Glasnost et la sécurité

Le but de ce document est de définir un cadre pour la mise en place d'une solution de sécurité autour du logiciel Glasnost. Différentes solutions seront étudiées et la plus appropriée, compte tenu des fonctionnalités de Glasnost et de son champ d'application sera préconisée.

1. Contexte et définitions

Glasnost est un logiciel de vote électronique, autour duquel viennent se greffer un certain nombre de services tels que la rédaction collaborative d'articles, la présentation d'informations sur Internet à l'aide d'un système de template, etc. La liste de ces services additionnels n'est pas limitée, grâce à la conception souple et modulaire de Glasnost.

Nous entendons par **système Glasnost** un logiciel Glasnost indépendant, implémentant toute ou partie des services mentionnés précédemment, sur une ou plusieurs machines ayant une même localisation physique. En réalité, un système Glasnost est constitué d'une multitude de serveurs locaux spécialisés qui travaillent de manière collaborative. Ces serveurs internes peuvent être répartis sur un ou plusieurs ordinateurs. Cependant, dans le cadre de ce document, nous ne descendront pas jusqu'à ce niveau de détail et nous nous contenterons de considérer un système Glasnost comme une entité unifiée et indépendante.

Un **serveur** est un logiciel ou un système délivrant des informations à des clients. Nous emploierons le terme de **client** pour désigner tout logiciel qui souhaite se connecter à un serveur. Parfois, les termes de client et d'utilisateur sont utilisés comme des synonymes, dans la mesure où l'utilisateur doit nécessairement passer par un logiciel client pour accéder aux ressources d'un serveur.

Par **utilisateur** nous désignons toute personne qui souhaite accéder à des services protégés proposés par un système Glasnost. En effet, Glasnost accepte deux types de profils : d'une part les « anonymes » (toute personne non identifiée), qui peuvent accéder à une partie restreinte des informations essentiellement en lecture, et d'autre part les utilisateurs identifiés qui, selon leurs droits, peuvent accéder à d'autres informations, rédiger des articles, créer et gérer des objets Glasnost, etc.

2. Deux aspects de la sécurité

Le problème de la sécurisation du logiciel Glasnost peut être divisé en deux parties : d'une part, l'authentification des systèmes Glasnost et la sécurisation des échanges entre eux, d'autre part l'authentification des utilisateurs se connectant à un système Glasnost.

2.1 Authentification des systèmes Glasnost

Les différents systèmes Glasnost peuvent être amenés à communiquer entre eux, afin d'échanger des données. Par exemple, un système peut demander à un autre système de lui communiquer un article ; un système peut faire participer ses utilisateurs à un vote situé sur un autre système. Les données échangées peuvent donc être sensibles et nécessitent l'authentification des systèmes : le système « client » doit être certain qu'il communique bien avec le système « serveur » souhaité. De même, le système « serveur » doit s'assurer de l'identité du système « client » qui demande l'accès à l'une de ses ressources.

Une fois établie l'identité des systèmes, un échange de données cryptées peut être initié, si cela se révèle nécessaire.

2.2 Authentification des utilisateurs

Les utilisateurs sont amenés à s'identifier afin de pouvoir accéder aux différents services d'un système Glasnost. La liste des services auxquels chaque utilisateur peut accéder ne fait pas partie du mécanisme d'identification, mais est gérée en interne par le système Glasnost, en associant l'utilisateur à différents groupes disposant de droits d'accès spécifiques. Par ailleurs, l'identification des utilisateurs doit être distinguée de la validation des bulletins de vote, cette partie étant traitée indépendamment.

L'authentification des utilisateurs doit permettre de s'assurer de leur identité. DAns le sens inverse, l'utilisateur doit être assuré qu'il communique bien avec le système Glasnost attendu.

2.3 Contraintes

Au niveau des systèmes GLasnost, nous avons peu de contraintes quant au choix des solutions de sécurisation, dans la mesure où il s'agit de logiciels que l'on peut configurer pour réaliser automatiquement les tâches d'authentification et de cryptage des échanges.

Au niveau des utilisateurs les contraintes sont plus importantes. En effet, les utilisateurs accèdent au système Glasnost *via* un navigateur Internet. Les méthodes d'authentification doivent donc être supportées par la majorité des navigateurs, sans ajout de plugin ni installation d'un élément logiciel quelconque. Tout au plus peut-on exécuter du code Java ou JavasScript sur le poste client.

Enfin, Glasnost étant un logiciel libre et open-source, nous devons respecter cette philosophie : les solutions adoptées pour la sécurisation devront, dans la mesure du possible, être elles-mêmes libres et s'appuyer sur des standards ouverts, largement connus et diffusés. Ainsi, l'utilisateur sera assuré qu'aucune information non souhaités ne soit transmise lors de la session d'authentification, et qu'aucun de espion ne sera utilisé à son insu. La sécurisation de Glasnost ne doit se traduire par aucun coût du point de vue de l'utilisateur.

