsifiable et vérifiable par quiconque disposant des outils adéquats, réduction du volume de documents à stocker, copie aisée des documents électroniques (signature comprise) adressés à plusieurs destinataires, recherche avancée et rapide des documents signés archivés, etc.

De nombreuses sociétés ont déjà opté pour la signature électronique pour leurs documents internes (signature engageant la responsabilité du personnel, …) ou leurs documents externes (contrats avec d'autres sociétés, marchés publics, …). Par exemple, l'armée américaine utilise des applications commerciales de signature électronique comme *Infomosaic's SecureXML Java Applet*[1] qui permet la signature de document via un navigateur comme Internet Explorer ou Mozilla (uniquement sous Windows). A l'Etat Major de la Défense belge par contre, les documents sont encore signés de manière classique, sur papier. De nombreux formulaires sont remplis de manière électronique, puis imprimés en plusieurs exemplaires afin d'être signés manuellement.

Les législations européennes et belges en matière de signature électronique ajoutent un intérêt supplémentaire à cette signature : sa valeur légale. Une signature électronique possède, sous certaines conditions, la même valeur légale qu'une signature manuscrite.

Ce travail de fin d'études analyse les fondements de la signature électronique, à travers la cryptographie et les infrastructures à clef publique. De plus, il offre une mise en pratique de la signature par le développement d'une application multiplateforme permettant la signature en ligne de formulaire.

Je remercie le Major Mees, le Capitaine Thonnard, et le Major Hallot, pour leurs précieux conseils et pour l'attention qu'ils ont apportée à ce travail de fin d'études. Je remercie également le Capitaine Raeves qui m'a permis d'assister à la présentation du produit *Keyvelo* de la firme *Keyvolution*.

[1] Voir référence [18]

Table des matières

Acronymes

AES	Advanced Encryption Standard : standard de chiffrement qui succède à DES
API	Application Programmable Interface : interface de programmation, qui définit comment un composant informatique peut communiquer avec un autre.
ASN.1	Abstract Syntax Notation One : standard de représentation de données utilisé par de nombreux protocoles de communication standardisés par l'IETF. Possède différents types d'encodage : BER, DER, PER, etc.
BER	Basic Encoding Rules : un des formats d'encodage de données ASN.1
CA	Certificate Authority : autorité de certification garantissant l'appartenance d'une clef publique à une certaine personne.
CMS	Cryprographic Message Standard : format standard de signature de fichier, utilisé entre autres dans S/MIME, et qui succède à PKCS #7
COM	Component Object Model : composant logiciel, technologie propre à Microsoft
CPS	Certificate Practice Statement (voir réf. [7]) : document émis par les autorités de certification, décrivant leurs pratiques de certification, leurs engagements via-a-vis des personnes tierces, etc. Ce document doit être consulté par toute personne utilisant un certificat émis par l'autorité.
CRL	Certificate Revocation List : liste de révocation de certificat
DER	Distinguished Encoding Rules (for ASN.1) : un des formats d'encodage de données ASN.1. Il s'agit d'un sous-ensemble de BER utilisé pour la signature de document.
DES	Data Encryption Standard, standard de chiffrement développé par IBM pour le NIST (National Institute of Standards and Technology), remplacé aujourd'hui par AES.
DLL	Dynamic Linked Library : bibliothèque de fonctions qui peut être chargée et déchargée en mémoire de façon dynamique.
DSA	Digital Signature Algorithm : un algorithme cryptographique à clef asymétrique largement utilisé.
IETF	Internet Engineering Task Force : association s'occupant du développement et de la publication de standards pour Internet.
MIME	Multipurpose Internet Mail Extensions : standard permettant d'inclure des fichiers multimédias dans les courriers électroniques.
OCSP	Online Certificate Status Protocol : protocole permettant d'interroger un serveur sur le statut de révocation d'un certificat.

PER	Packet Encoding Rules : un des formats d'encodage de données ASN.1
PGP	Pretty Good Privacy : logiciel utilisé mondialement pour chiffrer et signer des courriers électroniques.
PIN	Personal Identity Number : code composé généralement de 4 chiffres, utilisé pour authentifier l'utilisateur (carte de banque, carte SIM de GSM, carte d'identité belge,…)
PKCS	Public Key Cryptophic Standards : ensemble de standards publiés par les laboratoires RSA.
PKI	Public Key Infrastructure : infrastructure à clef publique dont le but est de certifier l'appartenance d'une clef publique à une certaine personne.
RFC	Request of Comments : nom donné aux standards de l'IETF.
RSA	Algorithme cryptographique à clef asymétrique créé par Ron **R**ivest, Adi **S**hamir et Len **A**dleman, basé sur les propriétés des grands nombres premiers.
SMTP	Simple Mail Transport Protocol : le protocole d'échange de courrier électronique le plus utilisé.
SSL	Secure Sockets Layer : protocole cryptographique assurant l'authentification et la confidentialité pour des communications via Internet.
TLS	Transport Layer Security. Successeur de SSL (voir SSL)
URI	Uniform Resource Identifier : standard (voir référence [8]) définissant la représentation d'une 'ressource' sous forme de chaîne de caractères.
URL	Uniform Resource Locator : les URL forment un sous ensemble des URI, et représentent des ressources Internet (protocole http, ftp, etc.)
X.509	Standard de l'ITU-T définissant le format ASN.1 d'un certificat liant la clef publique à son propriétaire.
WYSIWYS	What You See Is What You Sign

Liste des figures et tableaux

Introduction

La communication par Internet est devenue un outil incontournable dans notre société. L'ordinateur a gagné sa place dans chaque foyer et chaque bureau. Aucune entreprise militaire ou civile ne peut se passer de l'échange d'information par courrier électronique. Cependant, Internet est un canal de communication peu sûr. Une personne malveillante peut facilement intercepter des messages et en substituer d'autres. La signature électronique apporte un service important à la communication : l'authenticité, et l'intégrité des messages, mais également la « non répudiation » d'un document par son destinataire, c'est-à-dire le fait qu'une personne ne peut pas nier avoir signé le document. Les notions-clés comme la confidentialité, l'authenticité, l'intégrité, forment la base de la sécurité réseau.

L'objectif principal de ce travail de fin d'études est la réalisation d'une application dans le cadre de la signature électronique : signer le contenu d'un formulaire en ligne. Cependant, une bonne compréhension du fonctionnement de la signature électronique est nécessaire. La méconnaissance d'un élément pourrait aboutir à une erreur de conception plus ou moins grave, et mettre en danger la fiabilité de l'application. Le travail possède donc deux parties : une partie théorique et une partie pratique.

La notion générale de signature électronique et son fonctionnement seront donnés dans la partie théorique. Nous introduirons tout d'abord les bases de la sécurité informatique et de la cryptologie : chiffrement, fonction de hachage, signature et cryptanalyse. Quelques exemples d'algorithmes célèbres seront analysés : Chiffrement de César, DES, AES, RSA, SHA-1, etc. Nous verrons la distinction entre les opérations cryptographiques à clefs symétriques, et à clefs asymétriques. Le premier type utilise la même clef à l'émission qu'à la réception. Ceci peut être utilisé pour assurer la confidentialité, l'authenticité et l'intégrité d'un message, mais n'offre pas le service de non répudiation car le récepteur possède également la clef et pourrait donc lui-même générer une signature. C'est donc sur le deuxième type d'opération (clefs asymétriques) que repose la signature électronique. Nous verrons comment un signataire peut signer un message avec sa propre clef privée et comment le destinataire peut vérifier le message avec la clef publique du signataire. Bien entendu, il doit être « quasi impossible » de trouver la clef privée à partir de la clef publique ! Nous expliquerons également ce qu'on entend par « quasi impossible ».

La signature électronique requiert également l'existence d'une infrastructure à clef publique. Comment être certain qu'une clef publique appartient à une personne ? Ne s'agit-il pas de la clef publique d'un attaquant qui veut usurper l'identité d'une autre personne ? Le *PKI* ou *Public Key Infrastructure* résout se problème crucial, mais rend malheureusement l'utilisation de la signature électronique plus lourde.

Enfin, la partie théorique se termine avec les formats de signature standardisés comme *CMS*, *S/MIME, XML Signature* ou encore *XadES*, permettant l'interopérabilité entre différents logiciels. L'utilisation de formats standardisés rend aussi plus facile l'expertise en cas de conflit.

La seconde partie consiste à développer une application Internet permettant à un utilisateur de signer un formulaire directement 'en ligne'. Cette application doit pouvoir accéder aux ressources cryptographiques de l'utilisateur : sa clef privée. Celle-ci peut-être stockée de plusieurs manières : dans un simple fichier (clef logicielle), ou sur un support matériel (clef matérielle) comme une carte à puce (par exemple la carte d'identité belge) ou une clef USB qui protège cette clef, par exemple avec un code PIN.

Une justification concernant le choix des outils de programmation est donnée. Une des particularités de ce projet est l'intégration des ressources cryptographiques de Microsoft Windows dans la plateforme Java.

Cette partie est clôturée par une discussion sur le choix du format de document à signer comme HTML ou PDF.

Le code source du projet ainsi que sa documentation sont accessibles sur :

http://rcardon.free.fr/websign

1 Sécurité informatique et cryptologie

1.1 Sécurité informatique

Aujourd'hui, la communication par voie informatique est omniprésente, même dans des secteurs sensibles comme le secteur bancaire, judiciaire, ou militaire. Lors de la création des premiers réseaux informatiques, la sécurité avait une importance secondaire.

Un des plus anciens protocoles dans l'histoire de l'Internet est le protocole simple de transfert de courrier SMTP. Son premier standard (RFC 821) date de 1982. La sécurité de ce protocole était très faible : il ne permettait pas l'authentification de l'expéditeur, et ne s'occupait pas du chiffrement du message. Celle-ci a été incluse plus tard comme extension (cfr. RFC 2554), mais n'a pas résolu le problème car la plupart des agents de transfert de courrier acceptent encore du courrier non authentifié. Aujourd'hui, il est encore extrêmement facile d'envoyer un mail sous un autre nom et une autre adresse de messagerie, même avec de simples logiciels de messagerie comme Outlook Express.

Sur d'anciens réseaux locaux de type Ethernet *half/duplex* où plusieurs utilisateurs travaillent dans un même domaine de collision, il est aisé de capturer de manière passive et invisible des messages qui ne nous sont pas adressés, par exemple à l'aide du logiciel *Ethereal*. Aujourd'hui, la plupart des réseaux sont *full/duplex*, et il n'est plus possible de capturer de manière passive les messages d'un autre utilisateur. Par contre, il existe des attaques actives comme le *MAC spoofing* qui permettent de détourner les messages de leur destination pour les capturer et les transférer ensuite vers leur bonne destination.

Pour transmettre des informations de manière sécurisée via un canal non sûr comme le réseau Internet, il est donc nécessaire de prévoir des mécanismes de sécurité de bout en bout.

Lors de l'échange d'un message entre un expéditeur et un destinataire, on distingue plusieurs notions de *sécurité réseau*:

- *la confidentialité* : le fait de garder secret le contenu d'un message.
- *l'intégrité* : s'assurer que le message n'a pas été modifié, ou n'est pas tronqué. En général on ne peut pas garantir l'intégrité d'un message mais bien contrôler son intégrité.
- *l'authenticité* : s'assurer de l'identité de l'auteur du message.
- *la non-répudiation* : garder une preuve de l'authenticité du message, que l'émetteur ne peut nier avoir envoyé. Cette notion s'avère importante dans le cadre légal de la signature électronique.

A côté de la sécurité réseau, on parle également de *sécurité système*. Celle-ci prévient les attaques contre le système venant du réseau informatique comme les virus, les hackers, etc. (mise en place d'un firewall ou utilisation d'un logiciel anti-virus par exemple).

Dans ce travail de fin d'études, nous nous intéressons surtout à la sécurité *réseau*. La plupart des mécanismes de sécurité réseau reposent sur des algorithmes cryptographiques. Claude Shannon a publié en 1949 un article intitulé « *Communication Theory of Secrecy Systems* ». Il y définit plusieurs critères pour évaluer la sécurité d'un système cryptographique :

- *sécurité calculatoire* : un système est sûr au sens de la *théorie de la complexité* si la quantité de calculs nécessaire pour casser le système est beaucoup trop grande.

- *sécurité prouvée* : on prouve qu'un système A est sûr si le système B est sûr, mais on ne prouve pas la sécurité du système B. On réduit donc le problème A à un problème B bien connu réputé difficile, comme la factorisation du produit de grands nombres entiers dans le cas de RSA. Il s'agit d'une sécurité conditionnelle et n'est donc pas une preuve de sécurité.
- *sécurité inconditionnelle* : un système est inconditionnellement sûr si on peut démontrer qu'il ne peut pas être cassé même avec une capacité de calcul infinie.

La plupart des algorithmes reposent sur la sécurité calculatoire et/ou prouvée mais n'ont pas une sécurité inconditionnelle. De plus, on découvre de jour en jour de nouvelles méthodes de cryptanalyse qui réduisent la sécurité calculatoire de certains algorithmes.

La signature électronique sert par exemple à *authentifier* un serveur et un client lors de la création d'un canal sûr (e-banking). Elle offre aussi la possibilité de dématérialiser les documents signés à valeur légale : signature des appels d'offre et des candidatures dans les marchés publics, signature de contrats, de déclarations fiscales, de factures, envoi de lettres recommandées électroniques, etc. Dans ce cadre, la signature électronique permet d'assurer la *non-répudiation* du signataire : un signataire ne peut nier avoir signé un document, car on peut prouver que la signature vient de lui seul, car lui seul possède les moyens cryptographiques (sa clef privée) pour signer le document, à moins que quelqu'un ne lui ai *volé* ses moyens. Cette dernière remarque est importante : s'il est facile de copier une signature manuscrite, il peut être encore plus facile de copier ou voler une clef électronique. L'utilisation d'une clef non extractible sur un support spécifique (stick USB ou carte à puce), appelée *clef matérielle* (par opposition à *clef logicielle*), empêche la copie de la clef, et protège en général la clef contre le vol grâce à un mot de passe qui bloque le mécanisme en cas de dépassement du maximum d'erreurs successives autorisé (code PIN).

1.2 La cryptologie

La cryptologie, ou étude du secret, est divisée en deux branches : la cryptographie et la cryptanalyse. Alors que la cryptographie définit les systèmes, la cryptanalyse tente de trouver les failles dans ces systèmes.

Le début de la cryptographie remonte à l'Antiquité. Le chiffrement de César est l'un des plus anciens. Si dans le passé, on utilisait la cryptographie principalement dans le but d'échanger des informations confidentielles, aujourd'hui, on veut également s'assurer de l'intégrité et de l'authenticité des données, sans que celles-ci ne soient nécessairement confidentielles.

Dans les prochains chapitres, nous allons détailler la cryptographie à clefs symétriques et à clefs asymétriques.

2 Systèmes cryptographiques à clefs symétriques et cryptanalyse

Les systèmes à clefs symétriques sont ceux qui utilisent la même clef pour chiffrer et pour déchiffrer un message. Dans ce chapitre, quelques algorithmes de chiffrement classique et modernes sont décrits, sans entrer dans tous les détails. Le problème de distribution de clef sera ensuite discuté. Nous verrons également quelques notions de cryptanalyse. Le présent chapitre ainsi que le chapitre suivant sont essentiellement tirés de l'excellent ouvrage de Douglas Stinson [1], que nous recommandons au lecteur désirant plus de détails sur les propriétés mathématiques des algorithmes.

2.1 Cryptographie classique

2.1.1 Chiffrement par décalage et chiffrement de César

Le chiffrement de César, ainsi nommé car il aurait été utilisé par Jules César, est un chiffrement par décalage de 3 positions ($K = 3$) : la lettre *a* est chiffré par la lettre *d*, la lettre *b* par la lettre *e*, et ainsi de suite (*z* est chiffré par *c*, on considère l'alphabet comme cyclique). On traduit cela mathématiquement grâce à l'utilisation de l'arithmétique modulaire. Dans le cas où on ne considère que les 26 lettres de l'alphabet, on a :

$$\text{chiffrement : } e(x) = (x + K) \bmod 26$$

$$\text{déchiffrement : } d(y) = (y - K) \bmod 26$$

où $x \in \{0;1;...;25\}$ est le texte clair[2], y ou $e(x)$ le texte chiffré (*e*ncrypt), et $d(y)$ le texte déchiffré (*d*ecrypt). On a évidemment $d\left(e(x)\right) = x$. L'opérateur a mod b est le reste de la division de a par b. A chaque lettre de l'alphabet, on fait correspondre un nombre x sur Z_{26} : $a \to 0$, $b \to 1$, $c \to 2$,..., $y \to 24$, $z \to 25$.

Une recherche exhaustive de clef sera aisée car l'ensemble des clefs ne possède que 26 éléments (dont l'élément K=0 pour lequel le texte chiffré est égal au texte clair). Ce chiffrement n'est donc pas efficace dans l'hypothèse où l'attaquant connaît le système cryptographique utilisé.

2.1.2 Chiffrement par substitution

Le chiffrement par substitution est obtenu grâce à une table de permutation. Le chiffrement par décalage est un cas particulier du chiffrement par substitution. Dans le cas de l'alphabet, chaque lettre est permutée par une autre lettre de l'alphabet. Voici un exemple de table de substitution. « *age* » sera chiffré par « XOH ».

Tableau 1: Exemple de chiffrement par substitution

a	*b*	*c*	*d*	*e*	*f*	*g*	…
X	N	Y	A	H	P	O	…

Cette table constitue la clef de chiffrement. Il y a $26!$ clefs différentes possibles. On note mathématiquement la permutation par la lettre grecque π. Le chiffrement et le déchiffrement s'écrivent donc :

$$e(x) = \pi(x)$$

[2] ici l'ensemble des « textes clairs » est défini comme l'ensemble des 26 lettres de l'alphabet.

$$d(y) = \pi^{-1}(y)$$

L'espace des clefs est ici suffisamment grand mais ce chiffrement souffre d'un autre problème qui sera traité au paragraphe 'cryptanalyse'.

Ce chiffrement ne doit pas être confondu avec le *chiffrement par permutation* décrit plus loin, même s'il est aussi fait usage d'une table de permutation.

2.1.3 Chiffrement de Vigenère

Le chiffrement de Vigenère (par Blaise Vigenère, XIe siècle) est proche du chiffrement par décalage mais la clef K utilisée n'est pas la même pour tous les caractères. On travaille par bloc de m caractères. Mathématiquement, le chiffrement de Vigenère s'écrit :

$$e(x_1, x_2, ..., x_m) = (x_1 + k_1, x_2 + k_2, ..., x_m + k_m) \bmod 26$$
$$d(y_1, y_2, ..., y_m) = (y_1 - k_1, y_2 - k_2, ..., y_m - k_m) \bmod 26$$

Ce chiffrement est dit poly-alphabétique par opposition à mono-alphabétique où chaque caractère alphabétique est transformé en un unique caractère alphabétique. La cryptanalyse d'un système poly-alphabétique est de fait plus difficile.

2.1.4 Chiffrement par permutation

Ce chiffrement par bloc de m textes, consiste à permuter les m textes clairs entre eux (à distinguer du chiffrement par substitution qui permute la valeur individuelle de chaque texte clair par un autre texte clair). On a :

$$e(x_1, x_2, ...x_m) = \left(x_{\pi(1)}, x_{\pi(2)}, ...x_{\pi(m)} \right)$$
$$d(y_1, y_2, ...y_m) = \left(y_{\pi^{-1}(1)}, y_{\pi^{-1}(2)}, ...y_{\pi^{-1}(m)} \right)$$

2.1.5 Chiffrement en chaîne

Les chiffrements vus précédemment sont des chiffrements par bloc. Ceux-ci traitent chaque bloc de la même manière (avec la même clef). Par contre le chiffrement en chaîne calcule une nouvelle clef pour chaque bloc. Une séquence de clefs est engendrée à partir de la clef initiale.

Le chiffrement en chaîne est dit « synchrone » si cette séquence de clefs est engendrée de manière indépendante du texte clair.

2.2 DES

Le DES (Data Encryption Standard) a été développé par IBM en réponse à l'appel d'offre américaine du NIST (National Institute of Standards and Technology). Le DES est une adaptation d'un chiffrement plus ancien 'Lucifer'.

Cet algorithme est un chiffrement par bloc itéré : les données sont traitées par blocs de 64 bits et pour chaque bloc une 'fonction d'étage' g est répétée N fois. Cette fonction g doit être injective afin de pouvoir exécuter la fonction inverse g^{-1} lors du déchiffrement.

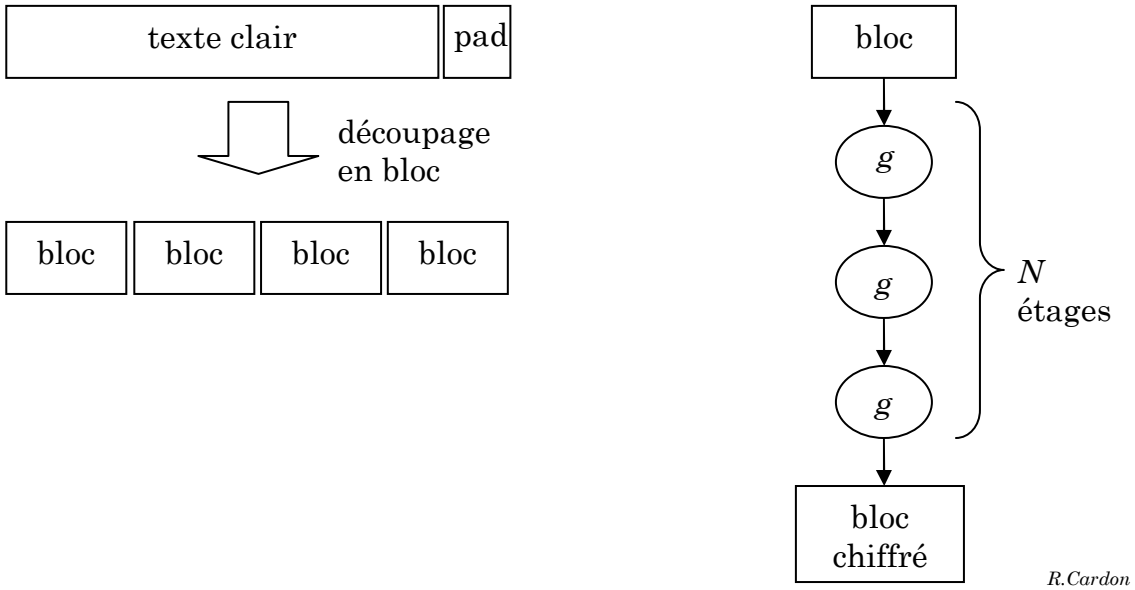


Figure 1 : DES schéma général

Pour le DES, cette fonction *g* fait appel au « schéma de *Feistel* ». Dans ce schéma, chaque bloc est divisé en deux parties de longueurs égales notées G^i et D^i. La fonction d'étage devient :

$$(G^i, D^i) = g(G^{i-1}, D^{i-1}, K^i)$$

avec
$$G^i = D^{i-1}$$
$$D^i = G^{i-1} \oplus f(D^{i-1}, K^i)$$

$\oplus$ est l'opérateur OU-exclusif bit par bit (souvent noté XOR en anglais). La particularité de ce schéma est que la fonction *f* ne doit pas être injective. En effet, il est possible de déchiffrer le message avec cette même fonction *f* :

$$G^{i-1} = D^i \oplus f(G^i, K^i)$$
$$D^{i-1} = G^i$$

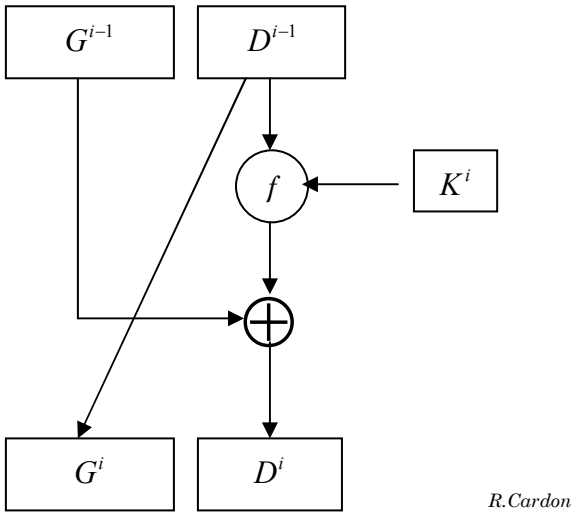


Figure 2 - schéma de Feistel

DES utilise des blocs de 64 bits et possède 16 étages. Ceux-ci sont divisés en deux parties G^i et D^i de 32 bits. Les 16 sous-clefs K^i ont chacune 48 bits et sont calculées à partir de la clef K de 64 bits (en réalité 56 bits, car la clef possède 8 bits de contrôle de parité) grâce à une fonction d'expansion. L'ensemble des sous-clefs forme la 'clef étendue' (de 48 x 16 = 768 bits). La fonction f dans DES est composée de substitutions S_1, S_2…S_6 (appelées S-boîtes) et d'une permutation P. La fonction d'expansion E permet d'augmenter le nombre de bits de D^{i-1} de 32 à 48 bits avant le mélange de sous-clef (opération OU-exclusif bit par bit avec la sous-clef K^i).

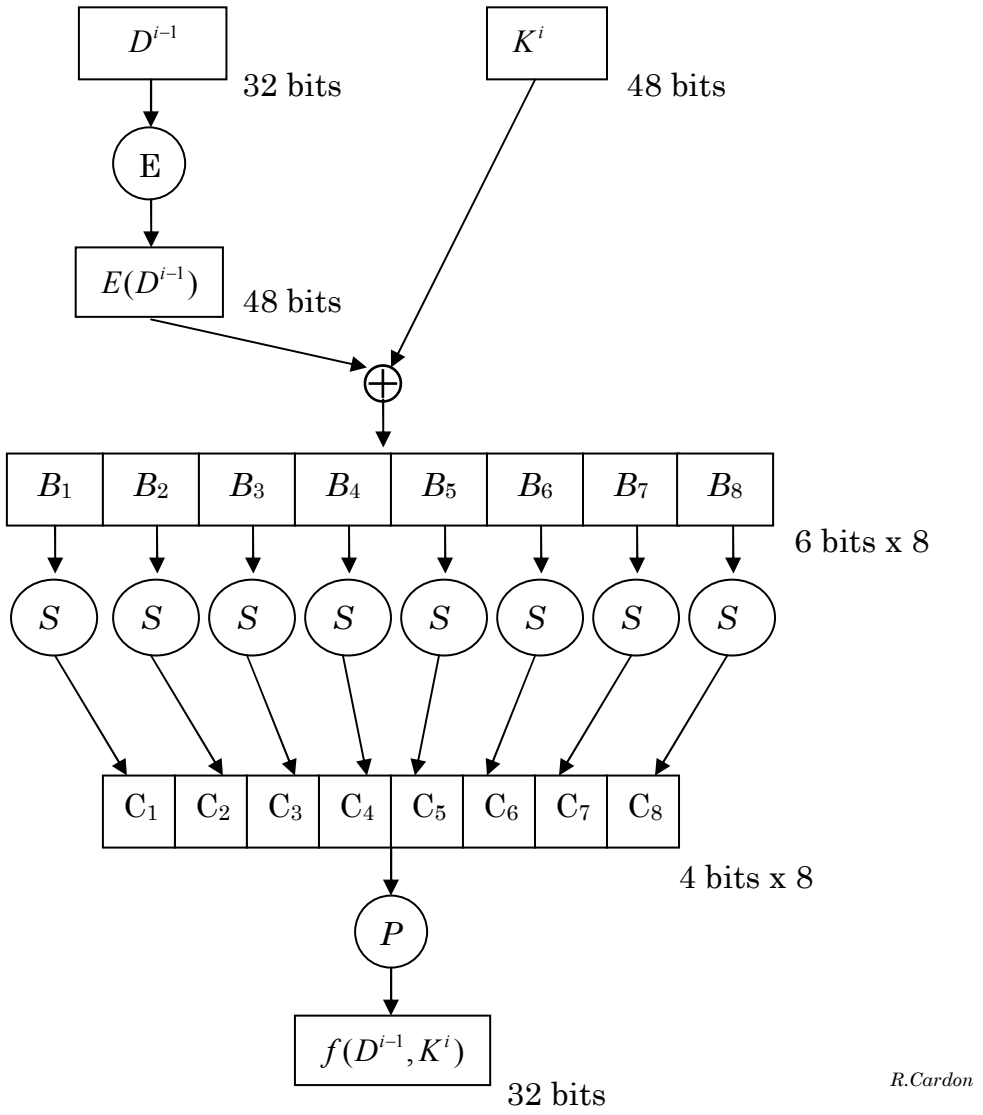


Figure 3 : détail de la fonction *f* de DES

Les 8 *S*-boîtes sont données sous forme de tableaux de $2^6 = 64$ éléments auxquels on a attribué des valeurs comprises entre 0 et 15 (2^4 éléments). Ces S-boîtes ne sont donc par des permutations comme dans le chiffrement par substitution, mais transforment l'ensemble $\{0,1\}^6$ vers l'ensemble $\{0,1\}^4$.

La définition de ces S-boîtes est très importante pour protéger l'algorithme de certaines attaques. Par exemple chaque nombre entre 0 et 15 doit apparaître un même nombre de fois dans une S-boîte (4 fois) pour éviter des attaques statistiques. D'autre part, il est

intéressant de noter que les S-boîtes ont été définies pour résister à la cryptanalyse différentielle. Cette analyse consiste à casser un code en se basant sur la différence de deux textes clairs. Lors de la découverte de cette attaque par *Biham* et *Shamir* (bien après la création du DES), on s'est rendu compte que les S-boîtes du DES avaient déjà été calculées pour résister de façon optimale à cette attaque…donc la cryptanalyse différentielle était bien connue des concepteurs de DES mais avait été gardée secrète !

On reproche à DES d'utiliser une clef de petite longueur (56 bits). Son prédécesseur *Lucifer* utilisait une clef de 128 bits. Mais il a été choisi à l'époque de limiter la clef à 64 bits, dont 16 bits de parité. Le progrès de la technologie ont permis la recherche exhaustive de clefs sur l'algorithme DES. En 1999, le réseau distribué *distributed.net* a permis de résoudre le concours DSE Challenge III des laboratoires RSA en 22 heures et 14 minutes, en testant 145 milliards de clefs par seconde !