3. Internet et la sécurité

Glasnost utilisant TCP/IP pour communiquer, nous devons nous pencher plus précisément sur les problèmes de sécurité liés aux réseaux de ce type. On peut distinguer essentiellement trois types d'interférences entre les informations transitant sur le réseau et une tierce personne mal intentionnée :

L'espionnage, qui laisse intactes les informations mais compromet leur confidentialité. Ce pourrait être par exemple le cas d'une personne interceptant un numéro de carte bancaire lors d'un échange avec un site de commerce électronique.

L'altération des données, qui consiste à modifier ou à remplacer les informations en transit avant qu'elles ne parviennent à leur destinataire. On peut imaginer par exemple que le contenu d'un bulletin de vote électronique soit altéré avant d'être envoyé au destinataire.

L'imposture, consistant à se faire passer pour une autre personne par usurpation d'id*entité (spoofing*) afin de se connecter à un serveur sous une fausse identité, ou, pour un serveur, à se faire passer pour un autre serveur. Un exemple typique est le cas d'un serveur se faisant passer pour un site de commerce électronique reconnu, mais encaissant les paiements sans jamais envoyer de marchandises en échange.

Différentes méthodes permettent de se protéger plus ou moins efficacement contre ces attaques :

Le cryptage des données permet aux deux parties de masquer les informations qu'elles échangent.

La protection contre les altérations assure le destinataire que les informations n'ont pas été modifiées entre le moment où elles ont été envoyées et le moment où elles sont reçues.

L'authentification, qui permet aux deux parties de s'assurer de l'identité de leur interlocuteur, et au destinataire des informations de déterminer sans équivoque l'origine des données reçues.

La non-répudiation, qui empêche l'expéditeur des informations de prétendre ultérieurement ne pas les avoir envoyées.

Dans le cadre de la sécurisation de Glasnost, nous allons plus particulièrement nous intéresser au problème de l'authentification, bien que les autres points évoqués puissent également trouver des réponses selon la méthode employée.

4. Les méthodes d'authentification

Dans le champ d'application qui nous intéresse, on peut distinguer deux grandes familles de systèmes d'authentification :

L'authentification par mot de passe. Dans ce cas, le serveur maintient un liste d'identifiants et de mots de passe et demande au client souhaitant se connecter de fournir son propre identifiant et son mot de passe. Le serveur vérifie la présence de l'identifiant dans sa liste locale et, s'il le trouve, contrôle que le mot de passe fourni par l'utilisateur correspond au mot de passe enregistré.

L'authentification par certificat. Le client dispose d'un certificat contenant son identité, une clé publique et des données cryptées au moyen de sa clé privée. Il envoie ce certificat au serveur auquel il souhaite se connecter et le serveur effectue sur ce certificat un certain nombre de contrôles lui permettant de s'assurer de la validité du certificat et de l'identité du client.

Nous analyserons plus en détail chacune de ces solutions dans les sections suivantes.

4.1 Authentification par mot de passe

L'authentification par mot de passe est basée sur le partage d'une information secrète (mot de passe) entre les deux parties cherchant à communiquer. Le client doit envoyer au serveur son identifiant et le mot de passe correspondant. Le serveur contrôle la présence de l'identifiant dans son fichier d'utilisateurs. Si l'identifiant est trouvé, il vérifie que le mot de passe envoyé correspond au mot de passe associé à cet identifiant.

Ce mécanisme est simple à mettre en place et très peu consommateur en ressources CPU. Ses principaux points faibles sont de la nécessité d'envoyer le mot de passe sur le réseau – entraînant un risque d'interception par une tierce personne – et le stockage d'un fichier de mots de passe sur le serveur – qui peut compromettre la sécurité du système s'il était révélé.

Différentes techniques permettent de réduire ces risques. Par exemple, au lieu de transmettre le mot de passe en clair, on envoie sur le réseau une « empreinte » calculée par hachage sur le mot de passe et certaines informations supplémentaires, comme un nombre aléatoire négocié entre les deux parties. De même, au lieu de stocker les mots de passe sur le serveur, on peut ne conserver que leurs « empreintes » (c'est le cas du fichier `/etc/passwd` d'Unix). L'un des défauts de cette parade est l'impossibilité de renvoyer à l'utilisateur son mot de passe en cas d'oubli, car il n'est pas possible de reconstruire le mot de passe à partir de son empreinte.

Certains protocole d'authentification par mot de passe (SRP par exemple), sont basés sur une série d'échanges de type question-réponse entre le serveur et le client. Mais même ces techniques robustes ne sont en général pas à l'abri d'une attaque de type « dictionnaire » : une tierce personne peut capturer les messages échangés au cours de la session d'authentification et en déduire le mot de passe utilisé. Ce type d'attaque est efficace dans la mesure ou la plupart des personnes utilisent des mots de passe courants et faciles à retenir, mais par la même occasion aisés à deviner.

L'authentification par mot de passe peut être intéressante lorsque le niveau de sécurité exigé reste faible. Il faut noter que ce type d'authentification ne fournit aucun indice sur l'identité réelle du client. Elle prouve simplement que l'utilisateur cherchant à se connecter a connaissance d'un identifiant et d'un mot de passe valides. Mais le serveur n'a aucun moyen de savoir si la personne se connectant est bien la personne supposée.

4.2 Authentification par certificat

Un certificat est un document électronique permettant d'identifier une personne, un serveur ou toute autre entité. Le certificat doit fournir une preuve reconnue de l'identité de son possesseur.

Le possesseur d'un certificat dispose d'une paire de clés : une **clé privée**, qui doit rester secrète, et une **clé publique** qui est largement diffusée. Le rôle du certificat consiste à associer de manière sûre et unique une clé publique avec une entité. Les certificats permettent d'éviter l'usurpation d'identité : la clé publique contenue dans le certificat ne pourra fonctionner qu'avec la clé privée possédée par le détenteur du certificat.

Les certificats sont délivrés par une **autorité de certification** (CA). Lorsqu'elle reçoit une demande de création d'un certificat, l'autorité contrôle un certain nombre d'informations concernant le demandeur avant de délivrer le certificat. Les méthodes de contrôle de l'identité du demandeur varient selon les autorités de certification. En général, les autorités de certification publient la procédure qu'elle utilisent. Le certificat délivré par une autorité de certification associe une clé publique particulière à l'entité identifiée par le certificat.

En dehors de la clé publique, les certificats contiennent d'autres informations, telles que la période de validité du certificat, le nom de l'autorité ayant délivré le certificat, un numéro de série unique, etc. Le certificat contient également la signature digitale de l'autorité de certification. Ainsi, le certificat peut fonctionner à la manière d'une « lettre d'introduction » pour les serveurs qui ne connaissent pas l'utilisateur, mais font confiance à l'autorité de certification ayant validé le certificat.

4.2.1 Signature digitale

Avant d'aborder en détail le mécanisme d'authentification par certificat, nous devons introduire la notion de **signature digitale**. Le possesseur d'une paire clé publique/clé privée peut utiliser sa clé privée pour crypter des données. Ceci présente peu d'intérêt pour la sécurité, car toute personne connaissant la clé publique correspondante (qui est par définition largement diffusée et facilement accessible) peut s'en servir pour décrypter les données. En revanche, on peut être assuré que seul le possesseur de la clé privée a pu les crypter.

La signature digitale exploite cette propriété. Pour signer des informations, le logiciel client crée une empreinte de ces informations (hachage) et crypte cette empreinte avec la clé privée de l'utilisateur. Les informations ainsi que l'empreinte cryptée sont alors envoyées au destinataire. Celui-ci décrypte l'empreinte au moyen de la clé publique de l'expéditeur, calcule sa propre empreinte des informations et compare les deux empreintes. Si elles correspondent, on peut être certain que les informations n'ont pas été altérées entre le moment de leur signature et leur arrivée à destination.

La signature digitale offre également un haut degré de non-répudiation. Le détenteur de la clé privée étant le seul à pouvoir fournir une signature digitale particulière pour les informations transmises, il pourra difficilement contester par la suite avoir envoyé ces informations.

4.2.2 Procédure d'authentification par certificat

Afin de s'authentifier auprès d'un serveur, le client signe des données aléatoires. Il envoie ensuite son certificat et les données signées sur le réseau. Le destinataire peut alors décrypter les données au moyen de la clé publique contenue dans le certificat afin de contrôler que l'expéditeur du certificat possède bien la clé privée correspondante.

La procédure complète d'authentification est décrite ci-dessous :

1. Le logiciel client dispose d'une base de données de clés privées qui correspondent aux clés publiques des certificats du client. Lorsque le client doit accéder à cette base de données (pour envoyer un certificat par exemple), il demande un mot de passe à l'utilisateur. Ce mot de passe ne sera saisi qu'une seule fois pour toute la session de travail.
2. Le client déverrouille la base de données et récupère la clé privée qui correspond au certificat à envoyer. Il utilise cette clé pour signer des données aléatoires provenant à la fois du client et du serveur. Ces données sont uniques pour chaque session d'authentification. Ces données et leur signature digitale constituent une preuve de la validité de la clé privée.
3. Le client envoie le certificat et les données signées au serveur.
4. Le serveur utilise la clé publique contenue dans le certificat pour valider les données signées et s'assurer ainsi de l'identité de l'utilisateur.

Le serveur peut effectuer des opérations d'identification complémentaires, en vérifiant par exemple que le certificat envoyé est enregistré dans l'entrée de ce client d'un annuaire LDAP. Le serveur peut ensuite déterminer si le client est autorisé à accéder aux ressources qu'il demande.