Un autre type d'attaque – l'attaque à texte clair connu – a été fructueux mais demande un nombre très important de paires 'texte clair / texte chiffré'. Or, en réalité, il est très improbable qu'un adversaire dispose d'autant d'information.

A l'heure actuelle, on n'utilise plus DES en tant que tel mais une variante nommée triple-DES, ou bien on fait usage de l'AES.

2.3 AES

L'AES, ou *Advanced Encryption Standard*, remplace le DES, suite à un nouvel appel d'offre lancé en 1997 par le NIST. C'est le *chiffrement de Rijndael* qui a été choisi parmi les quinze candidats qui remplissaient tous les critères nécessaires. Ce choix a été réalisé lors de conférences internationales et ce, de manière toute à fait ouverte. Outre la sécurité, Rijndael a de bonnes qualités concernant le coût de calcul, la simplicité et la flexibilité d'implémentation logicielle ou matérielle (carte à puce). Cet algorithme est le fruit de deux chercheurs belges *Daemen* et *Rijmen*.

L'AES résiste actuellement à tous les types d'attaques connus. Cependant, il est possible dans le futur qu'une faille soit découverte, ou que l'évolution de la technologie impose d'utiliser des clefs de plus grande longueur.

La clef de chiffrement a une longueur de 128, 192, ou 256 bits. Le nombre d'étages dépend de la longueur de la clef : 10, 12 ou 14 étages respectivement.

L'AES travaille par bloc de 128 bits, c'est-à-dire 16 mots de 8 bits. L'état à chaque étage est représenté par un tableau de 4 x 4 mots $s_{i,j}$:

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$
$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$
$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$

L'état est initialisé avec le texte clair de cette manière :

x_0	x_4	x_8	x_{12}
x_1	x_5	x_9	x_{13}
x_2	x_6	x_{10}	x_{14}
x_3	x_7	x_{11}	x_{15}

➔

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,1}$	$s_{1,2}$	$s_{1,3}$	$s_{1,0}$
$s_{2,2}$	$s_{2,3}$	$s_{2,0}$	$s_{2,1}$
$s_{3,3}$	$s_{3,0}$	$s_{3,1}$	$s_{3,2}$

AES n'utilise pas le schéma de Feistel comme DES. A chaque étage, on effectue 4 opérations :

– opération de substitution pour chaque $s_{i,j}$ individuellement, en utilisant l'unique S-boîte qui constitue une permutation de 2^8 (256) éléments. La définition de cette S-boîte n'est pas le fruit du hasard et fait appel à l'inversion dans les corps finis. Cette définition permet de bien résister à des attaques linéaires et différentielles (voir paragraphe *cryptanalyse*).

– opération de permutation des mots (shift des 2e, 3e et 4e lignes):

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$
$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$
$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$

➔

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,1}$	$s_{1,2}$	$s_{1,3}$	$s_{1,0}$
$s_{2,2}$	$s_{2,3}$	$s_{2,0}$	$s_{2,1}$
$s_{3,3}$	$s_{3,0}$	$s_{3,1}$	$s_{3,2}$

– opération sur chaque colonne faisant appel à la multiplication dans les corps finis.

– « mélange de sous-clefs », c'est-à-dire opération XOR avec la sous-clef K_i

L'AES effectue également une opération dite de 'blanchissage' : tout au début et tout à la fin de l'algorithme, on effectue une opération XOR avec une sous-clef. Ceci est considéré comme un moyen efficace pour empêcher un attaquant de commencer à chiffrer ou à déchiffrer lorsque la clef est inconnue.

2.4 Problème de distribution des clefs secrètes

La cryptographie à clef secrète se révèle peu pratique pour assurer l'authenticité d'un message. Si plusieurs utilisateurs partagent la même clef secrète, ils ne pourront pas s'assurer de la provenance du message. Il faut donc prévoir une clef secrète pour chaque couple de personnes qui ont besoin communiquer. Pour x personnes on a besoin de $C_x^2 = \frac{x!}{2!(x-2)!} = \frac{x(x-1)}{2}$ clefs secrètes. Pour 6 personnes cela correspond à 15 clefs (représentées par 15 doubles flèches dans la figure ci-dessous).

Pour 1000 personnes, cela correspond à 499500 clefs secrètes. La distribution des clefs est problématique.

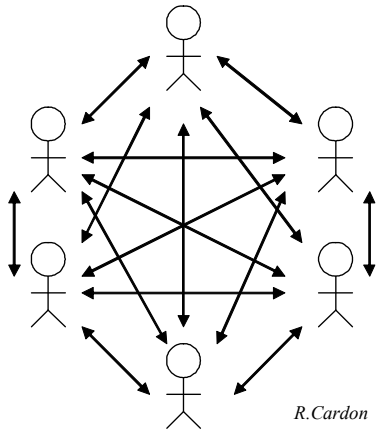
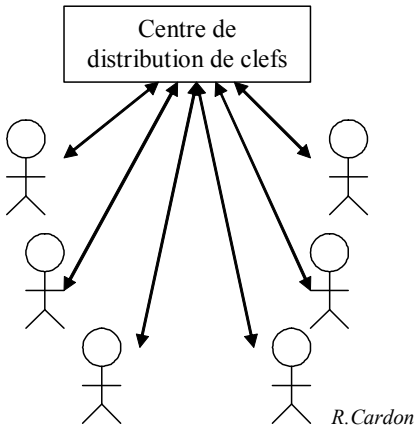


Figure 4 : Structure à clef secrète où chaque paire utilise une clef différente

En pratique, ceci n'est pas réalisable. On fera plutôt appel à une partie tierce de confiance jouant le rôle de centre de distribution de clef (*KDC* ou *Key Distribution Center*). Le centre de distribution de clef garde la clef secrète de chaque personne.

Lorsqu'une personne A veut communiquer avec une personne B, une clef temporaire appelée « clef de session » va être créée par le centre de distribution de clef. Pour cela, le schéma de *Needham-Schroeder* peut-être utilisé. Dans ce schéma, A fait une requête au centre de distribution de clef. Celui-ci renvoie à A une clef de session et un message destiné à B que A devra envoyer à B. Ce message contient la clef de session que seul B peut déchiffrer. Ceci évite que le centre de distribution de clef doive lui-même contacter B pour lui donner la clef de session. De plus, A ne connaîtra pas la clef secrète de B.



Kerberos est une variante du schéma de *Needham-Schroeder*. *Kerberos* est à la base un protocole d'authentification mais il permet également la création d'un tunnel sécurisé entre deux personnes avec une clef de session. Il rend les attaques plus difficiles en donnant un temps de vie limité aux clefs de session.

L'utilisation d'un centre de distribution de clef pose cependant les problèmes suivants :

- il doit être accessible en permanence,
- il contient en un seul endroit toutes les clefs secrètes. Si sa sécurité est compromise, cela peut s'avérer catastrophique.

Les systèmes cryptographiques à clefs asymétriques permettront de résoudre ces problèmes.

2.5 Possibilité de signature ?

Les systèmes à clefs symétriques permettent une application limitée de la signature : la clef secrète doit être transmise par voie sûre entre l'expéditeur et le destinataire. De plus ces systèmes ne permettent pas la non-répudiation d'une signature par l'expéditeur. Puisque le destinataire détient aussi la clef, il ne peut pas prouver que l'expéditeur a envoyé le message : le destinataire a peut-être composé lui-même ce message.

2.6 Cryptanalyse

En cryptanalyse, on part souvent du principe que le système cryptographique utilisé est connu (principe de *Kerchoffs*). Il est évident que l'utilisation du chiffrement de César ou « par décalage » se basait sur l'hypothèse contraire !

On définit différents *niveaux d'attaque* en cryptanalyse. Ces niveaux spécifient les connaissances dont dispose l'attaquant (par ordre croissant) :

- l'attaquant possède un texte chiffré,
- l'attaquant possède un texte clair et son texte chiffré correspondant,

- l'attaquant peut choisir un texte clair et le chiffrer (il possède une machine chiffrante),
- l'attaquant peut choisir un texte chiffré et le déchiffrer (il possède temporairement une machine déchiffrante).

Dans l'hypothèse de Kerchoffs, la connaissance d'un texte clair et de son texte chiffré correspondant suffit pour trouver la clef d'un chiffrement par décalage, et suffira souvent pour trouver la clef d'un chiffrement par substitution. Un bon algorithme doit pouvoir résister à tous ces niveaux d'attaques. On part également du principe que l'attaquant n'a pas une capacité de calcul infinie : un algorithme est considéré comme bon si le temps nécessaire pour le casser est beaucoup trop grand (sécurité calculatoire : voir paragraphe 1.1).

Un premier type d'attaque est la *recherche exhaustive de clefs*. C'est la méthode la plus simple mais souvent la moins efficace lorsque l'espace des clefs est grand. Cette méthode est efficace contre le chiffrement par décalage.

Un autre type d'attaque est l'utilisation des *propriétés statistiques de la langue* : on peut utiliser la probabilité d'apparition d'une lettre de l'alphabet, d'un groupe de 2 ou 3 lettres consécutives, et ainsi de suite. Ce type d'attaque convient bien contre le chiffrement par substitution, à condition bien sûr que les données chiffrées soient du texte rédigé dans la langue donnée. Le tableau suivant illustre donne la probabilité d'apparition des lettres de l'alphabet dans la langue anglaise :

Tableau 2 : Probabilités d'apparition des lettres de l'alphabet dans la langue anglaise.[3]

lettre	probabilité	lettre	probabilité
A	0,082	*N*	0,067
B	0,015	*O*	0,075
C	0,028	*P*	0,019
D	0,043	*Q*	0,001
E	0,127	*R*	0,060
F	0,022	*S*	0,063
G	0,020	*T*	0,091
H	0,061	*U*	0,028
I	0,070	*V*	0,010
J	0,002	*W*	0,023
K	0,008	*X*	0,001
L	0,040	*Y*	0,020
M	0,024	*Z*	0,001

Les deux types suivants de cryptanalyse ont déjà été mentionnés dans les paragraphes DES et AES :

- *Cryptanalyse linéaire :* la cryptanalyse linéaire est une attaque pour des algorithmes itératifs, qui tient compte de la dissymétrie d'une S-boîte. Il s'agit d'une attaque à texte clair et texte chiffré correspondant connu. L'attaquant doit posséder un nombre assez grand de paires texte clair / chiffré.
- *Cryptanalyse différentielle :* la cryptanalyse différentielle est semblable à la cryptanalyse linéaire. Elle se différencie principalement par le fait qu'elle compare le OU-exclusif de deux entrées avec le OU-exclusif de deux sorties. Il s'agit d'une attaque à texte clair choisi.

[3] Voir [1] page 26

2.7 Fonctions de hachage

Une fonction de hachage permet de s'assurer de l'intégrité des données. Elle permet de construire une courte empreinte numérique. La fonction de hachage doit être résistante aux collisions : il doit être difficile d'obtenir la même empreinte numérique en modifiant les données.

Plus formellement, trois problèmes doivent être difficiles à résoudre :

- étant donnée une empreinte numérique, il doit être difficile de trouver une chaîne de données correspondant à cette empreinte numérique. Dans ce cas, la fonction est dite à sens unique ou « *résistante à la préimage* ».
- étant donnée une chaîne de données, il doit être difficile de trouver une autre chaîne de données ayant la même empreinte numérique. Dans ce cas, la fonction est dite *« résistante à la seconde préimage* ».
- il doit être difficile de trouver deux chaînes de données ayant la même empreinte numérique. Dans ce cas, la fonction est dite « *résistante aux collisions*. »

Le troisième problème est le plus facile à résoudre. Il est similaire au paradoxe des anniversaires (probabilité de trouver deux personnes qui ont le même anniversaire). Inversement, si le troisième problème est difficile à résoudre, le premier et deuxième problème le seront également.

D'autre part, la difficulté de résoudre le premier problème n'implique pas nécessairement que le deuxième problème est difficile à solutionner.

C'est le deuxième problème qui a le plus d'intérêt concernant la sécurité informatique : il empêche une personne malveillante de substituer un message au message original.

De plus, une empreinte permet la détection d'erreur, mais pas la correction de celle-ci. Ceci est une conséquence logique de la résistance à la préimage (premier problème).

La fonction de hachage reçoit en entrée une chaîne binaire de longueur arbitraire, et renvoie en sortie une chaîne binaire de taille fixe. 160 bits est une taille typique.

L'attaque des anniversaires permet d'imposer une borne inférieure à la taille des empreintes. On peut démontrer[4] que la probabilité de trouver une collision est ½ lorsqu'on calcule l'empreinte d'un peu plus de $\sqrt{M}$ éléments aléatoires, où M est le nombre d'éléments dans l'ensemble des empreintes possibles. Une empreinte de 40 bits par exemple est peu sûre car on peu trouver une collision avec une probabilité d'un demi en hachant un peu plus de $\sqrt{2^{40}} = 2^{20}$ (environ un million) d'éléments.

Lorsqu'une personne veut s'assurer de l'intégrité mais également de l'authenticité d'un message échangé sur un canal peu sûr, il doit :

- soit recevoir une empreinte numérique via une voie sûre,
- soit recevoir une empreinte numérique générée par une fonction de hachage paramétrée avec une clef K secrète (connue uniquement de l'expéditeur et du destinataire). Une telle empreinte est appelée *code d'authentification de message* (MAC ou *message authentication code*).
- soit recevoir une empreinte numérique chiffrée à l'aide de la clef privé de l'expéditeur, et posséder la clef publique de l'expéditeur et être certain de l'origine de cette clef publique (voir chapitre 3).

[4] Voir [1] page 123

En pratique, on travaille avec des *fonctions de hachage itérées* : une *fonction de compression* est utilisée de manière cyclique sur l'ensemble de la chaîne à hacher (IV ou *initial value* est une valeur initiale publique) :

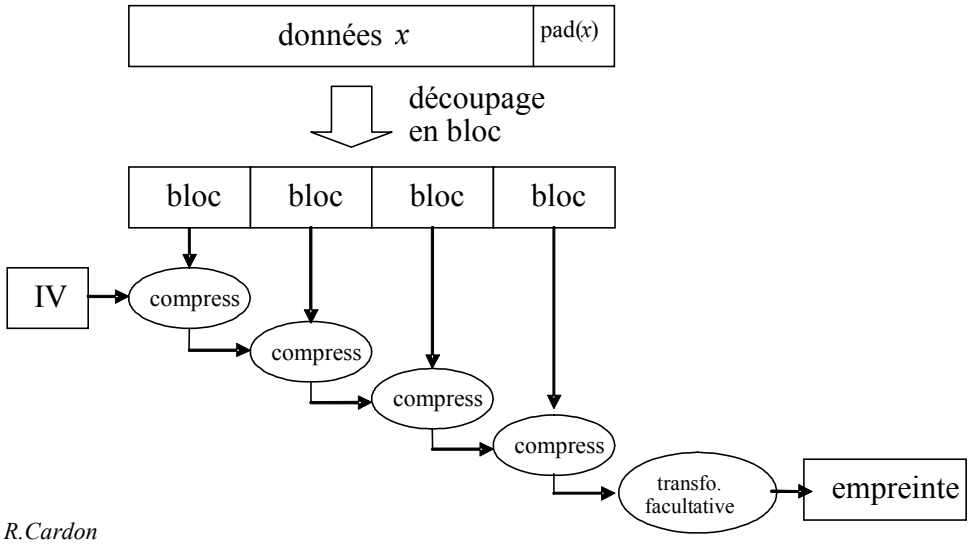


Figure 5 : fonction de hachage

Si la taille de sortie de la fonction de compression est différente de la taille de l'empreinte, une fonction de transformation en sortie est nécessaire.

La *construction de Merkle-Damgard* permet d'établir une fonction de hachage itérée résistante aux collisions, à partir d'une fonction de compression résistante aux collisions. Cette construction possède une sécurité prouvée[5].

2.7.1 SHA-1

SHA-1 est une variante mineure de SHA (*Secure Hash Algorithm*). Il s'agit d'une fonction de hachage itérée avec une empreinte de 160 bits. Sa première version 'SHA' a été proposée comme standard de fonction de hachage par le MITS en 1993 et succède au MD4 (Rivest, 1990) et MD5 (1992).

Des collisions sur la fonction de hachage MD4 et MD5 ont été découvertes. SHA constitue une amélioration mais a montré également ses faiblesses. Des corrections ont été effectuées et ont mené au SHA-1.

De nouvelles fonctions de hachage ont été développées : SHA-256, SHA-384, SHA-512. Le préfixe correspond à la taille de l'empreinte en bits.

Le SHA-1 travaille avec des blocs de 512 bits. La fonction de compression prend en entrée le bloc de 512 bits suivant et le résultat précédent de 160 bits, et retourne une valeur de 160 bits.

[5] voir section 1.1.

3 Systèmes cryptographiques à clefs asymétriques

3.1 Introduction à la cryptographie asymétrique

Dans les systèmes à clefs asymétriques, on utilise deux clefs différentes : une clef privée qui doit être gardée secrète par son propriétaire et que lui seul possède, et une clef publique qui peut être connue de tous. Le premier algorithme à clef asymétrique a été inventé par Diffie et Hellman.

Lorsqu'un expéditeur A envoie un message confidentiel, il le chiffre avec la clef publique du destinataire B. Ce dernier peut ensuite déchiffrer le message avec sa propre clef privée.

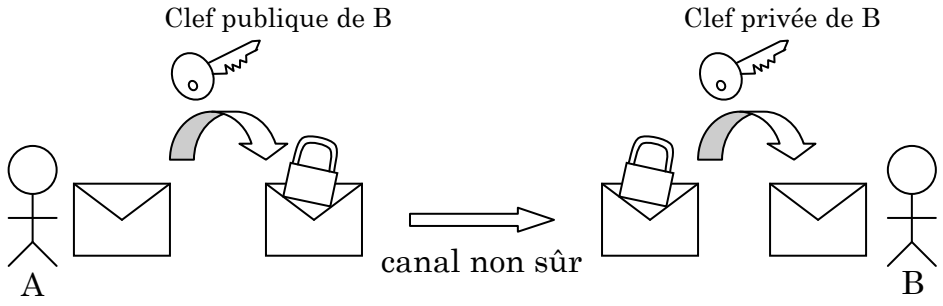


Figure 6 : chiffrement d'un message avec clef asymétrique

A l'inverse, lorsqu'un expéditeur A veut signer un message, il calcule l'empreinte numérique du message avec une fonction de hachage résistante à la préimage[6], puis chiffre cette empreinte avec sa propre clef privée. Le destinataire B peut vérifier l'intégrité et l'authenticité du message en recalculant l'empreinte, et en la comparant avec l'empreinte chiffrée par A qu'il déchiffre avec la clef publique de A. Notons que le chiffrement avec la clef privée est en général un algorithme différent que le chiffrement avec la clef publique.

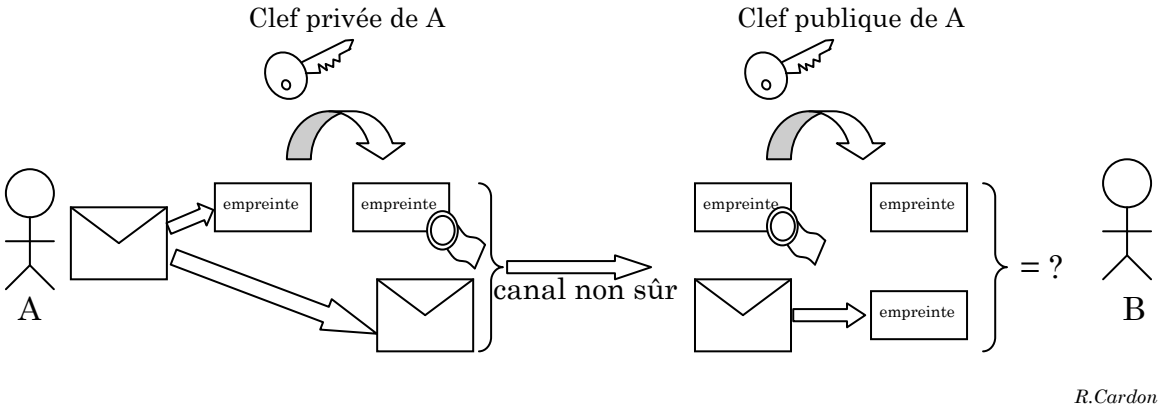


Figure 7 : Signature d'un message avec clef asymétrique

[6] voir section 2.7

On peut également combiner signature et chiffrement : l'expéditeur signe le message avec sa propre clef privée et chiffre le tout avec la clef publique du destinataire. Le destinataire peut ensuite déchiffrer le message avec sa propre clef privée et contrôler la signature avec la clef publique de l'expéditeur.

Que se passe-t-il si on inverse le mécanisme : chiffrement, puis signature ? Dans ce cas, un attaquant qui intercepte le message pourrait substituer facilement la signature par sa propre signature. C'est pourquoi, il est recommandé de signer le document avant de le chiffrer.

En pratique les algorithmes de chiffrement à clefs asymétriques sont plus lents que ceux à clef symétriques. C'est pourquoi les deux types de chiffrement sont combinés en pratique : pour chiffrer un message confidentiel, l'expéditeur génère d'abord une clef symétrique, appelée « clef de session », qui servira à chiffrer le message. L'expéditeur chiffre cette clef de session avec sa clef privée et l'envoie avec le message. Afin d'augmenter la sécurité, une nouvelle clef de session est créée pour chaque message.

On distingue les clefs privées logicielles et matérielles. A l'inverse d'une clef logicielle, une clef matérielle n'est pas extractible de son support (stick USB, carte à puce). Les algorithmes de chiffrement, de déchiffrement et de signature sont donc réalisés dans le support-même.

Les systèmes cryptographiques à clefs asymétriques se basent sur les principes suivants :

- le chiffrement (avec la clef publique) est une « fonction à sens unique à trappe » : il doit être très difficile de trouver l'inverse de cette fonction sans connaître la clef privée. Par contre, il ne faut pas perdre de vue que la fonction de chiffrement doit être injective : il doit exister une « trappe cachée », sans quoi le destinataire ne pourrait pas déchiffrer le message avec sa clef privée.
- connaissant la clef publique, il doit être difficile de déterminer la clef privée correspondante (sécurité calculatoire). L'algorithme RSA se base sur la difficulté de factoriser de grands nombres entiers, tandis que l'algorithme ElGamal se base sur le problème du logarithme discret.

3.2 La signature électronique

Comparons une signature électronique à une signature conventionnelle sur papier :

- il faut un moyen de '*coller*' la signature électronique au message signé. La signature électronique est calculée en fonction du contenu du message et ne peut donc pas être utilisée comme signature valide pour un autre message.
- *copier* la signature conventionnelle de quelqu'un d'autre est aisé, et il est difficile de *vérifier* si une signature est falsifiée ou non. Concernant la signature électronique, il est très difficile de copier la signature de quelqu'un d'autre (car la signature dépend du message) à moins qu'on ne lui vole ou copie sa clef privée, et il est facile de vérifier la validité de la signature (à l'aide de la clef publique du signataire accessible à tous).

Un document papier signé de manière conventionnel ne peut pas être reproduit facilement. Par contre un document électronique signé le peu. Pour éviter qu'un document électronique ne puisse être réutilisé (par exemple un document autorisant une autre personne retirer de l'argent sur son compte), il faut ajouter une date ou un autre élément identifiant le document de manière unique.

Il est important de distinguer deux applications de la signature électronique :

- authentification d'une personne,

– signature avec non répudiation.

Ces deux applications utilisent le même mécanisme : signature de données avec la clef privée du signataire.

Dans le cas de l'authentification, les données à signer sont des données aléatoires générées par l'autre partie. L'autre partie reçoit la signature, et est certaine de l'identité de la personne. Cette authentification n'a pas d'objectif de 'non répudiation'. Une fois que la signature a été vérifiée, elle est généralement jetée.

Dans le cas de la signature avec non répudiation, les données à signer sont un document (un contrat par exemple). La signature est conservée précieusement par le destinataire. Le signataire ne peut pas nier avoir signé le document.

La carte d'identité belge contient deux clefs privées. L'une sert à l'authentification et n'exige pas l'entrée répétitive du code PIN (utiliser par exemple lors de l'ouverture d'une connexion SSLv3 ou TSLv1 pour identifier le client), tandis que l'autre sert à signer des documents avec non répudiation et exige le code PIN à chaque opération.

Concernant la sécurité d'une signature, on distingue les attaques suivantes :

– *attaque sans message* : l'attaquant ne dispose d'aucun message signé.
– *attaque à message connu* : l'attaquant dispose d'un message signé (non choisi par l'attaquant).
– *attaque à message choisi* : l'attaquant choisit un message, et obtient une signature de celui-ci par la victime.

On distingue également les objectifs des attaques :

– *cassage total* : l'attaquant détermine la clef privée et peut donc signer n'importe quel message.
– *falsification sélective* : l'attaquant réussit à signer un message qu'il a choisi.
– *falsification existentielle* : l'attaquant réussit à signer au moins un message quelconque (non choisi).

La falsification existentielle ne porte en général pas atteinte à la sécurité d'une signature utilisant l'empreinte du message. En effet, si l'attaquant réussit à signer une empreinte quelconque (qu'il n'a pas choisie), et si la fonction de hachage est sûre (c'est-à-dire est résistante à la préimage), il trouvera difficilement un message correspondant à cette empreinte et il trouvera encore plus difficilement un message dont il peut choisir une partie du contenu.

Les signatures électroniques ne sont jamais sûre inconditionnellement : une personne possédant une capacité de calcul infinie pourra toujours trouver un message et une signature valide, en utilisant la fonction de vérification. Donc les signatures reposent sur la *sécurité calculatoire*[7].

Ces notions seront utilisées par la suite pour décrire certaines attaques existantes.

3.3 Problème de distribution de clef

La cryptographie à clef publique permet de résoudre les problèmes de distribution de clef rencontrés avec les systèmes symétriques. Pour cela, chaque personne devra faire confiance à une *autorité de certification* (*Certificate Authority* ou *CA*) et stocker la clef publique de l'autorité de certification (la clef publique doit être communiquée par une

[7] voir section 1.1

voix sûre pour garantir l'authenticité). L'autorité de certification n'a pas besoin de connaître la clef privée de ses utilisateurs, et elle n'a pas non plus besoin d'être accessible en permanence.

Lorsqu'une personne envoie un message chiffré, ou vérifie l'authenticité d'un message signé, elle a besoin de la clef publique de l'autre personne. Cette clef publique peut-être transmise dans un canal non sûr. Pour garantir l'authenticité de cette clef publique, la clef sera transmise avec un certificat signé par l'autorité de certification. Ceci est l'objet du prochain chapitre (chapitre 4).

3.4 Chiffrement RSA et factorisation des nombres entiers

L'algorithme de chiffrement RSA a été décrit par Ron Rivest, Adi Shamir et Len Adleman. Cet algorithme repose sur la difficulté de factoriser de grands nombres entiers.

On choisit p et n deux grands nombres premiers. On calcule $n = p.q$ et la fonction d'Euler $\phi(n) = (q-1)(p-1)$. La fonction d'Euler donne le nombre d'entiers positifs strictement inférieurs à n et premiers entre eux[8]. De plus on choisit un nombre entier a inversible dans $\mathbb{Z}_{\phi(n)}$ (en vérifiant que le $\text{PGCD}(a,\phi(n)) = 1$) et on détermine son inverse b tel que $ab \equiv 1 \left(\text{mod}\, \phi(n)\right)$ grâce à l'algorithme d'Euclide étendu. a et b sont appelés respectivement exposant de déchiffrement et exposant de chiffrement. On a :

$$e_K(x) = x^b \bmod n$$
$$d_K(y) = y^a \bmod n$$

où x et y appartiennent à Z_n

La clef publique est composée de n et b, tandis que la clef privée est composée de p, q et a.

Il est possible de démontrer[9] que $\left(x^b\right)^a \equiv x \left(\text{mod}\, n\right)$. L'exponentiation par a et b semble à première vue une opération longue, surtout lorsque a et b sont de grands nombres, mais ce n'est pas le cas : on utilise l'algorithme d'exponentiation rapide qui se base sur la décomposition binaire de l'exposant.

Pour déchiffrer un message sans connaître la clef privée, un attaquant devrait trouver a, en inversant b dans $\mathbb{Z}_{\phi(n)}$. Il doit donc trouver $\phi(n) = (p-1)(q-1)$ en factorisant n.

Pour appliquer l'algorithme RSA, il est d'abord nécessaire de générer les clefs. En pratique on génère de manière aléatoire de grands nombres p et q et on teste leur primalité jusqu'à l'obtention d'un nombre premier. Le *théorème de raréfaction des nombres premiers* nous donne une grandeur du nombre moyen d'essais qu'il faut réaliser : la quantité de nombres premiers inférieurs ou égaux à N, noté $\pi(N)$, est

$$\pi(N) \approx \frac{N}{\ln N}$$

Donc, si on tire au hasard un nombre *impair*, entre 1 et N, la probabilité de tirer un nombre premier est $2/\ln N$. Pour des nombres p et q de 512 bits (n a 1024 bits), on a une probabilité de $2/\ln 2^{512} \approx 1/177$ d'avoir un nombre entier. Dans notre exemple, il faudra donc tester en moyenne 177 nombres.

[8] Voir [2] page 354
[9] Voir [1] page 167

Il est souvent fait usage d'un algorithme de type « Monte Carlo positif » pour le test de primalité (par exemple l'algorithme de *Solovay-Strassen* ou de *Miller-Rabin*, ce dernier étant plus performant). Ce type d'algorithme est probabiliste : lorsque le nombre donné est premier, l'algorithme donnera toujours une réponse positive, par contre lorsque le nombre donné n'est pas premier, il y a une certaine probabilité que l'algorithme retourne une réponse positive, donc erronée.

La plupart des attaques contre RSA consistent à factoriser n. La méthode la plus naïve est la division par essais successifs : on tente de diviser n par tous les entiers impaires jusqu'à $\sqrt{n}$. Des méthodes plus sophistiquées ont été mises au point au fil du temps. Citons quelques méthodes : la méthode ρ de Pollard, la méthode $p - 1$ de Pollard, l'algorithme de factorisation utilisant les courbes elliptiques, etc. Une autre attaque, l'algorithme de Wiener, permet de retrouver l'exposant a sans chercher p et q, sous certaines conditions. Toutes ces attaques ont pour but le « cassage total » du système, c'est-à-dire la détermination de la clef privée. D'autres attaques permettent, par exemple, de distinguer deux textes chiffrés correspondant à deux textes clairs donnés.