4.2.3 Infrastructures de clés publiques (PKI)

Nous avons vu que la validation des certificats nécessite la présence d'autorités de certification, qui contrôlent les identités et délivrent les certificats. Si un serveur fait confiance à une autorité de certification, il peut également faire confiance aux personnes munies de certificats délivrés par cette autorité. Par ailleurs, une autorité de certification peut également délivrer des certificats à d'autres autorités, créant ainsi un réseau d'autorités de certification. Ces réseaux sont organisées selon deux types de schémas :

- **Une organisation hiérarchique**, dans laquelle une autorité de certification se trouve au sommet d'une pyramide. Elle délivre des certificats à des autorités de certification subordonnées. Ces autorités subordonnées peuvent à leur tour délivrer des certificats à des autorités de niveau inférieur. Ce modèle d'organisation est bien établi et largement répandu. Il est utilisé pour les certificats X.509 mis au point par la société Netscape.
- **Une organisation répartie**, aussi appelée « Web of Trust ». Dans ce modèle, tout détenteur de certificat est potentiellement une autorité de certification. Chacun peut signer le certificat d'autres personnes, devenant ainsi un « introducteur » de cette personne auprès des autres. On peut également accorder plus ou moins de crédit à la capacité d'introducteur de tel ou tel acteur du réseau (certains validant les certificats avec sérieux, d'autres signant les certificats sans prendre la peine de les contrôler). Ce modèle d'organisation a encore peut d'applications réelles. Il est notamment utilisé par OpenPGP.

4.2.4 Contrôle des certificats

Dans le cas d'une organisation hiérarchique d'autorités de certification, un certificat peut être délivré à une personne par une autorité que le serveur ne connaît pas. Cette autorité est-elle même authentifiée par un certificat délivré par une autorité de plus haut niveau, et ainsi de suite. Nous avons donc une chaîne de certificats qui suit une branche de la hiérarchie des autorités de certification.

Dans ce cas, vérifier la validité d'un certificat consiste à parcourir cette chaîne de certificats jusqu'au certificat d'une autorité reconnue par le serveur. Le processus est le suivant :

1. Le serveur vérifie la période de validité du certificat qu'il reçoit.
2. La signature du certificat est contrôlée grâce à la clé publique contenue dans le certificat.
3. Si l'autorité ayant émis le certificat est reconnue par le serveur, la vérification se termine avec succès. Sinon, le certificat de l'autorité de certification est localisé dans la chaîne des certificats et le processus de contrôle reprend à l'étape 1 avec ce nouveau certificat.

Dans le cas d'une organisation de type « Web of Trust », nous ne pouvons pas constituer de telles chaînes de certificats. La méthode employée pour contrôler la validité d'un certificat consiste à rechercher parmi les signatures du certificat un nombre suffisant de signatures auxquelles on peut faire confiance. Lorsque l'on accorde sa confiance à une personne en tant que signataire de certificats, on pondère cette confiance par un terme pouvant prendre cinq valeurs :

- aucune confiance
- confiance faible
- confiance moyenne
- grande confiance
- confiance absolue

L'algorithme de contrôle parcourt le réseau de signatures d'un certificat jusqu'à trouver au moins trois signatures de confiance moyenne ou une signature de grande confiance.

4.2.5 Avantages des certificats.

L'utilisation des certificats apporte une solution au problèmes d'authentification. Si un certificat est délivré par une autorité fiable ou par une autorité subordonnée, on peut être assuré qu'il correspond bien à la personne supposée le détenir. Ce système fonctionne dans le sens client vers serveur et également dans le sens inverse, permettant au client de s'assurer qu'il communique bien avec le serveur souhaité.

D'autre part, l'utilisation conjointe de certificats et de signatures digitales permet de s'assurer de l'intégrité des données signées et empêche toute contestation ultérieure concernant ces données.

Cependant, l'utilisation de la cryptographie par clé publique (signature digitale) est consommatrice en temps CPU et en trafic réseau. Elle ne peut être appliquée que dans le cas d'un faible volume de données échangées et se traduit par un délai de traitement notable. Dans la pratique, les protocoles comme SSL basés sur les certificats autorisent les deux parties, à la fin de la session d'authentification, à négocier une méthode de cryptage à base de clés symétriques. Les interlocuteurs peuvent alors communiquer de manière cryptée en utilisant des algorithmes bien plus rapides que la signature digitale.

5. Solutions préconisées pour Glasnost

La sécurisation du logiciel Glasnost concerne à la fois l'authentification des systèmes Glasnost souhaitant collaborer et l'authentification des utilisateurs se connectant à un système Glasnost.

L'authentification des serveurs est beaucoup plus importante et sensible que l'authentification des personnes. Par conséquent, la solution proposée pour l'authentification des systèmes devra être plus robuste. Quant à l'authentification des utilisateurs, une solution plus « légère » pourra être adoptée.

5.1 Authentification des utilisateurs

Le choix d'une solution pour l'authentification des utilisateurs est assez délicate. En effet, l'utilisation de certificats offre un meilleur niveau de sécurité et d'identification. Cependant, dans certains cas, une telle solution peut s'avérer trop lourde. Par exemple, un grand nombre d'utilisateurs peuvent être maenés à se connecter à un système Glasnost loors de l'organisation d'une consultation publique. Dans ce cas, la gestion des certificats peut poser des problèmes de ressources, tant en utilisation de la CPU qu'en charge réseau.

Par ailleurs, la gestion des certificats n'est correctement mise en œuvre que dans les navigateurs Internet récents et ne peut donc fonctionner pour des utilisateurs disposant de versions plus anciennes.

Pour toutes ces raisons, nous préconisons l'utilisation d'une méthode basée sur des paires identifiant/mot de passe.