3.5 Signature RSA

L'algorithme de signature RSA utilise le même procédé que l'algorithme de chiffrement RSA, sauf que la signature sig(x) utilise la clef *privée* de l'émetteur et que la vérification ver(x,y) utilise la clef *publique* de l'émetteur :

$$y = \mathrm{sig}(x) = x^a \bmod n$$
$$\mathrm{ver}(x, y) = vrai \Leftrightarrow x \equiv y^b \pmod n$$

En général x est l'empreinte du message, généré avec une fonction de hachage comme SHA-1.

Dans l'hypothèse que x est le message clair (et non l'empreinte), il existe une attaque de falsification existentielle fondée sur la propriété multiplicative de RSA[10] : si l'attaquant possède deux documents x_1 et x_2 avec leurs signature y_1 *et* y_2, il peut très facilement en créer un troisième (non choisi) $x_1.x_2$ dont la signature est égale à $y_1.y_2$ et est valide. Mais, comme expliqué dans la section 3.2, si x est l'empreinte du message, l'attaquant ne pourra pas trouver de texte clair correspondant.

3.6 ElGamal et logarithme discret

ElGamal est un autre type d'algorithme asymétrique qui se base sur le logarithme discret. La difficulté qui forme la fonction à sens unique à trappe est le calcul du logarithme dans un corps fini : connaissant α et β, le problème est de déterminer γ tel que :

$$\alpha^\gamma \equiv \beta \pmod p$$

On note $\gamma = \log_\alpha \beta$. S'il est très difficile de calculer ce logarithme, l'exponentiation pour calculer β à partir de α et γ est facile. Notons bien que l'on travaille dans un champ[11] fini, et non pas dans le champ infini de réels ! L'étude du logarithme discret exige donc une bonne connaissance de la théorie des ensembles. La clef publique contient $\{p, \alpha, \beta\}$ et la clef privée contient γ.

[10] voir [1] page 256

[11] Notez que *corps* est synonyme de *champ* (TE541). Un corps ou champ est un anneau qui admet un inverse pour tous les éléments non nul

Une particularité de cet algorithme est qu'il est probabiliste (par opposition à déterministe comme l'algorithme RSA). Ceci signifie que l'utilisation répétée de l'algorithme sur un même message donnera des signatures différentes. La raison est qu'un paramètre aléatoire k est utilisé dans l'algorithme. Il faut être très vigilant avec ce nombre k : il doit être tenu secret car la connaissance de ce nombre permet facilement de trouver la clef privée (cassage total !). Ce nombre n'est pas nécessaire pour la vérification de la signature, et ne peut jamais être transmis. D'autre part, on ne peut surtout pas signer deux documents avec le même nombre k. Si un attaquant sait que deux messages sont signés avec un même nombre k, il peut également trouver la clef privée, même sans connaître ce nombre k.

Tout comme la signature RSA, il existe aussi une attaque très simple de falsification existentielle contre la signature d'ElGamal. Par contre dans le cas d'ElGamal, cette attaque peut se réaliser sans aucun message signé (alors que l'attaque utilisant la propriété multiplicative de RSA demande la connaissance de deux messages signés).

3.7 Taille de la signature

La signature d'un message comme document légal (un contrat par exemple) impose l'utilisation d'un algorithme qui soit durable dans le temps. Il faut que la signature soit vérifiable encore plusieurs années après, tenant compte de l'évolution de la technologie.

Dans le cas d'ElGamal on recommande actuellement d'utiliser un module p de 1024 bits, ce qui correspond à une signature ElGamal de 2048 bits. Il existe des variantes de la signature d'ElGamal qui permettent de réduire la taille de la signature sans pour autant perdre en facteur de sécurité : il s'agit de la signature de Schnorr que se base sur ElGamal, et la signature DSA (Digital Signature Algorithm) qui se base sur ElGamal et Schnorr. Ce dernier a été choisi par le NIST en 1994 comme standard de signature. La signature est de 320 bits et la taille du module est paramétrable de 512 à 1024 bits.

4 Infrastructure à clef publique (PKI)

Il est nécessaire de s'assurer qu'une clef publique appartient bien à son prétendu propriétaire. Si Alice veut envoyer un message confidentiel à Bob, elle utilisera la clef publique de Bob pour chiffrer le message. Un attaquant, Oscar, pourrait envoyer sa propre clef publique à Alice en se faisant passer pour Bob. Alice chiffrera donc le message avec la clef d'Oscar et l'enverra à Bob. Si Oscar parvient à intercepter ce message, il pourra le déchiffrer, et pourra le transférer à Bob en le chiffrant avec la clef publique de Bob.

L'*infrastructure à clef publique*, mieux connue en anglais par l'abréviation *PKI* (*Public Key Infrastructure*), permet de garantir l'appartenance d'une clef publique à une personne par la mise en œuvre d'une hiérarchie de certification.

4.1 Certificat X.509

Un certificat est un document signé par une autorité de certification (CA ou *Certificate Authority*). Ce document contient entre autre une clef publique, le nom du propriétaire de cette clef et le nom l'autorité de certification. Le certificat X.509 est un certificat qui a été standardisé par l'ITU (International Telecommunications Union) et adopté comme standard Internet par l'IETF (RFC3280). Ces informations sont visibles sous Microsoft Windows dans les propriétés « Clé publique », « Objet », et « Emetteur ». L'exemple suivant montre un certificat issu de la carte d'identité belge électronique.

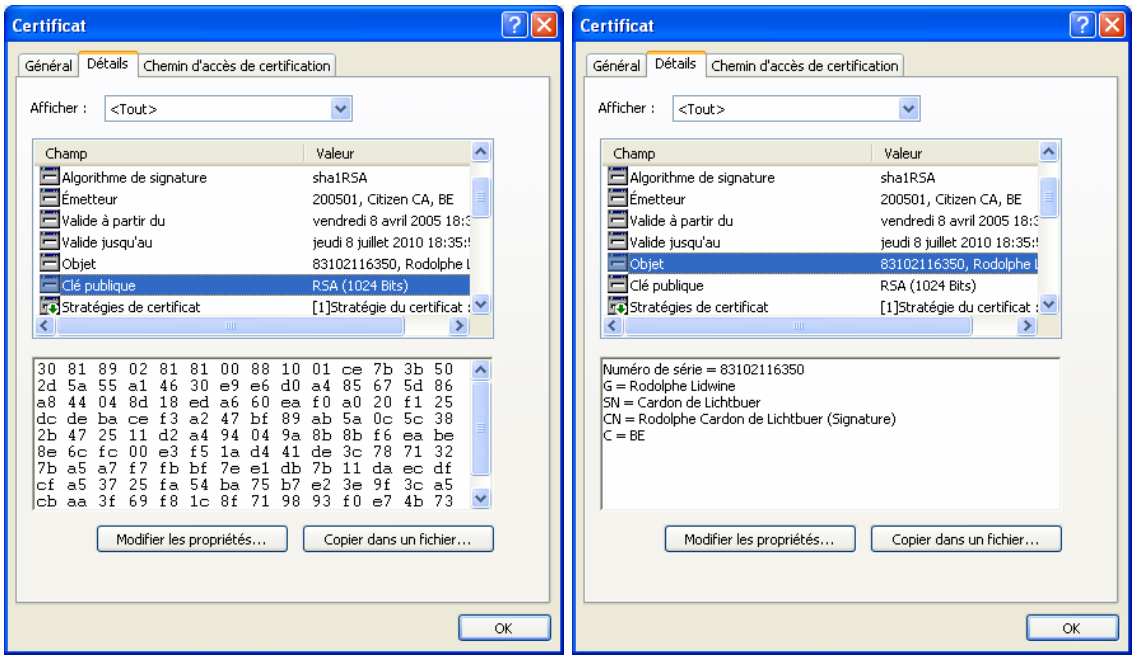


Figure 8 : Affichage d'un certificat dans Windows

Le certificat X.509 contient les informations suivantes :

- version (V3)
- numéro de série : unique auprès de chaque émetteur. Deux émetteurs différents peuvent utiliser le même numéro de série pour des certificats différents.
- algorithme de signature : algorithme utilisé pour signer le certificat (par exemple empreinte SHA-1 avec RSA)
- émetteur du certificat (*issuer name*) : le nom de la personne ou l'organisation qui signe le certificat

- période de validité : comprend la date de début et date de fin de validité.
- sujet du certificat (*subject name*)
- clef publique du sujet
- extensions (version 3 uniquement)
- signature du certificat

4.2 ASN.1 et OID

Le standard X.509 utilise le langage de définition de données ASN.1 (Abstract Syntax Notation One). ASN.1 n'est pas un format d'encodage en soi. Il existe différents formats d'encodage pour les données ASN.1 :

- BER : Basic Encoding Rules
- DER : Distinguished Encoding Rules.
- PER : Packet Encoding Rules

DER est un sous-ensemble de l'encodage BER. Avec BER, il est possible d'encoder la même information de plusieurs manières différentes. Ceci peut poser des problèmes avec la signature de données ASN.1 : un réencodage de la même information avec BER pourrait rendre la signature non valide. DER constitue une forme canonique de DER et résout ce problème.

BER, DER, et PER sont des données binaires qui doivent parfois être converties en données ASCII pour être échangées sur un réseau. Pour cela, on utilise l'encodage *base 64*. Ce dernier utilise un alphabet de 64 caractères imprimables (26*2 lettres de l'alphabet, 10 chiffres, 2 symboles '+' et '/') ainsi qu'un caractère complémentaire '='.

Le nom de l'émetteur et le nom du sujet du certificat ne sont pas de simples chaînes de caractère mais un *Dinstinguished Name* (DN) comme défini dans les standards X.500 et X.501. Il s'agit d'une séquence de *Relative Distinguished Name* (RDN). Chaque RDN possède un attribut et une valeur. Les attributs sont identifiés par leur *OID* (*Object Identifier*), qui est une suite hiérarchique de numéros :

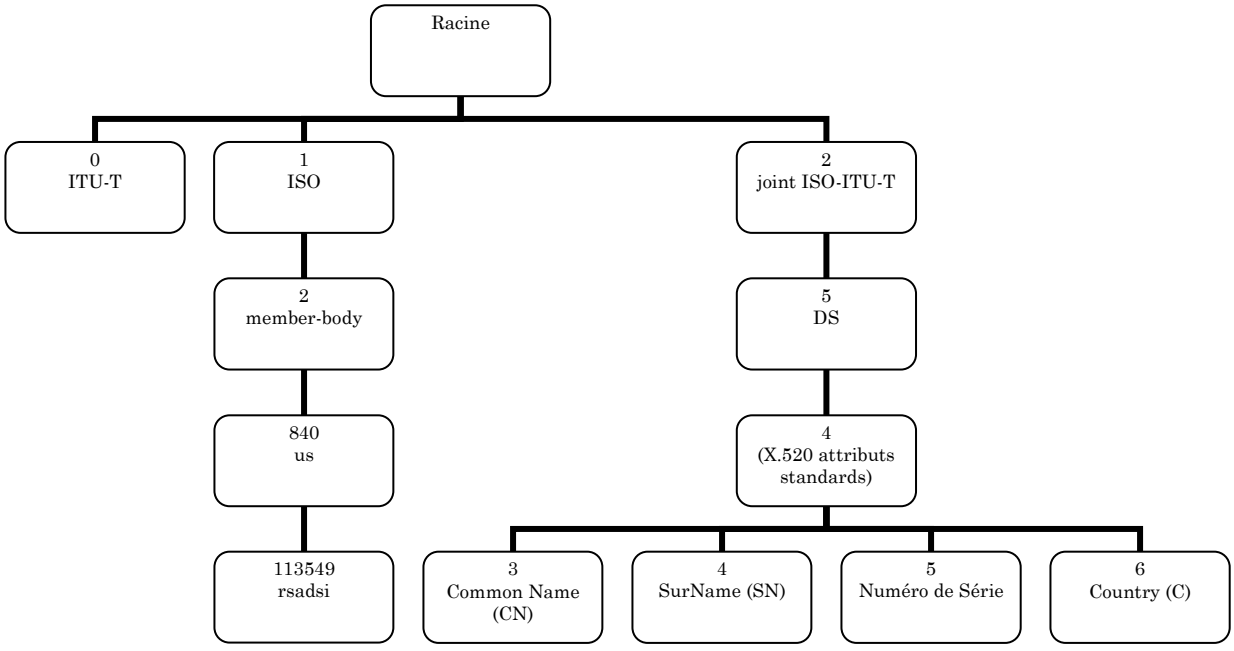


Figure 9: Object ID

Dans notre exemple issu de la carte d'identité belge, le sujet du certificat contient une séquence de 5 RDN :

Tableau 3 : Exemples d'Object ID

OID de l'attribut	*signification de l'attribut*	*valeur*
{2.5.4.6}	C (Country)	BE
{2.5.4.3}	CN (Common Name)	Rodolphe Cardon de Lichtbuer (Signature)
{2.5.4.4}	SN (SurName)	Cardon de Lichtbuer
{2.5.4.42}	G (GivenName)	Rodolphe Lidwine
{2.5.4.5}	numéro de série	83102116350

Le RFC 1485 de l'IETF définit une manière d'écrire les noms DN sous forme de chaîne de caractères. On écrit le RDN le plus significatif (ici C=BE) à la fin. Les RDN sont séparés par des virgules. Exemple :

```
Numéro de série = 83102116350, G = Rodolphe Lidwine, SN = Cardon de
Lichtbuer, CN = Rodolphe Cardon de Lichtbuer (Signature), C = BE
```

L'algorithme de signature est également identifié par un *OID*.

Tableau 4: Object ID des algorithmes de signature

OID	*Algorithme de signature*
{1.2.840.113549.1.1.2}	MD2withRSA
{1.2.840.113549.1.1.4}	MD5withRSA
{1.2.840.113549.1.1.5}	SHA1withRSA
{1.2.840.10040.4.3}	SHA1withDSA

4.3 Extensions de certificat X.509

Les extensions ont été rajoutées dans la version 3 du certificat X.509. Elles permettent d'indiquer l'emplacement de la liste des révocations de certificat, l'usage de la clef publique (chiffrement, authentification, signature de message électronique, signature de code, etc.). Une extension peut être marquée comme *critique*. Dans ce cas, il est obligatoire de pouvoir interpréter cette extension lors d'une opération de vérification de certificat. Une extension critique non reconnue doit provoquer l'échec de la vérification.

La définition des extensions de certificat X.509v3 est donnée dans le RFC 3280 [7]. Chaque extension est identifiée par un OID. Il existe des extensions standardisées, et il est possible de créer ses propres extensions privées. Voici la liste des extensions définies dans le RFC 3280 :

Tableau 5 : Extensions X.509 définies dans RFC 3280

OID	*Nom*
2.5.29.35	Authority Key Identifier
2.5.29.14	Subject Key Identifier
2.5.29.15	Key Usage
2.5.29.16	Private Key Usage Period
2.5.29.32	Certificate Policies
2.5.29.33	Policy Mappings
2.5.29.17	Subject Alternative Name
2.5.29.18	Issuer Alternative Name
2.5.29.9	Subject Directory Attributes
2.5.29.19	Basic Constraints
2.5.29.30	Name Constraints
2.5.29.36	Policy Constraints
2.5.29.37	Extended Key Usage

2.5.29.31	CRL Distribution Points
2.5.29.54	Inhibit Any-Policy
2.5.29.46	Freshest CRL
1.3.6.1.5.5.7.1.1	Authority Information Access
1.3.6.1.5.5.7.1.11	Subject Information Access

Authority Key Identifier est utilisé lorsqu'un CA possède plusieurs clefs privées. Dans ce cas, le nom de l'autorité ne suffit plus pour identifier la clef publique à utiliser. Il existe la même extension si le sujet possède plusieurs clefs privées : *Subject Key Identifier*. Cette extension est indispensable lorsqu'une clef est renouvelée périodiquement.

KeyUsage est une des extensions les plus utilisées, et souvent marquée comme critique. Elle indique pour quels usages une clef peut être utilisée. Voici la liste exhaustive des différents usages :

digitalSignature, *nonRepudiation*, *keyEncipherment*, *dataEncipherment, keyAgreement, keyCertSign, cRLSign, encipherOnly, decipherOnly*.

Les usages *digitalSignature* et *nonRepudiation* permettent de signer des objets autres que des certificats *(*usage *keyCertSign*) et autres que les listes de révocation de certificat *(*usage *cRLSign). digitalSignature* permet l'usage de la clef pour authentifier un message et assurer son intégrité. *nonRepudiation* permet l'usage de la clef pour assurer la non répudiation du message par le signataire. La carte d'identité belge contient deux clefs privées. La première ne permet que l'usage *digitalSignature*. La seconde ne permet que l'usage *nonRepudiation* et requiert l'entrée du code PIN à chaque opération. Lorsqu'un citoyen se connecte au site de l'administration belge, il utilise la première clef pour s'authentifier vis-à-vis du serveur (connexion SSLv3 ou TLSv1) et utilise la deuxième clef pour signer des documents avec non répudiation. Le standard recommande de consulter la politique de certificat de l'autorité de certification[12] pour une description plus détaillée de la différence entre *digitalSignature* et *nonRepudiation*.

L'extension *CertificatePolicies* contient une ou plusieurs informations sur la politique de certificat. Une des informations standard est un URI vers le *Certificate Practice Statement* (Enoncé des pratiques de certification) qui fait l'objet du paragraphe suivant.

Subject Alternative Name permet par exemple d'indiquer l'adresse de messagerie, ou l'adresse IP ou nom de domaine (dans le cas d'un serveur) du sujet.

BasicConstraints indique si le sujet est une autorité de certification ou une entité finale.

Extended Key Usage étend les usages possibles de la clef : authentification TLS côté serveur, authentification TLS côté client, signature de code, protection de mail, horodatage, signature de réponses OCSP.

Comme son nom l'indique *CRL Distribution Points* indique l'emplacement des listes de révocation. *Freshest CRL* indique l'emplacement de « Delta CRL » c'est-à-dire une mise à jour d'une autre CRL.

Authority Information Access contient des informations supplémentaires sur l'autorité de certification comme l'adresse du service OCSP, ou l'adresse du certificat de l'autorité supérieure.

4.4 Certificate Practice Statement (CPS)

Lorsque nous recevons un document signé accompagné d'un certificat émanant d'une autorité de certification, quelle confiance avons-nous en cette autorité ? Avons-nous la

[12] Politique de certificat : sera discuté au point 4.4.

même confiance pour tous les certificats émanant de cette autorité ? Quelles sont ses responsabilités ?

En réponse à cela, une autorité de certification publie un document appelé *Certificate Practice Statement (CPS)*. L'idée générale de ce document est décrite dans le RFC 3647 [7].

Le CPS décrit le fonctionnement général de l'autorité de certification. Elle décrit par exemple la manière dont les clients (*subscribers*) s'authentifient auprès de l'autorité d'enregistrement lors de la demande d'un certificat. La plupart des autorités de certification définissent différentes classes de clients en fonction du niveau de sécurité. Dans une classe de haut niveau de sécurité, le client devra par exemple se présenter physiquement au bureau d'enregistrement tandis que dans une classe de bas niveau, une simple vérification d'adresse e-mail sera effectuée. Ce document explique également les mesures de sécurité appliquées au sein du personnel pour la protection des clefs privées de l'autorité, les méthodes de renouvellement de clefs, les méthodes de révocation, l'usage des clefs, etc.

Le CPS comprend également les obligations légales de l'autorité. Elle peut, par exemple, s'engager à couvrir les dommages causés par une erreur d'enregistrement. Ces garanties sont toujours limitées à un certain montant, qui est plus élevé pour des classes à plus haut niveau de sécurité. Un certificat GlobalSign de type PersonalSign Demo 1 ne donnera absolument aucune garantie alors qu'un certificat de type PersonalSign3 donnera une garantie jusqu'à 37500 EUR[13].

Le CPS indique les obligations de la partie confiante ('relying party'). Celle-ci est la partie qui désire utiliser un certificat émanant de cette autorité, ou utiliser les services de révocation. Le plus souvent, une personne doit vérifier que le certificat n'a pas été révoqué, sinon elle ne pourra pas réclamer d'éventuels dommages en cas de conflit.

Il s'agit donc d'un contrat qui lie l'autorité de certification avec ses souscripteurs et avec la partie confiante. Il est évident qu'un logiciel de vérification de signature ne peut pas effectuer la lecture de ce document à notre place !

Les CPS sont généralement publiés sur le site web des sociétés de certification, et couramment à une adresse du type http://nom_de_domaine/repository/

Tableau 6 : adresses des CPS d'organisations réputées

Organisation	*adresse du CPS*
GlobalSign	http://www.globalsign.net/repository/
VeriSign	http://www.verisign.com/repository/
Thawte	http://www.thawte.com/repository/
Administration belge	http://repository.eid.belgium.be/

4.5 Liste de révocation de certificat X.509 CRL, et protocole OCSP

Lorsque la clef privée d'un utilisateur a été volée ou perdue, il est nécessaire de révoquer le certificat. Une liste CRL (*Cerficate Revocation List*) est un document signé par une autorité de certification qui contient une liste des numéros de série des certificats qui ont été révoqués, ainsi que leur date de révocation.

Le protocole OCSP (Online Certificate Status Protocol) défini dans le RFC 2560 (juin 1999), offre une alternative aux listes CRL : lorsqu'un client veut vérifier le statut d'un

[13] http://www.globalsign.net/repository/ Insurance Policy

certificat, au lieu de télécharger une longue liste CRL, il peut interroger un serveur OCSP, aussi appelé répondeur OCSP.

4.6 Hiérarchie de certification

Un certificat X.509 est signé par une autorité de certification digne de confiance. Pour vérifier la signature d'un certificat, il faut utiliser la clef publique de cette autorité de certification. Cette clef doit également être distribuée de manière sûre, par exemple via un certificat signé par une autre autorité de certification. On crée ainsi une chaîne de certification :

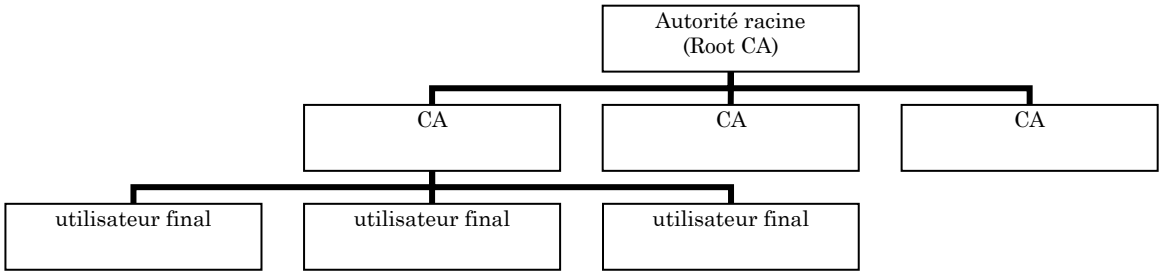


Figure 10 : Exemple d'hiérarchie de certification

L'autorité racine signe son certificat avec sa propre clef. Dans ce certificat racine, le nom de l'émetteur est identique au nom de l'objet.

Lors de la validation d'une signature, il faut réaliser deux opérations :

- vérifier la signature du document avec la clef publique du signataire : calculer à nouveau l'empreinte des données, la chiffrer avec la clef publique et comparer le résultat avec la signature du document.
- vérifier la validité de la clef publique :
    - établir la chaîne de certificats en cherchant de manière répétée le certificat dont le nom du sujet est égal au nom de l'émetteur du certificat précédent.
    - pour chaque certificat de cette chaîne :
        - vérifier la signature du certificat,
        - vérifier la période de validité,
        - vérifier les extensions marquées comme critiques,
        - contrôler que le certificat n'a pas été révoqué.

Le certificat racine doit être préalablement installé sur la machine de manière sûre. Il faut s'assurer de son origine. Une politique courante des éditeurs de systèmes d'exploitation est d'installer, par défaut, des certificats racines de sociétés réputées comme VeriSign, Thawte, …Cependant, l'utilisateur devrait consulter les CPS de ces sociétés lorsqu'il utilise leur certificat pour vérifier une signature.

5 Signature électronique : formats, services, et législation

Ce chapitre traite des points suivants : format standards de signature (PKCS#7, CMS, XML Signature,…), service d'horodatage, service de stockage de documents signés et législation européenne et belge.

5.1 Format de signature de document

5.1.1 PKCS #7, CMS

PKCS#7 *v1.5* (*Public Key Cryptography Standard*), nommé également *Cryptographic Message Syntax*, est un standard de RSA Labs publié en novembre 1993, qui définit un format pour l'échange ou le stockage de données cryptographiques (signées, chiffrées, signées et chiffrées, etc.) ou simplement l'échange ou le stockage de certificats (dans ce cas, aucune donnée n'est signée ou chiffrée).

Ce standard définit un type de donnée `ContentInfo` possédant deux éléments : le type de contenu `ContentType` et le contenu lui-même `content`.

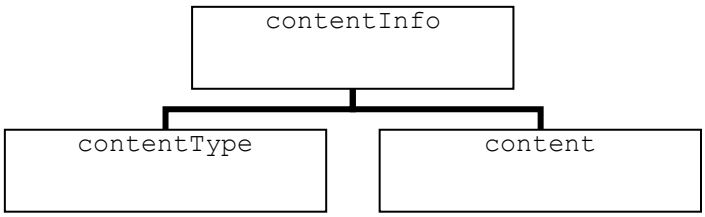


Figure 11 : PKCS#7 - Structure de base

`ContentInfo` peut avoir 6 types de contenu différents : `data`, `signedData`, `envelopedData`, `signedAndEnvelopedData`, `digestedData`, et `encryptedData`.

Les types `envelopedData` et `encryptedData` sont tous les deux utilisés pour l'échange de données chiffrées. Dans le premier cas (`envelopedData`), on chiffre le message avec une clef aléatoire et on chiffre également cette clef avec la (les) clef(s) publique(s) du (des) destinataire(s). Dans l'autre cas (`encryptedData`), on chiffre simplement le message avec une clef donnée, et la distribution des clefs doit se faire par d'autres moyens.

Analysons le cas où le contenu est de type `SignedData` :

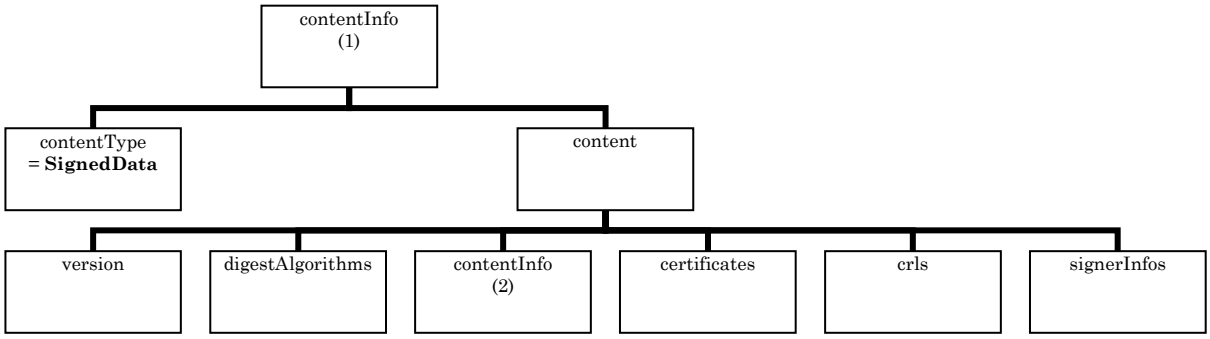


Figure 12 : PKCS#7 - Structure détaillée avec `signedData`

Notez que *content* contient lui-même un élément `contentInfo` (2) qui peut contenir un des six éléments déjà cités. Cette flexibilité permet d'imbriquer à volonté des objets `contentInfo`.

- `version` indique la version de l'objet.
- `digestAlgorithms` contient l'ensemble non ordonné des algorithmes de hachage utilisés pour signer le contenu de `contentInfo` (2). Plusieurs signataires peuvent utiliser des algorithmes de hachage différents.
- `contentInfo(2)` contient les données à signer. En général il s'agit d'un objet `data` (les données originales qu'on désire signer).
- `certificates` est l'ensemble, non ordonné, des certificats des signataires et éventuellement d'autres certificats faisant partie de la chaîne. Les certificats sont placés 'en vrac' dans cet ensemble et il peut y avoir plus de certificats que nécessaire. Il s'agit en quelque sorte d'un magasin de certificats. Lors de la vérification il faudra rechercher la chaîne de certificat pour chaque signataire.
- `crls` (***c**ertificate **r**evocation **l**ist**s***) est un ensemble non ordonné de listes de révocation de certificats.
- `signerInfos` contient des informations sur chaque signataire : algorithme de hachage, identification du certificat (nom de l'émetteur du certificat et numéro de série), signature, et éventuellement d'autres paramètres optionnels signés (par exemple la date de signature), ou non signés (par exemple une contresignature). Le type `SignedData`, `signerInfos` peut ne pas contenir d'élément (cas particulier où encore aucune personne n'a signé les données, ou lorsqu'on ne veut transmettre que des certificats ou listes de révocation).

Il est également possible de créer une *signature externe*, c'est-à-dire détachée du message. Dans ce cas `contentInfo` *(2)* ne contient pas d'élément. Ceci est pratique pour éviter la réplication des données stockées sur bande ou support non réinscriptible, ou pour des applications réseau où il est nécessaire de ne renvoyer que la signature (ceci est le cas du projet, voir chapitre 7).

PKCS#7 *v1.5* a été publié en mars 1998 dans le RFC 2315 de l'IETF. Depuis lors, l'IETF a repris le travail de développement et de maintenance et ce standard a évolué sous le nom *CMS* (pour *Cryptographic Message Syntax*) :

- RFC 2630 [CMS1] (juin 1999),
- RFC 3369 [CMS2] (août 2002),
- RFC 3852 (juillet 2004).

5.1.2 S/MIME

Le standard de courrier électronique sécurisé S/MIME version 2 est basé sur PKCS#7 v1.5 (ou RFC 2315), tandis que S/MIME version 3 est basé sur le RFC 2630 [CMS1].

S/MIME permet la signature ou le chiffrement de données MIME[14]. La signature peut se faire de deux manières :

- soit le contenu MIME est encapsulé dans la signature CMS (dans `contentInfo`). Dans ce cas, si le logiciel de messagerie du récepteur n'implémente pas S/MIME, le récepteur ne saura pas lire le message.

[14] MIME Multipurpose Internet Mail Extensions.

– soit le contenu MIME est détaché du message (signature externe). Cette deuxième solution est la plus utilisée car elle permet aux anciens logiciels de messagerie qui n'implémentent pas S/MIME de pouvoir lire le message. La signature se présente comme un fichier attaché. Un exemple du deuxième type est repris en annexe (page 65).

Notez que les en-têtes du message ne sont pas signés. Le sujet du message et les adresses e-mail de l'émetteur et du destinataire ne seront pas signés. Ceci est dû au fait que les agents de transfert de courrier (les serveurs SMTP) sont susceptibles de modifier ces en-têtes.

Un problème que le standard S/MIME a dû surmonter est le fait que l'encodage du contenu MIME peut changer lors de son transfert entre serveur SMTP. Par exemple, le retour chariot n'est pas encodé de la même manière sur Windows ou Linux. Lors de la vérification de la signature, il faut procéder à un réencodage selon les règles définies par S/MIME.

5.1.3 XML Signature

« *XML-Signature Syntax and Processing* » est un standard produit par le groupe de travail XML Signature de l'IETF/W3C. Ce standard définit une signature pour du contenu XML ou tout autre type de document (fichier binaire, etc.) et est publié sur le site officiel du W3C[15].

Il existe trois types de signature :

– une signature enveloppée : la signature XML est incluse dans le document XML signé.

– une signature enveloppante : la signature XML contient le document signé XML.

– une signature détachée : la signature fait référence à un ou plusieurs documents signés qui peuvent être de n'importe quel type.

Une signature XML a la structure suivante (où « ? » signifie zéro ou une occurrence, « + » signifie une ou plusieurs occurrences, et « * » signifie zéro, une ou plusieurs occurrences) :

```
<Signature ID?>
   <SignedInfo>
     <CanonicalizationMethod/>
     <SignatureMethod/>
     (<Reference URI? >
       (<Transforms>)?
       <DigestMethod>
       <DigestValue>
     </Reference>)+
   </SignedInfo>
   <SignatureValue>
  (<KeyInfo>)?
  (<Object ID?>)*
 </Signature>
```

Figure 13 : signature XML

SignedInfo contient les informations sur les données à signer. Chaque *Reference* possède

[15] Voir référence [13] ou http://www.w3.org/TR/xmldsig-core/

- une URI (*Uniform Resource Identifier*) d'un fichier binaire ou de données XML (sauf si les données XML sont incluses dans la référence-même).
- une séquence de transformations (*Transforms*) à effectuer avant la fonction de hachage. Ces transformations permettent par exemple de convertir des données XML en données binaires, ou de sélectionner seulement une partie des données (pour signer une partie d'un formulaire).
- une empreinte numérique *DigestValue* définie par son type *DigestMethod*.

SignedInfo contient également la mise en forme canonique *CanonicalizationMethod* pour convertir la structure XML *SignedInfo* en données binaire.

L'utilisation des transformations de données représente un danger. Une personne malveillante pourrait définir une transformation qui donne toujours le même résultat. Il pourrait alors substituer n'importe quelles données aux données originales, la signature restant valide. Lors de la vérification, il faut donc considérer comme 'signées' non pas les données avant transformation mais après transformation.

La forme canonique permet de surmonter le même problème qu'avec CMS et S/MIME car deux documents XML qui représentent la même information ne sont pas nécessairement encodés de la même manière.

L'élément *KeyInfo* peut contenir des certificats, une clef publique, etc. Contrairement à CMS qui peut contenir n'importe un ensemble de certificats quelconques, les certificats de *KeyInfo* doivent nécessairement appartenir à la chaîne de certificat de la clef utilisée dans la signature. Par contre, tous les certificats ne doivent pas être présents et il n'y a pas d'ordre défini entre les certificats. Si *KeyInfo* n'est pas donné, la clef doit être identifiée par d'autres moyens.

La validation d'une signature XML est réalisée en deux étapes :

- pour chaque référence, effectuer les transformations nécessaires et vérifier l'empreinte numérique,
- mettre *SignedInfo* sous forme canonique et vérifier la signature dans *SignatureValue* avec la clef publique du signataire.

XML Signature, permet comme CMS de signer des données supplémentaires comme la date de signature. Dans ce cas, la signature XML contient une référence vers un élément *SignatureProperties*.

5.1.4 XadES (XML Advanced Electronic Signature)

XadES, spécification technique de l'ETSI en cours d'évolution, définit une extension de XML Signature afin de créer des *signatures électroniques avancées* telles que définies dans la « directive 1999/93/CE du Parlement européen et du Conseil du 13 décembre 1999 sur un cadre communautaire pour les signatures électroniques ». Voir section 5.4.

La référence de cette spécification est « ETSI TS 101 903 V*x.y.z* » où V*x.y.z* est la version du document :

- V1.1.1 : http://uri.etsi.org/01903/v1.1.1/ (février 2002). Publié aussi comme note du W3C sur http://www.w3.org/TR/XAdES/
- V1.2.1 : non disponible (mars 2004)
- V1.2.2 : http://uri.etsi.org/01903/v1.2.2/ (avril 2004)
- V1.3.2 : http://uri.etsi.org/01903/v1.3.2/ (mars 2006)

5.1.5 PGP, OpenPGP

PGP (Pretty Good Privacy) est un logiciel commercial qui offre des services de signature électronique et de confidentialité pour les courriers électroniques et les fichiers de données. Son fonctionnement repose également sur des opérations cryptographiques à clefs asymétriques. PGP a été créé par Philip Zimermann, et sa version 1.0 est sortie en 1991. L'IETF a publié en 1998 le standard OpenPGP (sur base de PGP 5.x), laissant la porte ouverte à d'autres implémentations libres comme *Gnu Privacy Guard* (GPG).

Les messages PGP signés et/ou chiffrés sont convertis en base 64 avant d'être transférés. Afin de reconnaître le message PGP, un entête contenant `BEGIN PGP MESSAGE` est ajouté au début du message. PGP n'utilise pas ASN.1 ni les certificats X.509. Le système PGP possède son propre format de données[16]. Il possède également sa propre définition de certificat.

De manière similaire à S/MIME, PGP permet la signature de message électronique. Les signatures sont détachées du document signé. Une application de messagerie ne supportant pas PGP pourra tout de même visualiser le message.

PGP possède une structure de certification particulière : n'importe qui peut facilement signer le certificat d'une autre personne et jouer ainsi le rôle d'autorité de certification.

5.2 Service d'horodatage

L'horodatage consiste à apposer, à un document électronique, une date signée par une tierce partie. Ceci apporte la preuve que le document existait avant le moment de l'horodatage. En général, une empreinte digitale du document est envoyée à un serveur d'horodatage (*Time Stamp Authority*), celui-ci ajoute la date actuelle et signe le tout.

Il existe des protocoles standards d'horodatage :

- RFC3161 de l'IETF (août 2001) : Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP)
- OASIS (Organization for the Advancement of Structured Information Standards) Digital Signature Service (DSS) XML Timestamp : *http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=dss*

5.3 Service d'archivage et journalisation

L'archivage des documents à valeur probatoire au sein d'une entreprise est un problème important. Les coffres forts électroniques émergent sur le marché. Souvent, une entreprise ayant le besoin de stocker des documents importants (archivage de factures, de contrats, etc.) fait appel à des sociétés tiers d'archivage. Celles-ci mettent en place des systèmes de stockage redondants et dont l'accès est sécurisé. Elles prennent sur elles la responsabilité du stockage (dédommagement en cas de pertes de données). L'archivage demande donc un investissement non négligeable.

La journalisation consiste à garder en mémoire toutes les opérations effectuées, envoi, réception, archivage, destruction d'un document, etc. La journalisation est importante pour la sécurité.

5.4 Législation en matière de signature

La signature électronique peut-elle remplacer légalement la 'signature classique sur papier' ? En réponse à cela, une directive européenne à été publiée : *Directive*

[16] voir syntaxe de paquets dans [14] page 11.

1999/93/CE du Parlement européen et du Conseil du 13 décembre 1999 sur un cadre communautaire pour les signatures électroniques. Celle-ci est accessible sur le site officiel http://eur-lex.europa.eu/ (code CELEX : 31999L0093).

Cette directive a pout but la reconnaissance juridique des signatures électroniques dans les différents états-membres, et fixe les règles pour l'accréditation des prestataires de service de certification (c'est-à-dire les autorités de certification).

Elle définit deux types de signature : la signature électronique, et la signature électronique avancée. Cette dernière doit satisfaire à certaines exigences : par exemple, elle doit être créée avec des moyens que le signataire puisse garder sous son contrôle exclusif, et doit avoir un certificat qualifié (voir plus bas). Ces deux types de signature se distinguent par la charge de la preuve. Dans le cas d'une signature électronique avancée, la signature est présumée fiable et c'est la personne qui conteste la signature qui doit en apporter la preuve. Par contre, dans le cas d'une signature électronique 'simple', c'est l'autre personne qui doit prouver que cette signature est bien fiable.

Cette directive introduit la notion de « certificat qualifié ». Le prestataire de service de certification qui délivre des certificats qualifiés doit répondre à des exigences, comme faire preuve d'être suffisamment fiable. Afin d'identifier un certificat qualifié par rapport à un certificat ordinaire, l'IETF a publié le RFC 3039 « Internet X.509 Public Key Infrastructure – Qualified Certificates Profile ». Chaque carte d'identité belge électronique contient un tel certificat qualifié.

La directive prévoit la création de régimes d'accréditation des prestataires par les autorités afin d'améliorer le niveau de service fourni.

Cette directive a été retranscrite dans le droit belge par une loi publiée le 9 juillet 2001 : *Loi fixant certaines règles relatives au cadre juridique pour les signatures électroniques et les services de certification*. L'article 4 de cette loi contient :

> *Sans préjudice des articles 1323 et suivants du Code civil, une signature électronique avancée réalisée sur la base d'un certificat qualifié et conçue au moyen d'un dispositif sécurisé de création de signature électronique, est assimilée à une signature manuscrite, qu'elle soit réalisée par une personne physique ou morale.*

6 Quelques produits existants

La signature électronique est en pleine effervescence. Les outils de développement et les produits finis sont de plus en plus nombreux sur le marché. Ce chapitre donne un aperçu de quelques produits existants. Il en existe encore beaucoup d'autres.

6.1 La carte d'identité belge

La carte d'identité belge a été créée afin de moderniser l'administration belge. Elle permet par exemple au citoyen belge de remplir sa déclaration fiscale via le site fédéral *TaxOnWeb*. Cette carte possède une puce électronique contenant diverses données dont :

- les informations relatives au titulaire : nom, prénoms, date et lieu de naissance, adresse, nationalité, numéro de carte, etc. La fenêtre ci-dessous illustre ces informations.
- deux clefs privées et les certificats associés. Une clef est utilisée pour l'authentification (exemple : connexion authentifiée au portail de l'administration belge). L'autre clef est utilisée pour la signature de document avec non répudiation.

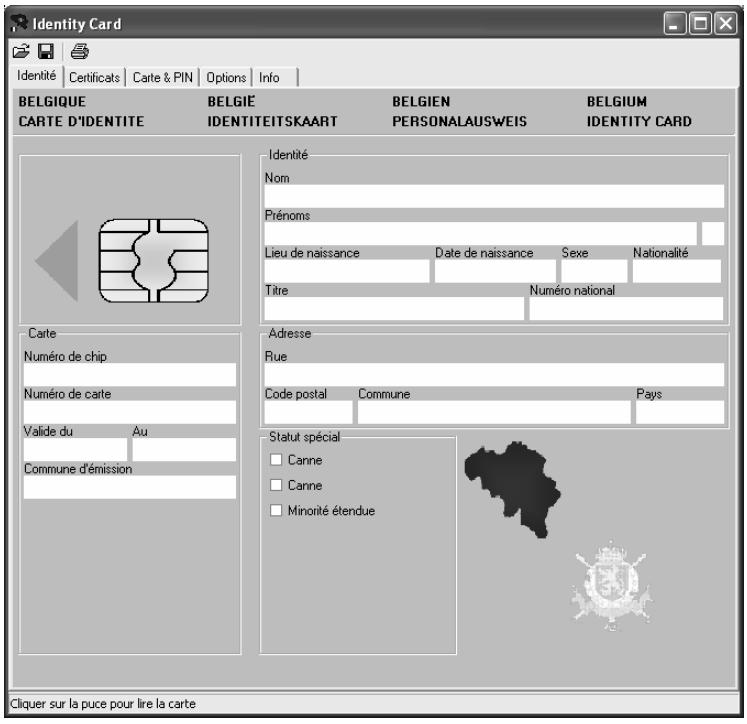


Figure 14 : Logiciel Identity Card

L'accès aux clefs privées et certificats se fait via un middleware développé pour l'Administration belge par la société Zetes. Ce middleware permet la communication entre un logiciel implémentant des fonctionnalités de sécurité (par exemple la signature électronique) et la carte à puce où les opérations cryptographique sont réellement effectuées (la clef privée ne peut être extraite de la puce). La figure ci-dessous est extraite de [15] :

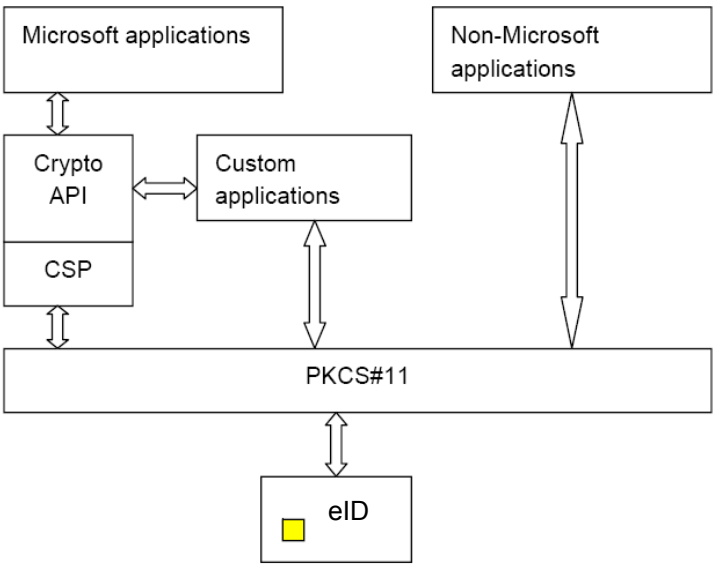


Figure 15 : Architecture du middelware eID

Le middleware offre deux accès possibles :

- via une bibliothèque PKCS#11 (aussi bien sous Windows que Linux). Le standard PKCS#11 créé par le laboratoire RSA définit une interface de type C entre une application et tout type de '*token*' cryptographique (carte à puce, clef USB, etc.). La bibliothèque PKCS#11 de la carte d'identité belge est basée essentiellement sur le projet libre OpenSC.
- via le *Belgian eID CSP* (*Cryptographic Service Provider*) développé pour Microsoft CryptoAPI. Ce CSP utilise lui-même la bibliothèque PKCS#11. Naturellement, cette solution n'est valable que sur la plateforme Windows.

Lors de l'installation du middleware (téléchargeable sur http://eid.belgium.be), des outils supplémentaires sont également installés sous Windows. Ceux-ci permettent d'installer facilement une référence vers les clefs privées de la carte d'identité ainsi que leurs certificats dans le magasin de certificats de Windows. Lorsque l'utilisateur désire s'authentifier sur un site Internet via Internet Explorer, ou désire signer un document dans Microsoft Office (Word, Outlook, …), il sélectionne simplement le certificat voulu dans son magasin de certificat Windows.

Les logiciels de la famille Mozilla (Firefox, Thunderbird,...) possèdent leur propre magasin de clefs et certificats. Ces logiciels ont l'avantage d'être multiplateformes, mais ont l'inconvénient de ne pas supporter Microsoft CryptoAPI sous Windows. Par contre l'utilisation de la carte d'identité belge reste possible avec l'utilisation de l'interface PKCS#11. L'utilisateur doit ajouter lui-même le module PKCS#11 en sélectionnant le fichier DLL du middleware dans le répertoire *system32* de son installation Windows.

6.2 Keyvelop

Le mercredi 12 juillet 2006 à Evere (MRC&I, Capt. Raeves) a eu lieu une présentation du produit *Keyvelop* par la firme *Serve Consult*, à laquelle j'ai eu la chance de participer.

Keyvelop est un service permettant d'envoyer une sorte de lettre électronique recommandée : l'expéditeur est averti de la bonne réception du message, et le destinataire ne peut nier avoir reçu le message. Voici son fonctionnement :

- l'expéditeur du message s'identifie auprès du service tiers appelé bureau de poste, et envoie une empreinte (message digest) du message au bureau de poste. La connexion avec le bureau de poste est sécurisée (connexion SSL).
- le bureau de poste fourni à l'expéditeur une clef de session permettant de chiffrer le contenu du message, ainsi qu'une signature de l'empreinte (horodatage). Le bureau de poste journalise cette action dans un journal.
- le client chiffre le message avec la clef de session et envoie le message au destinataire via un canal quelconque de transmission (messagerie électronique, disquette, CD, FTP, …)
- le destinataire ne peut déchiffrer le message qu'après avoir reçu la clef de session auprès du bureau de poste. Pour cela il doit à son tour s'identifier auprès du bureau. Cette requête est enregistrée dans un journal, et permet l'envoi d'un accusé de réception à l'expéditeur. Le destinataire ne peut donc pas nier avoir reçu le message.
- Le service offre ensuite la possibilité au destinataire de signer le message reçu.

Keyvelop permet de contrôler l'authenticité du message et de vérifier son intégrité grâce à la signature électronique de celui-ci.

L'identité de l'expéditeur peut être contrôlée à divers niveau de sécurité : soit par simple mot de passe, soit avec une authentification forte (clef privé hardware…). Le destinataire doit être inscrit auprès du service *Keyvelop*. Dans le cas contraire, il doit prouver son identité au service : en se rendant physiquement sur place, ou en s'authentifiant grâce à un certificat. C'est l'expéditeur qui choisit le niveau de sécurité du contrôle de l'identité du destinataire.

Keyvelop ne joue pas le rôle d'autorité de certification et ne fournit donc pas de certificat pour des clefs asymétriques de chiffrement, d'authentification, ou de signature. Pour cela, le client doit s'adresser à d'autres services agréés.

Le service *Keyvelop* journalise tous les événements : scellement du message par l'expéditeur, ouverture du message par le destinataire, signature ou révocation par le destinataire. Cette journalisation peut servir de preuve de bonne réception du message par le destinataire.

Enfin, la manière dont le message est envoyé à l'expéditeur est complètement libre : mail, support CD, FTP, site web, … Le message ne transite pas par le bureau de poste.

En résumé, le bureau de poste *Keyvelop* joue le rôle de personne de confiance qui journalise l'envoie et la réception des données. Il utilise le chiffrement comme astuce pour empêcher le destinataire de lire le message sans en avertir le bureau de poste.

Keyvelop utilise le standard S/MIME pour signer et chiffrer les documents.

6.3 Adesium

Adesium est une société française spécialisée dans la signature électronique et l'archivage de documents. Parmi les outils proposés, citons :

- *Adesium Aliso Sign* qui permet la signature de formulaire en ligne.
- *Adesium mySign* qui permet la signature, ou contre signature de simples documents au format CMS.

Figure 16 : Principe de fonctionnement d'Adesium Aliso Sign

6.4 Infomosaic

Infomosaic est une société américaine également spécialisée dans la signature électronique. Elle propose des outils de programmation pour divers langages de programmation.

Informosaic est utilisé par l'armée américaine pour la signature de certains formulaires.

Cette société a développé un applet Java permettant la signature de document à partir d'un site web. Nous avons étudié son fonctionnement avec la méthode de 'reverse engineering' à partir d'une démo accessible sur leur site internet.

L'applet Java ne fonctionne que sous Windows, car il fait appel à la bibliothèque Microsoft CryptoAPI. La plus grande partie du code est implémentée en C++ et accédé en Java via l'interface native Java (JNI ou *Java native Interface*). Cet applet est nécessairement signé, car il doit accéder à des ressources non accessibles par un applet non signé (enfermé pour des raisons de sécurité dans un 'bac à sable'). Par exemple un applet non signé ne peut communiquer en réseau que vers le serveur où il a été téléchargé, et ne peut pas accéder au système de fichiers.

Informosaic utilise la signature XML. Le document signé se présente comme un document XML compressé au format gzip. Le fichier signé est inclus dans le document XML et encodé en base 64.

6.5 Microsoft InfoPath

Le logiciel Microsoft InfoPath est conçu spécialement pour créer et remplir des formulaires. InfoPath utilise le standard XML. L'enregistrement du modèle de formulaire avec l'extension `.xsn` n'est rien d'autre qu'un fichier compressé CAB (Cabinet) contenant divers fichiers XML. Lors du remplissage d'un formulaire, les données sont également enregistrées dans un fichier XML. Ces données sont donc facilement exploitables dans d'autres applications.

InfoPath permet la signature du contenu complet ou partiel d'un formulaire, au format XML Signature.

6.6 OpenOCES OpenSign

OpenSign est un applet Java, d'origine danoise, permettant la signature en ligne de documents. Son code source est disponible sur Internet (www.openOCES.org).

Cet applet accède aux ressources Microsoft CryptoAPI via l'interface native de Java JNI, de façon similaire à Infomosaic. Il génère des signatures au format XML Signature, en

utilisant ses propres outils : NanoXML 2 Lite et une implémentation de XML Signature propre à OpenOCES.

L'implémentation Java pour accéder aux ressources de Microsoft CryptoAPI n'est pas basée sur l'architecture cryptographique de Java (voir paragraphe 7.6). Ceci rend sa réutilisation peu enviable par d'autres programmeurs, et empêche son utilisation directe avec des outils de signature XML existant comme Apache XML Signature.

7 Développement d'un outil de signature en ligne

7.1 Objectifs de la partie pratique

La partie pratique de ce mémoire consiste à développer une application permettant de signer facilement un document en ligne. Le besoin est formulé de cette manière : « une personne remplit un formulaire sur une page web, et le signe avant de l'envoyer à son destinataire. ».

Plusieurs outils permettent de remplir et signer un formulaire (formulaire PDF avec Adobe Acrobat, formulaires XML avec Microsoft Office InfoPath). Dans notre cas, nous nous intéressons aux formulaires HTML. Ces derniers sont remplis par l'utilisateur dans un navigateur tel que *Internet Explorer*, ou *Mozilla*.

Voici quelques exigences concernant l'outil à développer :

- il doit générer des signatures avec un format standardisé, de façon à ce que son analyse soit aisée par un expert, par exemple lors d'un litige entre le signataire et le destinataire.
- il doit permettre d'accéder à divers outils cryptographiques logiciels ou matériels comme la carte d'identité belge électronique.
- il doit s'installer facilement, et, si possible, être compatible avec d'autres systèmes d'exploitation que Microsoft Windows.
- sous Microsoft Windows, il doit pouvoir accéder au magasin de certificats de Windows.
- il doit être conçu de manière à s'assurer que le signataire puisse visualiser clairement ce qu'il signe (« WYSIWYS : *What you see is what you sign* »).
- il doit permettre au signataire de conserver une copie du document signé.
- il doit être documenté aussi bien pour l'utilisateur, que pour le développeur.
- il sera publié comme logiciel libre et protégé par une licence *opensource* adéquate.

7.2 Processus de développement

Le développement d'un logiciel comprend plusieurs phases. Les phases essentielles sont :

- l'analyse : fixer les objectifs et délimiter le problème, fixer les exigences à atteindre, analyser et sélectionner les outils nécessaires, créer l'architecture du programme, etc.
- l'implémentation : création du code.
- les tests.

Ces phases ne doivent pas nécessairement se dérouler séparément. Certains tests peuvent être indispensables durant l'analyse afin de déterminer quels outils conviennent le mieux. De plus, il est important de bien analyser l'ensemble du problème avant de débuter l'implémentation. Dans notre cas, le choix du langage de programmation est crucial.

Ce chapitre contient les éléments-clés de l'analyse, et donne le résultat de certains tests d'interopérabilité. Pour plus de détails sur l'analyse et l'implémentation, le lecteur intéressé peut consulter les fichiers sources disponibles sur le site internet (voir introduction).

7.3 Choix du langage de programmation

Le choix du langage de programmation dépend de plusieurs facteurs :

- outils disponibles, et leurs coûts,
- connaissances acquises du programmeur,
- performance de l'application : un logiciel C++ bien conçu aura probablement de meilleures performances qu'un logiciel écrit en Java, ou un logiciel interprété (Python, PHP).

Le premier facteur limite déjà fortement le nombre de langages possibles. Quels langages permettent le développement de logiciels 'embarqués' sur une page web, qui peuvent accéder aux ressources de Windows ? Java, ActiveX et JavaScript sont des solutions possibles pour le développement d'une telle application.

JavaScript et ActiveX permettent d'accéder au magasin de certificat Windows. Pour cela, Microsoft fournit l'outil CAPICOM, un composant COM utilisable avec divers langages dont JavaScript, VBScript, VisualBasic, .NET et Java. Par contre, un client Linux ne pourra pas utiliser l'application.

Java dispose d'une architecture cryptographique bien développée. Malheureusement, Java étant de manière innée 'multiplateforme', les outils existants ne donnent pas facilement accès aux ressources de Windows.

D'autre part un applet Java peut être signé par son éditeur. Ceci permet d'héberger un applet Java sur un site peu sûr. L'éditeur du logiciel ne doit donc plus nécessairement héberger lui-même l'application. Ceci est l'objet du point suivant.

Notre choix s'est porté sur Java, mais à une condition stricte : développer « l'outil manquant » permettant d'utiliser les ressources cryptographiques de Windows.

7.4 Tiers de confiance et signature de logiciel

Les sites marchands font souvent appel à des services tiers qui authentifient le client auprès de leur banque. Le client envoie les informations sensibles (login, mot de passe, numéro de carte, numéro provisoire de carte, numéro Digipass, etc.) directement à la personne de confiance via une connexion sécurisée, sans passer par le site marchand.

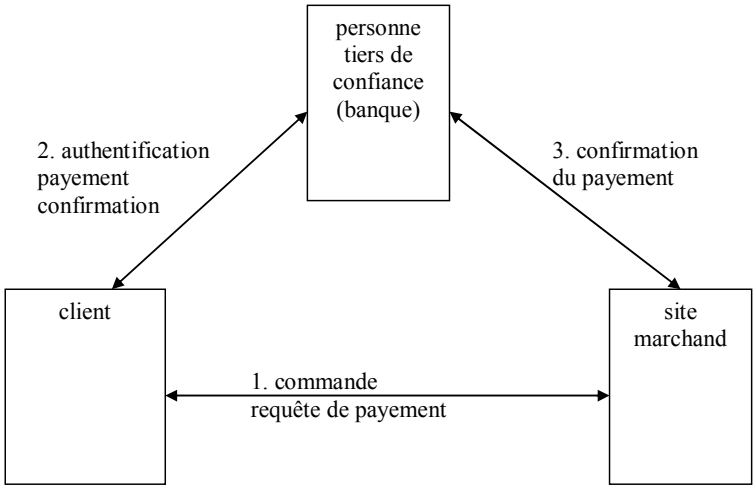


Figure 17 : schéma de principe du payement électronique sur Internet

Le principe de tiers de confiance dans le cadre de la signature électronique est également appliqué avec le *e*-notaire ou notaire électronique. Celui-ci joue le même rôle qu'un notaire ordinaire : stocker le document signé, horodater le document (garantir la date de la signature), garantir la bonne réception du document signé, etc.

Le logiciel de signature est un logiciel sensible : il doit recevoir des privilèges pour accéder à la clef privée du signataire et doit afficher au signataire le document qui sera réellement signé. Une personne malveillante pourrait envoyer un logiciel qui signe un autre document que celui présenté à l'écran. Le signataire doit donc être sûr de l'origine du logiciel. Pour garantir la fiabilité de l'application, l'existence d'une personne tierce de confiance est nécessaire : l'éditeur de logiciel. Celui-ci peut, par exemple, héberger lui-même l'outil de signature sur un site sécurisé auquel se connecte le signataire :

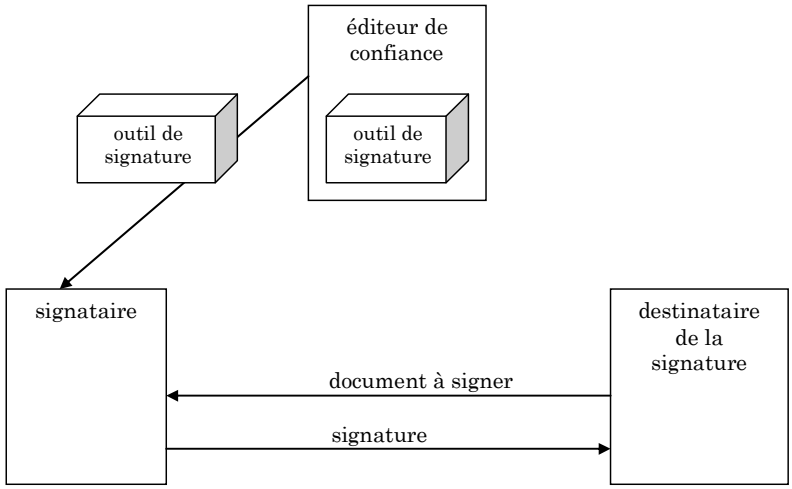


Figure 18 : Hébergement du logiciel par l'éditeur de confiance

Java et ActiveX permettent une autre approche grâce à la signature de logiciel. L'éditeur de confiance peut signer électroniquement son logiciel de tel sorte que l'hébergement du logiciel signé puisse se faire ailleurs, de façon non sécurisée. Dans ce cas, le destinataire de la signature peut lui-même héberger le logiciel de l'éditeur de confiance. Si le logiciel est modifié, la signature de l'éditeur de confiance ne sera plus valide, et le client refusera l'exécution de l'application.