En dehors de l'authentification, cette solution devra permettre :

1) D'éviter l'envoi du mot de passe en clair sur le réseau ;
2) De retourner par courrier électronique le mot de passe d'un utilisateur qui l'aurait oublié ;
3) De transmettre le mot de passe à des applications tierces (ceci permettrait, par exemple, de consulter un serveur de courrier électronique *via* l'interface Glasnost).

La méthode proposée est la suivante :

1) Le mot de passe utilisateur est crypté sur le serveur au moyen d'une clé privée du système Glasnost.
2) Lors de la connexion, l'utilisateur envoie son identifiant et une empreinte de son mot de passe (empreinte calculée par hachage, sur le poste client au moyen d'un scrypt JavaScript).
3) Le serveur contrôle la présence de l'identifiant dans sa base de données. Si l'identifiant est trouvé, il décrypte le mot de passe stocké, calcule sa propre empreinte et la compare à celle envoyée. (remarque : on pourrait également réduire le temps de calcul en supprimant l'étape de décryptage ; il suffirait pour cela de stocker dans la base de données le mot de passe crypté et son empreinte non cryptée).

Lors de la création du compte utilisateur, celui-ci doit répondre à une question non triviale. S'il perd ensuite son momt de passe, il pourra répondre de nouveau à la même question et le mot de passe lui sera renvoyé par courrier électronique.

Si le système Glasnost doit fournir le mot de passe de l'utilisateur à une autre application, il pourra le récupérer dans sa base de données, le décrypter et l'envoyer à cette application.

Remarque : cette solution d'authentification par mot de passe pourra, en cas de nécessité, être facilement remplacée par une authentification par certificats grâce à l'utilisation de la bibliothèque GnuTLS dont nous parlerons plus loin.

5.2 Authentification des systèmes Glasnost

L'authentification des systèmes Glasnost entre eux est le point crucial de la sécurisation. Pour cette raison, une simple authentification par mot de passe n'est pas suffisante. C'est donc vers une solution à base de certificats que nous devons nous orienter.

L'authentification par certificat est généralement reconnue comme étant plus robuste et fiable. Elle apporte une preuve de l'identité du client qui souhaite se connecter. En effet, l'authentification par certificat est basée à la fois sur une chose que le client *possède* (la clé privée qui correspond à la clé publique contenue dans le certificat) et sur une chose que le client *sait* (le mot de passe permettant de débloquer sa base de données de clés privées).

L'authentification par certificat peut fonctionner dans les deux sens, permettant, par exemple, à deux système Glasnost de s'authentifier mutuellement afin d'être certain que l'autre partie est bien celle que l'on suppose.

L'utilisation des certificats conjointement avec la signature digitale permet de contrôler l'intégrité des données transmises.

Si besoin, la phase d'authentification peut se conclure sur la négociation d'une méthode de cryptage par clés symétriques pour la suite des échanges entre le client et le serveur.

5.3 Choix d'une architecture de PKI

Nous disposons de deux types d'architecture : une architecture classique, hiérarchique, utilisant les certificats de type X.509 et une architecture appelée « Web of Trust » utilisant des certificats de type OpenPGP.

La seconde architecture est attrayant dans la mesure où elle utilise un réseau de confiance entre personnes, proche de la vie réelle et sans hiérarchie. Mais le système de pondération de la confiance dans les signatures n'est pas adapté au cas de Glasnost ou l'on souhaite une solution plus simple de tout ou rien : soit le certificat est accepté, soit il est refusé.

Nous préconisons donc l'adoption d'une architecture à base d'autorités de certification. Chaque système Glasnost ne sera pas nécessairement une autorité de certification, car cela implique un serveur dédié et la mis en place d'une solution logicielle plus lourde. Un système Glasnost pourra établir une liste d'autres systèmes Glasnost dont il accepte les certificats, voire d'autres autorités de certification existantes. Les système Glasnost mettant en place un service de certification pourront également se certifier entre eux.

5.4 Solution logicielle

L'authentification par certificats met en oeuvre une technologie assez complexe. Heureusement, il existe une solution libre facilitant l'implémentation d'une telle solution. La bibliothèque GnuTLS offre un ensemble de fonctions permettant de mener à bien les sessions d'authentification et de contrôler la validité des certificats.

GnuTLS support les protocoles sécurisés SSL 3.0 et TLS 1.0 (successeur de SSL). Elle gère les certificats de type X.509 et OpenPGP.

6. Présentation de GnuTLS

GnuTLS est une bibliothèque offrant une API pour accéder à des protocoles de communication sécurisés. Elle est écrite en code portable ANSI C et implémente les protocoles SSL 3.0 et TLS 1.0. La bibliothèque est distribuée sous licence GNU Lesser GPL. GnuTLS est constituée de trois parties indépendantes :

- la partie « protocole TLS », met en oeuvre le protocole TLS.
- la partie « certificats » est constituée de fonctions pour l'analyse et le contrôle des certificats.
- la partie « cryptographie » (bibliothèque libcrypt) offre le support pour diverses méthodes de cryptage des données.