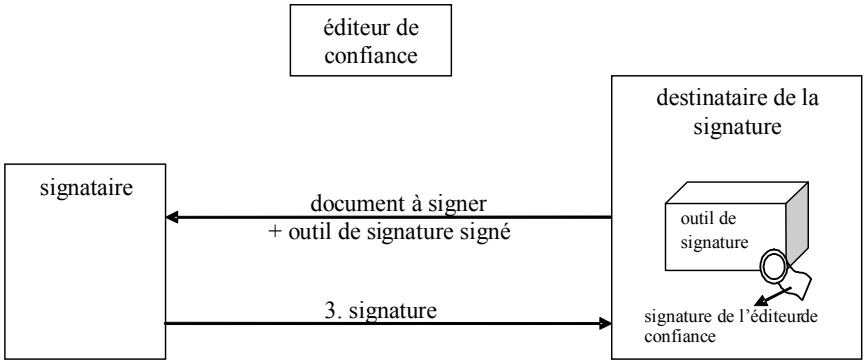


Figure 19 : Hébergement d'un logiciel signé

Afin de protéger son investissement et garantir que n'importe qui n'héberge et n'abuse pas de son outil de signature, l'éditeur de confiance peut empêcher l'exécution de l'outil de signature s'il n'est pas hébergé sur un site autorisé. Un applet Java a la possibilité de connaître l'adresse du serveur à partir duquel il a été téléchargé.

7.5 Déroulement général d'une signature

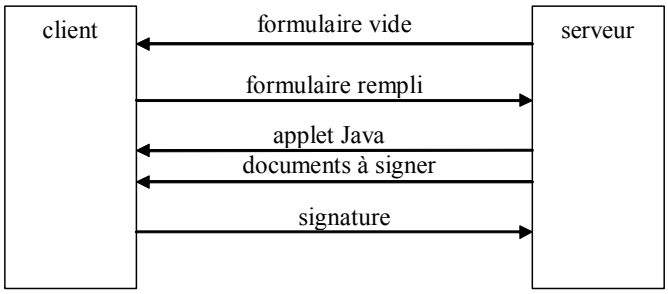


Figure 20 : déroulement général d'une signature

Après avoir complété le formulaire, le client envoie les informations au serveur. Celui-ci génère le document à signer. Ensuite, le client télécharge l'applet Java. Cet applet s'occupe lui-même du téléchargement des documents à signer, accède à la clef privée du client, signe l'ensemble des documents, et envoie la signature au serveur.

Définissons deux types de configuration du serveur :

- serveur *avancé* : serveur avec capacité de vérification de signature. Si la signature est valide (la clef publique fournie correspond à la clef privée qui a signé le document) et si la validation de la chaîne de certificats est réussie (le destinataire fait confiance au certificat du client), une telle configuration pourrait permettre une 'automatisation' de la tâche après la signature, c'est-à-dire la mise en application du contrat (par exemple, la livraison d'une commande).
- serveur *simple* : serveur n'ayant pas la capacité de vérification de signature. Le serveur ne fournit, comme service, que le stockage de la signature. Le destinataire devra contrôler lui-même la signature au moyen d'une application de vérification de signature.

Un serveur avancé doit bénéficier d'outils cryptographiques pour vérifier de la signature. Les bibliothèques PHP existantes ne semblent pas répondre à ces besoins. Une bonne solution est l'utilisation d'un servlet Java J2EE (application Java tournant sur un serveur). Il existe des logiciels libres comme JBoss et Tomcat qui permettent la mise en œuvre d'un tel serveur.

Dans ce projet nous nous sommes limités à un serveur PHP simple qui stocke les signatures. Le destinataire devra vérifier lui-même la signature à l'aide d'une application que nous avons développée.

7.6 Architecture cryptographique de Java

Java possède une architecture cryptographique permettant d'implémenter des fournisseurs de services cryptographiques (*cryptographic service provider*) avec une interface bien définie.

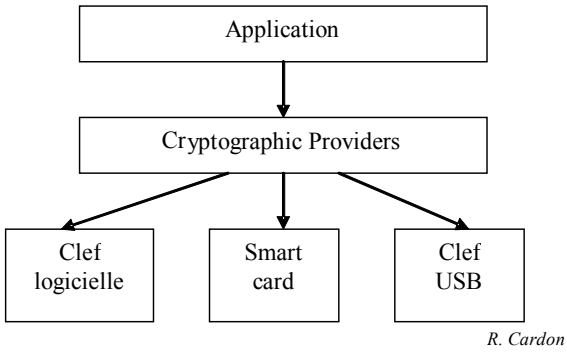


Figure 21 : architecture cryptographique de Java

Les clefs privées peuvent être stockées de manière logicielle (un fichier au format PKCS#8 ou 12, ou un magasin de clefs du système d'exploitation) ou matérielle (stick USB, carte à puce, …).

Un provider peut fournir divers services : calcul d'une empreinte numérique, chiffrement, signature, magasins de clefs et de certificats, création de clefs, etc.

Illustrons de manière simplifiée l'utilisation d'un provider de signature. On crée d'abord l'objet Signature avec l'algorithme de signature désiré (par exemple SHA1withRSA) et en option le nom du provider (le provider préférentiel est utilisé si le nom n'est pas spécifié) :

```
Signature sig = Signature.getInstance(String algorithm);
Signature sig = Signature.getInstance(String algorithm, String provider);
```

Puis, on initialise l'objet signature selon l'opération souhaitée (signature ou vérification) :

```
sig.initSign(PrivateKey privateKey);
sig.initVerify(PublicKey publicKey);
```

On appelle ensuite la fonction `update` de façon récursive sur l'ensemble des données à signer ou à vérifier :

```
sig.update(byte[] data);
```

Enfin, on appelle la fonction *sign* ou *verify* :

```
byte[] signature = sig.sign();
boolean sigValid = sig.verify(byte[] signature);
```

7.7 Accès en Java aux ressources cryptographiques de Windows

Comme annoncé précédemment, l'architecture cryptographique de Java possède un inconvénient : elle ne permet pas à priori d'accéder aux ressources cryptographiques de Windows. L'implémentation de Sun possède son propre magasin de certificat et de clefs. Un utilisateur qui a installé une clef privée dans le magasin de Windows ne pourra pas l'utiliser directement avec les services cryptographiques fournis par Sun. Pour accéder à ces ressources, il va falloir créer notre propre provider.

7.7.1 CryptoAPI, CAPICOM et .NET

En annexe, nous montrons deux manières pour un utilisateur de visualiser son magasin de certificats et de clefs dans Microsoft Windows. Mais comment y accéder de façon programmable ?

Trois possibilités nous est offertes actuellement, par ordre de parution :

- CryptoAPI
- CAPICOM
- classes cryptographiques de la plateforme .NET

CryptoAPI est une des API de Windows (Application Programmable Interface). Elle offre une interface du type langage C, à l'aide d'une bibliothèque DLL (wincrypt.dll). Cette interface est donc essentiellement utilisée pour des programmeurs C ou C++. Cette interface sera probablement de moins en moins utilisée pour faire place à .NET. Microsoft n'a cependant pas l'intention d'arrêter son support pour CryptoAPI.

CAPICOM est un objet COM (Component Object Model[17]), qui offre les mêmes services que CryptoAPI. Un objet COM est un composant logiciel, c'est-à-dire un élément du système qui offre un service bien défini et qui peut communiquer avec d'autres composants du système. COM est utilisé pour permettre la communication entre différents processus et permet la création dynamique d'objet dans tous les langages qui supportent cette technologie. Un objet COM qui est embarqué dans une page web est appelé également un objet *ActiveX*, nom familier pour les internautes, car cette technologie a montré ses vulnérabilités en matière de sécurité et a été victime de beaucoup de virus informatiques.

CAPICOM offre plusieurs avantages par rapport à CryptoAPI. Son utilisation n'est pas limitée au C ou C++, mais accessible également à JavaScript, VBScript, C#, Visual Basic, et même Java. De plus, contrairement à un fichier DLL classique (comme les API Windows), il est possible d'extraire à partir d'une bibliothèque COM les types d'arguments qu'une fonction donnée reçoit et le type de la valeur de retour. Ceci permet, par exemple, le contrôle des types lors de la compilation. Un fichier DLL classique ne contient que le nom des fonctions et il faut se référer à la documentation ou au fichier d'entête (.h) pour connaître les types de données utilisés.

L'inconvénient de l'utilisation d'un composant COM, comme nous le verrons plus loin, est la nécessité de l'enregistrer dans le système avant son utilisation. Cet enregistrement, qui modifie la base de registre Windows, requiert normalement des droits d'administrateur.

Enfin, la plateforme .NET de Microsoft, offre également la possibilité d'accéder aux ressources cryptographiques de Windows. Dans la bibliothèque des classes .NET, l'ensemble des fonctions similaires à CryptoAPI sont regroupées dans l'espace de nom *System.Security.Cryptography*. La plateforme .NET possède aussi le grand avantage d'offrir un ensemble de services disponibles dans tout langage qui utilise la technologie .NET (C#, C++, ASP, Visual Basic, J#). A l'inverse, son défaut est qu'un code doit nécessairement être compilé vers le langage .NET intermédiaire MSIL (*Microsoft Intermediate Language*) pour profiter de ces services.

7.7.2 Interopérabilité entre Java et les ressources cryptographiques de Windows

Il a été choisi de développer l'application en Java afin qu'elle soit multiplateforme, et qu'elle puisse être embarquée sur une page web, et signée par un éditeur tiers digne de confiance. Cependant, il est nécessaire de tenir compte des spécificités de chaque plateforme pour accéder aux ressources cryptographiques (magasin de Windows, de

[17] En extension de la technologie COM de Microsoft, il y a la technologie DCOM ou Distributed COM qui permet l'utilisation de composant COM à travers un réseau (de manière similaire à CORBA ou Common Object Request Broquer Architecture)

Mozilla, de Java…). Nous allons analyser ici les possibilités d'interopérabilité entre Java et Windows pour accéder aux ressources cryptographiques.

De nombreux outils existent pour rendre interopérables plusieurs systèmes d'exploitation, ou plusieurs langages de programmation. On distingue

- interopérabilité '*out of process*' :
    - communication par socket : le programmeur doit dans ce cas définir et implémenter lui-même tout le protocole de communication.
    - interopérabilité par l'utilisation de middleware ORB : CORBA, ou COM. Ces middlewares permettent l'interaction entre des objets appartenant à des processus différents. Cette solution est plus performante que l'utilisation de services web (débit binaire moins important).
    - les services web (*webservice*s) offre une communication via des protocoles de réseau standards : SOAP avec HTTP et XML. L'avantage est la (relative) simplicité de la mise en œuvre et l'utilisation de standards qui sont largement répandus dans les différents langages de programmation. Par contre cette solution n'est pas très performante en ce qui concerne la charge de calcul et le débit de données.
- interopérabilité *'in process'*: cette solution consiste à exécuter au sein d'un même processus des 'morceaux de code' compilés à partir de langages différents. Elle offre souvent de meilleures performances, mais est plus difficile à réaliser. Par exemple JNI permet l'exécution d'une bibliothèque C ou C++ dans un programme Java.

	Logiciels libres	*Logiciels commerciaux*
Java ↔ Win API, DLL	– JNI (Java Native Interface) – Jawin – NativeCall – Brian Boyter (CryptoAPI)	– JACAPI (pour CryptoAPI) – Infomosaic XML Signature Applet
Java ↔ COM	– Jawin – COM4J – JCOM	– JNBridge – Intrinsyc J-Integra for COM
Java ↔ .NET	– Mono	– JNBridge – Intrinsyc J-Integra Expresso (implémentation de CORBA en .NET) – Intrinsyc J-Integra for .NET (implémentation en java de .NET Remoting)

Figure 22 : quelques logiciels d'interopérabilité

7.8 Tests d'interopérabilité

7.8.1 COM : Jawin et COM4J

L'utilisation de CAPICOM semble la plus facile à première vue. Nous avons testé CAPICOM avec deux logiciels libres permettant l'utilisation de composants COM dans Java : Jawin et COM4J. Dans les deux cas, ces outils créent des classes 'stubs' pour Java. Ces classes sont des 'miroirs' du composant COM distant et sont utilisées de manière locale dans Java.

Ces deux outils ont donné des résultats satisfaisants. Nous avons rencontré quelques bugs avec Jawin et Sun JDK 1.5. Une explication de ces bugs est donnée en annexe.

L'utilisation d'un composant COM est aisée, mais son enregistrement préalable est requis. Des informations doivent être ajoutées dans la base de registre Windows. Lorsqu'un client COM veut accéder à un 'serveur' COM, il interroge cette base de données en donnant le nom du composant ou son numéro d'identification unique (GUID). Le problème est que la modification de la base de registre requiert des droits administrateurs.

Un test a été réalisé sous Windows XP version familiale, avec un utilisateur 'limité'. Ce test consistait d'une part à enregistrer le composant CAPICOM avec la commande *regsvr32*, et d'autre part à modifier la base de registre avec l'outil *regedit* (éditeur de registre). La commande `regsrv32` renvoie l'erreur « *CAPICOM could not be loaded, possibly due to insufficient access privileges on this machine* ». L'éditeur de registre génère également un message d'erreur lorsque l'on tente de créer ou de modifier une entrée du registre[18]. Par exemple, la création d'une clé dans HKEY_CLASSES_ROOT, provoque l'erreur suivante : « *Impossible de créer la clé : erreur d'écriture dans le Registre.* ». Avec un compte administrateur, ces opérations sont acceptées.

Nous n'avons pas étendu plus loin nos recherches. Ce problème de privilèges rend l'installation de l'outil cryptographique plus complexe. Pour accéder aux ressources cryptographiques de Windows, l'intervention d'un administrateur du système serait requise afin d'installer CAPICOM. L'utilisation de CryptoAPI avec JNI n'a pas cet inconvénient. Les questions suivantes donnent quelques pistes de réflexion : est-il possible d'utiliser un composant COM sans l'enregistrer ? Existe-t-il différents types d'enregistrement, par exemple un enregistrement au niveau 'utilisateur', et non 'système' et/ou qui serait temporaire ?

7.8.2 CryptoAPI et Jawin

Etudions maintenant l'utilisation de CryptoAPI avec JNI. L'utilisation de JNI sans outil supplémentaire exige des connaissances en C ou C++, ce que redoutent de nombreux programmeurs en Java. C'est pourquoi Jawin et NativeCall proposent tous deux un outil permettant l'utilisation de JNI sans implémenter du code C ou C++. Ces outils possèdent déjà une bibliothèque C(++) générique, qui sert de 'pont' entre Java et n'importe quelle bibliothèque native.

Nous avons testé Jawin avec CryptoAPI. Ce test nous a rendus perplexe sur l'utilité réelle d'un tel outil (Jawin). En effet, avant de maîtriser l'utilisation de Jawin, il est nécessaire de savoir ce qu'est une structure C, un pointeur, quelle taille occupe une variable, comment fonctionne l'allocation de mémoire… Des connaissances du langage C sont malgré tout nécessaires.

Une bibliothèque DLL classique de Windows possède des points d'entrée (*entry points*) qu'il est possible de découvrir en lisant le fichier (avec l'outil *Depency Walker* de Microsoft fourni avec le kit *Platform SDK*). Malheureusement, on ne peut tirer comme information que le nom de la fonction et pas les types d'arguments ou le type de retour. Il est donc difficile (mais pas impossible[19]) de créer des fichiers stubs de manière automatique comme avec un composant COM, ou une bibliothèque .NET (*JNBridge*[20]). C'est la raison pour laquelle les outils comme Jawin ou NativeCall sont peu performants.

Jawin possède un autre inconvénient majeur : il n'offre aucune aide pour traiter des structures C. Que faire lorsqu'une fonction retourne un pointeur vers une structure, et

[18] Certaines modifications de la base de données sont admises avec un utilisateur limité. Il est par exemple possible de créer une chaîne dans la racine de HKEY_CLASSES_ROOT.
[19] il est possible de retrouver les informations sur les fonctions, les structures, les énumérations, etc. en lisant les fichiers d'en-tête (*.h*)
[20] http://www.jnbridge.com/

que l'on désire lire la valeur du troisième élément de la structure ? Nous n'avons pas trouvé la solution avec la documentation fournie dans Jawin. Pour contourner cette limitation, nous avons créé une bibliothèque DLL additionnelle implémentant des fonctions de base, comme `readInt(pointer)` qui lit un entier à l'adresse `pointer`. Une structure qui ne contient que des pointeurs ou entiers de type *int*, et qui est compilée avec un processeur *x386* ou supérieur, aura ses éléments à l'adresse $x + 0$, $x + 4$, $x + 8$, $x + 12$… L'unité d'adressage est l'octet et chaque élément (pointeur ou entier) occupe 32 bits (= 4 octets).

7.8.3 CryptoAPI et JNI : la solution

Suite aux tests d'interopérabilité infructueux, la solution la plus adéquate semble finalement être JNI. Cette solution nécessite l'implémentation en C ou C++ d'une bibliothèque jouant le rôle de « pont » entre Java et les bibliothèques dynamiques de Windows comme CryptoAPI.

L'application est divisée en plusieurs packages. Les flèches du schéma suivant indiquent les dépendances entre les packages :

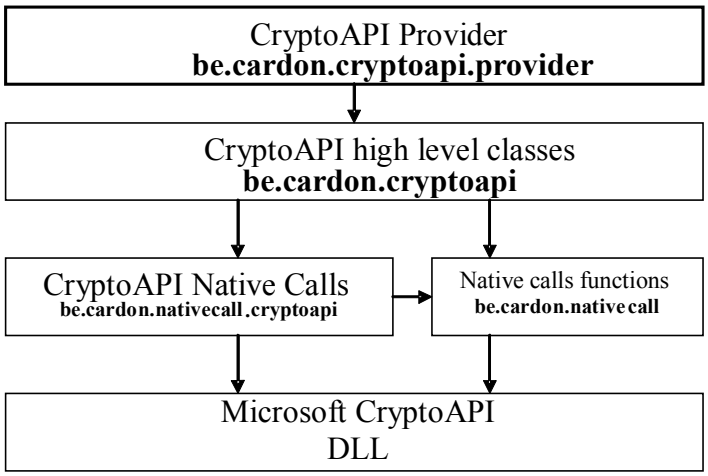
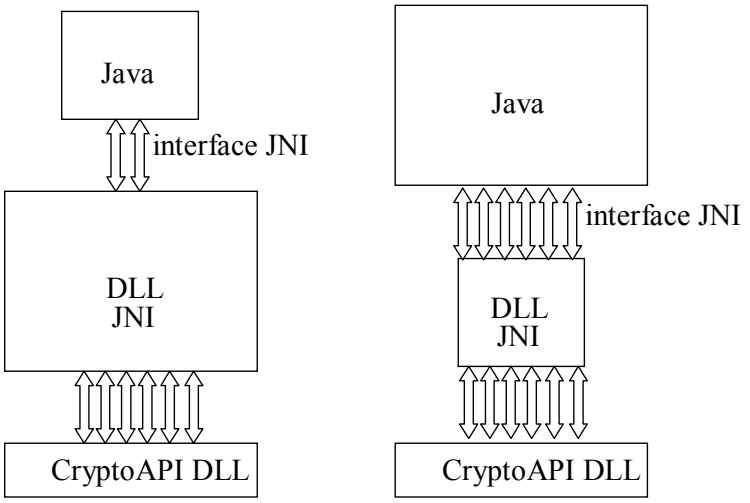


Figure 23 : architecture du provider java pour CryptoAPI

Ce provider offre actuellement deux types de service cryptographique : *KeyStore* et *Signature*. D'autres services, comme le chiffrement, pourraient être ajoutés dans le futur mais ils ne sont pas nécessaires pour la réalisation de notre application de signature.

L'interface JNI est composée de plusieurs fonctions statiques *native*s définies dans une classe Java, mais implémentées en réalité en C ou C++.



Plusieurs stratégies sont possibles pour l'implémentation d'une telle interface. Comme illustré sur le schéma ci-dessus, nous pouvons implémenter la plus grande partie soit en C++ (DLL), soit en Java. Dans le premier cas, l'interface JNI entre Java et la bibliothèque DLL JNI sera limitée à quelques fonctions bien spécifiques répondant à nos besoins. Dans le deuxième cas, l'interface JNI contiendra autant de fonctions que celles utilisées dans CryptoAPI. Cette deuxième approche a été utilisée pour les raisons suivantes :

- meilleure connaissance en Java,
- l'interface JNI plus large permet d'utiliser CryptoAPI de manière plus polyvalente sans devoir modifier la bibliothèque DLL JNI.

Par contre, du point de vue performance, cette deuxième approche pourrait s'avérer moins bonne (un programme C++ bien conçu s'exécutera généralement plus vite qu'un programme Java), mais ce problème est secondaire[21].

Deux bibliothèques DLL ont été implémentées, chacune associée à un 'package' :

- *cpp/nativecall.cpp* (**`be.cardon.nativecall`**) : fonctions pour lire et écrire des types de base (entiers, chaînes de caractère).
- *cpp/cryptoapi4java.cpp* (**`be.cardon.nativecall.cryptoapi`**) : appel de fonctions de CryptoAPI comme `CertOpenStore`, `CertGetCertificateChain`, etc. (cfr documentation MSDN)

Le package **`be.cardon.nativecall`** implémente les fonctions de base suivantes dans la classe `be.cardon.nativecall.LowLevelCalls` :

```
public native int readInt(int IntAddress) throws NativeCallException;
public native void writeInt(int IntAddress, int value) throws …;
public native short readShort(int IntAddress) throws …;
public native String readUTFString(int StringAddress) throws …;
public native String readUnicodeString(int StringAddress) throws …;
public native boolean readBoolean(int BooleanAddress) throws …;
public native void writeBoolean(int BooleanAddress, boolean bool) throws …;
public native byte[] readBytes(int firstByteAddress, int numberOfBytes);
```

[21] Notez que, contrairement à certaines implémentations, la nôtre traite toujours un document par flux : un document de plusieurs Go peut être signé sans problème de gestion de mémoire, puisqu'il sera traité bloc par bloc. Certaines implémentations se contentent d'offrir une fonction du style `sign(byte[] data)` où `data` est une variable contenant toutes les données. C'est par exemple le cas de l'implémentation d'OpenACES OpenSign.

```
public native int allocAndCopyBytes(byte[] data)throws NativeCallException;
public native int allocBytes(int numberOfBytes)throws NativeCallException;
public native void deleteAllocatedData(int firstByteAddress);
public native void writeBytes(int firstByteAddress, byte[] bytes);
```

Ce package contient des objets hérités de NativeObject qui jouent le rôle de proxy pour quelques types C :

- `NativeBoolean` : type élémentaire `bool`,
- `NativeByteArray` : type tableau `byte[]`,
- `NativeInt` : type élémentaire `int`,
- `NativeStucture` : type élémentaire `struct`,
- `NativeUTF8String` : null terminated UTF-8 string ,
- `NativeUnicodeString` : null terminated unicode string (UTF-16 little endian).

Ces objets peuvent être initialisés de deux manières :

- soit ils sont rattachés à un 'objet' natif existant, et reçoivent en paramètre l'adresse de l'objet natif,
- soit un nouvel objet natif est créé. Dans ce cas, la désallocation de la mémoire est gérée automatiquement avec le *garbage collector* de java. Lorsque l'objet java n'est plus utilisé, le *garbage collector* appelle la fonction `finalize()` de l'objet java. Cette fonction supprime l'objet natif de la mémoire.

Ce package a été conçu uniquement pour la version 32 bits de Windows. Le fonctionnement sur 64 bits exigera une adaptation des classes Java ainsi qu'une recompilation des bibliothèques natives. Le tableau suivant indique les tailles des types sur un système 32 bits et les types correspondants en Java :

Tableau 7 : « mapping » entre variables C et Java

type C	*taille sur système sur 32 bits*	*type Java*
byte	1 byte	byte
char (unicode)	2 bytes	char
short	2 bytes	short
int tout type de pointeurs	4 bytes (32 bits)	int
__int64	8 bytes	long

Le second package **`be.cardon.nativecall.cryptoapi`** contient les fonctions spécifiquement liées à CryptoAPI, ainsi qu'une dizaine d'objets hérités de `NativeStructure`, qui implémentent les structures définies dans CryptoAPI.

JNI permet une gestion appropriée des exceptions. La plupart des fonctions de l'API Windows retournent *false* en cas d'erreur. Les fonctions natives ont été implémentées de manière à générer une exception Java au lieu de retourner un boolean.

Ensuite, à un niveau plus élevé, on a le package **`be.cardon.cryptoapi`** qui dépend directement du package **`be.cardon.nativecall.cryptoapi`** et du package **`be.cardon.nativecall`.** Ce package offre une interface orientée objet qui cache tout l'aspect natif. On y retrouve les objets `CAPICertificate`, `CAPICertificateChain`, `CAPIHash`, `CAPIPrivateKey`, `CAPIStore`, etc. Ce package est orienté vers l'architecture Microsoft CryptoAPI.

Finalement, le package **`be.cardon.cryptoapi.provider`**, utilisant uniquement des objets de **`be.cardon.cryptoapi`**, est constitué d'un fournisseur de service cryptographique conformément à l'architecture Java (voir référence [11]). Ce fournisseur de service offre deux services : *KeyStore* (magasin de clefs et de certificats de confiance) et *Signature*.

Notons également l'existence de quelques outils nécessaires au bon fonctionnement de ce provider :

- la commande `System.getProperty("os.name")` retourne le nom du système d'exploitation. L'utilisation du provider exige que cette propriété contienne la chaîne « Windows ».
- `be.cardon.utils.LibraryLoader` offre la possibilité d'extraire une bibliothèque DLL vers un dossier temporaire afin de la charger en mémoire. Cet outil est nécessaire pour la création d'un applet distribué sous forme de fichier *.jar* signé contenant toutes les ressources (fichiers DLL inclus).

7.8.4 Test de signature : CryptoAPI et Sun

Le test suivant a pour objectif de générer la même signature

- avec le provider cryptographique standard de Sun, dans le langage Java,
- et avec Microsoft CryptoAPI, dans le langage C++.

Nous avons utilisé la même clef privée logicielle dans les deux cas. Le message à signer est un groupe de 3 bytes composés des lettres ASCII 'abc'.

Le code source est disponible dans

- `cpp\cryptoapitest.cpp`
- et src\be\cardon\sign\test\SignTestPKCS12.java

Le test a été fructueux : on obtient dans les deux cas une signature de 128 bytes, dont la représentation hexadécimale vaut en *big endian* (le byte le plus significatif en premier) :

```
1550105505D2760C85CA11D56DD939841F2C2995CFE659C379CE2216BF800EC1B3880FA72EF
CBACE347A772997E938B566CBB22B90AD3299B8D8383BCA97CE9186DA0BA60C218C1E5F422F
7EA5887FDA3DEF3529776B12758EF7B3EFFB3A931F0FDD463AB2315364890A11AC0DA14E7C9
0251DF56F4BCE2D32657CBFEECC8A0F
```

Notons que CryptoAPI renvoie la signature en *little endian* (le byte le plus significatif en dernier). La machine virtuelle Java fonctionne en *big endian* quelque soit la plateforme hôte. A l'inverse, les processeurs Intel *x386* et supérieur fonctionnent en *little endian*. Dans ce dernier cas, on a :

```
0F8ACCEEBF7C65322DCE4B6FF51D25907C4EA10DAC110A89645331B23A46DD0F1F933AFBEFB
3F78E75126B772935EF3DDA7F88A57E2F425F1E8C210CA60BDA8691CE97CA3B38D8B89932AD
902BB2CB66B538E99729777A34CEBAFC2EA70F88B3C10E80BF1622CE79C359E6CF95292C1F8
439D96DD511CA850C76D2055510501550
```

7.9 Communication entre l'applet et le serveur

Lors du téléchargement des documents à signer, l'applet doit assurer l'authentification du serveur. Idéalement, l'applet pourrait déléguer cette tâche au navigateur, ou utiliser la connexion sécurisée déjà établie entre le navigateur et le serveur. Ceci ne semble cependant pas prévu dans Java Sun.

L'applet doit ouvrir sa propre connexion avec le serveur. Une solution simple est de passer le certificat du serveur comme paramètre à l'applet Java. Lors de l'ouverture de la connexion SSL[22] entre l'applet et le serveur, l'applet vérifie que le certificat du serveur correspond avec le certificat passé comme paramètre. Ceci garantit que la connexion sécurisée avec l'applet est effectuée avec le même serveur.

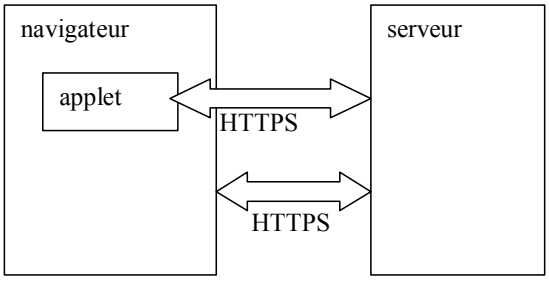


Figure 24 : Connexion entre l'applet et le serveur

La fonction `be.cardon.sign.communicator.HTTPSUtils.getSecureURL(URL address, Certificate serverCert)` est chargée de cela. Elle reçoit l'adresse du fichier à télécharger ainsi que le certificat du serveur qui doit correspondre au certificat reçu lors de la négociation à l'ouverture de la connexion SSL. Cette fonction crée un nouveau `TrustManager` qui cherche les informations dans un magasin de certificats `SimpleKeyStore`. Ce dernier ne contient qu'un seul certificat : celui qui a été passé comme paramètre à l'applet.

```
Provider ksProvider = new SimpleKeyStoreProvider();
KeyStore ks = KeyStore.getInstance("SimpleKeyStore", ksProvider);
…
ks.setCertificateEntry("serverCert", serverCert);
TrustManagerFactory tmf = TrustManagerFactory.getInstance(
        TrustManagerFactory.getDefaultAlgorithm());
tmf.init(ks);
```

Ensuite, un « contexte SSL » est initialisé à partir du `TrustManager` créé précédemment.

```
SSLContext sslContext = SSLContext.getInstance("TLS");
sslContext.init(null, tmf.getTrustManagers(), null);
```

Enfin ce contexte SSL est défini comme contexte par défaut pour les prochaines connexions SSL :

```
HttpsURLConnection.setDefaultSSLSocketFactory(
                          sslContext.getSocketFactory());
```

On distingue deux types de communication entre l'applet et le serveur :

– téléchargement d'un fichier à signer : la requête du fichier est réalisée par méthode GET avec trois paramètres : `sessionID`, `sessionKey`, `file`. `SessionID` est le numéro de « session de signature » qui est unique pour chaque requête de signature. Un utilisateur aura donc un `sessionID` différent pour deux requêtes de signature distinctes. `sessionKey` permet d'authentifier la session en cours. Enfin `file` est le nom du fichier à télécharger.

[22]Par abus de langage, on utilise l'abréviation SSL aussi bien pour exprimer le standard SSLv1 à v3, mais aussi le nouveau standard TLSv1 qui succède à SSLv3.