6.1 Introduction à TLS

TLS (Transport Layer Security) est le successeur de SSL (Secure Socket Layer). Il s'agit d'un protocole Internet défini par l'IETF et offrant des services de confidentialité et d'authentification au-dessus de la couche de toute couche de transport fiable.

6.1.1 Les couches TLS

TLS est un protocole en couches, constitué d'un *Record Protocol*, d'un *Handshake Protocol* et d'un *Alert Protocol*.

Le *Record Protocol* s'occupe du cryptage symétrique, de l'authentification des données et, optionnellement, de leur compression. La confidentialité des données est assurée au moyen de cryptage symétrique par blocs (`3DES`, `AES`, etc.) ou par flux (par exemple `ARCFOUR`). Dans le cas de cryptage par bloc, un nombre aléatoire de blocs non significatifs est ajouté aux données pour résister à l'analyse statistique des échanges. Les algorithmes de cryptage supportés par le Record Protocol sont les suivants :

- `3DES_CBC` : algorithme de cryptage de blocs utilisant un triple cryptage.
- `ARCFOUR_128` : algorithme de cryptage très rapide utilisant une clé de 128 bits.
- `ARCFOUR_40` : algorithme de cryptage `ARC4` utilisant une clé de 40 bits, ce qui est considéré comme une faible protection.
- `AES_CBC` : algorithme de cryptage remplaçant l'ancien algorithme `DES`.
- `TWOFISH_CBC` : algorithme de cryptage sur des blocs de 128 octets. N'est pas officiellement supporté par TLS mais implémenté par GnuTLS.

Pour le hachage, GnuTLS supporte les algorithmes suivants :

- `MAC_MD5` : algorithme `MD5` retournant un résultat sur 128 bits.
- `MAC_SHA` : algorithme `SHA` retournant un résultat sur 160 bits.

En ce qui concerne la compression des données, les algorithmes supportés sont :

- `ZLIB` : compression `ZLIB` utilisant l'algorithme `DEFLATE`.
- `LZO` : algorithme de compression très performant fourni par la bibliothèque GnuTLS-extra.

L'*Alert Protocol* permet de propager des messages d'alerte aux autres protocoles. Il permet d'informer les interlocuteurs sur les causes d'une échec ou d'autres conditions d'erreur.

Le *Handshake Protocol* est responsable de la négociation des paramètres de sécurité, de l'échange initial de clés et de l'authentification. Ce protocole est contrôlé par l'application.

6.1.2 Authentification des certificats X.509

Comme indiqué plus haut, les certificats X.509 contiennent des paramètres publics, un algorithme de clé publique et la signature de l'autorité de certification. Les méthodes d'échange de clés suivantes sont disponibles pour l'authentification X.509 :

- `RSA` : algorithme utilisé pour crypter une clé et l'envoyer au correspondant. Le certificat doit permettre d'utiliser cette clé pour le décryptage.
- `RSA_EXPORT` : l'algorithme `RSA` est utilisé pour crypter la clé et l'envoyer au correspondant. Dans l'algorithme `EXPORT`, le serveur signe temporairement des paramètres RSA avant des les envoyer au client.
- `DHE_RSA` : algorithme `RSA` utilisé pour signer des paramètres éphémères envoyés au correspondant. Le certificat doit autoriser l'utilisation de la clé pour la signature. Cet algorithme offre un excellent niveau de confidentialité.
- `DHE_DSS` : l'algorithme `DSS` est utilisé pour signer des paramètre éphémères qui sont envoyés au correspondant.

6.1.3 Contrôle du cheminement d'un certificat

Le contrôle du cheminement d'un certificat permet de savoir si le certificat est signé par une autorité reconnue. GnuTLS offre une fonction dont le résultat est un OU logique entre l'état des certificats d'une chaîne de certificats. Si le résultat retourné est VRAI, cela signifie que la chaîne de certificats aboutit à une autorité de certification reconnue.

Il faut remarquer que l'authentification du certificat du client est optionnelle en TLS. Un serveur peut demander un certificat au client. Dans ce cas le serveur envoie un paquet supplémentaire contenant une liste de signataires reconnus. Le client peut alors envoyer un certificat signé par l'une des autorités acceptées.

Annexes

A1. Création d'une API Python pour GnuTLS

Le logiciel Glasnost est écrit en langage Python et il n'existe pas actuellement d'API Python pour GnuTLS. Nous devrons donc créer cette API, ce qui est un processus relativement simple en Python.