– envoi de la signature vers le serveur : l'envoi de la signature est effectué par méthode POST avec trois paramètres : `sessionID`, `sessionKey` et `signature`. Le dernier paramètre contient le contenu du fichier XML encodé en base64.

7.10 Choix du format de la signature

Les standards existants offrent le choix sur le format de signature : CMS (*Cryptographic Message Syntax*), S/MIME, XML Signature, et XadES. Ces différents formats ont été discutés dans la section 5.1.

Il existe des outils de développement Java pour le traitement de signature CMS, S/MIME et XML :

– *Bouncy Castle*[23] : implémentation de nombreux algorithmes cryptographiques, création et traitement de données CMS et S/MIME.

– *Apache XML Security*[24] : création et traitement de données XML Signature (non basé sur la nouvelle API JSR 105[25]).

– *Sun WebServices XML Signature*[26]: cette bibliothèque utilise *Apache XML Security* et utilise la nouvelle API JSR 105.

Concernant XadES, les outils de développement libres accessibles actuellement sont peu nombreux. L'ETSI a publié en novembre 2003 un rapport de tests d'interopérabilité de quelques implémentations existantes[27] . Ce rapport énumère quelques implémentations :

– *Baltimore* pour Java : implémentation commerciale.

– *IAIK XML Advanced Electronic Signatures (XAdES) add-on for XML Security Toolkit (XSECT)*[28] : implémentation commerciale.

– OpenXadES[29] : logiciel libre, création de signature au format « Digidoc-XML » (un sous-ensemble non standardisé de XadES). N'implémente pas toutes les fonctionnalités XadES. Utilise Apache XML Security. Voir également le site dédié à la carte d'identité estonienne basé sur OpenXadES[30].

– *Digital Contract Project*[31] : implémentation commerciale.

Nous avons choisi d'utiliser *Sun WebServices XML Signature* car le format XML Signature est un format plus facilement 'lisible' que CMS et S/MIME et car cette bibliothèque utilise la nouvelle API standardisée.

Voici une remarque importante concernant la taille du logiciel : La signature XML exige l'utilisation d'un processeur XML et divers outils XML nécessaires à la mise en forme canonique. *Apache XML Security* demande notamment l'utilisation des bibliothèques :

– Apache Xerces (parser XML) : la bibliothèque XercesImpl.jar prend 1,14 MB

---

[23] http://www.bouncycastle.org/
[24] http://xml.apache.org/security/
[25] http://jcp.org/en/jsr/detail?id=105 : JCP (Java Community Process) est une organisation ouverte qui révise les spécifications de la technologie Java. Le JSR 105 est une spécification Java intitulé 'XML Digital Signature APIs'. Cette spécification décrit l'interface de programmation que doivent respecter les outils de développement de signature XML. Cette spécification constitue une extension à l'architecture cryptographique de Java.
[26] http://java.sun.com/webservices/xmldsig/index.jsp)
[27] http://www.etsi.org/Plugtests/History/DOC/XAdESFinalReport.pdf
[28] http://jce.iaik.tugraz.at/sic/products/xml_security
[29] http://www.openxades.org/
[30] https://digidoccheck.sk.ee/index.php?f=list
[31] http://www.frankcornelis.be/dcontract/

– Apache Xalan (processeur XSLT) : la bibliothèque Xalan.jar prend 2,93 MB

Heureusement, ces bibliothèques sont maintenant intégrées par défaut dans la plateforme *Java Sun* 1.5. Ceci permet de réduire considérablement la taille des 'petites' applications utilisant la technologie XML.

7.10.1 Spécifications de la signature XML

Le standard XML Signature est large et offre beaucoup de liberté sur son utilisation. Par exemple, une signature 'bien formée' ne doit pas nécessairement contenir le certificat du signataire. Dans ce cas, le certificat est transmis d'une manière externe.

Notre application comprend un outil de vérification de signature. Cet outil ne sait pas vérifier toutes les signatures XML. Ceci rendrait l'outil complexe et difficile à utiliser et n'aurait que peu d'intérêt. Nous avons donc défini une spécification qui limite l'ensemble des signatures vérifiables à un sous-ensemble de XML Signature. Cette spécification exige par exemple la présence d'au moins un certificat, et impose que les fichiers référencés soient locaux.

– La signature XML doit contenir au moins un certificat : celui dont la clef publique est associée à la clef privée.

– Les références ne peuvent pas contenir de transformations autres que les transformations standardisées servant à rendre une structure XML sous forme canonique. En effet, XML Signature permet de définir n'importe quelle transformation. Un attaquant pourrait définir une transformation qui donne toujours le même résultat…La signature serait donc toujours valide avec n'importe quelles données ! Lors de la vérification, il faut tenir compte que les données réellement signées sont les données après transformation.

– La signature XML peut contenir la propriété de signature (*signatureProperty*) suivante (exemple) comme référence signée :

```
<SignatureProperty Id="TimeStamp" Target="#Signature-1">
<FriendlyDate>lun., 18 sept. 2006 11:30:07 +0200</FriendlyDate>
<UTCMillisecsFromEpoc>1158571807562</UTCMillisecsFromEpoc>
  </SignatureProperty>
```

Ce *timestamp* protège le signataire contre une attaque 'replay' mais ne prouve nullement l'exactitude de la date de signature. Avec un tel procédé deux signatures d'un même document seront toujours différentes. Ceci est une pratique couramment recommandée dans la signature électronique. *FriendlyDate* contient la date avec un format convivial. Ce format est libre. *UTCMillisecFromEpoch* doit contenir un entier avec le nombre de millisecondes écoulées depuis le 1e janvier 1970 00:00:00.000 GMT[32]

– La signature XML ne peut pas contenir des références signées autres que des références vers des fichiers et la *signatureProperty* définie ci-dessus.

– Les références ne peuvent pas être des références absolues. Les références doivent être des références relatives par rapport à l'emplacement du fichier signature. Ceci implique que la signature XML ne peut pas contenir des références externes au système, demandant une connexion réseau.

– Le dossier contenant directement la signature doit être parent à tous les fichiers signés. Une référence ne peut donc pas commencer par `../`

[32] voir `java.util.Calendar.getTimeInMillis()`

7.11 Types de fichiers signés

7.11.1 HTML

HTML est sans doute le format le plus facile à générer à partir d'un serveur web dynamique comme PHP ou ASP. Cependant, HTML souffre de plusieurs problèmes.

Tout d'abord, si le code HTML est mal formé, différents navigateurs n'afficheront pas le même résultat. Certains seront plus souples et afficheront la totalité de l'information tandis que d'autres afficheront une erreur ou cacheront silencieusement certaines informations. Le langage XHTML basé sur XML offre une meilleure solution.

Ensuite, l'HTML permet d'embarquer des objets dynamiques : des images animées GIF, du code JavaScript, VBScript, du code DHTML…etc. Ceci ne convient pas pour la signature d'un document. Il faut interdire la signature de fichier HTML contenant de tels objets.

D'autre part, un document HTML peut inclure des liens hypertextes et des images qui ne sont pas intégrés dans la signature. Ainsi, la modification d'une image 'externe' à la signature ne modifierait pas la validité de la signature. Il faut également empêcher cela.

Idéalement, il faudrait analyser le fichier HTML avant la signature et contrôler qu'il ne possède pas d'objets illégaux, ce qui rend difficile son utilisation.

7.11.2 PDF

Le format PDF ne possède pas les défauts rencontrés avec HTML. Cependant la génération dynamique de fichier PDF requiert des outils spécialisés. Pour créer des documents PDF sous PHP, citons quelques outils :

- *PDFlib* : bibliothèque PDF utilisable avec de nombreux langages de programmation : Java, C, C++, Python, .NET, PHP, etc. Malheureusement cette bibliothèque très complète n'est pas libre, et la version gratuite affiche un logo gênant sur chaque page.
- *R&OS* : bibliothèque PDF libre entièrement écrite en PHP5, n'ayant besoin d'aucun module supplémentaire.
- *dompdf* : convertisseur HTML vers PDF pour PHP5 utilisant soit PDFlib, soit R&OS.

Nous avons utilisé *dompdf* avec *R&OS* pour générer des documents PDF à partir de documents HTML. Cet ensemble est libre de droit et facile à utiliser car il ne requiert que la connaissance du langage HTML.

7.12 Visualisation des fichiers à signer

Le signataire doit pouvoir visionner le document avant de le signer (WYSIWYS). Par mesure de sécurité, il est préférable que l'applet embarque lui-même un visionneur de document. Ceci permet d'éviter l'utilisation du visionneur installé 'par défaut' sur la machine. Ce dernier pourrait être infecté par un virus qui provoque l'affichage d'un autre document.

Les visionneurs PDF ou HTML libres pour Java sont peu nombreux. Nous avons testé deux outils : Swing JEditorPane et JPedal.

JEditorPane permet de visionner des documents HTML. Il a l'avantage d'être inclus dans la bibliothèque Swing. Celle-ci étant distribuée avec la plateforme Java, son utilisation n'alourdit pas la taille de l'applet. Par contre son implémentation actuelle est limitée à HTML 3.2 et ne supporte pas l'HTML 4.0. De plus, il faudrait idéalement contrôler que le fichier HTML ne contienne pas d'images externes ou d'objets animés

comme discuté précédemment. JEditorPane ne permet pas un tel contrôle. Celui-ci doit être réalisé d'une autre façon.

JPedal est un outil Java permettant l'affichage de document PDF. Il existe une version libre, mais limitée, de ce produit. Nous avons testé la possibilité d'intégrer cet outil dans l'applet. Un des problèmes rencontrés est le fait que la fermeture de la fenêtre JPedal provoque l'arrêt de l'application (commande *exit*). Il a donc fallu modifier toutes les commandes *exit* de JPedal. D'autre part l'utilisation de JPedal requiert le téléchargement jusqu'à 5 MB supplémentaires ! Enfin cet outil ne possède pas les performances ni la facilité d'utilisation d'Adobe Acrobat.

Bref, l'utilisation de visionneurs embarqués apporte des problèmes techniques difficiles à surmonter.

7.13 Choix de la bibliothèque graphique

Il existe différentes bibliothèques Java pour créer des interfaces graphiques : AWT (Abstract Windowing Toolkit), Swing, SWT (Standard Widget Toolkit), …

AWT est une ancienne bibliothèque qui utilise les ressources graphiques propres au système d'exploitation. Le nombre d'éléments disponibles est assez limité. AWT ne supporte que les éléments communs à toutes les plateformes (Windows, Linux, …)

Swing est une extension d'AWT. Il offre de nouveaux composants qui ne sont pas générés à partir du système d'exploitation. Par contre, les *layouts* (gestionnaires de mise en page) de Swing sont assez difficiles à utiliser.

SWT est une bibliothèque conçue par IBM qui se veut plus rapide et plus facile à utiliser que Swing. Cette bibliothèque est utilisée avec l'environnement de programmation *Eclipse*. Elle utilise davantage les ressources du système d'exploitation, améliorant ainsi les performances du logiciel. Par contre, contrairement à AWT et Swing, cette bibliothèque n'est pas incluse avec le JRE (Java Runtime Environnement) de Sun et ne convient pas aux applets qui exigent une petite taille de téléchargement.

Nous avons choisi d'utiliser Swing. Ceci a exigé un certain effort pour soigner la mise en forme. Quelques fenêtres de l'applet sont visibles en annexe.

7.14 Exemples de bugs rencontrés

7.14.1 Caractères Unicode

Lors de la signature d'un fichier nommé « état des lieux.doc », le `dereferencer` de Apache XMLSignature renvoie une erreur du type *invalid character* '%C3%A9'.

Les URI ne peuvent pas contenir des espaces. Par exemple l'URI

```
file:/c:/mes documents/
```

est incorrect. Les caractères interdits doivent être échappés. L'échappement du caractère espace est '%20' où 20 est le numéro ASCII du caractère d'espacement. Ainsi, l'URI correcte devient :

```
file:/c:/mes%20documents/
```

Les caractères Unicode peuvent (mais ne doivent pas) être échappés. Par exemple 'état des lieux.doc' est échappé '`%C3%A9tat%20des%20lieux.doc`'. Apache XMLSignature n'effectue pas correctement l'opération inverse. Il ne décode que les caractères d'espacement %20. Ainsi, après reconversion, on obtient '`%C3%A9tat des lieux.doc`' au lieu de '`état des lieux.doc`'. Pour corriger ce problème, il a fallu créer un

nouveau `dereferencer` qui utilise `java.net.URI` pour décoder correctement tous les échappements.

Pour plus d'information, voir référence [8] (RFC 3986), documentation de `java.net.URI`, et la nouvelle implémentation dans :

`be.cardon.sign.xmldsig.dereferencer.implementations.ResolverLocalFilesystem`

7.14.2 Adresses absolues et relatives

La fonction `java.util.URI.relativize(URI uri)` permet de déterminer une adresse relative. Par exemple, soient deux adresses absolues :

- l'adresse de base : http://www.siteweb.com/directory/
- l'adresse à relativiser : http:// www.siteweb.com/directory/search/index.html

Le résultat est l'adresse relative : search/index.html

L'implémentation de Sun fonctionne uniquement si le fichier à relativiser est 'enfant' de l'adresse de base. Si on a maintenant :

- l'adresse de base : http://www.siteweb.com/directory/
- l'adresse à relativiser : http:// www.siteweb.com/otherdirectory/index.html

Le résultat devrait être : ../otherdirectory/index.html

Mais la fonction ne retourne pas ce résultat (elle retourne la même adresse inchangée). Ceci n'est pas un bug mais une 'limitation' de l'implémentation de Sun, expliquée dans la documentation.

Comme nous avions besoin d'enregistrer les adresses relatives dans les signatures, il a été nécessaire d'implémenter un outil décodant correctement les répertoires. Ces outils sont :

- be.cardon.utils.Path
- be.cardon.utils.URIUtilities

Conclusion

L'applet que nous avons créé fonctionne et permet la signature d'un document au format XML à partir du magasin de certificat Windows, ou d'un magasin Java (fichier JKS). Ce projet a démontré qu'il est possible d'embarquer, sur une page web, un outil facile d'utilisation, ne demandant pas d'installation, multiplateforme tout en jouissant des ressources spécifiques à Windows, et distribuable de façon sûre avec la signature de l'éditeur. Il n'inclut cependant pas encore de 'visionneur' interne pour afficher le contenu d'un document à signer.

L'utilisation de la signature électronique s'amplifiera certainement à l'avenir, notamment avec l'utilisation de la carte d'identité belge.

Cependant les outils libres disponibles actuellement pour développer une application sont encore 'jeunes'. Par exemple l'implémentation de XML Signature pour Java a montré ses faiblesses et il est difficile de trouver des implémentations de XadES, l'extension de XML Signature en réponse à la législation européenne en matière de signature électronique 'légale'. De plus, le middleware de la carte d'identité belge possède encore de sérieux bugs.

La signature électronique est difficilement falsifiable, comparée à une signature classique. Cependant, elle présente de nouvelles difficultés : sa mise en œuvre exige les services d'une autorité de certification. L'organisation et le contrôle de celle-ci est importante pour la sécurité de l'infrastructure.

Les entreprises désireront également stocker en lieu sûr les documents signés, comme preuves légales. Les documents électroniques permettent de diminuer le 'volume' de papier mais exige une technologie de stockage sûre (redondance des données stockées, support de stockage fiable). Ce stockage a bien entendu un coût.

Tous ces facteurs contribuent au ralentissement du développement de la signature électronique.

Le développement de notre application a apporté un nouvel outil : l'intégration des ressources cryptographiques de Microsoft Windows dans la plateforme Java. Nous espérons que cet outil évoluera et apportera une aide aux développeurs désirant créer des applications polyvalentes.

Bibliographie

[1] Douglas Stinson, « Cryptographie, Théorie et pratique, 2e édition » (traduction de Serge Vaudenay, Gildas Avoine et Pascal Junod), Vuibert, 2002

[2] Professeur Patrick Verlinde, « Transmission de signaux numériques de télécommunications », Ecole Royale Militaire, 2006

[3] Williams Stallings, « Cryptography and Network Security – Principles and practices », Prentice Hall 2003 (Third Edition)

[4] Messaoud Benantar, « Introduction to the Public Key Infrastructure » , Prentice-Hall, 2002

[5] William Stallings, « Cryptography and Network Security, Principles and pratices 3d edition », Prentice-Hall, 2003

[6] Russell Housley, Warwick Ford, Tim Polk, David Solo, « Internet X.509 Public Key Infrastructure – Certificate and Certificate Revocation List (CRL) Profile », IETF RFC 3280, 2002

[7] Santosh Chokhani, Warwick Ford, Randy V. Sabett, Charles R. Merrill, Stephen S. Wu, « Internet X.509 Public Key Infrastructure – Certificate Policy and Certification Practices Framework », IETF RFC 3647, 2003

[8] Tim Berners-Lee, Roy T. Fielding, Larry Masinter, « *Uniform Resource Identifiers (URI): Generic Syntax* », IETF RFC 3986, 2005

[9] Bjarne Stroustrup, « Le langage C++, édition spéciale », Pearson Education, 2003

[10] Laura Lemay et Rogers Cadenhead, « Java 2 Student Edition », CampusPress, 2004

[11] Sun Microsystems, Inc, « Java™ Cryptography Architecture, API Specification & Reference »[33], 4 Augustus 2002

[12] Russell Housley, « Cryptographic Message Syntax (CMS) », IETF RFC 3852, 2004

[13] Donald Eastlake, Joseph Reagle, David Solo, « XML-Signature Syntax and Processing », W3C Recommandation 12 February 2002

[14] Jon Callas, Lutz Donnerhacke, Hal Finney, Rodney Thayer, « OpenPGP Message Format », IETF RFC 2440, 1998

[15] Patrick Andries, « Middleware Architecture Document » (Belgian eID), Zetes, 2003 (version 1.0)

[16] « Component Object Model », Wikipedia[34]

[17] Brian Boyter, « Integrate Java Cryptography with Windows », JavaPro[35], 2002

[18] « U.S. Army Deploys Signature Java Applet For Financial Disclosure Management », EWORLDWIRE (www.eworldwire.com), 31 mars 2005[36]

[19] Jean-Luc Parouty, Roland Dirlewanger, Dominique Vaufreydaz, « La signature électronique, contexte, applications et mise en oeuvre. », INRIA, 2003

[33] http://java.sun.com/j2se/1.4.2/docs/guide/security/CryptoSpec.html
How to Implement a Provider for the Java™ Cryptography Architecture :
http://java.sun.com/j2se/1.4.2/docs/guide/security/HowToImplAProvider.html
[34] http://en.wikipedia.org/wiki/Component_object_model
[35] http://www.fawcette.com/javapro/2002_07/magazine/features/bboyter/
[36] http://newsroom.eworldwire.com/view_release.php?id=11776

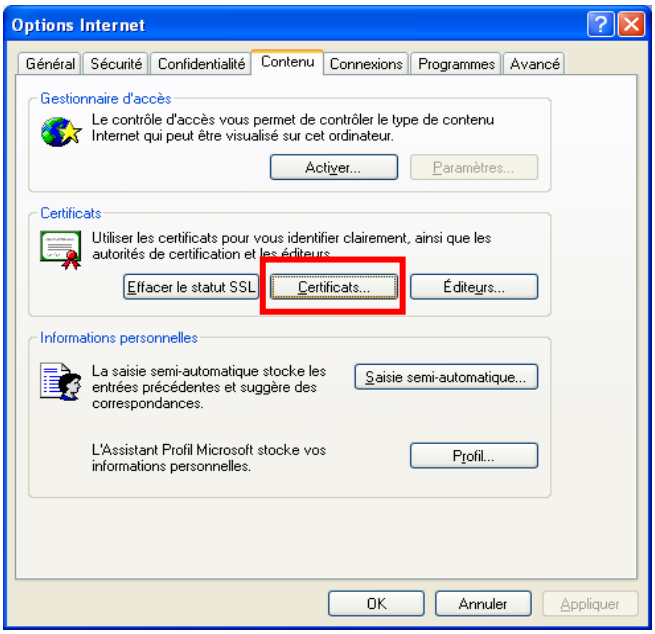
Annexe A : Accès convivial aux magasins de certificats

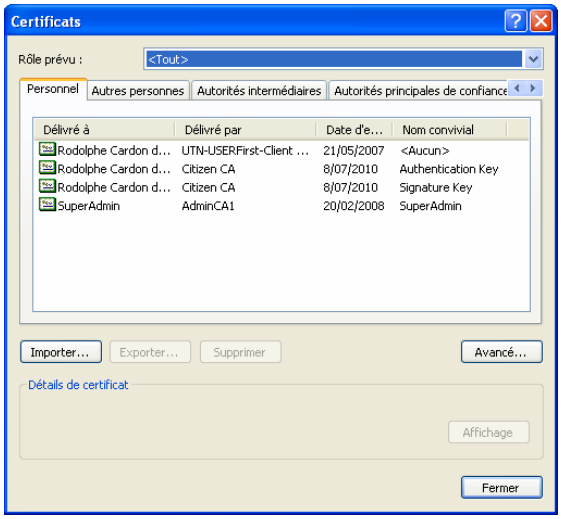
Magasin de certificats Windows

Alors que les interfaces comme CryptoAPI et CAPICOM permettent d'accéder de façon programmatique au magasin de certificats de Windows, cette annexe explique comment y accéder de manière graphique.

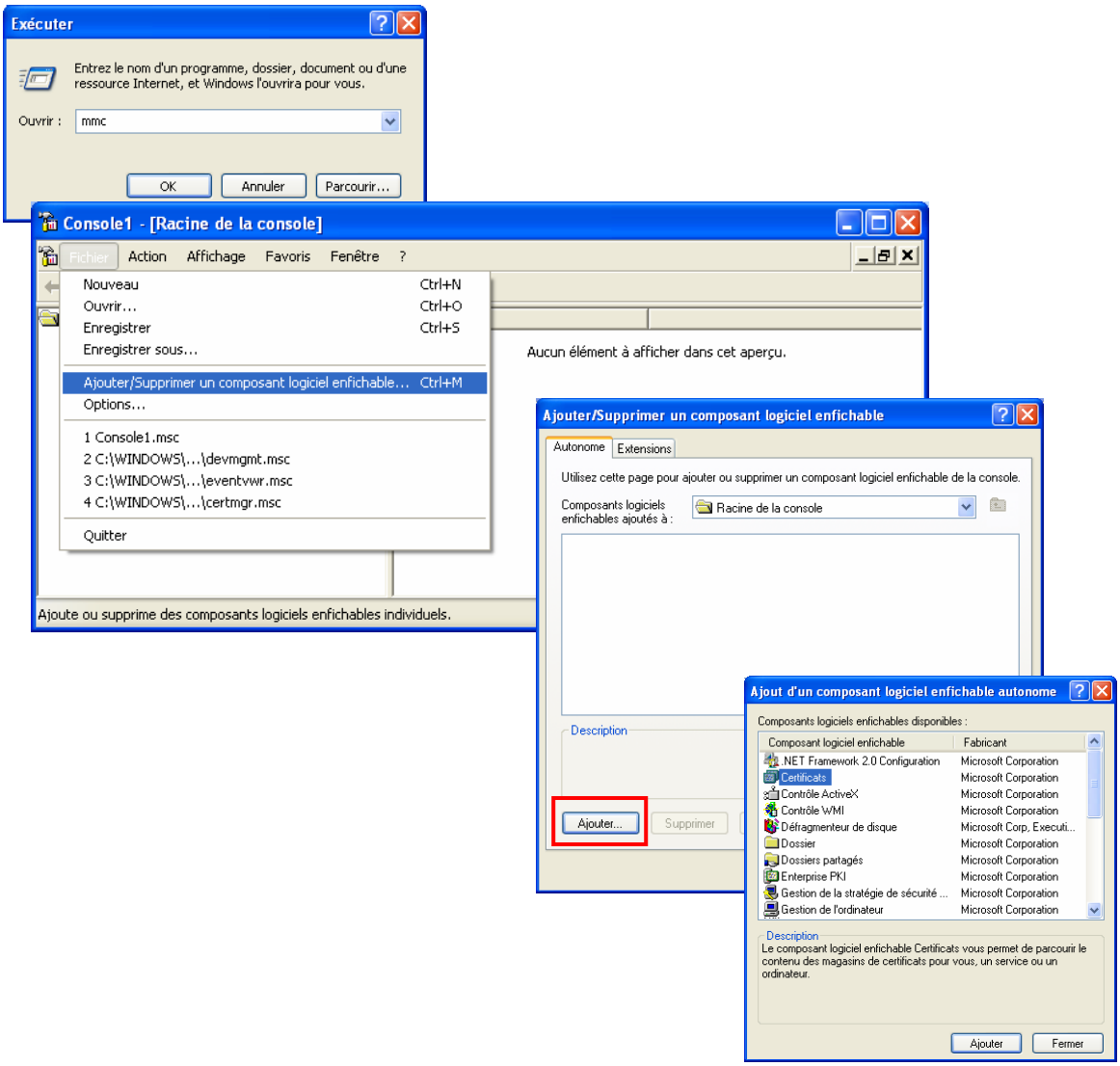
Il est possible d'afficher les magasins de certificats de deux manières :

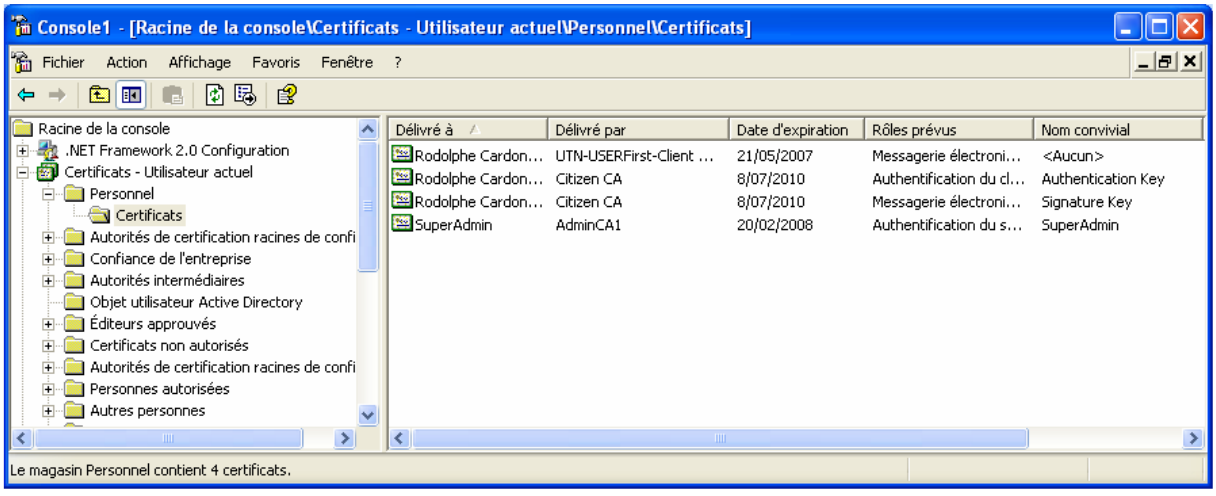
– Via Internet Explorer dans le menu *'Outils'* → *'Options Internet…'* → onglet *'Contenu'*→ *'Certificat…'*:





– Via Microsoft Management Console (la « console de gestion de Microsoft ») avec la ligne de commande *mmc* (Microsoft Management Console) dans Windows 2000, XP Familial ou Professionnel, ou 2003 : menu Fichier ➔ Ajouter un composant logiciel enfichable ➔ bouton « Ajouter… » ➔ Certificats.

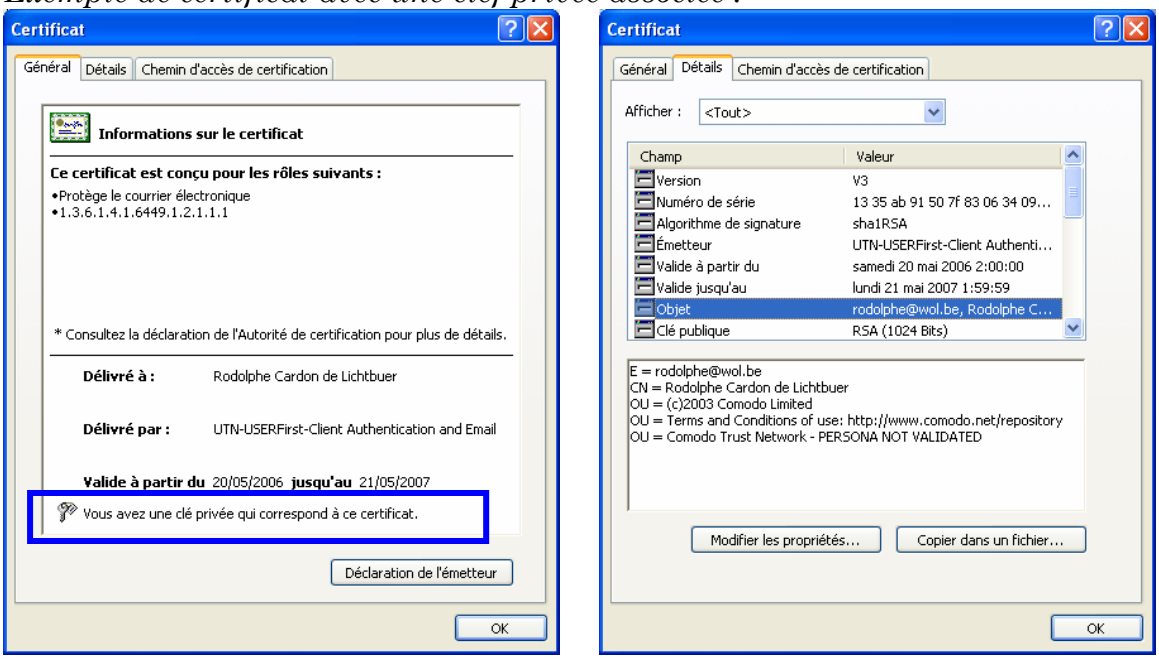




Windows possèdent plusieurs magasins de certificats par défaut qui ont un rôle bien défini. Voici une liste non exhaustive :

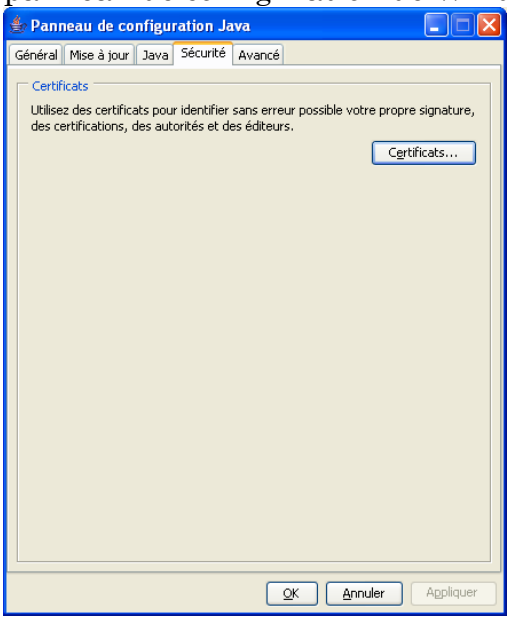
- Certificats personnels (nom du magasin : 'my') : certificats qui appartiennent à l'utilisateur et qui ont souvent une clef privée associée.
- Certificats des autorités intermédiaires (nom du magasin : 'CA'): certificat des autorités qui ne sont pas des autorités racine.
- Certificats des autorités principales de confiance (nom du magasin : 'root'): certificat des autorités racine (*root* CA). Ces certificats sont signés avec leur propre clef.
- Certificats des éditeurs approuvés (nom du magasin : 'trust') : certificat des éditeurs de logiciel dont on accepte explicitement l'exécution future d'application ou applet.

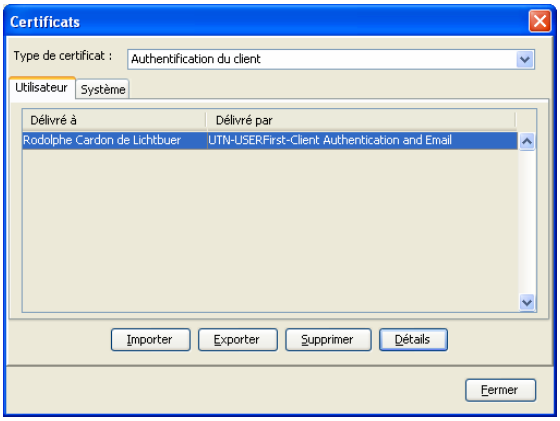
Exemple de certificat avec une clef privée associée :

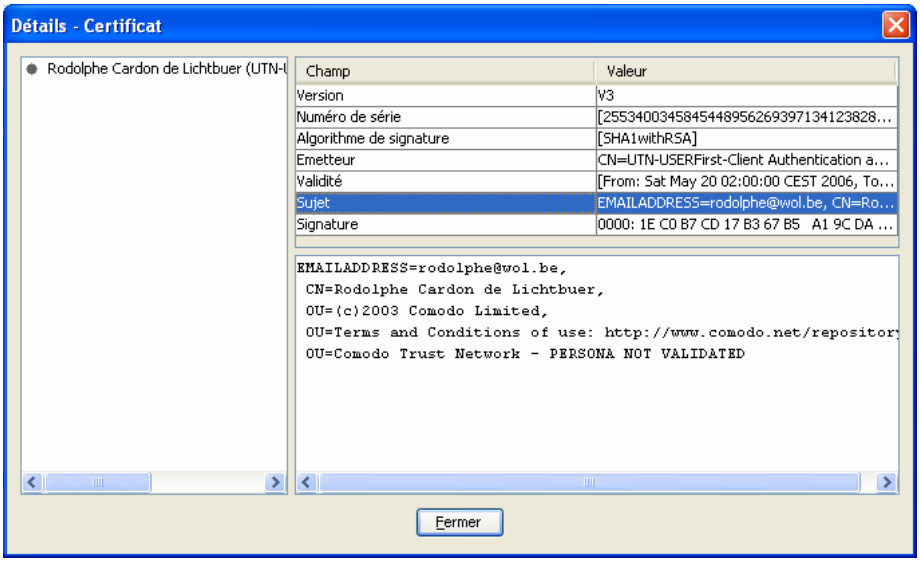


Magasins de certificats Java Sun et Mozilla

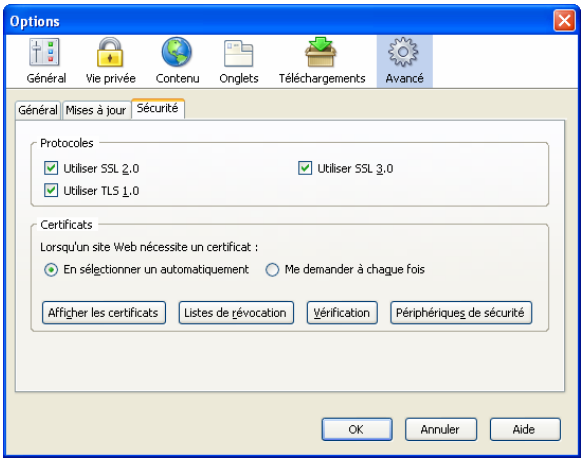
Dans Windows, on accède au magasin de *Java Sun* via l'icône *Java* dans le panneau de configuration de Windows, puis en cliquant sur l'onglet *Sécurité* :

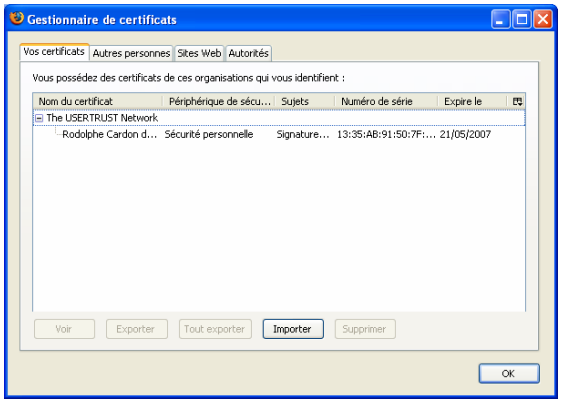


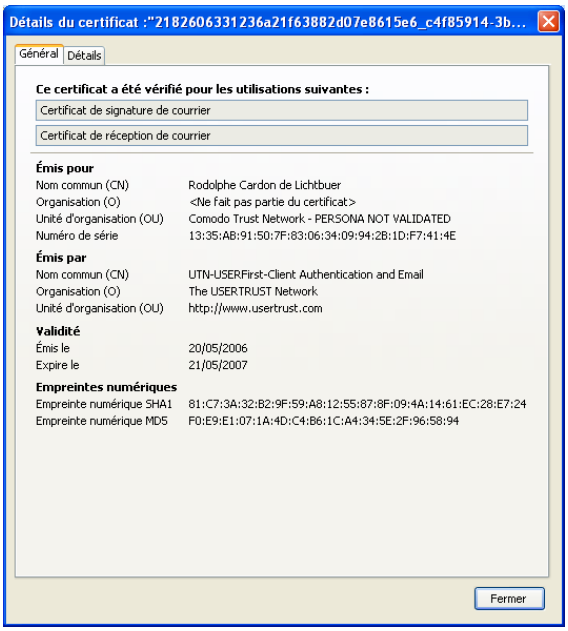


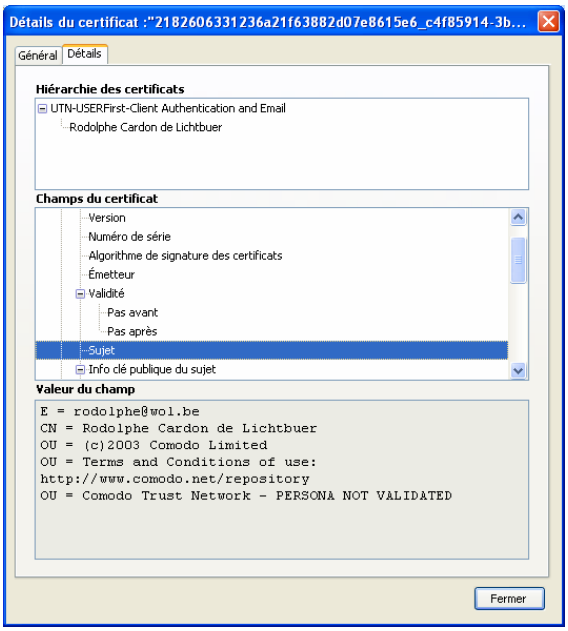


Pour Mozilla Firefox, on accède aux certificats dans le menu *« Outils »*→ *« Options… »* → bouton *« Avancé »* :









Annexe B : Bugs rencontrés avec Jawin

Les paragraphes suivants détaillent les problèmes rencontrés lors des tests d'interopérabilité avec Jawin, en vue d'accéder aux ressources cryptographiques de Windows à partir de Java.

– Protocole requis pour un chemin relatif

Le bug suivant est un bug mineur qui a été résolu assez rapidement.

Pour générer les fichiers stubs en Java, *Jawin Type Browser* utilise des transformations XSL. Les fichiers de configuration de Jawin contiennent les adresses des fichiers XML qui définissent ces transformations.

La machine virtuelle Java exige de donner, en plus du nom de fichier, le nom du protocole, c'est-à-dire 'file:/'. Sans doute que ce bug ne se présentait pas dans les versions précédentes de la machine virtuelle. Cette erreur semble étrange car il est fait usage du nom de fichier sans le chemin complet. Il s'agit d'un chemin *relatif* par rapport à un chemin de référence. Ce chemin de référence contient lui-même le protocole. Pourquoi la machine virtuelle exige donc telle qu'on spécifie à nouveau ce protocole ? Il semble donc qu'il ne s'agit pas d'un bug lié à Jawin, mais d'un bug de la machine virtuelle Java (JDK 1.5).

– Type de données COM n°30 inconnu

Ce bug a été partiellement corrigé. Dans les composants COM de Microsoft, les types de variables ont un numéro d'identification. Le numéro 30 représente un certain type de chaînes de caractère, utilisé notamment dans CAPICOM. Jawin Type Library ne connaît pas ce type 30.

```
[DEBUG] TypeParser.buildTypeDescription() failed to find a configured type for
property: CAPICOM_VERSION_INFO with type code: 30
[DEBUG] TypeParser.buildTypeDescription() failed to find a configured type for
property: CAPICOM_COPY_RIGHT with type code: 30
[DEBUG] TypeParser.buildTypeDescription() failed to find a configured type for
property: CAPICOM_MY_STORE with type code: 30
…
```

Afin de déterminer exactement à quel type correspond le numéro 30, *OLE/COM Viewer*[37] a été utilisé. L'ouverture de CAPICOM dans *OLE/COM Viewer* indique que le type 30 est un *LPSTR*.

```
module Constants {
        const long CAPICOM_MAJOR_VERSION = 2;
        const long CAPICOM_MINOR_VERSION = 1;
        const long CAPICOM_RELEASE_NUMBER = 0;
        const long CAPICOM_BUILD_NUMBER = 1;
        const LPSTR CAPICOM_VERSION_INFO = "CAPICOM v2.1";
        const LPSTR CAPICOM_COPY_RIGHT = "Copyright MS … ";
        const LPSTR CAPICOM_MY_STORE = "My";
        const LPSTR CAPICOM_CA_STORE = "Ca";
        const LPSTR CAPICOM_ROOT_STORE = "Root";
        ...
}
```

La documentation MSDN indique que LPSTR est un pointeur vers une chaîne de caractère 8-bit Windows (ANSI) se terminant par un caractère nul.

[37] Outil Microsoft fourni avec le kit Platform SDK (gratuitement) et d'autres produits de développement

– Variable de type *void* inexistant en Java

En java et en C, une fonction qui ne retourne rien est une fonction de type void. Par contre, C définit aussi une variable de type void comme une variable de n'importe quel type. Cette notion n'existe pas en Java. Jawin Type Browser commet une erreur en générant ceci (transformation d'une structure C) :

```
public class _CRYPT_KEY_PROV_INFO
{
      public static final String pwszContainerName;
      public static final String pwszProvName;
      public static final int dwProvType;
      public static final int dwFlags;
      public static final int cProvParam;
      public static final void rgProvParam;
      public static final int dwKeySpec;
}
```

En Java, la classe *Object* (java.lang.Object) désigne un objet quelconque (mais pas un élément primitif comme *int*), et doit être utilisée dans ce cas.

Annexe C : Un exemple S/MIME

L'exemple suivant est un courrier électronique signé au format S/MIME. La signature, qui est du type CMS (PKCS#7), est encodée en BER, puis convertie en base 64. Nous avons décodée celle-ci à l'aide d'OpenSSL.

Les données de cet exemple sont accessibles dans le dossier «`\exemples smime`».

– Document S/MIME

```
Message-ID: <4515139A.90104@free.fr>
Date: Sat, 23 Sep 2006 12:59:38 +0200
From: Rodolphe CARDON <rcardon@free.fr>
User-Agent: Thunderbird 1.5.0.7 (Windows/20060909)
MIME-Version: 1.0
To:  monsieurx@rma.ac.be
Subject: Contrat =?ISO-8859-1?Q?n=B0124?=
Content-Type: multipart/signed; protocol="application/x-pkcs7-signature";
micalg=sha1; boundary="------------ms040407070104030001050006"

This is a cryptographically signed message in MIME format.

--------------ms040407070104030001050006
Content-Type: text/plain; charset=ISO-8859-1; format=flowed
Content-Transfer-Encoding: 7bit

Monsieur X,

contenu du contrat....

--------------ms040407070104030001050006
Content-Type: application/x-pkcs7-signature; name="smime.p7s"
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename="smime.p7s"
Content-Description: S/MIME Cryptographic Signature

MIAGCSqGSIb3DQEHAqCAMIACAQExCzAJBgUrDgMCGgUAMIAGCSqGSIb3DQEHAQAAoIIFqzCC
AmQwggHNoAMCAQICEBg05P46uuRnNscYYh+/pM0wDQYJKoZIhvcNAQEFBQAwYjELMAkGA1UE
BhMCWkExJTAjBgNVBAoTHFRoYXd0ZSBDb25zdWx0aW5nICChQdHkpIEx0ZC4xLDAqBgNVBAMT
I1RoYXd0ZSBQZXJzb25hbCBGcmVlbWFpbCBJc3N1aW5nIENBMB4XDTA2MDgwMTA5MTE0MVoX
DTA3MDgwMTA5MTE0MVowQTEfMB0GA1UEAxMWVGhhd3RlIEZyZWVtYWlsIE1lbWJlcjEeMBwG
CSqGSIb3DQEJARYPcm9kb2xwaGVAd29sLmJlMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKB
gQDnBFiW+paeq5bSSXJ1P25ehNaRS3wFLeNwkIXYDJVxExg1xk/fWGBVcyFyZvdyAjPCM20h
V6yd3187IG25B955Jp+hLnswJRbxKrUq8ui1CW2kaljvjVpQUVV45DUtwlK16REMjF6AdP+n
vVWS3Fg8ooCwANmCNUpFWwrqtmulrQIDAQABozwwOjAOBgNVHQ8BAf8EBAMCBsAwGgYDVR0R
BBMwEYEPcm9kb2xwaGVAd29sLmJlMAwGA1UdEwEB/wQCMAAwDQYJKoZIhvcNAQEFBQADgYEA
JooMI8DXIvQvAnAzxFsgUVzRJ3cJtrChsjzpCOu+z4GAKxe47SJDFBpg3XiePsBwqIjJQTxV
cnk5wKLM5rnIowEv7LRWhEA92Qq/SayEEUFISGb6Q0HTJEeflo9+sUGuxBCnLKJ5Y/Th+dUS
GhvGSlhPws/dz8IL7Xw140gx1RcwggM/MIICqKADAgECAgENMA0GCSqGSIb3DQEBBQUAMIHR
MQswCQYDVQQGEwJaQTEVMBMGA1UECBMMV2VzdGVybiBDYXBlMRIwEAYDVQQHEwlDYXBlIFRv
d24xGjAYBgNVBAoTEVRoYXd0ZSBDb25zdWx0aW5nMSgwJgYDVQQLEx9DZXJ0aWZpY2F0aW9u
IFNlcnZpY2VzIERpdmlzaW9uMSQwIgYDVQQDExtUaGF3dGUgUGVyc29uYWwgRnJlZW1haWwg
Q0ExKzApBgkqhkiG9w0BCQEWHHBlcnNvbmFsLWZyZWVtYWlsQHRoYXd0ZS5jb20wHhcNMDMw
NzE3MDAwMDAwWhcNMTMwNzE2MjM1OTU5WjBiMQswCQYDVQQGEwJaQTElMCMGA1UEChMcVGhh
d3RlIENvbnN1bHRpbmcgKFB0eSkgTHRkLjEsMCoGA1UEAxMjVGhhd3RlIFBlcnNvbmFsIEZy
ZWVtYWlsIEIzc3VpbmcgQ0EwgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBAMSmPFVzVftO
ucqZWh5owHUEcJ3f6f+jHuy9zfVb8hp2vX8MOmHyv1HOAdTlUAow1wJjWiyJFXCO3cnwK4Va
qj9xVsuvPAsH5/EfkTYkKhPPK9Xzgnc9A74r/rsYPge/QIACZNenprufZdHFKlSFD0gEf6e2
0TxhBEAeZBlyYLf7AgMBAAGjgZQwgZEwEgYDVR0TAQH/BAgwBgEB/wIBADBDBgNVHR8EPDA6
MDigNqA0hjJodHRwOi8vY3JsLnRoYXd0ZS5jb20vVGhhd3RlUGVyc29uYWxGcmVlbWFpbENB
LmNybDALBgNVHQ8EBAMCAQYwKQYDVR0RBCIwIKQeMBwxGjAYBgNVBAMTEVByaXZhdGVMYWJl
bDItMTM4MA0GCSqGSIb3DQEBBQUAA4GBAEiM0VCD6gsuzA2jZqxnD3+vrL7CF6FDlpSdf0wh
uPg2H6otnzYvwPQcUCCTcDz9reFhYsPZOhl+hLGZGwDFGguCdJ4lUJRix9sncVcljd2pnDmO
jCBPZV+V2vf3h9bGCE6u9uo05RAaWzVNd+NWIXiC3CEZNd4ksdMdRv9dX2VPMYIB0DCCAcwC
```

```
AQEwdjBiMQswCQYDVQQGEwJaQTElMCMGA1UEChMcVGhhd3RlIENvbnN1bHRpbmcgKFB0eSkg
THRkLjEsMCoGA1UEAxMjVGhhd3RlIFBlcnNvbmFsIEZyZWVtYWlsIEl1zc3VpbmcgQ0ECEBg0
5P46uuRnNscYYh+/pM0wCQYFKw4DAhoFAKCBsTAYBgkqhkiG9w0BCQMxCwYJKoZIhvcNAQcB
MBwGCSqGSIb3DQEJBTEPFw0wNjA5MjMxMDU5MzlaMCMGCSqGSIb3DQEJBDEWBBSJQ4TP1u4v
IstvlVrallnxv9ziLTBSBgkqhkiG9w0BCQ8xRTBDMAoGCCqGSIb3DQMHMA4GCCqGSIb3DQMC
AgIAgDANBggqhkiG9w0DAgIBQDAHBgUrDgMCBzANBggqhkiG9w0DAgIBKDANBgkqhkiG9w0B
AQEFAASBgMK24+KFl6fZAygXQAcaIaW/CngGwURMfah2gUDtwJZniN+LdKc+TuQ/Vf49dE7X
W8N/RSG1g9WsymdLF8FQE5sQbf8panh11XpSi1nsyxjUfXRtpoWWnISLVwC7JKOC0+1wtA4C
EiCtMp+8wAOP1SHJbTOCk7Y+s/x0gPeI3HstAAAAAAAA
--------------ms040407070104030001050006--
```

– Décodage de la signature

OpenSSL permet de lire le fichier signature mime.p7s, avec la commande :

```
openssl asn1parse –inform DER –in mime.p7s –i >dump.txt
```

On obtient :

```
  0:d=0  hl=2 l=inf  cons: SEQUENCE
  2:d=1  hl=2 l=   9 prim:  OBJECT            :pkcs7-signedData
 13:d=1  hl=2 l=inf  cons:  cont [ 0 ]
 15:d=2  hl=2 l=inf  cons:   SEQUENCE
 17:d=3  hl=2 l=   1 prim:    INTEGER           :01
 20:d=3  hl=2 l=  11 cons:    SET
 22:d=4  hl=2 l=   9 cons:     SEQUENCE
 24:d=5  hl=2 l=   5 prim:      OBJECT            :sha1
 31:d=5  hl=2 l=   0 prim:      NULL
 33:d=3  hl=2 l=inf  cons:    SEQUENCE
 35:d=4  hl=2 l=   9 prim:     OBJECT            :pkcs7-data
 46:d=4  hl=2 l=   0 prim:     EOC
 48:d=3  hl=4 l=1451 cons:    cont [ 0 ]
 52:d=4  hl=4 l= 612 cons:     SEQUENCE
 56:d=5  hl=4 l= 461 cons:      SEQUENCE
 60:d=6  hl=2 l=   3 cons:       cont [ 0 ]
 62:d=7  hl=2 l=   1 prim:        INTEGER           :02
 65:d=6  hl=2 l=  16 prim:       INTEGER           :1834E4FE3ABAE46736C718621FBFA4CD
 83:d=6  hl=2 l=  13 cons:       SEQUENCE
 85:d=7  hl=2 l=   9 prim:        OBJECT            :sha1WithRSAEncryption
 96:d=7  hl=2 l=   0 prim:        NULL
 98:d=6  hl=2 l=  98 cons:       SEQUENCE
100:d=7  hl=2 l=  11 cons:        SET
102:d=8  hl=2 l=   9 cons:         SEQUENCE
104:d=9  hl=2 l=   3 prim:          OBJECT            :countryName
109:d=9  hl=2 l=   2 prim:          PRINTABLESTRING   :ZA
113:d=7  hl=2 l=  37 cons:        SET
115:d=8  hl=2 l=  35 cons:         SEQUENCE
117:d=9  hl=2 l=   3 prim:          OBJECT            :organizationName
122:d=9  hl=2 l=  28 prim:          PRINTABLESTRING   :Thawte Consulting (Pty) Ltd.
152:d=7  hl=2 l=  44 cons:        SET
154:d=8  hl=2 l=  42 cons:         SEQUENCE
156:d=9  hl=2 l=   3 prim:          OBJECT            :commonName
161:d=9  hl=2 l=  35 prim:          PRINTABLESTRING   :Thawte Personal Freemail Issuing CA
198:d=6  hl=2 l=  30 cons:       SEQUENCE
200:d=7  hl=2 l=  13 prim:        UTCTIME           :060801091141Z
215:d=7  hl=2 l=  13 prim:        UTCTIME           :070801091141Z
230:d=6  hl=2 l=  65 cons:       SEQUENCE
232:d=7  hl=2 l=  31 cons:        SET
234:d=8  hl=2 l=  29 cons:         SEQUENCE
236:d=9  hl=2 l=   3 prim:          OBJECT            :commonName
241:d=9  hl=2 l=  22 prim:          PRINTABLESTRING   :Thawte Freemail Member
265:d=7  hl=2 l=  30 cons:        SET
267:d=8  hl=2 l=  28 cons:         SEQUENCE
269:d=9  hl=2 l=   9 prim:          OBJECT            :emailAddress
280:d=9  hl=2 l=  15 prim:          IA5STRING         :rodolphe@wol.be
297:d=6  hl=3 l= 159 cons:       SEQUENCE
300:d=7  hl=2 l=  13 cons:        SEQUENCE
302:d=8  hl=2 l=   9 prim:         OBJECT            :rsaEncryption
313:d=8  hl=2 l=   0 prim:         NULL
315:d=7  hl=3 l= 141 prim:        BIT STRING
459:d=6  hl=2 l=  60 cons:       cont [ 3 ]
461:d=7  hl=2 l=  58 cons:        SEQUENCE
463:d=8  hl=2 l=  14 cons:         SEQUENCE
```