Python dispose d'un ensemble de fonctions, de macros et de variables qui permettent l'accès à la majeure partie du système d'exécution de l'interpréteur Python. Voici, à titre d'exemple, un squelette définissant un module « gnutls » fictif écrit en C et fournissant deux fonctions `hello_world` et `fct_vide` :

```
#include <Python.h>
#include <gnutls.h>

// liste des fonctions du module
static PyMethodDef GnutlsMethods[] = {
     ...
     {"hello_world", gnutls_hello_world, METH_VARARGS,
      "fonction hello world."},
     {"function_vide", gnutls_fct_vide, METH_VARARGS,
      "fonction qui ne retourne rien."},
     ...
     {NULL, NULL, 0, NULL}
};

// fonction qui retourne la chaine 'Hello world!!!'
static PyObject * gnutls_hello_world(PyObject *self, PyObject *args) {
     // La fonction doit retourner un objet Python
     return Py_BuildValue("s", "Hello world!!!");
}

// fonction qui ne retourne rien
static PyObject * gnutls_fct_vide(PyObject *self, PyObject *args) {
     PY_INCREF(Py_None);
     return Py_None;
}

// Fonction appelée au chargement du module
PyMODINIT_FUNC initgnutls(void) {
     PyObject *m ;

     // création du module et ajout des fonctions
     m = Py_InitModule("gnutls", GnutlsMethods);
}
```

Évidemment, d'est exemple n'est pas représentatif de tout ce qui est nécessaire pour définit un module. Suivant la complexité d'un module, on peut être amené à définir de nouveaux objets et de nouvelles exceptions propres à ce module. Pour les nouveaux objets, il est conseillé de fournir des fonctions de représentation, de désallocation, de manipulation et de comparaison entre objets.

Une fois compilé, l'exemple ci-dessus est appelé de la manière suivante dans un programme Python :

```
import gnutls
s = gnutls.hello_world()
print s
```

A2. Exemple d'un client supportant les certificats X.509

Cet exemple montre comment la bibliothèque GnuTLS permet de mettre en place de manière simple l'authentification par certificat du côté client. L'exemple suivant est un client TLS très simple aux fonctionnalités limitées, écrit en C :

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <gnutls/gnutls.h>

/* Un client TLS élémentaire.
 */

#define MAX_BUF 1024
#define CRLFILE "crl.pem"
#define CAFILE "ca.pem"
#define SA struct sockaddr
#define MSG "GET / HTTP/1.0\r\n\r\n"

int main()
{
   const char *PORT = "443";
   const char *SERVER = "127.0.0.1";
   int err, ret;
   int sd, ii;
   struct sockaddr_in sa;
   gnutls_session session;
   char buffer[MAX_BUF + 1];
   gnutls_certificate_credentials xcred;

   /* Autorise aussi la connexion aux serveurs disposant de clés OpenPGP.
    */
   const int cert_type_priority[2] = { GNUTLS_CRT_X509,
      GNUTLS_CRT_OPENPGP, 0 };

   gnutls_global_init();

   /* X509 */
   gnutls_certificate_allocate_credentials(&xcred);

   /* Fichier des autorités de certification */
   gnutls_certificate_set_x509_trust_file(xcred, CAFILE,
GNUTLS_X509_FMT_PEM);

   /* Connexion au serveur */
```

```
   sd = socket(AF_INET, SOCK_STREAM, 0);

   memset(&sa, '\0', sizeof(sa));
   sa.sin_family = AF_INET;
   sa.sin_port = htons(atoi(PORT));
   inet_pton(AF_INET, SERVER, &sa.sin_addr);

   err = connect(sd, (SA *) & sa, sizeof(sa));
   if (err < 0) {
      fprintf(stderr, "Erreur de connexion\n");
      exit(1);
   }
   /* Initialisation de la session TLS */
   gnutls_init(&session, GNUTLS_CLIENT);

   /* Utilistion des priorités par défaut */
   gnutls_set_default_priority(session);
   gnutls_certificate_type_set_priority(session, cert_type_priority);

   /* Place les pièces X.509 dans la session courante */
   gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE, xcred);

   gnutls_transport_set_ptr( session, sd);

   /* Exécute le handshake TLS */
   ret = gnutls_handshake( session);

   if (ret < 0) {
      fprintf(stderr, "*** Le handshake a échoué\n");
      gnutls_perror(ret);
      goto end;
   } else {
      printf("- Handshake réussi\n");
   }
   gnutls_record_send( session, MSG, strlen(MSG));

   ret = gnutls_record_recv( session, buffer, MAX_BUF);
   if (ret == 0) {
      printf("- La connexion TLS a été fermée\n");
      goto end;
   } else if (ret < 0) {
      fprintf(stderr, "*** Erreur: %s\n", gnutls_strerror(ret));
      goto end;
   } else if (ret > 0) {
      printf("- Received %d bytes: ", ret);
      for (ii = 0; ii < ret; ii++) {
         fputc(buffer[ii], stdout);
      }
      fputs("\n", stdout);
   }
   gnutls_bye( session, GNUTLS_SHUT_RDWR);

 end:
   shutdown(sd, SHUT_RDWR);
   close(sd);