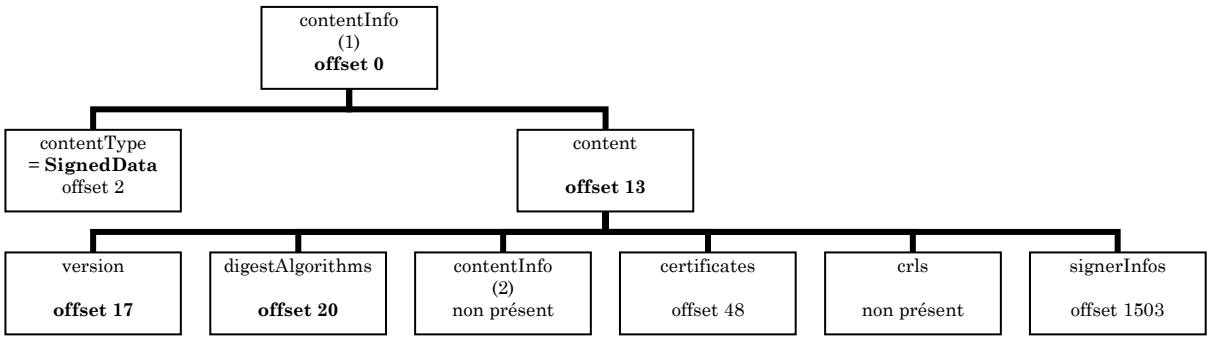
```
  465:d=9  hl=2 l=   3 prim:          OBJECT            :X509v3 Key Usage
  470:d=9  hl=2 l=   1 prim:          BOOLEAN           :255
  473:d=9  hl=2 l=   4 prim:          OCTET STRING      [HEX DUMP]:030206C0
  479:d=8  hl=2 l=  26 cons:         SEQUENCE
  481:d=9  hl=2 l=   3 prim:          OBJECT            :X509v3 Subject Alternative Name
  486:d=9  hl=2 l=  19 prim:          OCTET STRING      [HEX
DUMP]:3011810F726F646F6C70686540776F6C2E6265
  507:d=8  hl=2 l=  12 cons:         SEQUENCE
  509:d=9  hl=2 l=   3 prim:          OBJECT            :X509v3 Basic Constraints
  514:d=9  hl=2 l=   1 prim:          BOOLEAN           :255
  517:d=9  hl=2 l=   2 prim:          OCTET STRING      [HEX DUMP]:3000
  521:d=5  hl=2 l=  13 cons:      SEQUENCE
  523:d=6  hl=2 l=   9 prim:       OBJECT            :sha1WithRSAEncryption
  534:d=6  hl=2 l=   0 prim:       NULL
  536:d=5  hl=3 l= 129 prim:      BIT STRING
  668:d=4  hl=4 l= 831 cons:     SEQUENCE
  672:d=5  hl=4 l= 680 cons:      SEQUENCE
  676:d=6  hl=2 l=   3 cons:       cont [ 0 ]
  678:d=7  hl=2 l=   1 prim:        INTEGER           :02
  681:d=6  hl=2 l=   1 prim:       INTEGER           :0D
  684:d=6  hl=2 l=  13 cons:       SEQUENCE
  686:d=7  hl=2 l=   9 prim:        OBJECT            :sha1WithRSAEncryption
  697:d=7  hl=2 l=   0 prim:        NULL
  699:d=6  hl=3 l= 209 cons:       SEQUENCE
  702:d=7  hl=2 l=  11 cons:        SET
  704:d=8  hl=2 l=   9 cons:         SEQUENCE
  706:d=9  hl=2 l=   3 prim:          OBJECT            :countryName
  711:d=9  hl=2 l=   2 prim:          PRINTABLESTRING   :ZA
  715:d=7  hl=2 l=  21 cons:        SET
  717:d=8  hl=2 l=  19 cons:         SEQUENCE
  719:d=9  hl=2 l=   3 prim:          OBJECT            :stateOrProvinceName
  724:d=9  hl=2 l=  12 prim:          PRINTABLESTRING   :Western Cape
  738:d=7  hl=2 l=  18 cons:        SET
  740:d=8  hl=2 l=  16 cons:         SEQUENCE
  742:d=9  hl=2 l=   3 prim:          OBJECT            :localityName
  747:d=9  hl=2 l=   9 prim:          PRINTABLESTRING   :Cape Town
  758:d=7  hl=2 l=  26 cons:        SET
  760:d=8  hl=2 l=  24 cons:         SEQUENCE
  762:d=9  hl=2 l=   3 prim:          OBJECT            :organizationName
  767:d=9  hl=2 l=  17 prim:          PRINTABLESTRING   :Thawte Consulting
  786:d=7  hl=2 l=  40 cons:        SET
  788:d=8  hl=2 l=  38 cons:         SEQUENCE
  790:d=9  hl=2 l=   3 prim:          OBJECT            :organizationalUnitName
  795:d=9  hl=2 l=  31 prim:          PRINTABLESTRING   :Certification Services Division
  828:d=7  hl=2 l=  36 cons:        SET
  830:d=8  hl=2 l=  34 cons:         SEQUENCE
  832:d=9  hl=2 l=   3 prim:          OBJECT            :commonName
  837:d=9  hl=2 l=  27 prim:          PRINTABLESTRING   :Thawte Personal Freemail CA
  866:d=7  hl=2 l=  43 cons:        SET
  868:d=8  hl=2 l=  41 cons:         SEQUENCE
  870:d=9  hl=2 l=   9 prim:          OBJECT            :emailAddress
  881:d=9  hl=2 l=  28 prim:          IA5STRING         :personal-freemail@thawte.com
  911:d=6  hl=2 l=  30 cons:       SEQUENCE
  913:d=7  hl=2 l=  13 prim:        UTCTIME           :030717000000Z
  928:d=7  hl=2 l=  13 prim:        UTCTIME           :130716235959Z
  943:d=6  hl=2 l=  98 cons:       SEQUENCE
  945:d=7  hl=2 l=  11 cons:        SET
  947:d=8  hl=2 l=   9 cons:         SEQUENCE
  949:d=9  hl=2 l=   3 prim:          OBJECT            :countryName
  954:d=9  hl=2 l=   2 prim:          PRINTABLESTRING   :ZA
  958:d=7  hl=2 l=  37 cons:        SET
  960:d=8  hl=2 l=  35 cons:         SEQUENCE
  962:d=9  hl=2 l=   3 prim:          OBJECT            :organizationName
  967:d=9  hl=2 l=  28 prim:          PRINTABLESTRING   :Thawte Consulting (Pty) Ltd.
  997:d=7  hl=2 l=  44 cons:        SET
  999:d=8  hl=2 l=  42 cons:         SEQUENCE
 1001:d=9  hl=2 l=   3 prim:          OBJECT            :commonName
 1006:d=9  hl=2 l=  35 prim:          PRINTABLESTRING   :Thawte Personal Freemail Issuing CA
 1043:d=6  hl=3 l= 159 cons:       SEQUENCE
 1046:d=7  hl=2 l=  13 cons:        SEQUENCE
 1048:d=8  hl=2 l=   9 prim:         OBJECT            :rsaEncryption
 1059:d=8  hl=2 l=   0 prim:         NULL
 1061:d=7  hl=3 l= 141 prim:        BIT STRING
 1205:d=6  hl=3 l= 148 cons:       cont [ 3 ]
 1208:d=7  hl=3 l= 145 cons:        SEQUENCE
 1211:d=8  hl=2 l=  18 cons:         SEQUENCE
 1213:d=9  hl=2 l=   3 prim:          OBJECT            :X509v3 Basic Constraints
```

```
 1218:d=9  hl=2 l=   1 prim:          BOOLEAN           :255
 1221:d=9  hl=2 l=   8 prim:          OCTET STRING      [HEX DUMP]:30060101FF020100
 1231:d=8  hl=2 l=  67 cons:         SEQUENCE
 1233:d=9  hl=2 l=   3 prim:          OBJECT            :X509v3 CRL Distribution Points
 1238:d=9  hl=2 l=  60 prim:          OCTET STRING      [HEX
DUMP]:303A3038A036A0348632687474703A2F2F63726C2E7468617774652E636F6D2F5468617774655065726572736F6E
616C467265656D61696C43412E63726C
 1300:d=8  hl=2 l=  11 cons:         SEQUENCE
 1302:d=9  hl=2 l=   3 prim:          OBJECT            :X509v3 Key Usage
 1307:d=9  hl=2 l=   4 prim:          OCTET STRING      [HEX DUMP]:03020106
 1313:d=8  hl=2 l=  41 cons:         SEQUENCE
 1315:d=9  hl=2 l=   3 prim:          OBJECT            :X509v3 Subject Alternative Name
 1320:d=9  hl=2 l=  34 prim:          OCTET STRING      [HEX
DUMP]:3020A41E301C311A3018060355040313115072697661746546C6162656C322D313338
 1356:d=5  hl=2 l=  13 cons:      SEQUENCE
 1358:d=6  hl=2 l=   9 prim:       OBJECT            :sha1WithRSAEncryption
 1369:d=6  hl=2 l=   0 prim:       NULL
 1371:d=5  hl=3 l= 129 prim:      BIT STRING
 1503:d=3  hl=4 l= 464 cons:    SET
 1507:d=4  hl=4 l= 460 cons:     SEQUENCE
 1511:d=5  hl=2 l=   1 prim:      INTEGER           :01
 1514:d=5  hl=2 l= 118 cons:      SEQUENCE
 1516:d=6  hl=2 l=  98 cons:       SEQUENCE
 1518:d=7  hl=2 l=  11 cons:        SET
 1520:d=8  hl=2 l=   9 cons:         SEQUENCE
 1522:d=9  hl=2 l=   3 prim:          OBJECT            :countryName
 1527:d=9  hl=2 l=   2 prim:          PRINTABLESTRING   :ZA
 1531:d=7  hl=2 l=  37 cons:        SET
 1533:d=8  hl=2 l=  35 cons:         SEQUENCE
 1535:d=9  hl=2 l=   3 prim:          OBJECT            :organizationName
 1540:d=9  hl=2 l=  28 prim:          PRINTABLESTRING   :Thawte Consulting (Pty) Ltd.
 1570:d=7  hl=2 l=  44 cons:        SET
 1572:d=8  hl=2 l=  42 cons:         SEQUENCE
 1574:d=9  hl=2 l=   3 prim:          OBJECT            :commonName
 1579:d=9  hl=2 l=  35 prim:          PRINTABLESTRING   :Thawte Personal Freemail Issuing CA
 1616:d=6  hl=2 l=  16 prim:       INTEGER           :1834E4FE3ABAE46736C718621FBFA4CD
 1634:d=5  hl=2 l=   9 cons:      SEQUENCE
 1636:d=6  hl=2 l=   5 prim:       OBJECT            :sha1
 1643:d=6  hl=2 l=   0 prim:       NULL
 1645:d=5  hl=3 l= 177 cons:      cont [ 0 ]
 1648:d=6  hl=2 l=  24 cons:       SEQUENCE
 1650:d=7  hl=2 l=   9 prim:        OBJECT            :contentType
 1661:d=7  hl=2 l=  11 cons:        SET
 1663:d=8  hl=2 l=   9 prim:         OBJECT            :pkcs7-data
 1674:d=6  hl=2 l=  28 cons:       SEQUENCE
 1676:d=7  hl=2 l=   9 prim:        OBJECT            :signingTime
 1687:d=7  hl=2 l=  15 cons:        SET
 1689:d=8  hl=2 l=  13 prim:         UTCTIME           :060923105939Z
 1704:d=6  hl=2 l=  35 cons:       SEQUENCE
 1706:d=7  hl=2 l=   9 prim:        OBJECT            :messageDigest
 1717:d=7  hl=2 l=  22 cons:        SET
 1719:d=8  hl=2 l=  20 prim:         OCTET STRING      [HEX
DUMP]:894384CFD6EE2F22CB6F955ADA9659F1BFDCE22D
 1741:d=6  hl=2 l=  82 cons:       SEQUENCE
 1743:d=7  hl=2 l=   9 prim:        OBJECT            :S/MIME Capabilities
 1754:d=7  hl=2 l=  69 cons:        SET
 1756:d=8  hl=2 l=  67 cons:         SEQUENCE
 1758:d=9  hl=2 l=  10 cons:          SEQUENCE
 1760:d=10 hl=2 l=   8 prim:           OBJECT            :des-ede3-cbc
 1770:d=9  hl=2 l=  14 cons:          SEQUENCE
 1772:d=10 hl=2 l=   8 prim:           OBJECT            :rc2-cbc
 1782:d=10 hl=2 l=   2 prim:           INTEGER           :80
 1786:d=9  hl=2 l=  13 cons:          SEQUENCE
 1788:d=10 hl=2 l=   8 prim:           OBJECT            :rc2-cbc
 1798:d=10 hl=2 l=   1 prim:           INTEGER           :40
 1801:d=9  hl=2 l=   7 cons:          SEQUENCE
 1803:d=10 hl=2 l=   5 prim:           OBJECT            :des-cbc
 1810:d=9  hl=2 l=  13 cons:          SEQUENCE
 1812:d=10 hl=2 l=   8 prim:           OBJECT            :rc2-cbc
 1822:d=10 hl=2 l=   1 prim:           INTEGER           :28
 1825:d=5  hl=2 l=  13 cons:      SEQUENCE
 1827:d=6  hl=2 l=   9 prim:       OBJECT            :rsaEncryption
 1838:d=6  hl=2 l=   0 prim:       NULL
 1840:d=5  hl=3 l= 128 prim:      OCTET STRING      [HEX
DUMP]:C2B6E3E28597A7D903281740071A21A5BF0A7806C1444C7DA8768140EDC0966788DF8B74A73E4EE43F55FE3D
744ED75BC37F4521B583D5ACCA674B17C150139B106DFF296A7875D57A528B59ECCB18D47D746DA685969C848B5700
BB24A382D3ED70B40E021220AD329FBCC0038FD521C96D338293B63EB3FC7480F788DC7B2D
```

```
1971:d=3  hl=2 l=   0 prim:    EOC
1973:d=2  hl=2 l=   0 prim:   EOC
1975:d=1  hl=2 l=   0 prim:  EOC
```

Le premier nombre de chaque ligne indique l'*offset* de l'élément en octets par rapport au début du fichier. Voici ce que contient cette signature :

- `version` : version 2 (offset 17)
- `digestAlgorithms` (offset 20)
    - SHA1 (offset 24)
- `contentInfo` : non présent, il s'agit d'une signature détachée du contenu.
- `certificates` (offset 48)
    - 1e certificat (offset 52) : CN =Thawte Freemail Member (offset 241)
    - 2e certificat (offset 668) : CN = Thawte Pernonal Freemail Issuing CA (offset 1006)
- `crls` : non présent
    - `signerInfos` informations sur les signataires (offset 1503)
    - 1 seul signataire (offset 1507)
        - version (offset 1511)
        - identification du certificat : nom X500 de l'émetteur(offset 1514) et numéro de série de l'émetteur (offset 1616)
        - algorithme de hachage (offset 1634): SHA1
        - propriétés signées (offset 1645) : contentType, signingTime, messageDigest, S/MIME capabilities
        - signature (offset 1840)



Annexe D : Un exemple de signature XML

Voici un exemple de signature au format XMLSignature, conformément à [13], générée avec l'outil *WebSign*. Cette signature comprend deux références :

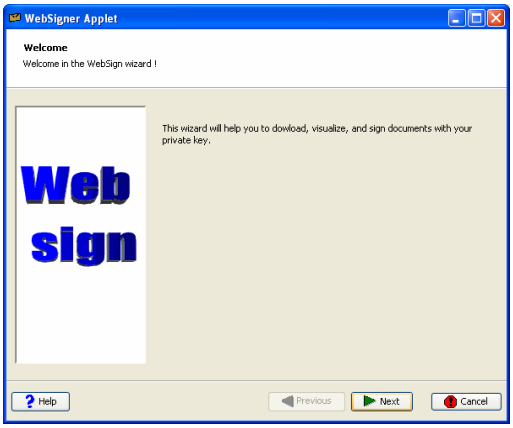
– la date de la signature, incluse à la fin du document XML

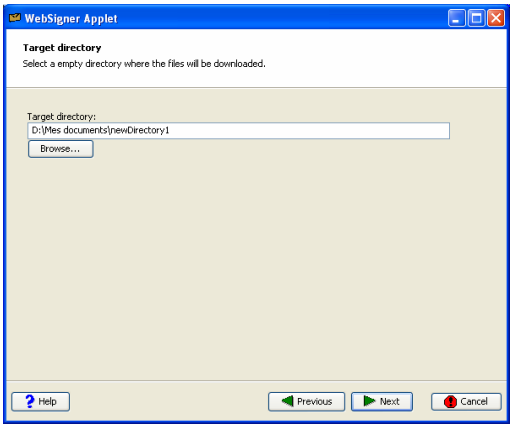
– un fichier 'contrat.doc'

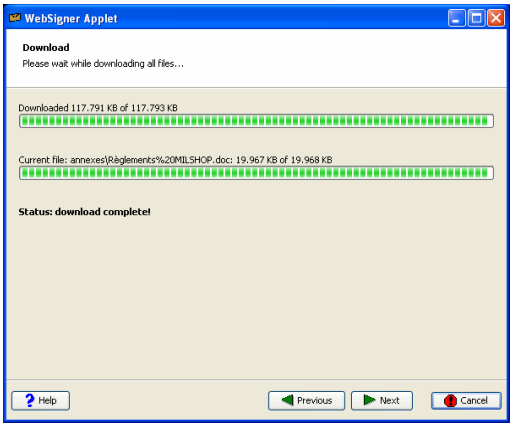
```
<?xml version="1.0" encoding="UTF-8"?>
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#" Id="Signature-1">
  <SignedInfo>
    <CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
    <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
    <Reference URI="#TimeStamp">
      <Transforms>
      <Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
      </Transforms>
      <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <DigestValue>dV5T7wd4iTCkUIuBJwLk8C3aFQ4=</DigestValue>
    </Reference>
    <Reference URI="contrat.doc">
      <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <DigestValue>5I0nnVeCcqo566wqnDyvqhIN1+A=</DigestValue>
    </Reference>
  </SignedInfo>
<SignatureValue>
   JW8WUV4JpEG7Xy5qA8N+59GX/H44Iv9Y2j9NNrn+gxm9Mfq/Yuu4ORyCVa8x/nPZOs6neoYRs89z
   brNV7Ojq6sgxpTau/EAkOYQtueB845CU6KOza816BO5xccgfyxk94G6Sf/+n/fz/pUmKfxFYhStX
   Gn48WEIIkgyGgYrk8q0=
</SignatureValue>
<KeyInfo>
  <X509Data>
     <X509Certificate>
      MIIEHTCCAwWgAwIBAgIQEAAAAAAAizj61uUojhxp4DANBgkqhkiG9w0BAQUFADAzMQswCQYDVQQG
      EwJCRTETMBEGA1UEAxMKQ2l0aXplbiBDQTEPMA0GA1UEBRMGMjAwNTAxMB4XDTA1MDQwODE2MzU1
      NloXDTEwMDcwODE2MzU1NlowgY8xCzAJBgNVBAYTAkJFMTEwLwYDVQQDEyhSb2RvbHBoZSBDYXJk
      b24gZGUgTGljaHRidWVyIChTaWduYXR1cmUpMRwwGgYDVQQEExNDYXJkb24gZGUgTGljaHRidWVy
      MRkwFwYDVQQqExBSb2RvbHBoZSBMaWR3aW5lMRQwEgYDVQQFEws4MzEwMjExNjM1MDCBnzANBgkq
      hkiG9w0BAQEFAAOBjQAwgYkCgYEAiBABzns7UC1aVaFGMOnm0KSFZ12GqEQEjRjtpmDq8KAg8SXc
      3rrO86JHv4mrWgxcOCtHJRHSpJQEmouL9uq+jmz8AOP1GtRB3jx4cTJ7paf3+79+4dt7Edrs38+l
      NyX6VLp1t+I+nzyly6o/afgcj3GYk/DnS3NOMwPXPLJxicMCAwEAAaOCAVIwggFOMEQGA1UdIAQ9
      MDswOQYHYDgBAQECATAuMCwGCCsGAQUFBwIBFiBodHRwOi8vcmVwb3NpdG9yeS5laWQuYmVsZ2l1
      bS5iZTAOBgNVHQ8BAf8EBAMCBkAwHwYDVR0jBBgwFoAUVjhIbORzA9iJt74pALnBmy56inEwOQYD
      VR0fBDIwMDAuoCygKoYoaHR0cDovL2NybC5laWQuYmVsZ2l1bS5iZS9laWRjMjAwNTAxLmNybDAR
      BglghkgBhvhCAQEEBAMCBSAwbQYIKwYBBQUHAQEEYTBfMDUGCCsGAQUFBzAChilodHRwOi8vY2Vy
      dHMuZWlkLmJlbGdpdW0uYmUvYmVsZ2l1bXJzLmNydDAmBggrBgEFBQcwAYYaaHR0cDovL29jc3Au
      ZWlkLmJlbGdpdW0uYmUwGAYIKwYBBQUHAQMEDDAKMAgGBgQAjkYBATANBgkqhkiG9w0BAQUFAAOC
      AQEAWBLVWoq8HtKHKpCxI5jMOydAm6KBoKpmWd+YmKQfNbZTjO2YX/JUQkECPp4e6drmh/AYCR9g
      VN5IKR+LGV3EVyqgunT3f2bXzdjCrRYKc/u3hUfkf0dqhCEjUs0TMCzp/J3MkoGGqUm1ddYUDyak
      yeUd/kBKesvj1BMKjUg4LmLZqu9bYBzGlHGDMXC0/KLdxPH8xVZhDDcKWqNxqRr3M14CLpJv++gN
      QIdZrxuC8dSZWXjma4Sj6r9slycGZbNkjz+Q0OZRuNMvcwk/XsYJEPl2RH2oSwTrDHifpQzdkFNa
      35ytetaTGfaALC4v/GcOz0ss6U6fKz+bR2P5PGLpmQ==
      </X509Certificate>
      <X509Certificate>
      MIID3DCCAsSgAwIBAgIQbiNUqcvVvTEsHnlbQHH22TANBgkqhkiG9w0BAQUFADAnMQswCQYDVQQG
      EwJCRTEYMBYGA1UEAxMPQmVsZ2l1bSBSb290IENBMB4XDTA0MTIyMzExMDAwMFoXDTExMDgyMzEw
      MDAwMFowMzELMAkGA1UEBhMCQkUxEzARBgNVBAMTCkNpdGl6ZW4gQ0ExDzANBgNVBAUTBjIwMDUw
      MTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALh7f7+jQgYSL4zJTZoJlisDDZ83/Zvd
      Qu3pldzUxDDTUV6GlEXG1dKI0XVg6NSNeES7rHCX/uwNZxf0zEYg/YSi7UXzpxWFt4PGZplCXV+q
      H+xJVIzvNCbK9E/k3Zp2rpoPmW0lGYtgRgVW3aB5deu4F37OBe1lnsmasYmfs+Bqge0tRJUUPU+o
      +AFLq8QtgpkKsHMHqq2qTKZW0BqsnvERPmJRcixFFLfWS9tBxhMxz+QcI0fFluiHloIJ+yvzqrv3
      lwfmZiS+yYOc87HwwqegNxvyD8QihJL7SOHBeTnFf/OqyyDm4tVYwGEcw0g02lctaXYxjkV2tyZ3
      5feXm0sCAwEAAaOB9zCB9DAOBgNVHQ8BAf8EBAMCAQYwEgYDVR0TAQH/BAgwBgEB/wIBADBDBgNV
      HSAEPDA6MDgGBmA4AQEBAjAuMCwGCCsGAQUFBwIBFiBodHRwOi8vcmVwb3NpdG9yeS5laWQuYmVs
      Z2l1bS5iZTAdBgNVHQ4EFgQUVjhIbORzA9iJt74pALnBmy56inEwNgYDVR0fBC8wLTAroCmgJ4Yl
      aHR0cDovL2NybC5laWQuYmVsZ2l1bS5iZS9iZWxnaXVtLmNybDARBglghkgBhvhCAQEEBAMCAAcw
      HwYDVR0jBBgwFoAUEPAMVpth6lc6tjWXbZ/duRSO2+YwDQYJKoZIhvcNAQEFBQADggEBACNL9AxH
      bquAB/nJwuHhFmRfq8DyClRWQzOiYHvntACTyCF3oaUty7NSwgNLPgXaL6Kfj4ZYpDvQifFlYrkb
      j42kdXt1inLW93g1bsi8Guid5m5L+oOWbyyNy2X7US4YRezFQh52RxJwnzi6bOzFRlBJHiLjSTDZ
      yuZFmbS+Xs2i16kyVKjDREpShSuop8dDR+w7kpGJ8U4oLt8cbW0/XQ849jXp5EXcu2a4PcpNIzB5
      0ts6ysBUqb9UeAcLXQn4/madyvSyI+lcfCOkaYBlYlqSAMilGt/MKivmFBgCqmk9gU1OiQGqacBr
      8iY8a8OEYLQ34rkkF+L0umkykOvjf18=
      </X509Certificate>
      <X509Certificate>
      MIIDlDCCAnygAwIBAgIQWAsFbFMk27JQVxhf+eWmUDANBgkqhkiG9w0BAQUFADAnMQswCQYDVQQG
      EwJCRTEYMBYGA1UEAxMPQmVsZ2l1bSBSb290IENBMB4XDTAzMDEyNjIzMDAwMFoXDTE0MDEyNjIz
      MDAwMFowJzELMAkGA1UEBhMCQkUxGDAWBgNVBAMTD0JlbGdpdW0gUm9vdCBDQTCCASIwDQYJKoZI
      hvcNAQEBBQADggEPADCCAQoCggEBAMihcekcRkJ5eHFvna6pqKsot03HIOswkVp19eLSz8hMFJhC
      WK3HEcVAQGpa+XQSJ4fpnOVxTiIs0RIYqjBeoiG52bv/9nTrMQHnO35YD5EWTXaJqAFPrSJmcPpL
      HZXBMFjqvNll2Jq0iOtJRlLf01MVdssUXRlJsW9q09P9vMIt7EU/CT9YvvzU7wCMgTVyv/cY6pZi
```

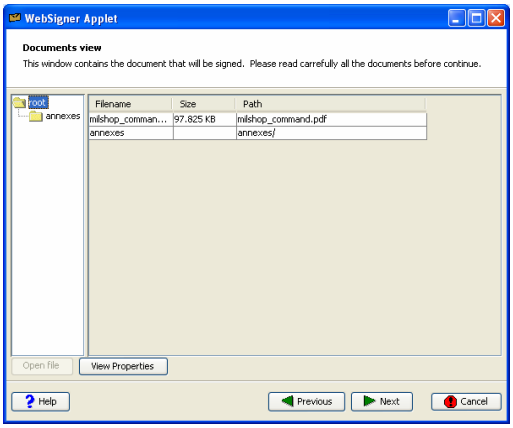
```
      fSsofxVsY9LKyn0FrMhtB20yvmi4BUCuVJhWPmbxMOjvxKuTXgfeMo8SdKpbNCNUwOpszv42kqgJ
      F+qhLc9s44Qd3ocuMws8dOIhUDiVLlzg5cYx+dtA+mqhpIqTm6chBocdJ9PEoclMsG8CAwEAAaOB
      uzCBuDAOBgNVHQ8BAf8EBAMCAQYwDwYDVR0TAQH/BAUwAwEB/zBCBgNVHSAEOzA5MDcGBWA4AQEB
      MC4wLAYIKwYBBQUHAgEWIGh0dHA6Ly9yZXBvc2l0b3J5LmVpZC5iZWxnaXVtLmJlMB0GA1UdDgQW
      BBQQ8AxWm2HqVzq2NZdtn925FI7b5jARBglghkgBhvhCAQEEBAMCAAcwHwYDVR0jBBgwFoAUEPAM
      Vpth6lc6tjWXbZ/duRSO2+YwDQYJKoZIhvcNAQEFBQADggEBAMhtIlGKYfgPlm7VILKB+MbcoxYA
      2s1q52sq+llIp0xJN9dzoWoBZV4yveeX09AuPHPTjHuD79ZCwT+oqV0PN7p20kC9zC0/00RBSZz9
      Wyn0AiMiW3Ebv1jZKE4tRfTa57VjRUQRDSp/M382SbTObqkCMa5c/ciJv0J71/Fg8teH9lcuen5q
      E4Ad3OPQYx49cTGxYNSeCMqr8JTHSHVUgfMbrXec6LKP24OsjzRr6L/D2fVDw2RV6xq9NoY2uiGM
      lxoh1OotO6y67Kcdq765Sps1LxxcHVGnH1TtEpf/8m6HfUbJdNbv6z195lluBpQE5KJVhzgoaiJe
      4r50ErAEQyo=
     </X509Certificate>
     <X509IssuerSerial>
       <X509IssuerName>
          SERIALNUMBER=83102116350, GIVENNAME=Rodolphe Lidwine,
          SURNAME=Cardon de Lichtbuer, CN=Rodolphe Cardon de Lichtbuer (Signature),
          C=BE</X509IssuerName>
       <X509SerialNumber>212676479325593114264945860897488511 68</X509SerialNumber>
     </X509IssuerSerial>
   </X509Data>
</KeyInfo>
<Object>
   <SignatureProperty Id="TimeStamp" Target="#Signature-1">
      <FriendlyDate>ven., 22 sept. 2006 12:12:20 +0200</FriendlyDate>
      <UTCMillisecsFromEpoc>1158919940265</UTCMillisecsFromEpoc>
   </SignatureProperty>
</Object>
</Signature>
```

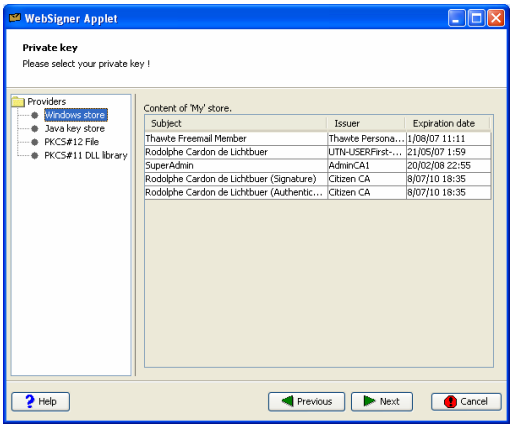
Annexe E : aperçu des fenêtres de l'applet

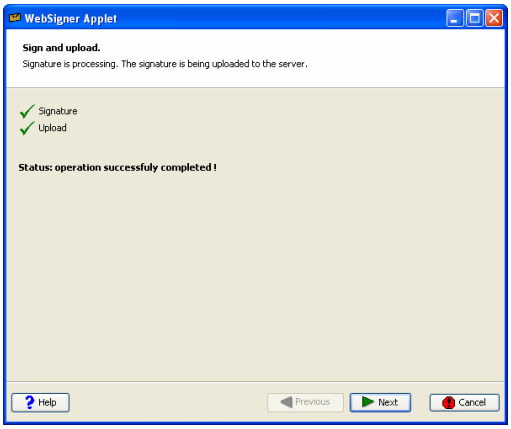


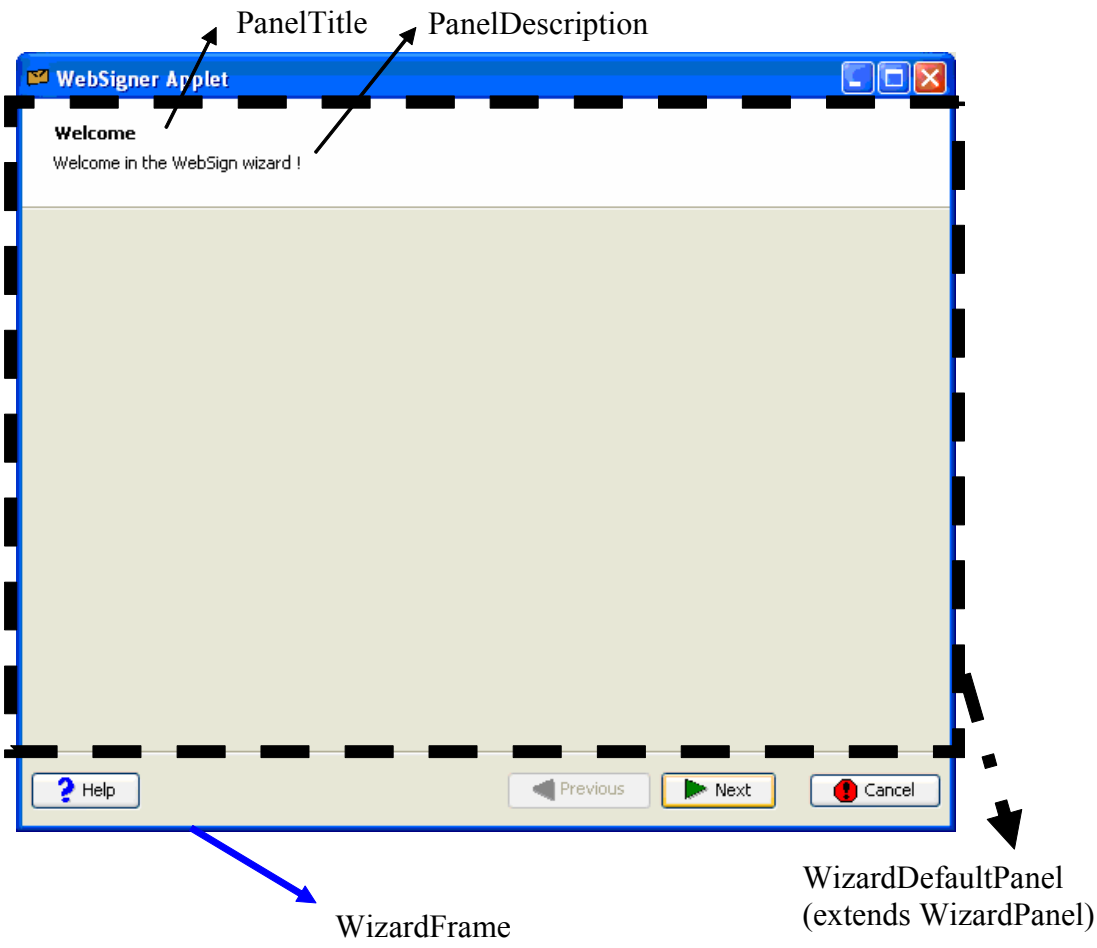


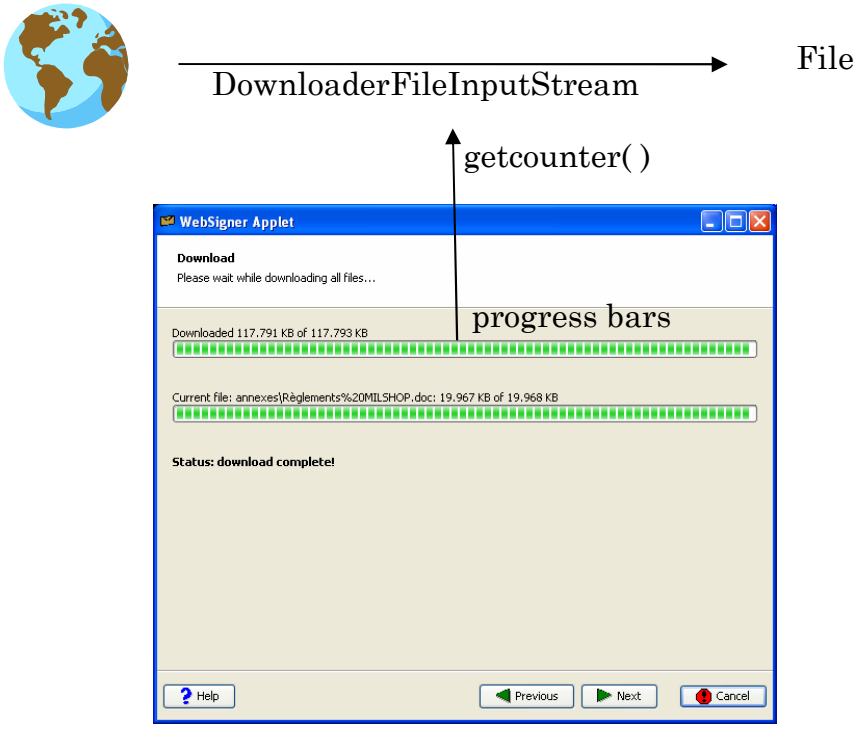









PanelTitle
PanelDescription
WebSigner Applet
Welcome
Welcome in the WebSign wizard !
Help
Previous
Next
Cancel
WizardFrame
WizardDefaultPanel
(extends WizardPanel)


DownloaderFileInputStream
File
getcounter( )
WebSigner Applet
Download
Please wait while downloading all files...
progress bars
Downloaded 117.791 KB of 117.793 KB
Current file: annexes\Règlements%20MILSHOP.doc: 19.967 KB of 19.968 KB
Status: download complete!
Help
Previous
Next
Cancel

Annexe F : architecture globale de l'application