   gnutls_deinit(session);
   gnutls_certificate_free_credentials(xcred);
   gnutls_global_deinit();
   return 0;
}
```

Bien que très simple, cet exemple montre clairement la facilité de mise en place d'une gestion de certificats en utilisant la bibliothèque GnuTLS.

A3. Exemple de serveur supportant les certificats X.509

Voici maintenant un exemple de serveur très simple supportant l'authentification par certificats X.509, basé sur la bibliothèque GntTLS :

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <unistd.h>
#include <gnutls/gnutls.h>

#define KEYFILE "key.pem"
#define CERTFILE "cert.pem"
#define CAFILE "ca.pem"
#define CRLFILE "crl.pem"

/* Exemple de serveur TLS.
 */


#define SA struct sockaddr
#define SOCKET_ERR(err,s) if(err==-1) {perror(s);return(1);}
#define MAX_BUF 1024
#define PORT 5556               /* ecoute du port 5556 */
#define DH_BITS 1024

gnutls_certificate_credentials x509_cred;

gnutls_session initialize_tls_session()
{
   gnutls_session session;

   gnutls_init(&session, GNUTLS_SERVER);
   gnutls_set_default_priority( session);

   gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE, x509_cred);

   /* demande du certificat client
    */
   gnutls_certificate_server_set_request( session, GNUTLS_CERT_REQUEST);

   gnutls_dh_set_prime_bits( session, DH_BITS);

   return session;
}

gnutls_dh_params dh_params;
```

```
static int generate_dh_params(void) {
   gnutls_dh_params_init( &dh_params);
   gnutls_dh_params_generate2( dh_params, DH_BITS);

   return 0;
}

int main()
{
   int err, listen_sd, i;
   int sd, ret;
   struct sockaddr_in sa_serv;
   struct sockaddr_in sa_cli;
   int client_len;
   char topbuf[512];
   gnutls_session session;
   char buffer[MAX_BUF + 1];
   int optval = 1;
   char name[256];

   strcpy(name, "Echo Server");

   gnutls_global_init();

   gnutls_certificate_allocate_credentials(&x509_cred);
   gnutls_certificate_set_x509_trust_file(x509_cred, CAFILE,
      GNUTLS_X509_FMT_PEM);

   gnutls_certificate_set_x509_crl_file(x509_cred, CRLFILE,
      GNUTLS_X509_FMT_PEM);

   gnutls_certificate_set_x509_key_file(x509_cred, CERTFILE, KEYFILE,
      GNUTLS_X509_FMT_PEM);

   generate_dh_params();

   gnutls_certificate_set_dh_params( x509_cred, dh_params);

   /* Socket operations
    */
   listen_sd = socket(AF_INET, SOCK_STREAM, 0);
   SOCKET_ERR(listen_sd, "socket");

   memset(&sa_serv, '\0', sizeof(sa_serv));
   sa_serv.sin_family = AF_INET;
   sa_serv.sin_addr.s_addr = INADDR_ANY;
   sa_serv.sin_port = htons(PORT);  /* Numero de port du serveur Server */

   setsockopt(listen_sd, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(int));

   err = bind(listen_sd, (SA *) & sa_serv, sizeof(sa_serv));
   SOCKET_ERR(err, "bind");
   err = listen(listen_sd, 1024);
   SOCKET_ERR(err, "listen");

   printf("%s Prêt. Ecoute du port '%d'.\n\n", name, PORT);

   client_len = sizeof(sa_cli);
   for (;;) {
      session = initialize_tls_session();
```

```
      sd = accept(listen_sd, (SA *) & sa_cli, &client_len);

      printf("- connection de %s, port %d\n",
             inet_ntop(AF_INET, &sa_cli.sin_addr, topbuf,
                       sizeof(topbuf)), ntohs(sa_cli.sin_port));

      gnutls_transport_set_ptr( session, sd);
      ret = gnutls_handshake( session);
      if (ret < 0) {
         close(sd);
         gnutls_deinit(session);
         fprintf(stderr, "*** Echec du handshake (%s)\n\n",
                 gnutls_strerror(ret));
         continue;
      }
      printf("- Handshake réussi\n");

      i = 0;
      for (;;) {
         bzero(buffer, MAX_BUF + 1);
         ret = gnutls_record_recv( session, buffer, MAX_BUF);

         if (ret == 0) {
            printf
                ("\n- Connexion GNUTLS interrompue \n");
            break;
         } else if (ret < 0) {
            fprintf(stderr,
                    "\n*** Données corrompues(%d). Fermeture de la
connexion.\n\n",
                    ret);
            break;
         } else if (ret > 0) {
            /* Retourne les données au client
             */
            gnutls_record_send( session, buffer,
                        strlen(buffer));
         }
      }
      printf("\n");
      gnutls_bye( session, GNUTLS_SHUT_WR);

      close(sd);
      gnutls_deinit(session);

   }
   close(listen_sd);
   gnutls_certificate_free_credentials(x509_cred);
   gnutls_global_deinit();
   return 0;
}
```

Références

The Open-source PKI Book, Symeon Xenitellis, OpenCA Team
http://ospkibook.sourceforge.net/docs/OSPKI-2.4.7/OSPKI-html/ospki-book.htm

Transport Layer Security (TLS), Treese, Housley, Bellovin & Mankin
http://www.ietf.org/html.charters/tls-charter.html

The GNU Transport Layer Security Library (GNUTLS), Free Software Foundation
http://www.gnu.org/software/gnutls/gnutls.html

The GNU Privacy Guard (GnuPG), Free Sotware foundation
http://www.gnupg.org/